Massiv Core Documentation

Programmer's Guide

Stepan Vondrak, MFF UK `<stoupik at users dot sourceforge dot net>`
Marek Vondrak, MFF UK `<markoid at users dot sourceforge dot net>`
Petr Tovarys, MFF UK `<boovie at users dot sourceforge dot net>`
Ondrej Pecta, MFF UK `<octa at users dot sourceforge dot net>`
Martin Havlista, MFF UK `<hafik at users dot sourceforge dot net>`
Marek Svantner, MFF UK `<marekus at users dot sourceforge dot net>`

Massiv Core Documentation: Programmer's Guide

by Stepan Vondrak, Marek Vondrak, Petr Tovarys, Ondrej Pecta, Martin Havlista, and Marek Svantner

Table of Contents

List of Figures

List of Tables

List of Examples

Part I. Overview

This part guides you through the basic Massiv ideas quickly. It explains what the Massiv is about, why we decided to develop it in the way we did and what primary spheres of functionality it offers to its user. Finally you'll get acquainted with the most fundamental ideas of the Massiv distributed object model.

Table of Contents

1. Introduction

Nowadays, with the rapid development of the internet, there always emerge new world-wide network applications. One of them, having its origins just a few years ago, are the *massively multiplayer online games* (MMO games). The common feature of these games is that they consist of an imaginative world that is simulated by one or more servers and of many (up to thousands) players who are connected to the simulation simultaneously controlling their characters and interacting with the simulated world in the real time.

The simulation mentioned typically runs 24 hours a day 7 days a week and players can take part from any corner of the world. After a player disconnects, his character cannot interact with the world any more, but it keeps its qualities and possessions until the player connects again.

As obvious, development of such a distributed system (especially when the simulation is spread upon more servers) is an extremely time-consuming task requiring a good financial background and it seems that only commercial companies could afford to undertake such a difficult development. And here comes the Massiv to propose a solution to this situation.

The Massiv creates a platform for easier development of the MMO games. It provides functionality to hide the distribution from the programmer to some extent. It also provides an object model for an efficient simulation. However, the programmer still has to respect that the system is distributed, but still his situation is much easier, because much work has been already done for him.

There is a significant difference between the Massiv and other existing MMO games. Whereas existing games usually simulate the world on several servers located in the same *cluster* (in the same broadcast domain), the Massiv is designed to run on many servers located anywhere in the world. This makes the development more difficult, because it brings in new issues such as longer delays, security or locating object efficiently. The main advantage of this architecture is that it enables more independent subjects to set up a simulation together without having to be physically in the same location. Thus, the Massiv should be more open to the non-commercial community.

Note that along with the Massiv Core library (the one this book is all about) this distribution contains also a demonstration application (we call it simply the Demo). When discussing non-trivial features of the Core, this book often references the Demo as a real-world example.

Although we talk about a *world simulation*, a *player*, etc. throughout this documentation, please note that the Massiv can have a much more general usage. It is not limited only to games, but you can use it for anything that conforms to the object model that will be described below.

The goal of this book is to get you acquainted with the Massiv programming as quickly as possible. However, note that you won't be familiar with every detail and trifle of the Core right after you finish reading. You also have to use the Massiv Core Reference Guide generated from the source codes and to make a few experiments to get more experienced before you can start developing a 'real' application using the Massiv.

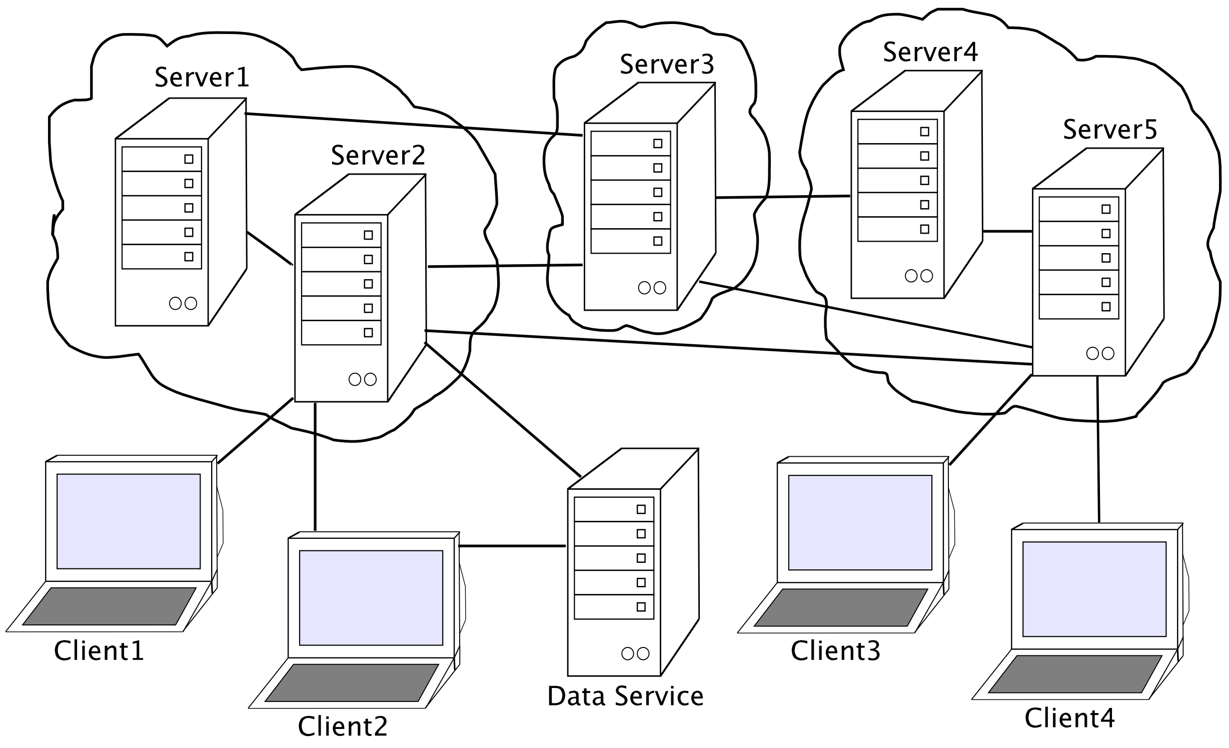
2. Architecture

2.1. Types Of Nodes

The Massiv Core library basicaly counts with three types of *nodes* (note that more Massiv nodes may run on the same physical node - computer):

- *Servers* collaborate on the world simulation.
- *Clients* enable players to connect to the simulation, interact with the imaginative world and provide them a presentation of the actual state of the world. Note that while the Demo distinguishes privileged and unprivileged clients, this distinction is implemented at “application” level, i.e. it's feature of the Demo, not of the Massiv Core.
- *Data service* nodes provide dynamic download of binary and textual data that can't be efficiently represented by an object.

Figure 2.1. Massiv deployment diagram - example



The picture shows one of many possible deployment diagrams for a Massiv application involving five servers, four clients and one data service. You can observe several issues from the diagram:

- The servers are located in potentially different broadcast domains (or clusters). The domains are graphically represented by bubbles.

- Not each server has an explicit connection to every other server. Virtually they are connected with each other, but physically not all of these connections are maintained all the time and are reestablished *on demand*.
- Each client that is connected to the simulation may be connected to at most one server (plus the data service).

Note

Each Massiv node doesn't of course necessarily have to run on a different computer. But even for such a local application, the individual processes will communicate via TCP/IP with each other.

The advantages of the fact that the servers don't have to belong to the same cluster (broadcast domain) were already mentioned in the introduction.

2.2. Consistency and Failure Resistance

The concept with many servers placed in geographically different locations brings some serious problems. To keep the simulation running, all participating servers must be up. If any server broke down, the others shouldn't be able to continue, because the world would immediately become inconsistent and this inconsistency could not be fixed another way than by restarting all the servers from the last consistent backup.

Note that no assumptions about the data service have been made; its semantics and usage is defined by the application using the Core and thus we cannot tell (from the Core's point of view) to what extent a failure of the data service matters or not.

2.3. Basic Properties Of The Core

This section briefly lists the basic features of the Core. See Section 2.4, “Core Internal Architecture Survey” for general overview about how these features are achieved. You will also find links to the relevant chapters there.

- *Object Orientation*

  The Massiv provides a fully object-oriented interface to its user. It provides its own sophisticated object model (see Chapter 3, *Object Model Introduction* for more general information) that provides fundamentals for implementation of most of the following features.

- *Distribution*

  Because of a large number of players and a potential complexity of the imaginative world it is typically needed to spread the simulation over more servers. To cope with this, the Massiv comes with an object *migration* (enables object to move to another node, either transparently or on demand). The migration is one of the main concepts of the Massiv - even sending messages is implemented as a migration of message objects.

The main drawback of the simulation distributed over arbitrary locations is the unavoidable performance loss. It is obvious that we could never construct even nearly as efficient system as anyone who uses a computer cluster. Another disadvantage is an increased number of persisting bugs, because the complexity of debugging of a system such as Massiv is significant. There also are greater demands on system administrators - they must be able to behave as an efficient team, which might be quite difficult because of the distribution all over the world.

- *Security*

  The necessity of secure connection between client and server is obvious; it is needed to prevent players from cheating and other parties from doing harm. But together with the concept of servers located apart from each other there arises also the requirement of secure connections between servers and between a server and the data service. Classical MMO games didn't have to implement the security on this level. This security even more prolonges the communication delays, but cannot be avoided.

- *Interactivity*

  Players must be able to interact with the world in the real time. Also the simuation and presentation must be real-time.

- *Persistence*

  Because the world simulation runs generally continuously, the system must be able to ensure a world persistence to some extent. In the Massiv, this means that it must be able to generate a game backup (archive) at any moment and transparently (without giving players a chance to notice). The persistence in the Massiv ensures that the game always can be returned to the last consistent state. The frequency of archivations can be determined by administrators of the system. See Chapter 23, *Archivation and Startup* for more detail info about the archivation.

- *Consistency*

  The Core itself implements complex protocols to ensure various aspects consistent. The Core user doesn't have to cope with these problems again and thus his situation is much easier. Some of the consistencies are:

  - *Consistent Archive*

    Although the archivation process runs simultaneously on several servers (and each server generates its own part of the archive), it must be ensured that all objects stored in all archives correspond to their state at the same simulation time.

  - *Consistent Pointers*

    Users of the Core can work with special pointers that can refer to objects on other nodes. However, because objects can migrate freely, it requires additional mechanism to ensure consistency of the reference (pointer) and the referee (object).

  - *Consistent Replicas*

    Replication is a concept of duplication of some object on other servers to reduce the network communication and thus the latency. However, it must be seen to it that these replicas are in a consistent state with originals.

- *Time Synchronization*

  Time synchronization is needed for scheduling events. Obviously all parts of the simulated world must share the same time.

- *Load Balancing*

  Load balancing makes use of the migration and manages a distribution of objects over servers so that there is no server that is much more encumbered than the others.

- *Data Download*

  Data service nodes provide a dynamic download of data that are needed by client and server nodes. The data can be both textual and binary. The data service can be used to distribute configuration files to all nodes or to provide data required by the client for the presentation of the world, such as 3D models and textures.

- *Presentation*

  Presentation itself is not implemented by the Core. However, the Core provides some mechanisms that can be used to implement it. For example, the replication can be used to replicate the presentation-related object to clients . The programmer using the Massiv Core should be careful about separating the presentation data from the data that is used between servers during the simulation. This will help to prevent players from cheating and/or causing some inconsistencies in the simulated world.

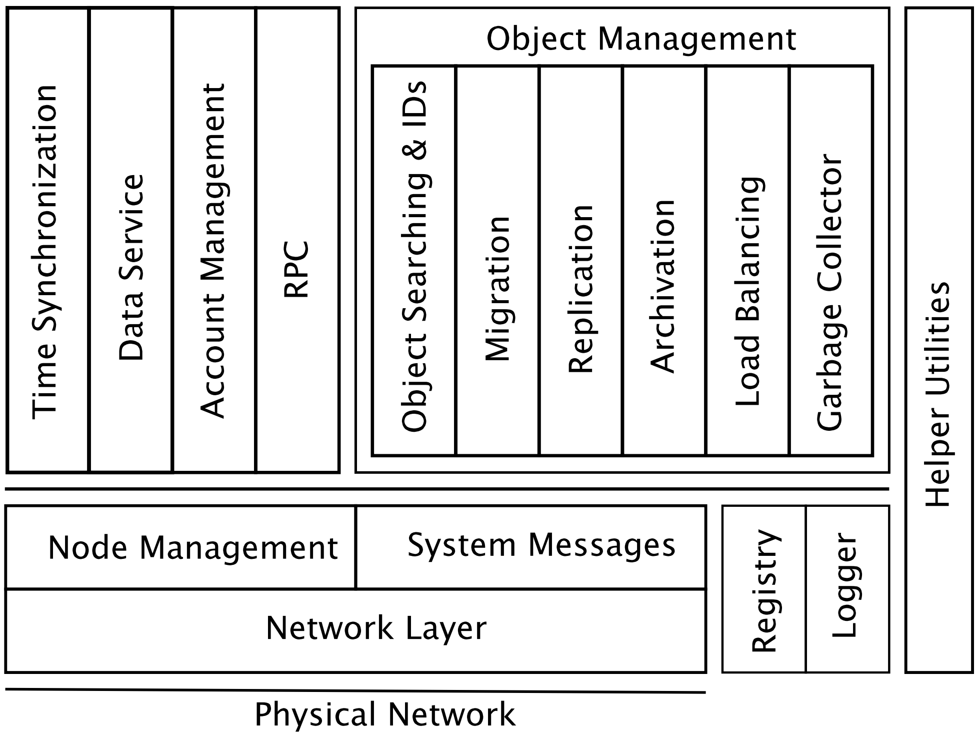
- *Auxiliary utilities*

  The Massiv also provides number of external utilities that can be useful to system administrators. For example, there is a generator of configuration files, generator of RSA keys, tools for work with volumes/archives and others. See Chapter 25, *Auxiliary Utilities* for more information about external tools.

2.4. Core Internal Architecture Survey

The following picture tries to give you an idea about the Massiv internal structure. It doesn't even hope to present a complete or exact information; a graph that would represent it would be very difficult and it won't be shown in this book (the Core user doesn't need to know it). The picture shows only the most significant Massiv modules and to what layers they *approximately* (the Massiv doesn't have a strict layered architecture) belong.

Figure 2.2. Survey of the internal architecture of the Core



The following modules are involved in the previous image:

- *Network Layer*

  The special network layer enables to communicate over the network using an interface similar to the standard iostreams. It is a more simple way than using standard sockets libraries. See also Appendix A, *Network Layer*.

- *System Messages*

  System messages are simple serializable objects that can be sent between nodes. Most subsystems of the Core use them to communicate over network instead of lower-level networking interface.

- *Node Management*

  Node manager stores information about all known nodes and status of nodes connected to the simulation. System message delivery algorithms cooperate with the node manager when delivering the messages.

- *Registry*

  The registry is a persistent database that is used for the Massiv configuration as well as for storing statistical values, etc. See also Chapter 19, *Registry*.

- *Logger*

  The logger enables to log messages, filtering them according to several criteria and sending them to potentially more destinations. It was one of the most significant tools for debugging the Massiv. See also Chapter 20, *Logger Library*.

- *Object Searching*

  Because objects in the Massiv can transparently move to other network nodes, a facility that would be able to localize a given object in the simulation is necessary. It is also needed to have a mechanism able to give objects new identificators so that there are no collisions in the whole system. The *Object Searching and IDs* module accomplishes this task. See also Section 4.4, “Object Identification”.

- *Migration*

  The migration means moving Massiv objects to other nodes, either transparently or on demand. For more information see Chapter 6, *Migration*.

- *Replication*

  The replication enables to create a Massiv object replicas on other nodes, which improves the network throughput. Some operations can be performed on replicas instead of accessing the original object over the network. See also Chapter 7, *Replication*.

- *Archivation*

  The archivation is responsible for storing the actual state of simulation. The task is non-trivial, because all servers participating in the simulation must store the state *transparently* and *relatively to the same time*. See Chapter 23, *Archivation and Startup*.

- *Load Balancing*

  The load balancing module tries to avoid the situation when some servers are overloaded, whereas others are idle. It should transparently migrate some objects from the idle servers. See also Appendix B, *Load Balancing*.

- *Garbage Collector*

  The garbage collector destroys all Massiv objects that are not needed any more. We do not specify now what this exactly means; refer to Section 5.4, “Garbage Collector”.

- *RPC*

  RPC stands for *remote procedure call*. It enables to call methods of objects that are located on other network nodes in a simple way. See also Chapter 8, *Remote Procedure Call*.

- *Data Service*

  The data service manages all simulation data (models, textures, etc.) in a tree hierarchy, where, for example, a specific texture can be replaced by its ascendant temporarily. This enables even players with a modem connection to download all data dynamically while they are already playing. See Chapter 24, *Data Service*.

- *Account Management*

  It manages creation of client accounts.

- *Time Synchronization*

  The time synchronization is responsible that the simulation time on all participating servers does not differ too much. For more information, see Chapter 18, *Simulation Time*.

- *Helper Utilities*

  A set of auxiliary utilities useful for the Massiv simulation administrators. For more information see Chapter 25, *Auxiliary Utilities*.

3. Object Model Introduction

The purpose of this chapter is to give the reader an overview of the Massiv object model and to make him familiar with the basic principles that will be described in the second part of the book in detail.

The Massiv defines and implements its own distributed object model built on top of the C++ object model. The model can be separated into two independent parts:

- Object model design - what can be done with the objects and how.
- Mapping to the C++ object model and discussion of implementation issues.

3.1. Object Model Design Goals

At the early development stages the team had to carefully clarify what the basic model design goals were, what could be done with objects, how and to what extent the object manipulation would be supported (automated) by the Massiv Core. The generic C++ object model had to be extended to allow implementation of new features that were not supported by the standard but were required to satisfy our distributed object model goals. The usage of the new model should have been as close to the standard model as possible. We were certainly inspired by some existing challenging middlewares but also found many of their principles useless and impractical for purposes of MMO games and so developed our own techniques which we believe to be really novel.

The following design goals were taken into consideration:

- *Unique object identification*

  Objects must be uniquely identifiable (addressable) through out the distributed object space. The identification of an object never changes and does not depend on the object's current location. Object identifications are not recycleable, they are persistent and can be passed to other nodes.

- *Location transparency*

  Objects can migrate. Effect of an operation should not depend on the relevant object's current location. The Core must be able to localize an object even if it migrates. The Core must be able to migrate objects not only in consequence of user requests. The actual migration operation must be transparent to the user and the Core must not require the user to participate in the migration process (cooperate with the Core).

- *Automatic persistency*

  The Core must be able to automatically serialize objects and perform consistent backups. Objects can point to other objects and such pointers are persistent too.

- *Automatic replication*

Any object can be replicated to other nodes. The Core must be able to gain access to either original object (primary replica) or its replica (secondary replica).

- *Flexible object migration and replication granularity*

  Objects can be optionally grouped into migration and replication groups. The Core ensures that the groups will always be processed as an aggregate (objects from the same migration group will always be located on the same node). Groups can be changed at run-time and the way of how they are defined must respect the dynamic nature of the changes. The definition should be automatic, possibly driven by object pointers.

- *Object to object migration as a primary way of object collaboration*

  Object migration is the only directly supported way of data exchange between objects and object interaction. Migrations are addressed by objects. This is basically an agent model, where agents (objects) migrate in order to accomplish their tasks. Once delivered to an object, a callback is called on the delivered object. Almost everything else (RPC and messaging, for example) can be built on top of the object migration.

- *Automatic object management*

  Objects can be dynamically created and automatically accessed from all nodes. This includes even short time living objects. There is no need to discriminate between local "unmanaged" objects and the objects managed by the Core. A form of automatic (non-cooperative) garbage collector is essential in this kind of environment.

- *Simplicity, scalability and efficiency*

  The proposed object model should be easy to use and as efficient as possible.

3.2. Object Model Mapping to C++

As can be easily seen, the design goals presented above require non-trivial extensions to the underlying run-time and/or language. For example, in order to ensure that objects can migrate transparently, the Core must be able to automatically serialize objects, create object instances, etc. Since objects can point to remote objects (due to the transparent migration the determination of what is local and remote would be volatile), the Core have to have a concept of general persistent pointers. Such pointers must be able to reliably reference any object managed by the distributed model, regardless if it is local or remote (local native pointers can not be used at all). The mapping to the C++ language has to be straightforward and error prone. The Core has to be able to dereference such pointers and detect potential errors (such as invalid references, etc.).

We did not want to implement a completely new language or modify an existing one too much, so we decided to implement all the needed extensions in pure C++. This results in fact that the model itself is tightly coupled with the C++ language and the new features are accessible through regular language constructs. Application objects that should be managed by the Core must be written (implemented) in a "standardized" way enforced by the model. The rest of the book describes this in detail.

3.3. Object Model Basics

The model allows to implement serializable objects that would be managed, persistent, could migrate, safely reference each other, be replicated, garbage collected, etc. Migration and replication groups of objects are defined implicitly as transitive closures of persistent pointers that are treated, for purposes of migration/replication group enumeration, as if they were bidirectional. The introspection API allows to access object metainformation at run-time. This is utilized by the Core for implementation of the features above but could be used by an application too. The object to object migration is a primary way of object collaboration. The RPC is built on top of migrations.

3.3.1. Properties

One of the key features the Core provides is the *automatic data serialization*. There is a set of predefined *primitive* types that must be used by the user instead of the native C++ types because the latter do not implement the *serialization interface*. These types are called the *properties* and are derived from `Massiv::Core::Property`. These properties can be simple integers, floats, strings or complex compound types that were built from other properties. If their structure can be changed at run-time they are called *containers*. Currently arrays, dictionaries and sets have been implemented. Containers always hold stored properties by value and are strongly typed (i.e. array of integers, array of pointers, set of strings, etc.). Any property is serializable without an user's assistance.

Generally, properties can by hierarchized into tree-like dynamic structures (properties can be added/ removed at run-time). Each property maintains a reference to the owner property and various replication related attributes. When property value changes, the write operation is propagated to the root of the tree and the owner properties are thus notified that the value has changed. This allows an efficient replication of the whole hierarchy as only the subtrees that have been modified since the last replication update need be serialized. A property hierarchy can be enumerated at run-time.

Note

When talking about a hierarchy of properties, we do not mean some dynamic tree-like structure managed by the programmer. The programmer uses standard constructs - classes with propertie,s container properties, etc. Because of the way the property types are implemented, the Core knows everything about property hierarchy - it can easily determine list of properties that are members of an object or a container, which object or container owns given property, etc. And that's the hierarchy that we talk about. The hierarchy can change at run-time, because properties can be added and removed from containers.

3.3.2. Objects

The Core allows to define objects that will be fully managed by the Core on the application level. Any such object is a special kind of property and must be derived from `Massiv::Core::Object` (or a subclass). Managed objects can contain other properties as their data members, including managed objects, or can inherit from other managed objects. Multiple inheritance is allowed too as far as `Massiv::Core::Object` is inherited exactly once. The structure of managed objects, mainly the inheritance hierarchy and data members, must be described in external `.idl` files. IDL files are used to generate helper classes that are utilized by the Core to perform automatic object instantiation, serialization and the like. See Chapter 9, *Introduction to IDL* for more information about the IDL.

Managed objects can be classified according to the type of their usage, instantiation and lifetime.

- Objects with a *pointer semantics*

  They are often instantiated as "stand-alone" (addressable, on the heap) objects. Such objects have their unique identification (`Massiv::Core::ObjectId`), can migrate and can be referenced by persistent pointers. They are often accessed through pointers only and are destroyed by the garbage collector.

- Objects with a *value semantics*

  They are instantiated on the stack, owned by other objects or containers. Such objects are not addressable, can not migrate and can not be referenced. They are used as "value objects". Their life time corresponds to the lifetime of the owner object or, if instantiated simply on the stack, standard scoping rules.

- *Throwable objects*

  Managed exceptions objects, exceptions thrown by application code or the Core itself. The Core is able to manage and propagate them to caller nodes during remote calls.

Note

Under special circustances, objects with the value semantics can be instantiated also as "stand-alone" objects and objects with pointer semantics can be instantiated on the stack using a special construct. This is because the Core can instantiate *any* property either "stand-alone" (on the heap) or on the stack. The only thing that must be ensured is a proper object initialization. The above taxonomy classifies objects according to their "most common usage" or "how they should be used". The classification could be done on the per-instance basis.

3.3.3. Pointers

The Core implements a concept of persistent pointers to objects. These pointers encapsulate `Massiv::Core::ObjectId` and can point to components (super classes) of addressable objects only. In this way they are similar to Java references.

The pointers can have either *local* or *remote* dereference semantics. The pointers with the local semantics are able to access local objects (including local replicas) and fail to dereference if the target object is remote. The access to both methods and attributes is gained (in other words everything that could be accessible through a corresponding native C++ pointer would be accessible too). The pointers with the remote semantics access RPC methods of referenced objects. *The important thing is that both local and remote objects are referenced by the managed pointers and the C++ native pointers must not be used at all.*

Pointers can also be either *strong* or *weak*. This is related to the *garbage collector* and the way how pointers are interpreted by it. GC periodically scans for unreachable objects (objects unreferenced by strong pointers), and deletes them. Scanning originates from the program stack and special *GC root* objects. See also Section 5.4, “Garbage Collector”.

Strong pointers can reference local objects only. Thus they also implicitly define migration groups of objects. If an object *A* that strongly references object *B* migrates (or *B* migrates), both *A* and *B* will migrate. *This ensures that strong pointers will always point to local objects*.

Pointers can be annotated by *pointer replication flags*. When enumerating the replication group of an object, pointer replication flags mask must be given. Replication group is then defined as a transitive closure of bidirectional pointers that match specified replication flags mask.

Note

Automatic (implicit) migration and replication group definition/management is a key property of the Core that makes it unique. *It gives a programmer an opportunity to easily partition his objects to compact groups and isolate data flow between the groups* (a programmer can thus express his intention that a referenced object should always be local in respect to the object holding the relevant pointer; access to such object can be optimized then). The implicit definition of replication groups offers an easy-to-use object replication for free. *The groups change at run-time according to pointer changes.*

Part II. Using Object Model

Now you already know the basic ideas and it is time you immersed deeper into the Massiv object model. This part gives you a much more complete information than what you got in the overview. It should train you enough to be able, with some help of the Massiv Core Reference Guide, to start programming using the object model that encapsulates all the key features of the Massiv.

Table of Contents

4. Managed Objects

4.1. Introduction

From the Object Model Introduction chapter you should already have been acquainted with the Massiv object model basics, which also includes the basic ideas of managed objects. This chapter extends the information about managed objects in a much more detailed way. However, in order to present the full picture here, it also summarizes the basic facts that were already mentioned above.

As you already know from the previous chapter, the Core is proposed to be able to perform some operations on some of the *application objects* that would be very hard to implement only using the C++ constructs available in the standard. The objects can be migrated to another node, replicated to other nodes to make the communication more efficient, communicate via the RPC mechanism, saved into an archive and restored again. Least used objects can be swapped out and in again on demand; objects that are no longer needed may be automatically destroyed.

It is probably obvious that the operations above require the object to be able for example to save itself into a stream (*serialization*) and be restored again on a potentially different node.

On the other hand, it is clear that the Core must handle these objects in a special manner - for example, it must be careful during construction of the object as one object can be instantiated multiple times (for the first time and then each time it is restored from an archive, swapped in or migrated). It *must not* use the C++ native pointers to refer them because the target object could migrate making the pointer invalid, which could result into the application crash.

There are much more rules and principles about these objects and they will be all explained below. The most significant advantage the Core provides to its user is that all the mentioned special operations can be done *purely automatically* by the Core. It doesn't mean that you would never have for example to request replication manually; it means that you as the application programmer don't have to write any code in your object to make it serializable, archivable, etc. All the functionality is "granted for free" to you. The only cost you pay for all this is that you have to describe the relevant class in a special meta-language (IDL) and to respect a few implementation limitations.

Note

We say that the objects that can be handled as described are *managed* by the Core. Therefore they are called *managed objects*. Also the relevant classes are called *managed*.

Managed objects in general are instances of any C++ class that is derived (either directly or indirectly) from the `Massiv::Core::Object` base class and respects some implementation limitations.

The implementation limitations required by the managed objects may sometimes be too restrictive. For example, if some types of objects are always instantiated on the stack or handled as a value (i.e. passed into RPC calls by value, etc.), using the standard instantiation would be too cumbersome. For this reason the Core distinguishes among multiple kinds of managed objects. They have been already listed in the previous chapter, but let's repeat the survey once more because it is an important issue:

- Objects with the *pointer semantics*

  They are often instantiated as *stand-alone* (addressable, on the heap) *objects*. Such objects have their unique identification (`Massiv::Core::ObjectId`), can migrate and can be referenced by persistent pointers. They are often accessed through pointers only and can be garbage collected (i.e. automatically destroyed when the Core concludes they are not needed any more).

- Objects with the *value semantics* (value types)

  They are instantiated on the stack and owned by other objects or containers. Such objects are not addressable, can not migrate and can not be referenced. They are used as "value objects". Their life time corresponds to the lifetime of the owner object or, if instantiated simply on the stack, standard scoping rules.

- *Throwable objects*

  Managed exceptions objects that can be thrown by application code or the Core itself.

The actual requirements of each specific category of objects will be described below in Section 4.3, “Managed Object in Detail”.

4.2. Managed Data

The managed objects should not work with classic C++ class members. The reason is that these don't implement the *serialization interface* that is needed for a functional support for class serialization that was mentioned in the previous section.

Instead, the Core proposes special data objects that fully conform to the principle that an user doesn't have to assist a class to become serializable, etc.

The Core distinguishes between two major types of managed data: *properties* and *lightweight serializable types* (we will simply call them *STypes* in the further text).

Whereas the latter implement only the serialization interface and don't provide any enhanced functionality, the former also contain some extra information about property ownership, etc. See Section 4.2.1, “Properties” for more information about the properties and Section 4.2.2, “Lightweight Serializable Types” for the STypes.

Note

The serialization process of properties doesn't consume more CPU time than the serialization of STypes, although some other operations on STypes may be more efficient. Also the format of binary or textual serialization of some variable is the same regardless on whether it is a property or a corresponding SType.

Note

In fact the managed objects themselves are *also* managed data (specifically properties). It is perfectly legal to have managed objects as data members of another managed object.

4.2.1. Properties

Properties are instances of classes derived (directly or indirectly) from the `Massiv::Core::Property` base class that, moreover, conform to some special implementation rules. Usually the relevant type names' first letter is `P`.

Properties have the following specific traits or features (see below for more detailed information):

- Each property has a pointer to an *owner* property used to build hierarchies of properties. For example, if a property is simple member of property of class `Foo`, its owner is an object of type `Foo`. If a property is member of a container, the containter is its owner. Each property must be a member of some hierarchy - even those that are instantiated on the stack. A property that is not owned by another property is called *root* of the ownership hierarchy. Property hierachy information is maintained by the Core automatically.
- Whenever a property is changed, the information about the change is propagated up the relevant property hierarchy. Finally the root of the hierarchy is informed. This is useful for making some part of the hierarchy *dirty*. During replication, only the *dirty subhierarchy* has to be transmitted over the network which saves the bandwidth.
- Each property is able to enumerate all properties it owns - objects know about all properties, containers know about all contained properties.
- Each property has its *replication flags* that determine what part of the property hierarchy would be replicated together with the object (see Chapter 7, *Replication* to learn more about the replication).

Note

Each property contains many auxiliary attributes. Thus, it consumes relatively large piece of memory. If you want a more efficient handling with the type and don't need the special features mentioned above, you should use STypes rather than properties. See also Section 4.2.2, “Lightweight Serializable Types”.

The Core provides a reasonably wide set of property types available for the application. Properties can be either *plain types* such as integers, floating point numbers, *pointers* (in fact, pointers are also plain) or *containers* (the Core offers *arrays*, *dictionaries* and *sets*). Containers can hold instances of other properties by value (but always all the properties contained within must have the same type as the containers in Massiv are *strongly typed*), which includes even managed pointers and objects.

The odds are that the Core user will never need to write his own property (which wouldn't be as simple as just implementing the relevant class - you also would have to modify the IDL preprocessor). However we will still mention the implementation requirements, because

- you will acquire a better confidence about what actions you can do safely with properties
- managed objects are the only properties that will be implemented by an user

The following section describes the property implementation and usage issues. Any property implementation must be replicable and must contain:

- Explicitly defined constructors. When a property is created, it should register to some existing property hierarchy or become a root of a new one. By default, properties are being registered to the hierarchy with root `StackRootObject` that primarily contains all properties instantiated on the stack.

  If the property should be inside another hierarchy, it must be explicitly reinitialized. This can be done using several methods, for example `initialize_from_owner()` or `initial-ize_object()`.
- The assignment operator (`operator=()`). See the Massiv Core Reference Guide, module Properties for more information.
- Serialization interface (see also Massiv Core Reference Guide, module Properties).
- Enumeration interface to access owned properties. For example this interface contains the `for_each_owned_property_do( ... )` method that enables to perform a specified operation on each property owned by the actual one. For more information see also Massiv Core Reference Guide, module Properties).

This chapter doesn't contain a list of specific property types available to the application. Instead, please refer to Section 10.13, “Property and Argument Types” for a somewhat brief information about the types usage and semantics. For the complete information see also the Massiv Core Reference Guide, module Properties.

4.2.2. Lightweight Serializable Types

Lightweight serializable types (also called *STypes*) are similar to the properties, but are much simplified. They only implement the *serialization interface* and some useful methods (see the Massiv Core Reference Guide, module Lightweight Serializable Types, to find out which ones), but don't contain the attributes such as the owner pointer, replication flags, etc.

In practice the STypes are useful only to be stored in containers or instantiated on stack more efficiently than properties.

Note

Unlike properties, the lightweight serializable type names are usually prefixed with the `S` letter.

Note that STypes and property types are often defined *in pairs*. I.e. for each SType there exists a corresponding property and vice versa. There sometimes is also a native type (meaning *C++-native*) defined for each SType.

The property and native types for some specific SType can be obtained via the following type definitions nested into any SType class (where *`NATIVE`* and *`PROPERTY`* would be replaced by the relevant type name):

```
typedef NATIVE    NativeType;   // Native type corresponding to the SType.
typedef PROPERTY  PropertyType; // Property type corresponding to the SType.
```

Note

If SType is not coupled with any native type, `SType::NativeType` should be set to the type that is returned by the `read()` method (see below).

Each SType must have the following methods implemented (only those that might be interesting for the application have been enlisted):

- `read()` returns the SType value that should be convertible to the native type.
- `to_string()` returns object value as a string.
- `memory_size()` returns a number of bytes occupied in the memory by the object.

4.3. Managed Object in Detail

This section covers most of information about managed objects. However, you need to get acquainted with the IDL and some other topics more properly to be able to write a managed class in practice. Chapter 13, *Creating Managed Class* shows in detail and using a practical example how to create a managed class step-by-step. However, before reading it, you should already have read all the chapters between this and that one.

Note

Note again that the managed objects are properties as well (`Massiv::Core::Object` inherits `Massiv::Core::Property`). They can act not only as property hierarchies roots, but they can also be owned by another properties.

Note

Each managed object is associated with two special objects. The *metaobject* holds the meta-information about the relevant class (its base class, method list, ...) and enables *introspection* that is useful for the Core to instantiate managed objects, etc. (but can be used by the application as well).

The *object factory* is responsible for objects and replicas instantiation.

Let's first look at an overview about what steps you need to accomplish when implementing a managed class. A step-by-step example is available in Chapter 13, *Creating Managed Class*.

- Implement the class in C++ according to the rules mentioned below in Section 4.3.1, “Implementation”.

- Write an IDL description for the class. In particular the description contains information about the inheritance hierarchy, class methods (together with specification which parameters are *input* or *output*), owned properties, class attributes (such as whether the class is permitted to be archived, etc.) The IDL preprocessor generates source code for the relevant metaobject, object factory and several more auxiliary classes.

For more information about the IDL Chapter 9, *Introduction to IDL* is a good starting point. See also Chapter 12, *Metaobjects* for more information about metaobjects.

- Add an entry into the relevant `idl.list` file. Each entry specifies one idl file involved in an application and is needed to make the preprocessor to take the file into account.

  To get more information about the `idl.list` see Section D.2, “The idl.list File”.

4.3.1. Implementation

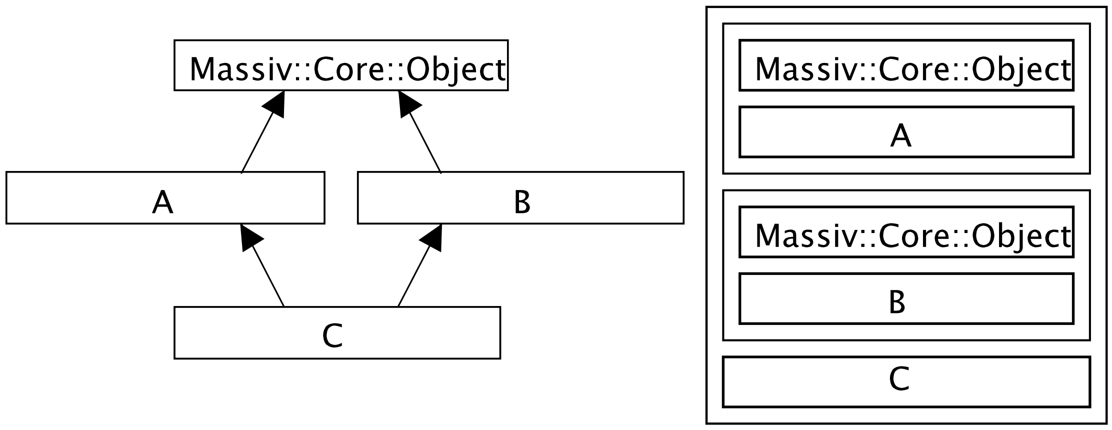
As already mentioned, managed objects technically are instances of any class derived (directly or indirectly) from the `Massiv::Core::Object` base class that moreover conforms to some implementation restrictions and limitations. All the limitations are described in the following list:

- *Inheritance*

  Firstly, all managed classes should use only the *public* inheritance.

  If you want to use the multiple inheritance for your managed objects, it is an important requirement that the objects must inherit `Massiv::Core::Object` *exactly once* (i.e. only one instance of the `Massiv::Core::Object` component is permitted). For example the situation in the following picture is invalid:

  Figure 4.1. Misused multiple inheritance



  The left half of the figure represents the inheritance hierarchy, whereas the right one shows the component structure of the result class `C`.

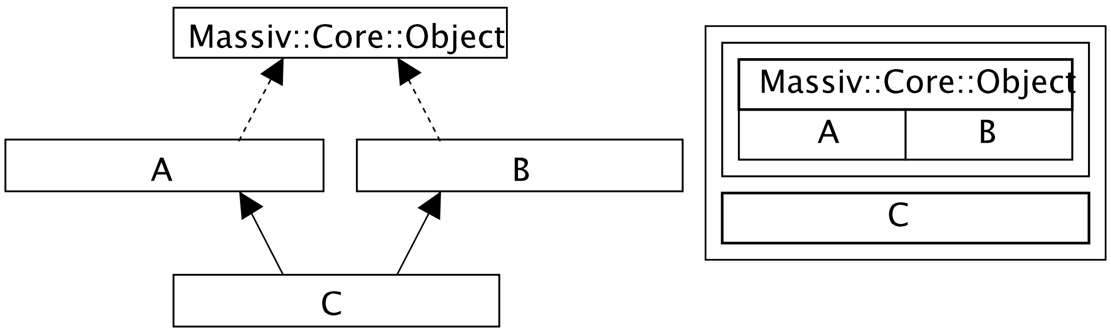
  Classes `A` and `B` both inherit `Massiv::Core::Object`. The class `C` is derived from both `A` and `B` using the multiple inheritance.

  To avoid multiple instances of the `Massiv::Core::Object` in the managed object, it is *strongly* recommended either to use virtual base classes or to avoid multiple inheritance at all. Example:

```
class A : virtual public Massiv::Core::Object
    {
    ...
    };

class B : virtual public Massiv::Core::Object
    {
    ...
    };

class C : public A, B
    {
    ...
    };
```

The class `C` structure is shown in the following figure:

Figure 4.2. Multiple inheritance properly used



The dashed lines stand for the virtual inheritance (i.e. the base class marked `virtual`).

- *Construction*

  Because managed objects can be generally constructed multiple times, the application programmer should not write his own user-defined constructors. Instead, the Core takes over the responsibility for managed objects instantiation and initialization (the relevant code is generated on the IDL basis).

  Instead of constructors, the programmer must supply the `initialize()` pseudoconstructor that will be called by the Core *only once* after the object's first instantiation. The pseudoconstructor must properly initialize the object itself, its base classes (typically by calling the inherited `initialize()` methods) and member objects (the similar way).

  `initialize()` is not being called automatically by object factories. You must either call it manually or use special templates described below (see Section 4.3.2, “Instantiation and Finalization”).

Note

Although they are also managed object descendants, value types and throwables are exception from this rule, see below. This fact adds some implementation requirements, but simplifies their usage.

- *Access specifiers of methods*

  All methods defined within managed objects that can be called using RPC (i.e. methods that are specified in the IDL) must be declared *public*.

- *Overriding `Massiv::Core::Object`'s methods*

  Some of the methods in `Massiv::Core::Object` are virtual. However, in derived managed objects, the programmer should override *only* those that are *explicitly* tagged as virtual in the `src/core/object/object.h` file. Other methods, even those that are declared virtual in some of the `Massiv::Core::Object`'s ancestor classes, may *not* be overriden.

- *`MASSIV_OBJECT`* macro

  Unlike methods, properties owned by managed objects can be *private* or *protected* as well as *public*. To enable the Core to initialize and handle a managed object properly, the relevant metaobject and object factory must be allowed to access all properties contained in the object. This can be achieved by declaring metaobject and object factory as *friend classes*. You can use the `MASSIV_OBJECT` macro to do this for you.

```
class MyManagedClass : public Massiv::Core::Object
    {
    MASSIV_OBJECT( MyManagedClass );
    ...
    }
```

4.3.2. Instantiation and Finalization

Because managed objects has to be initialized by the Core, you cannot instantiate them using the standard C++ `new` operator. Instead, there exist some special templates the application programmer is supposed to use.

The following list enumerates possible ways how to instantiate managed objects:

- `CreateObject` macro

  Creates a new class instance using the relevant object factory, does all the basic initialization and calls the user-defined `initialize()` pseudoconstructor.

- `ObjectOnStack` template

Use `ObjectOnStack` when you want to create an object on the stack. It is a template that encapsulates the "target" object on the stack. You need to use it to enable the Core to take control over the proper object initialization process. It is used as a *handle* to the stored object that can be accessed either by a defined conversion or explicitly using the `dereference()` method or the `operator->()`.

`ObjectOnStack` also calls the `initialize()` pseudoconstructor automatically. All parameters passed to the `ObjectOnStack` will be passed to `initialize()`. Note that output parameters are not allowed.

Note

You cannot reference objects created by `ObjectOnStack` by `ObjectPointer` instances.

Note

You don't need to use the `ObjectOnStack` template for the value types, because they can be constructed and initialized using the standard constructors.

The following example shows how to create an object on the stack. Let's suppose we work with a managed class `Foo` and a value type `ValueTypeFoo`, both having a method `foo`:

```
{
ObjectOnStack< MyManagedClass > my_object( ... );  ❶
MyValueTypeClass                my_value_type_object( ... );  ❷

my_object->foo();  ❸
( ( MyManagedClass & ) my_object ).foo();  ❹
my_value_type_object.foo();  ❺
}  ❻
```

❶ Instantiation of managed object with pointer semantics on the stack. `initialize()` will be called automatically.

The ellipsis stands for a list of parameters that would be passed to the `MyManagedClass`'s `initialize()` method.

❷ Instantiation of value type on the stack. There is no need to call `initialize()`, because standard constructors are sufficient. It can be handled the same way as unmanaged objects.

❸ Invocation of the `foo` method on the object created using `ObjectOnStack`.

❹ This is another way how to invocate the method on the managed object on the stack. As you can see, this is much more cumbersome than the previous one and thus we do not recommend it. However this line should demonstrate that the retyping operator or the `ObjectOnStack` wrapper works.

❺ Unlike in the previous case, here the value type doesn't have any "wrapper" and thus neither dereferencing using the `operator->()` nor retyping is needed.

❻ At the end of scope both `my_object` and `my_value_type_object` will be destroyed.

- *ObjectFactory*

Of course you can create object using the relevant object factory. However, this isn't a way you are supposed to take. It is more difficult and you would have to call `initialize()` by hand.

Managed object that are not created on the stack will be destructed automatically by the garbage collector (see Section 5.4, “Garbage Collector” for more information). The on-the-stack objects are destroyed according to the standard scoping rules. Because they cannot be referenced by managed objects, there cannot remain any invalid pointer to the object after it is destroyed.

4.3.3. Referencing

It is strictly banned to use native C++ pointers or references to reference managed objects. The reason should be already obvious: the Core can for example migrate or swap the object out anytime. The native pointer would thus become invalid and its dereferencing would cause an application crash.

A safe way to reference managed object is to use `ObjectPointer`. Pointers won't be discussed in this chapter, see Chapter 5, *Pointers* instead.

4.3.4. ValueTypes

Value types are descendants of the `Massiv::Core::ValueType` class. Use value types for managed objects that can be handled *as a value*, i.e. they don't need some special initialization, copying and handling from the Core. Their purpose is to enable application to implement property-like structures *outside the Core*.

Value types advantages and drawbacks are summarized in the following list:

- It is more simple to create them on the stack.
- They can be used as arguments or result types of RPC methods.
- They can be used in place of properties.
- They can be instantiated the same way as properties in most of places.
- They should not be created as a stand-alone objects (i.e. on the heap).

The most significant difference of an implementation of a value object from an implementation of a "standard" managed object is that the former doesn't have the `initialize()` pseudoconstructor. Thus, they cannot be instantiated using the `CreateObject` or `ObjectOnStack` constructs. Instead, you should handle them as values (which means, besides others, that you also cannot use C++ operator `new` for their instantiation).

The following list summarizes implementation restrictions you must respect while writing a value type class:

- *Construction*

  Unlike for general managed objects, you are supposed to write your user-defined *constructors* for value types (even to override the default constructor). You should use general managed object instead in case that the fact the object can be instantiated multiple times matters.

Each constructor must call `initialize_object()` as its first command. It enables Core to do some initialization of its own.

Default constructors are expected to call `default_constructed()` to indicate that the object has been created the default way.

Warning

Default constructors of value types *must not* interact with the simulation.

- *Copy constructor*

  Each value type must have a copy constuctor implemented. It must also call the inherited copy constructor.

- *Destruction*

  Value types cannot have user-defined destructors.

- *RPC methods parameters*

  Value types that can be used as method parameters in RPC calls cannot contain strong pointers (pointers that always references local object, see Chapter 5, *Pointers*) within. The reason is that the process of transmitting them over the network is simplified.

- *Embedding*

  Value types can embed each other in the natural way.

- *Property restrictions*

  Value types are properties, thus all restrictions of properties apply for them (unless stated otherwise).

The following code listing shows a complete example of a value type implementation:

```
class MyPair : public ValueType
    {
public:
    MyPair()
        {
        initialize_object();  ❶
        default_constructed();  ❷
        }

    MyPair( int a, int b ) :
        a( a ), b( b )
        {
        initialize_object();
        }

    MyPair( const MyPair & pair ) :
        ValueType( pair ),
        a( pair.a ), b( pair.b )
        {
        initialize_object();
        }

    PInt32  a;
    PInt32  b;
    };


class Derived : public MyPair  ❸
    {
public:
    Derived() :
        MyPair()
        {
        initialize_object();
        default_constructed();
        }

    Derived( int a, int b, int c ) :
        MyPair( a, b ), c( c )
        {
        initialize_object();
        }

    Derived( const Derived & derived ) :
        MyPair( derived ),
        c( derived.c )
        {
        initialize_object();
        }

    PInt32  c;
    };
```

❶ `initialize_object()` must be the first command in each constructor.
❷ The `default_constructed()` indicates that the object has been constructed the default way.
❸ Of course value objects can inherit each other. The same rules as for managed objects apply, i.e. inheritance must be public and should be virtual if multiple inheritance is used.

Value types can be handled as values:

```
{
MyPair          pair;  ❶

PArray<MyPair>  pair_array;
pair_array.push_back( pair );  ❷
...
}
```

❶ Value types can be instantiated as properties. No `CreateObject` or `ObjectOnStack` constructs.

❷ They can be handled as values - inserted into containers, etc.

4.3.5. Throwable Objects and Exceptions

"Throwables" are managed objects (in fact, they are value objects) that can be thrown as exceptions. The Core is able to propagate them to the caller nodes during remote calls. In fact they are *one purpose objects with value semantics*.

You can throw a throwable object using the standard `throw` keyword. However, you can also create the object by hand and throw it using the `raise()` method (it throws a copy of the throwable object).

Note that not all exceptions used by the Massiv are managed objects. The Core itself implements its own hierarchy with base class `Massiv::Core::Exception` inherited from `std::exception` (the generic exception defined in the STL library). The `Massiv::Core::Throwable` class is also included into this hierarchy - it adds the managed objects' functionality to the exception interface of `Massiv::Core::Exception`.

Note

The exceptions implemented by the Massiv that are not `Throwable` descendants should be neither thrown nor catched by the application code (the Core should never let its internal exception leak into the application code). If a non-managed exception is thrown by the Core on the target node during the RPC call, the Core *itself* will catch it and remap (wrap) into the `Massiv::Core::Lib::CoreException` that is inherited directly from `Massiv::Core::Throwable`. It enables even the internal Core exceptions to be transmitted back to the caller nodes.

The only exceptions the application code should work with are `Massiv::Core::Lib::RuntimeException` descendants. Feel free to define as many these exceptions as you need, but don't forget that the managed exceptions need their IDL description.

Note

The Massiv exceptions doesn't make (nor intend to make) the Massiv "*application-programmer-resistant*". They should be used for notification about extern failures that can't be simply influenced by the programmer. For example sending wrong parameters (wrong format, type, ...) using the RPC won't probably throw an exception but the application will crash. It is the programmer's responsibility to make sure he uses the Massiv Core correctly (use the debug build of the Core to force more extensive tests).

Note

Managed exceptions that won't be catched at all *won't* stop the simulation; they only will be processed by the logger.

4.3.6. Callbacks and Event Scheduling

An application programmer can schedule an object for migration (i.e. transmitting the object to another node, see Chapter 6, *Migration*). The Core will inform the object about a result of the operation by calling a defined callback (a virtual method that can be overriden in the specific managed object).

The main methods that can be used to make an object migrate are:

Method	**Notes**
`EventHandle schedule_to( ... )`	Enables to schedule the object migration to a specific time, The migration is addressed by another object (referenced by a weak pointer). The event is scheduled *public*, i.e. it can be cancelled anytime using the handle returned from the method.
`void migrate_to( ... )`	This method does the same as the previous one, but doesn't return any handle. Thus, the event is scheduled *private* and cannot be cancelled.
`void deliver_asap_to( ... )`	The main difference from `migrate_to()` is that if the target object is local, the delivery will be done *immediately*. Note that the informative callback will already have been performed when `deliver_asap_to` returns. However, it is a good idea not to make any assumptions about this order.

Let's now have a look at the callbacks that are used by the Core to inform the object about the migration result. Note that the callback will be always called *after* the migration, i.e. potentially on a different node than where the migration was requested.

Callback	**Notes**
`void delivered_to( ... )`	Called upon the successful object delivery. The system passes the destination object (guaranteed to be local) and the delivery time as parameters.
`void delivery_failed( ... )`	Called when the object migration has failed. The system passes the failure reason as a parameter.

To get a complete picture, let's describe one more managed object callback. This one has just a little in common with migration:

Callback	**Notes**
`void object_updated( ... )`	Notifies object about its state change [a].

[a] Possible change values that can be passed to `object_updated()` are:

- `INSTANCE_CREATED`: New instance of the object has been created.
- `INSTANCE_MIGRATED`: The object just migrated to the local system (or has been loaded from the swap/archive).
- `INSTANCE_FINALIZE`: The garbage collector has just triggered the object deletion. This is a good place to perform the object-specific cleanup.
- `REPLICA_CREATED`: New replica of the object has been created. `REPLICA_UPDATE` and `REPLICA_UPDATED` will be sent too.
- `REPLICA_UPDATE`: Object replica contents will be updated shortly.

- `REPLICA_UPDATED`: Object replica contents has been updated.
- `REPLICA_DESTROY`: Object replica will be destroyed.
- `SIMULATION_STARTUP`: Notifies object about starting the simulation. A managed class will be notified *only if* the `simulation_startup_notify` is set to `true` in the IDL.

4.3.7. Class Kinds

Typically nodes running a Massiv application are divided into servers and clients. Servers collaborate on the simulation, whereas clients are responsible for presentation of the simulation state to users (players). Both client and server nodes work with managed objects, but, however, they don't need the same set of them. Servers work with object that are needed for the simulation, clients with presentation objects and shared objects are used for communication between them.

The Massiv offers a mechanism that enables to link each type of node with code of only those objects that it really needs for its work. It is advantageous both for efficiency (it cuts down the size of the executable files) and safety (it prevents client from illegaly modifying classes that are able to influence the simulation).

This mechanism is called *class kinds*. The Massiv distinguishes among three class kinds: `KIND_SERVER`, `KIND_CLIENT` and `KIND_SHARED`. Only the last kind is supported explicitly by the Core; the others are defined in `src/core/object/object.idl`. The *kind* for any managed class can be specified inside the relevant IDL file (see Example 10.1, “A class description in the IDL” for example).

Managed classes of the same class kind are grouped into a single compilation/linkage unit. Each node supports its specific set of class kinds, i.e. it has the relevant meta-data (needed to instantiate and manage the classes) and the code of the classes.

The class of kind that is not supported by some node is called *alien* class in respect to that node.

The following list shows the rules that you need to respect while implementing managed classes:

- A managed object of *any* kind can be inherited from another object of `KIND_SHARED`.
- A managed object inherited from another object of another class kind than `KIND_SHARED` must be of the same class kind as its ancestor. For example, you *cannot* create managed object of `KIND_SHARED` by deriving it from `KIND_SERVER`.

The following list summarizes rules that are obligatory to the Massiv build system and the Core:

- `KIND_SHARED` classes code will be linked with code of *all* types of nodes.
- `KIND_SERVER` classes code will be linked with code of *only* server nodes.
- `KIND_CLIENT` classes code will be linked with code of *only* client nodes.

A node can't instantiate or otherwise use objects that are alien in respect to it. It is not surprising, because we already know the node doesn't have the relevant code and meta-information available. As a consequence, `KIND_SERVER` objects aren't allowed to migrate or to be replicated to client nodes and vice versa. This ensures for example that clients can *never* modify server objects.

Despite the previous paragraph, a node can reference alien objects using the *remote* pointers. It enables calling methods using the RPC (see Chapter 8, *Remote Procedure Call*) even on "aliens".

4.4. Object Identification

4.4.1. ObjectId Overview

All stand-alone managed objects are uniquely identifiable (addressable) through out the distributed simulation. Each object is assigned an unique number called `ObjectId`. The `ObjectId` of an object never changes and does not depend on the object's current location.

4.4.2. ObjectId Pool

Once a new managed object is created, the Core assigns an unique `ObjectId` to the object. Because managed objects can and often will be created on more than one node in the same time, the Core provides mechanism to prevent assigning the same `ObjectId` to two or more objects.

Each node assigns new `ObjectId`s from its own pool of unused `ObjectId`s. The pool is managed by the Core's internal object `ObjectProvider`. Once a new object is created, the Core asks the `ObjectProvider` to generate a new `ObjectId` for the newly created object. Because each node has its own `ObjectProvider` and identification of the `ObjectProvider` is encoded in the `ObjectId` itself, it is ensured that two objects created on different nodes will never be assigned same `ObjectId`.

`ObjectProvider`s are identifiable through their `ProviderId`s. The `ProviderId` is encoded in all `ObjectId`s generated by the corresponding provider.

4.4.3. Mandatory And Optional Part

`ObjectId` consists of two parts. The first part is mandatory, the second part is optional.

- The mandatory part of `ObjectId` contains information that is needed to properly localize the object in the distributed simulation. The mandatory part uniquely identifies the object and there can't be two or more objects with the same mandatory part.

  The mandatory part consists of *`ObjectProvider` identification*, *`ObjectProvider` relative number* and *tracked-by-provider* flag.

  The *`ObjectProvider` identification* is `ProviderId` of the `ObjectProvider` which generated the `ObjectId`.

