Project no. IST-033576

XtreemOS

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

Extended Version of a Virtual Node System
D3.2.14

Due date of deliverable: November 30^{th}, 2009
Actual submission date: December 04^{th}, 2009

Start date of project: June 1^{st} 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
Universität Ulm
James-Franck-Ring O-27
89069 Ulm
Germany

Version 1.0 / Last edited by Jörg Domaschka / December 03^{rd}, 2009

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
0.1	31/10/09	Jörg Domaschka	ULM	first draft
0.2	02/11/09	Christian Spann	ULM	added section about model checking
0.3	02/11/09	Jacobo Giralt	BSC	SEDA/AEM overview
0.5	30/11/09	Martin Pfeil	ULM	Evaluation results
0.6	01/12/09	Jörg Domaschka	ULM	Comments reviewer #1
0.7	01/12/09	Jörg Domaschka	ULM	Comments reviewer #2
0.8	01/12/09	Jörg Domaschka	ULM	Stripped down evaluation section
0.9	02/12/09	Jörg Domaschka	ULM	final polishing

Reviewers:
Ales Cernivec (XLAB) and Benjamin Aziz (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 focuses on three core themes. Firstly, it presents the usability of the Virtual Nodes framework by discussion a demo application. Furthermore, it contains an extensive evaluation of the framework with respect to performance and availability.

Secondly, this document is concerned with our efforts on integrating Virtual Nodes with other components of XtreemOS, mainly AEM and Distributed Servers. For each of both components, we present the architectural approach, the current state regarding the implementation and a list of future work.

Thirdly, we made attempts to improve the quality of our code. We did modularise our previously monolithic framework implementation in order to allow maintaining components individually. Moreover, we have used model checkers to prove that our various scheduler implementations are correct.

1 Introduction

After the first specification in D3.2.5 [4] and a functional evaluation in D3.2.9 [8] this deliverable is concerned with an extension of the Virtual Nodes framework. The last months were characterized by various real-world and integration efforts. In particular, we presented a replicated POP3 server at the XtreemOS review meeting (c.f. Section 2).

In addition we have also expended effort for integrating Virtual Nodes with distributed servers whose feasibility has been subject to Deliverable D3.2.10 [7]. As before, this is ongoing work, so that we only present architectural results mainly from a Virtual Nodes perspective.

Furthermore, we have been working on applying the framework to the Application Execution Environment (c.f. Section 4). This is still work in progress, so that this document will only present preliminary results. Mainly, it focuses on the software architecture of the integration approach. Moreover, it will only focus on the extensions required for Virtual Nodes and ignore modifications to AEM.

Apart from that we worked on proving the correctness of our deterministic scheduling algorithms using the Spin model checker [10]. The modifications and extensions led to the introduction of a new modularised software layout of Virtual Nodes together with a sophisticated configurator.

Finally, we have executed an extensive performance and reliability evaluation of Virtual Nodes. In the following sections we will discuss these topics.

2 Demo

For the review session in June 2009, ULM presented a fault-tolerant POP3 server based on the Virtual Nodes framework. This section first gives a short introduction about the effort it took to take a ready-to-use open source POP3-server implementation and to make it fault-tolerant. Afterwards, we discuss the demo set-up and discuss its weaknesses.

2.1 Preparation

As a basis we used the open-source POP3-capable *Java Mail Server*[1]. We modified the implementation to fit the Virtual Nodes requirements. Overall, it took us one person week to do all modifications and to get the software running reliably. Figure 1 shows a diagram that summarises the time spent on different

[1] http://www.ericdaugherty.com/java/mailserver/

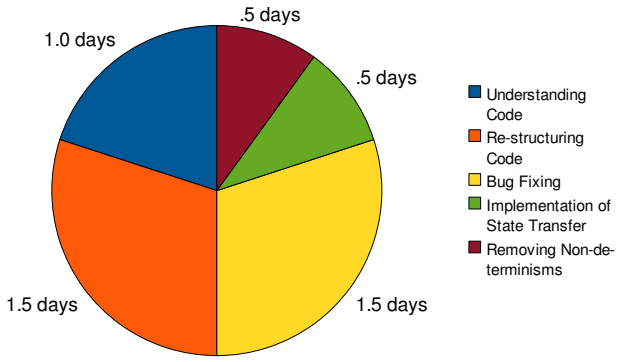


Figure 1: Time required for porting JMS to Virtual Nodes [days]

tasks. It clearly shows that the actual tasks that are required for enabling replication, namely removing non-determinisms and adding support for state transfer, together were less time-consuming than all other tasks individually.

Adding state transfer took half a day and resulted in additional 200 lines of code that are mainly responsible for copying files. Removing non-determinsm required adding about 100 lines code that mainly handle access to the file system. For instance, it ensures that invocations to, `File.list()` return a deterministically ordered list of files instead of a randomly ordered list (as specified in the Java API).

In contrast, it took three days to first restructure the poorly written code and to track and remove bugs. The code restructuring was necessary as Virtual Nodes require object-orientation. That means, they operate on object instances and cannot deal with `static` methods and code that has been developed following the functional programming paradigm. Furthermore, even though the author claims that the code is multi-threading enabled, we found out that this is not entirely true. The author dabbled at applying synchronisation and mutual exclusion. Due to the idempotence of most of the POP3 operations those errors barely show up in a non-replicated environment. In a replicated scenario where a client can connect to any replica, those may become an issue.

2.2 Set-up

For the demo session we used the set-up shown in Figure 2. Two replicas were located on `hykrion` and `hynreck` in ULM. In addition, we had set-up other entities running on `qvin` – also located in ULM. `MessageGenerator` is a tool built by ULM that generates e-mails at a configurable rate and injects them in the

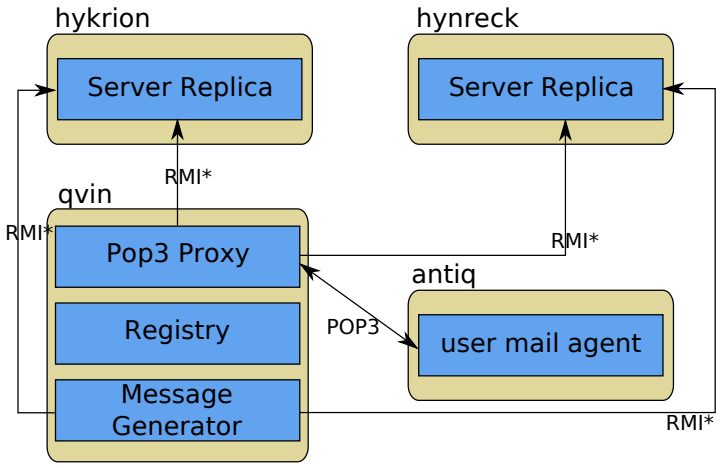


Figure 2: Set-up of demo application

replicas using the Virtual Nodes protocol (denoted by *RMI**). `Registry` is an entity similar to a Java RMI registry. Here, the replicas store their contact information. Clients use it to retrieve this information and thus to contact the replicas. Finally, `Pop3Proxy` is an entity that translates incoming POP3 messages to remote method operations to the replicas resulting in messages in Virtual Nodes format. In addition, a mail client (Thunderbird) resided on a laptop in Rennes and contacted `Pop3Proxy` in order to retrieve mails from the server group.

The goal of the demo was to show Virtual Nodes are working, so it is not surprising that the set-up is not fully fault-tolerant. Clearly, the replica state located on hykrion and hynreck is replicated and thus protected against failures of individual replicas. All entities on qvin, however, are not fault-tolerant as they are not replicated. That means, if qvin fails, the mail service becomes unavailable. In order to make the entire set-up fault-tolerant, one might apply multiple other XtreemOS entities. First of all, replacing the registry with the XtreemOS directory service, allows to have a stable instance that replicas as well as clients can use to store and retrieve contact information. For the proxy to become fault-tolerant, it is possible to apply Distributed Servers [3] and allow each replica to have its own proxy instance, as discussed in Section 5. Note, that it is not required to provide resilience for `MessageGenerator`, as this entity represents a sending client whose fault-tolerance can never be guaranteed. The same holds for the mail agent.

