Project no. IST-033576

XtreemOS

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

Design and Specification of a Virtual Node System [a]
D3.2.5

Due date of deliverable: December 01st, 2007
Actual submission date: December 03rd, 2007

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
Department of Distributed System
Ulm University
James-Franck-Ring O-27
89069 Ulm
Germany

Version 1.0 / Last edited by Jörg Domaschka / 03rd December, 2007

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)	

[a] The code of the virtual node implementation will be uploaded to the gforge repository as soon as it has been tested

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.0	02/11/07	Jörg Domaschka	ULM	first draft
0.1	08/11/07	Jörg Domaschka	ULM	internal review
0.2	26/11/07	Jörg Domaschka	ULM	included comments of internal review
0.3	29/11/07	Jörg Domaschka	ULM	added Sections 6, 7, 8
1.0	03/12/07	Jörg Domaschka	ULM	final check

Reviewers:
Erica Yang (STFC), Javier Noguera (T6)

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

Abstract

This document presents the design and specification of the XtreemOS Virtual Node infrastructure. Virtual Nodes provide fault-tolerance for applications by replicating them over multiple nodes. As replication performance heavily depends on application characteristics and replication protocol, Virtual Nodes offer a wide variety of paramters that can be tuned in order to optimise the system. In the following we give a survey on replication protocols, describe how all of them can be integrated in a common framework, and discuss the restrictions they impose on the replicated application. Moreover, we present the system run-time behaviour and an example application. We do not cover all details about programming Virtual Nodes, as this will be subject to the *Virtual Node Programmer's Guide*, a companion paper

Contents

1 Overview and Goals

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 (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. The development process for a replicated application is similar to developing a Java RMI application, applications using Virtual Nodes can be implemented as if they would use a regular RMI object. This is due to our RMI-based frontend.

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.

Throughout the text we use the term *service* for an entity that can be interacted with in a request-response manner. We assume services comparable to remote objects. That is, the service has a well-defined interface that allows only a limited number of types of requests. Unlike Web Services our services are *stateful* and have *identity*. By the introduction of state, we explicitly allow the occurrence of operations that manipulate this state. Identity, in turn, means that multiple services with the same interface may exist even on the same host and each of them can be distinguished from each other. Furthermore, each of the request types is associated with a method implementation at the service. We do not restrict the operations that are allowed during the execution of the method. However, to ensure that all replicas of one service are consistent with each other, it is necessary to treat methods special that might produce a nondeterministic outcome. As we do not see much sense in replicating services that are exclusively used by a single client, we assume that a service is likely to be accessed concurrently by multiple parties. In consequence it becomes necessary for the programmer to ensure mutual exclusion in the service implementation. Multithreading, in turn, might again raise consistency conflicts due to unpredictable scheduling decisions. To allow the service developer to use the standard Java programming model, we use an additional code transformation step in the development process that links the service implementation to an application level scheduler which comes with the

Virtual Node library and ensures determinism even in face of concurrency.

What we do currently not cover with Virtual Nodes is the deployment of both code and replicas. It is out of the scope of our work so far. For the time being the implementation requires that there be an administrator who decides which parts of an application have to be replicated, chooses the replication factor and selects the nodes the replicas run on. Accordingly, the decision if and where to start new replicas is also left to the administrator. It is clear that this assumption will have to be dropped in order to realise a system that remains manageable even in the face of medium to high churn. Nevertheless, our assumption is reasonable for this document, because the automatisation of Virtual Nodes is widely orthogonal to the replication functionality, so that adding additional management capabilities will not invalidate the information presented here.

The remainder of this document is structured as follows. In the next section we have a look at different replication protocols. Section 3 discusses the static architecture and with it the different components the system consists of. In Section 4 we consider run-time issues such as adding new replicas and handling nested invocations. Section 5 gives an introduction to how to program services using Virtual Nodes.

2 Overview on Replication Strategies

One of the goals of Virtual Nodes is to offer a framework for various replication protocols. It is possible to divide replication protocols in two main categories: active and passive, as well as several sub-categories. In the following we give a short overview on the different classifications following *Wiesmann et al.* [13] and *Défago and Schiper* [4].

2.1 Active Replication

Active replication (State Machine Approach) protocols lack a centralised entity. All replicas receive and also process a request without comunication. As a consequence active replication requires that replica implementations be deterministic. That means, when each of them starts in the same state and gets the same input in the same order, then they will all produce the same output and change their internal state in an identical way while processing input. A convenient mechanism to ensure the same order on inputs is a total order multicast (abcast) [8].

There are two approaches how messages are passed to the replicas. In the first approach clients send a request to all replicas and accept the first valid response. In the second approach the client sends the request to a single replica which in turn forwards it to the other replicas.

Semi-active replication is similar to active replication. The difference is that it does not require that the implementation be deterministic. For each non-deterministic operation one of the replicas takes the role of a leader, executes the operation and forwards the result to the other replicas.

2.2 Passive Replication

In the basic version of passive replication (Primary-Backup), the client sends its request to the primary replica which executes the request and forwards the state changes to its follower replicas. When the primary fails, the client notices this by either timeout or some failure detector and re-sends the same request to the replica that has become the new primary. In case the former leader has not crashed, but was just suspected by accident, then it is forced to crash in order to avoid inconsistencies. In an extension to that basic version (Multi-Primary) the primary replica is not fixed, but changes due to some scheme. This requires coordination of interfering updates running at different primaries. In another extension (Coordinator-Cohorte) the client sends its request to all replicas in order to avoid the need for re-issuing it in the case of primary failure.

Semi-passive replication uses the same mechanism to pass requests to the replicas than coordinator-cohorte. However, the decision which replica processes the request depends on some algorithm that is similar to a consensus algorithm. Instead of each replica proposing an update value only a single one does so—the current coordinator—while all other replicas propose a void value. In case one of the follower replicas suspects the leader to have crashed it computes its on value, i.e., it executes the request and proposes this value to the other replicas. Eventually, there is a consensus [8] on which value to chose.