  The *`ObjectProvider` relative number* is a number which identities the object in the provider's pool.

  tracked-by-provider is one-bit flag which specifies whether the object is tracked by the object provider. Objects with this flag set can be found in the distributed simulation more quickly than other objects because the object provider monitors which node the objects resides on. Why not mark all objects *tracked*? The reason is that all *tracked* objects increase network communication among the nodes whenever the objects migrate. Also because the object providers store additional information on which nodes the objects are located, memory requirements of the providers increase as well.

- The optional part of `ObjectId` is also called `ObjectInfo` because it contains additional information about the object.

  What the "optional" word means? It means that even if you omit the `ObjectInfo` part of the `ObjectId`, you can still address the object in the simulation by the truncated `ObjectId`.

  The `ObjectInfo` consists of `ClassTypeId` and *archivable* flag. The `ClassTypeId` is identification of the object's class and specifies the object's dynamic type. The *archivable* flag specifies whether the object is archivable or not.

  Since the Massiv supports class kinds and separate compilation, the optional part may not be interpreted by all nodes properly (node does not understand type information of an alien class, for example). If such node can not interpret the additional information it either discards/truncates it or leaves it uninterpreted.

4.4.4. `ObjectId` Uniqueness

`ObjectIds` are never recycled. Even if an object is destroyed, the Core will never assign its `ObjectId` to another object. One `ObjectProvider` can generate about four billions of object identifications so one doesn't need to worry whether the pool can get exhausted. Anyway if this would become an issue, the Core supports two or more `ObjectProviders` owned by one node.

4.4.5. Object Searching

The Core uses sophisticated methods how to find an object in the distributed simulation as quickly as possible. The searching for objects is used mostly for migrations when an object is migrated to another object (see Chapter 6, *Migration* for more information). In such case the latter object must be found somehow in the simulation.

You don't need to know how exactly the searching algorithms internally work. The important is that it is ensured that if the object exists somewhere in the simulation, it will be found.

To find an object, the Core uses several localization methods:

- First of all, each node maintains a cache of recent migrations. The cache tells for an object that migrated from the node some time ago, on which node the object is probably located. The Core supports automatic corrections of the cache if the data in it are not valid any more. When an object is found and the content of the cache is not correct for the object, the cache is automatically updated.
- Each `ObjectProvider` maintains a cache of its *tracked* objects. Each tracked object notifies its `ObjectProvider` when it migrates so the `ObjectProvider` knowns where the object should be located.
- If the object is found neither in the former nor the latter cache, the Core performs *global search* to find the object. During *global search* the node contacts all other nodes and asks for the object. The

Core must properly handle situations when the object is not on any of the nodes but is currently being transferred over the network.

4.5. Managed Objects Defined By the Core

Most managed objects implemented by the Core are internal. However, there are two that are interesting even for the application: `NodeObject` and `AccountObject`. See Chapter 14, *Special Objects* for more information about them.

5. Pointers

This chapter describes the pointer model that is used in the Massiv.

5.1. Why Managed Pointers

One of the basic design goals is to allow the Core to delete local object instances without notifying an user. This is needed for implementation of transparent object migrations, when object instance on the local node is deleted, transmitted over the network and created again on the other node, object replication, etc. Because of this it is obvious that the application can not use native pointers (or references) to reference managed objects. *The Core must know all the references the application has made for various reasons (garbage collection, implicit migration groups definition). There must be a way to detect what objects are active (their code is running or data being used).*. References must be durable and must be able to identify all objects reliably, including remote objects and objects that migrated out of the local node. There should not be any major differences between a *local pointer* (that references an object that just happens to be present on the local node) and a *remote pointer* in terms of declaration and basic use. The Core should prevent client from dereferencing invalid pointers.

The Core solves these problems by defining `Massiv::Core::ObjectPointer`, which is a replacement for native pointers. It is a sort of "smart pointer" that satisfies all the discussed properties. *Using object pointers is nearly the only legal way how the application can reference managed objects, even local objects must be referenced by them. When we are talking about managed pointers, object pointers are ment.* .

Note

It is obvious that at some point managed pointers must be converted to native C++ pointers in order to access object members or invoke their methods. However this is done internally by a pointer logic. Accessed objects are *active-pinned (activated)* before the access and *unpinned* when the native reference is released (if you are interested in the trick that achieves this consult the Massiv Core Programmer's Documentation). The Core refuses to delete pinned objects (such a request will be automatically delayed until the object is unpinned). This allows application to access object members through the C++ `this`.

5.2. Overview

`ObjectPointer` instances represent safe portable references to managed objects, throw exceptions if incorrectly used and are able to catch such common errors as dereferencing invalidated or `NULL_ID` pointers, bad casts (if *type information* is available), etc. `ObjectPointer` instances hold this kind of static and dynamic information:

- *Static information* (statically encoded into the pointer instance or set up at the moment of pointer creation and can not be changed since then):

  - Identification of the interface of the object being refered, *cast object* responsible for accessing the interface (for more information about the cast object, see Section 5.3.2, “Using Cast Object”).
  - Dereference semantics (local/remote).
  - Pointer interpretation utilized by the Garbage Collector (see the Massiv Core Programmer's Documentation for more detailed information).
  - Pointer replication flags, used to define migration/replication groups of objects.

- *Dynamic information*

  - `ObjectId` of the referenced object, including its type information if available.

Note

ObjectPointers can point to "stand-alone" (addressable) objects only. For example one can not create `ObjectPointer` that accesses fourth element of an array embedded into an `Object`. In this way ObjectPointers are similar to Java or CORBA references.

Warning

Objects instantiated on the stack are not addressable.

5.2.1. Pointer characteristics

Object pointer instances can be characterized by three main properties:

- *Is it a stack pointer or a pointer property?*

  Stack pointers are ObjectPointers that can be instantiated on the stack only and that are not properties. They are always used as temporary pointers only.

  Pointer properties are real properties and thus can be stored in managed objects and containers. They can also be instantiated on the stack. The discrimination between stack pointers and pointer properties is purely because of implementation efficiency. Stack pointers are "stripped" versions of pointer properties, nearly as effecient as the native C++ pointers.
- *Is that pointer strong or weak?*

  This object pointer characteristics is related to pointer interpretation by the Garbage Collector and whether the object can point to remote objects.

  Weak pointers can reference remote objects and do not keep referenced objects alive (GC ignores weak pointers while scanning for unreachable objects).

Strong pointers always reference local objects and are processed by the Garbage Collector while scanning for unreachable objects. *The local characteristics is important as Core ensures that it is not violated when an object migrates. It is said that strong pointers implicitly define migration groups of objects* (see Section 6.4, “Migration Groups”).

- *What is the dereference semantics? Local or remote?*

  What happens when an object pointer is dereferenced? Two semantics have been implemented: *local* and *remote*. Local semantics allows to access local objects only, both its methods and properties. Remote semantics gains access to methods of local or remote objects, every call is translated to a RPC call. Object's properties can not be accessed via remote pointers.

5.3. Using pointers

Note that not all combinations of the three pointer characteristics (strong/weak, stack/property, local/remote dereference) are allowed, there may be restrictions on what objects can be merged into a migration group, strong pointer properties can not reference object replicas, etc. This limitations are called *pointer policies*. They are either checked at compile-time or run-time.

For your convenience and to simplify pointer usage and to name valid pointer characteristics combinations, the following abbreviations were defined (read on for more complete explanation):

Table 5.1. Managed Pointer Types

Pointer instance	Description	Flags [a]	Use primarily for
`Pointer< Object >`	Strong stack pointer to Object	S, L	Reference local object from the stack. Prevent it from being garbage collected.
`WeakPointer< Object >`	Weak stack pointer to Object	L	Reference local or remote object from the stack. Dereference locally. Use for accessing object replicas.
`Remote< Interface >`	Remote weak stack pointer to Interface	R	Reference local or remote object from the stack. Perform RPC.
`PPointer< Object >`	Strong pointer property to Object	P, S, L	Define a migration group of objects. Prevent referenced object from being garbage collected.
`PWeakPointer< Object >`	Weak pointer property to Object	P, L	Reference object from the same or other migration group or reference object replica.
`PRemote< Interface >`	Remote weak pointer property to Interface	P, R	Reference object from the same or other migration group or reference object replica. Perform RPC.

[a] Pointer type flags:

P: property
S: strong pointer, always points to local object
L: local dereference semantics
R: remote dereference semantics

Note

Here, `Object` denotes any managed object, `Interface` any managed interface (pure virtual managed object). The discrimination between `Object` and `Interface` is for explanatory reasons only (properties of `Object` are not accessible by remote pointers), it is okay to declare `Remote< Object >` or `PRemote< Object >` though.

As can be seen from the table the Core offers three types of pointers, all in the property or stack-only variant. Since checked pointer conversions from one type to other type are possible, one need not use weak pointers with local dereference semantics at all (they can be replaced by remote pointers; remote pointers can be converted to strong pointers before dereference in order to get local dereference semantics).

The following sections describe pointer usage in detail.

5.3.1. Declaring Pointers

Managed pointers are implemented as template classes that represent the reference and are accompanied by a set of overloaded operators so that the classes could mimic the C++ pointers. Their declaration looks like this:

```
/* Pointers with local dereference semantics. */

template< class Type, class Cast = DefaultCast< Type > >
class Pointer;

template< class Type, class Cast = DefaultCast< Type > >
class WeakPointer;

template< class Type, class Cast = DefaultCast< Type > >
class PPointer;

template< class Type, class Cast = DefaultCast< Type > >
class PWeakPointer;

/* Pointers with remote dereference semantics (RPC). */

template< class Type >
class Remote;

template< class Type >
class PRemote;
```

For example `Pointer< MyObject >` represents strong stack pointer to `MyObject`. Declarations of other pointer types would look the same.

Note

Pointers are default-initialized to point to `NULL_ID`.

Note

We have not explained the meaning of the optional *Cast* parameter yet. It refers to the *cast object* functor that is responsible for returning of the correct C++ pointer if managed pointer is locally dereferenced. Its purpose is to convert a C++ pointer to general object to a pointer to *Type*. If managed class inherits *Type* more times, *Type* is ambiguous within the class hierarchy and pointer dereference will fail. In that case user has to provide own *cast object* that would resolve such ambiguity. This is described in the following section. However if you do not use multiple inheritance, do not inherit the same class multiple times (for example virtual inheritance is used, which is supported by Massiv) or do not want to create pointers to ambiguous *Types*, you can skip this section.

5.3.2. Using Cast Object

One of the problems the Core has to deal with is what component of the referenced object should be accessed if pointer is dereferenced. Note that the reference must be represented at least by `ObjectId` of the referenced object and *Type* of the referenced component. Obviously for some object instances this would not be sufficient if *Type* is inherited multiple times by those instances. Because of this, the reference representation was augmented to hold a *Cast* cast object. Cast object represents the referenced component more precisely - it identifies not only its *Type* but also provides code how the component should be accessed (cast from C++ pointer to general object to the C++ pointer to the requested component).

The Core provides default implementation of the cast object called `Massiv::Core::DefaultCast`. It works correctly as long as *Type* is inherited once only by the relevant class. If that is not true then the default cast object would not be able to resolve the *Type* ambiguity and the dereference may fail. Then user has to implement its own cast object.

Any cast object must inherit from `Massiv::Core::CastObject`.

```
class CastObject
    {
public:
    virtual VariantPointer operator()( ObjectProperty * object ) const = 0;
    /* Implements cast from ObjectProperty * to requested component. */
    virtual const std::type_info & get_target_type_info() const = 0;
    /* Returns type information of the referenced component. */
    };
```

For more information on how cast object should be implemented consult `Massiv::Core::CastObject` in the Massiv Core Programmer's Documentation.

Note

User-defined cast objects are utilized by pointers with local dereference semantics only. Pointers with remote dereference semantics operate as if default cast object was used (requested interface must be unique for the referenced object or the operations would fail).

5.3.3. Assigning To Pointers

Ability to change pointer values and reference other objects is one of the key operations pointers have to support. Managed pointers overload operator=() and allow for such changes. Any such operation results in change of the pointer state, namely `ObjectId` of the referenced object. The actual assignment operation is preceded by optional implicit pointer conversions and validation or verification of the assignment.

Implicit pointer conversions:

- *Pointer upcasting*

  Ability to convert pointers to subclasses to pointers to superclasses. Also known as pointer subsumption. Requires compile-time check.

- *Pointer mutation*

  Ability to convert pointers with one semantics to pointers with different semantics. For example implicit conversions of strong pointers to weak pointers of the same type (reference the same interface). The backward conversion requires a run-time check.

Assignment verification:

- *Either compile-time or run-time interface test*

  Tests if referenced object implements requested interface. For example if one wants to change the value of a `Pointer< MyClass >` the compiler or system would test if the object inherits from `MyClass`.

  The *compile-time interface test* is performed when assigning a pointer of a known type to other managed pointer *without an explicit cast*. In this case implicit pointer conversions (in C++ pointers style, upcasts only) are allowed and this is checked by the compiler. Violation results in compile-time error.

  Note

  The compile-time check is similar to *concept checks* used by various C++ template libraries to test if a given template parameter supports a requested interface. If a violation is detected compile-time error in *check_default_cast()* will be issued.

  The *run-time interface test* utilizes type information of the referenced object. The test is performed when assigning an *explicitly casted pointer* (or arbitrary `ObjectId`) to other managed pointer. If type information of the referenced object is not available (or can not be interpreted by the local node), no test is performed. Pointer dereference may fail then. This can only happen when creating a reference to a remote object and system is not able to determine actual dynamic type of the referenced object.

  Note

Type information is stored as an optional part of the object's `ObjectId`. It is always present if object is local but can be missing if object is remote. System can discard it if it is not able to interpret it - for example if there is no code for such object (object can not be instantiated on the local node, it is said to be *alien* to that node).

In the Demo, client nodes do not know type information of server-only objects. This is related to class kinds. See Section 4.3.7, “Class Kinds”.

- *Additional pointer policy tests*

  Tests if pointer with given characteristics can actually reference particular object. Always tested at run-time.

Note

For example strong pointers must not point to remote objects.

In the following table there are described supported assignment operations. The table shows what operands can be used as arguments to pointer's operator=() and what checks (compile-time or run-time) are performed. Run-time errors are signalled to the application by raising exceptions (see Section 5.3.11, “Exceptions”) or asserts (program logic errors, static errors).

Table 5.2. Pointer assignment variants

Operand	Semantics	Exceptions
Pointer with the same characteristics	Copy pointer. Always succeeds.	
Pointer with a different characteristics	Copy and convert pointer. Allow implicit pointer conversions only. Assignment is validated at compile-time (upcast) and run-time (mutation, policy test).	ObjectNotOnLocalSystemException, PointerPolicyViolationException
Explicitly casted pointer	Copy and convert pointer. Allow any implicit pointer conversion and *checked downcast*. Assignment is validated at run-time (interface test, mutation, policy test).	IllegalPointerConversionException, ObjectNotOnLocalSystemException, PointerPolicyViolationException
`null` pseudo-keyword	Reset pointer to point to NULL_ID. Always succeeds.	
`ObjectId`	Create pointer to arbitrary object identified by its id. Assignment is validated at run-time (interface test, policy test).	IllegalPointerConversionException, ObjectNotOnLocalSystemException, PointerPolicyViolationException

Operand	**Semantics**	**Exceptions**
C++ pointer	Create pointer from C++ pointer to local managed addressable object. Assignment is validated at run-time (policy test). Allows to assign C++ *this* to managed pointers.	PointerPolicyViolationException

Note

Managed pointers implement the same set of constructors with similar semantics.

The use of the presented variants should be quite straightforward and is illustrated at the end of the section. Nevertheless there are some issues that need to be addressed before and will be explained now:

- *How to force an explicit cast*

  Managed pointers are not casted in a C++ way using its cast operators. A special construct is provided instead. It is semantically similar to C++ *dynamic_cast*<> (although the pointer type to cast to is determined automatically) and is forced by calling *convert()* method on the pointer that should be casted. See Section 5.3.3, “Assigning To Pointers” [48] example.

  Note

  convert() translates the pointer to a special form that will trigger the conversion when it is assigned to a pointer. The pointer type to cast to corresponds to the pointer type where the "converted pointer" is assigned to. convert() does not do any conversion until assigned as the target pointer type is unknown at the time of the call to convert().

  Warning

  Product of convert(), the special pointer form, can not be stored by application. It must be immediatelly assigned to a managed pointer so that the conversion would be triggered.

- *Implicit pointer conversions may fail*

  Unlike C++ pointer implicit conversions, implicit conversions of managed pointers can fail and throw exceptions. That is because of pointer policy testing (must be done at run-time) and implicit pointer mutation. We did not want application programmers to use explicit casts when converting between weak and strong pointers and between their stack-only and property pointer variants because the code would become unreadable. It is up to application logic to ensure that the actual operation would not fail or handle the potential exception properly.

  Note

  Implicit conversions of strong pointers to weak pointers will always succeed. See Section 5.3.10, “Pointer Policies”.

- *Assigning C++ this*

  Managed pointers accept C++ *this* and other C++ pointers. However the pointers must point to addressable objects or the assignment would fail. For example *this* pointer pointing to an object instantiated on the stack can not be assigned to a managed pointer.

An example:

```
class X : public Object { ... };
class Y : public X { ... };
class Z : public Y { Pointer< Z > method(); };
...
Pointer< Z > Z::method()
  {
  Pointer< X > ptr_x;
  Pointer< Y > ptr_y;
  Pointer< X > other_ptr_x;
  WeakPointer< X > weak_ptr_x;

  ptr_y = CreateObject< Y >(); ❶
  ptr_x = ptr_y; ❷
  other_ptr_x = ptr_x;
  weak_ptr_x = ptr_x; ❸
  //ptr_y = ptr_x; /* Compile-time error. Need explicit cast. */
  ptr_y = ptr_x.convert(); ❹
  ptr_x = null;

  return this;
  }
```

❶ This is how stand-alone objects are created. You should already be familiar with this construct.
❷ Implicit pointer conversion (upcast).
❸ Implicit pointer conversion (mutation).
❹ Explicit pointer conversion (downcast). Use method convert() to force an explicit cast of the pointer.

5.3.4. Comparing Pointers

Another important feature managed pointers support is the ability to compare them. Both comparing with managed pointers and C++ pointers have been implemented, however with different semantics. When comparing two managed pointers, `ObjectId`s of the referenced objects are compared only, *the other characteristics, mainly what components are referenced, are ignored*. When comparing a managed pointer with a C++ pointer, pointer to the referenced component (as returned by the *cast object*) is compared with the C++ pointer.

Note

The actual implementation does not follow the wording exactly, which allows for various optimizations. However the effect is the same.

Managed pointers overload operator==() and operator!=() so that they could be compared as C++ pointers. Since the results of comparisons do not depend on the order of their operands it is sufficient to summary all the valid operand combinations in the following table. The operations will always succeed.

Table 5.3. Pointer compare variants

Operand types	**Semantics**
Two managed pointers with arbitrary characteristics	`ObjectId`s are compared.
One managed pointer, one C++ pointer	C++ pointers are compared.
One managed pointer and a `null` pseudo-keyword.	Test if pointer points to `NULL_ID`.

Warning

Comparing pointers with `ObjectId` is illegal as well as casting pointers explicitly for the purposes of comparison (one of the operands is a product of convert() method). Managed pointers with different characteristics can be compared without a cast.

Warning

Managed pointers are compared by `ObjectId`s only. This can yield surprising results in multiple inheritance hierarchies.

For example if a class `Foo` was inherited multiple times to object *A*, the result of the comparison of `Pointer< Foo >` and C++ *this* pointing to the same object *A* would always be "not equal" as `Foo` would be ambiguous. However if *this* was converted to a managed pointer before and then compared with `Pointer< Foo >` the result would be "equal" as both the pointers would reference the same object.

In single inheritance hierarchies the results will be exactly the same as if C++ pointers were compared (the pointers either point to the unrelated class hierarchies, and then object ids are different, or to the same hierarchy and simple object id test is sufficient because of pointer subsumption).

5.3.5. Dereferencing Pointers

Managed pointers provide a secure way to dereference self and access components of referenced objects. When dereferenced, an active proxy [1] object, bound to the referenced object, is returned by value. Its type and implementation depends on pointer deference semantics. The proxy then delegates the access to either bound local object (and fails if there is none; proxy with *local dereference se-*

[1] Proxy objects serve as local gates to referenced objects. Any access to a referenced object is intercepted by a proxy object and can be translated to a remote procedure call, for example. These object proxies are called active as they are returned by value and they are deleted as soon as the object is no longer dereferenced (local proxy can be used to measure how long the object has been active or whether it is active now). This is transparent to the application.

mantics) or a local RPC stub that translates the access to a RPC call on the bound remote object (proxy with *remote dereference semantics*).

If a pointer has a local dereference semantics, the proxy tests if referenced component can be accessed locally (referenced object or its replica must be present on the local system and the component must not be ambiguous), pins the object to the local node (*object is active since then*) and gains full access to the object's component (as determined by pointer's cast object). When the component is no longer dereferenced the object is ready to be unpinned. Until unpinned, the object remains active and can not be deleted or can not migrate. If bound object is not present on the local node, nor its replica, the dereference will fail.

Proxy with remote semantics access methods described by IDL only. Stubs, generated for each RPC method, are responsible for translation of the local call to a RPC call. For information how RPC works see Chapter 8, *Remote Procedure Call*.

Pointers overload operator->() so that they could be dereferenced in usual C++ way. Attempts to dereference invalid pointers (point to `NULL_ID`, ambiguous or unsupported components, remote objects, etc.) are rejected and one of the following exceptions will be thrown:

- ObjectNotOnLocalSystemException
- IllegalPointerConversion

Note

The errors that could not be catched at assign time will probably be detected at dereference time. Remember that no interface test is performed if the type information of the referenced object is unavailable. However the object could migrate to the local node in the mean time and the interface test would be then performed at the time of the pointer dereference.

5.3.6. Special Operations

Unlike C++ pointers, managed pointers define additional *member methods* that can be used to query pointer state. This information can be used to test if actual pointer operation (for example an assignment to another pointer type, a pointer dereference, etc.) will succeed. Since some of the pointer functionality is turned off in release mode (see Section 5.3.13, “Differences In Debug And Release Mode”), incorrect use will either be not detected, program will simply crash or its state become inconsistent.

Warning

Programmer must be aware of the differences and must test in advance if certain operations could be processed safely.

The following table lists some of the most useful methods. For the complete list consult the Massiv Core Programmer's Documentation.

Table 5.4. Pointer member methods

Method	Semantics
is_local()	Test if the pointer can be dereferenced locally in the given *context* without an error. Must be rechecked whenever the context changes (the Core runs in between the two contexts, each context belongs to a different simulation tick) unless the pointer is strong. See also Section 17.1, “The Model Used By the Core”.
convert()	Force an explicit cast of the pointer.
get_object_id()	Get `ObjectId` of the referenced object.
inherits(const std::type_info &)	Test if the referenced object inherits from a specified class. Throws `UndefinedClassTypeException` if the object is remote and its dynamic type is unknown.

An example:

```
class MyClass : public Object { ... };
class MyDerivedClass : public MyClass { public: PInt32 my_integer; ... };
...
WeakPointer< MyClass > ptr = ...;
try
  {
  Pointer< MyDerivedClass > ptr_derived = ptr.convert(); ❶
  ptr_derived->my_integer = 2;
  }
catch( Throwable & )
  {
  }
```

❶ Pointer conversion may fail because of two reasons. Either ptr refers to a remote object (thus conversion to a strong pointer is rejected) or the referenced object does not inherit from `MyDerivedClass`.

The same program can be rewritten so that no exceptions would be raised:

```
WeakPointer< MyClass > ptr = ...;
if( ptr.is_local() && ptr.inherits( typeid( MyDerivedClass ) )
  {
  Pointer< MyDerivedClass > ptr_derived = ptr.convert();
  ptr_derived->my_integer = 2;
  }
```

5.3.7. Migration And Replication Groups

Pointer properties can be anotated by *pointer replication flags* that are used for implicit definitions of migration and replication groups of objects. The flags are set by IDL *ptr_repflags* property attribute.

Replication group of an object is defined as a maximal set of objects reachable from that object over bidirectional pointer properties whose pointer replication flags match a given mask. Replication flags masks are "yes-no" pairs that specify what bits of pointer replication flags must be set (yes bits) and what bits must not be set (no bits). Replication groups induced by a predefined fixed [`REPMASK_MIGRATE_YES`, `REPMASK_MIGRATE_NO`] pointer replication flags mask are called *migration groups of objects*. The mask is chosen so that *any replication group (induced by a different mask) is a subset of the corresponding migration group*. During the enumeration process pointer properties are treated as if they were bidirectional.

Replication groups are always defined with respect to a specified mask. There is no single replication group of an object. For example the replication group of an object for the purposes of its replication from a server to a server node (induced by one mask) may be different from the replication group of the same object for the purposes of its replication from a server to a client node (induced by another mask). The masks are set globally for all allowed replication destination node types. When an object should be replicated, the Core determines type of the replication (for example “replication to server”), looks up the replication mask related to the particular replication type and enumerates the replication group of the object using the obtained mask. Specified bit of the pointer replication flags triggers replication of the referenced and the owner object to the corresponding node type.

Note

For example when replicating a `Player` object to a server node, not only `Player` but also its inventory and other helper objects (part of `Player`'s migration group) would be replicated. However when replicating the same `Player` object to a client node, only the `Player` object and its inventory would be replicated. This is controlled by the setting of the pointer replication flags.

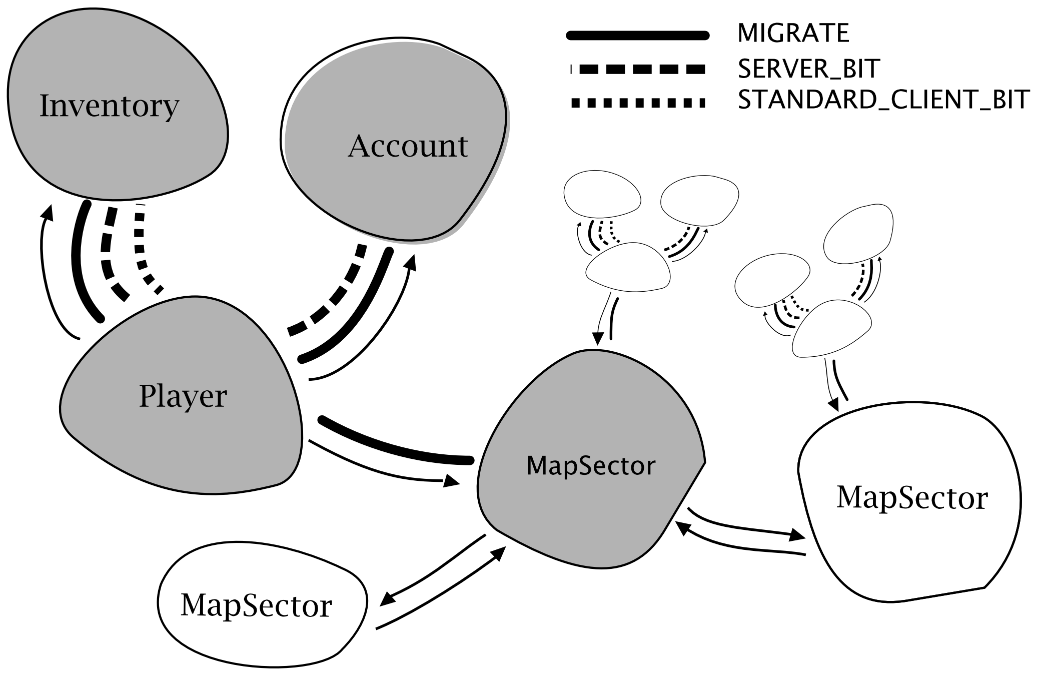
Migration groups are replication groups with a predefined `REPFLAGS_MIGRATE` pointer replication flags bit set. If this bit is on then the referenced object will always be (must be) present on the same node as the object that owns the pointer and the pointer will implicitly define a migration group of objects. It is also said that the pointer triggers the migration of the referenced object or the owner object if the other object migrates. *Such a pointer need not be strong. However the Core automatically sets this bit for strong pointer properties.*

Note

Pointer properties that define replication/migration groups are treated bidirectionally. This is needed to ensure that strong pointers always reference local objects, for example. Otherwise if B migrated to an another node, the strong pointer from A to B would be pointing to a remote object. Although this makes implementation of ObjectPointer and group enumeration more complex this proved to be very useful. For example `Player` object can reference its `Inventory` object by a single strong pointer and both migration groups of `Player` and `Inventory` are the same (`Inventory` need not reference `Player`). *Together with implicit replication/migration group definitions this is one of the key properties that make the Massiv unique.*

Confused? See the example:

```
class Inventory : public Object { ... }; /* All ptr_repflags = 0 */
class Account : public Object { ... };   /* All ptr_repflags = 0 */

class MapSector : public Object
  {
public:
  ...
  /* ptr_repflags = 0 */
  PRemote< MapSector > neighbor[ 8 ];
  };

class Player : public Object
  {
public:
  ...
  /* ptr_repflags = MIGRATE | SERVER_BIT | STANDARD_CLIENT_BIT */
  PPointer< Inventory > inventory;

  /* ptr_repflags = MIGRATE | SERVER_BIT */
  PPointer< Account >   account;

  /* ptr_repflags = MIGRATE */
  PPointer< MapSector > sector;
  };
...
Pointer< Player > player = ...;
Pointer< Inventory > inventory = ...;
Pointer< Account > account = ...;
Pointer< MapSector > sector = ...;

player->inventory = inventory;
player->account = account;
player->sector = sector;
```



Suppose that we have named pointer replication flags bits so that `MIGRATE` triggers object migration (defines a migration group), `SERVER_BIT` triggers replication to server nodes and `STANDARD_CLIENT_BIT` replication to standard client nodes. Inventory, Account and MapSector objects contain weak pointers with ptr_repflags = 0 only, then:

- Corresponding Player, Inventory, Account and MapSector objects form the same migration group. If more players are linked to the same sector all the Players (and their Accounts and Inventories) will implicitly belong to the same (single) migration group. On the other side, different MapSector objects belong to different migration groups.
- Replication group of the Player enumerated due to replication to a server node will contain the Player object and its Account and Inventory objects. The MapSector and the other Player, Account and Inventory objects (the rest of the migration group) will not be replicated to server nodes.
- Replication group of the Player enumerated due to replication to a standard client node will contain only the Player object and its Inventory object.

Note

In the example we were assigning strong pointers to the Player object which caused the merging of the migration groups resulting in the creation of a single migration group containing all the involved objects. First, the migration group of the Inventory object and the Player object was merged, then the migration group of the Account object was joined, etc. Since we were working with strong pointers all the groups had been local. If a remote migration group should have been merged, our Player object would have to migrate to the remote group (taking its current migration group with self) so that the join could have been finished on the remote node. See Section 6.5, “Requesting Migration”.

Note

Right now you might be confused. How do I assign `ptr_repflags` to a pointer? What values can be assignmed to `ptr_repflags`? The answer is: `ptr_repflags` is an *attribute* of pointer properties, it's value is specified in IDL. For more information see Section 7.5, “Replication-related IDL Attributes” and Section 10.8, “Attributes”.

5.3.8. Pointers To Forward Declared Classes

When creating recursive data structures, one has to use pointers to forward declared classes in order to avoid cyclic header inclusion. Pointer implementation allows to declare managed pointers to forward declared classes. However at the time of the pointer use (for example dereference), the referenced class must have already been seen by the compiler. This requires to organize source code in such a way that headers contain class declarations only and the full definition, including method bodies, is provided in `.cpp` files.

Header file:

```
...
class A;
class B;

class A : public Object
  {
public:
  void set( Pointer< B > ptr );
  PPointer< B > ptr_b;
  ...
  };

class B : public Object
  {
public:
  void set( Pointer< A > ptr );
  PPointer< A > ptr_a;
  ...
  };
```

CPP file:

```
...
void A::set( Pointer< B > ptr )
  {
  ptr_b = ptr;
  }

void B::set( Pointer< A > ptr )
  {
  ptr_a = ptr;
  }
```

Note

Classes must be forward-declared in the IDL too. See Section 10.9.1, “Forward Declaration”.

5.3.9. Pointer Replicas, Pointers To Object Replicas

Pointer property semantics changes if the property is actually a replica. In that case the object that owns the property is also a replica. This means that pointer's *static* characteristics will be partially ignored. In particular *all pointer property replicas are considered weak* regardless to their static characteristics. That is because not replicated portions of migration groups, thus remote, can be referenced from the replicated parts by originally strong pointers.

Note

In the example above the `Player` object replicated to a standard client node would have its pointers to `Account` and `MapSector` pointing to the original (remote) objects and the objects would not be replicated to that node.

Warning

This has a consequence that the conversion of `PPointer< Object >` to `Pointer< Object >` can fail if the property is a pointer replica since we are effectively converting a weak pointer property to a strong stack pointer.

The other issue is how object replicas can be accessed by other non-replica pointers (either regular stack pointers or non-replica pointer properties). Let us note that object replicas have the same object id as the original objects and are accessible by pointers with local dereference semantics only. There are never the original object and its replica present on the same node at the same time (for more information see Section 7.4, “Accessing Object Replicas”). Obviously *strong pointer properties can not reference object replicas* because the corresponding original objects are remote. Also *strong stack pointers can not reference object replicas* (such references would prevent the Core from updating strong-referenced replicas and could block incoming migrations too, see Section 7.10, “Allowed and Illegal Operations”). However *they can be referenced by weak pointers of any other characteristics*.

Warning

Calling a SRPC from within a replica (if an object replica is active) is forbidden as it would prevent the Core from updating the replica.

5.3.10. Pointer Policies

Pointer policies state what pointers, with given characteristics, can reference particular objects and what the Core guarantees if such a pointer exists. The following object and pointer properties are investigated:

- What is the dynamic type of the object?
- Is the object local or remote?
- If the object is local is it a replica?
- Is the pointer a replica?

Although most of the policies have already been mentioned in the previous sections they have been summarized in the following lists and tables.

Policies applicable to both stack pointers and pointer properties:

- *Object's dynamic type is utilized to validate pointer operations. Can not create pointers to objects that do not support requested interfaces, can not dereference pointers to ambiguous interfaces*

  The violation is signalled to the application by raising `IllegalPointerConversionException` exception, either at assign time (this is the interface test already described at Section 5.3.3, “Assigning To Pointers”) or dereference time.

- *Strong pointers must reference local objects only*

Assignment of a remote object to a strong pointer results in `ObjectNotOnLocalSystemException` exception. The *Core guarantees that if the assignment succeeded at some time, the referenced component was unambiguous and the referenced object was not explicitly deleted by user in the meantime, then the pointer dereference at a later time would always succeed*. The concept of migration groups ensures that this policy is not violated when objects migrate.

In particular *objects referenced by strong-stack pointers can not migrate until the pointers are released*. If an object A migrates all the objects that reference A by strong pointer properties or are referenced from A by strong pointer properties will migrate together with the object.

Note

This feature is crucial. It allows to access objects without worrying if the objects are still local.

```
class X : public Object { .... void f(); };
class Y : public Object { .... void g(); };
...
Pointer< X > strong_ptr = ...;
WeakPointer< X > weak_ptr = ...;
Remote< Y > remote_ptr = ...;

strong_ptr->f();

if( weak_ptr.is_local() )
  {
  weak_ptr->f();
  ...
  remote_ptr->sync_g(); ❶
  ...
  strong_ptr->f();
  weak_ptr->f(); ❷
  }
```

❶ This performs a SRPC call on a remote object. The Core runs meanwhile and the object referenced by weak_ptr can migrate out of the local node.

❷ The dereference may fail as the referenced object can no longer be local. See threading model Section 17.2, “When the Core Runs” to get information on when the Core can run.

The above example could have been rewritten so that the racing conditions would have been avoided:

```
if( weak_ptr.is_local() )
  {
  Pointer< X > ptr = weak_ptr; ❶
  ...
  remote_ptr->sync_g();
  ...
  strong_ptr->f();
  ptr->f();
  }
```

❶ This will fail if weak_ptr points to an object replica. See the following policy.

- *Strong pointers can not reference object replicas*

  Object replicas can be referenced by weak pointers only . This applies to stack pointers too. If strong stack pointers to object replicas were allowed the Core would have to freeze updates of the referenced replicas in order to ensure consistency and fulfil the requirement that the replicas would not be deleted while still strong-stack-referenced. Also incoming migrations turning object replicas to regular non-replica objects would have to be blocked. In the overall the system would be stalled.

Warning

When using weak pointers with local dereference semantics one has to be careful when the pointers are dereferenced. For example if a previous dereference succeeded, a later dereference may fail if the Core ran in between the two dereferences. This is the semantics of weak pointers and a neccessary feature that allows to implement an effecient replication model. For more information see threading (execution) model Section 17.3, "What Happens When the Core Runs".

- *Pointers to objects allocated on the stack or member objects owned by other objects can not be created*

  These objects are not addressable because they do not have an `ObjectId`.

The following policies are applicable to pointer properties only. They ensure migration groups consistency. The policies are violated due to program logic errors only and they are signalled to the application through a generic `PointerPolicyViolationException` exception:

- *Pointer properties with `REPFLAGS_MIGRATE` pointer replication flags bit set can not reference remote objects*

  Such pointers define migration groups of objects. Strong pointer properties are this kind of pointers.

- *Pointer properties with `REPFLAGS_MIGRATE` pointer replication flags bit set can not point to object replicas*

  The original objects are remote and have the same object ids as their replicas.

- *Pointer properties with `REPFLAGS_MIGRATE` pointer replication flags bit set can not be used to link archivable and non-archivable migration groups*

  If this was allowed we would get a single migration group of objects containing both archivable and non-archivable objects.

- *Pointer property replicas are always weak*

  Static pointer characteristics is overloaded if the pointer is a replica.

Most of the policies take effect if doing implicit pointer conversions when pointer characteristics changes (pointer mutation). The behavior is summarized in the following table. It can be seen that *conversions to weak pointers always succeed (the plus signs) and the other conversions fail if the referenced object is remote or an object replica* :

Note

The following table ommits several kinds of pointers. It is because a mutation to them will always succeed. They are: `WeakPointer< X >`, `Remote< X >`, `PWeakPointer< X >` and `PRemote< X >`.

Table 5.5. Pointer mutation

From\To	**Pointer< X >**	**PPointer< X >**
Pointer< X >	+	+
WeakPointer< X >	Fails if points to a remote object or a replica	Fails if points to a remote object or a replica
Remote< X >	Fails if points to a remote object or a replica	Fails if points to a remote object or a replica
PPointer< X >	May fail if from pointer replica pointing to a remote object or will fail if points to a replica	Will fail if from pointer replica pointing to a remote object or an object replica
PWeakPointer< X >	Fails if points to a remote object or a replica	Fails if points to a remote object or a replica
PRemote< X >	Fails if points to a remote object or a replica	Fails if points to a remote object or a replica

There is always a way to check if certain operation is correct so that the policy violation exception could be avoided. The check can be done on the application level by using these methods:

- *`local_pointer`*.is_local()

  Tests if the pointer with local dereference semantics can be dereferenced. See Section 5.3.6, “Special Operations”.

- *`local_pointer_property`*.is_replica()

  Tests if the pointer property is a replica.

- *`local_pointer`*->is_replica()

  Tests if the pointer points to an object replica (assuming that the pointer can be dereferenced locally).

- *`local_pointer`*.inherits(...)

  Tests if the referenced object implements a requested interface.

5.3.11. Exceptions

Pointer subsystem throws managed exceptions if an incorrect use of a pointer is detected. If an exception is thrown, the state of the involved pointer is rollbacked to a previous consistent state. The following table describes the exceptions from the point of view of the pointer subsystem, the exceptions can be used by other subsystems as well:

Table 5.6. Exceptions thrown by the pointer subsystem

Exception	Why	When
CanNotAccessObjectException	Operation on an object could not be processed.	Dereferencing a `NULL_ID` pointer.
IllegalPointerConversionException	Object does not implement the requested interface, or the interface is ambiguous. See Section 5.3.2, “Using Cast Object”.	Assigning or dereferencing pointers.
ObjectNotOnLocalSystemException	Trying to operate on a remote object.	Dereferencing a pointer with local dereference semantics pointing to a remote object. Assigning a remote object to a strong pointer.
PointerPolicyViolationException	Pointer property specific policy was violated.	Assigning a pointer.
RemoteCallFailedException	Attempt to issue a remote procedure call failed.	Processing a RPC request.
UndefinedClassTypeException	Object's dynamic type is unknown.	Requesting a type information of a remote object.

5.3.12. Differences From C++ Pointers

This chapter summarizes the basic differences between C++ pointers and managed pointers. As could have been seen, managed pointers try to imitate C++ pointers as authentically as possible, but also add non-trivial extensions such as type and reference safety or implicit migration and replication group definition. Some of the features already supported by C++ pointers are implemented by managed pointers differently (or not at all) and that's why this chapter is present:

- Pointer operations automatically throw exceptions if an illegal operation is detected
- To force an explicit conversion, when an implicit conversion is not available, use convert() method. IllegalPointerConversionException will be thrown if type information on the involved object is available and the conversion is illegal. Type information might not be available if the object is remote and of unsupported class kind (see Section 4.3.7, “Class Kinds”). In that case no test is performed! The potential error should be detected when the invalid pointer is used to invoke a re-

mote call on that object.

- Use `null` pseudo-keyword to assign `NULL_ID` values to a pointer or compare the pointer with `NULL_ID`
- Pointers to non-addressable objects (allocated on the stack, member objects) can not be created

5.3.13. Differences In Debug And Release Mode

Full pointer policy testing can be quite an expensive task. However most of the policies that are checked at run-time are intended to catch program logic errors. This means that the policy violation is often systematic and once the program has been debugged and it has been verified that the policies are no longer violated, the checks can be turned off and gain significant program speed up.

If the application is compiled in *debug mode* all the tests that can be performed locally are turned on. This includes full pointer policy testing. When compiled in *release mode* the following tests are turned off or are different:

- *No null or ambiguous pointer dereference test*

  An attempt to dereference such a pointer is functionally equivalent to the dereference of a C++ NULL pointer.

- *Limited pointer policy testing*

  The checking of the policies whose violation would be signalled by a generic `PointerPolicy-ViolationException` exception is turned off. These policies ensure consistent use of pointers with respect to the definition of migration groups and the use of object replicas.

Warning

Application programmer must be aware of the differences. The program is considered ill formed if it violates the policies that are checked in debug mode only. Programmer can check the policy on the application level if needed.

5.3.14. Advanced Techniques

Managed pointers can be combined with C++ references pointing to object members if extra attention is given. These techniques are considered *advanced* as one could get along with managed pointers only. For example in Java you can not create a reference to an object member.

Warning

In Massiv it is okay to create and pass short-time aliasing C++ references pointing to local members to other local methods as far as the object that owns the referenced members is (and will remain) active. Such references must not be stored (kept) by the callee. Similarly, C++

references to data allocated on the stack can be passed to local methods if they are released when the callee exits. Simple standard scoping rules will help you to understand this.

```
void print( const Property & p )
  {
  std::cout << p;
  }

void MyObject::method()
  { ❶
  std::cout << *this;
  print( this->my_property ); ❷
  }

/*
const Property & MyObject::get_my_property() const
  {
  return this->my_property; ❸
  }
*/
```

❶ Suppose that MyObject instance has been called through a managed pointer. MyObject instance has already been pinned when program got to this point. Application can now manipulate with MyObject instance as usually.

❷ Pass my_property to print() by C++ reference, print() must forget the reference at its exit.

❸ Wrong! Can not export a C++ reference to an object that won't be active any more.

5.4. Garbage Collector

All stand-alone managed objects are garbage collected. This means that the application does not manage memory allocation and object's lifetime by itself but it is done automatically by the Core. Programmer simply creates objects by a new()-like construct and lets the Core delete them when they are no longer used. The Core periodically scans for such unused objects, basically the objects that can not be reached by strong pointers, and deletes them. The Garbage Collector is not conservative, during the mark phase, when the collector is marking "reachable" objects, meta-information of examined objects is used to determine what references a particular object holds. The Garbage Collector activity is fully transparent and does not require user's assistance.

5.4.1. The Model

This section describes the garbage collector model implemented in the Core.

Basic single-process garbage collectors usually work as follows. When the underlying run-time determines that there is low free memory, a new GC run is triggered. Its purpose is to find and delete "lost" objects, the objects that can not be reached from program stack and global variables through pointers. During this process all threads are stopped so that the GC sees consistent memory image. The implementation is often conservative and does not utilize any meta-information (everything is treated as if it could be a pointer). This has the consequence that the GC is not able to detect all unreachable objects, might be ineffecient or blocks the program for quite a long time (GC runs do not remain unspotted).

In a distributed environment the problem is much more complex and conventional algorithms used by local garbage collectors can not be used for various reasons. Also the GC runs may be quite lengthy and threads can not be blocked during the process. Because of this most distributed systems do not have a garbage collector at all, require a non-trival user's assistance or are too conservative (GC can not tell if an object is unreachable, may cause serious memory leaks).

In the Massiv a special form of a local garbage collector is used. There are no global variables, the whole simulation state is kept in managed objects. Instead of determining what objects are reachable from global variables (which would not have sence because objects can transparently migrate whereas global variables can not), the garbage collector determines what objects are reachable from special *GC root objects*. An object can be a GC root object either statically (if its class has an IDL gc_root flag set) or dynamically by promoting it to a root object. Objects with scheduled migrations are automatically promoted to GC roots until the migration takes place. Apart from GC roots, the scanning process also originates in pointers residing on the program stack. During the scanning, *the Garbage Collector is interested in strong pointers only* . Weak pointers are ignored and this is consistent with other GC-based systems that support weak references.

Note

To conclude, the scanning process originates in GC root objects and stack-strong-referenced objects. Strong pointers keep the referenced objects alive. One might wonder where the local semantics of the GC comes from. Remember that strong pointers are the only pointers that are processed by the GC when it scans for reachable objects and that strong pointers can not point to remote objects.

When the Core starts a new GC run. It first scans for reachable objects from GC roots and the program stack (the mark phase) and then deletes the unreachable objects (sweep phase). Only stand-alone non-replica objects are garbage collected.

Note

Active objects are not garbage collected too, but they are not considered to be GC roots. Their migration groups won't also be collected if the active objects are referenced by chains of strong pointers originating on the program stack.

5.4.2. GC Roots

GC root objects are stand-alone objects handled by the Garbage Collector in a special way. Such objects are somewhat privileged over non root objects because they will never be collected by the Garbage Collector unless it was explicitly instructed to do so. Moreover the scanning for reachable objects originates in these objects. This has a consequence that objects reachable from GC root objects by strong pointers will not be collected too. In other words *GC root objects make other objects alive*. For completeness it is worth saying that stack strong pointers have a similar function as the scanning originates also in stack-strong-referenced objects.

The Garbage Collector semantics can be expressed in the language of migration groups as well.

- Migration groups are deleted at the next GC run if they do not have GC root objects and are not referenced from the stack by strong pointers.
- Weak-referenced portions of migration groups (objects referenced by weak pointers only) are also deleted.

There are two kinds of GC root objects:

- *Permanent GC roots*

  A stand-alone object is a permanent GC root if its class defines gc_root = true IDL class attribute. All stand-alone instances of such a class are GC roots.

  Permanent GC root objects can be turned to non-root objects at the end of its life. This ensures that the migration group will be collected then. Use System::dispose_gc_root().

- *Dynamic GC roots*

  A stand-alone object instance can be promoted to a GC root at the run-time. The promotion is on the per-instance basis.

  Object instances are automatically promoted to GC roots if they have pending migrations. They are automatically demoted back to non-root objects when the migrations finish. This allows to form and migrate a migration group without a permanent GC root object. Such migration groups can be used to implement messaging:

```
class Message;

class Sender : public Object /* gc_root = true */
  {
public:
  void send
    (
    WeakPointer< Receiver > receiver,
    Pointer< Message >      message,
    const STime &           delivery_time
    );
  ...
  };

class Receiver : public Object /* gc_root = true */
  {
public:
  void message_delivered
    (
    Pointer< Message > message
    );
  ...
  PPointer< Message > last_message;
  };

class Message : public Object /* gc_root = false */
  {
```

```
protected: /// Object interface.
  void delivered_to
    (
    WeakPointer< Object > object,
    const STime &         delivery_time
    );
  ...
  };

void Sender::send
  (
  WeakPointer< Receiver > receiver,
  Pointer< Message >      message,
  const STime &           delivery_time
  )
  {
  message->migrate_to( receiver, delivery_time ); ❶
  }

void Message::delivered_to
  (
  WeakPointer< Object > object,
  const STime &         delivery_time
  )
  { ❷
  Pointer< Receiver > receiver = object.convert();
  receiver->message_delivered( this );
  }

void Receiver::message_delivered
  (
  Pointer< Message > message
  )
  {
  last_message = message; ❸
  }
```

❶ Requesting a migration promotes the message to a dynamic GC root object.

❷ Message object was delivered to the migration addressee. This is called by the Core when the migration finishes. The object has already been demoted back to a non GC root object, however it is stack-strong referenced from the Core until the exit from the method.

❸ If message was not assigned to last_message the Message object would be garbage collected at the next GC run.

5.4.3. The API

This section describes public and semi-public APIs to the Garbage Collector. Although the application mostly should not care, as the Garbage Collector works automatically and need not be controlled by the application at all, the knowledge of the API can be advantageous. However when working with the GC directly non-trivial knowledge related to the Core implementation is required. Reading the Massiv Core Programmer's Documentation is highly recommended.

The Garbage Collector can be controlled either through `System` class, which defines a public API that abstracts all Core subsystems, or directly through `GarbageCollector` class. The direct access is semi-public and should be used for diagnostics and debugging purposes only.

Garbage collector related API provided by the `System` class is summarized in the following table. It discusses how to explicitly force garbage collection, explicitly delete an object or dispose a migration group containing a GC root object:

Table 5.7. Public API to the Garbage Collector

Method	Description
System::force_gc()	Forces immediate garbage collection. Although the GC decides itself when to perform the collection a way to explicitly force the collection might be useful.
System::dispose_gc_root(local_pointer)	Demotes a permanent GC root object referenced by the local_pointer to a non-root object. The object and its migration group will eventually be garbage collected at once.
System::collect_object(local_pointer)	Instructs the GC to collect the object referenced by the local_pointer as soon as possible. This method allows to delete objects explitly. However rest of the migration group remains alive at least to the next GC run.

System::collect_object() can not always delete referenced objects immediatelly. That is because such objects may be active, for example. If this is the case, the object is tagged and will be deleted at the next GC tick:

```
class X : public Object
  {
public:
  void f()
    {
    Pointer< Object > self = this;
    System::collect_object( self );
    }
  ...
  }
```

There is no need to worry about unspotted dangling pointer dereferences. The delete operation equals to an immediate object migration (without the rest of its migration group, of course) to a "thrash can". Any attempt to access such an object will fail because the object is no longer local and can not be localized on remote nodes (ObjectIds are not recycled). Thus, *pointers with both local and remote dereference semantics pointing to the deleted objects will always fail to dereference*. Pointer validity can be tested in the common way.

Warning

When deleting an object explicitly keep in mind that other objects from the same migration group will remain alive and their strong pointers to the deleted object will be invalidated (fail to dereference, but won't be reset to `NULL_ID`). The same steps must be taken as if the application was written in plain C++. However invalid dereferences will be catched.

Suppose that we have a migration group with a single GC root object. If the root object was explicitly deleted by System::collect_object() there would be a risk that the rest of the migration group would be alive until the next GC run. This may cause the problems explained in the previous warning. However *if the root object was disposed by System::dispose_gc_root() the migration group would be deleted atomically at a future GC run*. In the case of multiple GC roots all the roots will have to be disposed.

Note

This is valid under a condition that basicaly none of the objects in the migration group is active at the time of the collect. The condition can simply be fulfiled by preventing the GC from running if there might be active objects. This is the default behavior and can be changed via the registry settings.

Direct API to the Garbage Collector is provided by the `GarbageCollector` global object. The API is semi-public and gains access to the features ranging from diagnostics and debugging functions, statistics, settings, the Garbage Collector state to migration and replication group enumeration. The complete documentation can be found in the Massiv Core Programmer's Documentation and requires the knowledge of the Core internals. Some of the features are explained in the following section.

5.4.4. Running And Configuring GC

Garbage Collector runs are triggered either automatically by the Core logic, when an object limit count (adapted by recent memory use) is reached, or explicitly by calling `System::force_gc()`. There is a variety of settings that can be used to setup the Garbage Collector. See Section 27.7, “Garbage Collector”.

6. Migration

6.1. Overview

The *migration* is the process when one or more managed objects are moved from one node to another one, either transparently or on demand.

Object migration is the only directly supported way of data exchange between objects and object interaction. Migrations are addressed by objects. This is basically an agent model, where agents (objects) migrate in order to accomplish their tasks. Once delivered to an object, a callback is called. Everything else (RPC and messaging, for example) can be built on top of the object migration.

From previous chapters you should already know:

- What managed objects exist and how they can be grouped together into migration groups (see Section 5.3.7, "Migration And Replication Groups").
- How the simulation is distributed over several nodes and how these nodes are accessible to managed objects via node objects (see Section 2.1, "Types Of Nodes" and Chapter 14, *Special Objects*).

6.2. Communication Between Objects Using Migration

When we were designing how the managed objects will communicate while the objects are distributed among several nodes and thus can't access each other directly, there were several possibilities how to implement communication between the remote objects. We found both asynchronous and synchronous RPC useful and worth implementing but too complex for the lowest level of communication among objects.

We knew that we wanted something similar to asynchronous message sending. But the protocol based on asynchronous messages itself was not suitable for us as it would introduce two types of objects - managed objects and message objects. But it was just one step from the final solution which is implemented in the Massiv.

In the Massiv, managed objects and messages are the same. Managed objects can behave like messages and messages can behave like objects. What does that mean? Because each managed object is also a message, it can be sent to another objects. And because each message is also an object, it can have properties and methods which can be accessed and invoked.

The migration is the process when one or more messages (objects) are delivered (migrated) to another object. Here is the right point to make clear the difference between delivery and migration - there is no difference at all. When we are looking on some object as it is a message, the message is "delivered" to its recipient object. Actually, the object is migrated to the same node where the recipient resides and the object (message) is received by the recipient. You should have no problem understanding when “delivery” is interchanged with “migration” (and vice versa) without being explicitly stated.

The described model where objects and messages are the same, has some nice features.

- Because messages are also managed objects, we don't need to take extra care of the messages. For example, the messages are automatically archived as any other object is.
- Messages can contain any managed data.
- Messages can expose some specific behavior because the the messages are totally under the control of the user of the Core. For example, some messages don't only need to be delivered to some object and then destroyed, but they can be delivered to the object and then they can return to the sender of the message (or they can be sent again to any other object). This feature is heavily used in implementation of synchronous RPC where the RPC messages are used to bring back results of the RPC invocation.
- There doesn't need to be some global handler which will pass the message to its recipient once the message is delivered. The handler is part of the message object - when the message is delivered to its recipient, one of its methods is invoked with recipient as its argument. The method is *virtual* so each message can have its own handler which can do whatever is appropriate for the message. For example the recipient of the message doesn't need to notice at all that the message was delivered to him if the message's handler doesn't call any method of the recipient.

6.3. Migration Types

The Core provides two types of migration:

- *Migration between nodes*

  Migration between nodes is the proces where the Core moves a group of objects from node *A* to another node *B*. The objects are first destroyed on the node *A*, then transferred over the network to the node *B* on which they are finally recreated to their state before the migration.

  By default this migration is transparent to the objects because on the node *B* the objects are recreated to the same state as on the node *A* before their destruction. Sometimes, mostly for debuging purposes, it is usefull to know when an object migrates. See Section 6.8, “Detecting Migrations Between Nodes” for more information about this problem.

- *Migration between objects*

  This type of migration is also called *delivery* of an object to another object. The destination of the migration is some object in the simulation. The migrated object is moved to the node where the destination object exists and delivery callback is invoked on the migrated object with the destination object passed as an argument. This type of migration is the basic principle on which the entire communication among remote objects (objects which can't access directly each other) is built upon.

The two types of migrations are presented only for clarity because user of the Core doesn't need to distinguish between them. There is one common interface how to request migration of an object to either another object or to some node in the simulation (nodes can be addressed on object level too, see Section 14.2, “Node Object”). Also the migration between objects is an extension of the migration between nodes because the migrated object can travel among several nodes before the destination object is found in the simulation.