3 Modularisation

Virtual Nodes is highly configurable. This leads to the fact that parts of the system are entirely independent of other parts. Thus, for re-using individual components not all of the Virtual Nodes code is required. Consequently, we decided to modularise the code. As a result of those modularisation efforts, Virtual Nodes now consist of a set of around 40 modules. In most cases each component is mapped to an interface and an implementation module. In case of schedulers and replication strategies there are multiple different implementation modules. In order to allow exchanging existing or adding new implementations easily, implementation modules purely depend on interface modules. As well as interface modules depend only on other interface modules.

Apart from that, individual modules now (together with their transitive closure of dependencies) can be used as stand-alone components such as the group communication abstraction or the schedulers. In particular such a set-up permits that components be tested individually, but nevertheless be started by the standard Virtual Nodes mechanisms.

Loading and initialising modules is realised by a new configuration infrastructure that is implemented as an own software module. Similar to the OSGi framework, the configurator requires that each module comes with an `Activator`. An activator specifies properties it depends on and properties it provides. In addition, an activator can initiate that other activators be loaded. In particular this allows activators of interface modules to specify which implementation shall be used. Depending on provided and required properties, the configurator loads the activators in the correct order or aborts initialisation if dependencies cannot be satisfied.

Our efforts of modularising our system have already proven to be useful, as extensions for both AEM and Distributed Server integration can be implemented as mostly stand-alone modules without interfering with Virtual Nodes core code.

4 AEM Integration

In an XtreemOS system there are two kinds of nodes: core nodes and resource nodes, differ on the kind of services they run. Basically, a resource node is just a server where jobs are run, while a core node manages higher level concepts such as *job*, *reservation*, *dependence* and *user metric*.

From the fault tolerance perspective, resource node services are not worth being replicated. They store data that is exclusively relevant to the node they run on. If the node fails, and thus the job does as well, this data is of no use for any other node. Thus, replicating its data will not represent any added value. On the

other hand, if a core node fails, the jobs it managed should not be affected. User requests for jobs that are managed by a failed core node can be also handled by another core node if it has a replica of the job's state.

Summarising, the following core node entities may be subject to replication: Job Manager, Reservations Manager and Checkpoint/Restore Job Manager. In a first proof-of-concept approach we will focus on replicating the Job Manager.

In the following subsection we will first present a short overview on AEM. Then we discuss the DIXI communication infrastructure. AEM is based on DIXI so that an integration requires an adapter that wires Virtual Nodes to DIXI. Finally, we conclude with a report on the current status and future tasks to be carried out.

4.1 AEM System Architecture

The AEM architecture is based on the Staged Event Driven Architecture (SEDA [14]) first published by Matt Welsh and presented in Deliverable D3.3.3-4 [5, 6]. From a simplified point of view, services in this architecture are asynchronous event machines and communication consists of event/message passing. A system consists of a stack of stages. Each stack has an incoming and an outgoing queue. The incoming queue of stage s_i contains messages that have arrived from stage s_{i-1}, the stage below s_i in the stack. Messages in the outgoing queue are put there by stage s_{i+1}. External messages enter the system at the bottom-most stage. Internal messages are created at any stage and can be used for inter-stage communication. When a message enters a stage from the lower (upper) layer, it is processed according to the stage policies. Afterwards, it is passed up (down) one stage if required. In order to ensure isolation and re-usability of stages, each stage comes with its own concurrency policy; that is, its own thread pool. A thread assigned to a stage never leaves this stage, but only uses the queues for communication.

Every service in this architecture extends the `Abstract2WayStage` class, which is the top stage of the SEDA stack. Below this stage, there are the Message Bus and the Communication Stage that build on the MINA[2] communication framework. Services and stages are initialized by the XtreemOS Daemon, as represented in Figure 3. Stages' event queues are grouped in the `EventMachine` root object.

The Message Bus Stage[3] provides communication capabilities between services and stages. It parses message headers and sends events to the corresponding service.

The Communication Stage[4] uses Apache Software Foundation's MINA li-

[2] http://mina.apache.org/

[3] eu.xtreemos.system.communication.bus

[4] eu.xtreemos.system.communication.net

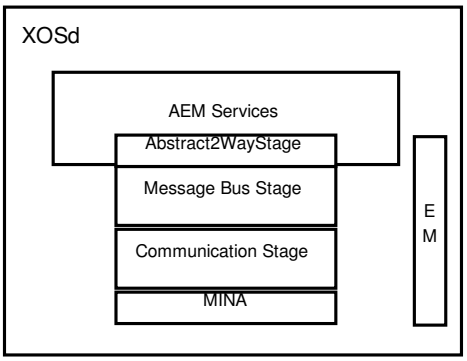


Figure 3: A view of the SEDA stack inside the XOS Daemon

brary as network framework. The XtreemOS Daemon creates a thread to listen to the selected port for incoming event messages. Those messages are serialized `ServiceMessage` objects.

4.2 Communication Stack

Overview

Recall from Figure 3 that the communication happens according to the communication stack. Messages come into the system and are passed up the SEDA stack as events until they are either dropped or reach a service that is able to process them. The messages that pass through `CommunicationStage` are of type `ServiceMessage` whose implementation does directly represent the fact that messages are supposed to carry information of a remote method call.

Virtual Nodes use two kinds of communication systems. The group communication system (GCS) is used for internal, i.e. intra-replica communication, whereas the clients use some external communication interfaces to access the Virtual Node. This infrastructure is depicted in Figure 4. The external communication interfaces do also serve as a means to access other, i.e. remote, services from within a replica of a Virtual Node.

For an integrated approach it is necessary to have all messages that are targeted to a replicated entity (say a `JobMng`) be received by its corresponding Virtual Nodes instance. Subsequently, this instance hands the message over to the replication protocol, which, finally, hands it over to the replicated entity. More clearly, messages/events must not arrive at, for instance, the `JobMng` except when the Vir-

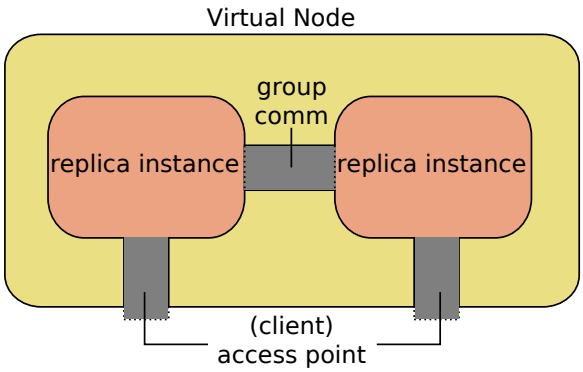


Figure 4: Communication Facilities of a Virtual Node

tual Nodes framework has handed over the message. The same holds for replies and invocations to other services: any message directed to a replicated service must pass the Virtual Nodes framework. Accordingly, all interaction of a service has to be intercepted. In addition, the fact that a request issued by a client might be directed towards a crashing (crashed) replica, requires to re-execute requests. This is only possible if messages can be identified uniquely and thus trackable starting from the first client. Furthermore, it is required to map nested invocations to their original request which results in another id to be added to messages. As we try to minimise changes to existing code, we do not directly add these ids at client-side, but use a level of indirection.