2.3 Replication Protocols for Virtual Nodes

For the first prototype of the Virtual Node infrastructure we target the implementation of both an active and a passive replication protocol. To enable non-deterministic operations we also integrate a hook to realise semi-active replication. Note that some literature propagates that there is no need for determinism in passive replication (e.g. [13]). This, however, is not true in the general case, but only when the service implementation is self-contained and in addition the leader only forwards its entire state or state changes to the followers. The self-containment guarantee is among others violated in the following cases:

- The service invokes a method at another service
- The service writes data to a shared file system or uses other ways to make data externally available.

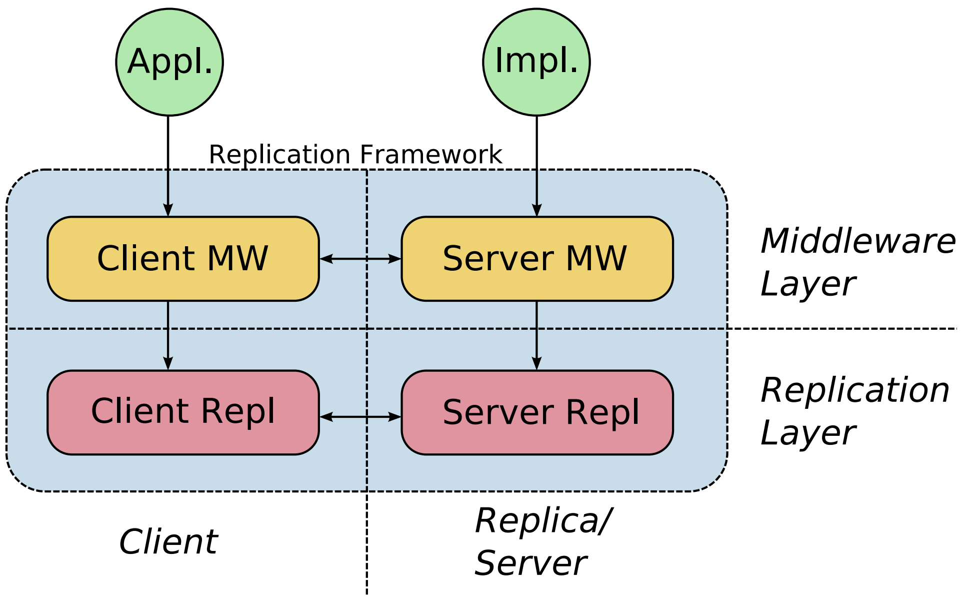

Figure 1: The Overall Structure of the Virtual Node Infrastructure

- Parts of the service state can also be accessed otherwise, e.g., by shared memory.

If the leader crashes before the update message was forwarded to the followers the new leader has to re-execute the request and in consequence also invoke the external service which might lead to inconsistencies; the same holds for data written by the service if it is accessible by third parties.

Forwarding the entire service state is very expensive in case of large states. An option to reduce the costs is to transmit only the part of the state that was modified by the last method invocation. Yet, such a procedure requires insight to the service which is not available for a general purpose replication framework operating outside the service. For that reason the system offers the option to serialise the state only after a number of requests. Between two serialisation points (*snapshots*, *checkpoints*) the system logs the requests. As a new leader has to re-execute those requests if the primary fails, the service implementation also has to be deterministic for passive replication. Analogous to semi-active replication the system offers hooks to also log the results of non-deterministic operations.

3 System Description

Design-wise the Virtual Node infrastructure consists of four parts (cf. Figure 1). The vertical division distinguishes two parties: server and client. The horizontal division separates features unrelated to replication such as interface description, marshalling and application binding from replication-related issues such as the management of membership, consistency, communication, and scheduling. We use the term *middleware layer* for the first part that is closer to the application and *replication layer* for the part concerned with replication issues. Figure 1 shows all the resulting four components together with client application and service implementation which are both outside the Virtual Node infrastructure. The figure also sketches the dependencies between the components. The application depends on the middleware layer at client-side as this defines how to bind to the service and how to invoke methods. The middleware layer at client-side depends on the client-side replication layer which defines the respective interfaces. The same holds for the server side. The client-side parts also depend on their respective counterparts at server side and vice versa. The replication layer of both the client and the server side have to be compatible with each other. Each of the components consists of several sub-components for some of which we provide multiple implementations. Currently, the choice which of them to use is a compile-time consideration. For the prototype due to M18 we focus on a start-up time configuration. On the long run, we are planning to allow dynamic configuration changes at run-time. In the following, we discuss the layers in detail.

3.1 Client-Side Middleware Layer

Our client side middleware layer is completely Java RMI compatible. That is, it uses classes and interfaces provided by the `java.lang.rmi` package to hide from the application the fact that it uses a proprietary communication logic. In consequence, the application treats the reference to the service as a Java RMI stub. Regarding functionality, the middleware layer realises two tasks that are encapsulated in the *Marshalling* and *Binding* sub-components shown in Figure 2. The first is concerned with parameter marshalling and unmarshalling. Secondly, it cares about binding issues. The first issue is easy to understand: Parameters of a method call have to be transferred by wire and thus have to be (de-)serialised. The need for binding support is less obvious. However, when the client application passes the service, that is, technically speaking the stub, as a parameter to another object, then the parameter gets marshalled if the target object resides on another node. In that case the stub is serialised and again desiralised at the target node. During the deserialisation process the newly created stub has to get all information required to instantiate the replication layer including its configuration and contact

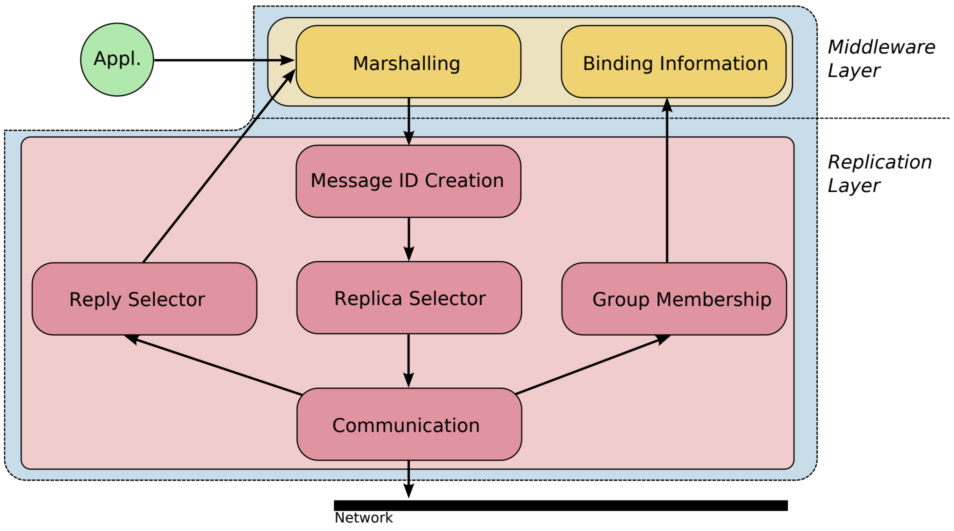

Figure 2: The Information Flow at Client-Side

information for the replicas. As the group of replicas can change over time and the middleware layer has no direct access to this information we have integrated a callback mechanism used by the replica layer to inform the middleware layer about changes. A detailed discussion on that issue can be found in [6].

