Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Reproducible Evaluation of a Virtual Node System
D3.2.9

Due date of deliverable: November 30th, 2008
Actual submission date: December 4th, 2008

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2.4

Responsible institution: ULM
Editor & and editor's address: Jörg Domaschka
Abt. Verteilte Systeme
Universiät Ulm
James-Franck-Ring O-27
89069 Ulm
Germany

Version 1.0 / Last edited by Jörg Domaschka / December 4th, 2008

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	01/11/2008	Jörg Domaschka	ULM	Structure
0.1	02/11/2008	Jörg Domaschka	ULM	Introduction
0.2	03/11/2008	Jörg Domaschka	ULM	RMI
0.3	03/11/2008	Jörg Domaschka	ULM	Configuration,Middleware Layers
0.4	04/11/2008	Jörg Domaschka	ULM	Middleware Adapter,Join Protocol
0.5	05/11/2008	Jörg Domaschka	ULM	Example
0.6	05/11/2008	Jörg Domaschka	ULM	Group Communication,XOS
0.7	25/11/2008	Jörg Domaschka	ULM	Update, reviewer comments
0.8	29/11/2008	Jörg Domaschka	ULM	Update, reviewer comments
0.9	30/11/2008	Jörg Domaschka	ULM	Update, reviewer comments
0.10	04/12/2008	Jörg Domaschka	ULM	spell checking

Reviewers:
Ales Cernivec (XLAB) and Arenas Alvaro(STFC)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
3.2.4	Design and implementation of a virtual node system	ULM*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

This document is the sequel of Deliverable 3.2.5 and discusses the evaluation of the Virtual Nodes framework since M18. During that phase we mainly focused on the usability of the infrastructure so that a large part of this document deals with the user view on Virtual Nodes. There are three user roles: users who start a system using Virtual Nodes such as administrators. They have to know, how to configure the framework. Then, there are developers that use the features of the framework to build a fault-tolerant application and need to know how to use the API of the Virtual Node middleware layer. And finally, there are middleware developers that want to integrate Virtual Nodes in their middleware system and need to know the low-level features of the framework. Apart from the user view we present the current state of a performance evaluation and discuss shortcommings in the group communication systems we currently use.

Contents

1 Introduction

The purpose of Virtual Nodes is to help the programmers to replicate services for both performance and fault-tolerance reasons. The main target we are aiming at is to minimise the effort by (a) maximising the reusability of existing replication-unaware code and by (b) making replication issues as far as possible transparent to the service developer. At the same time, it shall be easy to make use of Virtual Nodes for the programmer of an application and for a developer of a replicated distributed service. For that reason the Virtual Node software written in Java is provided as a library developers can link to their programmes.

As we aim for supporting different kinds of applications, we provide multiple replication strategies. Furthermore, we allow the composition of the replica group to change at runtime. The reasons for that are on one hand to ensure that long running services do never lose their fault-tolerance guarantees which would be reduced in case of node failures. On the other hand, when replication is used for performance reasons, it might happen that the current number of nodes is not sufficient to answer all requests. In both cases, new replicas have to be integrated in the current group of replicas.

As decribed in Deliverable 3.2.5 [3] and shown in Figure 1 the Virtual Node infrastructure is composed of four parts. A middleware layer and the replication layer, which come with a client-side and a server-side part each. The functionality of the replication layer is covered by D3.2.5. In this document we mainly focus on the user view that requires a discussion of the middleware layer. Regarding the replication layer, we only present changes happened since the last deliverable.

In order to increase the ease of use for the user, we first have to define who can be a user of our framework. We were able to identify three user roles: users who simply start a system using Virtual Nodes such as administrators. They have to know, how to configure the framework. Then, there are developers that use the features of the framework to build a fault-tolerant application and need to know how to use the API of the Virtual Node middleware layer. And finally, there are middleware developers that want to integrate Virtual Nodes in their middleware system and need to know the low-level features of the framework.

The remainder of this document is structured as follows. In Sections 2 to 4 we discuss the different user views: configuration of the system, the framework's low-level API, and how to build applications using the Virtual Nodes RMI middleware layer. Section 5 sketches a join protocol for multithreaded replicas, followed by a discussion of the group communication systems of Virtual Nodes in Section 6. In Section 7 we discuss the current state of a performance evaluation. This paper concludes with a discussion on how Virtual Nodes can be used in XtreemOS. All sections contain a discussion of future work and open issues, if any.

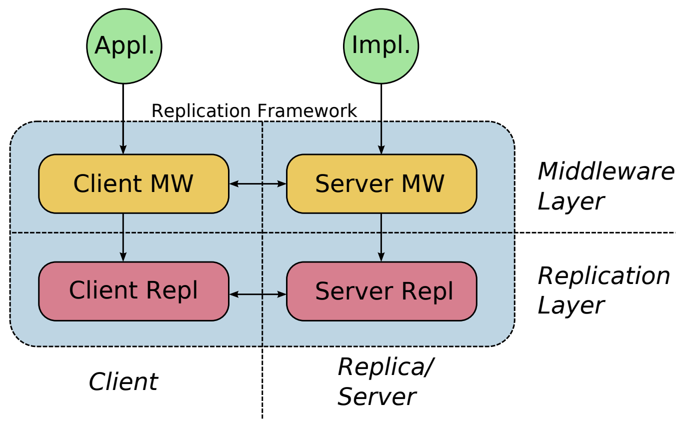


Figure 1: Virtual Node Architecture

2 Configuration

Virtual Nodes is a highly configurable replication infrastructure. The replication core provides a variety of configuration options for group communication, request scheduling, client-replica communication, and replication protocol. Apart from the fact that the system has to have a way to be configured at all, we want it to be easily configurable.