Figure 5 sketches the communication stack that is used in replicated scenarios. We introduced an additional stage, the `ReplicationStage`. Messages that are not directed towards replicated services just pass through this stage without any modifications. For messages directed to replicated services there are two options. If the message is issued by a client, and thus not yet replication aware, it is passed on to a replica stub stage where the message is modified to fulfill replication requirements (e.g., ids are added) and then relayed to the replicas. Of course this approach is only resilient to node failures when only requests from local clients are proxied. This is ensured by the way an AEM-extension treats addresses of replica groups. The extension is subject to work package 3.3, so that we will not discuss this issue in detail here. Requests from remote hosts are directly forwarded to their respective vnode stage each containing an instance of a Virtual Node as well as a replica. Accessing Virtual Nodes from the replication stage requires some extensions to Virtual Nodes that are discussed in the next section.

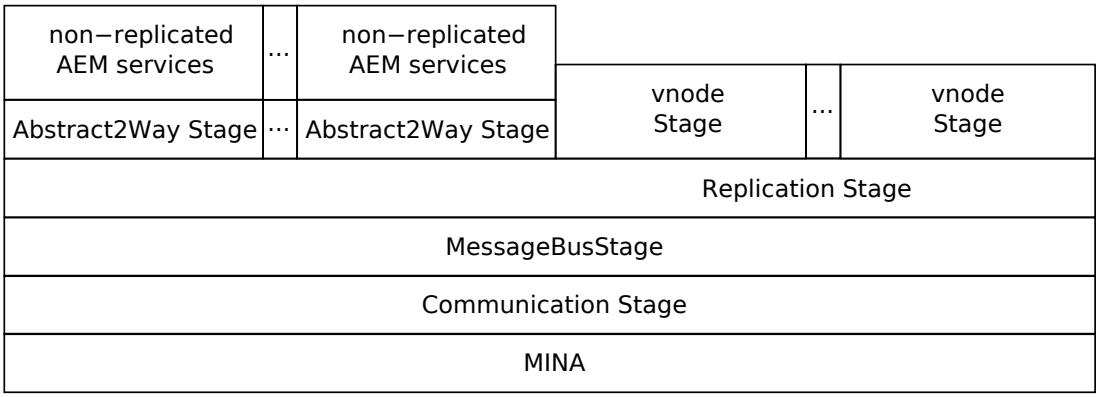


Figure 5: Modified Communication Stack for Replication Support

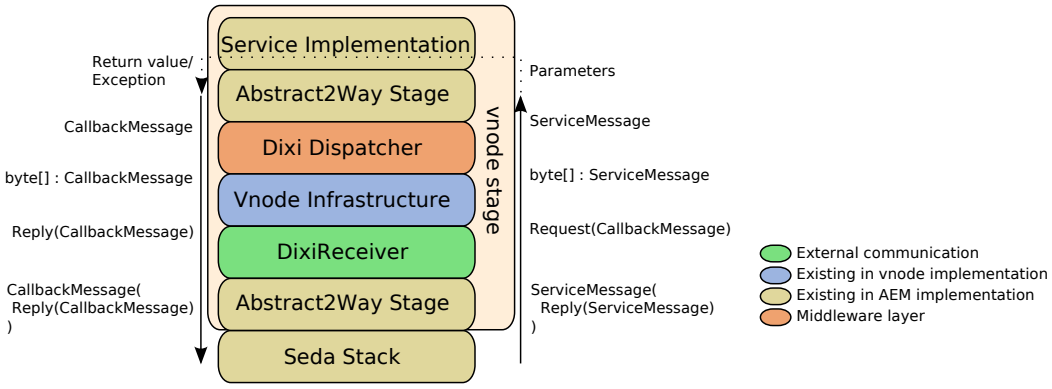


Figure 6: Message and data flow in an integrated version on server-side

Virtual Nodes Perspective

From the perspective of the Virtual Nodes architecture two custom extensions are required in order to be able to deal with the DIXI stack, namely the external communication layer and the middleware layer. Conceptually, those two layers are unrelated. The external communication layer is concerned with receiving messages, while the middleware layer deals with parameter (un-)marshalling. However, as in DIXI messages and parameters are tightly coupled, both entities will share functionality for handling DIXI messages.

The flow of messages and data is shown in Figure 6. As one can see, messages arrive at the vnode stage using the regular DIXI stack. Here, they are not directly handed over to the service implementation, but have to pass the replication infrastructure. For that reason the Virtual Nodes framework uses its own

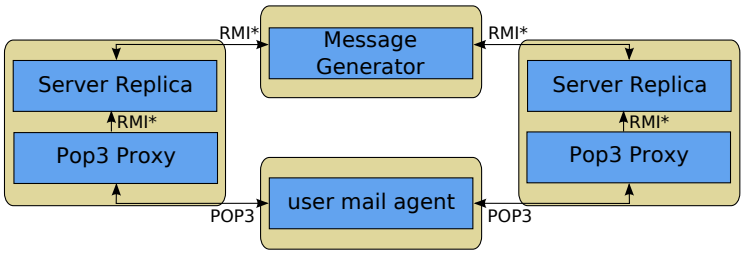


Figure 7: Possible demo set-up for an integrated approach

`Abstract2WayStage` where messages can be enqueued. That is the external communication layer implemented for DIXI. Here, the message is wrapped into a Virtual Nodes message and relayed to the replication protocol. From there, it is relayed to the middleware layer (`DixiDispatcher`) where the DIXI message is unwrapped and then inserted in the in-queue of the service implementation. The reply uses the same route in inverted direction.

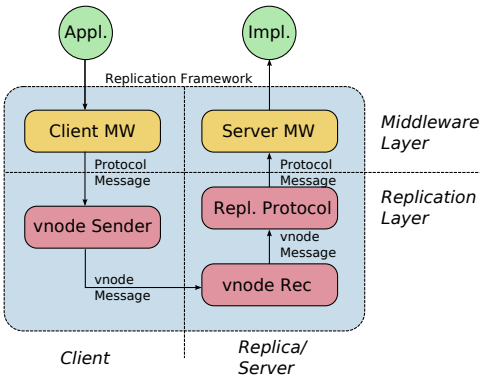
In addition, `DixiDispatcher` is registered as the services' out-queue. This allows to intercept messages in the service implementation issues a nested invocation, and to let them be processed by the regular Virtual Nodes mechanisms.

4.3 Current State

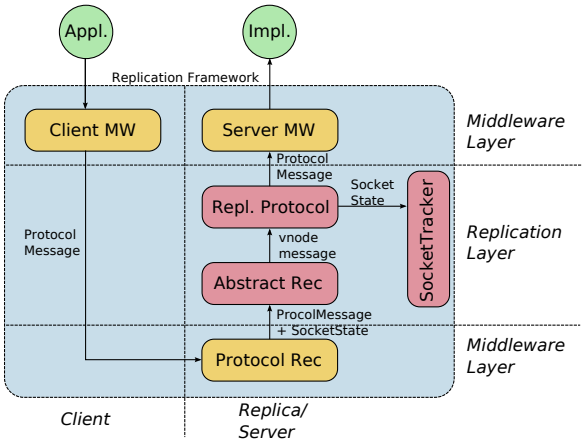
At the time of writing this, the integration is as far as supporting the replication of individual self-contained DIXI services. Services are self-contained if they do neither depend on other service nor access the file system. Adding support for services interacting with each other is considered future work and will be supported by the end of the project.

5 Distributed Server Integration

In a previous deliverable discussing the integration of Virtual Nodes with distributed servers [3], we agreed that one of the replicas functions as a contact node and receives all incoming requests. We further stated that active replication is the only replication scheme that makes sense, as providing fault-tolerant distributed servers means replicating socket state. Nevertheless, from a Virtual Nodes perspective there is no need to distinguish both protocols. Our approach uses already existing mechanisms within the replication protocols, so that it is totally agnostic regarding the replication protocol. Nevertheless, some modifications and extensions to the workflow are required.