3.2 Client-Side Replication Layer

The replication layer is invoked by the middleware layer calling a method `invoke( String, byte[])` with the `String` representing a signature identifying the method and the `byte[]` containing the marshalled parameters. The sub-components and data flow described in the following are also shown in Figure 2. The replication layer then generates a unique message id consisting of the unique identifier of the host it is running on (the IP address), a `java.rmi.server.UID` generated at the moment the stub was initialised, and a consecutive number. Subsequentially, the message is assembled. Afterwards, the replication layer selects a replica to contact, opens a connection and send its request to that replica. Which selection strategy the *Replica Selector* uses is a configuration issue. The decision may be influenced by the replication protocol. It is better to always select the same replica (i.e. the primary) in case of passive replication and use a randomized selection strategy for load balancing in case of active replication. Finally, the request is sent and the invoking thread blocks until a reply has been received. As the reply may contain the return value of more than one replica a *Reply Selector* decides which of them to chose. Just as the selection strategy, the reply selection

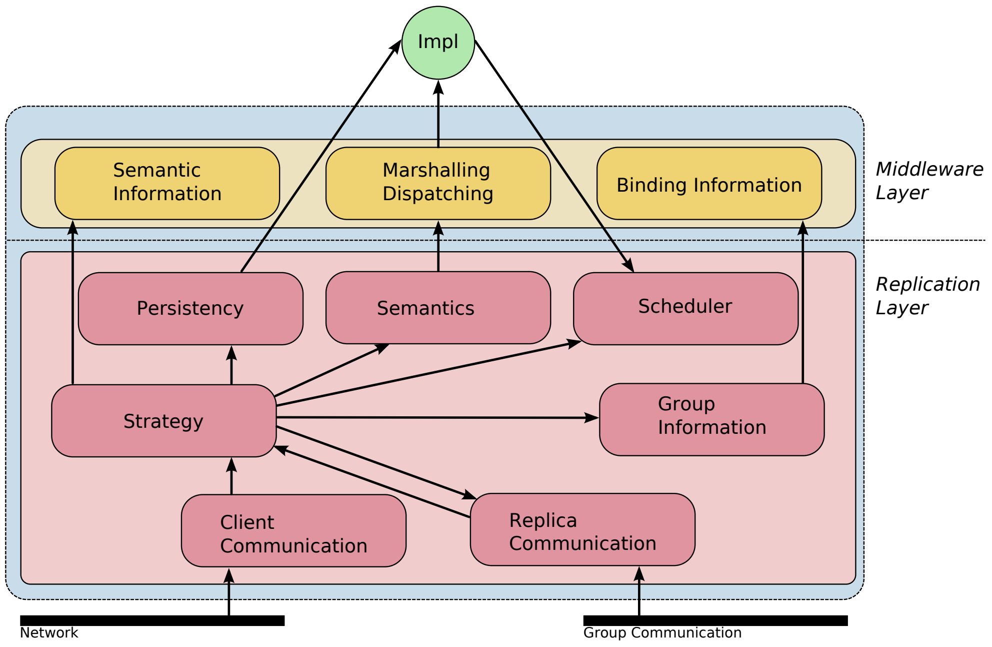

Figure 3: The Information Flow at Server-Side

strategy is an configurable parameter. In the first prototype we only provide a single implementation that returns the first valid reply. If any error has occurred during the invocation so that no valid replies are available (e.g. the contacted replica crashed), the selection strategy chooses another replica to contact and the request is re-sent. If no further replicas are known to the Replica Selector, the system returns a `RemoteException` to the client.

The request messages also always contain the current view of the client-side replication layer on the replica group, i.e., which replicas currently form the group and their contact information. The reply it receives from the replicas contains the latest version of this view if the client view lacks up-to-dateness. In that case, the replica layer triggers the callback to the middleware layer to pass the information about the changes.

3.3 Server-Side Replication Layer

The replication layer at server side is far more complex than the other components of the system. Here, most of the sub-components are concerned with ensuring consistency of the replicas. As Figure 3 clearly shows, the sub-component implementing the replication strategy (*strategy* component) is the central entity at server side. All incoming events triggered by external entities (aka clients) or other replicas first pass through the strategy sub-component. This is due to the fact that passive and active replication strategies only have little in common when it comes to reacting to events. All common patterns are sourced out to the *Semantics* sub-component, which is mainly concerned with ensuring at-most-once semantics of method executions. For that purpose it caches incoming requests together with their respective outgoing reply using the request's message id as a key. In order not to punish idempotent or stateless applications such as Web or NFS server the caching can be turned off. Furthermore, the Semantics sub-component contains the implementation of remote methods provided by the Virtual Node infrastructure, that is methods the client can invoke, but which are not part of the service implementation. We will discuss infrastructure methods in more detail in Section 4.3.

Some replication protocols require that the state of the service implementation be made persistent either by sending it to other replicas or by writing it to stable storage. This functionality is implemented by the *Persistency* sub-component.