6.4. Migration Groups

Managed objects are grouped into migration groups, as you should already know from Section 5.3.7, “Migration And Replication Groups”. For simplicity you can think of migration group as a maximal set of objects where each object from the group is reachable via strong pointer properties by at least one another object from the group, and any object outside the group is not reachable by any object from the group via strong pointer properties.

In other words, if object *A* has strong pointer to object *B*, both of the objects belong to the same migration group. The important thing is that it is ensured that if an object is located on a node, all objects from its migration group are located on the same node as well and the objects can access all other objects from the migration group directly by dereferencing the strong pointer properties.

Because objects from one migration group must always reside on the same node, all of the objects must migrate together whenever one of them have to migrate to another node.

All objects from one migration group always migrate together. Whenever one object from the migration group migrates, all other objects from the migration group migrate as well together with the object.

6.5. Requesting Migration

There are three methods of `Object` which can be used to request delivery (migration) of the object to another object: `migrate_to()`, `schedule_to()` and `deliver_asap_to()`.

```
void Object::migrate_to
    (
    WeakPointer< Object > destination_object,
    const STime &         delivery_time,
    bool                  notify = true
    );

EventHandle Object::schedule_to
    (
    WeakPointer< Object > destination_object,
    const STime &         delivery_time,
    bool                  notify = true
    );

void Object::deliver_asap_to
    (
    WeakPointer< Object > destination_object,
    bool                  can_deliver_to_replica = false
    );
```

The *`destination_object`* specifies the object to which the migrated object should be delivered.

The *`delivery_time`* is the time when the migrated object should be delivered to the destination object. Note that that the Core doesn't ensure that the object will be delivered exactly at the specified time to the destination object. The *`delivery_time`* specifies only when the migration will begin because there is no way how to detect how much time the migration will take since the destination object must be first found in the simulation and also the network transmission time is unpredictable. The migrated object will be delivered exactly at the specified time only if the destination object is on the same node as the migrated object (in such case the migrated object is not migrated at all). The variable duration of the migration should not be a problem because once the migrated object is delivered, the Core tells him the original *`delivery_time`* and so the object can easily compute how much time the migration took and do whatever is appropriate.

The *`notify`* specifies whether the migrated object should be notified when it is delivered to the destination object. If it is *true*, the Core calls `delivered_to()` (or `delivery_failed()` in case the delivery failed) on the migrated object once it is delivered to the destination object. If the *`notify`* is *false*, the migrated object is not notified of the finished migration (neither `delivered_to()` nor `delivery_failed()` is called).

In most cases you will request deliveries with notifications (the *`notify`* set to *true*, which is the default) since the delivered objects will often do something with the destination objects (call their methods, access their properties etc).

Migrations without notifications are useful when you want to migrate an object without letting it to know. In such case the migration to the destination object just means moving the object to the node where the destination object resides. This principle is used by the Core to distribute managed objects among server nodes when performing load balancing (Appendix B, *Load Balancing*).

The `migrate_to()` tells the object to migrate at specified time to the *`destination_object`*.

The `deliver_asap_to()` tells the object to migrate to the *`destination_object`* as soon as possible. The major difference from general `migrate_to` is that the object is delivered immediatelly if *`destination_object`* is a local object instead of scheduling the migration so that it could be processed at next simulation tick (if possible). If *`destination_object`* is a local object, migration callbacks (see Section 6.6, “Migration Callbacks”) will already have been executed by the time when `deliver_asap_to()` returns. If the object is not local a migration to the *`destination_object`* is scheduled to current simulation time and the method behaves like `migrate_to()`.

The *`can_deliver_to_replica`* determines whether the object can be delivered to a consistent local replica of the *`destination_object`*. If a migration is scheduled the object is always delivered to the original object.

`schedule_to()` does the same what `migrate_to()` does with the only difference: the methods returns a “handle to the scheduled migration” (the handle is called `EventHandle` because scheduled migrations are implemented as events of the objects but it is nothing you should worry about). The returned handle can be used to cancel the migration. Note that the migration can be cancelled only if you have both the handle to the migration and strong pointer to the migrated object (i.e. the object is local). In other cases the result of the cancellation is undefined. The migration may but also may not be cancelled.

Corresponding property for the `EventHandle` is `PEventHandle`. To cancel the migration call `kill()` method on the handle:

```
void EventHandle::kill();
void PEventHandle::kill();
```

Note that there is also performace difference between `migrate_to()` and `schedule_to()`. The latter adds extra data to the object which must be transferred over the network when the object is migrated. If you don't need to cancel migrations you should always use the `migrate_to()` method.

Warning

The Core is not able to migrate active or pinned objects. The methods used to request object migrations either deliver the objects directly to local objects (`deliver_asap_to()`) or *schedule the migrations* so that they could be processed later (possibly at next system tick) when the objects are not active any more. In particular the migration will still be pending when the program returns from `migrate_to()`.

If an object requests its migration and then immediatelly issues a SRPC call the object will not be able to migrate at least until the SRPC completes. The object will be blocked and may block other objects too - object migrations initiated on the same node are processed by the Core in the order of their delivery times. Thus if an object `A` schedules a migration to be processed at time `T1` and an object `B` schedules a migration to time `T2` > `T1`, and *both the objects are located on the same node*, `B` will not be able to migrate until `A` migrates (`B` will be blocked by `A`).

It is highly recommended not to touch objects with pending migrations. When their migrations complete corresponding callbacks will be upcalled (see Section 6.6, “Migration Callbacks”).

6.6. Migration Callbacks

Once an object is delivered to its recipient, it is notified by `delivered_to()` or by `delivery_failed()` in case the delivery failed.

```
virtual void Object::delivered_to
    (
    WeakPointer< Object > destination_object,
    const STime &         delivery_time
    );

virtual void Object::delivery_failed
    (
    DeliveryFailure     failure,
    const STime &       delivery_time
    );
```

The `delivered_to()` method is called if the migrated object is successfully delivered to the *`destination_object`*.

The `delivery_failed()` method is called if the migration failed for some reason.

The *`destination_object`* and the *`delivery_time`* are the arguments passed to the functions `migrate_to()`, `schedule_to()` and `deliver_asap_to()` when a migration is requested.

If the migration has been successful, the Core ensures that the *`destination_object`* pointer points to a local object (object located on the same node) and thus it can be accessed directly by dereferencing the pointer. The pointer is weak and thus deliveries to local object replicas by `deliver_asap_to()` are possible (see Section 5.3.10, "Pointer Policies"). However it is ensured that if `destination_object` does not point to a replica it is internally strong-referenced from the Core. It is also ensured that `this` object is strong-referenced too.

If the migration has failed, the *`failure`* holds the failure reason.

Table 6.1. Migration failure reasons

Value	**The migration failed because...**
`DESTINATION_NOT_ACCESSIBLE`	The destination object is not accessible in the simulation. Either the object was already destroyed or the object didn't exist at all.
`EVENT_KILLED`	The migration was cancelled. See description of `schedule_to()` mthod in Section 6.5, "Requesting Migration".

6.7. Preventing Migrations

It is possible to prevent migrations of all objects of a class. If the class is declared with *no_migrate* attribute set to `true` (see Section 10.9.3, "Class Attributes"), all instances of the class won't be allowed to migrate. If you create an instance of the class, the object will always reside on the node on which it was created. If you will try to migrate these objects, the Core will log a warning message and discard the migration.

6.8. Detecting Migrations Between Nodes

By default the migration between nodes is transparent to the objects and you should never need to detect it because there can be many reasons why an object is migrated to some node (not only because the object or another one requested it) and because the Core will never tell you the exact reason. The Core provides a generic callback, which is called whenever an object is recreated on a node. The name of the callback is `object_updated()`. The callback has a single *`reason`* parameter that, for the purposes of object migration, is always set to `INSTANCE_MIGRATED`. `INSTANCE_MIGRATED` can't be used to detect migration between nodes, because the Core uses this *`reason`* also when the object is loaded from the swap file or restored from the archive file. In most cases an application programmer should not care and if the callback is used at all it should be used for debugging purposes only.

6.9. Migration And Garbage Collector

Objects with scheduled migrations are never automatically destroyed by the Garbage Collector even if they are not permanent GC root objects. Objects with scheduled migrations are promoted to GC root objects. See Section 5.4, “Garbage Collector” for more information about GC root objects.

6.10. Migration And Nodes

As you already know, the Massiv divides all nodes into three groups: *client* nodes, *simulation servers* and *data service* nodes. Because data service nodes do not participate in the simulation, they can never be destination of migrations and thus will not be mentioned in the following sections.

6.10.1. Migration Between Server Nodes

Migration among server nodes is the the basic principle the entire communication among remote objects is built upon. There is no restriction on this type of migration because server nodes are always trusted so you can migrate any object to any server node at any time.

6.10.2. Migration From Client To Server Nodes

Because server nodes can't trust any data from client nodes, there are many restrictions on object migrations from client to server nodes:

- Only one object can migrate from a client node to a server node at once i.e. any migration group sent from client nodes to some server node must consist of only one object. If a client node sends two or more objects as a part of one migration message, the server node discards the message away and the client is disconnected from the simulation.
- Only objects of explicitly selected classes can migrate from client to server nodes. When you create a new class, you can specify in its *IDL* description whether the class is allowed to migrate from client to server nodes (see *client_server_migrate* attribute in Section 10.9.3, “Class Attributes”). If the client node sends an object which is not allowed to migrate from client to server nodes, the server node discards the message and the client is disconnected from the simulation.
- The destination of the object which is being sent from a client node to a server node must be an account object of the client (see Chapter 14, *Special Objects*). Its up to account's logic to process the object and optionally forward the message to another objects in the simulation. Again, if the client is trying to communicate directly with an object different from its account object, it is disconnected from the simulation.
- Of course the data itself of the object which is being sent from the client node to the server node is verified as well.

Once the object is successfully verified by the server, new *ObjectId* is generated for the object because its current *ObjectId*, generated by the client, is not persistent and can't be used as a valid *ObjectId* inside the simulation. Having new *ObjectId*, the object can be finally delivered to the account object of the client.

6.10.3. Migration From Server To Client Nodes

Similarly to migration from client to server nodes, also the migration from server to client nodes have some specifics. First of all, there are no restrictions on which and how many objects can migrate to client nodes. But you should have in mind that the client nodes don't support all classes supported by the Core on server nodes but only a subset.

Because object owned by client nodes are not part of the simulation and the client nodes can't be trusted, once an object is migrated from servers to a client node, it can't ever return back in the simulation. The server nodes also loose track of the object and don't monitor the object any more.

6.11. How Migrations Are Used By The Core

6.11.1. Load Balancing

The Massiv provides facilities for automatic load balancing to decrease network and CPU load (see Appendix B, *Load Balancing*). The Core uses transparent migrations to migrate objects to less loaded nodes in order to keep uniform resource loads.

6.11.2. Synchronous And Asynchronous RPC

The Massiv provides remote procedure call (RPC) mechanism that allows you to call methods of remote objects (see Chapter 8, *Remote Procedure Call*). The RPC is implemented by migrating `RPCObject` between the caller and the callee object.

7. Replication

7.1. Overview

The replication is used to keep read-only copies of managed objects on one or more nodes. These copies are called *object replicas*, or simply *replicas*. The Core automatically keeps object replicas up-to-date during their lifetime.

Replication serves two main purposes:

- To keep up-to-date copies of objects needed for world presentation on client nodes. Server nodes decide which objects are relevant for the presentation (i.e. visible and/or audible) and request their replication to client nodes.
- To optimize communication between objects, especially to reduce the communcation latency. If server node-side code wants to read properties (or call constant methods) of objects potentially owned by a different server node, it can request replication of such objects in advance and access object replicas instead of accessing the objects themselves over the network (using *synchronous RPC* for example).

On a single node, either object itself or its replica can exist, but never both of them. Replicas have the same object id as the original object and can be accessed using a weak pointer to the object (see Section 7.4, "Accessing Object Replicas").

7.2. Replication Model

Each managed object keeps list of nodes it should be replicated to, and for each node the simulation time when replication of the object to the node should be stopped. When the Core decides to update replicas on a node, it gathers set of objects that should be replicated to the node, determines all objects belonging to their *replication groups*, and sends the *replication update* to the node.

Note

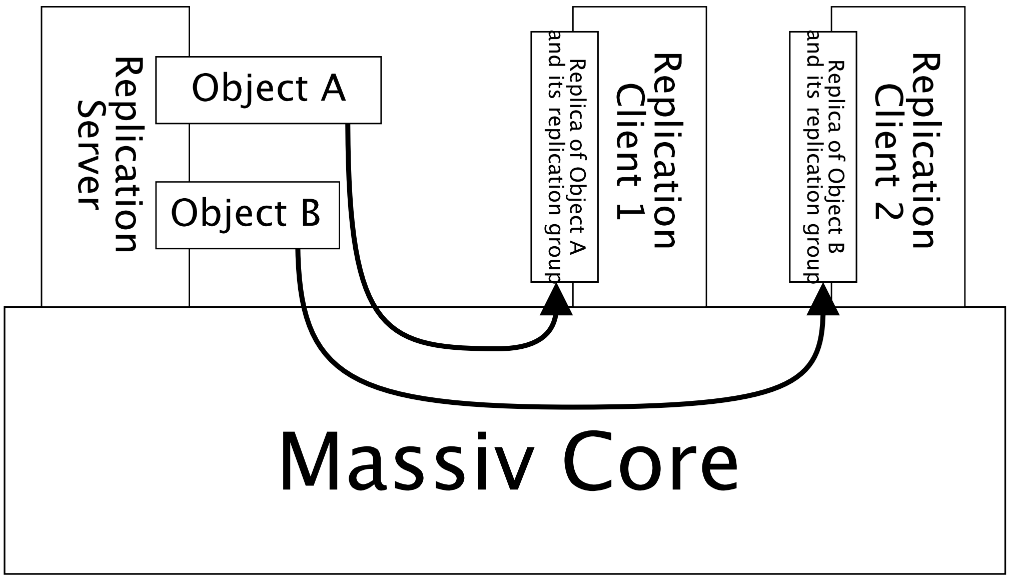
Replication of replicas is illegal.

In this chapter, when talking about a given object and its replication, *replication server* refers to the node that owns the object, and *replication client* refers to each of the nodes the object is replicated to. Somethimes this may be shortened to *server* and *client*. When talking about different types of nodes, as described in Section 2.1, "Types Of Nodes", *server node* and *client node* terms will be always used.

7.2.1. Simple Example

The Figure 7.1, "Replication Example" shows a simple situation, where *Replication Server* owns two objects, *Object A* and *Object B*, and replicates them to *Replication Client 1* and *Replication Client 2*, respectively. The Core keeps the copies of the objects and their replication groups on replication clients, until the *replication timeout* simulation time is reached.

Figure 7.1. Replication Example


Replication Server
Object A
Object B
Replica of Object A and its replication group
Replication Client 1
Replica of Object B and its replication group
Replication Client 2
Massiv Core

The figure does not indicate whether *Object A* and *Object B* belong to the same replication group or not. If they belonged to the same group, the same set of objects would be replicated to both replication client nodes, even if the replication was “triggered” by a different object.

7.2.2. Replication from Client Nodes

Only server nodes are allowed to replicate their objects. Any client node that tries to replicate an object to a server node will be disconnected from the simulation. There are two reasons for this behavior:

- Replication from client nodes to server nodes seems useless. None of the two scenarios where replication may help (world presentation and communcation optimizations) applies to client nodes.
- Replication from client nodes to server nodes would be dangerous and insecure. It would allow client nodes to spam server nodes with tons of useless objects.

7.2.3. Replication and Migration

When an object, that is replicated to several nodes, migrates, the new owner node becomes responsible for replication of the object. All replication requests, along with their timeouts, will migrate with the object.

Object remembers replication request even when the replication client node of the request is the same as the owner node of the object. Such requests are perfectly legal. No replicas on the owner node will be created, but if the object migrates to a different node, the new owner node will automatically replicate the object back to the former owner node.

Because client nodes are not allowed to replicate their object, when an object migrates to a client node, all replication requests of the object are forgotten and its replicas are destroyed.

7.2.4. Replication and Class Kinds

You should already know what *class kinds* mean. If not, refer to Section 4.3.7, “Class Kinds”.

It is allowed to replicate objects of any kind to a node of the same type, but when replicating to a node of different type, all replicated objects must be of `KIND_SHARED`.

7.3. Requesting Replication

This chapter describes how to request replication of an object and how to cancel a replication request.

7.3.1. Using Methods of `Object`

To request replication of an object to given node, call `replicate_to()` method of `Object`. To cancel replication of an object, call `cancel_replication`:

```
void Object::replicate_to
    (
    WeakPointer< Object > destination_node,
    const STime &         timeout
    );

void Object::cancel_replication
    (
    WeakPointer< Object > destination_node
    );
```

The *`destination_node`* argument must point to a node object (see Section 14.2, “Node Object”). This is an important difference from migration - while migration is addressed by destination *object*, replication request is addressed by destination *node*.

The *`timeout`* argument specifies simulation time when replication of the object will be stopped and the object replica will be destroyed. The replication timeout will be set to this value, even if replication request of the object to given node exists, and its replication timeout is larger. Setting the timeout to the past (default-constructed value of `STime` is always in the past) will effectively cancel the replication. Actually the `cancel_replication` is just a properly named shortcut for `replicate_to` with such *`timeout`*.

7.3.2. Node Object Pointers

Pointers in general are described in Chapter 5, *Pointers*, and node objects are described in Section 14.2, “Node Object”. This section just repeats the most useful information regarding object replication.

Pointer to a node object can be easily constructed from a node id:

```
NodeId node_id = ...;
WeakPointer< Object > node_object( ObjectId( node_id ) );
```

To get node id of the local node, call:

```
const NodeId local_node_id = System::get_local_node_id();
```

Because `replicate_to()` can be called remotely (see Chapter 8, *Remote Procedure Call*), it's easy to request replication of a remote object `foo` to local node:

```
Remote< Object > foo = ...;
foo->async_replicate_to( ObjectId( System::get_local_node_id() ), STime( 60.0 ) );
```

Obtaining pointer to node object of a client node is also relatively easy. When client connects to the simulation, it calls method `client_connected()` of `AccountObjectInterface` and passes pointer to its node object as the only argument:

```
virtual void AccountObjectInterface::client_connected
    (
    const WeakPointer< NodeObjectInterface > & node_object
    ) = 0;
```

Refer to Section 14.3, “Account Object” for information about account objects. You should also check implementation of account object in Massiv Demo (files `src/demo/lib/shared/account_object.*`).

7.3.3. Requesting Replication by RPC

Remote procedure call to a object can automatically trigger its replication to callee node. This is described in Section 8.5.1, “Triggering Replication by RPC”.

7.4. Accessing Object Replicas

Replicas of objects have the same object id as the objects they are replicas of. On every node, exactly one of the following statements is true for object of given object id:

- The object is local.
- The object is remote and its replica exists on the node.
- Neither the object itself nor its replica exists on the node. The object is either remote or does not exist at all.

As described in Chapter 5, *Pointers*, objects pointed to by a strong pointer are always local. However, weak pointer may reference a remote object. If the object is remote, and its replica exists on local node, weak pointer can be used as if the object was local, and it will access the replica instead of the object.

The following example shows how to determine whether an object is local, replica of the object is local, or the object is remote. Moreover, if a replica of the object exists, it determines whether it is *consistent* or not (the replica consistency is described in Section 7.7, “Consistency of Replication Groups”):

Example 7.1. Determining State of an Object

```
WeakPointer< Object > foo = ...;
if( foo.is_local() )
    {
    if( foo->is_replica() )
        {
        if( foo->is_consistent() )
            {
            /* Consistent replica of foo is local. */
            }
        else
            {
            /* Inconsistent replica of foo is local. */
            }
        }
    else
        {
        /* foo is a local object. */
        }
    }
else
    {
    /* foo is a remote object, or does not exist at all. */
    }
```

The Core restricts operations on replicas; especially it's illegal to modify properties of replicas. For complete list of allowed and illegal operations, refer to Section 7.10, “Allowed and Illegal Operations”

7.5. Replication-related IDL Attributes

The IDL description allows you to specify many attributes of classes, properties and methods. This sections describes attributes related to replication. The syntax of attribute definition is documented in Section 10.8, “Attributes”, complete lists of class, property and method attributes are in Section 10.9.3, “Class Attributes”, Section 10.10.1, “Property Attributes” and Section 10.11.1, “Method Attributes”, respectively. If you are not familiar with the IDL concept at all, you should first read Chapter 9, *Introduction to IDL*.

Two replication-related property attributes, commonly called *replication flags*, exist: `repflags` and `ptr_repflags`. They are used to choose which objects belong to a replication (or migration) group and which properties of those objects should be replicated, based on *type* of destination node. Both `repflags` and `ptr_repflags` are of enumeration type *ReplicationFlags*.

The following sections assume that standard values of the *ReplicationFlags* attribute type, as defined in `src/core/object/object.idl`, are used. However, the Core allows you to define your own replication flag constants and per-node type replication masks as explained in Section 7.5.5, “Custom Replication Flags and Masks”.

7.5.1. Node Types

As you already know, the Massiv divides all nodes into three groups: *client* nodes, *simulation servers* and *data service* nodes. Because data service nodes do not participate in the simulation, they can never be destination of replications or migrations, and will not be mentioned in the following sections.

Default replication flag constants distinguish two client node types: *standard client* and *privileged client*. The Core will support this distinction in the future, but currently it treats all client nodes as equal. When using recommended settings (see Section 7.5.4, “Recommended Node Replication Masks”), all client nodes are treated as *privileged* for replication purposes.

Note

This distinction is used only for replication purposes when choosing which properties should be replicated and which objects belong to a replication group, it does not add any special privileges to client nodes. You can still easily implement per-account privileges as can be seen in the Massiv Demo - only players with *root* accounts may directly edit the terrain and entities.

7.5.2. Choosing Properties To Replicate

Property attribute `repflags` allows you to choose which properties of a managed object should be serialized when replicating the object to a node of given type. It can be assigned one of the values defined in Table 7.1, “Property replication flags”. The default value of the `repflags` attribute is `REPLICATE` - all properties are replicated to all node types by default.

Table 7.1. Property replication flags

Value	Bits set	Meaning
`NONE`	*none*	Illegal value. `MIGRATE` bit of `repflags` must be always set.
`MIGRATE`	`MIGRATE`	Do not replicate, only migrate.
`SERVER`	`MIGRATE | SERVER_BIT`	Replicate only to simulation servers.
`PRIVILEGED_CLIENT`[a]	`MIGRATE | PRIVILEGED_CLIENT_BIT`	Replicate only to privileged clients.
`STANDARD_CLIENT`[a]	`MIGRATE | STANDARD_CLIENT_BIT`	Replicate only to standard (non-privileged) clients.
`PRIVILEGED`[b]	`SERVER | PRIVILEGED_CLIENT`	Replicate to simulation servers and privileged client nodes.
`CLIENT`	`PRIVILEGED_CLIENT |`	Replicate to client nodes (both standard

Value	Bits set	Meaning
	`STANDARD_CLIENT`	and privileged).
`REPLICATE`	`SERVER | CLIENT`	Replicate to all node types.

[a] Because the Core does not distinguish standard and privileged client nodes yet, this `repflags` value should not be used. Use `CLIENT` instead.
[b] Because the Core does not distinguish standard and privileged client nodes yet, this `repflags` value should not be used at all.

Replication flags are specified for each property which is member of a managed object. If the property is a container property, its replication flags apply to both the container and the properties it holds.

Note

When replication protocol decides if a property should be replicated, it checks only the value of its `repflags` attribute, even if the property is a managed object or a container. For example, an array property with `SERVER` `repflags` would not be replicated to client nodes at all, even if it contained an object with properties with `REPLICATE` `repflags`.

7.5.3. Replication and Migration Groups

The `ptr_repflags` property attribute of pointer properties is used to define replication and migration groups. Possible values are described in Table 7.2, “Pointer replication flags”. The default value of the `ptr_repflags` attribute is `NONE` for weak pointers and `MIGRATE` for strong pointers - weak pointers do not define a group by default, and strong pointers define a migration group by default.

Migration and replication groups in general are documented in Section 5.3.7, “Migration And Replication Groups”.

Table 7.2. Pointer replication flags

Value [a]	Defines a group when... [b]
`NONE`	pointer does not define a group
`MIGRATE`	migrating
`SERVER`	migrating, or replicating to a simulation server
`PRIVILEGED_CLIENT`[c]	migrating, or replicating to a privileged client
`STANDARD_CLIENT`[c]	migrating, or replicating to a standard non-privileged client
`PRIVILEGED`[d]	migrating, or replicating to a simulation server or a privileged client
`CLIENT`	migrating, or replicating to a client node
`REPLICATE`	migrating or replicating

[a] Bit values are exactly the same as in Table 7.1, “Property replication flags”.
[b] Because replication groups are subsets of migration groups, all pointers with `ptr_repflags` set to anything but `NONE` define a migration group.
[c] Because the Core does not distinguish standard and privileged client nodes yet, this `ptr_repflags` value should not be

used. Use CLIENT instead.

[d] Because the Core does not distinguish standard and privileged client nodes yet, this ptr_repflags value should not be used at all.

Warning

Strong pointers must always define at least a migration group. The IDL processor will automatically change default value (NONE) of ptr_repflags of simple strong pointer properties to REPLICATE (actually, value of another property attribute, strongptr_repflags, is used, but you should never override its default). However, if a strong pointer is stored inside a structural property (such as a pair or a container), ptr_repflags of the property must be set manually in the IDL.

Pointers contained within member objects and container properties of an object define migration/replication groups too. If a pointer is member property of a managed object, ptr_repflags specified in IDL of the corresponding class are used. If a pointer is a member of a non-object container property, ptr_repflags of the container are used. This means that it's impossible to mix pointers with different ptr_repflags in container properties (such as the pair, for example).

Note

It's possible for pointers to define a replication group even if their value is not replicated, if repflags is *more restrictive* than ptr_repflags. This feature is probably not very useful.

Warning

Be careful when mixing objects of different kind (described in Section 4.3.7, “Class Kinds”) in a group. Migration of object of unsupported kind will crash destination node, replication of object of unsupported kind *and all objects replicated along the object* will be ignored by destination node.

7.5.4. Recommended Node Replication Masks

When interpreting values of repflags and ptr_repflags attributes, the Core compares them with *replication masks* for given node type.

The Core does not define any replication flags except the MIGRATE bit, other bits are defined in the IDL. Values described in previous sections document recommended flag values defined in src/core/object/object.idl and the Core does not know their values. Therefore default replication masks are pretty useless: when they are used, all properties are always replicated, and all pointers with the MIGRATE ptr_repflags bit set define both migration and replication group, no matter what the destination node type is.

To use flags defined in src/core/object/object.idl with meaning described in previous sections, override mask settings on each node to these values:

```
[ Settings/ObjectManagement/Replication ]
server_repflags_no : integer = 0
server_repflags_yes : integer = 3  ❶
```

```
client_repflags_no : integer = 0
client_repflags_yes : integer = 5  ❷
```

❶ These masks match iff `MIGRATE` and `SERVER_BIT` are set.

❷ These masks match iff `MIGRATE` and `PRIVILEGED_CLIENT_BIT` are set, i.e. all clients are *privileged*.

In the Massiv Demo, these settings are stored in a separate file, `src/demo/config/common/massiv.replication.conf`.

7.5.5. Custom Replication Flags and Masks

As already mentioned, the Core does not force you to use replication flag values defined in `object.idl`. To define custom flags, you must first understand how the Core interprets the flags.

Let's begin with few simple definitions:

- *Replication flags* - an integer. Each bit of the integer corresponds to a flag. The Core defines single flag, `MIGRATE`, as 0x01.
- *Replication mask* - a pair of replication flags, called *yes*-mask and *no*-mask.

It is said that replication flags *match* a replication mask, if all bits set in the *yes*-mask are set in the *flags* too, and no bits set in the *no*-mask are set in the *flags*. In other words, the *yes*-mask specifies which bits of the *flags* must be set in, and the *no*-mask specifies which bits of the *flags* must not be set.

Different replication masks are used for migration and replication, as described in Table 7.3, “Replication masks”. The Core uses the mask to enumerate migration or replication group - pointers with `ptr_repflags` attribute *matching* the mask define a group. When the Core migrates or replicates an object, only properties with `repflags` attribute that *matches* the mask are serialized.

Table 7.3. Replication masks

Operation	Mask source	*Yes*-mask	*No*-mask
Migration	constant	`MIGRATE`	0
Replication to server node	registry [a]	server_repflags_yes	server_repflags_no
Replication to client node	registry [a]	client_repflags_yes	client_repflags_no

[a] Replication-related settings are stored in the `Settings/ObjectManagement/Replication` registry node.

To define custom replication flags, you must:

- Override values of the *ReplicationFlags* enumeration attribute type. There is no easy way to do this, because the attribute type is defined in `object.idl`, which is part of the Core sources and is included by all IDL files. Currently the file must be modified directly.

- Choose replicate-to-server and replicate-to-client masks and set their values on all nodes.

It's not recommended to override default replication flag values. It's even not very useful, because current version of Massiv uses only type of destination node to choose replication masks. This may change in the future.

7.6. Replication and Migration Compared

This section tries to point out that the replication is *not* the same thing as copy-and-migrate. It may seem that the Core simply duplicates objects that have to be replicated and then migrates them to replication clients. The following table lists the biggest (and most obvious) differences between the copy-and-migrate protocol that could be implemented on top of the Core, and the replication:

Table 7.4. Differences between the copy-and-migrate and the replication

copy-and-migrate	**replication**
It's impossible to easily duplicate the whole migration/replication group.	Complete replication groups are replicated.
Duplication creates new objects with new object ids.	An object and its replicas have the same object ids.
Migration transfers object once. There is no easy way to update the migrated object.	Replication continuously updates replicas until replication request times out, and then destroys them.
Migration transfers all properties of the migrated object.	Replication allows you to select which properties should be replicated to a given type of node. Replica updates are very efficient, especially if changes to replicated objects are not frequent, or only some properties of the objects are modified.

7.7. Consistency of Replication Groups

Replicas belonging to the same replication group on a replication client node are said to be *consistent*, if all of them have the same "age", i.e. all of them have been updated at the same time (it's assumed that replication updates contain complete groups). Replication protocol tries as hard as possible to keep replication groups in the consistent state. However, this is not always possible. Replicas may sometimes become inconsistent if a replication group is split into two or more parts, part of the group migrates to another server and a replication update of one of the parts is received before an update of other parts.

Example of an inconsistency: Assume there is a replication group consisting of two objects, `A` and `B`, on server node *Server1*. There is a pointer from `B` to `A` which defines the replication group. The client node *Client* has replicas of the replication group. Then:

1. The pointer from `B` to `A` is changed and the replication group is split into two groups on *Server1*.
2. The object `B` migrates to the server node *Server2*. The object `A` stays on the server *Server1*.
3. In the next replication update, *Client* receives an update of the object `A` from the server *Server1*.

In this case we say that the replica of the object `B` is in an inconsistent state on the *Client*, because it's in an older state than object `A`, has a pointer defining a replication group to `A`, but the objects should no longer be in the same replication group.

Note

`A` and `B` may also be groups of objects of course.

Inconsistencies can cause nasty problems. When working with inconsistent replicas, nothing can be assumed about objects the replicas reference. In this case, the pointer references a more recent version of the object replicas (which can be fatal if the objects mentioned are part of a dynamic data structure such as list). Pointers of inconsistent replicas could also reference non-local or non-existing objects.

This can't be prevented on replication servers and can't be fixed by replication client. The replication client can only detect this situation, and report it to the programmer.

For the programmer only one thing is important - some replicas may be marked as inconsistent for a while. How this can be detected is shown in Section 7.4, “Accessing Object Replicas”. If a replica is inconsistent, you can't trust its replication-group-defining pointers. They may point to another inconsistent replica, to a replica in a more recent state, to a local object (obviously in a more recent state too), or to a non-existing object.

Any automatic or semi-automatic optimizations, performed by the Core on replicas, require replicas to be consistent. If they are not, the optimization will not take place. For example, the *replica-optimized const SRPC* will not be optimized if the replica is inconsistent and the regular SRPC will be performed instead.

7.8. Callbacks to Replicas

Whenever the Core makes “important” change to an object, it calls its `object_updated()` virtual method. This method has single argument *`reason`*. Replication protocols will call this callback method with *`reason`* set to one of the following values:

Table 7.5. Replication-related `object_update()` reasons

Reason value [a]	Used when
`REPLICA_CREATED`	Signalled after a new instance of object replica has been created and initialized. Properties of the replica are still invalid at this time. Both `REPLICA_UPDATE` and `REPLICA_UPDATE` object changes will be signalled after `REPLICA_CREATED`.

Reason value [a]	**Used when**
REPLICA_UPDATE	Signalled before update of replica properties.
REPLICA_CREATED	Signalled after update of replica properties.
REPLICA_DESTROY	Signalled before destruction of a replica.

[a] The type of the `reason` argument is enumeration *UpdateReason* defined in class `Object`. Therefore fully-qualified name of the `REPLICA_CREATED` (for example) is `Massiv::Core::Object::REPLICA_CREATED`.

Replication protocols ensure that replication update always contains up-to-date versions of all objects belonging to a replication group. If multiple objects are created by the update, `REPLICA_CREATED` callback will be signalled on all new objects as they are instantiated. All objects will be created before contents of any object is updated.

If multiple objects are destroyed by an update, `REPLICA_DESTROY` callback will be signalled on all objects-to-destroy before any object is actually destroyed.

`REPLICA_UPDATE` and `REPLICA_UDPATED` callbacks are signalled as objects are updated. Therefore you should not assume anything about validity of pointers and about objects belonging to the replication group of the updated objects when handling these callbacks.

> **Note**
>
> Refer to Massiv Core Reference Guide for more information about all available object callbacks.

7.9. Replica Manager

Callbacks described in the previous section may be useful, but they have many drawbacks. Especially before-update and after-update callbacks are a bit useless, because they are signalled during replication group update and other objects belonging to the replication group of the updated object can't be safely accessed.

Another problem is that the object itself must handle the callbacks. This may be a problem when you want to monitor changes to replicas of world objects on a client node, and modify their visual representation, for example. Because code of such objects is shared between client and server nodes, modifications to client-side structures may be pretty nasty.

The *replica manager* singleton object tries to solve all these problems. It allows you to register callback functions that will be invoked when objects with given object id or given class are modified.

Standard usage of `ReplicaManager` is shown in Example 7.2, “Usage of replica manager”. For complete documentation of the `ReplicaManager` class, see the Massic Core Reference Guide. What's important about the callbacks:

- `REPLICA_UPDATED` callback is invoked for each updated replica, after contents of all replicas from an update have been modified. This means that at the time of this call, all objects belonging to the replication group of the updated object are already up-to-date.

- `REPLICA_GROUP_UPDATED` callback is invoked for each replica that has not been modified by the update, but that belongs to a replication group of a replica that has been modified. `REPLICA_GROUP_UDPATED` callbacks are invoked after `REPLICA_UPDATED` callbacks.
- `REPLICA_DESTROY` callback is invoked before any replica is destroyed or updated. All objects belonging to the replication group of the object-to-destroy still exist. The callback is invoked whenever replicas are about to be destroyed. This includes handling of standard replica update messages, handling of an incoming migration, node reinitialization and node shutdown.

In the Massiv Demo, `ReplicaManager` is extensively used by all *game manager* classes, their sources can be found in directory `src/demo/client/game`.

Example 7.2. Usage of replica manager

```
using namespace Massiv::Core;

/* A structure. */
struct MyStruct;

/* A managed object. */
class MyObject;

/* The callback function. */
void my_callback
    (
    ReplicaManager::UpdateReason  reason,
    WeakPointer< Object >         replica,
    int                           user_tag,
    VariantPointer                user_data
    )
    {
    /* Cast replica pointer to correct type. */
    WeakPointer< MyObject > my_replica( replica.convert() );

    /* Cast user data to correct type. */
    MyStruct * const my_struct( variant_cast< MyStruct * >( user_data ) );

    switch( reason )
        {
        case ReplicaManager::REPLICA_UPDATED:
            /* Contents of my_replica have been updated. */
            ...
            break;

        case ReplicaManager::REPLICA_GROUP_UPDATED:
            /* An object belonging to replication group */
            /* of my_replica has been updated. */
            ...
            break;

        case ReplicaManager::REPLICA_DESTROY:
            /* Replica my_replica is about to be destroyed. */
            ...
            break;
        }
    }

...

int my_tag = ...;
MyStruct my_struct;

/* Register callback monitoring changes to all objects of class MyObject. */
ReplicaManager & replica_manager = Global::replica_manager();
replica_manager.register_callback
    (
    typeid( MyObject ),
    &my_callback,
    my_tag,
    VariantPointer( &my_struct )
    );

/* Unregister all registered callbacks to the my_callback method. */
replica_manager.unregister_callback( &my_callback );
```

7.10. Allowed and Illegal Operations

Working with replicas is not completely foolproof. In the debugging build, the Core tries to check all operations with replicas and throws an exception whenever programmer tries to perform an illegal operation. Some of these tests are expensive and are not done in the release build.

The following operations are illegal:

- Referencing replicas from strong pointers, unless the pointer is owned by a replica.
- Writing to properties of a replica.
- Reading unitialized properties of a replica (properties that are not replicated because of the value of their `repflags` property attribute are unitialized).
- Using synchronous RPC from method called on a replica.

The following operations are potentially dangerous:

- Calling a non-const method of a replica. It will probably write to the replica.
- Calling a const method of a replica. It may still write to properties of other objects in replication group of the replica, but you should probably not mark such methods as const. The method may try to construct strong pointer to the replica (`this`), or perform another illegal operation. Const methods may try to affect the simulation in any way - create and destroy other objects, migrate them, etc. It's recommended to add `hope( !is_replica() );` to all const methods that can't be called on replicas, and document that.

- Using synchronous RPC while working with replicas, because their state may change in any way during the call. They may be updated (which may be a problem if you iterate over a replicated structure), become inconsistent, or even be destroyed by the Core. There is no way to *pin* a replica, and such operation won't be implemented in the future versions of the Massiv, because it could stall both replication and migration protocols.
- Working with inconsistent replicas, obviously.

The following operations are safe:

- Replica-optimized SRPC (see Section 8.5.2, “Synchronous RPC Optimizations”). The call will not be performed locally if the replica of the callee object is inconsistent.
- Calling method `deliver_asap_to()` of an `Object` with parameter *`can_deliver_to_replica`* set to `true`. It will never deliver the object to an inconsistent replica.

Warning

The lists above are not exhaustive.

7.11. Massiv Demo Examples

Check the Massiv Demo for some real-world examples of replication usage. On server side, replica-optimized SRPC calls are used, for example in `src/demo/lib/server/robot.cpp`. Manual replication is used too. For example, in `src/demo/lib/server/sector.cpp`, `Sector::get_tile_elevation()` accepts neighbor-sector tile indices. If the neighbor sector is local or a replica of the neighbor exists, correct elevation is returned. Otherwise, elevation is estimated from current sector, and replication of the neighbor is requested.

There is one interesting feature in server-to-client replications. All presentation-related data of entities and sectors are stored in special classes. For example, class `Entity` has strong pointer to class `ClientEntity`, which contains all presenation data of an entity. This separation is actually required because of class kinds. `ClientEntity` is `KIND_SHARED` (i.e. known to both client and server nodes), while `Entity` is `KIND_SERVER` (i.e. its definition and implementation is unknown to client nodes). If presentation data were stored directly in `Entity`, it would have to be `KIND_SHARED`. That would mean that all classes used by `Entity` would have to be `KIND_SHARED` too, but that would include nearly all classes of server lib.

Because of this, the `repflags` attribute is not used in demo at all. Pointers from `Entity` to `ClientEntity` have `ptr_repflags` set to `SERVER` - the objects are in the same migration group and in the same replication group when replicating an entity to a server node. But when a `ClientEntity` object is replicated to a client node, `Entity` will not be in its replication group.

The client use the *replica manager* to monitor changes to all the `Client*` objects. Source of replica managers can be found in directory `src/demo/client/game`.

7.12. Configuration and Statistics

Configuration of replication protocols is documented in Section 27.15, “Replication”. You should not have to modify the default settings, except replication masks, as explained in Section 7.5.5, “Custom Replication Flags and Masks” and Section 7.5.4, “Recommended Node Replication Masks”.

Statistics regarding outgoing replication are stored in subnodes of the `/Statistics/Replication/Server` registry node, and statistics about incoming replication are stored in subnodes of the `/Statistics/Replication/Client` registry node.

8. Remote Procedure Call

8.1. Overview

This chapter describes various flavors of the remote procedure call (RPC) mechanisms provided by the Massiv. RPC allows you to call methods of managed objects, even if they are not owned by local node, using syntax nearly identical to that of local calls.

To be able to call a method, it must be described in the IDL of its class. If you are not acquainted with the IDL concept yet, refer to Chapter 9, *Introduction to IDL*. In all examples in this chapter, it's assumed that a class `Foo` with the following IDL description exists:

Example 8.1. RPC Example - IDL for class `Foo`

```
class Foo : Massiv::Core::Object
    {
    method< const > bar( out int32 result, in int32 param ) : bool;
    method baz( out int32 result, in int32 param ) : bool;
    };
```

Both methods, `bar` and `baz`, take single *in* argument (`param`) and return boolean type and single *out* argument (`result`). Method `bar` is constant.

Moreover, it's assumed that a valid remote pointer to object of type `Foo`, `foo`, exists. To be able to use *Remote* pointer to class `Foo` defined in the IDL description contained in `foo.idl`, one must include the generated header `foo_rpc.h`:

```
#include "foo_rpc.h"

Remote< Foo > foo = ...;
```

To see how to obtain such pointer, check Chapter 5, *Pointers*.

8.2. RPC Model

To make a RPC request, dereference the *Remote* pointer to get a stub object. Stub objects are automatically generated from the IDL and serve for two purposes:

- To change parameters affecting the way the call will be performed, and
- To perform the call.

Methods that set options for a remote call are universal - they are the same for each stub object. All of them return a reference to the stub object, which allows you to set multiple options and perform the call in single statement. The methods will be introduced in the following sections, for a complete list check Section 8.8, “RPC Reference Guide”

For each method defined in the IDL, two methods are implemented by the stub:

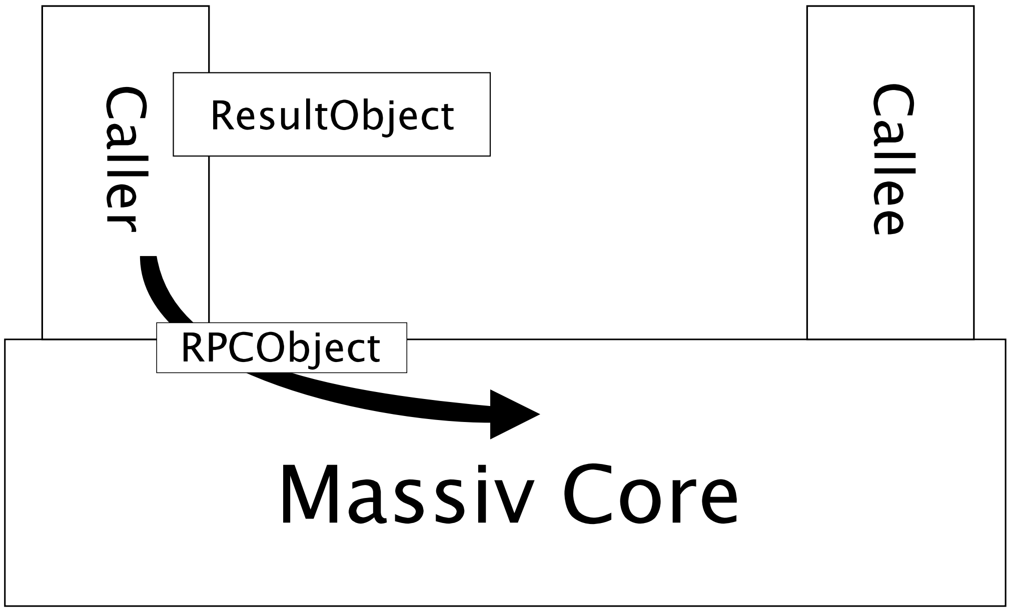
- `async_bar()` performs asynchronous remote call (see Section 8.3, “Asynchronous RPC”).
- `sync_bar()` performs synchronous remote call (see Section 8.4, “Synchronous RPC”).

The RPC is implemented on top of the migration protocols. Standard localization protocols are used when sending RPC requests and results. Both requests and results will be archived properly when remote call is pending. Check Chapter 6, *Migration* to understand potential restrictions and problems.

Execution of the standard remote call can be divided into the following steps:

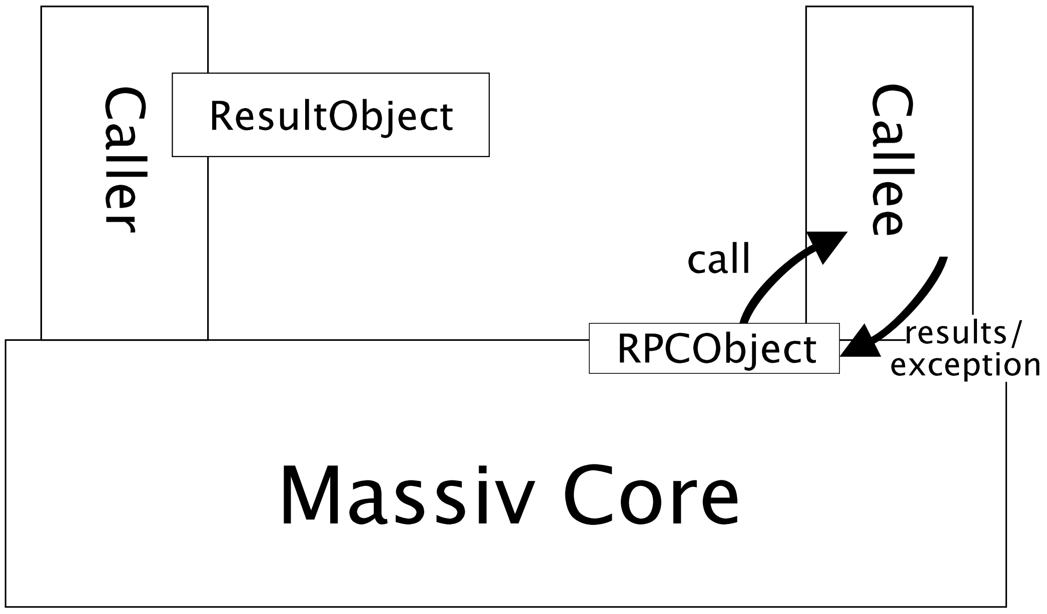
1. When a stub object is instructed to perform a call, it creates a *RPCObject*, marshalls method arguments into the object and migrates it to the callee object. The stub will also create a *ResultObject*, which will remain on the caller node and track status of the pending call (see Figure 8.1, “Inititating Remote Call”).

Figure 8.1. Inititating Remote Call



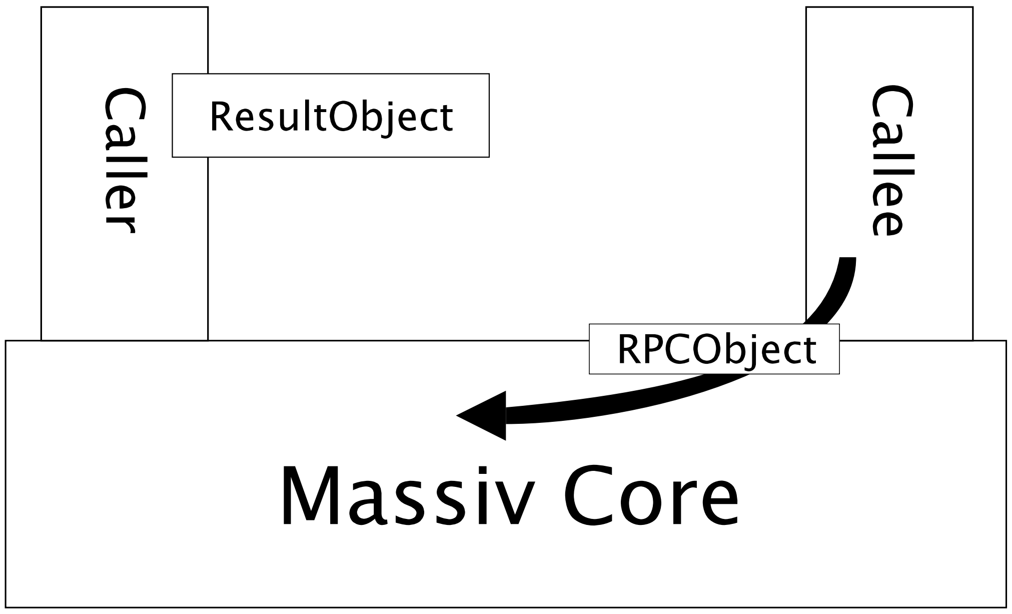
2. When the *RPCObject* with the call request is delivered to the callee object, it performs the call. If the call returns successfully, *RPCObject* will collect the results (values of method result type and *out* and *inout* arguments). Otherwise, if exception is thrown by the method or if the *RPCObject* can't perform the call, information about the error will be stored in the *RPCObject* (see Figure 8.2, “Performing Remote Call”).

Figure 8.2. Performing Remote Call



3. The *RPCObject* is sent back to *ResultObject*, which has been created in the first step (see Figure 8.3, “Sending Back Call Results”)

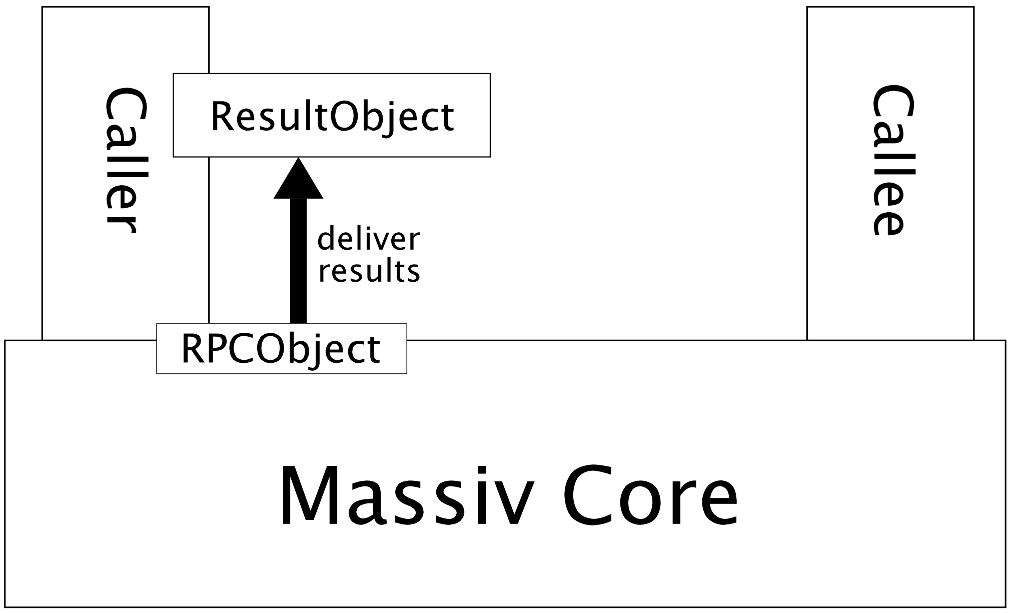
Figure 8.3. Sending Back Call Results



4. When the *ResultObject* receives the *RPCObject*, it changes its state according to the result of the call. If the call is asynchronous, the caller should check state of the *ResultObject* and do whatever he wants.

If the call is synchronous, the *ResultObject* is managed by the stub. When its state changes, stub will either simply return (if the call has been successful) or throw an exceptiong (if the call failed). The execution of the caller code will then resume (see Figure 8.4, “Delivering Call Results”)

Figure 8.4. Delivering Call Results



Note

These figures describe synchronous remote call, and asynchronous remote call when delivery of results is requested. In real code, the most common variant of the RPC used is probably simple asynchronous call, which ignores results of the call. No *ResultObject* is created in that case, and *RPCObject* ignores results of the call and remains on the callee node until it's destroyed by the garbage collector.

8.3. Asynchronous RPC

The most simple and probably the most common remote call variant is *asynchronous remote call*. It has *best effort* and *one-way* semantics: the call will be performed at most once and the callee does not inform the caller about the results of the call. As long as the callee object is reachable by the standard localization protocols, the call will be performed exactly once.

8.3.1. Immediate Asynchronous RPC

To initiate an asynchronous call to method `FUNCNAME`, call the *async_FUNCNAME* method of the stub object:

```
/* Call method bar of object foo, passing 1 as param. */
foo->async_bar( 1 );

/* Call method baz of object foo, passing 2 as param. */
foo->async_baz( 2 );
```

This call will create a RPC request, schedule its migration to the callee object, and return immediately. When the request is delivered to the callee node, the target method will be called. The result returned by the method and modifications of the *out* (or *inout*) arguments will be forgotten. There is no way to determine the status and the result of the call by the caller.

As you can see, *async_* methods that perform the call don't have the same signature as the methods they call. The following differences apply:

- The *out* arguments are not passed to the *async_* methods.
- The *inout* arguments are passed to the *async_* methods, but they will not be modified.
- The return type of the *async_* methods is always Pointer< ResultObject >. You can safely ignore it, beacuse it will be always a `null` pointer, unless reply to the call is requested by the caller (see Section 8.5.4, “Getting Reply to Asynchronous RPC”.)

8.3.2. Scheduling Asynchronous RPC

In the example in the previous section, the request was sent to the callee node as soon as possible. Execution of the call can be also scheduled to some specific simulation time using the `param()` method of the stub object:

```
/* Call method bar of object foo, passing 1 as param. The call will be
   performed when the global simulation time reaches 1000 seconds. */
foo->param( RPC_TIMED, STime( 1000.0 ) ).async_bar( 1 );

/* Call method baz of object foo, passing 2 as param.
   The call will be performed in 10 seconds. */
foo->param( RPC_DELAYED, STime( 10.0 ) ).async_baz( 2 );
```

The `param()` method is described in RPCStubs, examples of advanced use of this method can be seen in Section 8.5, “Advanced Techniques”.

8.4. Synchronous RPC

In addition to the *asynchronous* RPC, the Massiv also implements the *synchronous* RPC (also called *SRPC*). Unlike asynchronous call, synchronous call will block execution of the current thread until the request is delivered, callee method returns and call results are delivered back to the caller node. Context may switch to different threads while the current thread is blocked. See Section 17.1, “The Model Used By the Core” for more information about the Massiv threading model.

Synchronous calls in the Massiv have *best effort* semantics, they will be performed at most once. The call request may have several outcomes: its delivery may either succeed or fail, and if it succeeds, the call itself may either succeed or fail. The Massiv will always try to inform the *ResultObject* about the call outcome (see Section 8.2, “RPC Model”). If the results are delivered to the *ResultObject*, it will either return them to the caller (in case of success), or throw an exception. If the delivery of the results fails, information about the call will be lost, the call will timeout and the relevant exception will be thrown too.

8.4.1. Performing The Call

To call a method `FUNCNAME` synchronously, use the *sync_*`FUNCNAME` method of the stub object:

```
/* This variable will hold value of method out argument "result".
Int32 ires;

/* Call method bar of object foo, passing 1 as param.
   Store the value of the out argument "result" into variable ires,
   and the value of the return type into variable bres. */
bool bres = foo->sync_bar( ires, 1 );
```

Unlike asynchronous method stubs, synchronous method stubs have the same signature as the method they call. The Massiv will pass all *in* and *inout* arguments over the network to the callee method, and transfer the results (*out* and *inout* arguments and return type value) back to the caller. The semantics of synchronous call is nearly the same as that of standard local C++ call. However, there are several differences. The most obvious one is that the call may block execution of current thread for quite a long time, and it may even timeout. The SRPC is also the only place where the Massiv may switch context to another thread. More differences and limitations are explained in the following sections.

8.4.2. SRPC Exceptions

As already mentioned several times, the *sync_* stub method may throw an exception. This section tries to clarify when and why this may happen. For more generic information about the Massiv exceptions, see Section 4.3.5, “Throwable Objects and Exceptions”.

There are two reasons why an exception may be thrown:

- Call request has been successfully delivered, the method has been called and it has thrown an exception. The exception will be delivered back to the callee node and rethrown there.
- The call has failed. The Massiv Core throws an exception that indicates the failure.

8.4.2.1. Exceptions Thrown By Callee

The Massiv Core is able to deliver exceptions thrown by the callee method back to the caller. Because the standard migration is used to implement the RPC, the Massiv exceptions transferred over the network must be throwable managed objects (see Section 4.3.5, “Throwable Objects and Exceptions”). (which means, besides others, that they must be properly described in the relevant IDL). If a callee throws such exception, it will be delivered back to the caller and rethrown there.