(a) In Orginal Version



(b) With Protocol-specific Receiver using Distributed Servers

Figure 8: Invocation Flow

In order to allow Virtual Nodes to use distributed server functionality slight changes to the architecture where necessary. One of the most outstanding features of an integrated version is that clients do not require Virtual Node-specific code anymore. Instead they can use an off-the-shelf middleware system to access the replicated service. Such a middleware system might be a regular RMI stub, a SOAP interface or any specified protocol. Figure 7 sketches how the demo scenario from Section 2 may look in an integrated version. Now, each replica is linked to its own (basically state-less) proxy instance that translates messages to Virtual Nodes messages. It is worth noting that the client is unaware of the fact that there are more than one node processing its requests. Location and replication are kept transparent by the underlying mobile IPv6 protocol. Thus, the client does not have to rely on a registry anymore. Instead, it uses a well known IPv6 address.

Replicas only understand Virtual Nodes messages. Thus, we have to add an adapter-like entity that is able to transform protocol-specific messages to Virtual Nodes-messages. Figure 8(a) shows the previous layout and workflow, Figure 8(b) sketches the current approach. As one can see, the Virtual Node-specific receiver has been replaced by a protocol-specific receiver whose input is piped into an abstract Virtual Nodes receiver.

It is crucial to keep track of open sockets on other nodes. As soon as a replica fails, another replica takes over all its open sockets and re-executes the requests respecting at-most-once semantics, so that it is able to send a reply to the waiting client. Taking over a socket means that a serialised socket state is installed in the operating system, gets initialised, and the client connection is taken over following the mobile IPv6 protocol. We decided to let the oldest replica take over all open sockets of the failed replica. Note that it is indeed possible that multiple replicas have open connections to clients. This may happen, because distributed servers come with their own group management functionality including a failure detector, which decides which group member is the current contact node. As discussed previously, the task of this failure detector is to minimise the down-time experienced by clients. Thus, it uses a very short timeout that may result in many false positives and in consequence in many relatively light-weight contact node switches. In contrast the Virtual Nodes failure detector will use much higher timeouts in order to avoid false positives, because on this level, they result in a member being excluded from the group. Excluding a member in turn means that this member's state is not kept consistent with the other replicas' respective state anymore. Consequently, diagnosing a member as failed requires to initiate a new state transfer in order to let the falsely excluded machine re-join the group again.

The whole functionality of socket recovery and re-execution of requests is implemented in a module called `SocketTracker` that is brought up by the protocol-specific receiver. Accordingly, the protocol-specific receiver serialises the socket state after a complete message has been received (where the meaning of *complete* depends on the protocol being used). The serialised socket is appended to the message and passed on to the replication protocol. In order for the `SocketTracker` to get the required information, we introduce a `MessageFilterRegistry` that is invoked before a request is actually being executed (i.e. is inserted in the scheduler). Yet, in order to stay generic `MessageFilterRegistry` is not tied to `SocketTracker`, but introduces a specific interface `MessageFilter`. Instances of `MessageFilter` can be installed at the registry. `SocketTracker` implements this interface.

Summarising, the new workflow for processing a request is as follows. New steps are marked with *.

1. The client sends a request to the contact node via mobile IP6 using a regular

protocol such as SOAP, RMI, or POP3.

2. The contact node freezes the state of the socket that the request was received through and appends it to the request.*

3. The modified request is handed on to the replication protocol where it is made persistent.

   - For active replication this means that the request is broadcast to all other nodes where it is processed.
   - For passive replication this means that either the request or its effect is made persistent by the persistency layer.
   - For passive replication this means that the request or its effect is made persistent by the persistency layer.
     - In case of requests, no additional work has to be done, because the request already contains the serialised socket.
     - Application states, do not include any information about socket states, as the application does not know about sockets. Thus, the passive replication protocol has to ensure that the `MessageFilterRegistry` including all information about socket states is serialised together with the application state.*

4. After having processed the request, the contact node sends the reply to the client.

Discussion

The protocol does allow that open (in fact any) client-side protocols are used as long as connections are not multiplexed. That is, a second requests is not sent concurrently to another request using the same connection (i.e. socket). For now two requests are considered concurrent to each other if the first byte of the second request is sent while the last byte of the reply to the previous request has not yet been received. A request is previous to another request when the first requests first byte is sent before the second requests first byte.

The protocol ensures that requests are processed in a fault-tolerant manner. Yet, it does not ensure full failure-transparency. That means clients can experience a communication error (such as a connection timeout). A disconnect happens when the contact replica crashes while receiving the request before it is made persistent by the replication protocol. During that period the system lacks transparency for clients. As no state modifications are issued while a request is being received, there is no risk of inconsistent replica states. Furthermore, a client may

experience a connection timeout when the detection of a node failure takes longer than e.g. a TCP-connection needs to timeout. Configuring the timeout appropriately, is an administration task and cannot be handled by the framework.

5.1 Current State

At the time of writing this, we have finished the implementation of `SocketTracker`. We are currently realising a Distributed Server-aware protocol-specific receiver for POP3. In a next step, we will implement `MessageFilterRegistry`.

6 Verification of Scheduling Algorithms

One core feature of Virtual Nodes in order to minimize the performance loss of replication is to allow multi-threading. But there is one invariant that must hold to be able to replicate a service while allowing this feature: the determinism of the scheduler. It must be guaranteed at all times, that the critical sections of the code are accessed in the same sequence on all replicas. Only this way the state on all replica goes through the same sequence of intermediate states and stays consistent throughout the execution.

For this task we provide multiple scheduling algorithms that can be combined with the active and passive replication schemes. It is relatively easy to prove the correctness of SEQ and SAT [8]. These two algorithms execute the requests sequentially and only differ in their ability to allow wait/notify semantics. They do not allow real multi-threading, i.e., there is only a single active thread at a time. Regarding the algorithms that allow real multi-threading such as LSA (Loose Synchronisation Algorithm) [2] , PDS (Preemptive Determinstic Scheduling) [1] and MAT (Multiple Active Threads) [13], proving correctness is not trivial anymore. So the authors of LSA and PDS used a logical proof in the proposing papers for the algorithms to check the determinism of the algorithm and beside that – model checking.

There are two correctness properties that must be verified to prove the correctness of a distributed algorithm. The internal one regarding the correct behavior of the algorithm on one single replica and the external one regarding the overall behavior of multiple replicas. Basile et al. [1] only checked the internal behaviour of their two algorithms, because it is not easy to construct a distributed model of such an algorithm with a finite state space which is required for model checking.

The Spin Model Checker [10] was used by the authors of LSA and PDS to prove internal correctness and so we decided to use that tool as well to test the MAT scheduling algorithm which has not yet been checked with a model checker.

As we wanted to have something to compare MAT against regarding the state-space, we did a re-implementation of PDS whose original implementation is kept private by its authors. Checking an algorithm with Spin requires to design a model of that algorithm with a predefined limited set of commands. The so called *Process Meta Language (PROMELA)*[5] is a C-like language with support for concurrent processes and bounded communication channels between processes. The restriction on a few language constructs results from the way the model-checker works. It simply checks every possible state in the interleaving of the different processes for a violation of formerly defined invariants. As a result, all elements used have to be bounded with respect to the number of states they can be in. Thus, unbounded elements (e.g. floats, random number generators, unbounded channels) do not appear in the model. Moreover, there are not even methods, but only a slightly advanced support for macros. Consequently, the design of an easy to read code is not easy.