The *Client Communication* sub-component serves as an interface for the outside world. This is the contact point clients address their requests to and also the only part of a replica that is visible. The communication with other replicas happens through the *Replica Communication* sub-component, which hands over incoming messages to the Strategy component. Those messages will mainly be requests, when using active replication, state update when using passive replication Finally, the Replica Communication sub-component can also receive view change messages, which indicate that either one replica is suspected to have crashed or that a new node requests to become one of the replicas. A change in the composition of the replica group eventually reaches the *Group Information* sub-component, which stores the current group composition and is the entity to tell if the view a client has sent to the service is still valid. The detailed actions triggered by a change in the group composition are subject to Section 4.3. Finally, there is the *Scheduler*. Each request that is processed is executed in its own thread. The scheduler is an application-level entity that restricts the number of threads being visible to the system (JVM) scheduler.

Three of the sub-components in the server-side replication layer are subject to configuration: The scheduler, the replication protocol, and the replica communication. Note that the configurations of those components are not fully independent. However, the current prototype does not enforce a reasonable configuration. In the following three subsections we have a closer look at the available configuration options of each of these sub-components.

Basic Strategy	Persistency	State Serialisation	Adapted Variants
Active Replication	———	———	Semi−Active
Passive Replication	Stable Storage	After Each Request	Semi−Passive
	Stable Storage	After *n* Requests	Semi−Passive
	Stable Storage	Interval Based	Semi−Passive
	Other Replicas	After Each Request	Semi−Passive
	Other Replicas	After *n* Requests	Semi−Passive
	Other Replicas	Interval Based	Semi−Passive

(a) Strategy

Abbr.	Name	Features
SLT	Single Logical Thread	———
SAT	Single Active Thread	Condition Variables
LSA	Loose Sync. Alg.	Cond. Vars., multithreaded
MAT	Multiple Active Threads	Cond. Vars., multithreaded

(b) Scheduler

Figure 4: Configuration Options for Replication Strategy and Scheduler

Strategy Instances

The system supports both replication protocols described in Section 2: active and passive replication. Active replication was implemented in a straight forward way. A request is passed to all replicas which execute it and finally the result is returned to the client. There are no configuration options. For passive replication a variety of parameters can be set. First of all, there are two different ways to make the information persistent. It can either be written to stable storage or be sent to the follower replicas. The first approach allows to start the new primary replica on demand leading to higher take-over time in case of failures. A basic prerequisite therefore is that all hosts that potentially run a replica share a common storage, such as a distributed file system. The second approach requires that the follower replicas be up and running, so that they can handle the messages they receive. For both approaches the system provides a persistency sub-component realising exactly this behaviour. In addition, the passive replication strategy allows to log requests and only serialise the entire state after a configurable number of requests or after an also configurable amount of time has elapsed. How to add support for non-deterministic operations is sketched in Section 5.1, configuration issues are discussed in Section 5.3.

```
void s_lock(Mutex);
void s_unlock(Mutex);

void s_wait(CondVar);
void s_wait(CondVar, long);
void s_wait(CondVar,long,int);

void s_notify(CondVar);
void s_notifyAll(CondVar);
```

Figure 5: Scheduler Interface

Scheduler Instances

As already described above, the schedulers operate on application level and reduce the number of threads that the system scheduler considers running or ready to run. This is realised by redirecting any operation related to synchronization and condition variables to the Virtual Node scheduler. For that reason the scheduler interface shown in Figure 5 reflects all operations known from Java synchronization prior to Java version 5. During the extended development process (cf. Section 5.1) all occurences of those operations are redirected to the Virtual Node scheduler. At run-time there is exactly one scheduler per service instance.

Currently we support four different kinds of schedulers. The *SLT (single logical thread)* [9] scheduler realises a strictly sequential execution of requests in which the thread for the next request is not started until its predecessor has finished its execution. The *SAT (single active thread)* scheduler implements an algorithm [14, 5] which also has only a single thread running at a time, but in contrast to SLT is able to start another thread in case the currently running thread blocks. This allows the use of condition variables, i.e., `wait` operations, something that is not feasible using SLT. SAT is able to use the full capacity a single processor node provides. Yet, as it offers at most one runnable thread to the lower level schedulers, it is not able to utilise the facilities of multi-core processors or multiprocessor systems. For that reason the Virtual Node infrastructure also supports two additional algorithms with support for real multithreading. The *LSA (lose synchronisation algorithm)* [2] scheduler uses a leader-follower scheme to assign locks in deterministic order. The leader executes lock requests in arbitrary order and broadcasts the order to its followers. The algorithm used by the *MAT (multiple active threads)* [10] scheduler is an extension to the SAT variant. It uses a privileged primary thread that is allowed to request locks whereas all other threads are allowed to perform computations, but are blocked once they request a lock.

Please note that for passive replication strategies currently only the SLT scheduler is suited. This is due to two reasons. Firstly, if request logging is not used, the service state has to be serialised after each method invocation. Moreover, the serialised state must not contain changes caused by the execution of other threads. This cannot be guaranteed in case of multithreading. Secondly, when request logging is used, so that the entire state is serialised only from time to time, multithreading is feasible in between two checkpoints. However, this requires that it can be guaranteed that by the time the snapshot is to be taken, no threads are manipulating the state. Yet, this is impossible in general as a thread might have called `wait`. Due to the blocking characteristics of this operation and the uncertainty of the next `notify` call, there are no guarantees.

Replica Communication

In the current implementation, the replica communication sub-component is a facade to a third party group communication system (GC). The preconditions an external GC has to satisfy are rather low. First of all it has to support an—ideally uniform—total-order broadcast [8] among the replicas. Secondly, the Virtual Node implementation assumes virtual synchrony for messages. That is one message can always be associated to a set of replicas. In addition the GC has to provide a mechanism that allows to get information about group changes. Currently, we support the JGroups[1] group communication system. As it provides only a non-uniform abcast, we plan to also add support for the SPREAD[2] GC. To integrate a new GC system in the current infrastructure, the developer has to provide a GC-specific factory, which is able to handle the GC startup and configuration. Furthermore, (s)he has to include the new GC-specific factory in the general factory.

3.4 Server-Side Middleware Layer

The middleware layer at server side contains three sub-components. The *Marshalling/Dispatching* sub-component is the server-side counterpart to the *Marshalling* sub-component at client side. Accordingly, its interface is identical to the client-side replication layer: `invoke(String, byte[])`. Again, the `String` represents the method to call, and the `byte[]` contains all serialised method parameters. By the method information, the *Dispatcher* determines how to interpreted the information in the `byte[]`. Afterwards, the unmarshalling happens followed by the method invocation at the object implementation. The

[1]www.jgroups.org
[2]www.spread.org

Binding sub-component fulfils the same task as at client side. If the service implementation passes itself as a reference to some remote service or returns `this` to an invocation, the serialisation process has to generate a stub and thus be aware of the current membership information. Finally, the *Semantic Information* sub-component contains semantic information about the methods implemented in the service, if the developer has provided such information. This topic is targeted in more detail in Section 5.