Basically, there are three ways to configure a system: at compile-time, at start-up time, and at run-time. We have extended the compile-time configurability from Deliverable 3.2.5 to a start-up time configurability. We did that by providing a proxy class for each configurable element that encapsulates the respective configurable entity. Thus, for the schedulers there is a `SchedulerProxy`, for the replication protocol there is a `ReplicationProxy`, etc. All of those proxy classes take a `Configuration` in their constructors that contains the system configuration. `Configuration` maps each of the configurable entities to a `ConfigurationOption`. As not all entities have the same configuration options, `ConfigurationOption` is an interface that has to be implemented by a class specific for the entity. Thus, there is an `SchedulerConfigurationOption` for schedulers, a `ReplicationConfigurationOption` for the replication protocol and so on. The base classes are shown in Figure 2.

A `Configuration` is created by the constructor shown in Figure 2. If it is required to read the configuration from a file or string, this has to be realised

```
public enum ConfigurableEntity { G_COM, PROT, SCHED, C_COM}

public interface ConfigurationOption { }

public final class Configuration {

 public Configuration(Map<ConfigurableEntity, ConfigurationOption>) {
  /* ... */
 }

 public final ConfigurationOption configForEntity(ConfigurableEntity){
  /* ... */
 }
}
```

Figure 2: Configuration class

external to the `Configuration` class. We chose this approach, as we want the configuration to be data format independent. If (de)serialisation is required, configuration parsers can be implemented. Virtual Nodes come with a parser that creates a `Configuration` from a string object, as we use a stringified representation to send the configuration over wire.

In some cases the options provided by the approach sketched so far, are not sufficient to handle the variety of configurations. This is especially true, if there are different configurations not only depending on the `ConfigurableEntity` such as scheduler or replication protocol, but also within a common entity. For instance this is true for the group communication system (GCS). As Virtual Nodes do not come with their own GCS, they have to use existing ones, that use different approaches regarding their configuration. For that reason we allow `ConfigurationOptions` to contain so-called `ConfigurationAddOns`, information that is specific for a certain instance of a configuration.

The current approach as presented here, only works as long as the list of configuration options is static and does not change. As soon as users add arbitrary new classes to replace or extend existing implementations or add new configurable entities that are not contained in `ConfigurableEntity`, an architecture with a sole configuration class might not scale anymore. Thus, we consider finding a more powerful configuration mechanism as future work. The same holds for making the system re-configurable, that is configurable at run-time.

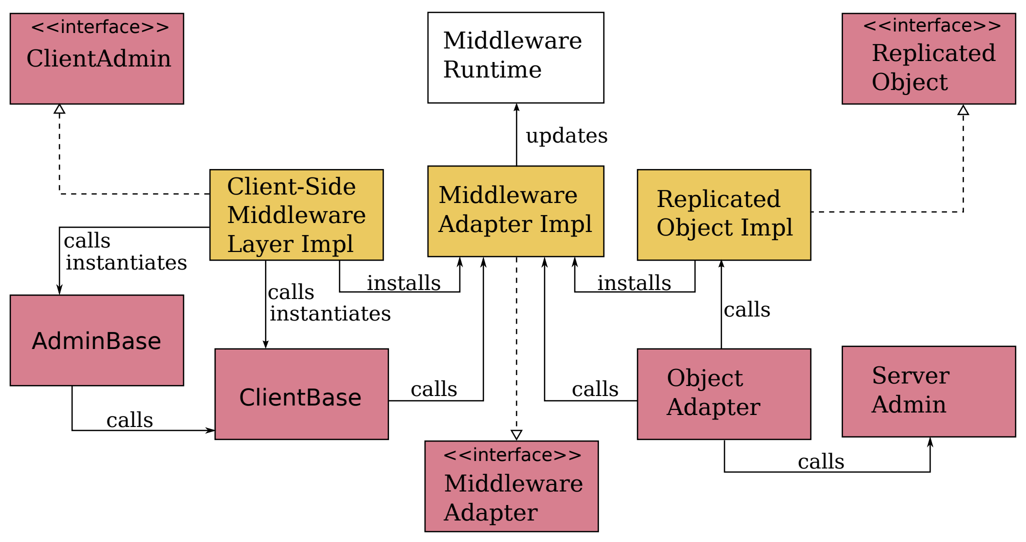


Figure 3: Middleware-Core Interaction

3 Providing Middleware Layers

In its basic version the interaction of middleware layer and replication layer for method invocation is straight forward: the middleware layer passes a method and its parameters to the replication layer. This, in turn, transmits it to the server-side replication layer that, in turn, invokes the server-side middleware layer in a way consistent with the replication protocol. In this system architecture the middleware layer defines how parameters are marshalled, i.e. which external data representation shall be used, how methods are identified, e.g. based on names in CORBA and signatures in Java RMI. Furthermore, the middleware layer also defines how stubs look like, how they they are initialised and passed to other machines, as well as how binding happens. This requires a callback mechanism from the replication layer to the middleware layer. In the next section we will explain why this is required and how this requirement can be satisfied in a generic way.