Other exceptions will be remapped to a single managed exception, `Massiv::Core::Lib::CoreException`. If the exception inherits standard `std::exception`, its `what()` message will be copied to the `CoreException`.

8.4.2.2. Exceptions Thrown By The Core

If the call fails, an exception will be thrown too. The Core may throw one of the following exceptions:

- `Massiv::Core::Lib::IllegalPointerConversionException` : Thrown when the Massiv is unable to perform conversion from pointer to `Massiv::Core::Object` to pointer to object of type of the class the method is defined in. That would be `Foo` in the example above. There may be two reasons why this conversion can't be performed:

    - The conversion is ambiguous, because the object inherits the class multiple times. In the example this would mean that object `foo` is of type that inherits class `Foo` multiple times. You should avoid such inheritance hierachies, check Chapter 5, *Pointers* for more reasons.

    - The pointer used in the call points to class of incompatible type. In the example above it would mean that the object `foo` does not inherit class `Foo`. While the Massiv tries to check all potentially dangerous pointer assignments at run-time, it's not always able to determine if value assigned to a pointer actually points to an object of a compatible type, if the object is not local. Again, see Chapter 5, *Pointers* for in-depth information about pointers.

- `Massiv::Core::Lib::RemoteCallFailedException` : Many reasons why this exception may be thrown exist:

    - The call has been cancelled. This happens when the node shuts down (or when the call timeouts, see below). Note that the remote method may have been called even if this exception is thrown.

    - The call timed out and has been cancelled by the Core. Something weird happened, or either the request or the reply has been thrown away because of security reasons (see Section 8.4.3, “SRPC Security and Limitations”).

    - The request can't be delivered to the callee object. It no longer exists, or its localization has failed (but probability of that happening is nearly zero, something like million-to-one).

    - The caller tried to use wrong combination of call options and call variant. Actually, the Massiv is a bit messy regarding these problems. `Massiv::Core::Lib::InvalidArgumentException` should be probably thrown instead. Actually, the Massiv may even fail with an assert or throw a different exception if really dumb combination of arguments is used. To be safe, always use arguments that make sense. (This is not a bug, it's a feature - you don't want to pass random arguments to the Massiv and then catch exceptions to see what happened. Bad arguments are considered internal error by many parts of the Massiv API, even if they originated in code written on top of the Core.)

8.4.3. SRPC Security and Limitations

The Massiv security model expects that server nodes can be trusted. However, the Massiv does not trust client nodes. For example, we can't expect that client nodes always reply to requests. Because of that reason, it's illegal to call synchronously methods of objects owned by client nodes.

8.4.4. Advantages and Disadvantages of SRPC

The greatest advantage of synchronous RPC is obvious: its semantics is similar to that of standard local call and it's the easiest way to get call results. However, it's not recommended to use SRPC between servers too often in the Massiv because of the following reasons (most of them apply especially if you implement game-like application on top of the Massiv; you should have probably used another middleware if you wanted to implement different kind of application):

- The Massiv distribution model allows large distances (and pings) between server nodes. Synchronous calls may block for quite a long time, which may harm real-time requirements of the simulation.
- SRPC introduces cooperative threading into the simulation. While a stub waits for call results, other thread may (and probably will) run. The same object could be reentered by the other thread to service another request in the meantime. It's suprisingly easy to make serious and hard-to-debug errors in such environment, especially if you are not used to it. Make sure you know *exactly* in what state object performing a SRPC may be, and what may happen when someone wants to access the object or to call its method.
- If you try to solve the problem above using a locking mechanism, be aware of potential deadlocks.

These reasons apply to client-to-server communication too. Most simulation-related requests from client should be asynchronous and the replication should be used to send presentation-related data from servers to clients.

However, there are some cases when the SRPC is really useful and should be used. One such example is the movement of entities in the Massiv Demo. When an entity wants to move to a different location, it must first make sure that it can move to the new location and then move there. All entities *reserve* area around them to make sure that they never collide, and both area around the old location and the new location must be reserved before the entity can actually change its position. If the movement fails, the entity remains at the old location. If the movement succeeds, the entity cancels reservations around the old location.

In the Massiv Demo we have chosen to divide the world into evenly-sized sectors. These sectors can be arbitrary distributed among the servers and entities can freely move between the sectors. This means that there are no “teleports when moving between different areas of the world” problems/ hacks. However, it also makes the implementation of the entity movement a bit tricky. Even this simple collision system requires a modification of several data structures during the movement, and those data structures might be owned by different server nodes.

The implementation of `do_move()` method makes heavy use of the SRPC. During its exection, it does not modify entity data, and all modifications done to other data structures leave them in consistent state and are reversible. When an entity wants to move, and it's not currently moving, it simply calls `do_move()`. However, if it's already moving, it just remembers that it wants to move and where it wants to move to. If `do_move()` fails, it reverts modified data structures to their previous state. If it succeeds, it changes entity position. In both cases, if another movement was requested while `do_move()` had been moving the entity, it will try to move the entity again.

This implementation is optimal. If an entity is not near a boundary between sectors owned by different nodes, `do_move()` will finish immediately, because the local SRPC is optimized (see Section 8.5.2, "Synchronous RPC Optimizations"). Otherwise it may take a while, but the implementation keeps calls between different nodes at the minimal rate, and the movement does neither block nor interfere with execution of other entity methods.

For more details check `src/demo/lib/server/entity.cpp`, methods `move_entity()` and `do_move()`.

8.5. Advanced Techniques

This section describes several advanced RPC techniques. The subsections are sorted roughly by their difficulty (in the increasing order) and usefulness (in the decreasing order). You should understand the replication (Chapter 7, *Replication*) before reading this section.

8.5.1. Triggering Replication by RPC

Both synchronous and asynchronous RPC can be used to trigger a replication of a callee object to a caller node:

```
/* Call asynchronously method bar of object
   foo, passing 1 as param and request replication
   of foo to local node for 60 seconds. */
foo->request_replica( STime( 60.0 ) ).async_bar( 1 );

/* Synchronous call with replication request. */
Int32 ires;
bool bres = foo->request_replica( STime( 60.0 ) ).sync_baz( ires, 2 );

/* Asynchronous call scheduled to the future */
/* combined with replication request. */
foo->param( RPC_DELAYED, STime( 10.0 ) ).
    request_replica( STime( 60.0 ) ).async_bar( 2 );
```

Usefulness of this will be obvious in the next section.

8.5.2. Synchronous RPC Optimizations

The Massiv offers two methods of SRPC optimization. The first one is really simple and automatic - if the callee object is local, no `RPCObject` and `ResultObject` objects will be created. The call will be performed directly instead of the standard RPC-over-migration mechanism, and will be nearly as fast as the direct call using `Pointer` or `WeakPointer`.

Note

It is ensured that if the "target" object is local and not replica, it is *stack-strong* referenced during the call. Thus, it cannot migrate away causing an inconsistency.

Warning

In the current implementation the semantics of locally optimized SRPC calls is different from the standard SRPC. If a callee throws an unmanaged exception, it will not be remapped to `Massiv::Core::Lib::CoreException`.

Note

Asynchronous calls are never optimized this way, even if they are sheduled to ASAP (default behavior). This ensures that semantics of asynchronous calls is always the same.

The second optimization is similar to the first one, but it must be explicitly requested. It allows to optimize SRPC to *const* methods as local call even when the callee object is not local, if consistent replica of the object exists on the caller node. To enable this optimization, use:

```
/* Call method bar of object foo synchronously. If foo is local */
/* object, or if a consistent replica of foo exists, the call */
/* will be optimized. */
Int32 ires;
foo->param( RPC_ALLOW_LOCAL_REPLICA_CALL ).sync_bar( ires, 1 );

/* It's illegal to call non-const methods with the */
/* RPC_ALLOC_LOCAL_REPLICA_CALL flag set. This call will fail. */
foo->param( RPC_ALLOW_LOCAL_REPLICA_CALL ).sync_baz( ires, 2 );

/* This way, you will request replication of foo if it's */
/* not local and its replica does not exist yet, and */
/* optimize the call otherwise. */
foo->param( RPC_ALLOW_LOCAL_REPLICA_CALL ).
    request_replica( 60.0 ).sync_bar( ires, 3 );

/* This version is a shortcut for the same thing. */
foo->optimize_replica( 60.0 ).sync_bar( ires, 4 );
```

Note

Currently the last two commands are not implemented 100% optimally. If a consistent replica exists, the replication request will never be sent to the callee node. When replication timeouts, the local replica will be destroyed by the callee node, and next call will be sent over the network. That call will trigger replication again. You should be aware of this behavior.

Warning

Let's assume the `baz` method sets variable of the `Foo` object to `param`, and `bar` returns its previous value in `result`. Check the following code:

```
/* Set value of foo to 1. */
Int32 ignore;
foo->sync_baz( ignore, 1 );

/* Read value of foo to ires. */
Int32 previous_value;
foo->optimize_replica( 60.0 ).sync_bar( previous_value, ignore );
hope( previous_value == 1 );
```

It's actually wrong! The assert (using the Massiv debugging hope macro here) may fail, iff all of the following conditions are met:

1. Object `foo` is not local.
2. Consistent replica of `foo` exists.
3. Replica of `foo` still contains an old state of the object when the `bar` is called (the call to `bar` occured *before* the replica had been synchronized with its original.

As you can see, bugs like this might be hard to debug, because the conditions of failure are met only rarely.

Note

This optimization is the only place where the Massiv checks a value of the *const* method attribute. That's why it was said that the definition of this flag is a bit vague. At the time it was introducted to IDL its meaning was “the method is *const* in the C++ point of view”. However, in reality it means “replica-optimized SRPC to this method is illegal”. You may want to mark non-*const* (in C++ sense) methods as *const* in IDL, if calling this method may be dangerous if object is replica.

8.5.3. Asynchronous RPC to Replicas

This special feature allows you to call methods on replicas of a given *local* object, instead of on the object itself. Note that this is different from the SRPC optimization mentioned above. The call request will be delivered to all known replicas of the object, but never to the object itself. This weird feature allows object (or anyone else) to communicate with all its replicas.

For example, it can be used to implement “special effects” on client nodes. This may include, but is not limited to:

- Sound effects. When an entity should make a sound, it sends request to play a given sample to all its replicas. All clients which see the entity will play the sound effect.

- Graphic effects, for example sparkles, explosions. Pretty much the same thing.

- Trigger animation of an entity. You could use replicated variable of an entity object to indicate which animation it should play. However, that's not very useful way to trigger one-shot animations, such as firing a gun, slashing a sword, etc.

Unfortunately, because the Massiv Demo lacks special effects, this feature is not used there at all.

Only single `param()` flag is needed to change behavior of asynchrnous call to call-to-replicas:

```
foo->param( RPC_REPLICAS ).async_bar( 1 );

/* You may delay this call as well. */
foo->param( RPC_REPLICAS | RPC_DELAYED, 10.0 ).async_bar( 2 );
```

Note

The object (`foo` in the example) must be local. Otherwise the Core does not know anything about locations of its replicas and will not deliver the calls.

Note

It's illegal to request reply to asynchronous call to replicas (see next section), or to perform synchronous call to replicas.

8.5.4. Getting Reply to Asynchronous RPC

As you already know, the implementation of SRPC creates two objects. The *ResultObject* stays on the caller node, while the *RPCObject* migrates to the callee node (call request) and back to the caller (results). You can actually do the same thing with the asynchronous RPC too. If you set the `RPC_RESULTS` flag, the *async_* call will return a pointer to the *ResultObject*, and the *RPCObject* will be instructed to return back to the *ResultObject* with the call results. You can check the *ResultObject* regularly to monitor the state of the call:

```
/* Initiate the call. */
Pointer< ResultObject > result = foo->param( RPC_RESULTS ).async_bar( 1 );

/* Later somewhere in a galaxy far away... :) */

/* Check state of the call. */
if( result->state != ResultObject::STATE_UNKNOWN )
    {
    if( result->state == ResultObject::STATE_SUCCESS )
        {
        /* Get a packet with method results. */
        std::auto_ptr< MethodPacket > packet( result->create_results() );

        /* Get the value of the boolean return type as a string. */
        const std::string bres_str( packet->get_argument_value( -1 ) );

        /* Get the value of the integral out argument result. */
        const std::string ires_str( packet->get_argument_value( 0 ) );

        /* Cast to the real packet type. */
        METHOD_PACKET( Foo, bar ) * const results =
            checked_cast< METHOD_PACKET( Foo, bar ) * >
            ( packet.get() );

        /* Access results directly. */
        const bool bres = results->_result;
        const Int32 ires = results->result;

        /* Can access param too, but its value will be random. */
        const Int32 irand = results->param;
        }
    }
```

It's illegal to request reply from client-side objects (the only one with well-known object id is client node object).

See Section 8.8, “RPC Reference Guide” for complete listing of public methods of *ResultObject* and *MethodPacket*.

8.5.5. Dynamic RPC

A call where method name and arguments are not known at compile-time and are constructed at run-time (usually from strings) is called a *dynamic call*. Metaobjects (see Chapter 12, *Metaobjects*) provide several methods that can be used to perform a dynamic synchronous RPC. The following example demonstrates a dynamic call using method which identifies the method to call by a string. See the Massiv Core Reference Guide for in-depth reference guide to the `MetaObject` API.

Example 8.2. Dynamic RPC

```
using Massiv::Core;

const MetaObject * const metaobject = foo->get_metaobject();  ❶

const std::string method_name( "bar" );  ❷

const SInt32 param = ...;  ❸
std::stringstream arguments;
{
  TextWriter tw( arguments );
  const Serializer::Description desc;
  param.text_write( tw, desc );
}

TextReader tr( arguments );  ❹
tr.next_line();

std::auto_ptr< MethodPacket > results  ❺
    (
    metaobject->remote_call_method
        (
        foo,
        method_name,
        tr
        )
    );
```

❶ This is the easiest way to obtain a pointer to a metaobject of given managed object. See also Section 12.2.1, “Obtaining a MetaObject”.

❷ In this case we identify the method to call by its name.

❸ Here a stream containing all *in* and *inout* arguments is constructed. The arguments are expected to be stored in their textual serialization form.

In the example, a serializable type and a text writer object are used to ensure that the argument is serialized correctly. If the method `foo()` had multiple arguments, method `write_space()` of `TextWriter` would be used to separate the arguments.

Note

In this case standard operator<< could be used to write the integer to the stream, because that's how integers are serialized to text streams. However, we wanted to demonstrate how to write any property (or corresponding serializable type) to a stream without any knowledge of serialization internals.

The `Serializer::Description` object must be passed to all serialization methods. It's used to describe special serialization options during migration and replication. Always use default-constructed serializer description here, it ensures that all properties will be serialized as needed by the remote call.

❹ A `TextReader` object is constructed from the argument stream. The `next_line()` method must be called to intruct the reader to read in the first (and only) line of the stream.

❺ This is how the call itself is made. It may throw a `Lib::RemoteCallFailedException` exception with “Invalid method name” description if the method name is invalid (the object does not have such method), serialization exception if the arguments are not stored properly in the stream, or any RPC-related exception as described in the previous sections. On success, a new method packet with method results is returned. It's documented in Section 8.8, “RPC Reference Guide”.

Note

`TextWriter` and `TextReader` classes simplify work with line-oriented textual streams. If you find them useful, you can freely use them in your Massiv-based application. For example, the Massiv Demo uses these classes to parse all textual data objects. Refer the Massiv Core Reference Guide for more information about the classes and their methods.

Note

Currently there is no easy way to perform a dynamic asynchronous RPC. You could check how the `remote_call_method()` methods are implemented (see `src/core/object/metaobject.cpp`) and use it as a starting point for implementation of dynamic asynchronous RPC. It should be quite easy; asynchornous dynamic RPC has never been implemented because no-one really needed it yet.

8.6. Configuration and Statistics

For description of RPC configuration in brief implementation notes, please refer to Section 27.14, “Remote Procedure Call”.

Statistics regarding synchronous RPC are stored under the `/Statistics/System/SRPC` registry node.

8.7. Method Arguments and Results

This section describes the way arguments are handled by the RPC. Please refer to Section 10.13, “Property and Argument Types” for information about mapping between the IDL types and C++ *in/out/inout* types.

8.7.1. Pointers

It's illegal to use a strong pointer, or a structure containing a strong pointer, as an argument or the return type. Strong pointers always define a migration group (the group of objects that cannot be spread over more nodes), but the RPC can't be used to migrate more other objects.. Semantics of such calls would be hard to defined and probably non-obvious.

This means that you can't pass data structures consisting of multiple data objects, that from a migration groups implemented, as arguments of methods when using the RPC. You must use the standard migration instead. However, you can use arrays of objects and other data structures, implemented as properties, as method arguments.

Note

IDL compiler, factgen.pl, checks this.

8.7.2. Managed Objects

You can use a managed object as a method argument or a return type, as long as it is a *value type* (see Section 4.3.4, “ValueTypes”).

The IDL to C++ type mapping might indicate that the relevant objects are passed by reference. However, that's not completely true. They are passed by reference from the caller to the stubs and when calling the method of the callee object. However, they are copied and passed by value internally, especially over the network. Because of this the Massiv requires that the type of the actual argument (the object passed to the call) and the type of the formal argument (type specified in method declaration) must be the same.

Unfortunately, because references are used in the stub and method interfaces to save a few unnecessary copies, C++ compiler will not issue compile-time warning when you pass an object of a derived class instead of the correct one. The Massiv will check the types at run-time and throw `Massiv::Core::Lib::RemoteCallFailedException` if the types do not match.

8.8. RPC Reference Guide

This section briefly describes all *public* RPC-related classes, methods and enumerations. For in-depth documentation of all RPC classes and methods, check the Massiv Core Reference Guide (module RPC) and sources.

Name

RPCStubs -- client-side (caller-side) part of RPC implementation.

RPCStubs

Synopsis

```
class RPCStubs
    {
public:

    RPCStubs & param
        (
        int           flags = RPC_DEFAULT,
        const STime & time = STime()
        );

    RPCStubs & request_replica
        (
        const STime & timeout
        );

    RPCStubs & optimize_replica
        (
        const STime & timeout
        );
    };
```

Description

The synopsis is a bit misleading. For each managed class (class described in the IDL), single stub class will be generated. Each stub class implements the three methods mentioned above, as well as methods performing the call (described in Section 8.3, “Asynchronous RPC” and Section 8.4, “Synchronous RPC”). In the synopsis, *RPCStubs* stands for class name of any stub class.

Dereferencing a *Remote* pointer returns a stub object for a class of a given type. To be able to use *Remote* pointers to class described in `foo.idl`, include generated header `foo_rpc.h`.

You should never store a reference or a pointer to a stub object. Each of the three “universal” methods returns reference to the stub object itself, so they can all be called easily in a single statement. Actually, it's hard not to do the whole set-up-options-and-perform-the-call in a single statement, because dereferencing a *Remote* pointer always returns a new stub object instance with default options set. This means that you don't really have to care about the real stub class names.

Method `param()`

Use `param()` to change call flags and delivery time. The method has two optional arguments:

- *`flags`*: Combination of RPCFlags, described below.
- *`time`*: Simulation time when the remote method should be called. Depending on *`flags`*, it's

either absolute (if `RPC_TIMED` flag is set) or relative to current simulation time (if `RPC_DELAYED` flag is set).

Default parameters yield one-way asynchronous immediate call or immediate synchronous call.

The following public RPCFlags are defined:

Table 8.1. RPCFlags

Flag	**Description**
`RPC_TIMED`[a]	The *`time`* argument is absolute.
`RPC_DELAYED`[ab]	The *`time`* argument is relative to current simulation tieme.
`RPC_RESULTS`	When performing asynchronous call, the stub will return a *ResultObject* and return pointer to the object. It can be used to monitor call status and to retrieve call results. See Section 8.5.4, "Getting Reply to Asynchronous RPC".
`RPC_ALLOW_LOCAL_REPLICA_CALL`[c]	Enable replica-optimized SRPC. See Section 8.5.2, "Synchronous RPC Optimizations".
`RPC_REPLICAS`[b]	Call method of replicas of given object. This flag can't be combined with `RPC_RESULTS`. See Section 8.5.3, "Asynchronous RPC to Replicas".
`RPC_DEFAULT`	Default value of flags. Equal to `RPC_TIMED`.

[a] Exactly one of `RPC_TIMED` and `RPC_DELAYED` flags must be set.
[b] It's illegal to set this flag when performing a synchronous call.
[c] It's illegal to set this flag when performing an asynchronous call.

Because synchronous calls can't be scheduled into the future, you can't set the `RPC_DELAYED` flags or *`time`* argument to non-zero time.

Method `request_replica()`

Request replication of the callee object to the caller node for *`timeout`* number of seconds. See Section 8.5.1, "Triggering Replication by RPC".

Method `optimize_replica()`

Shortcut for

```
param( RPC_ALLOW_LOCAL_REPLICA_CALL ).request_replica( timeout );
```

See Section 8.5.2, "Synchronous RPC Optimizations".

Name

ResultObject -- object tracking status of a remote call, destination of call results.

ResultObject

Synopsis

```
class ResultObject : public Object
    {
public:

    enum State;

public:

    PEnum< State >        state;

    PPointer< Throwable > exception;

public:

    void cancel() const;

    std::auto_ptr< MethodPacket > create_results() const;
    };
```

Description

ResultObject is returned by asynchronous call stub when the `RPC_RESULTS` flag is set using the `param()` method. It can be used to monitor the call state and inspect its results.

Note

ResultObject is used internally to implement the SRPC and offers more methods and data for that purpose. You should use only features described in this documentation.

Call State

The `state` field of an *ResultObject* may contain one of the following values:

Table 8.2. ResultObject state

Name	Description
`STATE_UNKNOWN`	Waiting for a reply to the call request.
`STATE_SUCCESS`	The call has succeeded. Use the `create_results()` method to retrieve the results.
`STATE_DELIVERY_FAILED`	Failed to deliver the request to the callee object.

Name	**Description**
`STATE_UNSUPPORTED_INTERFACE`	The callee object does not implement the requested interface. Check Section 8.4.2.2, "Exceptions Thrown By The Core", description of IllegalPointerConversionException, to see when this may happen.
`STATE_EXCEPTION`	An exception has been thrown on the callee node. The `exception` filed of `ResultObject` will point to the exception. The exception was either thrown by the callee method, or generated by the Core. See Section 8.4.2.1, "Exceptions Thrown By Callee" for information about user exception remapping.
`STATE_CANCELLED`	The call has been cancelled, either by the `cancel()` method or by the Core.

Initial value of `state` is `STATE_UNKNOWN`. When information about call results is delivered back to the *ResultObject*, the `state` will change to a different value. Its value will never change afterwards.

Note

The enumeration type State is member of the `ResultObject` class, so the fully-qualified name of `STATE_SUCCESS` is `Massiv::Core::ResultObject::STATE_SUCCESS` for example.

Method `cancel()`

Call this method to try to cancel pending remote call.

Warning

This method does not guarantee that the call will be cancelled. It will be able to cancel the call only if the request has not migrated to the callee node yet.

Method `create_results()`

This method creates structure containing the results of the remote method call. The call `state` must be `STATE_SUCCESS`. The results are stored in a `MethodPacket` object (described below), the callee is responsible for its destructrion.

Name

MethodPacket -- structure containing call arguments and results.

MethodPacket

Synopsis

```
class MethodPacket
    {
public:

    virtual std::string get_argument_value
        (
        int index
        ) const;
    };

#define METHOD_PACKET( class_name, method_name ) ...
```

Description

Method packets are classes used to keep and serialize method arguments and results and to call methods on the callee node. They are automatically generated from the IDL class descriptions.

Class `MethodPacket` offers generic interface to all *method packet* classes. Method `get_argument_value` is its only method public for the Core user.

Method `get_argument_value`

This method returns value of given method argument as a string. It uses a textual serialization to generate the string. The argument *`index`* is zero-based index into a list of method arguments (`0` corresponds to the first argument, `1` to the second one, etc.). To get value of method return type, use `-1` as the *`index`*.

`Massiv::Core::Lib::InvalidArgumentException` will be thrown if the *`index`* is invalid (lower than or equal to -2, -1 if method does not return a value, or greater than or equal to the number of method arguments.) Undefined value will be returned if the packet has been returned by the `create_results()` method of `ResultObject` (i.e. the packet contains method results), and the *`index`* corresponds to an *in* argument.

Accessing Results Directly

The `METHOD_PACKET` macro expands to name of *method packet* class for given interface and method. For example `METHOD_PACKET( Foo, bar )` yields name of method packet for method `bar` described in the IDL of class `Foo`.

If you have a `MethodPacket` pointer to a *method packet* containing arguments/results of a known method, you can use the `METHOD_PACKET` macro to cast it to the pointer of the correct type. You can then use this pointer to access method arguments and results directly. All *in*, *out* and *inout* arguments are stored in the *method packet* object as *stype* data fields, with names equal to argument names. If the method returns a value, it will be available as `_result` data field.

See Section 8.5.4, “Getting Reply to Asynchronous RPC” for an example.

Packets for Virtual Methods

If class `Foo` introduces a virtual method `bar`, and class `Shoo` inherits `Foo`, there will still be only single method packet class for the `bar` method. `METHOD_PACKET( Shoo, bar )` will yield an invalid class name. This is consistent with IDL - you should not (must not) write description of method `bar` in the IDL description of class `Shoo`.

Simply, `METHOD_PACKET( C, m )` is a valid class name, iff the IDL description of the class `C` containts a description of method `m`.

9. Introduction to IDL

9.1. What Is IDL?

As already mentioned in the introduction, one of the main concepts in the Massiv is the distributed object model. Besides others, some of features of this model are:

- *RPC* (Remote Procedure Call) enables calling of methods on objects that may be located on another node.
- *Object migration* (and *replication*) - the migrating object must be serialized into a network packet/stream, sent over the network and then created and initialized from the packet.
- *Introspection* enables to control some properties and some information (such as name of the class, name of its ancestor, list of properties, etc.) about object at run-time. You can not only view the information, but also modify some parts of it (value of some attributes, etc.) which might is useful for example for an application debugging.

Although C++ doesn't provide a mechanism to manage the *meta-information* about classes and objects (such as name or ancestors), they are crucial for implementation of the features mentioned above. For example, when calling a method via RPC, it is needed to serialize information such as its name, type, actual parameters values and others into a network stream (so-called *marshalling*) and to reconstruct the call on the destination node after transmitting the information over the network.

There are more ways of defining the meta-information additionally. The one the Massiv makes use of is the *IDL* (Interface Definition Language). Perhaps it might be better to call it *ODL* (Object Description Language), but for some historical reasons we decided to keep the original designation.

The main idea of the IDL is that the required information is provided in a special form, often separated from the class declaration/definition, either written into an another file or into the same file in a form of source code annotation.

9.2. IDL in the Massiv

In Massiv, the IDL information is stored in external `.idl` files that are processed by the `src/core/factgen/factgen.pl` IDL compiler. For a given `*.idl` file the compiler outputs `*_generated.h`, `*_generated.cpp` and `*_rpc.h` files that hold implementation of object factories, metaobjects and declarations of the exposed RPC interfaces. See Chapter 11, *Classes Generated From the IDL* for more information about what output the IDL compiler exactly generates. Generated classes are compiled together with the application code using the integrated build system.

What the IDL compiler doesn't product is the standard C++ declaration of the classes described in an IDL file. Thus, the user who wants to write a managed object must provide

- the IDL description
- the C++ declarations of the classes contained in the IDL
- the C++ implementation of the classes

- a record in the relevant `idl.list` file (see Section 9.2.1, “idl.list files”).

9.2.1. `idl.list` files

The IDL processor won't automatically process every IDL file that would emerge in the source tree. To tell the processor about a new IDL you must insert a record into some of the `idl.list` files. For detailed information about this procedure refer to Section D.2, “The idl.list File”.

10. IDL Syntax

10.1. Basic Syntax Overview

As mentioned in the IDL introduction, the IDL descriptions must be contained in specific files that typically have the `idl` extension. You can find these files in the Core source tree as well as in the Demo.

These files are not completely standalone - they can import each other. Almost every IDL file needs some generic declarations that are contained in `src/core/object/object.idl`. This file defines all class, method and property attributes required by the Core, as well as the description of the `Massiv::Core::Object`. This file together with all public Core IDL files can be indirectly imported using the file `src/core/core.idl`.

The most frequent entities contained in the IDL are class descriptions (arbitrary number in each file) of *managed classes* (see Chapter 4, *Managed Objects*). Typically the IDL file name corresponds to the name of the header file that contains C++ definitions of the relevant classes.

For each IDL file `foo.idl` referenced from the `idl.list` (described in Section D.2, "The idl.list File"), a file `foo.h` must exist, because it's automatically included by the sources generated from `foo.idl`. However, classes described in `foo.idl` may be defined anywhere, as long as the files containing the class definitions are properly referenced from `foo.idl` using the "include_file" directive, as described in Section 10.12.1, "The include_header Directive".

Morever, if an IDL file is not referenced from any `idl.list`, no sources will be generated from it and the header with corresponding name may not exist. Such IDL files should not contain any class definitions. One example of such file in the Core is `src/core/core.idl`.

The following example should give you an overview of how class descriptions in the IDL look like in general before the detailed syntax description. It contains a complete description for a very simple and academical class `FooBar` located in the `my_application` namespace. It's base class is `Massiv::Core::Object`, it has several properties and methods. Note that the class definition does not need to end with a semicolon character. For more detailed information, see the remarks below the source code. If you don't have even an idea what about what the example actually means, skip it, continue reading and get back at the end of the chapter.

Example 10.1. A class description in the IDL

```
#import "core.idl"  ❶

namespace my_application {  ❷

class  ❸
    <
    kind = SERVER,  ❹
    root,
    tracked
    > FooBar : ::Massiv::Core::Object  ❺
    {
    enum State { }  ❻

    property< repflags = MIGRATE > State  state;  ❼
    property weak_pointer< Object >       buddy;
    property strong_pointer< FooBar >     next;
    property value_array< uint32< 8 > >   array;

    method set_buddy  ❽
        (
        out weak_pointer< Object > old_buddy,
        in weak_pointer< Object >  new_buddy
        );

    method< const > get_buddy() : weak_pointer< Object >  ❾
    }

} // namespace my_application
```

❶ This imports all public Massiv Core IDL files, the most important being the `src/core/object/object.idl` file, which defines all required attributes and describes the `Massiv::Core::Object`.

❷ Namespaces can be declared using the same syntax as in C++ and have the same semantics. In this example we define everything in namespace `my_application`.

❸ The `class` keyword begins a class description in the IDL. The description consists of the class attributes, name, inheritance information and of the class body (properties, methods, etc.)

❹ Class attributes specify some basic parameters for the class. In the example, the `FooBar` class would be a server-only class (*kind* attribute), garbage collector root (*root* attribute) and tracked by the object provider (*tracked*). The meaning of these attributes is described in Section 10.9.3, “Class Attributes”.

❺ A C++-like syntax is used to define list of base classes. In the example the class that all managed classes must inherit, `Massiv::Core::Object`, is inherited directly.

❻ This line indicates that an enumeration type `State` is defined withing the `FooBar` class. Its enumeration values are not used in the IDL so they are not defined at all. Note that the definition is not terminated by a semicolon character (it's not needed but can be used; in the future versions it might be required to make syntax more C++-like).

❼ The `FooBar` class containts four properties:

- Property `state`, of enumeration type `State` (defined on the line above). The `repflags` property attribute defines replication behavior of the property - in this case the property will migrate (as all properties do), but it will be never replicated.

- Property `buddy`, of type `weak_pointer`. Using the weak pointer for the reference to the `buddy` enables the `FooBar` object and its buddy to be located on different nodes.

- Property `next`, of type `strong_pointer`. The *strong pointer* ensures that this `FooBar` and the `next FooBar` are always located on the same node and migrate together in a single migration group.

- Property `array` is an example of a little bit more complex property type - an array of unsigned 32-bit integers. Only the 8 low bits of the integers are replicated.

❽ Description of a simple method that can be called using the RPC. It has single *out* and single *in* argument. As the name of the method and its arguments indicate, it probably sets the `buddy` property to `new_buddy` and returns its old value in `old_buddy`.

❾ Another method, this one probably returns value of the `buddy` property. This example shows how the return types are defined. The method is marked as `const`. The meaning of this attribute is a bit different from the C++.

10.2. Tokens

There are several types of tokens: identifiers, keywords, literals and special symbols, such as operators and punctuation. Blanks, horizontal and vertical tabs, new-lines, form-feeds and comments (as described below), collectively called *whitespace*, serve to separate tokens and are ignored otherwise.

10.2.1. Comments

As in the C++, the characters `/*` start a comment that terminates with the characters `*/` and the characters `//` start a comment that terminates at the end of the line on which they occur. `/* */` comments do not nest. The characters `//`, `/*` and `*/` do not have any special meaning within a `//` comment, and the characters `//` and `/*` have no special meaning within a `/* */` comment.

10.2.2. Identifiers

An identifier is a sequence of alphanumeric and underscore characters. The first character must be alphabetic or underscore. All characters are significant. Identifiers are case-sensitive.

There is only one namespace for IDL identifiers in each scope. Using the same identifier for a class and an enumeration type in the same scope, for example, is an error.

10.2.3. Keywords

IDL keywords are identifiers that have a special meaning within given scope. Keywords are *not* globally reserved, the following IDL is legal:

```
#import "core.idl"

enum property { }

class method : Massiv::Core::Object
    {
    property uint32 property;
    property bool class;
    property bool weak_pointer;
    method method();
    }
```

However, if C++ reserved words are used as identifiers, corresponding C++ definitions will be illegal and the generated sources won't compile.

10.2.4. Literals

This section describes the following literals: integers, floating-point numbers, characters and strings.

10.2.4.1. Integer Literals

Integer literals can be specified in on of the following forms:

- *decimal* (base 10): Sequence of digits that does not begin with `0` (digit zero).
- *hexadecimal* (base 16): `0x` followed by a sequence of digits and `a` to `f` or `A` to `F` characters representing number ten to fifteen.
- *binary* (base 2): `0b` followed by a sequence of 0 and 1 (digits zero and one).
- *octal* (base 8): `0` followed by a sequence of 0 to 7 (digits zero to seven).

10.2.4.2. Floating-point Literals

A floating-point literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal digits. Either the integer part or the fraction part (but not both) may be missing. Either the decimal point or the exponent (including the `e` or `E` character), but not both, may be missing. If both the decimal point and the exponent are missing, the literal is interpreted as a integer literal.

10.2.4.3. Character Literals

Single character enclosed in single quote (`'`) characters is a character literal. No escape sequences are supported in the character literals. While the parser recognizes character literals, they are not used in the IDL at all.

10.2.4.4. String Literals

String literal is a sequence of characters enclosed in double quote (`"`) characters. Escape sequences inside the double quotes are recognized, but not interpreted by the parser. For example token `"foo\"bar"` will be parsed as a single string, but the `\` character will be retained in the string.

10.3. Parser Directives

The parser directives are syntactic constructions very similar to the C/C++ preprocessor directives (although the IDL parser in the Massiv doesn't have any preprocessor). Each directive begins with the `#` character and cannot be split into more lines (the only exception is a nasty C++ compatible hack - enclosing the end of line into a multiline comment). List of all supported directives follows:

- **`#error`** ***`STRING`***

Prints `STRING` to the `stderr` and causes the processor quit with an error.

- **`#warn`** ***`STRING`***

  Prints `STRING` to the `stderr`.

- **`#include`** ***`STRING`***

  The IDL processor will attempt to find a file specified by *`STRING`* in all directories contained withing the list of paths that should be searched. If the file is not found, the processor will stop with an error. Otherwise the parsing will continue and the contents of the relevant file will be put in place of the **`#include`** directive.

- **`#import`** ***`STRING`***

  This directive has almost the same meaning as the `#include` directive. The difference is that nothing will be done if the file has already been imported or is just being processed. Importing should almost always be used instead of including to prevent duplicate symbol definitions.

10.4. Massiv IDL Grammar

The description of the Massiv IDL grammar uses a syntax similar to Extended Backus-Naur Form (EBNF). Table 10.1, "Massiv IDL EBNF" lists the symbol used in the grammar and their meaning.

Table 10.1. Massiv IDL EBNF

Symbol	**Meaning**
`::=`	Is defined to be
`|`	Alternative
`<text>`	Nonterminal
`"text"`	Literal
`*`	Repeat zero or more times
`+`	Repeat one or more times
`{}`	Group of several syntactic units
`[]`	Optional syntactic unit

The complete grammar follows. Note that it's not 100% correct everywhere. For example, all assignments are strongly typed in IDL, type of the expression that can appear on the right side of the assignment operator is determined by the type of object on the left side of the operator.

```
(1)                  <idl> ::= <definition>*
(2)           <definition> ::= <namespace>
                             | <enum>
```

```
                              | <attribute>
                              | <attr_default>
                              | <class>
                              | <include_header>
                              | ";"
 (3)             <namespace> ::= "namespace"
                                 <identifier>
                                 "{" <definition>* "}"
 (4)                  <enum> ::= "enum"
                                 <identifier>
                                 "{" <enum_body> "}"
 (5)             <enum_body> ::= <enumerator>
                                 { "," <enumerator> }*
                                 [ "," ]
 (6)            <enumerator> ::= <identifier>
                                 "="
                                 <int_expr>
 (7)             <attribute> ::= <attr_kind>
                                 <attr_type>
                                 <identifier>
                                 "="
                                 <const_expr>
                                 ";"
 (8)             <attr_kind> ::= <classattr_dcl>
                               | <propattr_dcl>
                               | <methodattr_dcl>
 (9)         <classattr_dcl> ::= "classattr"
                                 [ "<" [ <classattr_flags> ] ">" ]
(10)       <classattr_flags> ::= <classattr_flag>
                                 { "," <classattr_flag> }*
(11)        <classattr_flag> ::= "inherit"
                               | "idl_internal"
(12)          <propattr_dcl> ::= "propattr"
                                 [ "<" [ <propattr_flags> ] ">" ]
(13)        <propattr_flags> ::= <propattr_flag>
                                 { "," <propattr_flag> }*
(14)         <propattr_flag> ::= "idl_internal"
(15)        <methodattr_dcl> ::= "methodattr"
                                 [ "<" [ <methodattr_flags> ] ">" ]
(16)      <methodattr_flags> ::= <methodattr_flag>
                                 { "," <methodattr_flag> }*
(17)       <methodattr_flag> ::= "idl_internal"
(18)             <attr_type> ::= <buildin_type>
                               | <enum_name>
(19)          <buildin_type> ::= "bool"
                               | "int"
                               | "string"
(20)             <enum_name> ::= <scoped_name>
(21)           <scoped_name> ::= <identifier>
                               | "::" <identifier>
                               | <scoped_name> "::" <identifier>
(22)          <attr_default> ::= "attribute"
                                 <attr_name>
                                 "="
                                 <const_expr>
                                 ";"
(23)             <attr_name> ::= <scoped_name>
(24)                 <class> ::= <class_fwd>
```

```
                                | <class_dcl>
(25)            <class_fwd> ::= "class"
                                <identifier>
                                ";"
(26)            <class_dcl> ::= <class_header>
                                "{" <class_body> "}"
(27)         <class_header> ::= "class"
                                [ "<" [ <classattr_vals> ] ">" ]
                                <identifier>
                                [ <class_inheritance> ]
(28)       <classattr_vals> ::= <classattr_val>
                                { "," <classattr_val> }*
(29)        <classattr_val> ::= <boolclassattr_name>
                                | <classattr_name> "=" <const_expr>
(30)   <boolclassattr_name> ::= <scoped_name>
(31)       <classattr_name> ::= <scoped_name>
(32)    <class_inheritance> ::= ":"
                                <base_class_name>
                                { "," <base_class_name> }*
(33)      <base_class_name> ::= [ "virtual" ] <class_name>
(34)           <class_name> ::= <scoped_name>
(35)           <class_body> ::= <in_class>*
(36)             <in_class> ::= <enum>
                                | <attr_default>
                                | <class>
                                | <property>
                                | <method>
                                | ";"
(37)             <property> ::= "property"
                                [ "<" [ <propattr_vals> ] ">" ]
                                <type>
                                <identifier>
                                ";"
(38)        <propattr_vals> ::= <propattr_val>
                                { "," <propattr_val> }*
(39)         <propattr_val> ::= <boolpropattr_name>
                                | <propattr_name> "=" <const_expr>
(40)    <boolpropattr_name> ::= <scoped_name>
(41)        <propattr_name> ::= <scoped_name>
(42)                 <type> ::= <stype>
                                | <array_type>
                                | <class_type>
                                | <dictionary_type>
                                | <event_type>
                                | <pair_type>
                                | <pointer_type>
                                | <set_type>
(43)                <stype> ::= <bool_type>
                                | <color_type>
                                | <enum_type>
                                | <float_type>
                                | <floatq_type>
                                | <int_type>
                                | <orientation_type>
                                | <string_type>
                                | <time_type>
                                | <vector_type>
                                | <vlint_type>
```

```
(44)            <bool_type> ::= "bool"
(45)           <color_type> ::= "color"
(46)            <enum_type> ::= <enum_name>
(47)           <float_type> ::= float32
                              | float64
(48)          <floatq_type> ::= { floatq32 | floatq64 }
                                "<" <floatq_traits> ">"
(49)        <floatq_traits> ::= <string_literal>
(50)             <int_type> ::= <bitint_type>
                                [ "<" <bit_width> ">" ]
(51)          <bitint_type> ::= int8
                              | int16
                              | int32
                              | int64
                              | uint8
                              | uint16
                              | uint32
                              | uint64
(52)            <bit_width> ::= <int_expr>
(53)     <orientation_type> ::= "orientation"
(54)          <string_type> ::= "string"
(55)            <time_type> ::= "time"
(56)          <vector_type> ::= "vector2"
                              | "vector3"
(57)           <vlint_type> ::= vlint8
                              | vlint16
                              | vlint32
                              | vlint64
                              | vluint8
                              | vluint16
                              | vluint32
                              | vluint64
(58)           <array_type> ::= <parray_type>
                              | <varray_type>
(59)          <parray_type> ::= "property_array"
                                "<" <type> ">"
(60)          <varray_type> ::= "value_array"
                                "<" <stype> ">"
(61)           <class_type> ::= <class_name>
(62)      <dictionary_type> ::= <dictionary_kind>
                                "<" <dictkey_type> "," <dictvalue_type> ">"
(63)      <dictionary_kind> ::= "dictionary"
                              | "multi_dictionary"
(64)         <dictkey_type> ::= <type>
(65)       <dictvalue_type> ::= <type>
(66)           <event_type> ::= "event_handle"
(67)            <pair_type> ::= "pair"
                                "<" <pairvalue_type> "," <pairvalue_type> ">"
(68)       <pairvalue_type> ::= <type>
(69)         <pointer_type> ::= <pointer_kind>
                                "<" <class_name> ">"
(70)         <pointer_kind> ::= "strong_pointer"
                              | "weak_pointer"
                              | "remote_pointer"
(71)             <set_type> ::= <set_kind>
                                "<" <setvalue_type> ">"
(72)             <set_kind> ::= "set"
                              | "multi_set"
```

```
(73)        <setvalue_type> ::= <type>
(74)               <method> ::= "method"
                                [ "<" [ <methodattr_vals> ] ">" ]
                                <identifier>
                                "("
                                [ <arguments> ]
                                ")"
                                [ <return_spec> ]
                                ";"
(76)      <methodattr_vals> ::= <methodattr_val>
                                { "," <methodattr_val> }*
(77)       <methodattr_val> ::= <boolmethodattr_name>
                              | <methodattr_name> "=" <const_expr>
(78)  <boolmethodattr_name> ::= <scoped_name>
(79)      <methodattr_name> ::= <scoped_name>
(80)            <arguments> ::= <argument>
                                { "," <argument> }*
(81)             <argument> ::= <pass_semantics>
                                <type>
                                <identifier>
(82)       <pass_semantics> ::= "in"
                              | "out"
                              | "inout"
(83)          <return_spec> ::= ":"
                                <type>
(84)       <include_header> ::= "include_header"
                                <string_literal>
                                ";"
(85)           <const_expr> ::= <bool_expr>
                              | <int_expr>
                              | <string_expr>
                              | <enum_expr>
(86)            <bool_expr> ::= <bool_literal>
(87)         <bool_literal> ::= "0"
                              | "1"
                              | "true"
                              | "false"
(88)             <int_expr> ::= "-"*
                                <int_literal>
(89)          <string_expr> ::= <string_literal>
(90)            <enum_expr> ::= <identifier>
```

The grammar of literal nonterminals (`<int_literal>` and `<string_literal>`) is not included as they were already described. Note that neither float nor character literals are currently used by the IDL.

10.5. Name Lookup and Scoping

The fully-qualified identifier names and name lookup rules are similar to C++. Because the IDL has no *using* directive, no overriding and the IDL ignores class inheritance when performing a name look-up, the actual rules are much simplier.

Each identifier has exactly one global name, usually called *fully-qualified identifier name*. It's constructed by separating list of scope identifiers the identifier is defined in, beginning at the root scope, by `::`, and prepending `::` to this name. Identifiers of the following elements introduce a new scope:

- *namespace* (see Section 10.6, “Namespaces”)
- *enumeration* (see Section 10.7, “Enumerations”)
- *class* (see Section 10.9, “Classes”)

Note

In C++, fully-qualified name of enumerants defined within an enumeration type does not include the identifier of the enumeration type. In IDL it does.

The following example shows definitions of several identifiers and their fully-qualified names in comments:

```
/* root scope */

/* namespace ::foo */
namespace foo {

/* class ::foo::Bar */
class Bar
    {
    /* class ::foo::Bar::Baz */
    class Baz
        {
        }

    /* enum ::foo::Bar::MyEnum */
    enum MyEnum
        {
        /* enumerant ::foo::Bar::MyEnum::NUMBER */
        NUMBER = 1,
        }
    }

/* class ::foo::Bar2 */
class Bar2 : Bar
    {
    }

}
```

Unlike C++, inheritance in the IDL does not introduce new global identifier names for the inherited identifiers. Therefore `::foo::Bar2::Baz` is *not* a name of `::foo::Bar::Baz` and can't be used to refer to it.

The IDL has a single namespace for all identifiers within a given scope. You can't declare multiple identifiers with the same name in the same scope, even if their entity type is different.

The name lookup rules are really simple. You can refer to an identifier either using its fully-qualified name, a qualified name (`<scope-name>::<identifier>`), or just a literal. If the name begins with `::`, it's a fully-qualified name, which is an unique identification of an identifier in IDL, and the name resolution is obvious. Otherwise the name resolution starts at current scope. If concatenation of fully-qualified name of the current scope, `::` and the name yields a fully-qualified name of an existing identifier, the name resultion successfully stops at the identifier. Otherwise the same rule is applied to the parent scope of the current scope, and so on, until the identifier is found, or the root scope is reached without a success.

10.6. Namespaces

Namespaces in the IDL exactly correspond to the C++ namespaces. Namespace definition satisfies the following syntax, which is the same as the C++ namespace syntax:

```
(3)            <namespace> ::= "namespace"
                               <identifier>
                               "{" <definition>* "}"
```

As mentioned in Section 9.2, “IDL in the Massiv” the user must provide the C++ defintion and implementation of the classes described in the IDL. The IDL class description should be located in the same namespace as the C++ class. Otherwise the IDL processor wouldn't be able to generate object factories and metaobjects that would be able to work with the relevant managed class.

The namespace specified in the IDL doesn't have anything in common with the namespace where the object factories and metaobjects are generated (for both of these, the `Massiv::Generated` namespace is used always).

10.7. Enumerations

Enumeration types are similar to enums in C++. Their definition satisfies the following syntax:

```
(4)                 <enum> ::= "enum"
                               <identifier>
                               "{" <enum_body> "}"
(5)            <enum_body> ::= <enumerator>
                               { "," <enumerator> }*
                               [ "," ]
(6)           <enumerator> ::= <identifier>
                               "="
                               <int_expr>
```

Syntax of the integral expression that can be used on the right sight of the enumerator assignemnt is described in Section 10.8.5.2, “Integer Expressions”. Unlike in the C++ enumerations, all IDL enumerants must have assigned a value.

Enumeration type can be defined either in a namespace scope (see Section 10.6, “Namespaces”) or in a class scope (see Section 10.9, “Classes”).

Enumeration types are used in two contexts:

1. As attribute types. In this case definition of the enumerants and their values is important. See

Section 10.8.2, “Attribute Types” for more information.

2. As property types. In this case only the name of the enumeration type is used in the IDL to C++ mapping. The enumeration should contain no enumerants.

Single enumeration type may be used in both contexts, but it's not recommended to do so.

> **Note**
>
> So far, the only place where the constants defined in enumerations may be used is an expression in the context of an assignment to an attribute of a relevant enumeration type. Enumeration contants can't be used in integer expressions.

10.8. Attributes

Attributes allow you to specify special information about declarations of classes, properties and methods in a generic way. For example, the C++ function *specifiers virtual* and *const* would be implemented as method attributes in the IDL (those method attributes actually exist, although the *const* attribute has a slightly different semantics).

You can think of attributes as type key-value pair. The attributes are divided into three kinds: class attributes, property attributes and method attributes. Because the IDL has only single namespace for all identifiers within given scope, it's impossible to define two attributes of different kind with the same name.

Each class, property or method always has the same set of class, property or method attributes, with possibly different attribute values. All attributes must be declared before any class, property or method declaration.

The IDL language itself does not define any attributes, attributes are defined directly in the `idl` files. However, some attributes are required by the IDL processor, and other attributes are required by the Core. All these attributes are declared in `src/core/object/object.idl`, and the IDL processor will reject all `idl` files that do not include declarations of those attributes. The attributes are described in Section 10.9.3, “Class Attributes”, Section 10.10.1, “Property Attributes” and Section 10.11.1, “Method Attributes”.

Values of all non-internal class, property and method attributes are queriable at run-time. Names of the C++ attribute variables are derived from fully-qualified names of the attribute identifiers: the leading `::` is stripped and other `::` are replaced by the underscore character. See Chapter 12, *Metaobjects* for more information about metaobjects and object introspection.

> **Note**
>
> All attributes defined in `src/core/object/object.idl` are declared in the root scope, therefore their C++ names are the same as their IDL names.

> **Note**

Unless you want to modify the Massiv Core, you won't have to define your own attributes. Actually, currently the only way to define new attributes is to modify `src/core/object/object.idl` directly. However, the following sections should help you to understand how the attributes work in general and how to specify their values.

10.8.1. Attribute Definition

Attribute definition satisfies the following syntax:

```
 (7)            <attribute> ::= <attr_kind>
                                <attr_type>
                                <identifier>
                                "="
                                <const_expr>
                                ";"
```

Whether the attribute is a class, property or method attribute is specified by `<attr_kind>`, which can be one of “classattr”, “propattr”, or “methodattr”, optionally followed by attribute flags enclosed in the < and > characters. Attribute flags are simple literals separated by commas that further specify behavior of the attribute. Each attribute kind can be flagged as “idl_internal”, which instructs the IDL processor not to export its value to metaobject introspection structures. If class attribute flag “inherit” is set, the attribute value lookup will change as described in Section 10.8.6, “Attribute Value Lookup”.

The `<attr_type>` specified attribute type, the `<identifier>` specified its name. The `<const_expr>` defines default value of the attribute. It must be a constant expression of type equal to the type of the attribute.

10.8.2. Attribute Types

All attributes are typed. Attribute type definition satisfies the following syntax:

```
(18)            <attr_type> ::= <buildin_type>
                              | <enum_name>
(19)         <buildin_type> ::= "bool"
                              | "int"
                              | "string"
```

The following table describes all attribute types that can be used, and to which C++ types they map in the metaobject introspection structures:

Table 10.2. Attribute Type

IDL Type	C++ Type	Description
bool	bool	A boolean attribute. Can be either `true` or `false`.

IDL Type	C++ Type	Description
int	int	An integral attribute. Can be assigned an arbitrary int value.
string	std::string	A string attribute.
name of an enumeration type in-dentifier	int	An enumeration type attribute. Can be assigned any constant enumeration expression of given enumeration type.

10.8.3. Attribute Default Values

Attribute values are specified by a constant expression of given type, as documented in the next section. Each attribute has a default value, specified in its definition. This default value can be overridden using the following syntax:

```
(22)        <attr_default> ::= "attribute"
                               <attr_name>
                               "="
                               <const_expr>
                               ";"
```

The attribute override is valid until the end of the current scope, where it reverts back to override valid before the scope beginning, or until it's overridden by another “attribute” keyword.

Warning

The attribute default value overridde can cross file boundaries. You should never override the default within a scope that may cross file boundaries.

The following example shows how the overriding works:

```
/* Define a class attribute with default value set to 1. */
classattr int attr = 1;

namespace foo {
/* Default value is still 1. */

attribute attr = 2;
/* Default value is 2 now. */

attribute attr = 3;
/* Default value is 3 now. */

namespace bar {
/* Default value is still 3. */

attribute attr = 4;
/* Default value is 4 now. */

} // namespace foo::bar

/* Default value is 3 again. */
} // namespace foo

/* Default value is 1. */
```

10.8.4. Attribute Value Assignment

In each class definition and property or method declaration, values of attributes of corresponding kind can be specified using the following syntax:

```
(27)         <class_header> ::= "class"
                                [ "<" [ <classattr_vals> ] ">" ]
                                <identifier>
                                [ <class_inheritance> ]
(28)       <classattr_vals> ::= <classattr_val>
                                { "," <classattr_val> }*
(29)        <classattr_val> ::= <boolclassattr_name>
                              | <classattr_name> "=" <const_expr>
(30)   <boolclassattr_name> ::= <scoped_name>
(31)       <classattr_name> ::= <scoped_name>
```

Similar syntax is used to specify property and method attributes.

Class, method or property attribute values are specified directly after the “class”, “method” or “property” keyword, respectively, as a list of name=value assignments separated by commas, enclosed in < and > characters. The attribute specification is optional. Assigning `true` to a boolean attribute can be equally written as the attribute name without any value assignment (i.e. just `foo` instead of `foo = true`).

10.8.5. Expressions

Attribute value can be specified in several context:

- Default value specification in attribute definition.
- Default value override.
- Attribute value assignment in class defintion and property or method declaration.

In each of these contexts, constant expression of corresponding type is used on the right side of the assignment. Currently, the IDL parser can only evaluate primitive constants in place of expressions.

10.8.5.1. Boolean Expressions

In the context of assignment to a boolean, the following values can be used:

- `true` or `1`
- `false` or `0`

10.8.5.2. Integer Expressions

In the context of assignment to an integer, a integer literal (see Section 10.2.4.1, “Integer Literals”), optionally beginning with any numbers of minus (`-`) characters, may be used.

10.8.5.3. String Expressions

In the context of assignment to a string, a string literal can be used.

10.8.5.4. Enumeration Expressions

In the context of assignment to an attribute of enumeration type, identifier of any enumerant defined withing the enumeration type may be used. Note that in this case name lookup and scoping rules do not apply, an identifier of an enumerant is used directly.

10.8.6. Attribute Value Lookup

For each class, property and method, values for all attributes of given kind are always defined. If an attribute is not assigned a value in a class, property or method declaration, the default value of the attribute is used. The default is specified in attribute definition and can be overridden as described in Section 10.8.3, “Attribute Default Values”.

The only exception to this lookup rule are class attributes with the “inherit” attribute flag set. If a class has a base class (all classes except `Massiv::Core::Object` do) and value of an “inherit” class attribute is not specified in the definiton of the derived class, attribute value from the base class is used, even if it's not directly specified in its definition and has been determined by applying this rule recursively. If a class inherits multiple base classes, and value of an unassigned “inherit” attribute is not the same in all base classes, the IDL is ill-formed - class attribute value must be always explicitly assigned in this case.

Note

Default attribute value lookup is never done for “inherit” class attributes, except for the definition of the `Massiv::Core::Object` class. Therefore it's pretty useless to override defaults of “inherit” class attributes.

10.9. Classes

All managed classes defined in the C++ must be described in the IDL. This section documents class description (also reffered to as class definition) syntax.

10.9.1. Forward Declaration

As in C++, classes in IDL can be forward declared using the following syntax:

```
(25)            <class_fwd> ::= "class"
                                <identifier>
                                ";"
```

Class attributes are not specified in a forward declaration.

10.9.2. Class Definition

Class definition consists of a class header and a class body. Classes are defined (or described) using the following syntax:

```
(26)           <class_dcl> ::= <class_header>
                               "{" <class_body> "}"
(27)        <class_header> ::= "class"
                               [ "<" [ <classattr_vals> ] ">" ]
                               <identifier>
                               [ <class_inheritance> ]
(35)          <class_body> ::= <in_class>*
(36)            <in_class> ::= <enum>
                             | <attr_default>
                             | <class>
                             | <property>
                             | <method>
                             | ";"
```

The class header specifies an optional list of class attribute values, the class name and an optional class inheritance specification.