As expected both algorithms passed the internal correctness tests. But the challenge to prove external correctness still remained. It came to our mind that, if there is only one possible access order to a critical part of the code, then the external correctness holds for any number of replicas. We do not have to take care of the communication layer here, because we use a group communication framework that guarantees the ordered delivery of all messages to all nodes (atomic broadcast semantics). During the verification we continually build the access sequence to the exclusive parts of the code. This sequence is not modified when the model checker jumps back and forth in the tree that spans all possible executions paths. We had to use a memory access mechanics out of the control of the model checker to be able to do that. Normally, when the model checker has finished the search in one path and heads back to test other subbranches, it would revert all changes to any variable it had made during the execution of the former subbranch. This is inevitably necessary for normal execution testing, as otherwise the tests would prove nothing. Normally one would hard-code the invariants into the model checker using *Linear Temporal Logic (LTL)* and afterwards run the model against them. For that approach we would have to compute the sequence of access before the run, which would be very difficult to do for more complex scenarios. And still this state space can grow really big without much effort. We tested PDS and MAT with 10 parallel requests and 5 to 10 mutex variables for critical code access and reached 3-6 million states and the memory bounds of our 4 GB RAM test machine.

We only checked simple lock/unlock tests, because wait/notify and nested invocations itself take place inside a guarded codeblock. So if the exclusive access is guaranteed in this simple case, all other properties will hold, too. As stated

[5] http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html

above we tested the algorithms with up to 10 parallel running requests that compete for 10 synchronisation variables. This is sufficient because of two reasons. You do not need more than n threads running on a machine, when you have n cores (except with heavy I/O). So, there is no need for a model-checked proof of infinitely many parallel requests. Also one has to keep in mind that the algorithm has been proven logically for that case. And coming from that direction, if two requests competing for one synchronisation monitor cannot break determinism, one can prove inductively that this holds for $n + 1$ threads and mutexes as well.

A second approach was to directly model-check our Java implementation of the schedulers. There is a tool called JavaPathFinder(JPF) [11] developed by NASA to model-check mission critical Java software. While trying to model-check our code with this tool it turned out that this is hardly possible. Currently, JPF only supports part of the Java 1.4.2 and above, even though the source code of JPF uses Java 6 features. Yet, the fact that our code makes heavy use of features that have been introduced in Java 5 and Java 6, requires us to wait for future releases of JPF.

7 Evaluation

Within this section, we present the results of an extensive evaluation of Virtual Nodes. The results presented here are part of a much more in-depth analysis that can be found in [12].

Figure 9 gives an overview of the evaluation process. The model of the Virtual Nodes replication framework is mainly parameterised by the stochastic processes of arrival, service, failure and recovery. Here, the stochastic processes determine the inter-arrival, service, inter-failure and inter-recovery times. We generate these times according to the processes' probability distribution function/probability density function (*Distribution Generation*) and write them to a plain text file (*Input Data*).

Based on the input data, the *Client* schedules the different, possibly concurrent, events. That is, it submits requests to the *Virtual Node* which simulates processing of the request for a specified amount of time as well as it starts and stops replicas in order to simulate failures and recovery. The collected data is written to a *Log File* and is then analysed by the various *Analysis Tools* we implemented for the evaluation. The Analysis Tools generate some kind of *Report*, which contains the results of the analysis.

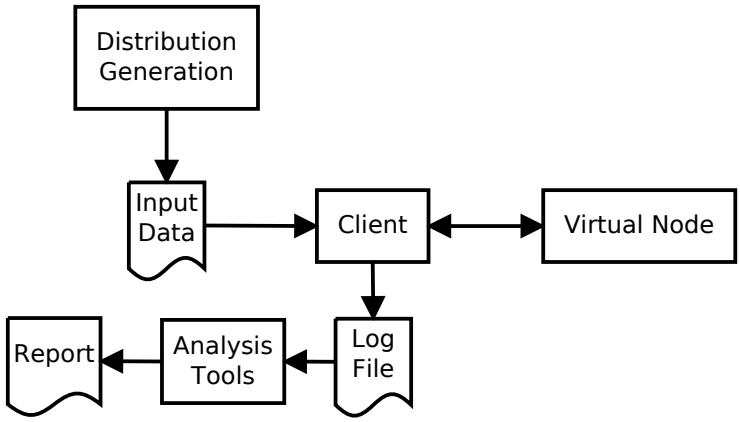


Figure 9: Evaluation Overview

7.1 Evaluation Setup

The evaluation setup is shown in Figure 10 and involves three entities. A client (class `Client`) is responsible for issuing requests, failures and recovery. The times when requests, failures and recovery occur have to be specified by a file that contains the times between successive requests, failures and recovery respectively; for sake of brevity we will refer to requests, failures and recovery as tasks. The advantage of specifying tasks a priori and relatively to each other and passing this timing information to the client in form of files is that the client can issue tasks according to any kind of stochastic distribution and experiments can be repeated with identical statistical properties. Tasks are scheduled by a timer (class `Timer`), which is similar to the implementation of `java.util.Timer` but must be started explicitly. A background thread schedules tasks for execution using `Object.wait(long)` and runnable tasks are then executed in a separate thread in order to keep the delay for the overall schedule at a minimum. The client creates an a priori schedule for requests before starting the evaluation. In contrast, failures and recovery are scheduled on demand; that is, the next failure is only triggered if the number of running replicas is not zero and the next recovery is only scheduled if not the maximum number of replicas are up.

Requests sent from the client to the service (interface `Service` and class `ServiceImpl`) are accompanied with a service time. The service simply executes a `for-loop` for the specified amount of time in order to generate load on the replica machines. To simulate different state sizes, the service has a field of type `byte[]` which can be varied in size when initially creating the service object. Each machine that eventually hosts a replica, runs a so-called replica manager (interface `ReplicaManager` and class `ReplicaManagerImpl`). The

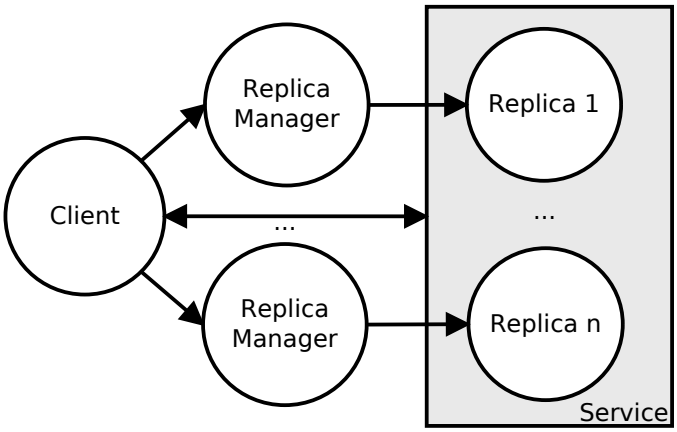


Figure 10: Evaluation setup

replica managers are necessary to remotely start or shutdown replicas. Furthermore, the service uses the replica manager for logging purposes.

Figure 11 shows the experimental setup. We are using five machines, connected by a switched 100 Mbit Ethernet, in total; Table 1 lists the basic configuration of the machines. Two machines, `Hykrion` and `Hynreck`, are connected by a switched 1 Gbit Ethernet. The responsibilities of the nodes are as follows:

- Client: `Emma`
- Registry: `Emma`
- Replicas: `Betty`, `Hykrion`, `Hynreck`, `Zenzi`

We use the latest development build of the Virtual Nodes replication framework extended with capabilities to log the point in time when events happen.