Figure 3 sketches the classes and interfaces of the core framework and a generic middleware layer as well as their interaction. The following sections discuss them in detail; refer to this figure for the global architecture.

3.1 Adapter Concept

The fact that the middleware layer decides how a stub is serialised and passed to other nodes raises an interesting challenge. Basically, there are two different ways how a middleware layer might serialise its stubs. The first one, used by Java RMI, is to serialise entire objects and to deserialise them at the receiver-side. The other one used by systems such as CORBA and ICE uses an external object representation (*IOR* in the case of CORBA). Whereas the first approach serialises the stub on demand, the second may not. A CORBA ORB for instance stores the IOR and maps it to the stub. When the stub is passed on to another machine it uses the mapped IOR. This works fine, as long as the information contained in this data structure (IOR) does not change.

In the case of Virtual Nodes, however, this is not true any more due to replica groups changing over time. The middleware layer is in charge of initialising the replication layer, because the middleware layer is all a client application can interact with. As the replication layer has to be able to communicate with the replicas of the Virtual Node it is belonging to, there is the need to initialise it with contact information. As the group of replicas is likely to change, it is advantageous to use the latest information available. When a stub is serialised on demand this is not a big issue, as the stub can pull the required information from the replication layer. In case the middleware layer internally stores an external representation of the stub new group information has to be pushed in the middleware system.

This gets us in a dilemma. The middleware layer knows about middleware internals. The replication layer knows about changes of the replica group, but it neither knows about middleware internals nor about the format the information is represented within the middleware runtime: Thus, it cannot call a method at the middleware layer and even if it could, this would destroy the genericity of our approach, as replication and middleware layer would become circularly dependent.

The middleware layer is responsible for initialising the replication layer. During this process we allow to pass a `MiddlewareAdapter` to the replication layer. The replication layer will invoke the `newMemberList` method (c.f. Figure 4) at the `MiddlwareAdapter` everytime the replica group changes. The adapter has the task of transforming the information of the `MemberList` to a middleware internal data representation and of injecting this information into the middleware runtime. Note that the adapter is required on both client and server-side, as also the server might have to pass a stub/reference to the client. This is the case when a method invocation returns `this`. Instead of serialising the object, a stub is created and returned. An extensive discussion of the adapter concept and its realisation can be found in [5].

```
public interface MiddlewareAdapter {
        public void newMemberList(MemberList ml);
}
```

Figure 4: `MiddlewareAdapter` Interface

```
public class ClientBase {
        public ClientBase(ObjectId, MiddlewareAdapter, MemberList);
        public SingleReply sendCall(MethodId methodID, byte[] parameters);
        public byte[] serialise();
        public static ClientBase deserialise(byte[],MiddlewareAdapter);
}
```

Figure 5: `ClientBase` Interface

3.2 Client-Side

The interface the replication layer provides to the middleware layer at client side consists of mainly two classes: `ClientBase`[1] and `AdminBase`. The latter is a Virtual Nodes-internal middleware layer for special methods required for administrating the Virtual Node at run-time. We discuss administration methods in Section 3.4

As shown in Figure 5 `ClientBase` is instantiated with an `ObjectId`[2], a `MiddlewarAdapter`, and a `MemberList`. `ObjetId` uniquely identifies the Virtual Node this `ClientBase` instance is bound to. The value is created by the first replica and has to be passed to the `ClientBase` during startup process. `MemberList` is a set of replica addresses that `ClientBase` can connect to for sending requests. This list is not static and can change over time by update messages from the replica group [3].

The only method of `ClientBase` used for sending requests is `sendCall`. It takes a `MethodId` and `byte[]` as arguments. `MethodId`[3] is representation of the methods that can be invoked at the Virtual Node. Middleware systems using the core framework can extend this class in order to use their own way of repre-

[1]all classes mentioned in this section are in package `eu.xtreemos.vnode.client` unless another package is specified.

[2]in `eu.xtreemos.vnode.common`

[3]in `eu.xtreemos.vnode.common`

```
public interface ReplicatedObject {
    public boolean isReadOnly(MethodId) throws UnknownMethodException;
    public Reply dispatch(MethodId, byte[]) throws UnknownMethodException;
}

public ViewContainer(MiddlewareAdapter){
        /* */
}
```

Figure 6: Server-side middleware interaction

senting methods. The `byte[]` parameter represents the marshalled arguments to the method call. The content of both parameters is treated in a purely opaque way by the replication layer, which does not try to interpret its content.

`ClientBase` is not serialisable. Instead it has to be brought up manually by each clients willing to use it. The reason behind that is similar to the arguments for the introduction of the `MiddlewareAdapter`. We do not know what information the client stub can handle for transmission and in which format this information has to be. However, any stub should be able to transmit a sequence of bytes, so that we offer two methods for serialising the relevant information of `ClientBase` to a byte array and initialising it from a byte array.