3.5 Future Work

The current implementation is a prototype lacking sophisticated features that should be supported by a final implementation. We will try to reach this goal within the restrictions imposed by time and human resources. In detail the features include:

- **Infrastructure Sharing:** Currently, the software assumes that each Virtual Node runs in a Java Virtual Machine on its own. This wastes resources as services instances of different Virtual Nodes might easily share the infrastructure. For instance they could share a common contact address as it is typical in CORBA, Java RMI, or Web Service technology.

- **New Threads:** Because of the restricted multithreading model, the creation of new threads of activity during the execution of a request is not supported in the current version.

- **Uniqueness:** The use of IP addresses as the identifier for a client is not truly unique, as clients might share a common IP adress. This should be taken into account in the next version of the software.

- **Extendibility:** The use of Java as the implementation language should not limit the possibility to port the current implementation to other platforms. In that case, it would be welcome to have an integrated approach that — comparable to CORBA systems — allows different collaborating hosts to run the Virtual Nodes architecture implemented in different programming languages. This would also be a first step towards mulitversion programming. In detail this would require the following changes to the current implementation:

  - **Java Serialization:** The use of Java Serialisation has to be abandoned, as it is not or difficult to port to other programming languages. In addition this might also reduce the amount of data to be transmitted, as only required data will be sent.

  - **External Data Representation:** An external data representation format is to be specified. CORBA CDR could be an option.
  - **Reflection:** All parts of the system that are subject to Java reflection have to be replaced by a mechanism that is portable to other languages. Here, too, much work has already been done for the CORBA specification.
  - **Middleware:** As Java RMI is not suitable for other programming languages than Java, an alternative middleware layer has to be supported.

- **Additional Scheduler:** Literature knows another algorithm for multithreaded scheduling: *preemptive deterministic scheduling (PDS)* algorithm [1]. As it provides completely different performance characteristics than the others, we want to integrate it in our replication framework.

- **More Concurrency:** We have discussed [7], but not implemented yet, ways how to further increase the concurrency if determinism is a hard requirement.

- **Evaluation:** We have not yet evaluated the performance of the framework and the respective performance of the schedulers. This is a high priority goal for the time after M18.

4 Runtime Behaviour

So far, we have mostly focused on the description of the static architecture of the Virtual Node software. In this section, we discuss dynamic behaviour and cover things that have not or only insufficiently been presented so far. We start with a discussion of the regular operation in the next subsection; i.e., what happens when no nodes fail. Then, in Subsection 4.2 we show how to start a Virtual Node service and in the succeeding subsection we present how the system reacts to changes in the replica group composition. Here, we also present how to add new replicas to a running Virtual Node. Finally, in Subsection 4.4 we sketch how the system deals with nested invocations.

4.1 Normal Operation

During normal operation, that is in the absence of new replicas joining or existing replicas crashing or leaving, the system provides replicated method invocation to a set of clients. Most of the individual steps that happen during this process have already been sketched in Section 3. Here, we provide an in-depth view. We do not explicitly distinguish between different replication protocols. Instead, we present a generalised protocol that covers all of them. For dedicated implementations some phases of this protocol might be empty.

The first step to be done for being able to execute remote methods at a Virtual Node is **binding**. Binding enables the client-side to contact the server(s). As we use a Java RMI layer to hide the Virtual Node infrastructure, binding to a Virtual Node means to bind to a remote reference (stub) representing a Virtual Node. Concretely, this is loading the seralised form of the stub from somewhere (e.g., a naming server). During the deserialisation process, also the replication layer of the client-side is initialised.

Once the binding is finished, the client is able to invoke remote methods. All layers that were described in the last section are passed: The middleware layer at client-side marshals the parameters and passes them on to the replication layer. Here, a replica is selected (*contact replica*) and a request is sent to it. Beside the method to invoke and a unique request ID, the request contains the current view the client has on the replica group composition and a list of former methods from the same client that were successfully returned to the invoker. At server-side, first the request is logged (not in all protocols). Afterwards, the request is spread to all other replicas supposed to process it. There, the information about successful former requests is evaluated and cache entries matching those requests are removed. If the ID of the current request is already contained in the cache together with a reply, i.e. the request was already process before, the cached reply is returned. If the ID is contained in the cache without a reply, i.e., the request is still being processed, then the current request is ignored. Otherwise, the request is subsequentially passed on to the middleware layer where the parameters are unmarshalled, the service is invoked, and the results are marshalled again. The replication layer assembles the reply, adds it to the cache, and passes it back to the contact replica. The contact replica collects a configurable number of replies and merges them to a single reply to which it adds the latest information on the current replica group composition. Then it makes the new state persistent. Finally, the reply is sent back to the client where it is evaluated and one of the individual replies is returned back to the calling process.

The invocation process described so far may be too heavyweight for operations that do not modify the state of the service, so called *read-only operations* or *queries*. As the system cannot recognise those operations on its own, it depends on the developer who has to specify them so the knowledge is available in the Semantic Information sub-component. For executing those kinds of requests consitency requirements can be relaxed. Thus, it is neither required to log the request, nor to spread it to all replicas, nor to make the new state persistent. The consequence of all those relaxed requirements is that it becomes possible to execute one request exclusively at a single replica. For services which have a load pattern

```
 java.lang.remote.Remote
        startup(String[3], Object);
```

Figure 6: Initialisation Method

with a high quota of queries, this optimisation yields not only availability but also performance benefits compared to a single server solution.