For sake of brevity, we use the following names to refer to the different group compositions:

- *Group 1:* `Hykrion`
- *Group 2:* `Zenzi`
- *Group 3:* `Hykrion` and `Zenzi`
- *Group 4:* `Hykrion`, `Hynreck` and `Zenzi`
- *Group 5:* `Hykrion`, `Hynreck`, `Zenzi` and `Betty`

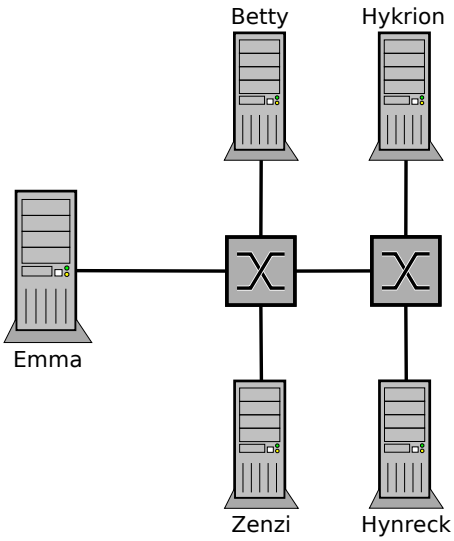


Figure 11: Experimental setup

Computer	**Configuration**
Betty	AMD Athlon 64 3800+ 2 GB RAM 250 GB HDD Ubuntu 8.04 Sun Java SE Environment 1.6.0_16
Emma Zenzi	AMD Athlon 64 3200+ 1 GB RAM 160 GB HDD Ubuntu 8.04 Sun Java SE Environment 1.6.0_16
Hykrion Hynreck	Intel Core 2 Duo 3 GHz 4 GB RAM 320 GB HDD Ubuntu 8.04 Sun Java SE Environment 1.6.0_16

Table 1: Computer Configurations

The basic procedure for the experiments is as follows. In order to avoid side effects caused by the Java Virtual Machine, a warm-up phase, in which the replicated service was called 1000 times, is carried out before each measurement series. Additionally, each measurement series is repeated three times with identical input data to reveal the influence of network conditions and system footprint as well as scheduling accuracy and influences of the Java runtime environment. Identical input data means that the same inter-arrival, service, inter-failure and inter-recovery times files are used. Furthermore, each measurement series consists of at least 2500 request to capture representative data.

7.2 Experiments Without Failures

In this section, we evaluate a Virtual Node using active or passive replication. The following experiments focus on collecting data allowing to compare both replication protocols in fault-free operation. Based on preliminary results [12] we conduct the experiments with an average inter-arrival time of 100 milliseconds and for two state sizes (0 and 0.25 MB). A measurement series consists of 2500 requests and each measurement series is repeated three times. The stated mean and standard deviation values are computed over the in total 7500 requests. Furthermore, we perform the measurements for different utilisations of the execution phase sub-model. The selected utilisation factors $\rho_{Execution}$ are 0.1, 0.25, 0.5, 0.75 and 0.9; that is, the average service time is 10, 25, 50, 75 and 90 milliseconds respectively.

Instead of parametrising the model with the theoretical values of the inter-arrival time (100 ms) and service time ($10,\ 25,\ 50,\ 75$ and 90 ms), we use the mean values of the sample data provided in the events file. The inter-arrival time samples have a mean of 102.14 milliseconds and the standard deviation is 105.73 milliseconds. The mean and standard deviation of the service time samples is shown in Table 2.

	Theoretical service time				
	10	25	50	75	90
Mean	9.58	25.25	49.26	74.81	92.03
Standard deviation	10.24	26.34	49.22	73.29	92.86

Table 2: Mean and standard deviation of the service time (2500 samples) in milliseconds

The client was configured to wait for replies from all correctly working replicas. In case of active replication, we use the `FairSelector` which selects the

contact replica in a round-robin fashion from the set of correctly working replicas. In case of passive replication, we use the `PrimarySelector` which always contacts the current primary replica.

7.2.1 Active Replication

In active replication, the replicas only exchange state information if a replica is started. That means, in fault-free operation with a fixed replica group, state size has no influence on performance. Therefore, it is sufficient to perform the measurements for one state size only; we choose to do this for state size 0 MB.

Tables 3 and 4 list the mean and standard deviation of the measured round-trip delay times. We first note that the standard deviations are high; in some cases

Group	**Utilisation**		
	0.1	0.25	0.5
1	14.50 ± 11.54	38.77 ± 36.39	104.46 ± 101.55
2	24.29 ± 37.06	50.44 ± 49.28	110.94 ± 101.86
3	29.69 ± 23.85	56.55 ± 44.98	114.09 ± 102.06
4	30.19 ± 22.49	54.69 ± 42.90	115.08 ± 102.51
5	31.04 ± 22.62	55.12 ± 42.72	114.16 ± 101.74

Table 3: Mean and standard deviation of the round-trip delay times for active replication in milliseconds

Group	**Utilisation**	
	0.75	0.9
1	274.00 ± 250.52	1091.40 ± 1092.98
2	279.98 ± 253.57	1095.78 ± 1082.76
3	285.37 ± 253.58	1106.54 ± 1093.97
4	284.70 ± 252.54	1102.87 ± 1086.05
5	286.88 ± 254.53	1107.63 ± 1085.37

Table 4: Mean and standard deviation of the round-trip delay times for active replication in milliseconds

higher than the mean value. It is clear, that round-trip delay times can not be negative as suggested here by the standard deviations but it describes the variability of the round-trip delay times. Highly variable round-trip delay times are not

completely foreclosed as the inter-arrival times as well as the service times also possess a high standard deviation.

Apparent is that the round-trip delay times increase clearly with the service time. The reason for this sharp-cut increase is the scheduling policy. Requests are being processed in sequential order; thus, newly arriving requests are being queued if one request is currently in service. When requests arrive almost at the same rate at which they are being processed, it is more likely that requests are queued and obviously the time spent in the queue increases with the number of queued requests as well as with the service time. Analysing the time it takes until a request leaves the system, that is the time span between the arrival at the replica machine and the end of the execution of `service()` of the class `ServiceImpl` (henceforth simply execution time), supports this intuitive view of sequential scheduling.

Comparing the execution times (Tables 5 and 6) to the round-trip delay times, we see that the execution times play a dominant role in the round-trip delay times. For utilisation 0.25, the proportion is already above 60% for all groups and increases up to at minimum 96% for utilisation 0.9.

	Utilisation		
Group	0.1	0.25	0.5
1	10.73 ± 11.09	34.29 ± 35.66	98.98 ± 99.67
2	10.71 ± 11.07	37.72 ± 38.92	98.47 ± 98.48
3	10.72 ± 11.11	36.21 ± 38.14	98.05 ± 99.40
4	10.66 ± 11.05	34.95 ± 36.62	97.94 ± 98.58
5	10.77 ± 11.16	34.79 ± 36.66	97.47 ± 97.92

Table 5: Mean and standard deviation of the execution times for active replication in milliseconds

	Utilisation	
Group	0.75	0.9
1	267.23 ± 248.79	1083.49 ± 1091.61
2	269.02 ± 252.33	1083.35 ± 1081.90
3	267.74 ± 250.68	1076.64 ± 1080.67
4	266.11 ± 248.59	1069.37 ± 1073.41
5	266.52 ± 249.34	1068.69 ± 1069.70

Table 6: Mean and standard deviation of the execution times for active replication in milliseconds

Moreover, the tendency of increasing round-trip delay times as the number of replicas increases is present. Note that the client was configured to wait for replies from all replicas; that is, the stated round-trip delay times are upper bounds and are determined by the slowest machine. This is apparent for group 3, which consists of `Hykrion` and `Zenzi`. Adding further machines to the group, the increase of the round-trip delay times is small; in some cases they also decrease. Two possible reasons for this are load balancing and the point of consensus. The client chooses the contact replica in a round-robin fashion; that is, the load of arriving requests is balanced across the group members. In turn the overall utilisation of the machines falls. Intuitively, we expect that group communication costs increase with the number of group members. Here, group communication costs are mainly determined by the costs of total-order multicast because the group composition does not change during the experiments. Total-order multicast and consensus are equivalent [9]. A group reaches consensus if the majority of the group agreed to the subject in question. In turn, a total-order multicast is successful if the majority of the group received it; that is meant by the point of consensus. If the point of consensus is reached, the replicas that already received the request can execute it even if not all group members received the request yet. To point this up, Table 7 lists the mean and standard deviation of the coordination times at the group members of group 3, 4 and 5 for the service time 75 milliseconds.