3.3 Server-Side

At server-side the interactions with the middleware layer are inverse to those at client-side (c.f. Figure 6). The object that the replication layer sees has to implement the interface `ReplicatedObject`[4]. The replication logic will hand the opaque information, that is parameters and method identifier to this object where the message is finally processed and a `Reply` is returned. Furthermore, in order to determine whether the execution of a request can be optimised the replication logic has to know if a certain method is read-only, i.e. does not change the replica state. For that purpose a `ReplicatedObject` has an additional method that provides this information. If the middleware layer does not support this feature, it is save to always return `false` here. All methods that deal with `MethodIds` thrown an `UnknownMethodException` in case the middleware layer cannot handle this `MethodId` for whatever reason. This exception will result in an error returned to the client.

[4]in `eu.xtreemos.vnode.server`

```
public interface ClientAdmin {
    public ReplicaId startNewReplica(MiddlewareAdapter adap);
    public void shutdownReplica(ReplicaId id);
    public void shutdownService();
}

public class ServerAdmin {
        public Configuration getInstantiationCredentials();
        public void shutdownReplica(ReplicaId id);
        public void shutdownService();
}
```

Figure 7: Interfaces for Administration

Just as the client-side, the server-side also requires a `MiddlewareAdapter`. This is handed over at replication layer initialisation and will finally be used in `ViewContainer` that handles the current view the replica has on the replica group. Each time this view changes the `MiddlewareAdapter` is invoked.

3.4 Administration Methods

Administration methods provide a means to configure and influence the system at runtime. So far, only methods to add and remove replicas have been implemented. Yet, it can be imagined to add methods for other purposes such as security, reconfiguration, etc.

The admin methods are different on client and server-side. The main reason for this is the goal to not burden the programmer with additional complexity. At client-side the interface provides methods to start a new replica at the host this method is called, to shutdown individual replicas and to shutdown the entire service. The latter two methods are also available on server-side. However, starting a new replica cannot happen at server-side. Thus, this method is not present in the server interface. Instead, there is a method that provides a way for the client to get the current configuration from the server, which is a prerequesite for starting the new replica.

All those methods are implemented in the replication core and must not be overridden by the middleware layer. Nevertheless, it is necessary that the middleware layer offers the methods of `ClientAdmin` to the applications using it. In consequence the middleware layer has to treat those methods specially at client-side; it has to forward them to the `AdminBase` class in the replication layer instead of processing it by itself and marshalling its parameters. It does not have to provide support for them at server-side, as calls to them will be filtered out by the replication logic before.

4 The Java RMI Interface

This section gives an description on how a middleware layer for Java RMI was built and how it can be used. We start with a short description of the basic realisation techniques before we discuss a feature called local objects in the next section. Afterwards, we discuss the `Exporter`[5] class and how methods at server-side can be hidden from the client. This section concludes with a summary on the architecture and an example how to use this RMI layer.

4.1 Overview

For realising an RMI-compatible middleware layer on top of the Virtual Node core we decided to prefer ease of use over performance. Our goal is to generate stubs that are fully compatible to Java RMI and thus might be put in RMI registries. Furthermore, client applications using an RMI object with the same interface can be used without changes to the code.

We also decided to use the `java.lang.reflect.Proxy` to dynamically generate client-side stubs. The use and generation of those is more expensive than to generate stubs at compile-time. However, we do think, that this is a minor issue as replication per se is an expensive technique. The proxy accesses the replication layer by an entity called `ReplicationHandler` that extends `java.lang.reflect.InvocationHandler`. Furthermore, we use standard Java serialisation for parameter marshalling and unmarshalling.

Methods in Java are identified by their signature, that is their name and their parameters. We exploit this signature to identify methods across address spaces. Each method is identified by an identifier that contains the hash of the method name and the deep hash of the array of parameter types. The probability of having two methods with the same identifier in the same service implementation is very small. If it happens, though, the clash will be recognised when the first replica is instantiated, so that no undefined operations can take place at run-time.

Our implementation does not require the use of a middleware adapter, as RMI stubs are serialised and deserialised with their state and not represented by an external data structure.

Basically, a replicated service is like a remote object. That is, a stub is used to invoke methods at a server which, in our case, consists of multiple replicas.

[5]all classes mentioned in this section are in package `eu.xtreemos.vnode.rmi` unless another package is specified.

```
public interface LocalObject<T> extends Serializable {

        public void setProxy(PrivateProxy _prox);
}
```

Figure 8: Local Object

All replicas run the same service implementation. A service may implement an arbitrary number of interfaces all of which are by default accessible by the client. However, in order to support a wide range of applications we have made some extensions to the RMI model that will be discussed in detail in the next sections. In short, these are the following.

- Besides the service implementation in a remote location, our services support so-called local objects, that implement functionality at client-side. This is similar to AJAX Web technology and is a Java RMI mechanism of Shapiro's proxy principle [8] and comparable to *fragmented objects* [7] and *distributed shared objects* [9].
- Methods in the service implementation can be declared to be *ignored*. As a consequence they cannot be accessed remotely.
- Methods in the service implementation can be declared to be *hidden*. As a consequence they cannot be accessed from clients, but from the stub and the local object.
- Methods in the local object can also be declared to be *hidden*. As a consequence they cannot be accessed from clients.

4.2 Local and remote objects

For the system to be able to export and replicate an object, we require similar to Java RMI that the class of this object implements an additional interface. This interface, `RRMI` (*replicated RMI*), is a pure marker interface that extends `java.io.Serializable`. We require that the object be serialisable so that state transfer can happen without additional development efforts.