4.2 Starting a Service

Warning: This subsection describes a part of the software that is still work in progress

Starting a new replicated service is identical to starting the first replica of this service. The startup process for this purpose is different to just adding new replicas as it requires to establish the entire infrastructure including the middleware layer. Thus, the startup process heavily depends on the middleware layer. Furthermore, in order to allow the instantiation of other replicas, the nodes that shall run those replicas have to be able to use the remote invocation mechanism provided by the first replica, i.e., they have to use a stub. In consequence, the first replica has to provide a stub, which is again specific to the middleware layer. For the RMI-compatible middleware layer that ships with the Virtual Node software, all of the instantiation process can be done with calling the `startup` method of the `vnode.rmi.Stub` class shown in Figure 6. The parameters and return value are as follows:

- `String[3]`: The first entry of this array contains the system configuration string (c.f. Section 5.3). When the method returns, the second and third entry will contain the service ID and the ID of the replica, respectively.
- `Object`: The service to be replicated represented as an object. For building a stub, the system extracts the interfaces the service implements, as stubs can only be built out of interfaces and not out of classes. If the `Object` only implements marker interfaces or no interfaces at all, the stub will contain no methods except for the administration methods (see next section).

The step described here is comparable to the object exportation mechanism known from standard Java RMI. Apart from the initial startup there are various other methods to handle the replica group composition. All of them are provided by the replication layer. Thus, functionality-wise they do not depend on the middleware layer. However, in order for the client to be able to invoke those methods, the stub of the middleware layer has to have them in its interface. The methods are shown in Figure 7 and discussed in the next section.

```
String getInstantiationData();
void startNewReplica(String);
void killReplica(String);
void killService();
```

Figure 7: Administration Methods

4.3 Changing Replica Group Composition

Warning: This subsection describes a part of the software that is still work in progress

To cover all possible changes of the replica group composition it is necessary to distinguish three cases: adding a new replica, removing a replica by intention, and removing a replica due to a hardware or process crash. By default each service comes with additional methods in its interface; so-called administration methods which are shown in Figure 7. Those methods are added automatically when the dynamic proxy is created for the first time and are implemented in the replication layer at server side. To start a new replica the first thing to find out are aspects about the current configuration that are necessary for starting a new replica. Such data is for example, the chosen group communication system and its configuartion. This can be achieved by the `getInstantiationData()` method. The return value of this method can be used as a parameter for the `startNewReplica()` method that initialises the replication framework and service replica: In a first step, the group communication system is brought up and started. Once group membership is established, the new replica is known by the other replicas, but is not a full replica yet, as it does not know the current state. The first step to get the current state is to use the communication system to send a `getState` message to all other replicas, containing the external contact address of the new replica. At the receipt of this message all replicas stop accepting new requests and wait until all running threads have finished. Once this is the case, the oldest replica serialises the service state and replies to the new replica with a `setState` message containing the state and the message ID of the last request that is reflected in the state. With this message, all replicas add the new replica to their Group Information sub-component and also forward its contact information to the Binding Information sub-component of the middleware layer for that clients are able to access it. The new replica installs the state and applies all requests that have arrived between sending the `getState` and receiving the `setState` message which were received after the message with the message ID contained in the `setState` message.

The `killReplica()` method kills the replica with the ID represented by the parameter. The result is equivalent to a replica running on a crashing host. The group communication system detects the failure and signals this event to the replication layer which passes it on to the Group Information sub-component and also to the Binding Information sub-component. Finally, the method `killService()` kills all replicas belonging to a certain service.

4.4 Nested Invocations

In replicated systems special care has to be taken for *nested invocations*. Nested invocations are invocations to other services that happen while one request is processed at a service. Mostly harmless in case of non-replicated services, they become a serious consistency threat when the invoking service is replicated. The reason therefore is most obvious, if we assume active replication. As all replicas process the operation simultaneously, all of them will also invoke the external service. In consequence the external operation will be executed n times in case n is the number of replicas. As each invocation might change the service state and also have a different return value the consistency of the replicas will in the general case get lost. In the following we sketch our approach to handling nested invocations. We assume that the only kind of nested invocation we have to deal with are invocations to other services that are also implemented using our framework. In case the service implementation does not satisfy our assumption, the programmer has to assure consistency on his own. In Section 5.1 we present our approach to help him in with this task.

The first approach to avoid inconsistencies due to nested invocations is to ensure that all of the replicas use the same message ID for their respective invocation. In that case, the other service will recognise the requests as duplicates and deny a repeated execution. Nevertheless, all phases of a regular invocation process are executed. This is not a problem, if the invoked service is not replicated, but generates an overhead of messages if the target is also replicated. To circumvent such a high load on the target service we decided that in the default case only the contact replica executes the nested invocation and passes the result on to the other replicas. In case the contact replica fails while the others are waiting for a result, they will notice the failure due to the notification mechanism of the group communication system. Instead of the contact replica the oldest replica executes the nested invocation. This hard-wired strategy is due to two reasons: Firstly, it avoids the execution of an expensive election algorithm and secondly, studies have shown that it is very likely that nodes that have been online for a long time will remain online [12].

Also in the case of passive replication nested invocations need to be handled with care. Assume that the primary replica crashes after having started the invocation. The new primary has to use the same message ID to avoid a potential re-execution of the same request. Furthermore, when message logging is enabled, we do also log the replies of the nested invocation. Mainly, because of the following reason: When request logging is used, there is no upper time bound until the next snapshot, and as both services are in general unrelated there is also no guarantee how long replies are cached. Accepting an accidental re-execution due to flushed caches and a primary failure long after the nested invocation but before the next snapshot is not an option as it might invalidate replies already returned to the client therewith destroying the consistency of the entire system. Consider such a scenario: a request is re-executed based on the log. During the execution the process triggers a nested invocation that is not recognised as a duplicate, but is also re-executed. In the best case, its result is the same as before, in the worst case it corrupts the state of the callee. In the general case, it will simply return a different result than at the first invocation leading to different a state and reply at the caller. In consequence, the client which has already received the first reply cannot rely on it anymore, yet it does not know. To avoid such weird constellations we do not only log requests, but also replies to nested invocations.