The class body can contain definitions of enumeration types and subclasses, attribute default overrides and declarations of class properties and methods.

10.9.3. Class Attributes

Class attribute value assignments satisfy the following syntax:

```
(28)      <classattr_vals> ::= <classattr_val>
                               { "," <classattr_val> }*
(29)       <classattr_val> ::= <boolclassattr_name>
                             | <classattr_name> "=" <const_expr>
(30)  <boolclassattr_name> ::= <scoped_name>
(31)      <classattr_name> ::= <scoped_name>
```

The following table lists all class attributes defined in `src/core/object/object.idl`:

Table 10.3. Class attributes defined in the Core IDL

Name	Type	Default	Description
abstract	bool	`false`	If `true`, the class is abstract and can't be instantiated.
assignable	bool	`false`	If `true`, the class implements the assignment operator.
client_server_migrate	bool	`false`	If `true`, the client can migrate objects of

Name	**Type**	**Default**	**Description**
			this type to servers.
comparable	bool	`false`	If `true`, the class implements the == operator.
has_strong_pointer	bool	`false`	If `true`, the class contains a strong pointer. This attribute is automatically set by the factgen, you should not assign it a value. However, you can query its value at runtime.
kind[a]	ClassKind	`SHARED`	Kind of the class. It can be `SHARED`, `SERVER` or `CLIENT`. See Section 4.3.7, “Class Kinds” for more info.
no_archive[a]	bool	`false`	If `true`, the instances of the class won't be archived. Class `Massiv::Core::Lib::NodeObjectInterface` has this attribute set to `true`. You should not change value of this attribute: all node objects should be non-archivable, other objects should be archivable.
no_balance[a]	bool	`false`	If `true`, objects of this class will not be migrated by the load balancer.
no_migrate[a]	bool	`false`	If `true`, the instances of the class won't be allowed to migrate. Class `Massiv::Core::Lib::NodeObjectInterface` has this attribute set to `true`. You should not change the value.
node_object[a]	bool	`false`	Specifies whether the class `true` is a node object class. Class `Massiv::Core::Lib::NodeObjectInterface` has this attribute set to `true`. You should not change the value.
root[a]	bool	`false`	If `true`, the instances of the class will be permanent GC roots. See Section 5.4, “Garbage Collector” for more information about the garbage collector.
simulation_startup_notify[a]	bool	`false`	If `true`, the instances of the class will be notified when the simulation is restored from an archive. The `object_updated()` will be called, with *`reason`* set to `SIMULATION_STARTUP`. Setting this attribute to `true` is useful when an object needs to perform special initialization on simulation startup. For example, act-

Name	Type	Default	Description
			ive account objects probably contain information about the connected client. Because account objects are archived as any other objects, this information is stored in the archive. When a new simulation is started, no clients are connected, but account objects that were active when the startup archive was created may contain obsolete information about clients. Therefore the class `Massiv::Core::Lib::AccountObjectInterface` has this attribute set to `true`. The system ensures that all server nodes are running when the `object_updated()` is called. The order of object notifications is undefined. You must not use the synchronous RPC from the `object_updated()` callback.
throwable[a]	bool	`false`	If `true`, the instances of the class can be thrown as an exception using the C++ `throw`.
tracked	bool	`false`	If `true`, the instances of the class will be *tracked* by their *object provider*. This will make migrations of objects to the instances, including remote class to the instances, more efficient, and migration of the instances themselves a bit less efficient.
value_type[a]	bool	`false`	If `true`, the instances of the class are value types. You should not set this attribute directly. Inherit the class `Massiv::Core::Lib::ValueType` which sets the attribute automatically instead. Value types are described in Section 4.3.4, "ValueTypes".

[a] This class attribute has the "inherit" attribute flag set. Different attribute value lookup rules apply it the value is not specified in the class definition. See Section 10.8.6, "Attribute Value Lookup".

10.9.4. Class Inheritance Specification

Class inheritance is specified using the following syntax:

```
(32)    <class_inheritance> ::= ":"
                                <base_class_name>
                                { "," <base_class_name> }*
(33)      <base_class_name> ::= [ "virtual" ] <class_name>
```

```
(34)            <class_name> ::= <scoped_name>
```

All base classes must be defined at the scope of the definition of the derived classes. Forward declaration won't suffice. All base classes must be inherited as *public* in the C++ definition of the class.

Each managed class must inherit `Massiv::Core::Object` exactly once.

10.10. Properties

Declaration of a property satisfies the following syntax:

```
(37)              <property> ::= "property"
                                 [ "<" [ <propattr_vals> ] ">" ]
                                 <type>
                                 <identifier>
                                 ";"
```

Property declaration consists of an optional list of property attribute assignments, the property type and the property name. Property types are described in Section 10.13, “Property and Argument Types”.

It's possible to declare properties with the same name in both base and derived class. The IDL will treat them as two different properties, and they should be defined as such in the C++ definition of the classes.

Access to the properties in the C++ classes can be *public*, *protected* or *private*, as long as the classes are properly defined to be *friends* of their metaobjects, as described in Section 4.3.1, “Implementation”. The access is not specified in the IDL.

Warning

All class properties must be properly described in the IDL. If you forget to describe some properties, the IDL will be processed and the generated sources will compile without any error, but the properties you forgot to describe will not be serialized, migrated, replicated, or archived at all.

10.10.1. Property Attributes

Property attribute value assignments satisfy the following syntax:

```
(38)         <propattr_vals> ::= <propattr_val>
                                 { "," <propattr_val> }*
(39)          <propattr_val> ::= <boolpropattr_name>
                               | <propattr_name> "=" <const_expr>
(40)     <boolpropattr_name> ::= <scoped_name>
(41)         <propattr_name> ::= <scoped_name>
```

The following table lists all property attributes defined in `src/core/object/object.idl`:

Table 10.4. Property attributes defined in the Core IDL

Name	Type	Default	Description
repflags	ReplicationFlags	`REPLICATE`	This attribute specifies the migration and replication behavior of the property. See Section 7.5.2, “Choosing Properties To Replicate” for more information about the replication and for exhaustive enumeration of possible values. The default value means that the property will be always included in replications.
ptr_repflags	ReplicationFlags	`NONE`	This attribute specifies which objects belong to the same migration or replication groups, as described in Section 7.5.3, “Replication and Migration Groups”. It's useless to set value of this attribute on properties that are not pointers, or do not contain pointers.

10.11. Methods

Declaration of a method satisfies the following syntax:

```
(74)               <method> ::= "method"
                               [ "<" [ <methodattr_vals> ] ">" ]
                               <identifier>
                               "("
                               [ <arguments> ]
                               ")"
                               [ <return_spec> ]
                               ";"
```

Method declaration consists of an optional list of method attribute assignments, the method name, specification of method arguments and an optional specification of method return type.

It's illegal to declare methods with the same name in both base and derived class. The IDL does not support method overloading and hiding, and default argument values. Virtual methods should be declared only in the class where they are first defined in the C++, and RPC to virtual methods will work as expected.

Constructors, destructors and operators can't be declared in the IDL. All methods declared in the IDL must be *public* in the C++.

Note

You don't have to declare all methods defined in the C++ in the IDL. However, only the declared methods can be called using the RPC.

10.11.1. Method Attributes

Method attribute value assignments satisfy the following syntax:

```
(76)      <methodattr_vals> ::= <methodattr_val>
                                { "," <methodattr_val> }*
(77)       <methodattr_val> ::= <boolmethodattr_name>
                              | <methodattr_name> "=" <const_expr>
(78)  <boolmethodattr_name> ::= <scoped_name>
(79)      <methodattr_name> ::= <scoped_name>
```

The following table lists all method attributes defined in `src/core/object/object.idl`:

Table 10.5. Method attributes defined in the Core IDL

Name	Type	Default	Description
virtual	bool	`false`	If `true`, the method is *virtual* in the C++ sense. Setting this attribute correctly is not required as it's not (and will not be) used by the RPC at all. However, the value is exported to metaobjects, and therefore should be set correctly to prevent confusion.
const	bool	`false`	Meaning of this attribute is described in Section 8.5.2, “Synchronous RPC Optimizations”. It should be always `false` if the method is not *const* in the C++ sense. Set it to `true` if you want to allow replica-optimized RPC to the method.

10.11.2. Method Arguments and Results

Method argument specification satisfies the following syntax:

```
(80)            <arguments> ::= <argument>
                                { "," <argument> }*
(81)             <argument> ::= <pass_semantics>
                                <type>
                                <identifier>
(82)       <pass_semantics> ::= "in"
                              | "out"
                              | "inout"
```

The `<pass_semantics>` specify how the argument is passed over the network:

- *in* - the argument is passed from the caller to the callee. *out* - the argument is passed from the callee to the caller. *inout* - the argument is passed in both directions.

If the method returns a value, its type is specified using the following syntax:

```
(83)          <return_spec> ::= ":"
                                <type>
```

Argument and return type can be any type described in Section 10.13, “Property and Argument Types”, except for types that are, or contain, a strong pointer.

10.12. Special Directives

This section describes other special IDL directives that do not fit in any other category.

10.12.1. The `include_header` Directive

The “include_header” directive satisfies the following syntax:

```
(84)       <include_header> ::= "include_header"
                                <string_literal>
                                ";"
```

It instructs the IDL processor to include specified file in all files that are generated from the IDL file the derictive appears in. Use it if class described in an IDL files are defined in multiple C++ headers.

10.13. Property and Argument Types

This section describes all IDL types that can be used as types of properties, method arguments and method result types, and their mapping to corresponding C++ types. All the types that are managed classes are described in Section 4.2, “Managed Data” in detail; only a brief information is given here.

Every IDL type can be mapped to one or more native C++ types or special C++ classes, according to the actual context of usage. We distinguish five various usage contexts:

- *simple*

  The most simple C++ type that can hold values of this type on the stack. This type is used when passing a result from SRPC, for example.

  If there is no possible mapping to native C++ type, *serializable type* is used.

- *in argument*

  An input argument of a method. This context is used for example to pass *in* arguments to RPC stubs, and should be used for *in* arguments in C++ definition of methods declared in IDL.

  For simple types, it usually equal to *simple type*, otherwise it's a constant reference to *simple type*.

- *out argument*

  An output or input-output argument of a method. This context is used for example to pass *out* and *inout* arguments to/from RPC stubs, and should be used for *out* and *out* arguments in C++ definition of methods declared in IDL.

  This type is usually a reference to *simple type*.

- *serializable*

  The most lightweight serializable type that can hold values of this type. It's used in all contexts where serialization may be used, but full-blown properties are not needed. For example, generated structures containing RPC arguments use *serializable types*, also calls *stypes*.

  Only some types define lightweight *stypes*, usually the simple ones. If not serializable type exists for the IDL type, property will be used instead.

  For complete list of *stypes*, please refer to the Massiv Core Reference Guide, module Lightweight Serializable Types.

- *property*

  Properties are types that can hold values of this type, can be serialized, they know which property owns them, they remember time of their last modification, etc. All managed objects are properties.

  Properties should be the only types that are contained in managed objects. For a complete list of properties and their documentation, please refer to the Massiv Core Reference Guide, module Properties.

The following sections describe particular IDL types in detail. Each description begins with a table that shows to what C++ types the particular type will be mapped to in the specific context. The programmer needs to know that to be able to implement managed classes (see Chapter 13, *Creating Managed Class*).

Note

All the non-native C++ types used for the mapping of the IDL types are defined in the `Massiv::Core` namespace. However, for better lucidity, we leave this namespace out in the following tables. For example, we write `SBool` instead of `Massiv::Core::SBool`.

10.13.1. Boolean

Table 10.6. Boolean type mapping

Context	Type
IDL	bool
simple	bool

Context	Type
in argument	bool
out argument	bool &
stype	SBool
property	PBool

A classic boolean type that can store either the `true` or the `false` value.

10.13.2. Integers With Fixed-length Serialization

Table 10.7. Integral type mapping

Context	Type
IDL	int32< N >
simple	Int32
in argument	Int32
out argument	Int32 &
stype	SInt< Int32,N >
property	PInt< Int32,N >

int32< N > is an signed 32bit integral type. Only the lower *N* are serialized when the integer is transmitted over the network.

Many modifications of this type exist:

- *Unsigned integer*

  In the IDL the notation of the type is uint32< N >.

  In the other contexts, only replace Int32 by UInt32. For example, UInt32 & or SUInt< Int32,N >.

- *Integer with a bitwidth different from `32`*

  The supported bitwidths are `8`, `16`, `32` and `64`. The syntax is obvious - just put the desired bitwidth instead of the `32` value in the signed `32`-bit integer. Of course, you can do the same for unsigned types.

  This example shows some possible integer declarations.

  ```
  int8< 7 >       i1;
  int64< 32 >     i2;
  uint16< 8 >     i3;
  ```

- *Fully serializable integer*

  If the `N` parameter is ommited, all the bits contained within the number will be serialized. The example of a few possible IDL definitions follows:

Note

Similarly you can ommit the `N` parameter in the int8, int16 or int64 types.

```
int8            i1;
int64           i2;
uint16          i3;
```

10.13.3. Integers With Variable-length Serialization

Table 10.8. VLint type mapping

Context	Type
IDL	vlint32
simple	Int32
in argument	Int32
out argument	Int32 &
stype	SVli< Int32 >
property	PVli< Int32 >

The vlint also represents an integral type. The difference from the types that were mentioned above lies in the serialization. All bits are always serialized, but the values that are close to the zero require less bits in network serialization than integers with large absolute values.

This type is suitable for values that typically won't be too large, but their exact maximum is unknown, such as indices.

Similarly to the classic integers, you can still use either signed or unsigned variable length integers with the bitwidth among `8`, `16`, `32` or `64` bits.

The following example shows some of the possible definitions of the variable length integers:

```
vlint32         vli32;
vlunit16        vlui64;
```

10.13.4. Enumeration Type

Table 10.9. Enumeration type mapping

Context	**Type**
IDL	*ENUM*
simple	*ENUM*
in argument	*ENUM*
out argument	*ENUM* &
stype	SEnum< *ENUM* >
property	PEnum< *ENUM* >

The IDL name of an enumeration type is its scoped name. The enumeration type must be defined both in the IDL and the C++ sources. Its IDL definition may be empty - the constant values are irrelevant. In the table, the identifier *ENUM* is used.

Example - the C++ part:

```
class MyClass : public ::Massiv::Core::Object
    {
    enum Mono
        {
        COLOR_BLACK = 0,
        COLOR_WHITE = 1
        };
    };
```

Example - the IDL part:

```
class MyClass : ::Massiv::Core::Object
    {
    enum Mono { };  // We declare this empty.

    property Mono mono_colors;
        // The enumeration is used by its declared name.
    }
```

Note

The enumeration type may be declared in any namespace, as long as it's defined in the same namespace in the IDL and C++.

Warning

Enumeration types are serialized as integers and no value checking is done. Enumeration types are archived as integers too, therefore all the enumerants should be assigned values in the C++ definition of the integer, and these values should never be changed.

10.13.5. Floating-Point

Table 10.10. Floating-point type mapping

Context	Type
IDL	float32
simple	Float32
in argument	Float32
out argument	Float32 &
stype	SFloat
property	PFloat

Besides the single-precision 32-bit floating-point types, double-precision 64-bit types may be used too. Just replace the `32` with `64`.

10.13.6. Floating-Point With Quantized Serialization

Table 10.11. Quantized floating-point type mapping

Context	Type
IDL	floatq32< "*`TRAITS`*" >
simple	Float32
in argument	Float32
out argument	Float32 &
stype	SFloatQ< Float32, *`TRAITS`* >
property	PFloatQ< Float32, *`TRAITS`* >

This type is a floating-point type that uses quantization during replication (but *not* during migration). The quantization parameters are specified by the *`TRAITS`* class, which must be defined somewhere in the C++ sources. The *`TRAITS`* class is described in the Massiv Core Reference Guide.

Note

Because the IDL does not know anything about the *`TRAITS`* class, it must be specified as a fully-qualified C++ name in a string in the IDL.

10.13.7. String

Table 10.12. String type mapping

Context	**Type**
IDL	string
simple	std::string
in argument	const std::string &
out argument	std::string &
stype	SString
property	PString

ASCII string type.

Warning

During the transfer over the network, only 7 bits of each character are serialized. Do not use non-ASCII characters (characters with integral representation greater than 127) in strings. In the future an Unicode string type may be implemented.

10.13.8. Time

Table 10.13. Time type mapping

Context	**Type**
IDL	time
simple	Stime
in argument	const STime &
out argument	STime &
stype	Stime
property	PTime

The time type is used to hold value of the simulation time specified in seconds.

10.13.9. Event Handle

Table 10.14. Event handle type mapping

Context	**Type**
IDL	event_handle
simple	EventHandle
in argument	EventHandle
out argument	EventHandle &
stype	PEventHandle
property	PEventHandle

The event handle is a handle to a scheduled migration event. The handle can be used as a reference to the event when a programmer wants to cancel (*kill*) the event.

Warning

Migration event cancellation is not reliable. It will work only if the object that should migrate is owned by the same node as the object that tries to cancel the event.

10.13.10. Math Vectors

Table 10.15. Math vector type mapping

Context	**Type**
IDL	vector2
simple	SVector2
in argument	const SVector2 &
out argument	SVector2 &
stype	SVector2
property	PVector2

vector2 represents a 2-dimensional vector. Besides that the Core provides a 3-dimensional vector (vector3).

10.13.11. Orientation

Table 10.16. Orientation type mapping

Context	**Type**
IDL	orientation
simple	SOrientation
in argument	const SOrientation &
out argument	SOrientation &

Context	**Type**
stype	SOrientation
property	POrientation

Orientation type represents an angular orientation. Please refer to the Massiv Core Reference Guide, documentation of the `Massiv::Core::Orientation` template and documentation of the `src/core/property/orientation.h` file, for more information about orientations.

10.13.12. Color

Table 10.17. Color type mapping

Context	**Type**
IDL	color
simple	SColor
in argument	const SColor &
out argument	SColor &
stype	SColor
property	PColor

The color is represents a red-green-blue-alpha color with floating-point components in the range from 0.0 to 1.0. The components are quantized to 8-bit during replication.

10.13.13. Pointers

The following tables show a pointer type that would refer to a managed object of type *`CLASS`*. In the first table it is a *strong pointer*, in the second a *weak pointer* and in the third a *remote pointer*. To learn more about the pointers see Chapter 5, *Pointers*. For quick information about the difference of the three types of pointers, see Table 5.1, “Managed Pointer Types”.

Table 10.18. Strong pointer type mapping

Context	**Type**
IDL	strong_pointer < *`CLASS`* >
simple	Pointer< *`CLASS`* >
in argument	const Pointer< *`CLASS`* > &
out argument	Pointer< *`CLASS`* > &
stype	Pointer< *`CLASS`* >
property	PPointer< *`CLASS`* >

Table 10.19. Weak pointer type mapping

Context	Type
IDL	weak_pointer< `CLASS` >
simple	WeakPointer< `CLASS` >
in argument	const WeakPointer< `CLASS` > &
out argument	WeakPointer< `CLASS` > &
stype	WeakPointer< `CLASS` >
property	PWeakPointer< `CLASS` >

Table 10.20. Remote pointer type mapping

Context	Type
IDL	remote_pointer< `CLASS` >
simple	Remote< `CLASS` >
in argument	const Remote< `CLASS` > &
out argument	Remote< `CLASS` > &
stype	Remote< `CLASS` >
property	PRemote< `CLASS` >

10.13.14. Managed Class Type

Table 10.21. Managed class type mapping

Context	Type
IDL	`CLASS`
simple	`CLASS`
in argument	const `CLASS` &
out argument	`CLASS` &
stype	`CLASS`
property	`CLASS`

The IDL type name of a managed class is its scoped name. If you want to use a class as an array element or a method argument (or nearly anything else but simple member property of another managed class), you should probably define it as a *value type*, because value types can be easily constructed on the stack. See Section 4.3.4, “ValueTypes” for more information.

10.13.15. Pair

Table 10.22. Pair type mapping

Context	**Type**
IDL	pair< *FIRST*, *SECOND* >
simple	PPair< *PFIRST*, *PSECOND* >
in argument	const PPair< *PFIRST*, *PSECOND* > &
out argument	PPair< *PFIRST*, *PSECOND* > &
stype	PPair< *PFIRST*, *PSECOND* >
property	PPair< *PFIRST*, *PSECOND* >

The pair can hold a tuple of properties. In the table, *FIRST* and *SECOND* represent types in IDL syntax, and *PFIRST* and *PSECOND* represent corresponding property types.

10.13.16. Array

This section describes two types of arrays - the *value array* and the *property array*. Both of them will be described below in more detail.

Table 10.23. Value array type mapping

Context	**Type**
IDL	value_array< *TYPE* >
simple	PValueArray< *STYPE* >
in argument	const PValueArray< *STYPE* > &
out argument	PValueArray< *STYPE* > &
stype	PValueArray< *STYPE* >
property	PValueArray< *STYPE* >

The *TYPE* is a type description written in the IDL syntax. The *STYPE* is the corresponding serializable type (*stype*).

value_array< *TYPE* > is an array that can be dynamically resized using a special method. It is able to hold lightweight serializable types.

All objects occupying one concrete array must all be instances of the same type.

This array occupies less memory and is more efficient than the *property_array* (described below) because it accommodates relatively small serializable types instead of much larger properties. Consequently, the value_array cannot be used for types whose corresponding serializable type is a property. The following types can be used as elements of a value array:

- boolean type
- integral types, both with fixed-length and variable-length serialization
- enumeration types
- floating-point types, inluding quantized floats
- string type
- time type
- event handle type
- 2-D and 3-D math vector types
- orientation type
- color type

The IDL parser can itself distinguish which combinations are permitted and which not.

Note

Pointers can't be used as elements of value arrays, even though their *stype* is not a property.

To create a two-dimensional array use a property array of arrays of lightweight serializable types.

Table 10.24. Property array type mapping

Context	Type
IDL	property_array< *TYPE* >
simple	PArray< *PTYPE* >
in argument	const PArray< *PTYPE* > &
out argument	PArray< *PTYPE* > &
stype	PArray< *PTYPE* >
property	PArray< *PTYPE* >

As well as the *value_array* (see above) the *property_array* is a dynamic array. The difference is that the latter stores *properties*, rather than *serializable types*. Therefore it occupies more memory and is less efficient in speed; however the advantage is that the *TYPE* is not restricted as in the value arrays.

10.13.17. Set

Table 10.25. Set type mapping

Context	Type
IDL	set< *TYPE* >
simple	PSet< *PTYPE* >
in argument	const PSet< *PTYPE* > &
out argument	PSet< *PTYPE* > &
stype	PSet< *PTYPE* >

Context	Type
property	PSet< *PTYPE* >

Table 10.26. Multi-set type mapping

Context	Type
IDL	multi_set< *TYPE* >
simple	PMultiSet< *PTYPE* >
in argument	const PMultiSet< *PTYPE* > &
out argument	PMultiSet< *PTYPE* > &
stype	PMultiSet< *PTYPE* >
property	PMultiSet< *PTYPE* >

The set and multi_set types are containers that keep values of given type. In the set, no two elements will be the same, while the multiset can hold more values that are equal (according to their == operator).

In the table above, `TYPE` stands for a type in the IDL syntax, while `PTYPE` stands for corresponding property type.

Please refer to the Massiv Core Reference Guide for more information about sets and multi-sets.

10.13.18. Dictionary

Table 10.27. Dictionary type mapping

Context	Type
IDL	dictionary< *KEY*, *VALUE* >
simple	PDictionary< *PKEY*, *PVALUE* >
in argument	const PDictionary < *PKEY*, *PVALUE* > &
out argument	PDictionary< *PKEY*,*PVALUE* > &
stype	PDictionary< *PKEY*, *PVALUE* >
property	PDictionary< *PKEY*, *PVALUE* >

Table 10.28. Multi-dictionary type mapping

Context	Type
IDL	multi_dictionary< *KEY*, *VALUE* >
simple	PMultiDictionary< *PKEY*, *PVALUE* >

Context	**Type**
in argument	const PMultiDictionary < `PKEY`, `PVALUE` > &
out argument	PMultiDictionary< `PKEY`,`PVALUE` > &
stype	PMultiDictionary< `PKEY`, `PVALUE` >
property	PMultiDictionary< `PKEY`, `PVALUE` >

The dictionary and multi_dictionary types are associative containers that map the `KEY` values to the values of `VALUE`. While the dictionary maps each unique key to a single value, the multi_dictionary can map single key to multiple values.

In the table above, `KEY` and `VALUE` represent types in the IDL syntax, while `PKEY` and `PVALUE` represent corresponding property types.

Please refer to the Massiv Core Reference Guide, module Properties for more information about dictionaries and multi-dictionaries.

11. Classes Generated From the IDL

This chapter gives an overview of classes that are generated by the IDL preprocessor.

> **Note**
>
> To make the IDL preprocessor process a specific IDL file, you first have to add a relevant entry into the `idl.list` file (see Section D.2, "The idl.list File").

The following classes are generated into your source tree on the basis of the IDL. Let's suppose we have a managed class named simply `X` and that the C++ sources of the class are in `x.h` and `x.cpp`, respectively.

Table 11.1. Classes generated on the basis of the IDL

Class [a]	**File** [b]	**Notes**
`class MetaObjectImpl<X>`	`x_generated.h`	*Metaobjects* store all the meta-data about the class, such as the base class, etc.
`class ObjectFactoryImpl<X>`	`x_generated.h`	*ObjectFactories* enable mainly to instantiate objects and their replicas.
`RPCStubs<X>`	`x_rpc.h`	See below.
Other RPC helper classes [c].	`x_rpc.h`	See the Massiv Core Reference Guide.

[a] Class name without the `Massiv::Core` part.
[b] The name of the file that will be generated for the relevant class. It will be located in the same directory as the IDL file.
[c] For example implementation of the RPC packet for each method that is described in the IDL.

The RPC related generated classes are described right below; for information about metaobjects or object factories see the following chapter.

Because the application shouldn't use most of the generated objects explicitly (except for metaobjects), this book dosn't describe them in detail. For more information about either of them please refer to the Massiv Core Reference Guide.

11.1. RPC Related Classes Generated

This section gives a overview about what basic classes are generated by the IDL parser according to the relevant IDL description. It doesn't intend to provide a complete list of methods of the classes nor a full list of the classes, because the Core user generally doesn't need to work with most of them directly.

11.1.1. `RPCStubs` Objects

Each managed object has its own `RPCStubs` object assigned. This class declaration is generated on the IDL basis into *`X`*`_rpc.h` and the relevant implementation into *`X`*`_generated.cpp`, where *`X`* stands for the name of the idl file without the extension.

The `RPCStubs` object for any managed class can be obtained by dereferencing the relevant remote or weak pointer. It provides interface for calling the methods of the associated class via RPC.

The exhaustive information about handling the RPC mechanism can be found in Chapter 8, *Remote Procedure Call*. The following paragraphs only briefly summarize this inforamation.

The `RPCStubs` objects contain two categories of methods:

1. *Helper methods*

   These methods returns the *self* reference and thus can be chained for the programmer's convenience.

   - `param( flags, time )` enables to set the RPC parameters. `flags` can specify besides others an immediate or delayed RPC call mode. For setting the delay serves the second parameter.
   - `request_replica( timeout )` requests replication of the target object (in case the pointer the programmer has used for obtaining the stub is remote) for `timeout` seconds at the local node.
   - `optimize_replica( timeout )` requests replication of the target object the same way as the previous method. Moreover it allows the *optimized replica SRPC*. See Section 8.5.2, “Synchronous RPC Optimizations” for more information.

2. *Methods for performing RPC calls*

   Two `RPCStubs` methods is generated for each method that can be called using the RPC (i.e. that has its description in the IDL). Let's denote *`X`* the name of this original method.

   - `async_`*`X`*`( .. )` performs the asynchronous RPC. The signature of this method is the same as the original one except for that the *out* parameters are removed. See Section 8.3, “Asynchronous RPC” for more information.
   - `sync_`*`X`*`( .. )` performs the synchronous RPC. The signature of this method is the same as the original one (the one we are calling via the RPC). See Section 8.4, “Synchronous RPC” for more information.

   Note that if a class `X` is inherited from a class `Y`, the `RPCStubs< X >` is generated so that it inherits from `RPCStubs< Y >`. Therefore the programmer can use a stub object of a class even for RPC calls of methods contained in all of its antecedents.

12. Metaobjects

12.1. Overview

As mentioned in previous chapters, the Massiv Core needs information about classes and objects that are not provided by the C++ language itself to implement object serialization, replication, migration, RPC and other features. All *managed objects* must be described in IDL files, which are processed by the **factgen.pl** utility, that generates (besides others) source code of *metaobjects* - objects that provide all necessary information about managed classes, their inheritance, properties and methods.

While metaobjects are used extensively by the Core, a large part of their API is public and can be used by the programmer to perform *object introspection* (obtaining information about objects and their classes at run-time).

For each managed class defined in the IDL, also the two following C++ classes are generated:

- A class that implements `MetaObject` interface. It provides the *object introspection*.
- A class that implements `ObjectFactory` interface. It allows you to create object instances.

Metaobjects are singletons - single instance of `MetaObject` and `ObjectFactory` is created for each managed class known to the node during the node startup.

Both `MetaObject` and `ObjectFactory` are briefly described in this chapter. However, only an overview of metaobject structures and methods is provided. Please refer to the Massiv Core Reference Guide for in-depth information about the interfaces.

12.2. `MetaObject` class

`MetaObject` provides the following information about classes and objects:

- Class name and type information
- Inheritance information
- Description of class properties
- Description of class methods

12.2.1. Obtaining a `MetaObject`

Obtaining a pointer to a metaobject describing the most derived class of a local object is simple - just call the `get_metaobject()` method of the object.

Otherwise you have to use the `ClassManager` singleton and use one of its `get_metaobject()` methods as illustrated in this example:

Example 12.1. Obtaining a `MetaObject`

```
using namespace Massiv::Core;

/* Get reference to the class manager. */
const ClassManager & class_manager = Global::class_manager();

/* Get metaobject by fully-qualified class name. */
const MetaObject * const m1 = class_manager.get_metaobject( "Foo" );

/* Get metaobject by C++ class type info. */
const MetaObject * const m2 = class_manager.get_metaobject( typeid( Foo ) );

/* Get metaobject by class type id. */
const WeakPointer< Object > object = ...;
const ObjectId & object_id = object.get_object_id();
hope( object_id.is_object_info_available() );
const ClassTypeId class_type_id = object_id.get_class_type_id();
const MetaObject * const m3 = class_manager.get_metaobject( class_type_id );
```

The example should be pretty self-explanatory, except for the “get metaobject by class type id” part, which illustrates how a metaobject pointer can be obtained from an object pointer to a possibly remote object. In that case we must extract a *class type id* from its object id. Unfortunately some object ids may be missing the class type information, if the referenced object is remote. This may happen when an object id referencing an object of an *alien* kind is archived and then loaded back, for example. You can assume that the class type id is present when you know that the pointer you have extracted the id from never migrated, or migrated only between nodes of the same type (nodes supporting the same set of class kinds). To make it short and simple: you can use this approach on server nodes if the object pointer was retrieved from a server-only object.

None of the `get_metaobject()` methods that were mentioned so far returns `NULL`. They all throw the `Lib::UndefinedClassTypeException` exception if no metaobject is available for the given class. This may happen even in the third version (getting the metaobject by the class type id) if the object id references an object of an *alien* class kind. For each `get_metaobject()` method variant a corresponding method called `get_metaobject_nothrow()` exists, which never throws an exception and returns `NULL` on error instead.

12.2.2. Passing Pointers to `MetaObject` Methods

In the `MetaObject` API, pointer-to-object arguments are often used. However, they are usually not object pointers as described in Chapter 5, *Pointers*. Instead, these two pointer types are used:

- `PtrToObject` is a lightweight pointer to `Object`. Unlike simple `Object *`, it ensures that the referenced object will not be destroyed by the garbage collector, i.e. the pointer *pins* the object until it stops referencing it. To convert an object pointer to the `PtrToObject`, use the following code:

  ```
  Pointer< Object > object = ...;
  const PtrToObject ptr_to_object( object.dereference() );
  ```

- `VariantPointer` is similar to simple `void *`. If debugging is enabled, it remembers the type of the pointer it has been initialized with, and allows you to cast it only to the pointer of the exactly same type. The following example constructs a `VariantPointer` and then casts it back to the correct type:

```
Foo * const native_foo = ...;

VariantPointer variant_foo( native_foo );
/* variant_foo now knows it points to Foo *, regardless */
/* of the most derived class type of *native_foo. */

/* This is the only correct variant_cast that may be performed on variant_foo. */
Foo * const back_to_foo = variant_cast< Foo * >( variant_foo );

/* This fails in debug builds, unless Bar is typedef to Foo. */
Bar * const error = variant_cast< Bar * >( variant_foo );
```

  The variant pointers are used to pass pointers to object *components* (see Section 12.2.3.3, “Class Inheritance”). Most methods of metaobject for class `Foo` require that all variant pointers passed to them are of type `Foo *`. You can use methods `downcast()` and `upcast()` of the `MetaObject` cast pointers (and create corresponding variant pointers) at run-time, without any compile-time class type knowledge. The easiest way to get a variant pointer to the *most derived class* of given object, use the following code:

```
Pointer< Object > object = ...;

VariantPointer complete_object( object.get_complete_object() );
/* If the most derived class of the object is Foo, complete_object is */
/* now of type Foo *, even if the object pointer previously */
/* refered to some of its ascendants. This is the variant pointer most */
/* methods of the metaobjects expect. */
```

12.2.3. Class Information

This section describes how to obtain information about a class, such as class name, class type info, class attributes and inheritance hierarchy.

Note

From now on any method names refer to methods of the `MetaObject` class, unless specified otherwise. “The class” refers to the class described by the metaobject we're talking about.

12.2.3.1. Class Name and Type Info

To get fully-qualified name of a class, use the `get_class_name()` method. The methods `get_class_type_info()`, `get_class_pointer_type_info()` and `get_class_type_id()` return information about the class type.

12.2.3.2. Class Attributes

The `get_attributes()` method returns a reference to a structure containing values of all class attributes. You can also query value of an attribute *`foo`* using a `get_attribute_`*`foo`*`()` method. For complete list of class attributes defined in `object.idl`, refer to Section 10.9.3, “Class Attributes”.

12.2.3.3. Class Inheritance

In the metaobject terminology, a *component* refers to any class inherited, directly or indirectly, by a given class, including the class itself. For example, if `Foo` inherits classes `Bar` and `Baz`, `Bar` inherits `Xyzzy` and `Fyzzy` and `Baz` inherits `Xyzzy` and `Buzzy`, these are all *components* of `Foo`:

- `Foo` itself.
- `Bar`, inherited directly.
- `Baz`, inherited directly.
- `Xyzzy`, inherited via `Bar`.
- `Fyzzy`, inherited via `Bar`.
- `Xyzzy`, inherited via `Baz`.
- `Bazzy`, inherited via `Baz`.

The method `get_component_infos()` returns a reference to a constant vector (in the STL sense) of `ComponentInfo` structures, describing all *components* of the class. For each component, information about its inheritance and pointer to the relevant metaobject describing the component is provided.

Several variants of the `upcast()` method are offered, each of them converts a variant pointer to refer to a specified component. The method `inherits()` checks if a class inherits a class specified by given C++ type info.

12.2.3.4. Example

The following example prints class-related information about a given object.

Example 12.2. Class introspection

```
using Massiv::Core;
Pointer< Object > object = ...;
const MetaObject * const metaobject = object->get_metaobject();
hope( metaobject );

/* Get fully-qualified name of the most derived class of the object. */
std::cout << "class of the object is "
          << metaobject->get_class_name()
          << std::endl;

/* Get value of a class attribute. */
std::cout << "the class is "
```

```
          << metaobject->get_attribute_abstract() ? "" : "not"
          << " abstract"
          << std::endl;

/* Print information about direct and virtual base classes. */
std::cout << "inherits: " << std::endl;
const MetaObject::ComponentInfos & components( metaobject->get_component_infos() );
MetaObject::ComponentInfoIterator it;
for( it = components.begin(); it != components.end(); ++it )
    {
    const MetaObject::ComponentInfo & component = *it;
    if( component.is_parent() )
        {
        /* The class directly inherits the component. */
        std::cout << "    " << component.name << std::endl;
        }
    else if( component.virtual_component )
        {
        /* The class inherits the component virtually. */
        std::cout << "    virtual " << component.name << std::endl;
        }
    /* Other components are ignored here, they are inherited indirectly */
    /* via other components this code prints out. Their component.name */
    /* is not a simple class name, it contains complete "path" */
    /* to the component from the most derived class. */
    }
```

12.2.4. Properties

Each `MetaObject` keeps information about all properties of the class it describes, both defined in the class and inherited from base classes.

12.2.4.1. Property Information

The method `get_property_infos()` returns a reference to a constant vector of `Property-Info` structures. Each structure describes a single property - its name, type (as a string in the IDL syntax), reference to a component the property is defined in and a structure with values of all property attributes.

The method `find_property_infos()` can be used to retrieve information about properties matching given wildcard pattern. To obtain information about a property of an object when given a pointer to the property (see below) and the object, use one of the `find_property_info()` meth-od variants.

12.2.4.2. Getting Property Pointers

Given a pointer to a local object and an identification of a property of the object, the pointer to the property can be obtained using the `get_property()` method. To retrieve pointers to multiple properties of a given object matching a given wildcard pattern, use the `find_properties()` method.

12.2.4.3. Performing an Operation on a Set of Properties

To perform an operation specified by the `Property::PropertyOperation` functor on all properties of a given object, call the `for_each_property_do()` method.

12.2.4.4. Example

The following example prints types, names and values of all properties of a given object.

Example 12.3. Property introspection

```
using Massiv::Core;
Pointer< Object > object = ...;
const MetaObject * const metaobject = object->get_metaobject();
hope( metaobject );

/* Cast to pointer referencing the most derived class of object. */
/* Many MetaObject methods that access the object require this. */
/* This is a shortcut for metaobject->downcast(); */
VariantPointer complete_object( object->get_complete_object() );

/* Print info about all object's properties. */
const MetaObject::PropertyInfos & properties( metaobject->get_property_infos() );
MetaObject::PropertyInfoIterator it;
for( it = properties.begin(); it != properties.end(); ++it )
    {
    const MetaObject::PropertyInfo & info = *it;
    /* We could simply print complete description of the property in */
    /* the IDL syntax using the operator<< of the info object here. */

    /* Print property type in IDL syntax and property name. */
    std::cout << info.type << " " << info.name << " = ";

    /* Get pointer to the property. */
    /* Because complete_object points to the most derived class of */
    /* the object, the get_property_from_complete_object() method can be used. */
    /* Otherwise a bit slower but more generic get_property() would */
    /* have to be used. */
    const Property * const property = metaobject->get_property_from_complete_object
        ( complete_object, it );
    hope( property );

    /* Print value of the property. */
    /* Each property defines operator<< which prints its value */
    /* in a format easy-to-read for a human being. */
    /* If we wanted to print the value in textual serialization format, */
    /* we would have to call property's get() method or use a TextWriter */
    /* object bound to std::cout and call the text_write() serialization */
    /* method of the property. */
    std::cout << *property << std::endl;
    }
```

12.2.5. Methods

The `MetaObject` can be also used to obtain information about methods defined in the class. Note that no information about inherited methods (methods defined in base classes) is kept in the `MetaObject`. However, several methods automatically search metaobjects of base classes to provide information about all class methods (including the inherited ones).

12.2.5.1. Method Information

The method `get_method_infos()` returns a reference to a constant vector of `MethodInfo` structures. Each structure describes a single method of the relevant class - its name, description of arguments and a return type, and a structure with values of all method attributes. For each method argument, information about its name, type (as a string in the IDL syntax) and the calling semantics (called *argument kind* in the structure) is provided.

The method `find_method_infos()` can be used to retrieve information about methods matching a given wildcard pattern, including methods defined in ascendant classes. Given a wildcard pattern and a pointer to a local object, `find_methods()` finds all methods matching the pattern (including inherited ones), and returns information about the methods, including pointers to metaobjects of components the methods are defined in and correctly casted variant pointers to the components.

12.2.5.2. Dynamic Local Calls

The `MetaObject` allows you to call any IDL-described method of a local object dynamically (i.e. construct the call at run-time). Use one of the overloaded `call_method()` methods to perform such a call. The method to be called can be identified in many ways, arguments are specified in their textual serialization form, separated by whitespace. There is no way to obtain results of a dynamic local call.

The `MetaObject` of the class the method is defined in must be used when calling `call_method()`. `MetaObject` describing a derived class can't be used to perform the call. The call expects a variant pointer referencing the component of the type the metaobject describes. The `find_methods()` may provide useful to obtain pointer to the correct metaobject and correctly casted variant pointer. To simplify things a bit, several `call_methods()` methods are provided. These methods allow you to easily call all methods matching a given wildcard pattern on a local object, including methods defined in base classes.

Note

The dynamic local call API is not very nice and it does not allow you to obtain call results. You may want to use dynamic RPC even if the callee object is local.

12.2.5.3. Dynamic RPC

Probably the most useful part of the method-related API of the `MetaObject` class is the *dynamic RPC*. It allows you to construct and perform a remote call at run-time. In many ways it is similar to the dynamic local call API described above, but it's easier to use and allows you to examine call results.

Use one of the overloaded `remote_call_method()` methods to call a method of an object (remote as well as local) dynamically. Check Section 8.5.5, “Dynamic RPC” for detailed description of the dynamic RPC.

12.2.5.4. Example

The following example prints types, names and values of all properties of a given object.

Example 12.4. Method introspection

```
using Massiv::Core;
Pointer< Object > object = ...;
const MetaObject * const metaobject = object->get_metaobject();
hope( metaobject );
VariantPointer complete_object( object->get_complete_object() );

/* Unlike property infos, method infos do not contain descriptions */
/* of methods defined in base classes. We must examine all components */
/* manually. Because components contain information about the most */
/* derived class too, we don't have to treat it as a special case. */
const MetaObject::ComponentInfos & components( metaobject->get_component_infos() );
MetaObject::ComponentInfoIterator c_it;
for( c_it = components.begin(); c_it != components.end(); ++c_it )
    {
    /* Iterate over all methods of the component. */
    const MetaObject * const component_metaobject = c_it->metaobject;
    hope( component_metaobject );
    const MetaObject::MethodInfos & methods
        ( component_metaobject->get_method_infos() );
    MetaObject::MethodInfoIterator m_it;
    for( m_it = methods.begin(); m_it != methods.end(); ++m_it )
        {
        /* Print complete description of the method in IDL syntax. */
        /* We could also reimplement this manually by printing all */
        /* attribute values, argument declarations and the return type. */
        const MetaObject::MethodInfo & info = *m_it;
        std::cout << "method "
                  << info.attributes
                  << " "
                  << info
                  << ";"
                  << std::endl;
        }
    }
```

12.2.6. Massiv Core and Demo Examples

Power of the object introspection is demonstrated in the implementation of the console. The console allows you to read and write to all properties of local and remote objects, even to properties of member objects, elements of arrays and dictionaries, and properties of objects pointed to by migration-group pointers (i.e. objects owned by the same node as the object owning the pointer). You can also call methods of both local and remote objects, create and destroy objects, force migration of objects, etc. The console provides the bash-like command completion in almost every context regarding an object local to the node the console is connected to. The implementation of the command parsing and the completion is in the `src/demo/lib/shared/console.cpp` file, the commands themselves are implemented in many files, probably the most interesting is `src/demo/lib/shared/node_console_commands.cpp`.

Another nice example of the introspecition usage is the implementation of the `operator<<` of the `MetaObject`, which generates description of the class in IDL syntax. It is more simple and easier to understand than the console implementation. You can check it out in `src/core/object/metaobject.cpp`.

12.3. `ObjectFactory` Class

Object factories are special metaobjects that are used to instantiate and initialize managed objects (including its properties).

They are used by the Core to either instantiate new objects, migrated objects or object replicas. You can use them to create new objects too. However, using the `CreateObject` helper class as described in Section 4.3.2, “Instantiation and Finalization” is the recommended way to instantiate new objects.

One special object factory class is generated for each managed class; all of them inherit from `Massiv::Core::ObjectFactory`.

The object factories need an information about the inheritance hierarchy of their associated object to be able to properly initialize it. They also need information about its properties, etc. Therefore, each object factory keeps a reference to the metaobject associated with the same class as the relevant factory. That's one of the reasons why metaobjects are essential in the Massiv. An object factory for any object can be obtained from the `ObjectManager`.

The following list summarizes public methods contained withing an object factory class:

- `create_object()` creates a new instance of a relevant managed class.
- `create_object( object_id )` creates an instance of the relevant managed class using a specified `object_id`. This is useful for example when creating objects from archive (the user code will never create objects from archives itself, but despite that there might exist a reasonable usage for this method in the user code).
- `create_replica( object_id )` creates replica of the relevant managed class. The given `object_id` must be the same as of the original objects. The user code typically won't use this method because there exist more comfortable methods of requesting replication. See Chapter 7, *Replication* for more information.
- `clone_object( object )` makes a duplicate of the given object using the shallow-copy technique (i.e. there won't be shared properties between the two objects, etc).
- `raise_exception( object )` throws an `object` as an exception by value.

13. Creating Managed Class

13.1. Overview

This chapter provides a step by step tutorial, that will show you how to create a new library containing a simple managed class. The complete sources of this example are stored in `src/example/server_lib`. They are also listed in Appendix F, *Example Listings*.

We will show how to create a new library that will be built using the Massiv build tool **mkgen.pl**. Then we will create a simple "hello world"-like example. For educational puposes it will be split into two managed classes - an interface and an implementation. Both IDL description (see Chapter 9, *Introduction to IDL*) and C++ implementation (see Chapter 4, *Managed Objects* and other chapters) of the classes will shown.

You may also want to read Section D.1, "Massiv Build Tools". It contains a description of the build process in general, and a reference guide for the tools used to build a library or an application.

The example is split into an interface and an implementation for simple reasons: to show an example of class inheritance, attribute inheritance, description of virtual methods and other features. However, it may be useful to provide forward compatibility of the library too. If the implementation of the `Hello` class changed drastically, the Massiv Core would not be able to read archives containing an old version of the class. The Massiv Core does not support any form of class versioning directly. So instead of incompatible changes to existing classes, you should implement brand new classes that implement the same (and stable) interface (`ShinHello`, followed by `HelloNisei` and `ShinHelloNiseiZ` in the future, for example). Then you can either write a special code that converts old object versions to new ones, or just leave the old objects there if their old behavior is not harmful.

Note

The sources in the `src/example/server_lib` directory contain special comments that are used to automatically generate example copies in this document. You should ignore them. Also the line numbers in examples in this document are not the same as line numbers in the sources: the special comments as well as the first part with copyright and license information are not included in the examples.

Note

This relatively simple tutorial does not demonstrate all features of the Core. The features not used in this example include complex property data structures, usage of pointers, replication, migration and replication groups, usage or implementation of special objects such as node objects and account objects, heavy use of RPC and more. You will have to check the sources of the Massiv Demo for more complex examples.

13.2. Creating a Library

For each independent library, a new empty directory should be created. In this example, **mkgen.pl** will be used to create platform-specific makefiles that are then used to build the library. You can use any other build tool, but it's recommended to stick to **mkgen.pl**, because it supports sources that are generated at compile-time and can determine dependencies of those files.

The **mkgen.pl** tools parses simple script files in each directory and creates makefiles for specified platforms. The script files are called `makefile.gen`. This section describes all important parts of the `src/example/server_lib/makefile.gen` file:

Every `makefile.gen` describing a library with managed classes should begin like this:

```
# "Creating Managed Class" example.
# mkgen.pl will generate makefiles for selected platforms from this file.

# Required for all C++ projects.

STDCPP

# Include makefile.gen part generated by genmkgen.pl from idl.list.

include makefile_idl.gen
```

`STDCPP` tells **mkgen.pl** that it should link projects in this directory with the system C++ libraries. The `include makefile_idl.gen` line includes part of `makefile.gen` that will be automatically generated by the **genmkgen.pl** tool as described in the next section.

Global `makefile.gen` commands follow. These commands apply to all projects built in this directory:

```
# Directories to include from.

includes = . ../../core

# Precompile the massive core.h header if the compiler supports it.

precompile = core.h
```

The `includes` assingment sets list of directories to search when C++ files include other files. It's used both by **mkgen.pl** when it interprests the `#include` directives, and it's passed to the compiler by makefiles generated by **mkgen.pl**. The `precompile` assignment specifies that the `core.h` header should be precompiled, if the target platform supports it.

The last section of `makefile.gen` describes how to build the library:

```
# Create a library in this directory, containing objects compiled from
# all C++ sources, including the generated sources. This library will
# reference symbols from the Massiv Core shared library.

library example_lib_server
    sources = *.cpp
    sources += @GENERATED_SOURCES@
    shlibs = ../../core/massiv
endlibrary
```

The library will be called `example_lib_server`. It will be built from all C++ sources that are in the directory, plus all generated sources. The global `@GENERATED_SOURCES@` variable is set in `makefile_idl.gen`. The library will reference symbols from shared library called `massiv` built in the `src/core` directory.

13.3. Adding New IDLs to a Library

To build a library with managed classes, descriptions of the classes are required. These descriptions are written in special IDL files as described in sections below. To process the IDL files, **mkgen.pl** must know their names, names of files that are generated from the IDL files, and dependencies of those files. This information is automatically generated by the **genmkgen.pl** tool, which reads the `idl.list` file and generates `makefile_idl.gen`, which is then included into `makefile.gen`.

IDL files usually require (import) other IDL files, and some of them may belong to other library. These dependencies are described by the `depends` directives in the `idl.list` files:

```
# We use managed classes from the Core library.

depends ../../core
```

In this case, some IDLs from the `src/core` directory are required. All directories specified by the `depends` directive must contain an `idl.list` file. Note that the `depends` directive must be also used when a C++ source includes a source generated in an IDL file that is part of the `idl.list` in the specified directory. All required directories must be listed, `depends` won't be searched recursively.

List of all IDL files that belong to the library must be specified in the `idl.list` using the `idl` directive:

```
# List of files containing descriptions of managed classes.

idl hello_interface.idl
idl hello.idl
```

This example contains two classes, interface and implementation, each described in its own IDL file.

13.4. Interface Class

`HelloInterface` is an abstract class with a single method (`hello()`) and no implementation.

13.4.1. IDL Description of `HelloInterface`

Let's begin with the IDL description of the `HelloInterface` class.

The `#import` directive is similar to C++ `#include` with standard `#ifdef` guards - the specified file will be included only if has not been included yet. The `core.idl` file includes all public IDL files from the Core:

```
#import "core.idl"
```

All classes in this example are defined in the `example` namespace:

```
namespace example {
```

IDL class description uses syntax similar to C++ class definitions:

```
class
    <
    abstract,
    kind = SERVER,
    root
    >
    HelloInterface : ::Massiv::Core::Object
```

The key differences are:

- Values of special attributes may be set per-class. This is a generic way how to specify additional information about a class, similar to various compiler-specific C++ ways to specify such information (`__declspec`, `__attribute__`, etc.). In this case the class is *abstract* (objects of the class can't be instantiated), garbage collector *root* (see Section 5.4, “Garbage Collector”) and it's *kind* is `SERVER` - objects of the class can be owned only by server nodes, client nodes don't know anything about the class. This may seem a bit weird, but remember that client nodes are not allowed to call methods of any objects but their account object.
- Only the *public* inheritance is supported. Otherwise the inheritance specification is the same as in C++ - there may be multiple base classes and virtual inheritance can be used. The `Massiv::Core::Object` class is root of the hierarchy of managed classes and all managed classes must include it exactly once (so you should include it virtually if you intend to use multiple inheritance).

The class body follows. It may contain description of nested classes and enumeration types and description of class properties and methods:

```
    {
    method< virtual > hello
        (
        in string callee_name
        ) : string;
    }
```

The `HelloInterface` class is really simple. It contains single method called `hello()`. The method is *virtual* (this is specified by a method attribute), single string argument is passed to the method and it returns another string (Pascal-like syntax is used to specify the return type).

13.4.2. C++ Definition of `HelloInterface`

The C++ defintion of the class begins with standard `#includes`:

```
#ifndef MASSIV_CORE_H
#include "core.h"
#endif

#include <string>
```

The `core.h` header includes nearly all public headers from the Massiv Core that you can ever need. We precompile the `core.h` header on platforms that support that, therefore it should be included as the first headers everywhere (because most compilers only support precompiling of the first header). The standard `string` header is included too.

The C++ definitions must be in the same namespace as their IDL descriptions:

```
namespace example {
```

The definition of the class `HelloInterface` follows:

```
/**
 * Interface of hello classes.
 *
 * This is generic interface of all classes that implement the
 * hello() method.
 *
 * As described in the IDL, all Hello classes should be root objects
 * of KIND_SERVER kind.
 */
class HelloInterface : public ::Massiv::Core::Object
    {
public:

    /**
     * The "say hello" method.
     *
     * It should generate a hello message for the callee @a callee_name,
     * print it anywhere it wants and return it.
     */
    virtual std::string hello
        (
        const std::string & callee_name
        ) = 0;

    }; // class HelloInterface
```

As described in the IDL, the `HelloInterface`'s base class is `Massiv::Core::Object` and it has a single method, `hello()`. The method is pure virtual. Note that the IDL does not care that the method is pure (and there is no way to specify that), but it needs to know that objects of the class can't be instantiated, as specified by the *abstract* class attribute. The type of the only argument and the return type of the method follow type mappings described in Section 10.13, “Property and Argument Types”.

13.5. Implementation Class

The `Hello` class implements the `HelloInterface` interface. It remembers its name, number of times `hello()` has been called since the creation of the `Hello` object and the number of times the `hello()` method has been called since the most recent simulation startup.

13.5.1. IDL description of `Hello`

The relevant part of the IDL looks like this:

```
#import "hello_interface.idl"

namespace example {

class
    <
    simulation_startup_notify
    >
    Hello : HelloInterface
    {
    method register_to_naming
        (
        in string name
        ) : bool;

    property string   name;
    property vlint32  total_call_count;
    property vlint32  current_call_count;
    }

} // namespace example
```

As in C++, descriptions of all base classes must be included, otherwise the inheritance specification is not valid.

Only one class attribute is assigned a value directly. Setting the *simulation_startup_notify* attribute ensures that all `Hello` objects will be notified when the simulation starts. Because the *root* and *kind* attributes have the *inherit* semantics, and their value is not directly set, values from the `HelloInterface` class will be inherited; therefore `Hello` will be garbage collector root server-only objects. The *abstract* attribute does not have the *inherit* semantics, so the `Hello` class is not abstract.

Note

The only reason why the *root* attribute is set in the `HelloInterface` class is to make it default for all derived classes. It has no meaning for the `HelloInterface` class itself, because it is declared *abstract*.

The `Hello` class introduces a new method, `register_to_naming()`. It has three properties: `name`, `total_call_count` and `current_call_count`.

13.5.2. C++ implementation of `Hello`

The implementation of the `Hello` class is pretty straightforward, except for a bit weird `register_to_naming()`, which demonstrates usage of *RPC*.

13.5.2.1. Properties

The class has three properties:

```
Massiv::Core::PString   name;
    /**< Internal name of the object.
         It's used to print the hello message and to register
         the object to the global naming service. */

Massiv::Core::PVlInt32  total_call_count;
    /**< Number of calls to hello() since object creation. */

Massiv::Core::PVlInt32  current_call_count;
    /**< Number of calls to hello() since node startup. */
```

Their types correspond to types used in the IDL, as described in Section 10.13, “Property and Argument Types”.

13.5.2.2. Initialization

The `initialize()` method is called automatically on each object being created by the `CreateObject` helper class (described in Section 4.3.2, “Instantiation and Finalization”). It's a replacement of a standard C++ constructor - because at the time the real constructor is called the object is not initialized yet, you cannot modify its properties in the constructor. Unlike constructor, `initialize()` will never automatically call `initialize()` of base classes nor of member objects.