Apart from the implementation of the replicated object, the exporting process supports an additional feature that has turned out to be useful in the past [2]: *local objects*. Local objects are a part of the stub and can be used in order to provide

```
public static final<T> Object
                    exportObject( Configuration config,
                                  RRMIObject service,
                                  LocalObject<T> local )
                                      throws ExportException;
```

Figure 9: RMI Exporter

local functionality that is logically part of the service to be called but whose implementation does not make sense at server-side such as encryption, caching or access to the local file system. We will discuss the benefit of such an approach in Section 4.7. Objects that shall be used as local objects have to implement the interface `LocalObject` shown in Figure 8. As a local object is part of the stub, it has to be serialisable. Unlike remote objects, local objects can never be transparent to their developer, so that there is not necessity to provide a simple marker interface. In addition, a local object might just do some preprocessing and then call the remote part of the service. For such a scenario the local object has to have a reference to the request handler that invokes methods at the server-side. In our approach this reference is realised indirectly via a `PrivateProxy` that will be discussed in Section 4.4. One consequence of the intransparency of local objects, we decided to have only one local object per stub, because this significantly eases the implementation of the exporting process. Note that this approach does not limit the developer, as it is still possible to use one local object as a facade [6] to a set of other local objects.

4.3 Exporter Class

In this section we discuss how to export a service. We use the term *export* in a way similar to Java RMI. When an object is exported two things happen. First of all, an instance of the replication framework is initialised and the service to be replicated is wrapped by it. Secondly, a stub is created that provides access to the service.

Exporting is handeled by a class called `Exporter` that has a single public method `exportObject`, which is shown in Figure 9. It takes a `Configuration` an `RRMIObject`, that is, an object that can be exported, and a `LocalObject`. If exporting is not possible, it throws an `ExportException`; otherwise it returns a proxy that can be used by the client application. All action that is described in the following happens within this method.

The stub interface visible for a client is generated using the dynamic proxy of Java. By default all interfaces of both remote and client object are used in order to generate the proxy interface. Internally, the proxy uses the `ReplicationHandler` class. In its implementation it decides whether to pass calls to the replication layer or to the local object. In order to avoid ambiguous configurations we impose the following restrictions.

- There is no non-marker interface that can be implemented by both local object and remote object. If we allowed such an environment, it would no longer be clear whether to relay an invocation to remote or local object.
- There are no two methods with identical signature that may appear in local and remote object. Here, again it would be impossible to have a unique mapping from method to the location where those methods have to be executed.
- Methods that conflict with the administration interface (c.f., Section 3.4) are not allowed. `ClientAdmin` is automatically added to the proxy interface. As it does have a fixed implementation on server-side it must not be overridden, no matter on which side.

This approach is very restrictive; however, it seems reasonable at a first glance. Yet, looking closer brings up a dilemma. As discussed above, the classes of both remote object and local object have to be serialisable. This is not a problem, as `Serializable` is a marker interface. However, objects implementing `Externalizable` are serialisable as well and should be supported. As this interface comes with two methods it is not a pure marker interface anymore so that problems arise when both local and remote object implement it. Solving this dilemma requires to recognise that the methods provided by `Externalizable` should never be intended to be used remotely. Their sole purpose is to allow serialisation which is a purely local event.

There are two approaches to solve this issue. The first one is to filter `Externalizable` by some hard coded filter rules. This approach is straight forward and does not burden the programmer. On the other hand it is very inflexible, as it does not allow for an easy extension when other interfaces with similar functionality appear. An example of such an interface could be some local monitoring entity.

Consequently, we decided to go for the second approach which uses Java annotations. We do allow methods in both locations to be marked by `@Ignored` which means that the interface they are defined in will not appear in the proxy interface seen by the client implementation. Marking does not happen within interfaces or for entire interfaces, but happens per method of the implementation of either local or remote object. We chose this approach, because of several reasons: interfaces of the Java library cannot be used if methods have to annotated in the interface. Interfaces cannot be re-used in another context once they are annotated, if interface definitions are subject to annotations. And finally, multiple interfaces can define the same or equal methods with different annotations. This would lead to conflicts, even if the implementation is most certainly unambiguous. An important consequence of the way marking happens and the way proxies are constructed, is that all other methods defined in the same interface as a `@Hidden` method do also have to be `@Hidden`. As this is a non-trivial restriction we decided to throw an exception if only some methods of an interface are marked with `@Ignored`.

Note that this does not effect the superinterfaces of this interface. In particular, it is allowed that two non-marker interfaces `A` and `B` extend the same non-marker interface `C` with the methods of `A` being `@Ignored` and those of `B` are not. This is the case, because it is still possible to add `B` to the proxy interface. In contrast, using `@Ignored` for the methods of `C` would require all methods of both `A` and `B` to be `@Ignored`, because adding `B` to the proxy interface would automatically add `C`, too, which shall not appear there.