	Group		
Replica	3	4	5
`Hykrion`	6.26 ± 9.03	6.11 ± 8.61	6.34 ± 9.35
`Hynreck`	-	8.72 ± 11.38	8.49 ± 11.73
`Zenzi`	10.43 ± 13.40	9.39 ± 12.82	9.13 ± 12.53
`Betty`	-	-	9.44 ± 13.01
Average	8.35 ± 11.62	8.07 ± 11.17	8.47 ± 11.81

Table 7: Mean and standard deviation of the coordination times in milliseconds for utilisation 0.75

Not listed are the coordination times for group 1 (`Hykrion`) and group 2 (`Zenzi`); we note an average coordination time of 4.56 and 5.93 milliseconds respectively for both. Comparing replication (group 3) to no replication (group 1 and 2), the coordination time increases due to the costs of group communication. Adding `Hynreck` to the group (group 4), shows the two effects mentioned above. First, the coordination time of `Zenzi` decreases due to load balancing, and second, `Hykrion` can already start executing the request before `Zenzi` already received the request because passing the request to `Hynreck` is faster due to the

switched 1 Gbit Ethernet link between both machines. In total, the average coordination time of the group decreases too. For group 5, the effect of load balancing is again visible for `Hynreck` and `Zenzi`. In contrast, the execution at `Hykrion` is now delayed because `Hykrion` has to wait until a further node (here `Zenzi`) has received the request in order to reach consensus. Finally, we note that `Hykrion` always has the lowest coordination time of the members of group 3, 4 and 5. The reason is that JGroups establishes total ordering of the messages through a distinguished node called *sequencer*. In simple words, the contact replica sends each request it receives to the sequencer which adds a sequence number and broadcasts it to the other group members. The sequence numbers ensure that all nodes process the requests in identical order. The sequencer is operated on the node that is initially started; here, that is `Hykrion`.

Tables 8 and 9 list mean and standard deviation of the measured coordination times per group.

	Utilisation		
Group	0.1	0.25	0.5
1	3.53 ± 2.68	3.61 ± 3.57	4.18 ± 4.92
2	3.32 ± 3.66	4.82 ± 6.95	5.38 ± 10.31
3	7.24 ± 10.94	7.77 ± 11.28	8.03 ± 11.08
4	7.25 ± 8.92	7.31 ± 10.05	7.46 ± 10.38
5	7.34 ± 10.62	7.79 ± 10.64	8.29 ± 12.65

Table 8: Mean and standard deviation of the coordination times for active replication in milliseconds

	Utilisation	
Group	0.75	0.9
1	4.56 ± 6.37	5.01 ± 6.86
2	5.93 ± 9.58	6.44 ± 11.16
3	8.35 ± 11.62	9.44 ± 14.75
4	8.07 ± 11.17	8.78 ± 13.45
5	8.47 ± 11.81	9.19 ± 13.88

Table 9: Mean and standard deviation of the coordination times for active replication in milliseconds

7.2.2 Passive Replication

In passive replication, the state size plays an important role; that is, we have to perform the measurements for both state sizes. The primary replica of the groups of two, three and four replicas is always `Hykrion`. Furthermore, the primary replica updates the state of the backup after every request with the help of the group communication system.

Tables 10 and 11 list the mean and standard deviation of the round-trip delay times for passive replication and state size 0 MB. Again, the standard deviations of the round-trip delay times are high. Moreover, the tendency of increasing round-trip delay times as the utilisation $\rho_{Execution}$ as well as the number of replicas increases is present. More apparent than in active replication is that `Zenzi` is the least powerful machine. Comparing the round-trip delay times for `Hykrion` (group 1) and `Zenzi` (group 2), the round-trip delay times of `Zenzi` are clearly larger; for utilisation 0.75 and 0.9 they are several times larger. Somewhat unexpected are the differences of the round-trip delay times between both replication protocols for group 1 and 2. In case of a single replica, the performance of passive replication should be about the same size as of active replication. As for active replication, the execution times play a dominant role in the round-trip delay times. For utilisation 0.25, the proportion is already above 50% and increases up to more than 90% for utilisation 0.9. Furthermore, we note that execution takes a longer time than in case of active replication. The reason for this is that the workflow of active replication allows more parallelism than passive replication. In active replication, only the invocation of the service method is executed in mutual exclusion. In passive replication, also the state update of the backup replicas is performed in mutual exclusion. Hence, the overall time operating in mutual exclusion is larger for passive replication and thus the execution of successive requests is delayed for a longer time compared to active replication.

	Utilisation		
Group	0.1	0.25	0.5
1	18.84 ± 15.37	43.26 ± 38.33	112.78 ± 106.50
2	44.57 ± 41.12	78.48 ± 69.47	197.78 ± 185.41
3	26.28 ± 22.64	54.21 ± 45.62	125.18 ± 110.79
4	26.86 ± 22.83	52.65 ± 45.02	127.21 ± 112.78
5	38.55 ± 25.87	67.60 ± 47.27	133.04 ± 112.99

Table 10: Mean and standard deviation of the round-trip delay times for passive replication and state size 0 MB

Group	Utilisation	
	0.75	0.9
1	316.06 ± 292.01	1302.43 ± 1138.11
2	1417.09 ± 1794.15	9378.41 ± 8335.96
3	333.93 ± 298.55	1338.35 ± 1151.07
4	337.38 ± 297.85	1399.65 ± 1169.43
5	346.32 ± 299.07	1392.85 ± 1156.78

Table 11: Mean and standard deviation of the round-trip delay times for passive replication and state size 0 MB

Tables 12 and 13 list the mean and standard deviation of the measured agreement times. Apparent is that the agreement times of `Zenzi` are several times larger than of `Hykrion`. Moreover, they are several times larger than the coordination times of `Zenzi` in case of active replication. Furthermore, we note that the agreement times for the group of two, three and four replicas are larger than the coordination times for these groups in case of active replication; the difference is in the range of 29.21% to 110.19%. Possible reasons for this are load balancing in case of active replication and that the request size is small compared to the size of the replicated object.

Group	Utilisation		
	0.1	0.25	0.5
1	4.30 ± 10.35	3.81 ± 9.10	3.97 ± 8.45
2	19.29 ± 27.44	19.91 ± 28.33	20.23 ± 28.48
3	9.36 ± 17.35	10.81 ± 21.35	11.79 ± 21.10
4	10.28 ± 17.75	11.21 ± 21.60	12.73 ± 23.06
5	14.17 ± 14.57	15.02 ± 18.42	17.42 ± 23.12

Table 12: Mean and standard deviation of the agreement times for passive replication and state size 0 MB in milliseconds

Tables 14 and 15 list the mean and standard deviation of the round-trip delay times for passive replication and state size 0.25 MB. Again, the standard deviations of the round-trip delay times are high and the round-trip delay times increase with the utilisation. Moreover, the tendency of increasing round-trip delay times as the number of replicas increases is present. As expected, the round-trip delay times are larger compared to active replication as well as passive replication in case of state size 0 MB because of increasing costs of state updates. We note a minimum increase of the round-trip delay times of 7.4% to passive replication

Group	Utilisation	
	0.75	0.9
1	3.95 ± 7.29	4.42 ± 8.57
2	20.30 ± 28.76	21.48 ± 30.42
3	14.13 ± 25.32	13.89 ± 23.58
4	13.27 ± 21.37	14.32 ± 24.49
5	17.56 ± 22.79	17.46 ± 21.71

Table 13: Mean and standard deviation of the agreement times for passive replication and state size 0 MB in milliseconds

with state size 0 MB and of 10.61% to active replication.