Implementation-wise we attach context information to the thread that processes a request. The context contains the message ID of request that is currently processed, the service ID, as well as a link to the group communication system. Invoking another service results in also calling the replication layer of the stub associated with that service. Before generating the message ID of the request, it is checked whether there is a context associated with the current state. If that is the case, the new message ID is the message ID in the context plus the service ID plus a consecutive number. This ensures that all replicas generate the same message ID. In addition the SLT scheduler can exploit this way of constructing message IDs to recognise recursions. To enable the scheduler to perform optimisations, the scheduler is informed before the nested invocation and also when the result is received

4.5 Future Work

For the future we target several extensions to the current system. First of all, we target to support configuration changes at runtime. That is, we want to be able to change e.g. the replication protocol dynamically. Furthermore, we would like to provide replicas with a memory. In case of large states starting with a void state requires long state transfer. Allowing new replicas to re-use the persistent state of replicas having previously run on the same host can potentially reduce state transfer times as only updates have to be transmitted. In addition, we are looking for a way to realise the state transfer stream-based instead of block-based as the memory footprint is enormous in the second case when the state is large.

The current implementation is inherently insecure. Firstly, due to the fact that all clients are allowed to call administration methods. Secondly, the communication between clients and server as well as the inter-replica communication are plain text so that it is easily possible to sniff as well as to manipulate the data transmitted. Third, the replication protocols rely on a crash-stop behaviour of the replicas. We have not yet considered a non-benign behaviour of replicas which is supposed to happen in a real-life, wide-spread set-up of Virtual Nodes. We are planning to face all of these security threats in future versions of the software.

Earlier work [11, 3] has shown that some scenarios require explicit support for client-side code, e.g., for accessing client-side resources, encapsulated behind the service interface. We will provide such a functionality in a later version of Virtual Nodes.

Finally, dynamic proxies, which we use to realise the Java RMI surface, have turned out to be performance bottlenecks. This is especially true for local invocations. For that reason we consider generating proxies offline.

5 Programming Issues

Warning: This section describes a part of the software that is still work in progress

In this section we have a look at how to program Virtual Nodes. In addition we summarise the configuration options that have appeared throughout the text and we show reasonable combinations thereof.

5.1 Providing Virtual Nodes

In order to implement Virtual Nodes several things have to be taken into account. Regarding the way how the service has to be implemented we distinguish the algorithmic (which kind of operations are allowed, ...) and the technical (which information has to be provided, where to put it) point of view.

Algorithmic Considerations

From the algorithmic perspective we cannot give general guide how to implement an application ready for replicated execution, as the details depend on the replication protocol and its parametrisation. One basic issue for consistency is that the recoverability property is satisfied. Revcoverability in the context of object-oriented remote methods means that no reply returned to the client will be invalidated later on due to some events in the system. This property is enforced by all the scheduling algorithms presented in Section 3, if the following restriction is respected in programming Virtual Node applications:

- **Do not create threads!**
  As multithreading can destroy determinism even in passive replication a thread created in an arbitrary situation without the knowledge of the replication framework will cause harm. The only exception are threads for pure computation purposes whose input parameters have been determined before their creation and that their entire interaction users or other threads is restricted to returning their result to the creator thread. Apart from that there may be other cases with allow the creation of threads. However, such a decision has to be taken as the case arises.

For actively replicated set-ups or systems using passive replication with request logging we require this additional restriction:

- **Do only use deterministic operations!**
  Active replication is built on the assumption that all operations produce the same output, if provided with the same input. In passive replication the re-execution of a request due to the failure of the leader replica must not produce another output than the first execution.

If single-threaded execution is used, deadlocks can only be prevented, if the following constraints are respected:

- **Do not use condition variables!**
  Calling `wait` with a single thread will result in a deadlock.

- **Beware of cross invocations of services!**
  If a service A calls another service B while B also calls A, then both threads will end up blocked.

The Java libraries are per se not replication aware and might even be nondeterministic in some places. This is especially true when library classes use `wait/notify` or the language specification is unprecise or leaves implementation details intentionally open. One example therefore is `Object.hash()` which in Sun's reference implementation returns the memory address of that particular object. Another example is the method `Thread.getId()`; a thread ID is only required to be unique for the lifetime of a thread which is in general not sufficient to guarantee determinism.

Another critical issue are operations such as writing to a file. In the general case, this operation will be successful on all replicas and therewith deterministic. In some cases, however, it may fail on some replicas and become a nondeterministic operation. The only option to get back to a deterministic state is to kill the replicas for which the operation has failed (or for which the operation was successful). Nevertheless, a replica experiencing the failure is not allowed to kill

`@readonly`: no changes to the service state

Figure 8: Annotation

itself without knowing the results of the other replicas. Because, if the operation had failed on all of them, it would be deterministic again, and regular operation might continue. So far, the framework does not offer support for those issues.

Technical Considerations

The development process of the service is very similar to the development process of Java RMI objects. With small exceptions: At the first, the objects representing the service can have any type and are not forced to implement the `java.rmi.Remote` interface. However, it is required that the service object implement the `java.io.Serializable` interface or an interface extending it in order to be able to transfer the current state to a newly created replica. Recall the method from Figure 6 showing the interface to the method initialising the first replica. The interface of the stub is the transitive closure of methods in the interfaces implemented by the service object. The methods in the interfaces can be annotated as `@readonly`. In consequence the method will be executed with relaxed consistency guarantees. Where the annotation happens does not matter. It is possible to annotate the method inside the service implementation, or in the interface defining it. It is also possible to annotate an interface as a whole meaning that all of the methods in that interface are annotated.

The second change in comparison to the Java RMI development process is an additional code transformation step. We provide a script that searches for `synchronized` statements and replaces them with calls the scheduler's `lock` and `unlock` method. This also changes the classes, as it adds an additional field representing the scheduler. Only afterwards the code is ready to be compiled.

If the service implementation contains nondeterministic operations those can be treated as nested invocations. For that purpose we will provide an interface in the first prototype that helps the programmer to handle those kinds of operations. The interface is not specified yet, but it will be part of the context attached to the threads which process the requests. We will update this document once the specification is finished. However, using this hook destroys all replication transparency and turns the code unusable in a replication-unaware environment. In the future we plan to provide additional means for transparently handling those kinds of methods.

```
    while(true) {
      gstat = pas.update( playerNr, playerPos,
                          System.currentTimeMillis() );


      ...

      repaint();

      Thread.sleep(Config.sleeptime);
    }
```

Figure 9: PongAlong Client Code

5.2 Virtual Nodes without Java RMI

The layered architecture of the Virtual Node system allows to also use them when the application itself does not make use of Java RMI. The only precondition that has to be met is that the application works based on request-response interaction with its clients. In the most basic use case the existing communication layer (the part at client and server side responsible for communication) has to be changed in order to send all data to the replication framework instead of the network. Furthermore, apart from the one new thread that is created by the replication framework per request, in general no other new threads are allowed at server-side.