4.4 Hiding Methods

The existence of local objects brings up another issue. There might be methods at server-side that shall not be visible for clients, but for the local object. Assume, for instance, a local object that implements a file cache for a component system such as OSGi [1]. If the file is available at client-side this file shall be used, if not, it shall be loaded from the server. In consequence the client application must not have the possibility to bypass the local caching mechanism. In the system presented so far, the only way to ensure that this is not possible, is to use the `@Ignore` annotation and exclude the method from the proxy seen by the client. On the other hand, the local object has to have a way to call the server. This might happen using the interface offered by the replication layer. Yet again, this is a bad idea, as it would require the developer of the local object to know about (de)serialisation and construction of message identifier and would require to change all local objects when the interface to the replication layer changes. We satisfy those contradicting needs by using an additional level of indirection and by introducing a new annotation type. Instead of directly using the interface to the replication layer, the local object uses another proxy. On server-side we provide a new annotation `@Hidden` which removes the method from the public proxy accessible by the client, but leaves it in the private proxy used by the local object. Apart from that they are partially visible, the same rules hold for methods annotated with `@Hidden` and those annotated with `@Ignore`. Summarising the properties of private and public proxies:

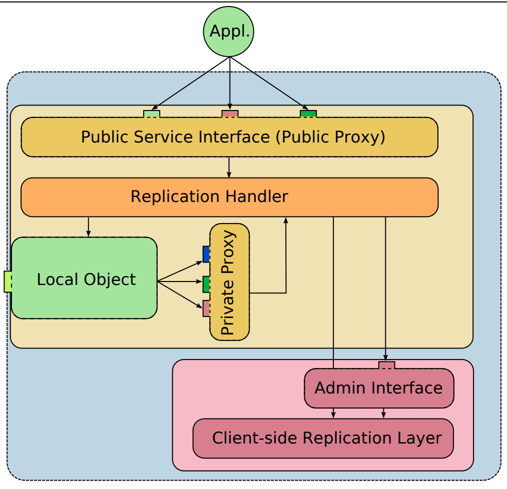
- Both proxies use the same `ReplicationHandler`.
- None of them contains methods marked with `@Ignore`
- Private proxies do not offer the interface of the local object, because the implementation can access those methods using `this`.
- Private proxies do contain server methods that are marked with `@Hidden`, the public proxies do not.
- Private proxies implement the marker interface `PrivateProxy`, public proxies implement the marker interface `PublicProxy`. This is for that the `ReplicationHandler` is able to distinguish them.

4.5 Performance Optimisations

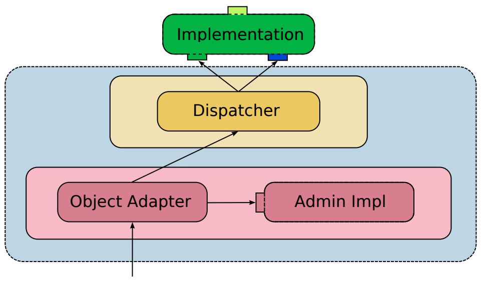
As discussed in Section 3 the replication layer queries the middleware layer about the execution characteristics of a method. If the method is read-only some optimisations will reduce the execution time. As there is no way to figure out automatically whether a method is read-only, we do rely on the programmer to tell us. She has to mark a method with `@Readonly` in order to switch on the optimisations. At initialisation time, the RMI middleware layer will build up a cache containing the identifiers of all read-only methods. This allows to answer the query quickly without having to use reflection each time.

4.6 Summary

Figure 10 sketches the architecture of the RMI middleware layer presented in this section. Furthermore, it shows how it is linked with the Virtual Node core. The arrows denote invocation sequences. The client application is bound to a dynamic proxy. This proxy offers all interfaces that are provided by the local object, the remote object, and the admin interface indicated by the light green, the dark green, and the red boxes. All invocations on the proxy result in a call to the `ReplicationHandler`. This entity decides whether this method invocation shall be relayed to the local object, serialised for the replication layer, or the administrator functionality in the replication layer is to be called. The local object has a proxy on its own, the private proxy. Just as the public proxy, it offers the interface of the remote object and access to the administration methods. However, in addition it contains all methods that were `@Hidden` from the public interface (symbolised by the blue box). It does not contain the interfaces of the local object, as those


Appl.
Public Service Interface (Public Proxy)
Replication Handler
Local Object
Private Proxy
Admin Interface
Client-side Replication Layer

(a) Client-side



(b) Server-side

Figure 10: RMI Layer Architecture

can be referred to using `this`. At server-side the `ObjectAdapter` decides whether an incoming call is for the admin interface. If so, it is passed to an entity that implements the admin interface. Otherwise, the request is passed up to the middleware layer. Of course the middleware layers at client and server-side have to use the same prototcol in order to be able to work together. For the RMI implementation of the middleware layer, there is a `Dispatcher` object that receives the call and decides which method to call. This method can either be from the public or hidden interface. As the figure shows, none of the entities makes use of methods that were marked by `@Ignore` and that are indicated by the very light green boxes at the local object and the service implementation.

4.7 Example

In this section we give a short example that makes use of all the features presented in the previous sections. Imagine a distributed version control system similar to GIT[6] with a local repository and a server-side repository. From a user point of view access to both repositories should be possible by using a single service. Assume further that for fault-tolerance reasons the server repository shall be replicated. A user works on her local repository, but can merge the changes in the local

[6]http://git.or.cz

```
public class RemoteRepository implements RRMIObject,
                                         RemoteRepositoryHiddenInterface,
                                         RemoteRepositoryPublicInterface {

        @Readonly @Hidden
        void checkout(String branch) { /* ... */ }

        @Hidden
        void mergeChanges(FileList fl) { /* ... */ }

        String ls() { /* ... */ }
}
```

Figure 11: Remote Object Interfaces

repository into the remote repository.