Any `Hello` instance initializes its name in `initialize()`:

```
void Hello::initialize
    (
    const std::string & the_name
    )
    {
    name = the_name;
    }
```

If you wanted to create a new `Hello` object called “name”, the code would look like this:

```
Pointer< Hello > hello
    (
    Massiv::Core::CreateObject< Hello >( "name" )
    );
```

13.5.2.3. Monitoring Object Changes

Because we want to count number of times the `hello` method has been called since the simulation startup, we must reset the `current_call_count` property to zero on each startup. We have set the *simulation_startup_notify* class attribute in the IDL description, so the `object_updated()` method will be called on each startup:

```
void Hello::object_updated
    (
    UpdateReason reason
    )
    {
    if( reason == SIMULATION_STARTUP )
        {
        current_call_count = 0;
        }
    }
```

Note

You can also monitor other changes of an object. See the Massiv Core Reference Guide for a complete documentation of the `Object::object_updated()` method.

13.5.2.4. Hello World

Now we can implement the `hello()` method:

```
std::string Hello::hello
    (
    const std::string & callee_name
    )
    {
    std::ostringstream oss;
    oss << "Hello to " << callee_name << " from " << name << std::endl;
    oss << "(called " << total_call_count << " times, " <<
           current_call_count << " since startup)" << std::endl;
    const std::string s = oss.str();

    total_call_count++;
    current_call_count++;

    Global::log_info( Status::FACILITY_LIB, Status::PRIORITY_LOW, s );
    return s;
    }
```

The method will create a message that greets the callee, and prints the name of the `Hello` object and values of the two counters mentioned above. Then it will increase the count, log the message and return it to the callee.

13.5.2.5. Massiv Demo Naming Service

The last method, `register_to_naming()`, is a bit more complex and demonstrates usage of the *dynamic RPC*. You won't usually have to write a method like this. For example the Massiv Demo does not use the dynamic RPC at all for example.

This method registers the `Hello` object to a naming service object of type

Demo::Lib::NamingService. Because it uses the dynamic RPC, the Hello class can be compiled without any header from the Massiv Demo and the library can be linked to any application, even if it does not implement a naming service at all. The call will fail in that case, destroy the Hello object and return false.

The register_to_naming() method is intended to be run from the console of the Massiv Demo. These commands create a Hello object and register it to the global naming service under the name “hello”:

```
/connect Server1
/createobject example::Hello > $OBJID
/rcall $OBJID example::Hello register_to_naming "hello"
```

If the registration succeeds, anyone can query the global naming service object for the pointer to an object called “hello”, and then use the HelloInterface interface to call the hello() method. If the registration failed, the Hello object is destroyed. Therefore it's safe to run these commands multiple times: exactly one object will remain registered to the naming service and other objects will destroy themselves.

The method expects a single argument - the name of the object. This is required because if an object is created from the Massiv Demo console, its initialize() method is not called:

```
bool Hello::register_to_naming
    (
    const std::string & the_name
    )
    {
    initialize( the_name );
```

We initialize two local variables. The self object pointer points to the Hello object itself. The failed is initialized to true and is set later to false on success:

```
    const WeakPointer< Object > self( this );
    bool failed = true;
```

All real work is enclosed within a try block. If an exception is thrown, we assume that the registration has failed.

```
    try
        {
```

Firstly, we ask the Core for the object id of the naming service object and create a remote pointer to the service object. The object id is stored in the registry in Settings/WellKnownObjectIdDatabase/naming_service_object. The Core defines location of this config variable, but it does not care about the interface of the naming service object at all - that's up to the application:

```
        const ObjectId naming_id = Global::well_known_object_id_database().
            get_naming_service_object();
        const Remote< Object > naming( naming_id );
```

Next, we acquire the metaobject of the Demo::Lib::NamingServer class:

```
const MetaObject * const metaobject = Global::class_manager().
    get_metaobject( "Demo::Lib::NamingService" );
```

We have to serialize the arguments for the register_object() method to a text stream. The expected arguments are:

- Name under which the object should be registered.
- Pointer to the object.
- A boolean flag; false means that the registration should fail if a different object is already registered under the same name.

```
std::stringstream ss;
{
TextWriter tw( ss );
const Serializer::Description desc;
name.text_write( tw, desc );
tw.write_space();
self.text_write( tw, desc );
tw.write_space();
const SBool replace = false;
replace.text_write( tw, desc );
}
```

Then we perform the dynamic RPC call:

```
TextReader tr( ss );
std::auto_ptr< MethodPacket > results = metaobject->
    remote_call_method( naming, "register_object", tr );
```

The register_object() method returns false if the registration failed. Because we don't know the type of the method packet, we must check its contents in their textual form.

```
if( results->get_argument_value( -1 ) == "true" )
    {
    failed = false;
    }
else
    {
    std::ostringstream oss;
    oss << "Object " << name << " already registered "
           " to the global naming service." << std::endl;
    Global::log_warning( Status::FACILITY_LIB,
        Status::PRIORITY_HIGH, oss.str() );
    }
}
```

Log a warning if the registration failed.

```
    catch( std::exception & e )
        {
        std::ostringstream oss;
        oss << "Failed to register " << name << " to naming service: "
            << e.what();
        Global::log_warning( Status::FACILITY_LIB,
            Status::PRIORITY_HIGH, oss.str() );
        }
```

If the registration failed, we destroy the `Hello` object.

```
    if( failed )
        {
        Massiv::System::dispose_gc_root( self );
        }
    return failed ? false : true;
    }
```

13.6. Linking With the Massiv Demo

The library we have created depends only on the Core and can be linked to any Massiv application. For example, to link it to the Massiv Demo, modify it following these steps:

1. Modify `src/demo/lib/server/idl.list`, which is the main IDL list of the Massiv Demo server node application. Add line containing `depends ../../../example/server_lib`, which ensures that the example managed classes will be included in the class list of the Massiv Demo server.
2. Add a line containing `libs += ../../example/server_lib/example_server_lib` to the `program demo_server` section of file `src/demo/server/makefile.gen` to link the Massiv Demo server with the example library.

Then you can either directly instantiate and use objects of the `Hello` class from other objects, or create the `Hello` object and register it to the naming service (from the code or on the console of the Massiv Demo), and then use it from anywhere like this:

```
#include "database/well_known_object_id_database.h"
#include "lib/server/naming_service.h"
#include "example/server_lib/hello_interface.h"

...
const ObjectId naming_id = Global::well_known_object_id_database().
    get_naming_service_object();
const Remote< Demo::Lib::NamingService > naming( naming_id );
Remote< example::HelloInterface > hello( naming->sync_get_object( "hello" ) ).convert();
const std::string result = hello->sync_hello( "callee" );
```

14. Special Objects

14.1. Overview

The Massiv Core provides interfaces for two managed objects, which must be implemented by the application. The former one, derived from `Massiv::Core::NodeObjectInterface`, is used to represent each node in the simulation. The latter one, derived from `Massiv::Core::AccountObjectInterface`, represents account of a client and is used as gateway between objects which resides on the client node and the objects in the simulation.

14.2. Node Object

As you already know, the Massiv simulation is distributed over several nodes. Node objects are special objects that represents these nodes in the simulation. Each node has its node object which resides on that node and is not allowed to migrate.

Each stand-alone managed object has its own unique `ObjectId` which makes the object addressable in the simulation. The same applies for node objects. The only difference is that their `ObjectIds` have a special form that allows the Core to easily distinguish between ordinary object ids and ids representing node objects. When a node object needs to be localized the Core contacts the corresponding node directly.

Once the Massiv Core is properly initialized, it asks the application to create the node object which will represent the local node in the simulation. Because service nodes don't participate in the simulation at all, the application should implement two types of node objects. One for server nodes and one for client nodes. Each node object must be derived from `Massiv::Core::NodeObjectInterface` abstract class.

```
class NodeObjectInterface : public Object
    {
public:
    virtual NodeId::Type get_node_type() const = 0;
    virtual Pointer< AccountObjectInterface > create_account_object() = 0;
    };
```

The `get_node_type()` returns type of the node it represents which should be either client or server node type.

The `create_account_object()` method is used by the Core when it needs to create a new account object. The node object serves here as an account object factory. See Section 14.3, “Account Object” for more information.

14.2.1. Functionality Of Server Node Objects

The Core does not enforce much requirements on the implementation of server node objects. They are utilized by the Core for automatic creation of account objects only, but the application can and often will extend their functionality beyond the requested requirements. They can serve as an application level interface to connected servers, for example.

The Core asks the application to create an account object (defined on the application level) for a new client when it is just being subscribed into the simulation. See Chapter 28, *Handling Accounts* for more information.

14.2.2. Functionality Of Client Node Objects

Client node objects are used by servers to communicate with the corresponding clients on the application level. Because server and client nodes collaborate by using object migration, the server nodes must be able to somehow specify that the destinations of the migrations are client nodes. Objects created on client nodes are not part of the simulation which is distributed over the server nodes and so the server nodes can't know which objects exist on the client nodes and therefore can't address those objects directly. The server nodes always migrate objects to the node objects of the clients. The identifications of the client node objects are always known to all server nodes.

14.3. Account Object

Account objects are special objects which represent accounts of clients subscribed into the simulation. Each client node has exactly one account object. Client node and account object are tightly coupled together. The account object represents gateway from the client to the simulation. Before new client nodes can connect for the first time to the simulation, their account objects must be created.

Because server nodes can't trust any data from client nodes, there are many restriction on communication between client and server nodes for security purposes (see Section 6.10.2, “Migration From Client To Server Nodes”). One of these restrictions is that any client node can interact with the simulation only through its account object. Thus the entire traffic from clients to servers is under control, which leaves only a small gap how clients can possibly (either willingly or by aaccident) break the simulation.

Each client has its own account object. Once the client gets connected to the simulation, disconnects from the simulation, etc., the Core notifies its account object about the event. This allows to hook specific actions associated with the event. Its up to the account object to do whatever is appropriate.

When a client is connecting to the simulation, the Core initializes network connections among the client and other server nodes only. Once the network connections are established, the Core leaves all other communication on the account object. The account object usually does application specific actions. For example, if the application was some RPG game, the account object would probably wake the client's hero up (which is another managed object) and put him into the simulated world, so that he could fight another horde of monsters.

Client nodes can migrate objects to their account objects only. If the rule is violated the Core disconnects the client from the simulation.

Each node object must be derived from `Massiv::Core::AccountObjectInterface` abstract class.

```
class AccountObjectInterface : public Object
    {
public:

    virtual void client_connected
        (
        const WeakPointer< NodeObjectInterface > &  node_object
        ) = 0;

    virtual void client_disconnected() = 0;

    virtual bool check_client_rpc_request
        (
        const Pointer< RPCObject > & request
        ) const = 0;
    };
```

The `client_connected()` method is called by the Core when the client with this account gets connected to the simulation. The Core calls the method using asynchronous RPC. *`node_object`* is a pointer to the node object representing the client's node. Account objects usually remember this pointer so that they could communicate with the connected client's node.

The `client_disconnected()` is called by the Core whenever the client of the account object gets disconnected from the simulation. There are many reasons why the client can be disconnected. For example the client intentionally quit from the simulation or network connection to the client was lost etc. Account objects usually forget the reference to the client's node object (that was remembered when `client_connected()` had been called) as no more messages can be delivered to the disconnected client.

Note that calls to `client_connected()` and `client_disconnected()` don't need to be properly paired. The Core ensures that for one `client_connected()` there will be at least one call to `client_disconnected()`, but there may be more calls to the `client_disconnected()` and so the account objects should count on it.

The `check_client_rpc_request()` is used to verify `RPCObjects` received from the client. When delivering a RPC request from the client to an object on a server, the Core checks whether the RPC callee inherits from `AccountObjectInterface`) and then calls this method which should perform additional, application specific, tests. Their purpose is to check that properites of the `RPCObject` are valid and that the client is not trying to call a "private" method, etc.

14.3.1. Account Objects in Massiv Demo

In the Massiv Demo, which is a sample application showing how the Massiv should be used in practice, the account objects serves for many purposes. This section gives you a simple overview how the account objects works in the Demo and how they can be used in real applications.

- *Player control*

  In the Demo each client has its own player. The player is an entity which can be moved around the map. Because the client can't communicate with its player directly, the account object receives input from the client and forwards it to the player which is able to react to the client's orders.

- *Replication*

  The account objects manage replication of the simulated world to the client. Once the client gets connected to the account object and starts controlling its player entity, the account object starts replicating an area around the player (this inludes surrounding map and nearby entities like sheep). Also the inventory with all items the player has is replicated to the client so it can manipulate with the inventory as well.

- *Console commands*

  The account object gives the client ability to execute some commands. The commands are sent by the client to the account object in a textual form. The commands are called console commands because the client's application provides a console, which can be used by the client to enter commands.

  The console commands sent by the client are executed either directly by the account object or are forwarded to some node object (the console is "connected to"). The account object is always linked with some node object. If a received command is not recognized by the account object, it forwards the command to the node object the account object is linked with.

  Some commands are accessible only for administrators. Whether a client is an administrator is determined by its account object. In the Demo there are two types of account objects: *normal* and *root*. The *normal* account object supports only some of all available commands. The *root* account object supports all commands.

  The account object supports commands which manipulate the client's player entity. For example the client can change name or chat color of the player. Other commands supported by the account object include commands for modifying elevation of the simulated map. These commands can be executed only by *root* account objects. Administrators can use these commands to modify map of the simulated world (although the Demo provides map editor which can be used to manipulate the map with few mouse clicks so there is no reason why to modify the map via account commands).

Part III. Writing Application Over Massiv

This part guides you through what you need to know to start writting a real application using the Massiv. This part contains all the marginal issues that are mostly not related to the substantial parts of the Core, but you, as the Massiv developer, can't do without them (or at least they might be quite useful to you).

Table of Contents

15. Massiv Application Skeleton

15.1. Overview

By now you should already be familiar with the distributed object model used in the Massiv. While the knowledge of the model is essential and allows you to write the logic of the distributed application, it does not give you any information about how to configure, setup, control and run the Core. This chapter will focus on these topics.

The Core can be controlled by utilizing two main classes: `Massiv::System` and `Massiv::Core::Global`. These are singleton objects with static methods. While the `System` class allows to control the Core in a more abstract way, the `Global` class enables to get access to concrete Core subsystems and allows a lower level access. Some of the accessed subsystems are private to the Core and need not (must not) be used by the application at all. See Chapter 16, *Global Objects And System Interface* for more information about the global objects.

Besides the *application-to-Core* communication, the Core sometimes needs to upcall/callback the application. The upcalls are either on the *object level* (special methods of managed objects are upcalled from the Core; mainly the migration deliveries and upcalls on special objects, see Chapter 14, *Special Objects*) or on the *application level* (they are global to the simulation). For example when a node connects to the simulation, the Core asks the application to create an *object level* representation of the local node (the application's response should be instantiation of an object derived from `Lib::NodeObjectInterface`, see Section 14.2, “Node Object”). Also since the model has been designed in such way that it is the Core that runs, the Core provides a *next tick* callback called at the end of each tick. This allows to perform additional application specific work that does not relate to the simulation directly, like redrawing the screen, handling keyboard input, etc.

As the application runs the system undergoes state changes. The states are called the *system phases*. It is strictly specified what can and what cannot be done in each particular system phase. When the application starts, the system is in the *uninitialized* state. The states can be changed using various calls to methods of the *System* class. Note that the Core can not be used if it is not initialized. The following diagram shows the system states, the transitions among them and the corresponding calls that cause the state changes:

Figure 15.1. System Phases

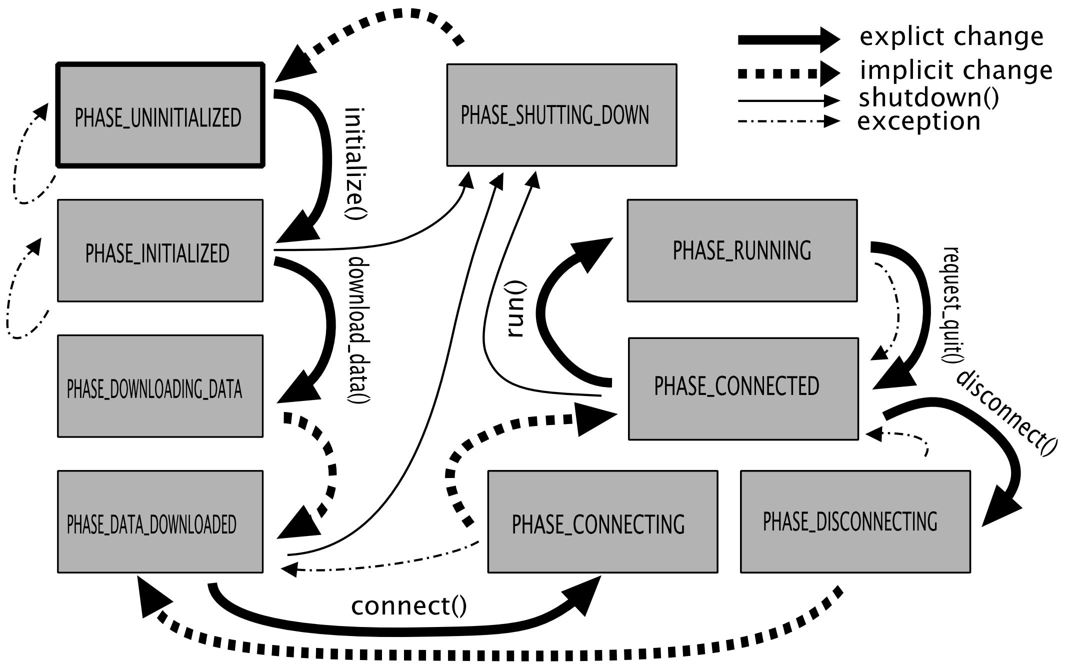


Table 15.1. System Phases

Phase	Semantics
`PHASE_UNINITIALIZED`	The Core is uninitialized and can not be used. This is the initial state.
`PHASE_INITIALIZED`	The Core has been initialized. It runs in an *anonymous* mode.
`PHASE_DOWNLOADING_DATA`	The Core is downloading minimal data required to start regular operation.
`PHASE_DATA_DOWNLOADED`	The minimal data was downloaded. The Core is ready to change its identity and register to the simulation.
`PHASE_CONNECTING`	The node is connecting / registering to the simulation using its real identity.
`PHASE_CONNECTED`	The node has connected to the simulation.
`PHASE_RUNNING`	The simulation is running.
`PHASE_DISCONNECTING`	The node is disconnecting from the simulation and changes its identity back to *anonymous*.
`PHASE_SHUTTING_DOWN`	The Core is shutting down and deinitializing itself.

Application's main function usually does the following steps. The steps will be discussed in the sections below in detail:

- Initializes the Core
- Downloads minimal prerequisite data
- Connects to the simulation
- Starts the main loop

- Disconnects from the simulation
- Shutdowns the Core

15.2. Initializing the Core

When the application starts the Core is uninitialized and must be explicitly initialized by the programmer. The initialization process is responsible for proper Core start up which includes instantiation of the requested global objects, their registration and initialization. To simplify the process and shadow the programmer off the initialization internals and subsystem dependencies as much as possible the Core defines a `Massiv::System::StartUpInfo` structure that drives the initialization process. As a result the Core can be started by a single call to the System class.

The `StartUpInfo` structure contains information that describe what subsystems will be required by the application and optional parameters that can affect the operation of that subsystems.

Note

Since the Massiv is an extensible pluggable system and not all applications generally require all of its subsystems, there is a way to specify what subsystems should be started. The system then resolves the start up dependencies (what already must be and must not be running when a specified subsystem is being started) and ensure the proper start up order. Although it is not currently used, the initialization process allows for subsystems reinitialization or restart at run-time.

The structure can either be filled by hand (the knowledge of the Massiv Core Reference Guide is highly recommended) or automatically according to a predefined application profile. Currently the profiles are classified by node types, there are profiles for *anonymous-only nodes*, *client nodes*, *server nodes* and *data service nodes*.

Note

Anonymous-only nodes can be used to implement support utilities only as nearly everything is not available. For example the Massiv filesystem utilities (see Chapter 25, *Auxiliary Utilities*) run under this (modified) profile. An application programmer should be interested in server, client and service profiles only.

To construct a StartUpInfo structure from a profile, use the following constructor:

```
typedef void RegisterClasses();
typedef void CreateNodeObject();
typedef void NextTick();
...
StartUpInfo::StartUpInfo
  (
  Core::NodeId::Type  node_type, ❶
  const std::string & conf_file, ❷
  RegisterClasses *   register_classes = NULL, ❸
  CreateNodeObject *  create_node_object = NULL, ❹
  NextTick  *         next_tick = NULL ❺
  );
```

❶ This determines what profile will be activated. The Core defines the following values: `TYPE_SERVER`, `TYPE_CLIENT`, `TYPE_SERVICE` and `TYPE_ANONYMOUS`. Their meaning should be obvious from the text above.

❷ The path to the main configuration file. May contain $HOME shortcut that will be expanded to current user's home directory.

❸ The callback whose purpose is to register all managed classes that the local node supports. Usually points to Massiv::Generated::register_classes() generated by the IDL compiler. The callback is fired at system initialization. See description of `class_list` command in Section D.2, “The idl.list File”.

❹ When the local client or server node connects to the simulation, an object level representation of the local node should be created. The callback is responsible for instantiation of an application defined class that inherits from `Lib::NodeObjectInterface` and represents the local node. See Section 14.2, “Node Object”.

❺ Callback invoked by the Core at the end of each tick. *Note that the SRPC can not be called from within the callback.*

The constructor takes a parameter that describes what profile should be used, path to the configuration file and optional callbacks. The callbacks are automatically registered when the structure is used to drive the system initialization. However they need not be specified in the structure. In that case they must be registered explicitly *before* the system initialization.

Note

The callback registrations are global and are not part of the system state. It means that they must be registered before the system initialization (it can be done automatically through the `StartUpInfo`) and are not reset when the system shutdowns. They also can be registered only once and cannot be unregistered or changed for other callbacks later.

Use `System::register_register_classes()`, `System::register_create_node_object()` and `System::register_next_tick()` to register your callbacks. There can be more callbacks registered to the same type of the event.

To initialize the Core do the following steps:

- Create a StartUpInfo structure.
- Register system callbacks unless they are specified in the structure.
- Pass the structure to `System::initialize()`.

15.3. Downloading Prerequisite Data

The Core requires downloading the prerequisite data before connecting to the simulation. This ensures that both the application and the Core have the prerequisite data available and *up to date*. Such a feature is essential as a manual synchronization of the data is unacceptable. For example server nodes must know each other and this kind of information is distributed using this mechanism.

Note

Note that servers may be located miles away from each other, almost certainly not on a LAN. Thus, manual updates to the node database, the database where the information on server nodes is stored, would be cumbersome. No need to mention how client nodes can profit from such feature.

The download mechanism is the same as the one used to download the data dynamically when the system is running, see Chapter 24, *Data Service*.

To download the data call `System::download_data()`. When done the Core is ready to connect to the simulation.

Note

In order to be able to connect to the data service, the node must know the service node's public credentials and the credentials that will be used to authentize self. Since the download process runs anonymously, the local node does not introduce itself using its real identity (which may be unknown at the time of the download) and uses a well-known anonymous credentials instead. Both the anonymous credentials and service node's public credentials are specified in configuration files. See Section 27.11, “Node Database”.

15.4. Connecting To the Simulation

Until connected to the simulation, the object model does not and cannot work properly. The connection serves for two main purposes. It ensures registration to the simulation and restoration of the previous state. Once connected the node is ready to participate in the simulation and service requests submitted by other collaborating nodes.

This is what happens when a node connects to the simulation:

- *The node registers into the simulation using given credentials*

  Prior connecting to the simulation the node operates in an anonymous mode. In such a mode the node can download prequisite data but can not register into the simulation (note that an extra connection and extra mechanism is used for downloading the data; it has nothing to do with the connection that is utilized to connect a node to a simulation). In order to register the node identity must be known. Then, whenever the node initiates a connection to a remote node it will be authentized using the supplied credentials.

- *A simulation state is restored from the archive (server nodes only)*

  The simulation state is being automatically archived. When a server node connects to the simulation its state is restored from the latest valid archive. See archive management Chapter 23, *Archivation and Startup*.

Warning

Simulation can not be started if there is a server node without an archive. This also applies when the servers are started for the first time (startup archives must be created first).

- *Node objects are created (server and client nodes only)*

  Node objects are object-level representation of the registered nodes, that participate in the simulation (see Section 14.2, “Node Object”). To enforce instantiation of a node object representing the local node the *create_node_object* callback will be fired by the system (see above).

Call `System::connect()` to connect to the simulation. The method takes a reference to an input stream (*login file*) holding the credentials the node will use to authentize itself (*node id* and its *RSA keys*). The standard configuration format is used to store login files (see Section 27.11, “Node Database”):

```
[ ]
node_id : string = "[ Server 1 ]"
rsa_private_key : string = "..."
rsa_public_key : string = "..."
```

Note

In the Demo client login files are encrypted using client supplied passwords. The encryption is implemented on the "application level".

15.5. Running the Main Loop

In order to run the simulation or offer services to other nodes the *main loop* must be started and entered. The loop can be entered only if the node has correctly connected to the simulation.

Call `System::run()` to enter the main loop. The Core will loop in the method until quit from the loop is requested or a non-application exception is raised. Note that the Core cancels pending operations (for example SRPC calls) if a quit from the loop was requested. Thus, the simulation state can be affected by the quit (i.e. the state prior to a call to `System::request_quit()` can differ from the state after the quit has been actually performed).

Note

This is not a problem unless the main loop is reentered again and the simulation restarted without a reconnect.

Warning

Managed exceptions uncaught by the application will not stop the simulation. They are simply ignored.

15.6. Disconnecting From the Simulation

Unlike other systems the Core allows to properly disconnect from the simulation. Although such a feature can be used by server and service nodes too, it is targetted primarily to client nodes as it allows them to disconnect and reconnect under different identities without the need for restarting the applications. On the other hand server and service nodes disconnect from the simulation rarely, usually due to infrequent maintenance purposes only. If a server node disconnects the whole simulation will be stopped.

Call `System::disconnect()` to unregister and disconnect from the simulation. Once done, the client's Core will be switched back to the anonymous mode, the simulation state forgotten and managed objects deleted. The system state will be the same like the one before calling `System::connect()` and in fact it is completely legal to call `System::connect()` again.

15.7. Shutting Down the Core

To correctly shutdown the Core and stop all running subsystems call `System::shutdown()`. The system state will be reset and the Core will remain unitialized. In order to use the Core from now on, it must be initialized again. *However the application doesn't need to be restarted.*

15.8. More On the System Loops

Most of the work done by the Core is processed within a context of *system loops*. They enable processing of blocking or lengthy operations. The actual work that needs to be done is divided to smaller parts, each being processed in a single loop iteration. The iterations are called *ticks*. At the end of each tick the *next tick* callback is fired by the system which allows to do an additional work in parallel or cancel the work being processed by the Core without the need for using worker threads. Since the Core threading model is single-threaded any worker thread must not interact with the Core nor the running simulation. For example when connecting to the simulation the Core runs the *connect loop*, when running the simulation the *main loop* is being run, etc.

> **Note**
>
> Time differences between two ticks (and two next tick callbacks) are not fixed. The actual difference depends on the amount of the work that needs be done within a single tick, recent CPU and resource loads, etc. Also when there is no work to be done the Core sleeps for a while so that CPU cycles are not wasted. This is controlled by the tick scheduler. However there are soft limits on minimum and maximum time differences that the tick scheduler tries to hold. They can be specified via registry (`Settings/System/Scheduler` node, see Section 27.16, “Scheduler”) and the default values are `0.005f` (the minimum difference) and `0.05f` (the maximum difference).

The system loops are entered by appropriate calls to the `System` class. When the work associated with a particular loop is done, the loop is left automatically. The successful completion of the work often results in a system state change. If the work can not be completed an exception will be thrown and the state will not be changed.

The system loops can also be cancelled by the application by calling `System::request_quit()`. If called the current system loop would be cancelled. That would result either in a simple quit from the loop (if no state change is associated with the loop) or throwing an exception (if the work was not completed and a state change was expected).

The following tables list all the system loops, associated state changes and what happens if an error occurs or quit is requested:

Table 15.2. *Download data* system loop

Loop	`System::download_data()`
Purpose	Download prerequisite data
Required state	`PHASE_INITIALIZED`
State if succeeds	`PHASE_DATA_DOWNLOADED`
State if fails	`PHASE_INITIALIZED`, exception
Quit effect	`PHASE_INITIALIZED`, ExSystemError exception

Table 15.3. *Connect* system loop

Loop	`System::connect( const std::istream & config )`
Purpose	Connect to the simulation
Required state	`PHASE_DATA_DOWNLOADED`
State if succeeds	`PHASE_CONNECTED`
State if fails	`PHASE_DATA_DOWNLOADED`, exception
Quit effect	`PHASE_DATA_DOWNLOADED`, ExSystemError exception

Table 15.4. *Run* system loop

Loop	`System::run()`
Purpose	Run main loop
Required state	`PHASE_CONNECTED`
State if succeeds	`PHASE_CONNECTED`
State if fails	`PHASE_CONNECTED`, non-managed exception
Quit effect	`PHASE_CONNECTED`

Table 15.5. *Disconnect* system loop

Loop	`System::disconnect()`
Purpose	Disconnect from the simulation
Required state	`PHASE_CONNECTED`

State if succeeds	`PHASE_DATA_DOWNLOADED`
State if fails	`PHASE_CONNECTED`, exception
Quit effect	`PHASE_CONNECTED`, ExSystemError exception

Table 15.6. *Generic* system loop

Loop	`System::generic_loop()`
Purpose	Loop until cancelled.
Required state	!`PHASE_CONNECTED` && !`PHASE_RUNNING`
State if succeeds	
State if fails	`exception`
Quit effect	

15.9. An Example

This section shows an example of an application skeleton for a client node. We will not show a skeleton for a server node here as most of its complexity relates to startup archive creation. For more information see Section 16.2.1, “Archive Management API”, the Massiv Core Reference Guide or the relevant part of the source codes of the Demo.

Note

To create a startup archive, initialize the Core as usually and download the prerequisite data. Do not connect to the simulation as that would restore the latest saved state (if there is any) or fail otherwise. Create global objects (initial simulation state), save the archive and shutdown the Core. Often only the first server creates the global objects, the remaining servers create empty archives instead.

```
#include "class_list.h"
...
std::string login;
bool        quit;
...
void create_node_object_callback()
    {
    CreateObject< ClientNode >();  ❶
    }
...
void next_tick_callback()  ❷
    {
    switch( System::get_phase() )  ❸
        {
        case System::PHASE_DATA_DOWNLOADED:
            ...
            /* Handle a login dialog. */
            ...
            if( enter_hit )
              {
              login = ...;
              System::request_quit();
              }
            if( escape_hit )
              {
              quit = true;
              }
            break;
        case System::PHASE_RUNNING:
            ...
            /* Draw world. */
            ...
            if( escape_hit )
              {
              System::request_quit();
              }
            break;
        }
    }
...
int main( int argc, char * argv[] )
    {
    std::istringstream config;
    ...
    System::initialize
        (
        System::StartUpInfo
            (
            NodeId::TYPE_CLIENT,
            "massiv.conf",
            Massiv::Generated::register_classes,
            create_node_object_callback,
            next_tick_callback
            )
        );
    ...
    try
        {
        System::download_data();

        while( !quit )
            {
            System::generic_loop(); ❹

            try
                {
                config.str( login );
                System::connect( config );
                System::run();
                System::disconnect();
                }
            catch( Exception & )
                {
                ...
                Handle exceptions here
                ...
                }
            }
        }
    catch( Exception & )
        {
        ...
        Handle exceptions here
        ...
        }
    ...
    System::shutdown();
    }
```

❶ For information about how a node object should be implemented refer to Section 14.2, “Node Object”

❷ Note that no cycle is needed in the next tick callback, because it is being called periodically.

❸ This is how the current system phase can be determined. It also gives an information about what system loop is running.

❹ Entering a generic loop in this state (`PHASE_DATA_DOWNLOADED`) ensures that a login dialog will be displayed and handled. When a user supplies his credentials, the loop is cancelled and the credentials are passed to `System::connect()`.

16. Global Objects And System Interface

The Massiv defines a concept of a central management of all global objects (i.e. singleton objects that are accessible globally from both the Core and the application, often called Core subsystems) represented by the `Global` static class. In addition to the `Global` class there is `System` class that can be used to control the Core in a more abstract way, without the need for digging into internal Core subsystems. A few words on this topic were already written in Section 15.1, “Overview”.

16.1. Global objects

`Global` class serves as a repository for all global objects defined in the Core. It holds references to them and and provides methods (one per each object) to access them. Moreover it publishes an interface for logging messages in a simplier way (therefore there is no need for using the logger explicitly).

Warning

Whenever a global object needs be accessed, corresponding reference must be obtained from the `Global` class and must not be kept by the caller. This allows to replace global object instances at run-time.

Global objects are created and properly registered to the repository by `System::initialize()` method. An attempt to access an unregistered object results in a run-time error (assert).

16.1.1. List of Global Objects

The following list summarizes global objects registered to the `Global` class. Note that not all the global objects can be utilized by the application directly. In fact some objects are actually private to the Core and must not be used by the application at all! However the list doesn't contain such "banned" objects.

Names of the methods, used for accessing particular objects, are quite straightforward. For example, to get a reference to `ThreadManager` you have to use the `Global::thread_manager()` method. For other objects the naming principle is the same.

You can find more information about all of the objects in the Massiv Core Reference Guide. Moreover, at some of them there is a link to yet another information source.

- `AccountManager`

  Manages client's account creation (in cooperation with the data service.

  See also Section 14.3, “Account Object”.

- `ClassManager`

Registers all known managed classes supported by the local node and keeps metaobjects for them. Each class is assigned an unique class type id, which can be transmitted over the network and is not platform specific (unlike standard C++ type info).

See also Section 12.2.1, “Obtaining a MetaObject”.

- `CriticalSectionManager`

  Manages critical sections creating.

  See also Section 17.5, “Worker Threads” or the Massiv Core Reference Guide, module Threads.

- `DataManager`

  Manages data objects and allows for dynamic (*on-the-fly*) data downloads while a player is already playing the game. It also enables substitutions of unavailable data by a more general one that has already been downloaded.

  See also Chapter 24, *Data Service*.

- `Logger`

  Logger enables making logs at run-time providing a possibility to sort messages according to several-criteria filters and send them to various destinations.

  See also Chapter 20, *Logger Library*.

- `ObjectManager`

  Object manager is responsible for local object management. It pushes the simulation forward.

- `PathManager`

  Enables to get filesystem paths to various Massiv-specific or system directories, such as *log*, *data*, *source_data* (see Chapter 24, *Data Service*), *archive*, *work* or *login* directory.

  See also the Massiv Core Reference Guide.

- `Registry`

  A hierarchical database that stores variables that may come either from configuration files, from various statistics or may be created for other reasons at run-time.

  See also Chapter 19, *Registry*.

- `ReplicaManager`

  Replica manager monitors updates of replicas. An user can register a callback that will be invoked everytime the specified replicas are updated.

  See also the Massiv Core Reference Guide.

- `SemaphoreManager`

  Manages semaphore instantiations.

  See also Section 17.5, “Worker Threads”.

- `ThreadManager`

  Manages worker threads.

  See also Section 17.5, “Worker Threads”.

- `VolumeManager`

  Manages a virtual file system that provides a single interface for accessing streams on the native file system as well as a set of *mounted* Massiv-specific volumes (such as `Volume` or `CompactVolume` images).

  See also Chapter 21, *Massiv Filesystem*.

16.1.2. `Global` Logging Interface

The `Global` logging methods are shortcuts for similar methods provided directly by the `Logger`.

The most useful methods are `log_debug`, `log_info`, `log_warning` and `log_error`, each having `facility`, `priority` and `message` parameters (except for the `log_error` method that has only the `facility` and `message` parameters, because the priority is automatically the highest possible. See Chapter 20, *Logger Library* to get better idea what is the difference between the methods and what the parameters stand for.

The following example shows a typical code that logs some non-trivial message:

```
int   line_number = ...;
...
std::ostringstream s;
s << "syntax error at line \"" << line_number << "\"";
Global::log_warning( Status::FACILITY_REGISTRY, Status::PRIORITY_HIGH, s.str() );
```

16.2. System Interface

`System` class allows to control the Core in a global way. It provides facilities for the Core initialization, shutdown, etc. and simplified access to Core subsystems. Some of the functionality can be accessed by calling appropriate subsystems directly, however this would require knowledge of the Core internals (as semi-public API would be used) and the use would be cumbersome. On the other side, the API provided by the `System` class is completely public.

The `System` API can be separated into these functional groups:

- *System callbacks*

  Registering system callbacks used to upcall the application from the Core. The callbacks must either be registered manually before the system initialization or automatically via entries in StartUpInfo structure that is passed to System::initialize(). See also Section 15.2, “Initializing the Core”.

Method
void register_register_classes(void (*callback)())
void register_create_node_object(void (*callback)())
void register_next_tick(void (*callback)())

- *Managing system phases, system loops*

  Initializing the Core, downloading prerequisite data, connecting to simulation, running simulation, disconnecting from simulation, shutting the Core down. See also Figure 15.1, “System Phases”.

Method	Semantics
Phase get_phase()	Determine current system phase.
void initialize(const StartUpInfo & info)	Initialize the system according to StartUpInfo structure.
void download_data()	Download prerequisite data.
void connect(std::istream & my_config)	Connect to the simulation using credentials stored in my_config.
void run()	Run the main loop.
void disconnect()	Disconnect from the simulation.
void shutdown()	Shutdown the system.
bool is_connected()	Check whether the system is connected to the simulation.
void request_quit()	Request quit from the current system loop.

- *Time*

  Retrieving the current simulation time and the local system time. See also Chapter 18, *Simulation Time*.

Method	Semantics
const STime & time()	Get simulation time.
const SystemTime & system_time()	Get local system time.

- *Garbage Collector API*

  Forcing garbage collection, deleting objects explicitly, disposing GC root objects. See also Section 5.4, “Garbage Collector”.

Method	Semantics
void force_gc()	Force garbage collection.
void collect_object("managed_pointer")	Force GC to collect object referenced by the managed pointer.
void dispose_gc_root("managed_pointer")	Tells GC that permanent GC root object should be demoted to a non-root object.

- *Well known object ids*

  Determining naming service object id.

Method	Semantics
const ObjectId & get_naming_service_object_id()	Get naming service object id.

- *Local node information*

  Obtaining local node id.

Method	Semantics
const NodeId & get_local_node_id()	Get node id of the local node.
bool is_server()	Test if the local node is a server.
bool is_client()	Test if the local node is a client.

- *Archive management*

  Creating initial (startup) archives, packing and unpacking archives. See also Section 16.2.1, “Archive Management API”, Chapter 23, *Archivation and Startup* or the Massiv Core Reference Guide.

Method	Semantics
void create_initial_object_provider(UInt32 id)	Ask the Core for a new ObjectId pool so that object instances could be created. To be used when a startup archive is being created.
void create_initial_archive()	Archive local simulations state. To be called

Method	Semantics
	from server nodes that did not connect to the simulation and where startup simulation state has just been prepared.
void unpack_archive(const std::string & archive_id, const std::string & directory)	Export archive streams from the volume image to external file system.
void pack_archive(const std::vector< std::string > directories, const std::string & archive_id)	Import archive streams from external directories to a new archive. Create an archive volume from one or more unpacked archives.

16.2.1. Archive Management API

The Core provides a mechanism for automatic simulation state externalization and restoration. Server nodes periodically create distributed snapshots of the whole simulation and store them to locally managed archive files. When servers are started and connect into the simulation (see Section 15.4, “Connecting To the Simulation”) their states are restored from the archives. This ensures that the simulation state is durable and survives server down times, crashes, etc. In order to be able to connect into the simulation as a server node there must be at least one valid archive so that the local server state can be restored from it. If there is no archive available (for example the server is going to be started for the first time), a startup archive must be created first. The following paragraph will describe this in detail.

To create a startup (initial) archive the following steps must be made:

- Initialize the Core and start the server in an anonymous mode.

  This is done by calling `System::initialize( .... )`, optionally followed by `System::download_data()`. Once initialized the Core operates in an anonymous mode which means that it can not cooperate with other nodes and the simulation dispatching is disabled.

- Create an object provider so that we will be able to create objects.

  Call `System::create_initial_object_provider( id )` with the `id` variable identifying the provider. The caller must ensure that it is unique in the system. Once done a new `ObjectId` pool is created and ready to generate object identifications. See Section 4.4, “Object Identification”.

- Prepare the "startup" simulation state.

  Typically some sort of "global" objects are created and events scheduled to them. Note that the server operates in an anonymous mode and the simulation dispatching is disabled. Events scheduled at this stage will not be fired until the server is restarted from the startup archive. This way, the startup archive logic can be implemented as a regular server lib code that will be ran after the server restart.

Warning

SRPC can not be used at this stage.

Note

Startup archives must be created on all registered servers. Typically "global" objects are created on the "first" server only, empty archives (with no objects) on the remaining servers.

- Archive the created simulation state and shutdown the Core.

  Call `System::create_initial_archive()`, then `System::shutdown()`.

```
...
class World : public Object
  {
public:
  ...
  void create_initial_world();
  ....
  };
...
void create_initial_archive
  (
  UInt32  my_provider_id,
  bool    create_initial_objects
  )
  {
  System::initialize( System::StartUpInfo( NodeId::TYPE_SERVER, ... ) );
  System::download_data();
  System::create_initial_object_provider( my_provider_id );
  if( create_initial_objects )
    {
    Remote< World > world = CreateObject< World >();
    world->async_create_initial_world(); ❶
    }
  System::create_initial_archive();
  System::shutdown();
  }
```

❶ The RPC is scheduled only. It will actually be called once the server is restarted from the startup archive.

Simulation state is externalized to archive files with a proprietary structure. If there is a reason to modify their content (for example when adding new data members to existing classes) they must be "unpacked" to an external file system first (`System::unpack_archive()`). Modifications can be performed then. When done the archive must be "packed" again (`System::pack_archive()`). Although unauthorized changes to the archives can break their consistency, cause application crashes (because of data integrity loss) or make the core refuse to load it, there are very special reasons why these operations are allowed.

17. Threading Model

This chapter describes the threading model used by the Core and the issues a programmer must deal with when the use of application-level threads is planned.

17.1. The Model Used By the Core

First of all it must be noted that the Core is not a multithreaded library. Although there may be multiple Core threads running in parallel the API exposed to the application is not thread-safe. This may look like a serious design flaw but hey, the Massiv is a distributed MMO game middleware intended to be ran on multiple nodes, not on a single CPU. If the Core had been multithreaded, so would have been the distributed application. This would result in application complexity growth (locking on the application level would be essential), performance losses due to the need for synchronization (distributed locking is often innefective) and the risk of distributed deadlocks. Needless to say that the transparences the Core currently supports would probably have to be relaxed or could not be implemented reliably at all.

The Core itself allocates *worker threads* for its own needs. These are used by the file system to implement asynchronous IO, for example. Besides these threads *there may be one or more simulation threads that are scheduled cooperatively*. Only one thread runs in a moment. However the Core is free to switch among the simulation threads whenever it is entered to perform a blocking operation. From the point of view of the application, it runs cooperatively and thread switches occur at well-known places only ("preemption points").

17.2. When the Core Runs

As the application runs and uses the object model, the Core is entered to peform various operations. Most of the operations are "immediate" operations which means that they are served immediatelly and other parts of the application (or Core subsystems) can not run in the meantime. However there may be blocking operations that can not be served immediatelly - the Core switches temporarily to an another simulation thread and lets the application or the Core run within the context of the other thread until the request can be completed. The blocking operations are the preemption points. *Currently, the only allowed blocking operation is a waiting for a SRPC reply.*

When the simulation is running the Core is looping in a system loop (see also Section 16.2, “System Interface”). It does its own work and when it receives a request to upcall the application (for example to deliver an object migration), the application is entered. Until the application returns from the upcall the Core will not run. However the application can issue a blocking operation. The application context will then be frozen and the Core will be reentered. Until the blocking operation finishes the Core is looping in a new instance of the system loop (probably also within a context of an another simulation thread) and serves other requests. The application can be upcalled in the meantime, including the object that waits for the completion of the blocking operation. When the blocking operation completes the application will be resumed from the saved context.

Note

The Core is always looping within a context of a system loop. If an upcalled client code issues a blocking operation, the new instance of the system loop is spawned, current application context is saved and the Core will be looping in the new loop instance. When the blocking operation finishes, the current loop is terminated and the saved context restored.

The context saving and restoring is implemented by switching to a different simulation thread. The old thread is suspended (thus the context saved) and a new thread is let run in the meantime. To restore the saved context a switch to the previously suspended thread occurs. Only one simulation thread runs in a time. When there is no free thread the system loop is simply nested (instead of spawning the loop within a context of a new thread, the loop is reentered within the context of the same thread). However the nesting allows to resume saved contexts in LIFO order only (the contexts are saved on the program stack, "older" contexts can not be resumed until "newer" blocking operations complete).

17.3. What Happens When the Core Runs

Although the previous section was a bit too much technical the major information was that the blocking operations can cause a preemption and both the Core and the application can run in the meantime. In particular object states after the blocking operation can be different from the states before the operation. This sections describes potential pitfalls.

Note

A preemption can occur due to a blocking operation only.

When the Core runs all Core subsystems run too. This means that objects can migrate or be garbage collected, for example. Object replicas can be deleted or become inconsistent. In order to prevent such a behavior objects must be strong referenced from the stack. Note that this does not work for object replicas and one has to be very careful when working with them. The following list summarizes common errors:

- *Weak-referenced objects can no longer be accessible or consistent*

  Weak-referenced objects accessible and consistent before the blocking operation can no longer be accessible or can be inconsistent when the operation finishes. Use strong stack pointers to "pin" local (non-replica) objects, copy replica state before issuing a blocking operation.

```
class RemoteObject : public Object
  {
...
  void rpc_method();
...
  };

class MyObject : public Object
  {
...
  void f( WeakPointer< MyObject > next );
...
```

```
  PRemote< RemoteObject > ptr;
  };

/*
void MyObject::f( WeakPointer< MyObject > next )
  {
  next->ptr->sync_rpc_method();
  next->ptr = null; /* May fail. */
  }
*/

void MyObject::f( WeakPointer< MyObject > next )
  {
  Pointer< MyObject > n = next;
  n->ptr->sync_rpc_method();
  n->ptr = null;
  }
```

- *Object can be reentered while waiting for a blocking operation completion*

  Currently the Core ensures that SRPC (the only allowed blocking operation) will never cause a deadlock because the "blocked" objects waiting for the reply are not locked and can be reentered. Application logic must deal with this.

```
class Counter : public Object
  {
  ...
  void add( int count );
  ...
  PInt32                  counter;
  PRemote< RemoteObject > ptr;
  };

/*
void Counter::add( int count )
  {
  int c = counter;
  ptr->sync_rpc_method();
  counter = c + count; /* counter may not be equal to c. */
  }
*/

void Counter::add( int count )
  {
  int c = counter;
  counter = c + count;
  ptr->sync_rpc_method();
  }
```

- *Object state can change during a blocking operation*

  Any object (including the blocked one) can be reentered while a blocking operation is in progress. There may be multiple pending blocking operations in progress. To deal with this problem (if it is a problem at all) either "recheck" the state when the blocking operation finishes, perform such operations at the end of the upcall or implement own "locking" mechanisms on the application level (reject or queue operations submitted in the meantime).

```
class Operation : public ValueType
  {
...
  };

class Operations : public Object
  {
public:
...
  void do_operation( const Operation & operation );
...
private:
...
  void do_operation_internal( const Operation & operation );
...
  PArray< Operation >     retry_queue;
  PBool                   inside_do_operation;
  PRemote< RemoteObject > ptr;
  };

void Operations::do_operation_internal( const & operation )
  {
  ...
  /* Blocking operation is performed here. */
  ...
  }

void Operations::do_operation( const & operation )
  {
  int i;

  /* Test for recursion ("lock"). */

  if( inside_do_operation )
    {
    retry_queue.push_back( operation ); /* We do not want to perform the operation now. Queue or
    return;
    }

  /* Do operation. */

  inside_do_operation = true;
  do_operation_internal( operation );

  /* Replay queued operations ("unlock"). */

  for( i = 0; i < retry_queue.size(); i++ )
    {
    do_operation_internal( retry_queue[ i ] ); /* retry_queue may have grown in the meantime. */
    }

  retry_queue.clear();
  inside_do_operation = false;
}
```

Maybe you should look at the implementation of sector tile reservations in the Demo. In order to move an entity, the corresponding tiles the entity would be occupying must be reserved. The reservation is done using SRPC. If another move request arrives while the previous move is still in progress the entity remembers the new position and once the previous move finishes the new move to the new position is automatically initialized. This differs from the example above in that way that entities remember the last requests only. There is also OPTIMISTIC_LOCK() macro that simplifies the accounting.

Warning

This is the price for using SRPC. The Core does not implement builtin synchronization primitives to be used by the distributed application because of several reasons:

- If locks were supported the application could deadlock. In the current model deadlocks are avoided.
- If locks were supported object could be locked for an indefinite time amount.

  While waiting for the lock objects would be pinned and could not migrate. The system would be stalled and archivation could not be initiated if there was a blocked object waiting for a lock.
- Due to the cooperative multithreading model locks can be simulated on the application level.

  However they can not be implemented because there is no way how to perform a "wait" operation and allow other threads to run in the meantime.

17.4. Configuration

See Section 27.14, “Remote Procedure Call” for information about how to set number of allowed threads.

17.5. Worker Threads

The Massiv provides platform independent interfaces to access threading and synchronization facilities. Of course they have been implemented primarily to be used by the Core but the API was also exposed to applications. However since the Core (and thus distributed applications) are single threaded worker threads spawned by an application *must not interact with the Core nor the application at all.* Nonetheless they can be used by UI and sound subsystems in the Massiv clients if standard polling mechanism via hooking next tick callbacks (see Section 16.2, “System Interface”) is not sufficient. The API is described thoroughly in the Massiv Core Programmer's Reference and will not be discussed here in detail.

Warning

The thread and synchronization API can be used locally only. It is not available on the "object level" and can not be used by the distributed application (i.e. locking local resources from remote nodes).

The API to the thread subsystem is in general similar to what you would find in common operating systems. It is accessible through `Thread` object instances and a `ThreadManager` global object. While the Thread objects return information on the corresponding threads (its static methods return information on the current thread), ThreadManager provides access to the whole threading subsystem and allows to launch new threads. When a new thread is launched corresponding Thread object is created.

To synchronize self with different threads synchronization primitives are implemented too. These are `CriticalSection` and `Semaphore`.

Critical sections implement simple guarded sections for mutual exclusion of involved threads (only one thread can enter the section, other threads are not allowed to step in unless the first thread has already left the section). To enter the section call `enter()` method. If an another thread has already entered the section the calling thread will be blocked until the other thread leaves it (`leave()`).

Semaphore is an another synchronization primitive. A positive integral counter is associated with each semaphore instance that basically denotes how many threads can enter the guarded section. Initially the semaphore value is set to the value specified at semaphore construction. Semaphores are accompanied by atomic operations that either increment (`up()`) or decrement (`down()`) the semaphore's value. If the value can not be decremented (as it is already 0) the calling thread is blocked in `down()` until an another thread increments it. Usually the value is decremented when the guarded section is entered and incremented when the section is left.

Both `CriticalSection` and `Semaphore` instances are created and managed by corresponding managers (`CriticalSectionManager`, `SemaphoreManager`), accessible as global objects.

The following example shows basic usage of the threading and synchronization subsystems. However there is no need to comment on the use of the interfaces as it is obvious and the exact description could be found in the Massiv Core Programmer's Reference if needed.

```
struct Item { ... };

void produce( Item & item );
void consume( Item & item );

int producer( VariantPointer arg );
int consumer( VariantPointer arg );

const int         max_count = 10;

Semaphore *       free_count = Global::semaphore_manager().create( max_count );
Semaphore *       full_count = Global::semaphore_manager().create( 0 );
CriticalSection * section = Global::critical_section().create();

int               count = 0;
Item              queue[ max_count ];
```

```
bool              quit;

Reference< Thread > p_thread = Global::thread_manager().create( producer, &quit );
Reference< Thread > c_thread = Global::thread_manager().create( consumer, &quit );

int producer( VariantPointer arg )
  {
  bool & quit = *variant_cast< bool * >( arg );
  int    num_items_produced = 0;

  while( true )
    {
    free_count->down();
      {
      section->enter();

      if( quit )
        {
        section->leave();
        return num_items_produced;
        }

      produce( queue[ count++ ] );
      num_items_produced++;

      section->leave();
      }
    full_count->up();
    }
  }

int consumer( VariantPointer arg )
  {
  bool & quit = *variant_cast< bool * >( arg );
  int   num_items_consumed = 0;

  while( true )
    {
    full_count->down();
      {
      section->enter();

      if( quit )
        {
        section->leave();
        return num_items_consumed;
        }

      consume( queue[ --count ] );
      num_items_consumed++;

      section->leave();
      }
    free_count->up();
    }
  }

void stop()
```

```
{
section->enter();
quit = true;
section->false();

/* Stop producer. */
free_count->up();
std::cout << "num_items_produced = " << p_thread->join() << '\n';

/* Stop consumer. */
full_count->up();
std::cout << "num_items_consumed = " << c_thread->join() << '\n';
}
```

18. Simulation Time

The simulation time is an abstract time used in important parts of the Massiv Core (event scheduling, RPC) and it can be used also by a programmer. It has nothing to do with a real time. When new simulation is started, the simulation time (also called "the massiv time") is simply zero. When the simulation is started from an archive, the simulation time is set to the time stored in the archive.

The simulation time is a decimal number that expresses number of seconds elapsed since the moment when a simulation was started. In the actual implementation, the precision of the simulation time is in miliseconds. Example: simulation time 130.435 means 130 second and 435 miliseconds since the simulation start.

To obtain the actual simulation time, call `Massiv::System::time()` method. Important fact is that the simulation time doesn't change during one Massiv tick. Thus calling a method above more times during one tick will return the same value of simulation time.

The aim is that the simulation time should be "as fast as" the real time. This means when one hour elapses measured in the real time, one hour (3600.000 seconds) elapses in the simulation time. However, the hardware clock of computers will run a little differently. The simulation time is computed on the basis of the hardware clock, so a simulation time synchronization is needed among Massiv servers, and clients' simulation time must be adapted to according to the synchronized simulation time. This solution leads to the fact that the simulation time is not "as fast as" real time, but "as fast as" the fastest hardware clock among massiv serves.

The class `Massiv::Core::TimeManager` manages the time synchronization.

Sometimes a programmer needs to have an access to the operating system time to be able to measure time with a high accuracy. Call method `Massiv::System::system_time()` to get an actual value of the system time - this value is a decimal number with the same meaning as simulation time. The following example shows a typical use of the system time:

```
SystemTime  old_time = Massiv::System::system_time();

while( Massiv::System::system_time() - old_time < 0.020f ) ) // 20ms
    {
    // do some work within 20ms
    }
```

Note

Unlike the simulation time, the value of the system time changes during one massiv tick.

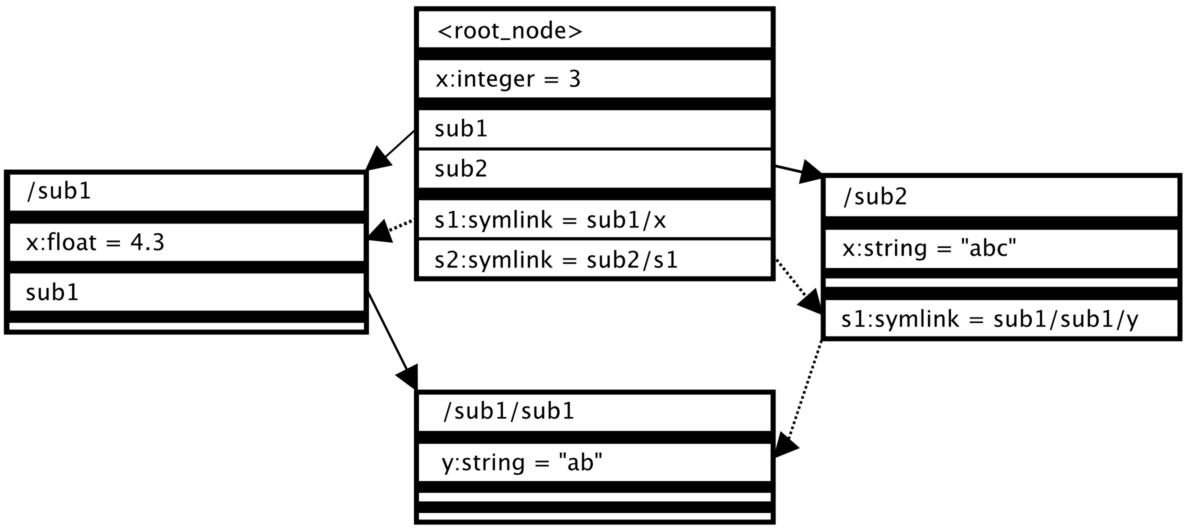
19. Registry

19.1. Survey

Registry module is primarily proposed for easy configuration and initialization of all the parts of Massive server or client. Nevertheless, this functionality has been further enhanced - now the Registry can serve as a hierarchical database for many more purposes (for example for storing various system statistics - user can obtain a reference to any variable in the registry and thus make the access or modification very effective). Registry can be filled by a program code, but it also provides methods for loading its contents from *configuration files* .