In case the server application uses one or more daemon threads, their integration in the system has to be done manually and can only be realised after having carefully investigated if doing so remains consistency guarantees. Depending on the application architecture and semantics the integration also fail.

5.3 Configuring Virtual Nodes

Warning: This section describes a part of the software that is still work in progress

Currently, the system can only be configured at pre-compile time by instantiating objects of the respective classes. We will update this section while we add means for a configuration at startup time.

6 Example Application

In this section we discuss a small example in order to illustrate the programming with Virtual Nodes. More detailed explanations are subject to a *Virtual Node Programmer's Guide* paper that is to be published before the first XtreemOS release.

```
public interface PAServer extends Remote {
  public GameStatus update(
                int playerNr, int pos,
                long timeStamp) throws ...
}
```

Figure 10: PongAlong Interface

```
String[] s = new String[3];
s[0] = config;
PAServerImpl server = new PAServerImpl();
PAServer stub =
     (PAServer) vnode.rmi.Stub.startup(
                              s, server);
```

Figure 11: PongAlong Server Instantiation

The example application `PongAlong` is a fault-tolerant version of tha Atari Pong [3] game. At run-time the set-up includes up to four players (clients) and a set of server nodes which are replicas of each other. Furthermore both clients and servers have to have sufficiently synchronized clocks, i.e., via NTP. The interaction is client-triggered: clients regularly invoke methods a the servers passing the position of their respective *racket* and their current system time t as parameters. The return value to this invocation is a set of values including the position of the ball at t and the other players' racket position. The client-side code is shown in Figure 9 (exception handling code omitted).

In order to implement such a scenario one interface with a single method `update` is required (see Figure 10). As the method changes the server state, it cannot be annotated `@readonly`. The server implementation `implements` the `PAServer` interface. The initialisation of the first replica is show in Figure 11: first the `String` array is created; afterwards, the server implementation. Finally, both are passed to the `startup` method that returns a stub which may for instance be registered at a Java name service.

7 Virtual Nods in XtreemOS

Although the Virtual Node framework mainly addresses application developers, it might also be of use for other workpackages in the XtreemOS project whose services require high availability and cannot tolerate downtime. Entities we con-

[3]http://en.wikipedia.org/wiki/Pong

sider as candidates are parts of the application execution management (WP3.3), in particular the XOSD, the XtreemFS metadata server (WP3.4), as well as central entities of the security infrastructure (WP3.5), such as Credential Distribution Authority and the Policy Service. The question which services Virtual Nodes shall be used for will be subject to the integration process.

On the other hand the Virtual Node framework might also make use of the functionality of services provided by other XtreemOS entities. For example the part of the Application Execution Management bundle concerned with ressource management and selection could play an important rule in automatising the Virtual Node administration. Again, the decisions whether this approach is feasible and how realise it, are subject to the integration process.

Finally, Virtual Nodes provide fault-tolerance by replication. However, they require that software be installed at client-side. Thus replication is not fully transparent to the users. In order to achieve full transparency we plan to merge them with the Distributed Server infrastructure described in Deliverable 3.2.2.

8 Conclusion

In this document we have presented the XtreemOS Virtual Node framework. We have mainly focused on the architecture showing how components at both client- and server-side collaborate in order to provide fault-tolerance by object replication. We have presented the individual configuration options for replication protocol, application-level scheduler, caching, and group communication system. In addition, we have sketched the runtime behaviour of the entire system including joining and failing replicas as well as nested invocations. Furthermore, the deliverable contains a discussion of potential future work and a small example showing how to programm Virtual Nodes. An in-depth discussion of programming will be subject to another docuement, the *Virtual Node Programmer's Guide*.

References

[1] C. Basile, Z. Kalbarczyk, and R. K. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 149–158. IEEE Computer Society, 2003.

[2] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose synchronization of multithreaded replicas. In *SRDS '02: Proceedings of the 21st IEEE Sym-*

posium on Reliable Distributed Systems (SRDS'02), page 250, Washington, DC, USA, 2002. IEEE Computer Society.

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

[4] X. Défago and A. Schiper. Specification of Replication Techniques, Semi-Passive Replication and Lazyt Consensus. Technical report, 2002.

[5] J. Domaschka, F. J. Hauck, H. P. Reiser, and R. Kapitza. Deterministic multithreading for java-based replicated objects. In *Proc. of the 18th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'06, Dallas, Texas, Nov 13-15, 2006)*, pages 516–521, 2006.

[6] J. Domaschka, H. P. Reiser, and F. J. Hauck. Towards generic and middleware-independent support for replicated, distributed objects. In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 43–48, New York, NY, USA, 2007. ACM Press.

[7] 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.

[8] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, Berlin, 1st edition, 2006.

[9] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded corba applications. In *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, page 263, Washington, DC, USA, 1999. IEEE Computer Society.

[10] H. P. Reiser, F. J. Hauck, J. Domaschka, R. Kapitza, and W. Schröder-Preikschat. Consistent replication of multithreaded distributed objects. In *SRDS '06: Proceedings of the 25st IEEE Symposium on Reliable Distributed Systems*, pages 257–266, 2006.

[11] H. P. Reiser, R. Kapitza, J. Domaschka, and F. J. Hauck. Fault-tolerant replication based on fragmented objects. In *Proc. of the 6th IFIP WG 6.1 Int. Conf. on Distributed Applications and Interoperable Systems - DAIS 2006 (Bologna, Italy, June 14-16, 2006)*, 2006.

[12] D. Stutzbach and R. Rejaie. Towards a better understanding of churn in peer-to-peer networks. Technical Report UO-CIS-TR-04-06, Department of Computer Science, University of Oregon, November 2004.

[13] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274. IEEE Computer Society Technical Commitee on Distributed Processing, 2000.

[14] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Deterministic scheduling for multithreaded replicas. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 74–81, Washington, DC, USA, 2005. IEEE Computer Society.