Realising such a system is easy with the means presented in the previous sections. We require a local object for access to the local repository and a remote, replicated object for access to the remote repository. Fragments of the two classes are shown in Figures 11 and 12. The remote object has three methods two of which - `checkout` and `mergeChanges` - are hidden from the public interface, as they have to be preprocessed by the client-side. Only `ls()` is publicly accessible. As `@Hidden` methods have to be in an interface of their own the remote object implements two different interfaces. The semantics of the `checkout` command is read-only so that the method is annotated accordingly. The local interface has two methods to preprocess the hidden remote methods. Both of them do some local work such as generating directory structures or checking for local changes before invoking the remote object. In addition serialising the entire local object including all its file handles is not a good idea, as those handles do not have a meaning on some other machine. For that reason we use the `Externalizable` interface so that we can reset all values with a scope limited to the current location and initialise them with the right values at the new location.

5 Join Protocol

This section presents a join protocol for adding new replicas to virtual nodes. Its main concern is to present the challenges and a high-level view of how to

```
public class LocalRepository extends LocalObjectImpl
                                    implements Externalizable,
                                                  LocalRepositoryInterface {

 public void checkout(String branch, String toDir) {
  /* some code */
  ((RemoteRepositoryHiddenInterface) privateProxy).
            checkout(branch);
  /* some code */
 }

 public void mergeChanges() {
  /* find out which files have changed locally */
  FileList fl = ...;
  ((RemoteRepositoryHiddenInterface) privateProxy).
           mergeChanges(fl);
 }

 @Ignore
 public void readObject(){ /* ... */ }

 @Ignore
 public void writeObject(){/* ... */}
}
```

Figure 12: Local Object Interfaces

solve them. We do not present any proofs here, nor a discussion down to the implementation level. These are subject to a technical report [4] that is currently being prepared.

5.1 Problem Statement

Virtual Nodes aim at providing support for applications that require availability and reliability. For long running applications this implies the need for being able to add new replicas at run-time. Such a new replica has to get a recent, consistent service state before it can answer requests. For that reason the serive state has to be serialised at an existing replica and then to be transferred to the joining replica. Virtual Nodes use two features that make joining new replicas difficult. First, we do not assume a closed replica group. That is, any node can host a replica. The location does not have to be known in advance, so that we cannot make any assumptions where the next replica will be instantiated. In consequence it is not possible to store data at all potential replica-locations before the service starts. Instead all state has to be obtained dynamically at run-time. Second, the framework uses deterministic multithreading algorithms for allowing both determinism and increased performance. This leads to several problems.

As replicas may join anytime, the requests for the service state may arrive at an arbitrary point in time at the service. When the state request arrives at the service, in general, multiple threads will be in execution[7] with some of them being blocked in a `wait` call. As Java does not provide any means to serialize a thread in execution it is not possible to serialise the state until all of them have terminated. However, threads blocked in `wait` will not terminate until the corresponding `notify` has been called. Thus, not starting new threads and waiting for all running threads to terminate is not an option. Instead, we have to find a way to let `notify` be called while at the same time, as few new threads as possible shall be started. Furthermore, state transfer requests should be processed quickly, as new replicas are likely to join in situations with a reduced reliability (e.g., when an old replica has failed).

Summarising the arguments made so far, an algorithm for a join protocol has to fulfil the following properties:

- It must not rely on statically defined locations of replicas.
- It has to deal with multiple and potentially blocked threads at all replicas
- The state has to be consistent. That is, it has to be a valid state and the joining replica has to be able to figure out which client requests are part of the state and which still have to be processed.

[7]where *in execution* means *not terminated*

- The downtime of the service has to be as small as possible.
- The latency experienced by the new node has to be as small as possible.
- New nodes need to have a means to propagate their contact information that all replicas can update their view on the replica group.
- The view update has to happen at the same logical time at all replicas.

5.2 Sketch of Solution

Our approach to overcome blocked threads in the schedulers is to allow pending requests to be processed one-by-one until the waiting threads are resumed and finally terminate. Note that there may exist bordercases which would never allow all threads to be terminated at the same time. This happens for instance when the only execution path to `notify` causes `wait` to be called. Another non-terminating sequence occurs if always a blocking thread precedes a notifying thread. However, we argue that those sequences and execution paths are seldom in practice and can be implemented differently if the programmer knows where the pitfalls of the system are. Yet, an entirely transparent approach to this problem is considered future work.

The algorithm uses three different kinds of messages. `GET_STATE` is broadcast by the replica when it requests the service state. This message also contains the replica's contact information. As a reply it will eventually receive at least one unicast `SET_STATE` message containing the state. Finally, after having installed the state, the replica sends a `GOT_STATE` broadcast to trigger a view update.