19.2. Registry Structure

Registry has a hierarchical tree structure. This hierarchy is based on one `root` node that must be always present. Any node in the registry is either leaf or contains at least one subnode. Besides subnodes, each node can store variables (variables stored in the Registry are so-called config variables) or symbolic link definition.



Structure of the registry (instance diagram - example). Note that each node consists of three sections - config variables, subnodes references and symbolic links.

19.2.1. Config Variables

As it was already stated, config variables are the special variables that are internaly used by the registry to store its variables. The advantage they bring is that they can be handled regardless their actual type. Nevertheless, when creating the variable, the type must be specified. Typically, this will be done in a configuration file using predefined keywords. The following table shows the possible config variable types in case you need to create config variable from program code:

Table 19.1. Config Variable Types

Type	Relevant ConfigVariable Descendant
integer (32 bits)	`CVarInteger`
large integer (64 bits)	`CVarLargeInteger`
float	`CVarFloat`
string	`CVarString`
boolean	`CVarBoolean`

19.2.2. Symbolic Links

Symbolic links (also aliases) symbolic names for variables stored under another name in the same registry node or under any name in another node. They contain textual path that specifies the target variable. Note that an alias target may be another alias, etc., until there is a variable in the chain.

Symbolic links are useful for example when there have to be the same variables in more nodes. Instead, we can have the real variables stored only once and more aliases set up to refer each of them.

19.3. Usage

Note

For detailed registry usage information, please refer to the Core Reference Guide. Here is only the most significant stuff.

19.3.1. Filling The Registry With Data

Let's assume we have a configuration file `registry.conf` containing the following text:

```
[ ]
x     : int     = 3
s1    : symlink = /sub1/x
s2    : symlink = /sub2/s1

[ sub1 ]
x     : float   = 4.3

[ sub2 ]
x     : string  = "abc"
```

Now the configuration contents can be loaded using the `Registry::load_config_file` method. Moreover, we will use `Registry::create_variable` and `Re-`

gistry::create_symbolic_link (as obvious, creates variable or symbolic link, respectively) to make the contents of the registry database be the same as in the picture in the Section 19.2, “Registry Structure”.

```
Registry    registry;

registry.load_config_file( "registry.conf" );

registry.create_variable( "y", "/sub1/sub1", CVAR_TYPE_STRING, "abc" );
    // This creates new node /sub1/sub1 and inserts a variable called
    // y of string type with default value "abc".

registry.create_symbolic_link( "s1", "/sub2", "/sub1/sub1/y", false );
    // This creates new symbolic link in the /sub2 node. This link
    // refers the y variable inserted in the previous call to
    // insert_variable. The last parameter (false) indicates that
    // the registry won't check whether the symbolic link target really
    // exists.
```

Note

The configuration file can include other configuration files, as described in the Chapter 26, *Registry Configuration File Syntax*. There are even more loading routines enabling for example loading configuration relatively to some specific registry node, etc.

Note

The registry doesn't allow you to delete variables or symbolic links you have created.

Note

You can whenever save the registry into file using the Registry::save_config_file(<filename>) method. Again, there are more possibilities - for example, it is possible to save only a subtree of the registry, etc.

19.3.2. Accessing Variables In The Registry

There are more possibilities how to access variables. The most common one is the method Registry::access_variable(<full_name>). This routine automatically handles symbolic links; should the <full_name> refer to a symbolic link (or symbolic links chain), all the links will be resolved and a reference to the target variable (not to an alias) will be returned. Reference to ConfigVariable is returned, but this is not limiting at all, because config variables have retyping ability for all common (C built-in) types (Use for example the read_string() method; similar for other types. These methods perform relevant retyping if needed.)

```
// Let's assume we have the registry contents the same as in the
// last example.

ConfigVariableFloat & cvar_float = registry.access_variable( "/sub1/x" );

std::cout << "/sub1/x = " << cvar_float.read_float() << std::endl;
```

```
    // Prints out "/sub1/x = 4.3".

ConfigVariableString & cvar_string = registry.access_variable( "/s2" );

std::cout << "/s2 = " << cvar_string.read_string() << std::endl;
    // Prints out "/s2 = ab" (although /s2 is actualy a link pointing to
    // the "sub2/s1", which is another symbolic link which points to
    // the /sub1/sub1/y at last).
```

19.3.3. Iterating and Searching The Registry

There are several options available for iterating the registry. For mere variable enumerating withing one node, use `Iterator` (or `ConstIterator`, respectively); to enumerate a single node's subnodes, `SubnodeIterator` (or `ConstSubnodeIterator`, respectively) is a good choice. If you want more complicated enumeration involving more nodes or whole subtree, you have to use the `SearchContext` and special registry methods.

Iterators

As mentioned above, the Registry offers several types of iterators. The `const` and "standard" version of iterator is more or less the same; the difference is only that you cannot modify a config variable using a constant iterator. The difference between 'within-node' and subnodes iterator is also obvious - the first can be used for enumerating variables, the second for subnodes. The last item that can be contained withing registry nodes - symbolic links are enumerated together with variables. It means that the 'within-node' doesn't distinguish between symbolic links and iterators.

As usual when working with iterators, there are several registry methods yielding *initial* iterators. The following table shows all of them. And, of course, they contain all the operators you would expect to be available when working with pointers.

Table 19.2. Basic Registry iterator methods

Method	Description
`begin( const std::string & path, const std::string & pattern )`	Returns an iterator refering the first variable (or symbolic link) in the node specified by `path`. If `pattern` is given, the iterator will ignore all items that don't match this pattern (wildcards).
`end()`	Returns an invalid iterator. This can be used in comparisons to detect that the iterator doesn't refer to a valid variable/symbolic link.
`begin_subnodes( const std::string & path, const std::string & pattern )`	Returns an iterator refering the first subnode of the node specified by `path`. If `pattern` is given, the iterator will ignore all subnodes that don't match this pattern (wildcards).
`end_subnodes()`	Returns an invalid iterator. This can be used in comparisons to detect that the iterator doesn't refer to a valid subnode.

Note

Each of the methods in the table returns either constant or 'standard' iterator, depending on the context.

The following example prints names of all variables and aliases contained in the root node that begin with the `s` character; then prints all subnode names:

```
registry.create_variable( "str1", "/", CVAR_STRING, "first string" );
    // lets insert one more value into the registry first:

Registry::Iterator  it( registry.begin( "/", "s*" ) );
    // creates the iterator and binds it with the root node

std::cout << "Variables:" << std::endl;

while( it != registry.end() )
    {
    std::cout << it->get_name() << std::endl;
    it++;
    }

Registry::SubnodeIterator   sub_it( registry.begin_subnodes( "/", "*" ) );
    // creates the subnodes iterator and binds it with the root node

std::cout << "Subnodes:" << std::endl;

while( sub_it != registry.end_subnodes() )
    {
    std::cout << *sub_it << std::endl;
    sub_it++;
    }
```

The preceding example should present an output similar to the following:

```
Variables:
str
s1
s2
Subnodes:
sub1
sub2
```

SearchContext

For more powerful Registry enumeration, you have to use the `SearchContext` class directly instead of iterators. `SearchContext` stores all the information about where exactly the last search had stopped. That enables the next search to continue from this spot. `SearchContext` has many methods to get information about the refered variable or to get the variable itself. The following table summarizes most significant methods:

Table 19.3. Registry Enumeration and Search Methods

Method	Description
`SearchContext open_registry_node( const std::string & path )`	Returns a search context bound with the registry node specified by the `path` parameter. This context can be used as a parameter of the following methods.
`bool find_variable( const std::string::value_type * pattern, SearchContext & context )`	Searches for a variable or symbolic link matching the given `pattern`. This search is not restricted to only one node; on the contrary, it traverses the whole registry tree. If the variable has been found, returns true and fills the `context` appropriately. If next time this `context` is passed to the method, the search will continue from where it finished the last time.
`bool search_registry_node( const std::string::value_type * pattern, SearchContext & context )`	This method is exactly the same as the previous one, but this is restricted to just one Registry node.

The `SearchContext` state is always specified with one of the following types:

Table 19.4. Registry Enumeration and Search Methods

Type	Description
`RRT_VOID`	The search context doesn't refer to anything.
`RRT_NOT_FOUND`	No variable found during the last search using this context.
`RRT_VARIABLE`	The context refers to a variable.
`RRT_ALIAS`	The context refers to an alias.

In the following example we are looking for all the variables called `x` in all the nodes contained in the registry:

```
SearchContext   sc( open_registry_node( "/" ) );

registry.find_variable( "x", sc );

while( sc.get_type() != RRT_NOT_FOUND )
    {
    std::cout << sc.get_name() << " = " <<
    std::cout << sc.get_variable().read_string() << std::endl;
    registry.find_variable( "x", sc );
    }
```

Possible output of this example could be:

```
x = 3
x = 4.3
x = abc
```

19.3.4. Settings

It is obvious that if `access_variable` was called each time some configuration variable is called, the efficiency would suffer a great deal (`access_variable` have to find the relevant node, then resolve symbolic links, etc).

The solution to this matter is the `Settings` class. It descendants keep references staight to the relevant config variables in the Registry. Thus, when we need to access some registry variable, we use this reference instead of calling the `access_variable`.

To make use of the Settings class, follow these steps (each of them accompanied with a relevant part of an example; for the example, let's suppose we want to have configuration variable `ping_delay` for client class `Network`):

- Create a new settings class that inherits from the `Settings` and that is specific for a class you are writting settings for (let's call it the *client class*). The new settings class should contain references to all the config variables you want to use within the client class. Also, it should declare a default constructor.

  Next, within the client class, create a variable - instance of your new settings class.

```
class Network
    {

    // some network stuff here ...

public:

    enum EDefaultPingDelay
        {
        DEFAULT_PING_DELAY = 100
        };

    class NetworkSettings : public Settings
        {
    public:
        NetworkSettings();
    public:
        Integer &       ping_delay;
        };

public:
    NetworkSettings     settings;
    };
```

- Implement the default constructor. It should set each reference withing the class to refer to the relevant config variable properly.

```
Network::NetworkSettings::NetworkSettings()
    :   Settings( "Network" ), ❶
        ping_delay
            (
            access_integer
                    (
                    "ping_delay", ❷
                    DEFAULT_PING_DELAY
                    )
            )
        {
        }
```

- Now you can access the variables directly from the client class using for example `settings.variable_name` syntax.

```
// Inside some of the Network methods implementation...
std::cout << "Ping delay = " << settings.ping_delay << std::endl;
```

❶ Variables managed by `Settings` classes are always stored in subtree of the `/Settings` node. The rest of the path is specified in `Settings` constructor parameter. In this example, all settings variables will be stored in the `/Settings/Network` registry node.

❷ This parameter specifies name of registry variable that will be bound to `ping_delay`. In this case, `ping_delay` will be stored as `/Settings/Network/ping_delay` in the Registry.

Warning

Obviously, the Registry must be initialized *prior* to initialization of any class using the `Settings`.

19.3.5. Statistics

Sometimes you need to watch some values and their changes in time, etc. The `Statistics` class provides the common interface for doing so.

It works very much the same as the `Settings` and thus we won't repeat all the information and examples from the previous section. For more information, please refer to the Massiv Core Reference Guide. Note that the client class is responsible for updating the statistics values.

Nevertheless, we will mention one useful feature of this class, measuring of time differences. Imagine you are writing some lengthy routine using the Core and you need to watch how long this routine runs. The following example shows how to do so (let's assume that `SomeClass` has the `statistics` containing the `time_difference` float variable):

```
void SomeClass::lengthy_method()
    {
    Statistics::MeasureSystemTime measure( statistics.time_difference );
    // Lengthy computation follows.
    ...
    // System time difference is assigned into the statistics.time_difference variable
    // at the end of the measure validity scope.
    }
```

Note

All the statistics variabes are stored in the subtree of `/Statistics`.

19.3.6. Configuration Files Syntax

See Chapter 26, *Registry Configuration File Syntax* for information about configuration files.

20. Logger Library

20.1. Survey

Logger Library can be used for runtime logging of various messages (debugging, informative, error, etc.). It is proposed to be fully configurable - log messages can be filtered based on several criteria and sended to various destinations such as standard output, file, string and others. These filter rules can be configured during initialization of the library and can be changed anytime.

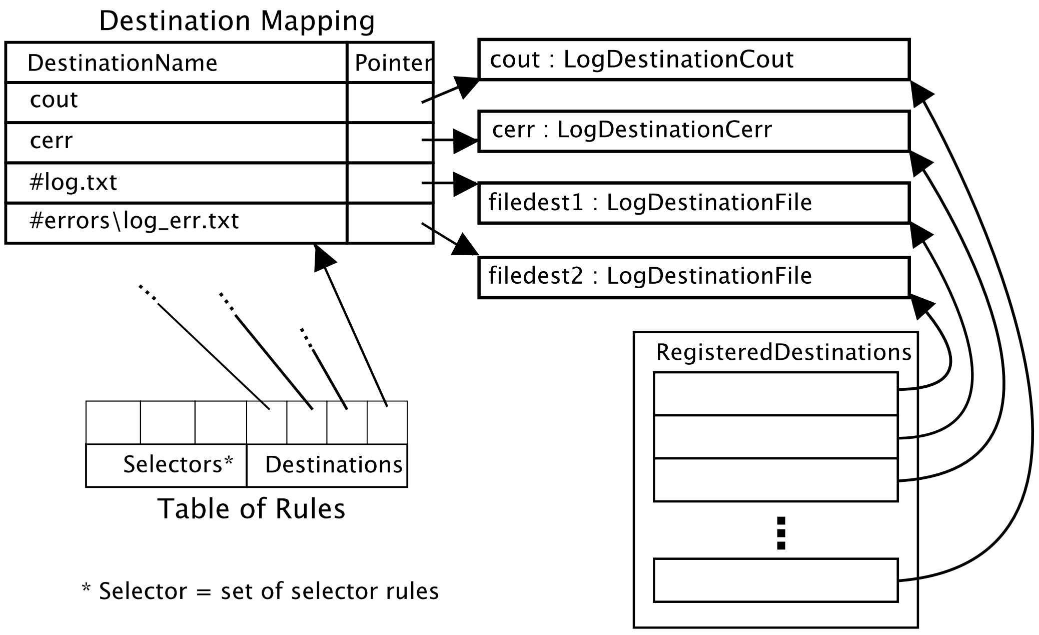
20.2. Logger Structure

Each logger instance firstly maintains a list of *registered destinations*, i.e. all output places for log messages. Moreover, it provides mapping of destination names to destination instance pointers (*destination mapping table*). Note that destination name is not a property of destination class.

Next, the logger has a table of *rules*. A rule is a filter that helps to determine whether each particular message should be sent to each of a specific set of destinations (each message has attributes such as priority, type, facility - i.e. particular part of the Core and status, which bears the information carried in the message). Each rule consists of *selector* and *rule destinations iterators set*. *Selector* works as a predicate - its parameters are the attributes of a log message. If it, applied to a log message, yields *true* the message matches the selector and will be sended to all output destinations pointed to by iterators in the rule destinations iterators set. In either case, next rule is then compared with the message and so on, until all rules have been processed.

Rule destinations iterators set consists of iterators pointing into the *destination mapping table*.

Selector is a set of *selector elements* - atomic conditions that express constrains on log messages (such as “priority greater or lower than some value”, etc.). To match any selector, a log message has to match all of the selector elements contained within.


Destination Mapping
DestinationName
Pointer
cout
cerr
#log.txt
#errors\log_err.txt
cout : LogDestinationCout
cerr : LogDestinationCerr
filedest1 : LogDestinationFile
filedest2 : LogDestinationFile
RegisteredDestinations
Selectors*
Destinations
Table of Rules
* Selector = set of selector rules

Structure of the logger library (instance diagram - example).

Note

The ***`#path/filename`*** string is being used for specification of a log destination determined for the output into a file. Note that the path must be relative, because it will be merged with the path that specifies the *logs root directory*. This root path can be set in the configuration files, see Chapter 27, *Understanding Configuration Values Related to the Core* for more information.

20.3. Usage

After a logger configuration, its usage is quite straightforward - generally just one method is used to process log messages (log message must be created first as an instance of `LogMessage` class):

```
LogMessage    message
                (
                STATUS_DEBUG
                  (
                  Status::FACILITY_TEST, Status::PRIORITY_LOWEST
                  ),
                "msg"
                );

logger.log( message );
```

Note

For your convenience you can use the more comfortable logging interface provided by the `Global` class, see Section 16.1.2, “Global Logging Interface” for more information.

Configuration of the logger is more complicated - the following is the general example of how to configure logger directly; for indirect configuration using registry configuration files see Logger configuration using the Registry.

First of all selectors must be created and selector elements created and inserted into appropriate selectors. Following lines create one selector consisting of two selector elements. The first selector element matches log messages of types `warning` and `error` and with priority greater or equal to `Status::PRIORITY_LOW` and from facility `FACILITY_TEST`. The second one matches messages of any type from any facility having its priority less or equal to `Status::PRIORITY_HIGH`. The whole selector matches all messages satisfying all of the previous constraints.

```
Logger::SelectorElement     selem1
    (
    Logger::SelectorElement::TYPE_GEQ |
                              Logger::SelectorElement::PRIORITY_GEQ,
    Status::FACILITY_TEST,
    Status::TYPE_WARNING,
    Status::PRIORITY_LOW
    );
Logger::SelectorElement     selem2
    (
    Logger::SelectorElement::PRIORITY_LEQ,
    Status::FACILITY_ANY,
    Status::TYPE_ANY,
    Status::PRIORITY_HIGH
    );

Logger::Selector            selector;
selector.insert( selem1 );
selector.insert( selem2 );
```

Initialization of logger itself follows: insertion of rules first.

```
Logger & logger = ( Global::logger() );
logger.clear_rules();

Logger::RuleIterator    ri;
ri = logger.insert_rule( selector );
assert( ri != logger.end_rule() );
```

Destination registration:

```
if( !logger.register_destination( logdest ) )
    {
    delete logdest;
    throw ExLoggerError( "Cannot register 'logdest'." );
    }
```

After the rules has been inserted into logger and destinations have been registered, each rule must be associated with its destinations (i.e. destinations for all messages matching each specific rule):

```
logger.insert_destination( ri, "test" );
    // lets suppose "test" is name of the logdest destination, i.e.
    // logdest's get_name() method returns "test" string.
```

Logger supposed it will be uninitialized:

```
logdest->reset();
logger.clear_rules();
logger.unregister_destination( logdest );
logdest = NULL;
```

20.4. Logger Configuration Using the Registry

Note

See Configuration Files Syntax for general information about how to write configuration files; see Registry for more information about Registry.

Logger instance can be initialized from registry using method `initialize_from_registry()` supposing that the registry itself has already been initialized and has read its configuration files.

20.4.1. Registry Nodes Involved Into Logger Configuration

The main node is `Logger`. If this node is not contained in the registry, the default settings are applied - i.e. all log messages are sent to `cout`. Otherwise the default settings are not considered.

If the registry contains the `Logger` node, it must contain also all of the following:

- `Logger/Destinations` node: describes all destinations that should be registered with the Logger.
- `Logger/SelectorElements` node: describes all selector elements that may be included into selectors.
- `Logger/Selectors` node: describes all selectors, i.e. their elements and destinations.

20.4.2. Logger Destinations Declaration

Logger destinations declaration is contained in the `Logger/Destinations` node. It is a set of strings, one for each destination. All of the strings may have arbitrary name, important is the strings' contents. See the following table for the full description:

Table 20.1. Logger Destinations

String contents	Meaning
cout	Standard output
cerr	Error output
#file	File, filename consists of path and filename.

Note

The `cout` destination is registered by default. However if you load the logger configuration from the registry, `cout` won't be registered unless it is contained in the configuration.

Example 20.1. Logger destinations declaration example

```
[ Logger/Destinations ]
dest1 : string = "cout"
dest2 : string = "#net.log"
dest3 : string = "#thrash/garbage.log"
```

The previous lines register standard output, `net.log` and `thrash/garbage.log` files as logger destinations.

Note

For the file log destination there should not be specified an absolute path, because the string stands for only a name of a log. There might be added some other path (path to the log directory, for example) before it sometimes. This would make the path invalid if the string in the configuration was an absolute path.

20.4.3. Selector Elements Declaration

Selector elements declaration takes place in the `Logger/SelectorElements` node. This node should contain a set of subnodes - one for each selector element - of arbitrary names. Each of the subnodes consists of up to four variable declarations: `priority`, `type`, `facility` and `negate`. Priority, type and facility are each log message's atributes - according to them the message is being filtered.

Priority and type are strings containing a comparison sign and a keyword. Comparison sign of one of <, >, <=, >= and =; possible keywords are described in the following table (see also `src/core/status/status.h`):

Table 20.2. Logger Selector Attributes

Settings	Possible keywords
priority	lowest, low, normal, high, highest
type	any, debug, info, warning, error

`facility` contains just a name of some facility. At present, posible facilities are: `any`, `generic`, `system`, `io`, `logger`, `network`, `registry`, `object_management`, `object_localization`, `archivation`, `crypto`, `file_system`, `lru_cache`, `synchronization`, `thread`, `data_management`, `time_management`, `test`, `node_database`, `archive_database`, `node_management` and `replication`. See `src/core/status/status.h` for complete list.

`negate` is a boolean variable. If true, a log message matches the selector element if it doesn't match any of `priority`, `type` or `facility` constraints.

Example 20.2. Declaring two selector elements of names `Element1` and `Element2`

```
[ Logger/SelectorElements/Element1 ]
priority        : string = ">= low"
type            : string = "<= warning"
facility        : string = "any"
negate          : boolean = false

[ Logger/SelectorElements/Element2 ]
priority        : string = ">= lowest"
type            : string = "= info"
facility        : string = "network"
negate          : boolean = false
```

20.4.4. Selectors Declaration

Each selector consists of a set of selector elements, each specifying an atomic condition, and set of destinations. All selector definitions are contained in the `Logger/Selectors` node; this node contains none or more subnodes (each for one selector) of arbitrary names. Each of these subnodes contains two more subnodes `Elements` and `Destinations`.

`Elements` subnode describes which selector elements should be included into this selector. It is a set of strings of arbitrary names - each string should contain name of one element declared under the `Logger/SelectorElements` node.

`Destinations` subnode determines all destinations the messages matching this selector will be sent to. Its items are either strings containing name of destination declared in the `Logger/Destinations` node or symbolic links refering to these variables.

Example 20.3. Selectors declaration

> **Note**
>
> Let's suppose all declarations from previous examples are valid.

```
[ Logger/Selectors/Selector1/Elements ]
element1        : string = "Element1"

[ Logger/Selectors/Selector1/Destinations ]
destination1    : symlink = Logger/Destinations/dest3
destination2    : string = "cout"

[ Logger/Selectors/Selector2/Elements ]
element1        : string = "Element2"

[ Logger/Selectors/Selector2/Destinations ]
destination1    : string = "#net.log"
```

In this example there are two selectors declared - `Selector1` and `Selector2`. The first of includes selector element declared as `Element1`, the second includes `Element2`. The first selector directs all matching messages to `#thrash/garbage.log` (`dest3`) and to `cout`, the second one sends messages to `#net.log`.

20.4.5. Example

Example 20.4. Configuration using the registry - complex example

```
# Note that this file should be a part of the main massiv
# configuration file or it should be included (directly or
# indirectly) using the !include directive.

[ Logger ]

#################################################
# Logger destinations declaration section
##########

[ Logger/Destinations ]
dest1 : string = "cout"
dest2 : string = "#net.log"
dest3 : string = "#thrash/garbage.log"



#################################################
# Selector elements declaration section
##########
```

```
### 'Element1' declaration ###
[ Logger/SelectorElements/Element1 ]
priority        : string = ">= low"
type            : string = "<= warning"
facility        : string = "any"
negate          : boolean = false

### 'Element2' declaration ###
[ Logger/SelectorElements/Element2 ]
priority        : string = ">= lowest"
type            : string = "= info"
facility        : string = "network"
negate          : boolean = false



#################################################
# Logger selectors declaration section
##########

### 'Selector1' declaration ###
[ Logger/Selectors/Selector1/Elements ]
element1        : string = "Element1"

[ Logger/Selectors/Selector1/Destinations ]
destination1    : symlink = Logger/Destinations/dest3
destination2    : string = "cout"


### 'Selector2' declaration ###
[ Logger/Selectors/Selector2/Elements ]
element1        : string = "Element2"

[ Logger/Selectors/Selector2/Destinations ]
destination1    : string = "#net.log"
```

21. Massiv Filesystem

The Massiv filesystem is a concept that enables to store many streams of data in a single file (managed by a *host* filesystem such as Fat32, NTFS, etc.). Each such file is called an *instance of the Massiv filesystem* or simply a *volume*. The streams inside a volume are managed the similar way as in some of the "real" (native) filesystems, but using the Massiv filesystem is likely to be more efficient than storing each stream of data in a separate file on the host filesystem.

There are two types of volumes - either a standard `Massiv::Core::Volume` or `Massiv::Core::CompactVolume`. The later stores the streams in a compressed form (using the same method as gzip).

In the Massiv the volumes are used for several purposes. For example, the compressed volumes are utilized for archivation (more described in Chapter 23, *Archivation and Startup*), whereas uncompressed ones are useful as *swap files* (to store managed objects that has been swapped out of RAM because they had been unused too long).

The Massiv provides an unique interface that enables to access streams on all volumes and the host filesystem (or any other "real" one) the same way. Moreover this interface is *platform independend*. It is implemented by `Massiv::Core::VolumeManager`. Only one instance of the volume manager is permitted. Nevertheless, you never need to access the instance explicitly, because almost all its methods are declared `static`. Volume manager works with special streams that are derived from the standard iostreams ones and that are defined inside the `Massiv::Core::VolumeManager` streams. See below for a self-explanatory example.

Note

You can use the `Massiv::Core::FileSystem` interface not only to open streams, but also to work with the directory structure (create or remove directories, etc.) See the Massiv Core Reference Guide for the complete information.

Before the volume manager can take any volume into account, it must have been *mounted*. The manager virtually creates a single filesystem that encapsulates the native filesystem and all mounted volumes (all of them are mounted always to the root of the volume manager filesystem). When requested to open some stream, the manager searches the native filesystem first and then all volumes in the same order as they have been mounted. To force it to ignore all filesystems/volumes except for one, prefix the stream name with either ***`volume_name`*****`://`** (for the mounted volumes, replace *`volume_name`* with the actual name of the relevant volume) or **`native://`** for the native filesystem.

Note

Each volume has an associated name that is specified when the volume is created. This name can be different from the volume image file and represents the volume in the volume manager's virtual filesystem.

The following example shows the basic usage of the volume manager, streams and the `FileSystem` interface. You should prefer this interace from the standard C++ API, because it is more general and platform independent.

Example 21.1. Volume manager usage

```
VolumeManager::mount( "volume1_image" );  ❶
VolumeManager::mount( "volume2_image" );
VolumeManager::format( "volume3_image", "volume3" );  ❷

VolumeManager::IFStream   ifs( "stream_path/stream_name" );  ❸
  ...work with the stream the same way as with a iostream...
ifs.close();  ❹

VolumeManager::OFStream   ofs( "native://stream_path/stream_name" );  ❺
  ...work with the stream the same way as with a iostreams...
ofs.close();

VolumeManager::OFStream   ofs( "volume1://stream_path/stream_name" );  ❻
  ...work with the stream the same way as with a iostreams...
ofs.close();

FileSystem &  fs = VolumeManager::file_system();  ❼
fs.create_directory( "the/directory/path" );  ❽

...

VolumeManager::dismount( "volume1_image" );  ❾
VolumeManager::dismount( "volume2_image" );
```

❶ These lines mount two volumes, the first stored in `volume1_image` and the second in `volume2_image` files on the host filesystem. Let's suppose that the first volume has name `volume1` and the second `volume2`.

❷ This way you can create a new empty volume that would be stored in the `volume3_image` file.

❸ This line shows how a stream can be opened. A general stream name is given here. Thus, more volumes can be potentially searched for it. The order is: the native filesystem first, `volume1` second, `volume2` third and finally `volume3`.

❺ Here the same stream is opened for writing. It will be searched for *only* on the native filesystem; all volumes will be ignored.

❻ This will open the stream on `volume1` and ignore all other volumes and the native filesystem. Note that the volume name must be given, not the image file name.

❼ `fs` will contain a reference to the volume manager's filesystem interface that provides access to all mounted volumes as well as to the native filesystem.

❽ This is only a small example of what you can do using the `FileSystem` interface. Note that even the `create_directory()` is more powerful than most of OS API equivalents, because it enables to create more nested directories using a single call.

This specific call creates a new directory on the `volume3` volume.

❾ This dismounts the volume from volume manager. It won't be taken into account any more.

You can also handle (pack, unpack, list) volumes manually using some extern utilities. See Chapter 25, *Auxiliary Utilities* for complete information.

22. Command Line Parser

22.1. Survey

Command Line Parser is a separate library providing a simple Unix-style command line parsing.

Command line is what you write for launching an application, i.e. name of the desired executable file followed by directives that specify how the application should behave.

This library supports the following form of directives:

```
application_name <switches section> <other parameters section>
```

Switches section consists of one or more switch sequences, where switch sequence means either a simple switch - a dash followed by a letter (**`-m`**, for example), or a parameter switch - simple switch followed by parameter, not neccessarily separated with blanks (i.e. **`-f <filename>`** or **`-ffilename`**, which has the same meaning). Moreover, switches may be grouped together. For instance, **`-g -h -i`** switches section has the same semantics as **`-ghi`**, where both **`g`** and **`h`** must be a non-parameter switches. **`i`** can be a non-parameter or parameter switch, in which case it should be followed by its parameter (**`-ghi parameter`** or even **`-ghiparameter`**).

Parameter section can be terminated in two ways:

- by a sequence not beginning of **`-`** that isn't a parameter of some parameter switch.
- by the **`--`** sequence.

Other parameters section is a list of arbitrary words that are passed 'as they are' to the program (with one exception - if there is some string enclosed into quotes, the quotes are removed before passing it to the program).

22.2. Usage

To use the command line parser utility, you have only to create an instance of the `CmdLineParser` class and initialize it with the `argv` and `argc` parameters passed to the `main()` function, plus two strings containing all parameter and non-parameter switches, respectively. Then call the

```
get_next( std::string & parameter )
```

routine in a loop until the parsing is finished. `get_next` returns:

- a letter specifying a non-parameter switch (*`parameter`* is will be empty).
- a letter specifying a parameter switch (*`parameter`* will obtain a parameter string).

- a − character - this indicates that the switch section has already been terminated; `parameter` obtains an argument string.
- a `\0` character (ASCII 0) - stands for the end of parsing; no more switches or parameters available.

Example 22.1. Cmdline parser usage example

```
int main( char ** argv, int argc )
    {
    try
        {
        CmdLineParser cmd_line( argc, argv, "vs", "f" );
            /* This initialization line sets up a command line parser
             * instance and makes it consider 'v' and 's' a
             * non-parameter switches, whereas 'f' a parameter one.
             */

        bool parse = true;
        while( parse )
            {
            std::string parameter;
            const char option = cmd_line.get_next( parameter );
            switch( option )
                {
                case '-':
                    /* Switches section has already been terminated.*/
                    process_parameters( parameter );
                    break;

                case '\0':
                    /* End of parsing. */
                    parse = false;
                    break;

                case 'v':
                    process_switch_v();
                    break;

                case 's':
                    process_switch_s();
                    break;

                case 'f':
                    /* 'f' is a parameter switch - therefore
                     * 'parameter' variable must now contain the
                     * parameter string.
                     */

                    process_switch_f( parameter );
                    break;

                default:
                    assert( 0 );
```

```
                  /* Invalid switch not caught by exception. */
            }  // switch
        }  // while
    }
    catch( ExMissingSwitch & )
        {
        /* There was only '-' on the command line, but the
         * character switch didn't follow.
         */
        ...
        }
    catch( ExUnexpectedCommandLineSwitch & )
        {
        /* There was a character switch that has been defined
         * neither in the parameter nor in the non-parameter
         * switches string.
         */
        ...
        }
    catch( ExMissingCommandLineArgument & )
        {
        /* There was a parameter switch on the command line, but
         * it was not followed by its argument.
         */
        }

...  //Continue processing of the 'main' routine.

return 0;
}  // main
```

Part IV. Administration

The Massiv provides some tools you can use for the system administration, such as configuration or cryptography keys generators, special archives viewers, and others. This part provides a survey of such tools, together with a necessary piece of theory you need to know to understand what these tools are really for.

Table of Contents

23. Archivation and Startup

23.1. Archivation

The archivation is very important in distributed systems. The Massiv uses a global archivation where the *global* means that it runs simultaneously on all of the servers and the whole simulation state is being archived - every server saves its own part of the simulation (which it is currently managing). The archivation process starts periodically on every involved server and runs transparently in the background and thus does not block server functions at all. Once the achivation is started, all servers are notified about this action and the global archivation starts. Archives are stored in the server's `./archives/regular/` directory as `massiv_`*`XXXXXXXXXX`*`.archive` files where *`XXXXXXXXXX`* represents the number of the created archive (e.g.`0000000001`). Because of high amount of the created files, old files are consecutively deleted and only recent ones are kept. After the archive creation, an acknowledgement file `massiv_`*`XXXXXXXXXX`*`.archive.__ack__` is created, which serves for the archive validation. Of course, the archives have its own validation system, but this created file means that no error occured during the archivation and data were successfully saved. Created archive files are descendants of the Massiv volumes and can be unpacked or listed by auxiliary utilities (see Chapter 25, *Auxiliary Utilities*). Note that because of archive's extended properties, it cannot be repacked as a new valid archive by the utilities and the `System` API (or the Demo server executable) must be used instead (see Section 16.2.1, “Archive Management API”).

For description of archivation configuration in brief implementation notes, please refer to Section 27.2, “Archive Database” and Section 27.3, “Archive Manager”.

23.2. Simulation Startup

Each started server enumerates all saved archive files that are present in the `./archives/regular/` directory and checks whether they are valid. Then all servers cooperate and negotiate the number of the latest global valid archive they would use for the simulation startup.

A special case of the server startup is the first time simulation launching which is realized from pregenerated server archives. One server must have initial archive containing a "special object", which can create the whole simulation, other servers have empty archives just for their own startup. As soon as the simulation is created from the "special object", the load-balancing feature ditributes objects to all servers where they can be then archived by the local transparent archivation.

For further instruction about the configuration generation please refer to Chapter 25, *Auxiliary Utilities*.

23.3. Adding or Removing Servers

There may be reasons why an existing server should be removed from the simulation or a new one added while preserving the simulation state. Since all servers must have valid archives or the simulation can not start the process of adding or removing servers is tighly coupled with archive management:

- Adding a new server

  Using the supplied command line utilities generate the configuration for the new server and register it into the node database (requires editing of the `source_data/config_nodes_public` public node data set).

  Using the Demo server executable create an empty startup archive for this server.

- Removing a server

  Unregister the server from the node database (requires editing of the the `source_data/config_nodes_public` public node data set).

  Find out the latest valid archivation id (we will call it *`id`*) and locate a different server that will remain registered in the simulation. Replace its archive number *`id`* by a merge of this archive and the archive number *`id`* on the server that is just being unregistered. To merge the two archives, use the Demo server executable to unpack them, each one to a different directory, and pack them together to a single new archive.

24. Data Service

To enable even users with a slow internet connection to start playing as soon as possible, it is important to avoid making them download many megabytes of data needed for the presentation before they can connect. The first way is to distribute all the data on a CD. This has many drawbacks, in particular problems with the distribution to the potentially "random" locations in the world. The second way is to enable them downloading the data dynamically, when they need them, during playing the game.

The class `Massiv::Core::DataManager` implements the functionality of the data management. It provides methods which can be used to work with data such as textures, sound data, maps, texts and other textual and binary data.

A global instance of the class `Massiv::Core::DataManager` is created on each Massiv node and plays a different role according to the hosting node type (*Master data service*, *Server* or *Client*).

- On the Massiv server or client node, the data manager downloads data from the master data service node, and offers them to an user.
- On the Massiv data service node, the data manager's task is to upload the data to other Massiv nodes. When a client or server connects to a simulation, it contacts the relevant master data service to synchronize the data with it.

On a specific node the data is stored in a subtree of one separate directory in files. This directory is called the *data directory*. The files can be located in subdirectories of the data directory and are called the *data files*. A data file name from the point of view of the data manager is not just a plain file name, but it is a relative path to the data directory.

Besides the data files, there are also two files in the data directory that describe the whole data system (in a textual way). These files are called `.description` and `.filelist`. On the master data service node, there is even another directory called the *source data directory* and a textual description of the source data in the file `.update_description`. The structure of this file will be described below. It is the only file that must be changed manually by the administrator; the other two mentioned files are updated automatically.

An administrator of the Massiv data server modifies the data in the source data directory (for example adds new data files, modifies the existing ones, modifies the source data description file, etc.). After that he should update the changes from the source data directory into the data directory. This update is processed during the master data service startup, or when the method `DataManager::request_master_data_update()` is called. After data update information about update is propagated to servers and clients they will update their local data.

The administrator needs to edit file `.update_description` which describes the source data system. See section Structure of .update_description file.

Each data file is *versioned* (i.e. contains information about its version) as well as the whole data system. This is essential for the data manager to be able to find out if some data has changed (new texture added, old texture modified, etc.). Clients and servers compare their local version information with the master data service to decide if it is necessary to download some new data.

24.1. `.update_description` structure

To be able to create and modify `.update_description` file it is necessary to know the structure of this file. Let's look at an example to understand.

```
###
# Images
node image_default
    type necessary_client
    file "image/default.png"

    node image_font_console
        type necessary_client
        file "image/font.png"
    endnode

    node image_background
        type client
        file "image/background.jpg"
    endnode
endnode # image_default

node model_default
    type necessary_client
    file "model/default.md2"

    node monster
        type client
        file "model/monster.md2"

        node player
            type client
            file "model/player.md2"
        endnode
    endnode # monster
endnode # model_default


###
# Data objects
dataobject image/default = /image_default
dataobject image/font_console = /image_default/image_font_console
dataobject image/background = /image_default/image_background
dataobject model/default = /model_default
dataobject model/monster = /model_default/monster
dataobject model/monster_dark = /model_default/monster
dataobject model/player = /model_default/monster/player
```

The structure consists of two parts - *data nodes* and *data objects*. The data nodes create a tree. Each *data node* is associated with exactly one *data file*.

A data node ancestor in the tree can substitute its descendant data nodes. This mechanism enables to use more general data temporarily until the specific data is actually downloaded. For example, in the tree above, `image/default.png` would be used instead of `image/background.jpg` until the later would be downloaded by the relevant client.

Data objects are mapped to data nodes, usually leaf data nodes (FIXME BUG TO SE MI TEDA NEZDA!). If a programmer wants to obtain a data object, the data manager algorithm searches for the first usable data node on a path from the corresponding data node to the root data node. It is guaranteed that the root data node is always present in the data system of each client (i.e. it never needs to be downloaded).

The consequence of the described mechanism is that the player can see (for example) "walking black-and-white boxes" instead of the real models of figures until the they are actually are available. This is a big advantage, because any player can start playing very quickly. However, if he doesn't like the default models and images, he can still wait until all the data is available.

Let's describe structure of a file. Comments are parts of lines starting with the # character and ending at the end of the line. A data node declaration begins with the node keyword and ends with keyword endnode.

Between these keywords there are located properties of the relevant data node as well as other (submerged) nodes. Submerging of the data nodes declarations is the way how to create the tree.

Each data node has several mandatory properties:

- *name* stands for a data node name; it must be located on the same line after the node keyword.
- *file* specifies a name of the data file associated with the relevant data node; only one data file is allowed in one specifiec data node.

There exist also some optional properties:

- *type* specifies the data node type; the possible values are `necessary_client`, `necessary_server`, `client`, `server`; there can be several types for one data node; the purpose of this property is to declare which data is client or server specific and which is necessary for the simulation.
- *obsolete* declares the relevant data node as obsolete; it cannot be used in the simulation any more.

The second part of the `.update_description` file consists of the data objects. A data object definition begins with the dataobject keyword followed by the data object name (the / character is permitted), = and the data node path to the existing data node. Don't forget to use space between all of the parts.

Note

Two data objects can be mapped to the same data nodes. For example models of two characters are identical, but textures will differ.

Note

Data files are unique for data nodes. Different data nodes must own different data files.

24.2. Data Objects

A data object is a C++ object, implemented by the class `Massiv::Core::DataObject`, which provides an interface needed to handle the data. A programmer needs to know a name of the data object he wants to use. To obtain the data object, call `DataManager::get_data_object()` method. The `DataObject` object has an attribute `data`, which is inherited from the class `std::ifstream` and can be use to read the relevant data from a stream using the stadard C++ interface.

The following example shows how to get texture data (represented by an image file).

```
DataManager &   data_manager    = Global::data_manager();
DataObject *    warrior_texture = data_manager.get_data_object( "warrior_texture" );
warrior_texture->open();
  // read data from attribute warrior_texture->data
  // using the data member
warrior_texture->close();
```

The point is that the programmer doesn't need to know what data will be contained in the data object. All he must know is just the data object name.

24.3. Download/upload speed

Data download speed (server and client nodes case) and data upload speed (data service case) is configurable, see section configuration of the data manager

25. Auxiliary Utilities

25.1. Short info about auxiliary utilities provided with the Massiv

Several simple administration utilities are provided with the Massiv. Source of the utilities are located in the `src/utility` directory. The following table summarizes all the utilities:

Table 25.1. Utilities

Name	Function
gencfg	Generate configuration for a set of nodes
generate_key_pair	Generate RSA key pair
pwd_encrypt	Encrypt a file
pwd_decrypt	Decrypt a file
pack_volume	Pack files into a volume
unpack_volume	Unpack files from a volume
list_volume	List contents of a volume

The following table lists names of the utility binaries when installed on Unixish systems. Different names are used to prevent name clashes. If you want to install the utilities into directory specified in the `PATH` environment variable on Win32 systems, you should probably rename them in similar way.

Table 25.2. Un*x utility binary names

Utility Name	Name of the Binary
gencfg	massiv-gencfg
generate_key_pair	massiv-genkey
pwd_encrypt	massiv-encrypt
pwd_decrypt	massiv-decrypt
pack_volume	massiv-pack
unpack_volume	massiv-unpack
list_volume	massiv-list

The following pages contain thorough description of the individual utilities.

Name

gencfg -- generate configuration for a set of nodes

gencfg

Synopsis

`gencfg` [OPTION...]

Description

Create simulation config files for one data service node, given number of server nodes and one client node with administrator privileges.

The following options are recognized:

`-s` *`N`*
: Create configs for *`N`* server nodes.

`-i` *`service_ip service_port server1_ip server2_port`* ...
: IP addresses and names for service and server nodes.

`-n`
: Generate simulation config files for Nullsoft Installer (NSIS).

`-p`
: Set administrator password for client account.

Example

Generate config files for 2 server nodes, one client and one service node, using localhost IPs for the server and service nodes:

gencfg `-s 2`

Notes

You should not have to use this utility to setup the Massic Demo. Use binary installation under Win32 and the **massiv-setup** utility under Linux (see Section E.1.4, “Compilation under Linux (GCC)”).

Known Bugs

The optional password is passed on the command line, which is not very secure.

Name

generate_key_pair -- generate RSA key pair

generate_key_pair

Synopsis

`generate_key_pair` seedlen

Description

Generate a pair of RSA keys. The keys will be saved in binary form into files `rsa_priv.rsa` and `rsa_publ.rsa` and as a part of massiv configuration file into `rsa_keys.conf`. The *`seedlen`* option specified key length in bits.

Name

pwd_encrypt -- encrypt a file

pwd_encrypt

Synopsis

`pwd_encrypt` filename password

Description

Encrypt file *`filename`* using given *`password`* and write encrypted file into *`filename`*`.crypted`.

Note

This utility is used to encrypt login configuration files in the Massiv Demo.

Known Bugs

The password is passed on the command line, which is not very secure.

Name

pwd_decrypt -- decrypt a file

pwd_decrypt

Synopsis

`pwd_decrypt` filename password

Description

Decrypt file *`filename`* using given *`password`* and write decrypted file into *`filename`*`.decrypted`.

Note

This utility can be used to decrypt Massiv Demo login configuration files.

Known Bugs

The password is passed on the command line, which is not very secure.

Name

pack_volume -- pack files into a volume

pack_volume

Synopsis

`pack_volume` [OPTION...] volume file...

Description

Pack given files into volume *`volume`*.

The following options are recognized:

`-c`
: Use compact volume file format. Currently format of existing volumes is not recognized automatically. Note that Massiv archives are stored in compact volume format.

`-e`
: Erase source files after being packed.

`-f`
: Format (clear) the volume first. This option must be used when creating a new volume file.

`-h`, `-?`
: Show help.

`-m` *`mask`*
: Wildcard mask used to filter files when recursion is used.

`-n` *`name`*
: Internal name of the archive. Used only when a new archive is created.

`-p` *`path`*
: Destination path in the volume.

`-r`
: Pack directories recursively.

`-R`
: Pack directories recursively, do not use mask specified by the `-m` option to match subdirectory names.

Example

Pack all `*.png` files from directory `images` to a new compact volume `volume.image` called *IMAGES*:

```
pack_volume -f -c -n IMAGES -m '*.png' -R volume.image images
```

Name

unpack_volume -- unpack files from a volume

unpack_volume

Synopsis

`unpack_volume` [OPTION...] volume path...

Description

Unpack files from given volume directories (*`path`*) of volume *`volume`*.

The following options are recognized:

`-c`
: Use compact volume file format. Currently format of existing volumes is not recognized automatically. Note that Massiv archives are stored in compact volume format.

`-d` *`path`*
: Directory where to unpack the files. Default is current directory.

`-e`
: Erase source files after being unpacked.

`-h`, `-?`
: Show help.

`-m` *`mask`*
: Wildcard mask used to filter files when recursion is used.

`-r`
: Pack directories recursively.

`-R`
: Pack directories recursively, do not use mask specified by the `-m` option to match subdirectory names.

Name

list_volume -- list contents of a volume

list_volume

Synopsis

`list_volume` [OPTION...] volume

Description

List contents of volume *`volume`*.

The following options are recognized:

`-c`
: Use compact volume file format. Currently format of existing volumes is not recognized automatically. Note that Massiv archives are stored in compact volume format.

`-d`
: Print directory names even in non-recursive mode.

`-h`, `-?`
: Show help.

`-m` *`mask`*
: Wildcard mask used to filter files when recursion is used.

`-o`
: Omit header with general volume image information.

`-p` *`path`*
: Directory of the volume to list.

`-r`
: Pack directories recursively.

`-R`
: Pack directories recursively, do not use mask specified by the `-m` option to match subdirectory names.

`-s`
: Print stream sizes.

`-w` *`width`*
: Width of the size information column (used with the `-s` option).

26. Registry Configuration File Syntax

This chapter describes the configuration syntax of the registry.

Note

An example of a configuration file can be found in

`massiv/src/core/test/massiv.conf`

Note

Configuration files are fully case-sensitive.

26.1. Variable, Node and Alias Names

Variable, node and alias names can contain none of `!()[]=\"'` nor space nor tab. Moreover, all non-printable (i.e. with ASCII value < 32) are also invalid. Symbol `#` is valid, but, however, it cannot be contained at the beginning of the name, because whole line would be considered a comment.

Path is formed of none of more node names and one variable or alias name, all of them separated by `/` from each other.

26.2. Structure of the File

Configuration file consists of blocks, each block refers to one particular node in the registry; new variables are inserted relatively to this node. However, symbolic link “addresses” are always considered relatively to the root registry node.

Generally, each block looks like this:

```
[ Relative or absolute node definition ]
One or more variables definition...
```

`Relative or absolute` node definition is mandatory even if the following section refers to the registry root node - in such case, it consists only of `[ ]` character sequence.

Variable definition syntax should be used as follows:

```
variable_name : type = value
```

or

```
variable_path/variable_name : type = value
```

`variable_path` relates to the actual relative node in the block. type can be one of following strings, written in lowercase and not quoted:

- boolean ... Typicaly `true` or `false`, but there are also acceptable conversions. `0` yields `false`, whereas all non-zero values `true`.
- int ... Integer value, typicaly 32bit wide.
- largeint ... Integer value, typicaly 64bit wide.
- float ... Floating point number.
- string ... String value.

`value` must satisfy a syntax of each specific type. See examples bellow....

26.3. Additional Stuff

26.3.1. Comments

Comments begin with the `#` sign, continues as far as the end of line; everything that comes after the `value` field is also ignored.

26.3.2. Blanks

Blanks are spaces and tabs and they are ignored; nevertheless, for example blank still cannot appear in the middle of a variable name, as described in section *variable, node and alias names*.

26.3.3. Includes

Include directive has form:

```
!include <path/file>
```

Include must always be placed on a separate line. path should be a relative path to another (so-called nested) configuration file, which will be processed at this point before continuing processing of the original file.

26.4. Example

Example 26.1. Registry configuration file syntax

```
# Beginning of the first section; note that a section declaration must
# come BEFORE the first variable definition!!!

[ ]
host_name : string = "atrey.karlin.mff.cuni.cz"
```

```
    # Defines a host_name variable in the root node of the registry.


# Beginning of the second section
[ foo_vars ]
foo_var_x : int = 13
    # Defines a foo_var_x variable in the foo_vars subnode of the root
    # node.

foo/foo_var_y : largeint = 33333333333333
    # Defines foo_vars/foo/foo_var_y variable.

link_foo_foo_var_y : symlink = foo_vars/foo/foo_var_y
    # Defines a symbolic link to foo_vars/foo/foo_var_y variable.
    # Note that full path must be specified; thus, line
    #           foo/foo_var_y
    # wouldn't be correct as it wouldn't refer to the right variable.


# Beginning of the third section
[ bool_vars ]
var_true : boolean = true
var_false : boolean = false
    # This is how logical values are specified. Just 0 or 1 wouldn't
    # be sufficient.

# Note: we do not implement a strict syntax checking; thus, even the
#       following lines are correct (comments shows the equivalent
#       lines):
var_correct1 : boolean : false = true
    #   Everything between the 'type' field and the first '='
    #   character is ignored. Thus, the line above has the same
    #   meaning as
    #     var_correct1 : boolean = true

var_correct2 ignored : boolean = false ignored
    #   Everything between the 'variable_path/variable_name' and the
    #   first ':' character is ignored. Similary, all that is after
    #   the 'value' field is also ignored. Thus, the line above has
    #   the same meaning as
    #     var_correct2 : boolean = false
```

27. Understanding Configuration Values Related to the Core

Note

For general information about the configuration files and their syntax, please refer to Chapter 26, *Registry Configuration File Syntax*.

27.1. Account Manager

Configuration of the account manager:

```
[ Settings/AccountManager ]
manual_management : boolean = true ❶
```

❶ Set value to true if manual account management is required. So, newly created client accounts are not automatically added to file `source_data/config_nodes_private`, but to file `source_data/config_nodes_private.manual` and administrator must manually merge these two files to add new client accounts to the client account database. Default value is false, that is, newly created accounts are automatically added to the client account database.

27.2. Archive Database

Configuration of the archive database:

```
[ Settings/ArchiveDatabase ]
keep_archives_count : integer = 5 ❶
milestone_divider : integer = 10 ❷
```

❶ A number of most recent archives kept on the disk.

❷ This value defines that every n-th archive will be kept on the disk permanently. Setting this option to `1` will keep all archives created.

27.3. Archive Manager

Configuration of archive manager:

```
[ Settings/ArchiveManager ]
auto_archivation_delay : float = 600 ❶
auto_archivation_enabled : integer = 10 ❷
local_archivation_duration : float = 60 ❸
```

❶ A time period between two occurs of archivation.

❷ Defines whether auto archivation's enabled or not. Setting this option to `true` is higly recommended.

❸ This value defines an amount of time (in seconds) when archive manager waits in idle phase until it initiates a new distributed archivation in case if auto archivation is enabled.

27.4. Balancer

Configuration of the balancer:

```
[ Settings/Balancer ]
enabled : boolean = true  ❶
data_flow_optimization_enabled : boolean = true
resource_load_optimization_enabled : boolean = true
memory_load_optimization_enabled : boolean = true

min_data_run_period : float = 5  ❷
max_data_run_period : float = 120
resource_run_period : float = 10
memory_run_period : float = 60

samples_per_run_fraction : float = 0.02 ❸
size_of_object_samples_fraction : float = 0.05 ❹
data_flow_fraction : float = 0.75  ❺
data_flow_aging_coef : float = 0.9  ❻
object_timeout : float = 30 ❼
cpu_load_weight : float = 0.5  ❽
rank_compare_factor : float = 0.6 ❾
balance_object_count_fraction : float = 0.05 ❿
```

❶ Is Balancer globally enabled? If yes, are given balancer optimizations enabled?

❷ Times between two runs of given balancer optimizations.

❸ Suggested number of object-to-object samples per run relative to the total number of objects. The actual value is obtained by multiplying the constant by `statistics.num_objects`.

❹ Suggested minimum total size of object-to-object samples in a run, as a fraction relative to `memory_load`. The actual value is obtained by multiplying by `statistics.memory_load`.

❺ Minimum data flow fraction that must be consumed by the "major" consumer node in order to trigger group migration to that node. A float value from [0, 1] range.

❻ A coefficient from [0, 1] that determines how much old samples are obsoleted. Their weight will be multiplied by this coefficient.

❼ Swap-out migration group if it has not been touched for `object_timeout` seconds.

❽ Factor used when comparing resource load ranks of two nodes to determine whether one of the nodes is "substantially less loaded than the other node". A float value from [0, 1] range. The lesser values imply the bigger required differences in node ranks.

❾ Factor used when comparing resource load ranks of two nodes to determine whether one of the nodes is "substantially less loaded than the other node". A float value from [0, 1] range. The lesser values imply the bigger required differences in node ranks.

❿ How many objects to migrate in a single resource optimization run. Defined as a fraction relative to the total number of objects registered to the simulation. Must be sufficiently small or the resource optimization will not converge.

27.5. Data Service

Configuration of the data manager:

```
[ Settings/DataManager ]
download_type : integer = 1 ❶
data_service_node : string = "[ Service 1 ]" ❷
max_download_speed : integer = 4096 ❸
check_data_version_interval : integer = 60 ❹
mount_data_volume : boolean = false ❺
data_volume_image_name : string = "data.image" ❻
```

❶ Download type of the data manager. Specifies which data files will be downloaded before node is connected to the simulation. Values can be:

- 0 - no download. Value used for master data service.
- 1 - necessary download. Only data files with a flag "necessary data file" will be downloaded before connecting to the simulation. Default value for server and client nodes.
- 2 - full download. Download all data files before connecting to the simulation.

❷ Node id of a data service node from which data will be downloaded first. Currently only master data service node id is supported.
❸ For server and client nodes - maximum speed of data download in bytes per second. For data service - maximum speed of data upload in bytes per second for a one connection.
❹ Interval is seconds between checks if new data version is available for download.
❺ Set value to true to store downloaded data to a volume. Else data will be stored in a directory. Default value is false.
❻ Define volume image name if data are stored in a volume. Default value is "data.image".

27.6. File Acknowledgement Manager

Configuration of the file ack manager:

```
[ Settings/FileAckManager ]
create_ack_delay : float = 10 ❶
```

❶ Default time delay between ack creation request and ack file creation itself (in seconds).

27.7. Garbage Collector

Garbage Collector related configuration can be separated to two distinct groups:

- Garbage Collector Settings

  Allows to slightly change semantics of the Garbage Collector:

```
[ Settings/ObjectManagement/GC ]
can_run_any_time : boolean = false ❶
```

❶ Set this to true to allow GC to run even when objects are or might be active. Since active objects can not be deleted and are not treated as GC roots this may cause that migration groups will not be deleted atomically. See `System::dispose_gc_root()`.

- Garbage Collector Tuning

  Controls automatic triggering of the Garbage Collector runs. Expert knowledge is required in order to be able to change these settings without causing a serious harm to the application performance.

```
[ Settings/ObjectManagement/GC ]
max_iterations_per_tick : integer = 256 ❶
max_run_period : float = 30 ❷
limit_delta : integer = 50 ❸
max_limit_delta_fraction : float = 2 ❹
thrashing_ratio : float = 0.8 ❺
thrashing_coef : float = 0.2 ❻
lazy_ratio : float = 0.5 ❼
lazy_coef : float = 1.2 ❽
max_unpin_time_delta : float = 1 ❾
```

❶ Maximum number of internal GC iterations within a GC tick.
❷ Maximum time difference between two GC runs.
❸ Trigger next GC run when object count reaches the current limit delta value plus number of local objects at the time of the previous GC run. Adapted at the end of the GC run.
❹ Maximum limit delta propotionally to the total number of local objects.
❺ The GC gets in the "thrashing mode" if number of finalized objects at the last GC run is greater than thrashing_ratio multiplied by the total number of local objects.
❻ Limit delta adaptation coefficient used when the GC is in "thrashing mode".
❼ The GC gets in the "lazy mode" if number of finalized objects at the last GC run is less than lazy_ratio multiplied by the total number of local objects.
❽ Limit delta adaptation for the "lazy mode".
❾ Issue a warning if objects have been active-pinned longer than the preset value (to get information on what active pinning is, consult the Massiv Core Reference Guide). Used for application performance and migration stalls tuning due to object pinning.

27.8. Logger

The logger configuration is more complicated and requires some knowledge of the logger stucture. Therefore it has been separated as a special part of the relevant chapter. See Section 20.4, “Logger

Configuration Using the Registry” for more information.

27.9. Network

Configuration of the network:

```
[ Settings/Network ]
blowfish_key_length : integer = 16 ❶
blowfish_key_timeout : integer = 3600 ❷
max_idle_seconds : integer = 10 ❸
max_pings : integer = 3 ❹
max_receive_bytes : integer = 32768 ❺
max_server_life : integer = 60 ❻
next_ping_delay : integer = 2 ❼
tcp_preallocated_buffers : integer = 10 ❽
udp_preallocated_buffers : integer = 10 ❾
```

❶ Default key length for blowfish ciphre. Higher value increases security, but slows down message crypting and decrypting.

❷ Blowfish key life in seconds. After this time period blowfish regeneration process exchanges new blowfish ciphre. Lower value increases security, but decreases the network performance.

❸ A time period after which `PING_MESSAGE/DISCONNECT_REQUEST_MESSAGE` is sent to the other side. The message type depends on auto close feature setting by the node. When `true`, request to close the connection is sent, otherwise `PING_MESSAGE` to find out, whether the other side is alive.

❹ A amount of allowed ping messages sent to the other side without reply. After this amount of tries, CONNECTION LOST is signalized to the higher layers.

❺ An amount of bytes allowed to receive in one network tick on a connection. Higher value increases network performance, but decreases performance of the Core.

❻ A time period after which server (connection acceptor) closes new accepted connection, when no data exchange appeared in the connection creation process.

❼ A time period between two consecutive sends of `PING_MESSAGE` when no `PING_REPLY_MESSAGE` received.

❽ An amount of the preallocated buffers reserved for the TCP message sending. Higher value increases network performance, but consumes more PC memory.

❾ An amount of the preallocated buffers reserved for the UDP message sending. Higher value increases network performance, but consumes more PC memory.

27.10. Node Manager

Configuration of the node manager:

```
[ Settings/NodeManager ]
block_disconnected_client_timeout : float = 3 ❶
```

❶ How long to block connections from disconnected clients in seconds.

27.11. Node Database

Node database keeps information about nodes the can participate in the simulation and the information how the remote nodes can be contacted from other nodes (published node credentials). This includes network addresses, RSA public keys, account objects of client nodes, etc. It also holds complete local node credentials, including a "private" part that is not published to other nodes. The private parts are read from external login files (see Section 15.4, “Connecting To the Simulation”) when nodes connect into the simulation and are used to authentize them, thus they must be kept in secret.

For each registered node the following public entries are held:

```
node_id : string = "[ Client 1 ]" ❶
rsa_public_key : string = "..." ❷
address : string = "localhost" ❸
port : integer = 0
name : string = "MyClient" ❹
account_object : string = "[ 1 2 0 ]" ❺
```

❶ The node being described.
❷ The node's public RSA key. Together with its private RSA key (stored in the node's login file) used to authentize this node.
❸ The node's address and port. Must be filled for `SERVER` and `SERVICE` nodes so that other nodes can initiate connections to them. `CLIENT` nodes leave these fields blank and they are ignored.
❹ Optional node name.
❺ Client node's account object. See Section 14.3, “Account Object”. Ignored by `SERVER` and `SERVICE` nodes as they do not have associated any account objects to them.

The entries are stored in node data objects or statically in the registry. Usually there are only two static records:

- Master data service node credentials

  `Settings/NodeDatabase/master_data_node`

- Complete anonymous node credentials

  `Settings/NodeDatabase/anonymous_node`

Both are used to download minimal prerequisite data before connecting to the simulation. A node uses anonymous node credentials to authentize self when contacting a master data service to download the prerequisite data. The data service node credentials are also stored statically.

Note

Since there is no login file for anonymous nodes the private part of the anonymous credentials is also stored in the registry. The following fields are added:

```
rsa_private_key : string = "..."
```

The dynamic part of the node database is automatically downloaded as a prerequisite data.

To connect to the simulation under a given identity corresponding login file must be supplied. The information stored in the file (the private part of the node credentials) is combined with the corresponding entries in the node database and complete local node credentials are obtained. The node can then connect into the simulation using these credentials. This is a structure of a login file:

```
[]
node_id : string = "[ Client 1 ]" ❶
rsa_public_key : string = "0Z0\r\x06\t*\x86H\x86\xf7\r\ ..." ❷
rsa_private_key : string = "0\x82\x01S\x02\x01\x000\r\x06\ ..." ❸
```

❶ Local NodeId.

❷ Node's public RSA key. Must match the corresponding entry already published in the node database.

❸ Node's private RSA key. Used to authentize the node.

> **Note**
>
> For information about the configuration generation please refer to Chapter 25, *Auxiliary Utilities*.

27.12. Object Manager

Configuration of the object manager:

```

busy_time_limit : float = 0.25 ❶
debug_force_random_local_migration : boolean = false  ❷
debug_force_swap_out_of_objects_with_scheduled_events : boolean = false  ❸
debug_random_local_migration_probability : float = 0 ❹
log_events : boolean = false ❺
log_process_migration_messages : boolean = false ❻
max_schedule_delay : float = 4 ❼
```

❶ Set `statistics.too_busy` to `true` if time difference between ticks at level zero is greater that this limit.

❷ Whether to enable random local migrations - object that must be delivered to destination object is passed to migration protocol layer even thought destination object is local. Note: Ignored if the Core is not compiled in debug mode.

❸ Whether to swap-out all objects which have scheduled one or more events for current tick. The objects are swapped-out before any event is processed in current tick.

❹ Specifies probability of local migration if debug_force_random_local_migration is true. Should be in 0.0 to 1.0 range, other values may cause random behavior.

❺ Log each *delivered_to()* and *delivery_failed()* events. Should be used only for debug purposes because it can really slow down entire system.

❻ Log each migration message that is beeing processed.

❼ Warn if global schedule has been blocked by a "busy" object for at least `max_schedule_delay` seconds.

27.13. Path Manager

Configuration of the path manager:

> **Note**
>
> Special string *$HOME* can be used in a directory path. Every occurance of this string will be replaced be the home directory of currently logged user. Example: on unix system - */home/username*, on win32 system - *"C:\Documents and Settings\username"*

```
[ Settings/PathManager ]
login_directory : string  = "$HOME/.massiv/client/login" ❶
source_data_directory : string  = "./source_data" ❷
data_directory : string  = "./data" ❸
archive_directory : string  = "./archives" ❹
log_directory : string  = "./log" ❺
work_directory : string = "." ❻
```

❶ Path to a directory with login files.
❷ Path to a directory with source data. Used by master data service only.
❸ Path to a directory with data.
❹ Path to a directory with archives. Used by server nodes only.
❺ Path to a directory with log files.
❻ Path to a directory for temporary files.

27.14. Remote Procedure Call

Configuration of synchronous RPC:

```
[ Settings/System/SRPC ]
min_stack_size : integer = 65536  ❶
num_threads : integer = 1  ❷
optimize_local_calls : boolean = true  ❸
reply_timeout : float = 30  ❹
statistics_period : float = 5  ❺
```

❶ Minimum stack size reserved for the Core, in bytes. If there is less stack space than the preset limit, and there are no free SRPC threads left, synchronous call requests will be rejected.

❷ Number of simulation threads used to implement SRPC. When the Core runs out of free threads, it will start using less optimal technique to implement SRPC - stack recursion. The largest disadvantage of stack recursion is its *last-in-first-out* semantics; no SRPC request serviced by the recursing thread can be finished before the most recent one. If `num_threads` is `1`, the SRPC request can be blocked indefinitely until the Core runs out of stack (see `min_stack_size` above), if too many SRPC requests are made in such environment. Note that this problem can't happen if `num_threads` is at least `2`.

Value of `num_threads` is ignored if the node has been started with the `single_threaded_srpc` flag set to `true` (see the Massiv Core Reference Guide, documentation of the `Massiv::System::StartUpInfo`).

❸ Allow optimizations of local synchronous calls by direct calls? Setting this option to `false` is useful only when debugging the Core. Its value is ignored in optimized builds.

❹ SRPC request will be cancelled if reply is not received in `reply_timeout` seconds.

❺ Time period between two consecutive updates of RPC statistics, in seconds.

27.15. Replication

Configuration of replication protocols:

```
[ Settings/ObjectManagement/Replication ]
client_repflags_no : integer = 0   ❶
client_repflags_yes : integer = 1
server_repflags_no : integer = 0
server_repflags_yes : integer = 1

min_client_update_period : float = 0.2  ❷
max_client_update_period : float = 1
min_server_update_period : float = 0.5
max_server_update_period : float = 2

max_udp_update_size : integer = 1500  ❸
debug : boolean = false  ❹
```

❶ Replication masks are used to determine which objects belong to a replication group and which properties should be serialized during replication. *Their default values are pretty useless* and should be overridden in a configuration file as explained in Section 7.5.4, “Recommended Node Replication Masks”

❷ Minimal and maximal periods between two consecutive replica updates sent to client or server nodes, in seconds. Slower update speeds (longer periods) are used when bandwidth to the nodes seems to be saturated.

❸ If replication update packets is larger than this value (in bytes), TCP will be used to send it instead of UDP. When using TCP, update period may become larger than `max_*_update_period`.

❹ Enable debugging log messages? In release builds, the default value is `false`, in debug builds the default is `true`. If its value is `true`, the messages may still be filtered by standard logger filters.

27.16. Scheduler

Configuration of the scheduler:

```
max_sleep_time : float = 0.05 ❶
min_sleep_time : float = 0.005
sleep_time_decrement_constant : float = 0.01 ❷
sleep_time_increment_constant : float = 0.005
```

```
tick_delta_high_limit : float = 0.02 ❸
tick_delta_low_limit : float = 0.005 ❹
```

❶ Maximum and minimum sleep time.
❷ Constant added to/ subtracted from sleep_time if idle.
❸ If time delta between two ticks is larger than `tick_delta_low_limit`, sleep time will decrease.
❹ If time delta between two ticks is smaller than `tick_delta_low_limit`, sleep time will increase. This is the evidence that there were very little work to do in the previous tick.

27.17. Time Manager

Configuration of the time manager:

```
[ Settings/TimeManager ]
time_synchronization_interval : integer = 10 ❶
```

❶ Server nodes send each other time synchronization messages. This value is an initial value of interval between particular time synchronization messages is seconds.

27.18. Volume Manager

Configuration of the volume manager:

```
[ Settings/VolumeManager ]
flush_period : float = 5 ❶
volume_cache : integer = 2097152 ❷
```

❶ How often the volume manager flushes file systems in seconds.
❷ Cache size per mounted volume in bytes.

```
[ Settings/VolumeManager/Volumes ] ❶
... : string = volume_image
[ Settings/VolumeManager/ReadOnlyVolumes ] ❷
... : string = read_only_volume_image
```

❶ Registry node holding arbitrarily named string config variables. Each variable contains a path to a volume image to be mounted at core startup.
❷ Registry node holding a list of paths to volume images to be mounted read only at core startup.

27.19. Well Known Object ID Database

Configuration of the well known object ID database:

```
[ Settings/WellKnownObjectIdDatabase ]
```

```
naming_service_object : string = "[ null ]" ❶
```

❶ Object id of naming service object.

28. Handling Accounts

When new simulation is created, there are few nodes in the node database - master data service node, server nodes, anonymous node also called anonymous account, and one administrator client node also called administrator account. In this state, no clients can connect to the simulation, because they don't have their own account.

Newly installed massiv client knows information just about master data service node and anonymous account (this information are stored in a file `massiv.server_nodes.conf`) Client connects to the master data service using anonymous account and downloads data necessary for connecting to the simulation (including node database). Node database is managed by master data service and is stored as necessary data object. In this state, client knows all server nodes, but it doesn't have account for connecting to the simulation.

Client sends a request to create account to the master data service, master data service sends a request to some server node to create special object called *account object* (see chapter special objects), after that master data service saves new client account to the node database, updates data and distributes them to massiv servers. In this state, server knows the client and it can successfuly connect to the simulation.

Note

When account creation request is sent from a client, it takes awhile till server nodes update their data with actualized node database. This time can be a few tens of seconds in a case of automatic account management, or more (days) in a case of manual account management, see account manager settings.

Appendix A. Network Layer

A.1. Locking And Unlocking Buffers

The Network subsystem uses both TCP and UDP protocol for communication. More reliable TCP is used in the connection creation process, sending control messages specific for the lower network layer (e.g. *PING*, *DISCONNECT REQUEST*, etc.) and for the migration. The unreliable UDP protocol is used mainly in the object replication process (although if the replication data is large, the replication may use TCP as well). Every time a higher Massiv layer wants to send something to another node, the Network is asked to *lock* a reliable or unreliable buffer.

```
Network      net;
NodeId       nodeid = ...;
Netobuffer * buffer_tcp = net.lock( nodeid, STREAM_RELIABLE,
                                    NET_MIN_USER_MESSAGE, true );
  // This locks a TCP buffer for node nodeid.
  // Then a user message is written into the buffer using the standard <<
  // operator. The resulting buffer can be scheduled to be sent using the unlock()
  // method. The message may be sent either immediately or in the next network
  // tick, according the last lock parameter. In this specific case,
  // the buffer would be sent immediately, because the last parameter is
  // true.

Netobuffer * buffer_udp = net.lock( node, STREAM_UNRELIABLE, NET_MIN_USER_MESSAGE );
  // This locks a UDP send buffer for node nodeid.
  // The buffer will contain an user message (due to the NET_MIN_USER_MESSAGE).
```

There are some permanent buffers (the number of permanent buffers is configurable via settings), which can be used in case they are ready to be locked. In case there is no available permanent buffer, the Network automatically creates a *dynamic* one. When data is written into a buffer, the higher Network layer must unlock the buffer to inform the lower layer that it is ready to be sent.

```
Network      net;
Netobuffer * locked_tcp = ...;
net.unlock( locked_tcp );
  // This unlocks an already locked TCP buffer.

Netobuffer * locked_udp;
net.unlock( locked_udp );
  // This unlocks an already locked UDP buffer.
  // UDP buffer can be also unlocked using the method
  // unlock_unreliable( locked_udp, false ).
  // The false parameter indicates that the buffer is not completely ready
  // to be sent. It's going to wait for an explicit flushing with
  // flush( unlocked_udp ) or for being locked again.
```

The main difference between the TCP and UDP buffers is that in the case of UDP more messages can be added into the buffer, even if it is already prepared to be sent (of course the destination of all the messages must be the same network node). This feature helps to increase the network throughput.

After being unlocked the message body is encrypted using the blowfish ciphre (see Section A.2, “Managing Connections”) and placed into a list of outgoing messages, which will be sent at the end of the network tick. UDP messages also contain the additional CRC checksum of the encrypted message body, to ensure integrity of the data received by the target node.

A.2. Managing Connections

All the communication between nodes is encrypted for higher security. Every node involved in the simulation has an unique RSA key pair for authentication during the connection creation process. After the successful authentication the server generates a *Blowfish ciphre* which is used for both the UDP and TCP buffers encryption. Because this guarantees better performance but lower security than RSA encryption/decryption, this key is periodically regenerated if the connection is used for a longer time (this time value is also cofigurable via settings). The Network implements the connection auto-close feature between some types of nodes (Server-Server or DataService-Server) to save resources and optimize the network performance. Any auto-closed connection still seems to be open to the higher levels; in case there is some request, the connection is transparently established again and the data delivered as if no connection close had ever occured. This feature can be managed from higher layers.

```
Network       net;
NodeId        nodeid = ...;

net.control_connection_auto_close( nodeid, true );
  // This enables auto close feature to node nodeid.
```

When some node wants to auto-close a connection to another node, it sends a request and waits for permission. The reply depends on auto-close connection feature setting on the other node.

A.3. Connection Creation Process

The connection creation process uses the reliable TCP protocol. After the TCP connection has been successfully created, both sides exchange also UDP port number in order to know where to send UDP packets to.

Table A.1. Process of the successful connection creation

Connection acceptor - Server	Connection initiator - Client
Listens on a *well known port*	
	Initiates a TCP connection to the server's well known port (configurable in configuration files).
Accepts the connection and sends `SERVER_PORT_MESSAGE` with the server's NodeId	
	Receives the `SERVER_PORT_MESSAGE` with the server's NodeId and sends the `NODE_ID_MESSAGE` with the client's NodeId.
Receives `NODE_ID_MESSAGE`, generates blowfish ciphre and sends RSA-encrypted `BLOWFISH_KEY_MESSAGE`	

Connection acceptor - Server	**Connection initiator - Client**
	Receives, verifies and decrypts the `BLOW-FISH_KEY_MESSAGE` and sends back RSA-encrypted `KEY_CONFIRMATION_MESSAGE`
Receives, verifies and decrypts the `KEY_CONFIRMATION_MESSAGE`, creates a UDP socket and sends `UDP_PORT_MESSAGE` containing the port number that should be used for sending UDP messages to this node.	
	Receives the `UDP_PORT_MESSAGE` with the server's UDP port and sends back empty `UDP_CONNECT_MESSAGE`. The connection has been created.
Receives the empty `TCP_CONNECT_MESSAGE`. Sends back the empty `TCP_CONFIRMATION_MESSAGE`. The connection has been created and confirmed.	
	Receives the empty `TCP_CONFIRMATION_MESSAGE`. The connection has been confirmed too.

After the synchronnous TCP connection creation process begins the asynchronous UDP setup procedure - server (in the meaning of connection acceptor) must get informed about the client's (initiator's) UDP port. Every second the client sends the `UDP_INIT_MESSAGE` to the server's UDP port until it receives the TCP `UDP_HANDSHAKE_MESSAGE` from the server.

A.4. Configuration And Statistics

For description of the network configuration in brief implementation notes, please refer to Section 27.9, “Network”.

Statistics regarding network are stored under the `/Statistics/Network` registry node.

Appendix B. Load Balancing

The Massiv provides facilities for automatic load balancing. It is implemented by the `Balancer` class which monitors data flow among migration groups and nodes registered into the simulation, performs transparent object migrations in order to decrease network or CPU load, swaps out not-recently-used objects and migrates objects to less loaded nodes in order to keep uniform resource loads. To make the balancing possible transparent object migrations without application's assistance had to be implemented. This is a key feature of the object model and probably the most important object model design goal. As a result objects can migrate not only due to application logic decisions. However the migrations triggered by the load balancing subsystems are completely transparent to the application. If you are not interested in how the load balancer works you can safely ignore this appendix.

B.1. Optimization Techniques

There are three different optimization techniques. They run independently and are often triggered when there is enough information to perform the optimization (for example data flow statistics between migration groups and remote nodes is available):

- *Local data flow optimization*

  The optimizer keeps track of data flow generated by local migration groups and nodes that consume the generated data. Whenever an object migrates, local SRPC call is made or a replication update is sent, the data amount, that would have to be transmitted over the network, is accounted to a migration group responsible for the production of the data flow. For example if an object A issues a RPC call to object B, RPCObject is created and migrated to B. The migration cost of the RPCObject is charged to the object A as the migration was initiated by that object.

  Note

  When data flow needs be accounted its producer must be identified first. To simplify the process the data is always treated as if it was produced by the last object upcalled from the Core. Direct local calls made by the application since the last upcall are ignored. Although the producer may not be identified reliably it will always be an object from the "correct" migration group unless local weak pointers are used to call other objects directly. This semantics is similar to CORBA `_this()`.

  Migration groups that produce data flow consumed by "single major" nodes (it is not directed to other nodes or the consumed portion of the data is negligible with respect to the major nodes) will be migrated to that nodes. This will reduce data flow between the local node and the major nodes.

  The statistics are gathered locally on a per-object basis. When enough data is collected the statistics are aggregated by migration groups and major consumer nodes are identified. Migration groups are then migrated to the major consumer nodes unless it is cheaper to leave them on the local node.

Note

If we were running this kind of optimization only all objects would finally end up on a single node.

- *Global resource optimization*

  Unlike the local data flow optimization whose aim is to decrease network traffic locally, from a local node to remote nodes, the global resource optimization tries to optimize resource load in the global. The optimizer keeps track of resource loads on all nodes and migrates objects among the nodes to ensure uniform loads.

  Each node "ranks" its resource load which is in turn distributed to all other nodes. The most loaded node periodically elects a subset of its local objects and migrates them to the least loaded node. Since the number of objects to be migrated must be sufficiently small (in order to converge), the optimization can be a lengthy process and it may take a while to find an equilibrium.

  Note

  Objects are elected by a random process which implies that the sets need not be "compact" (with respect to some data flow criterion). However if the optimization is combined with the local data flow optimization, the other optimizer will push "companion" objects to that node soon.

- *Local memory optimization*

  Migration groups that were not touched for some time are transparently swapped out and read back into memory when needed, upon a Core request. Swap-out operation is technically equal to a "migration to a swap file", which means that swapped-out objects are not local.

  Note

  This is completely different from memory swapping implemented by various operating systems. Our approach ensures that idle objects do not pollute Core tables and that "working object set" is kept sufficiently small. This allows to build large distributed worlds as most of them would probably be idle.

  The swapping always operates on a per migration group basis. Objects are read into memory when their scheduled events should be dispatched or an object is delivered to them. As a result RPC calls wake up the objects too.

If load balancing must be disabled for some reason one can do so via the appropriate settings in the registry (see Section 27.4, “Balancer”). An another option is to mark classes as `no-balance` (see Section 10.9.3, “Class Attributes”) which prevents the Balancer from performing optimization on instances of such classes. In the most cases `no_balance` IDL attribute is used as a hint to the Balancer that says that instances of the class are transient and will be collected by the Garbage Collector soon thus no optimization is required.

B.2. Configuring the Balancer

The balancer configuration is described in Section 27.4, “Balancer”.

Various statistics can be found under `/Statistics/Balancer` registry node.

Appendix C. Core Requirements

The Core library currently supports Windows (98 or newer) and Linux operating systems. However, all the platform-specific code is separated into special directories so that porting Massiv onto another platform should be quite straightforward.

As for the hardware configuration, you need to meet the following requirements (note that only x86 hardware platform is supported; no other one has been tested):

> **Note**
>
> Massiv has not been tested on big-endian systems and thus probably won't run on them. However, the appropriate modifications to make it running should be quite straightforward.

Minimal hardware configuration

- procesor Pentium I 60Mhz (for very simple applications)

  procesor Pentium II, Duron, Athlon 400Mhz (for other applications)
- 32MB RAM
- arbitrary network card

Recommended hardware configuration

- procesor Pentium III, Duron, Athlon 1Ghz
- 64MB RAM
- arbitrary network card

Appendix D. Auxiliary Utilities to Build Massiv and Documentation

This chapter describes tools provided along the Massiv that can be used to build projects based on the Massiv and to compile Doxygen and DocBook documentation. While you will have to use some of them (such as the IDL processor **factgen.pl**), using most tools described in this chapter is not required by highly recommended.

D.1. Massiv Build Tools

Building application based on the Massiv (and the Massiv Core itself) is not a simple task. Many soure that are required to compile and link the application are automatically generated, mostly from the IDL. If you wanted to use your favorite build tool, you would have to teach it how to call external tools to generate the files. You could write complex makefiles that accomplish that, or generate the files manually and then use less sophisticated build tools.

It's highly recommended to use tools described in this section that do all the hard work almost automatically.

The following text tries to describe the whole build process of a Massiv project, assuming Massiv build tools are used exclusively.

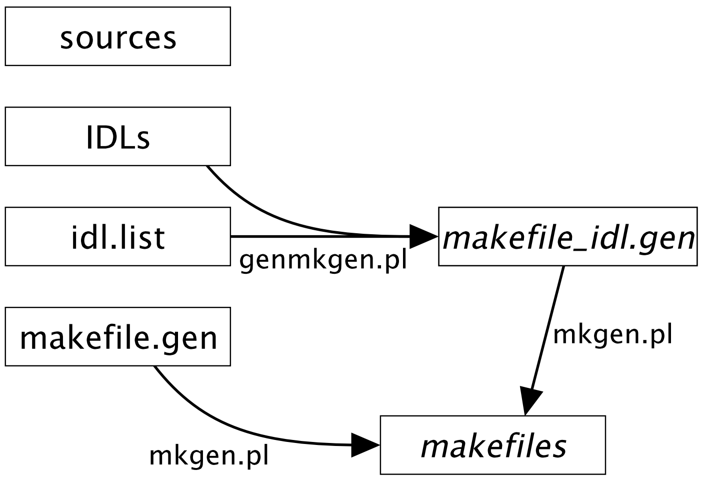
Massiv build tools work with *projects*. A project is either a library (static or shared) or a program. The programmer usually has to write the following files for each project:

- *C++ sources* containing implementation of both standard C++ and Massiv managed classes.
- *IDL descriptions* of all managed classes.
- *`idl.list`* - this files contains list of all IDL files and some additional information. Syntax of the file is described in Section D.2, "The idl.list File".
- *`makefile.gen`* - a file that descibes how to build the project The **mkgen.pl** tool reads this file and generates makefiles for selected target platform.

For example, the Massiv Demo client application contains many C++ sources stored in several directories and no IDL files. It depends on three library projects - the Massiv Core library, and shared and client-side demo libraries. Even though it does not conain any IDL files, it depends on libraries with IDL files and its sources include several generated files from those libraries, therefore it must contain an `idl.list` file too.

To be able to build a Massiv project, makefiles must be generated first. This phase is usually called *configuration*. The most simple configuration consists of two steps illustrated by this figure:

Figure D.1. Configuration phase



1. Generate `makefile_idl.gen` files from `idl.list` files. This step is performed by the **genmkgen.pl** tool. It determines information about dependencies between C++ sources and IDL files written by the programmer and the generated sources, and writes the information to file that must be included from `makefile.gen` in the same directory.
2. Generate makefiles. This step is performed by the **mkgen.pl** tool. It reads all `makefile.gen` files and generates makefiles for specified target platform.

You can use scripts in the massiv source root directory to perform the configuration step. On Windows, run the **configure.cmd** command, on Linux (and other supported Unixish systems), run the **configure** script.

You can various arguments to the **configure** script - you can specify which compiler should be used, where development libraries are located, which compilation and linking options should be used, directories where the binaries and libraries should be installed, and more. Run **configure --help** for detailed help. The script first checks if local environment - if correct compiler version is present, if development versions of all required libraries are properly installed, etc. Then it generates several files required by the build and installation process and runs **genmkgen.pl** and **mkgen.pl**.

The Win32 script is much more simple - it just runs the two tools. Several files automatically generated by the unixish script must be edited manually. For each such file, a file with thesame name and `.example` extension appended exists. It contains default settings and documentation of the settings. The files that must be edited manually are:

- `mkgen/config/msvc.config`
- `src/demo/client/config.h`
- `src/demo/server/config.h`
- `src/demo/service/config.h`

The configuration phase must be perfomed when any `makefile.gen` or `idl.list` is changed, when new sources are added to a project or sources are removed from a project or when source dependencies change.

After the configuration phase, makefiles for all specified target platforms will be created in each project directory. The makefiles can correctly generate sources from IDL as required, compile the sources and build the projects as illustrated by the following figure. All utilities mentioned in the figures are described later in this chapter. The utilities are automatically called from the makefiles, you never have to call them manually.

Figure D.2. Generating sources

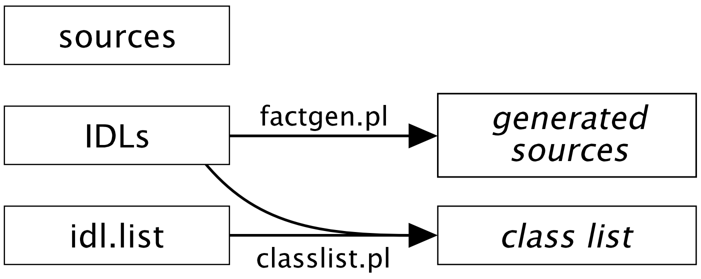
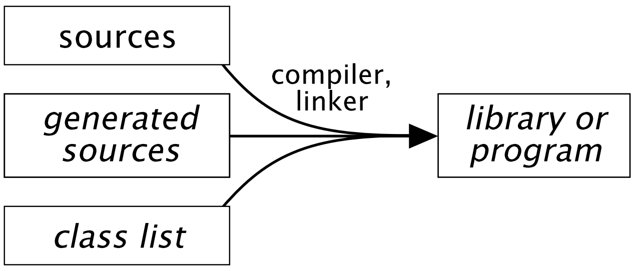


Figure D.3. Compile and link the project



The makefiles define several targets:

- *all* (default target) - build all projects specified in `makefile.gen` in current directory. You can define multiple projects in single `makefile.gen`, but it's not recommended.
- *generate* - generate all sources of local projects.
- *degenerate* - remove all generated sources.
- *library_`FOO`* - build library *`FOO`*. It must be either local library project, or a library local project depends on.
- *program_`FOO`* - build local program project *`FOO`*.
- *`FOO`_obj* - compile object *`FOO`*.obj. The object must belong to a local project. The *`FOO`* should be

only the file name without extension; if the corresponding source is located in a subdirectory, the name of the directory should not be specified.

Note

There is no *clean* target. The *degenerate* target deletes generated sources and no objects, libraries or programs. However, those are stored in completely seprate tree in `.bin` directory in the Massiv source root, so you can easily remove them manually.

The following pages briefly describe all Massiv commands. Most of them all called automatically from the generated makefiles, so they only basic description is provided. All the tools are written in Perl and contain standard perldoc documentation - run **perldoc *`FOO`*** to display documentation of the *`FOO`* command.

Name

mkgen.pl -- generate makefiles

mkgen.pl

Synopsis

`mkgen.pl` {COMMAND} [OPTION...] [directory...]

Description

Generate or clean makefiles from `makefile.gen` files found in specified directories and their subdirectories. If no directories are specified, current directory and all its subdirectories are processed.

Commands

The *COMMAND* must be one of:

`-g | --generate`
    Generate makefile for specified platforms.

`-c | --clean`
    Remove makefiles generated for specified platforms.

Options

Useful (but not all) options are:

`-p | --platform` *name*
    Build or clean makefiles for specified platform.
    This option may be used multiple times to specify multiple platforms. Currently only two platforms are really useful - *unix* and *msvc*.
    If the option is not used, **mkgen.pl** will try to guess local platform.

`-v | --verbose`
    Be more verbose. May be used multiple times.

Syntax of `makefile.gen`

The complete syntax of `makefile.gen` is quite complicated. In this section, only the most useful features will be described. Moreover, we will not differentiate between features implemented directly by the **mkgen.pl**, features defined in per-platform support files (as long as they serve the same purpose on all platforms) and macros defined in files automatically included from each `makefile.gen`. For more information about `makefile.gen` syntax and documentation of **mkgen.pl** in general check contents of the `doc/mkgen` directory. Note that while the documentation is nearly complete, it's quite sketchy and probably hard to understand.

The syntax is line-oriented. Lines beginning with a hash character (`#`) and empty lines are ignored. To split long lines into multiple lines, use a backslash character (`\`) at the end of each line that continues on the next line.

Each `makefile.gen` file should start with lines indicating which libraries are used by all projects described in the file. These settings mainly affect which directories will be searched for system headers:

- *STDCPP* - standard C++ libraries
- *SDL* - Simple DirectMedia Layer libraries
- *OPENGL* - OpenGL and GLU libraries

If a project contains an `idl.list`, a mkgen script describing rules and dependencies required to generate sources from IDL files will be generated from the list. Always include this file into `makefile.gen` using the following line: `include makefile_idl.gen`.

Assignments to lists that apply for all projects described by `makefile.gen` should follow. The syntax of list *`foo`* assignment is:

`foo` `= items`
: Set *`foo`* to `items`.

`foo` `+= items`
: Add `items` to *`foo`*.

`foo` `-= items`
: Remove `items` to *`foo`*.

`items` is list of whitespace separated words. If the list assigned to is list of file name, wildcard patterns can be used in `items`. Wildcard patterns are expanded to list of files that match the pattern in current directory. Note that if a list contains item `foo.cpp`, and you subtract `*.cpp` from the list, the `foo.cpp` will be removed from the list only if file with that name actually exists. Also note that currently the `-=` syntax is supported only when defining the *sources* list of a project.

The useful global lists you can assign to are:

- *includes* - list of directories to search when including a file from C++ sources.
- *defines* - list of macros to always define when compiling a C++ source file.
- *precompiled* - name of C++ header to precompile. This list should contain only single item.

Each `makefile.gen` should contain definition of at least one project. To define a program project called *`foo`*, use the following syntax:

```
program foo
  ...
```

```
endprogram
```

The project name must be unique.

Three project types are recognized:

- A program, defined in `program` - `endprogram` block. This compiles and links an exectuable with the specified name.
- A statically linked library, defined in `library` - `endlibrary` block.
- A shared library, aka dynamically linked library, defined in `shared_library` - `endshared_library` block. When compiling an object that belongs to a shared library *`foo`*, macro `OPTION_BUILD_SHARED_`*`foo`* will be defined.

Project options are defined by list assignments inside the project block. The following lists exist:

- *sources* - list of C++ sources the project consists of. When a project contains an `idl.list`, you can use assignment `sources += @GENERATED_SOURCE@` to add list of all sources generated from the IDL files to the list of sources to compile.

- *libs* - list of libraries to link with, if the project is a program, or list of libraries to build before current project, if the project is a library. A library should be specified using a relative path to the directory where the `makefile.gen` for the library is located, concatenated with the library name.

- *shlibs* - list of shared libraries to link with, if the project is a program. If the project is a library, this list should contain all shared libraries that this library *might* reference, ie. require some symbols defined in some of the referenced shared libraries. All required libraries must be specified correctly, otherwise link of a program that used this library may fail on some platforms.

  When compiling an object belonging to the project, macro `OPTION_USE_SHARED_`*`foo`* will defined for each referenced shared library *`foo`*

- *syslibs* - list of system libraries to link with. The list should contain a symbolic name of the library - on Win32 without the `.lib` extension and on Unices without the `lib` prefix and the `.a` (or `.so`) suffix. Several macros are predefined to link program with some useful libraries:

  - *STDCPP_LIBS* - standard C++ libraries
  - *SDL_LIBS* - Simple DirectMedia Layer libraries
  - *OPENGL_LIBS* - OpenGL and GLU libraries

Note

It's not recommended to define multiple projects in single file, unless you really know what you are doing.

Examples

The best examples are all `makefile.gen` files that are part of the Massiv Core and the Massiv Demo sources.

Notes

The **mkgen.pl** can be found in the `mkgen` subdirectory of the massiv source root directory.

Name

genmkgen.pl -- generate `makefile_idl.gen`

genmkgen.pl

Synopsis

`mkgen.pl` [OPTION...] [directory]

Description

Generate `makefile_idl.gen` files from `idl.list` files found in specified directory and its subdirectories. If no directory is specified, current directory and all its subdirectories are processed.

Options

The only option is:

`-v | --verbose`
: Be more verbose. May be used multiple times.

Notes

Syntax of `idl.list` files is described in Section D.2, “The idl.list File”.

The **genmkgen.pl** can be found in the `src/core/factgen` subdirectory of the massiv source root directory. Check its perldoc documentation for more detailed info.

Name

factgen.pl -- generate metaclasses and rpc stubs from IDL

factgen.pl

Synopsis

`factgen.pl` [OPTION...] {file.idl}

Description

Generate `file_generated.h`, `file_generated.cpp` and `file_rpc.h` from given `file.idl`.

This is the IDL processing tool. When using standard Massiv build tools (**genmkgen.pl** and **mkgen.pl**), you won't ever have to use this tool directly.

Options

The options is:

`-I | --include` *`directroy`*
: Add the *`directory`* to the list of directories to search when including or importing an IDL file.

`-v | --verbose`
: Be more verbose. May be used multiple times.

Notes

The **factgen.pl** can be found in the `src/core/factgen` subdirectory of the massiv source root directory. Check its perldoc documentation for more detailed info.

Name

classlist.pl -- generate class list files from `idl.list` files

classlist.pl

Synopsis

`classlist.pl` [OPTION...] [directory]

Description

Generate `class_list.list`, `class_list.h` and `class_list.cpp` files from `idl.list` in specified directory.

Options

The only option is:

`-v | --verbose`
    Be more verbose. May be used multiple times.

Notes

What the class lists are and syntax of `idl.list` files is described in the next section.

The **classlist.pl** can be found in the `src/core/factgen` subdirectory of the massiv source root directory. Check its perldoc documentation for more detailed info.

D.2. The `idl.list` File

The `idl.list` is used by two utilities:

- It's read by the **genmkgen.pl** tool. The tools generates `makefile_idl.gen` file from each `idl.list`. These files are used to generate makefiles that drive compilation and linking of Massiv programs and libraries. The **genmkgen.pl** uses contents of `idl.list` to determine which IDL files exist, which files should be generated from the IDL files, how to generate the sources and dependencies of the generated files.
- The **classlist.pl** tool read the file to determine set of all managed classes that exist and to write their list and generate sources that instantiate and register metaobjects of all the classes at program startup.

The programmer has to write an `idl.list` in each directory that contains a project that either defines or references a managed class.

The format of `idl.list` files is line-oriented. Empty lines and lines beginning with a hash character (`#`) are skipped. Other lines may contain the following commands:

- `depends` *`dirname`*

  Specifies a directory containing another `idl.list` file that contains IDL files required by projects in this directory. An IDL file is required if a C++ source includes a source generated from the IDL file.

  This command will also add the *`dirname`* to the list of directories to search when importing or including an IDL file. Moreover, it's used by the `class_list` command as described later.

- `directory` *`dirname`*

  This command specifies the directory which the IDL processor should search when an IDL file includes or imports another IDL file. All the directories specified by the `depends` commands will be searched too.

- `idl` *`filename`*

  This line specifies one IDL file that should be processed. All existing IDL files must be listed in exactly one `idl.list`.

- `class_list`

  The Core needs to know list of all managed classes that are defined in a node executable to correctly instantiate their metaobjects and factories on node startup. This list must be generated in exactly one library (or program), that is linked to the resulting executable. To build the list, use the `class_list` command in the `idl.list` of the library (or program). The `idl.list` that contains the `class_list` command must specify all libraries with managed classes that will be linked to the exectuable as its `depends`.

  Makefiles generated for the library (or program) project with the `class_list` command will generate `class_list.cpp` and `class_list.h` sources from the `idl.list`, using the **classlist.pl** tool. The sources will define single function, `Massiv::Generated::register_classes()`, which registers all managed classes to the Massiv Core. Pointer to this function is used at node startup, as described in Section 15.2, “Initializing the Core”.

> **Note**
>
> Each of the commands except `class_list` may be used multiple times.

D.3. Documentation Tools

This section briefly describes how to build Massiv documentation.

D.3.1. Doxygen

All sources of the Massiv Core, Massiv Demo, utilities and examples are documented using Doxygen comments. Doxygen (http://www.doxygen.org) is a documentation system that can generate nice documentation from a set of documented sources. The Massiv Core Reference Guide mentioned many times in this book is generated by the Doxygen too.

To generate documentation of the Massiv Core (the Massiv Core Reference Guide) or documentation of the Massiv Demo, follow these steps:

1. Install the Doxygen.
2. Optionally install Graphviz (http://www.research.att.com/sw/tools/graphviz/) if you want to generate nicer diagrams.
3. Enter the `doc/doxygen` subdirectory of the Massiv source root directory.
4. Copy `config.doxygen.example` to `config.doxygen` and edit it. If you have install Graphviz, you should set `HAVE_DOT` to `YES`.
5. On systems with unixish **make**, you can use supplied Makefile to build the documentation. Otherwise run Doxygen directly; to build the Massiv Core Reference Guide, use `core.doxygen` config file, to build the Massiv Demo documentation, use `demo.doxygen`. You may have to create the output directory (`html/core` for example) manually, because Doxygen can't create nested directories automatically.

D.3.2. DocBook

The documentation you are reading now is written in DocBook (http://www.docbook.org). Its XML sources can be found in `doc/docbook` subdirectory of the massiv source root directory.

Building DocBook documentation is not an easy task. Please refer to `doc/tech/documentation.txt` for information about required tools and documentation of the `doc/docbook/docgen.pl` utility that is used generate the documentation.

Appendix E. Compiling Massiv

E.1. Compilation of the Core

E.1.1. Downloading Massiv from CVS

If you are reading this file, you probably already have the sources. Nevertheless, you can checkout them from SourceForge CVS using the following commands:

```
cvs -d:pserver:anonymous@cvs.massiv.sourceforge.net:/cvsroot/massiv login
cvs -z3 -d:pserver:anonymous@cvs.massiv.sourceforge.net:/cvsroot/massiv co massiv
cvs -z3 -d:pserver:anonymous@cvs.massiv.sourceforge.net:/cvsroot/massiv co demo_data
cvs -z3 -d:pserver:anonymous@cvs.massiv.sourceforge.net:/cvsroot/massiv co demo_install
```

The first line logs you as an anonymous user to the CVS - hit enter when asked for a password. The second line checkouts the source codes. You need to use the last two lines only if you want to run the Demo. The third one enables to download simulation data (textures, models, etc.) and finally the last line checkouts some stuff that could help you create installers for Windows.

E.1.2. Compilation Steps Common for All Platforms

1. *Download Cryptopp*

   Download Cryptopp (a cryptography library) from http://www.eskimo.com/~weidai/crypto42.zip. Unpack it into `src/ext/cryptopp` (under Linux use the `-a` option to covert CR/LF newlines properly). Don't compile Cryptopp separately, i.e. using the provided `dsw` project.

2. *Download the SDL libraries*

   Download and install the following libraries:

   - SDL (http://www.libsdl.org) - a portable library for graphics, sounds, etc.
   - SDL_image (http://www.libsdl.org/projects/SDL_image) - a portable library for loading/saving images from/to files.

3. Download OpenGL and the GLU library

   Probably the OpenGL is already installed on your system. If not, refer to http://www.opengl.org for information how to install it. Note that besides the OpenGL you need also the GLU library. Even this one is probably already preinstalled; seek for more information on the same website.

E.1.3. Compilation under Windows (MSVC)

Note

Requirements: MSVC 6.x or 7.x with a command-line compilation support installed.

You have to accomplish the following steps to compile the Massiv Core library from source code under Windows:

1. *Download Perl*

   A working possibility is ActivePerl (http://www.activestate.com) or the Perl of Cygwin (http://www.cygwin.com/setup.exe) for example. ActivePerl seems to be the more effective choice bringing less problems. Note: In case you decide installing Cygwin, you should choose "Default text file type: unix" during the installation.

2. *Download STLPort*

   Download STLPort (a STD C++ library implementation) from http://www.stlport.org/download.html. Massiv relies on an ANSI-conformant STD C++ library. You are required to use STLPort under MSVC 6.x. If you are the lucky one and own MSVC 7.x, you can use the Dinkumware STD C++ library which is shipped with it.

3. *Compile STLPort if you want to use it.*

   If you chose to use the STLPort library, compile it with the `iostream` support according to instructions included within the distribution:

   You should go to the `src` directory of the STLport and run
   `nmake` /f vc6.mak clean all
   command if using MSVC 6.x or
   `nmake` /f vc7.mak clean all
   if using MSVC 7.0.

4. *Generate makefiles*

   Run
   `configure.cmd`
   from the Massiv root directory to generate makefiles and some other stuff. Re-run each time you add or remove sources, change #includes, etc.

5. *Configure compilation*

   Copy `mkgen/config/msvc.config.example` to `mkgen/config/msvc.config` and edit it. Especially you need to edit `msvc.config` to choose between the `Debug/Release` build and to set paths to your STL and SDL libraries. All you need to do should be obvious when you see the file contents.

   Note

You don't have to regenerate makefiles when `msvc.config` is changed.

6. *Compile auxiliary utilities*

   Compile `makedir.c`, `runin.c` and `touch.c` in `mkgen/win32` and move it to somewhere in your `PATH`.

Note

This compilation is as simple as the following command:
`cl {utility_name.c}`

7. *Compile Massiv*

   Run
   `nmake` /f makefile.msvc
   from `src/core` to compile the Core library.

Note

You should have no troubles compiling the Demo executables now. The Demo is located in `src/demo` directory. In each `server`, `client` and `service` subdirectory there is a `config.h.example` file. Copy it to `config.h` and edit it. You have to set the path to the main configuration file appropriately. After that, just run
`nmake` /f makefile.msvc
from each of the `server`, `client` and `service` subdirectory.

E.1.4. Compilation under Linux (GCC)

Note

These are an instructions to compile not only the Massiv Core, but also how to make the Demo running. These instruction are already available in a simple text file `massiv/doc/INSTALL` in the source tree. It may contain more up-to-date information.

Note

Requirements:

- gcc 3.0, 3.2 or better with the corresponding version of libstdc++.
- SDL 1.2+ headers and libraries, including SDL_image library.
- OpenGL 1.1+ and GLU headers and libraries.
- Perl 5.0 or newer.

You have to accomplish the following steps to compile the Massiv Core library from source code under Linux/GCC:

1. *Configure*

   To configure Massiv for your development environment, run
   `./configure`
   script in the `massiv` directory (the one containing `doc` and `src` subdirectories). If it fails, try running
   `./configure` --help
   to find out how to fix the problem. You also might examine the `configure.log`.

2. *Compile and install*

   To compile the Massiv Core library, demo game client, server and data service and various utilities, run **make** in the `massiv` directory which contains the configure script from mentioned in the first step.

   Note

   The Demo is located in the `src/demo` directory. In each `server`, `client` and `service` subdirectory there is a `config.h.example` file. *Before* you start the Demo compilation, you have to copy it to `config.h` and edit it. You must set the path to the main configuration file appropriately.

   To install the Core library, demo game and utility binaries, demo game data, setup script and basic documentation into directories specified in the Configure step, run
   `make` install
   in the same directory. If you only want to install the demo game client, run
   `make` install_client

3. *Setup demo game servers*

   To setup a demo game server, run the **massiv-setup** script from an empty directory. If you run it from empty `~/.massiv` with no parameters, the default setup (three simulation servers, master data service and client with default privileged account, all on localhost) will be used. You can then run the data service, simulation servers and the client using the run-`*` scripts generated in the `~/.massiv` directory. If you want a different setup (run servers on multiple machines, use different number of simulation servers, store runtime data in different directories, etc.), run
   `massiv-setup` --config=*`FILENAME`*

   Contents of the setup configuration (*`FILENAME`*) is documented in the default config which is used when no `--config` option is used. Run
   `massiv-setup` --help
   to determine location of the file. It is possible to configure for example the following:

   - How many simulation servers will be used
   - IP addresses of the master data service and simulation servers
   - Paths where run-time data (i.e. contents of subdirectories created by massiv-setup in current

directory) will be stored on server machines.

4. *Running the Demo Game Servers*

   Distribute contents of directories corresponding to run-* scripts generated in the previous step to all the server machines (their addresses and locations of the run-time files have been and configured before). Use run-* scripts to run the servers. The master data service must be running before simulation servers can be launched.

5. *Running Demo Client With Root Account*

   Configuration of the client has already been generated. Copy contents of the `client` directory to the directory specified by `massiv-setup` config. You can use this config on any machine as long as it's stored in a correct directory. Use **run-client** to run the client. Use `default` as login name and no password.

   Warning

   This is pretty insecure, make sure that no one can read contents of the `client/login/default` file). Or you can reencrypt this file using **massiv-decrypt** (with no password) and **massiv-encrypt** (with new password).

6. *Provide the Client Installation Package*

   To run the demo, the massiv-client binary and libmassiv.so library is required. To create the initial run-time environment for the client, copy contents of the `client` directory created in step 3 without `login/default file`, anywhere, and modify `config/paths.conf`. Distribute contents of this directory to clients.

   Users can then run Massiv client from the same directory on any computer, or optionally edit `config/paths.conf` if they want to store config files and run-time data in other directories. As the first step, they must create a new account. Any client can create accounts from the login dialog in Massiv Demo. However, unlike the "default" root account, new accounts created from the login dialog can't be used to edit the world terrain and entities, unless their privileges are modified by user with root account. If client wants to use her account on other computer, she can simply copy `login/LOGINNAME` to corresponding directory on the new computer.

E.2. Compilation of Documentation

You probably won't have to recompile documentation as it is already provided within the distribution. However, if you want to make some change to the documentation, please refer to `massiv/doc/tech/documentation.txt` for information about how to compile the user documentation from the DocBook sources and to `massiv/doc/tech/documentation_doxy.txt` for information how to generate programmer's documentation from the C++ source code.

Appendix F. Example Listings

This chapter contains listings of all example sources stored in the `src/example` directory.

F.1. Creating Managed Class

Listings of examples described in Chapter 13, *Creating Managed Class*. Their sources are in `src/example/server_lib`.

Example F.1. `makefile.gen`

```
# "Creating Managed Class" example.
# mkgen.pl will generate makefiles for selected platforms from this file.

# Required for all C++ projects.

STDCPP

# Include makefile.gen part generated by genmkgen.pl from idl.list.

include makefile_idl.gen

# Directories to include from.

includes = . ../../core

# Precompile the massive core.h header if the compiler supports it.

precompile = core.h

# Create a library in this directory, containing objects compiled from
# all C++ sources, including the generated sources. This library will
# reference symbols from the Massiv Core shared library.

library example_lib_server
    sources = *.cpp
    sources += @GENERATED_SOURCES@
    shlibs = ../../core/massiv
endlibrary
```

Example F.2. `idl.list`

```
# "Creating Managed Class" example.
# Contents of this file are read by genmkgen.pl which
# generates parts of makefile.gen.

# We use managed classes from the Core library.

depends ../../core

# List of files containing descriptions of managed classes.

idl hello_interface.idl
idl hello.idl

# This not the main managed class library,
# so do not generate a class list here.
#class_list SHARED SERVER
```

Example F.3. `hello_interface.idl`

```
#import "core.idl"

namespace example {

class
    <
    abstract,
    kind = SERVER,
    root
    >
    HelloInterface : ::Massiv::Core::Object
    {
    method< virtual > hello
        (
        in string callee_name
        ) : string;
    }

} // namespace example
```

Example F.4. `hello_interface.h`

```
#ifndef EXAMPLE_HELLO_INTERFACE_H
#define EXAMPLE_HELLO_INTERFACE_H

#ifndef MASSIV_CORE_H
#include "core.h"
#endif

#include <string>

namespace example {
/**
```

```
  * Interface of hello classes.
  *
  * This is generic interface of all classes that implement the
  * hello() method.
  *
  * As described in the IDL, all Hello classes should be root objects
  * of KIND_SERVER kind.
  */
class HelloInterface : public ::Massiv::Core::Object
    {
public:

    /**
     * The "say hello" method.
     *
     * It should generate a hello message for the callee @a callee_name,
     * print it anywhere it wants and return it.
     */
    virtual std::string hello
        (
        const std::string & callee_name
        ) = 0;

    }; // class HelloInterface

} // namespace example

#endif // EXAMPLE_HELLO_INTERFACE_H
```

Example F.5. `hello.idl`

```
#import "hello_interface.idl"

namespace example {

class
    <
    simulation_startup_notify
    >
    Hello : HelloInterface
    {
    method register_to_naming
        (
        in string name
        ) : bool;

    property string   name;
    property vlint32  total_call_count;
    property vlint32  current_call_count;
    }

} // namespace example
```

Example F.6. `hello.h`

```
#ifndef EXAMPLE_HELLO_H
#define EXAMPLE_HELLO_H

#ifndef MASSIV_CORE_H
#include "core.h"
#endif

#ifndef EXAMPLE_HELLO_INTERFACE_H
#include "hello_interface.h"
#endif

namespace example {

/**
 * Simple implementation of the HelloInterface.
 */
class Hello : public HelloInterface
    {
public:

    /**
     * Object initialization.
     *
     * The internal name of the object (not to be confused with
     * name under which it's registered to the garbarge collector)
     * will be set to @a the_name.
     */
    void initialize
        (
        const std::string & the_name
        );

    /**
     * The "say hello" method.
     *
     * The message generated by this implementation will contain
     * the callee name, name of the Hello object, and counters
     * indicating how many times this method has been called since
     * creation of this object and since last simulation startup.
     */
    virtual std::string hello
        (
        const std::string & callee_name
        );

    /**
     * Register the object to global naming service.
     *
     * This method will change name of the object to @a the_name,
     * and try to register the object to the global naming service
     * under that name. If the registration fails it will destroy
     * the object.
     *
     * @note This method will fail unless the example library is
     *       linked with the Massiv Demo. However, because it uses
     *       dynamic RPC, it can be compiled without the library.
     */
    bool register_to_naming
        (
        const std::string & the_name
        );

protected:

    /**
     * Object update callback.
     */
    virtual void object_updated
        (
        UpdateReason reason
        );

public:

    Massiv::Core::PString   name;
        /**< Internal name of the object.
             It's used to print the hello message and to register
             the object to the global naming service. */

    Massiv::Core::PVlInt32  total_call_count;
        /**< Number of calls to hello() since object creation. */

    Massiv::Core::PVlInt32  current_call_count;
        /**< Number of calls to hello() since node startup. */

    }; // class Hello

} // namespace example

#endif // EXAMPLE_HELLO_H
```

Example F.7. `hello.cpp`

```
#include "core.h"
#include "hello.h"
#include "database/well_known_object_id_database.h"
#include <sstream>

using namespace Massiv::Core;

namespace example {

void Hello::initialize
    (
    const std::string & the_name
    )
    {
```

```
    name = the_name;
    }

std::string Hello::hello
    (
    const std::string & callee_name
    )
    {
    std::ostringstream oss;
    oss << "Hello to " << callee_name << " from " << name << std::endl;
    oss << "(called " << total_call_count << " times, " <<
           current_call_count << " since startup)" << std::endl;
    const std::string s = oss.str();

    total_call_count++;
    current_call_count++;

    Global::log_info( Status::FACILITY_LIB, Status::PRIORITY_LOW, s );
    return s;
    }

bool Hello::register_to_naming
    (
    const std::string & the_name
    )
    {
    initialize( the_name );

    const WeakPointer< Object > self( this );
    bool failed = true;
    try
        {
        const ObjectId naming_id = Global::well_known_object_id_database().
            get_naming_service_object();
        const Remote< Object > naming( naming_id );

        const MetaObject * const metaobject = Global::class_manager().
            get_metaobject( "Demo::Lib::NamingService" );

        std::stringstream ss;
        {
        TextWriter tw( ss );
        const Serializer::Description desc;
        name.text_write( tw, desc );
        tw.write_space();
        self.text_write( tw, desc );
        tw.write_space();
        const SBool replace = false;
        replace.text_write( tw, desc );
        }

        TextReader tr( ss );
        std::auto_ptr< MethodPacket > results = metaobject->
            remote_call_method( naming, "register_object", tr );

        if( results->get_argument_value( -1 ) == "true" )
            {
            failed = false;
            }
        else
            {
            std::ostringstream oss;
            oss << "Object " << name << " already registered "
                   " to the global naming service." << std::endl;
            Global::log_warning( Status::FACILITY_LIB,
                Status::PRIORITY_HIGH, oss.str() );
            }
        }
    catch( std::exception & e )
        {
        std::ostringstream oss;
        oss << "Failed to register " << name << " to naming service: "
            << e.what();
        Global::log_warning( Status::FACILITY_LIB,
            Status::PRIORITY_HIGH, oss.str() );
        }

    if( failed )
        {
        Massiv::System::dispose_gc_root( self );
        }
    return failed ? false : true;
    }

void Hello::object_updated
    (
    UpdateReason reason
    )
    {
    if( reason == SIMULATION_STARTUP )
        {
        current_call_count = 0;
        }
    }

} // namespace example
```