In more detail, the sequence is as follows. The new replica (the joiner) sends a `GET_STATE` message to all group members. The message contains the joiner's identifier and contact address. When a running replica processes this message it switches into the one-by-one mode until the scheduler queue is empty. Then the oldest replica serialises its state and sends it to the joiner via a `SET_STATE` message. After the receipt of this message the joiner replies with a `GOT_STATE` message which triggers the update of the replica group view at all replicas. The protocol is straight forward only in the absence of failures and sequencial joins. The technical report presents a more detailed discussion on how to handle replica failures and multiple, concurrent joiners. Basically, the correctness relies on the GCS feature that messages are in total order. We partition replicas in three groups according to their state: *full member*, *joiner*, and *potential joiner*. In addition, we order the members in each group by the logical time their last state message was received. We call the position in the ordering the age of a replica; the technical report shows that the join problem can be solved if there is exactly one oldest replica at all times and that our protocol guarantees that.

6 Group Communication

Evaluation of the group communication systems described in D3.2.5 [3] has shown that none of the two group communication systems, *Jgroups*[8] and *SPREAD*[9], we planned to support is perfectly suited for replication with Virtual Nodes. Jgroups lacks support for uniform multicast. That is, the sequencer might deliver and process requests bypassing the ordering of the group communication. In consequence this replica might have processed a request and also replied to it before other replicas have even seen this request. If the sequencer replica has sent a reply and crashes before the message was delivered to at least one other replica, the client experiences a phantom update and might continue working on wrong assumptions. This can not be tolerated in a fault-tolerance infrastructure.

Spread, in turn, does support uniform multicast, but lacks support for dynamic replica groups. Machines that want to run Spread do have to run a Spread daemon that has a static configuration of hosts that might join a communication group. It is not possible to update the configuration without restarting the daemon. Such a static configuration cannot be tolerated in a highly dynamic environment such as peer-to-peer grid systems.

As a consequence we consider three options for future work. First, evaluate how to extend Jgroups with uniform multicast. Second, evaluation of how to change Spread so that it does allow dynamic composition of replica groups. Third, investigation of how to support our own group communication system.

7 Performance Evaluation

Performance evaluation is still work in progress and will be subject to the sequel of this document. The delay is caused by the fact that the code has not yet reached a maturity level that would allow long running evaluations. Another cause for the delay is that it is very time-consuming to set up a testbed that allows to evaluate the effect of node failures. We clearly underestimated this issue.

Nevertheless, there are plans what tests Virtual Nodes should be stressed with. First of all, we want to evaluate the overhead induced by the replication framework. That is, we have a test with a single replica that is accessed via the RMI middleware. Then we compare the results to accessing the same service via standard Java RMI. In a second test we want to evaluate the scalability of the Virtual Node system in LAN. We start with a single replica and increase the number from test to test until the overhead prevents that any work can get done. We do that test for each replication strategy implemented. Another test will evaluate scalability

[8]www.jgroups.org

[9]www.spread.org

in WAN. Then we want to investigate the down time caused by the failure of a single replica depending on the replication strategy and the network environment. Finally, the performance of the join protocol shall be measured.

This set of test run will give a good overview on the performance of Virtual Nodes. It will show when to use the framework and when to use other techniques.

8 Relevance to XtreemOS

As future work, we consider mainly the investigation of the feasibility of an integration of Virtual Nodes with other components in XtreemOS. Candidates for such an effort would be CDA and RCA of WP3.5, the JobManager of WP3.3 and a XATI front-end for Virtual Nodes so that all services using XATI as their middleware layer can be subject to replication.

This integration requires a careful analyses of the services' code in order to find locations that use code that is not deterministic. This code has to be replaced by a deterministic implementation. As there is no general approach to make implementations deterministic we have to evaluate that on a per-case basis.

9 Conclusion

In this deliverable we mainly discussed the user view on the system. We presented details that are relevant to the three different roles we were able to identify for users. For administrators we discussed how to configure the framework. For application developers we explained how to access the features of the Virtual Node RMI middleware layer. And for middleware developers we sketched how to integrate Virtual Nodes into their middleware system.

Regarding the replication layer we presented an approach to a state transfer protocol that allows to add new replicas to a running system in spite of multithreading withing the replicas. We discussed weaknesses of current open-source group communication systems and identified ways how to overcome them. Finally, we presented the current state of a performance evaluation and discussed the relevance of Virtual Nodes to the rest of XtreemOS.

References

[1] OSGi Alliance. Osgi service platform, core specification 4.1, 2007.

[2] P. Baumann. Konzeption und Implementierung einer dezentralen und fehlertoleranten Versionsverwaltung. Studienarbeit SA-I4-2006-16, Universität Erlangen-Nürnberg, December 2006.

[3] J. Domaschka. D3.2.5: Design and specification of a virtual node system, November 2007.

[4] J. Domaschka. A join protocol for actively replicated, multithreaded systems. Technical Report to be published, Institute of Distributed Systems, Ulm University, Germany, 2008.

[5] J. Domaschka, A. I. Schmied, H. P. Reiser, and F. J. Hauck. Revisiting deterministic multithreading strategies. In *Proceedings of the 9th International Workshop on Java and Components for Parallelism, Distribution and Concurrency (in conjunction with IPDPS 2007, Long Beach, CA, USA, March 26, 2007)*, 2007.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[7] F. J. Hauck, E. Meier, U. Becker, M. Geier, U. Rastofer, and M. Steckermeier. A middleware architecture for scalable, QoS-aware and self-organizing global services. In *Proc. of the 3rd IFIP/GI Int. Conf. on Trends towards a Universal Service Market*, LNCS 1890, pages 214–229. Springer, 2000.

[8] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.

[9] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999.