Group	Utilisation		
	0.1	0.25	0.5
1	23.98 ± 24.54	49.42 ± 43.41	122.27 ± 112.55
2	85.03 ± 61.92	147.07 ± 117.80	616.39 ± 512.48
3	33.21 ± 25.00	62.54 ± 47.27	134.45 ± 114.96
4	38.19 ± 22.17	63.16 ± 46.47	141.24 ± 116.85
5	54.73 ± 30.03	77.11 ± 46.13	171.83 ± 121.02

Table 14: Mean and standard deviation of the round-trip delay times for passive replication and state size 0.25 MB in milliseconds

The increased costs of state updates are of course reflected in the agreement times; Tables 16 and 17 list the mean and standard deviation of the measured agreement times. Again, the agreement times of `zenzi` are several times larger than of `Hykrion`. Compared to the coordination times of active replication, the agreement times in case of state size 0.25 MB are at minimum more than 50% larger. In case of passive replication with state size 0 MB, the agreement times increase by at minimum 8.98% for state size 0.25 MB.

Tables 18 and 19 list the mean and standard deviation of the round-trip delay times in case of storing state updates to local disk and to a NFS folder respectively. Apparent is that the round-trip delay times for NFS are extremely large compared to the ones for local disk as well as for the group communication system. The measurements for utilisation 0.75 and 0.9 failed in case of NFS storage because of an exception due to too many open network sockets.

More interesting, the differences of the agreement times are small compared to the large differences of the round-trip delay times. Tables 20 and 21 list the mean and standard deviation in case of storing state updates to local disk and

Group	Utilisation	
	0.75	0.9
1	350.10 ± 317.20	1651.32 ± 1334.88
2	14350.62 ± 11342.22	40115.09 ± 26093.92
3	378.19 ± 330.52	1775.22 ± 1370.86
4	376.35 ± 331.83	1876.43 ± 1407.38
5	397.86 ± 339.57	1912.17 ± 1394.03

Table 15: Mean and standard deviation of the round-trip delay times for passive replication and state size 0.25 MB in milliseconds

Group	Utilisation		
	0.1	0.25	0.5
1	8.98 ± 21.96	8.61 ± 20.91	7.74 ± 16.58
2	38.86 ± 34.08	40.07 ± 33.82	41.45 ± 33.98
3	12.53 ± 18.26	13.86 ± 21.16	14.70 ± 21.21
4	12.43 ± 12.53	13.44 ± 17.97	15.22 ± 21.04
5	17.67 ± 14.37	17.99 ± 16.64	19.89 ± 20.30

Table 16: Mean and standard deviation of the agreement times for passive replication and state size 0.25 MB in milliseconds

Group	Utilisation	
	0.75	0.9
1	7.27 ± 13.77	7.72 ± 14.64
2	42.84 ± 36.33	46.04 ± 39.22
3	15.40 ± 21.82	15.86 ± 21.11
4	15.67 ± 21.41	15.84 ± 20.06
5	20.88 ± 21.43	20.88 ± 21.95

Table 17: Mean and standard deviation of the agreement times for passive replication and state size 0.25 MB in milliseconds

Storage	Utilisation		
	0.1	0.25	0.5
HDD	17.74 ± 12.63	41.91 ± 37.12	111.02 ± 105.11
NFS	10406.90 ± 10688.72	34522.72 ± 230404.30	55814.08 ± 35977.30

Table 18: Round-trip delay times for passive replication with alternative state update distribution and state size 0.25 MB in milliseconds

Storage	Utilisation	
	0.75	0.9
HDD	307.58 ± 282.70	1293.31 ± 1157.54
NFS	-	-

Table 19: Round-trip delay times for passive replication with alternative state update distribution and state size 0.25 MB in milliseconds

to a NFS folder respectively. The agreement times for NFS are round about 35 times as large as for local disk storage. Compared to the agreement times in case of state updates using the group communication system, the agreement times for NFS are round about 10 times larger. This explains the vast increase of the round-trip delay times and the exception due to too many open network sockets. Because the rate of agreement is less than the arrival rate, the number of requests waiting for agreement grows constantly. Moreover, this delays the execution of successive requests. In turn, the number of requests waiting for execution grows too. For utilisation 0.1, 0.25 and 0.5, the system is still able to process the large number of queued requests but with a large delay. In case of utilisation 0.75 and 0.9, the Virtual Node is no longer able to handle the requests. That is, it still accepts new requests but the delay of processing is so large that the software exceeds the number of allowed network connections and the experiment fails with a corresponding exception.

Storage	Utilisation		
	0.1	0.25	0.5
HDD	3.09 ± 3.11	2.85 ± 2.18	2.87 ± 1.62
NFS	107.03 ± 53.08	109.19 ± 35.17	106.18 ± 28.97

Table 20: Agreement times for passive replication with alternative state update distribution and state size 0.25 MB in milliseconds

Storage	Utilisation	
	0.75	0.9
HDD	3.11 ± 1.62	2.97 ± 2.08
NFS	-	-

Table 21: Agreement times for passive replication with alternative state update distribution and state size 0.25 MB in milliseconds

7.2.3 Comparison

We already analysed the round-trip delay times of a Virtual Node using active and passive replication respectively. Figure 12 compares the relative difference of the measured round-trip delay times of passive replication with state size 0 MB to active replication. In almost all cases, active replication performs better than passive replication. Somewhat unexpected but explainable is the better performance of active replication in case of groups of one replica only.

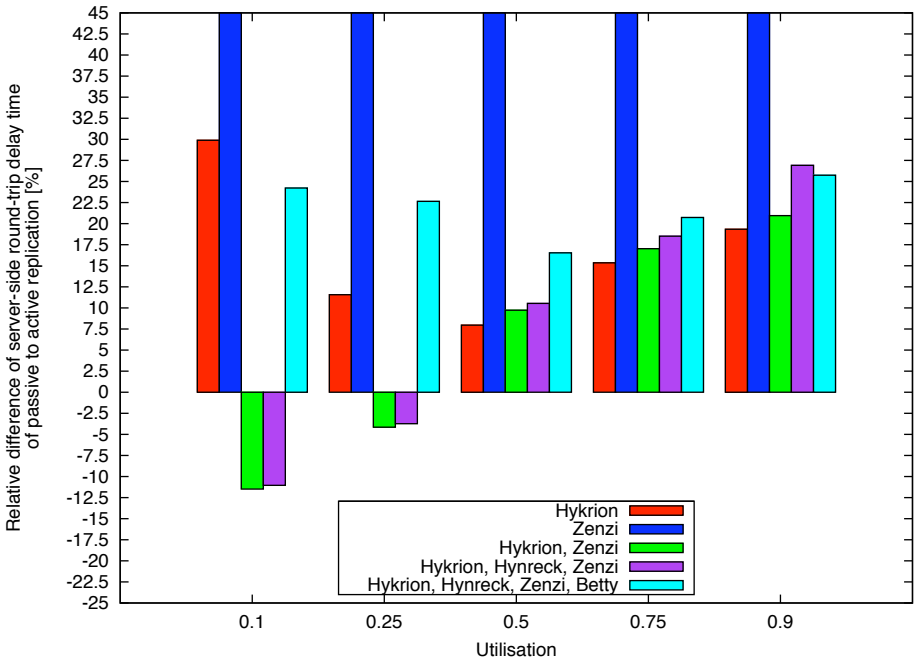


Figure 12: Relative difference of the round-trip delay times of passive replication with state size 0 MB to active replication in percent

As already discussed before, the workflow of active replication allows more parallelism than passive replication. Hence, the execution of successive requests is

less delayed. For Hykrion, passive replication degrades performance by 7.96% to 29.89%. Note that the diagram cuts off values above 40%. In case of Zenzi, passive replication degrades performance by 55.58% to 755.87%. Passive replication provides better performance only in four cases. In case of utilisation 0.1 and 0.25, the group of two and three replicas improve performance by round about 11% and 4% respectively. Moreover, the tendency is present that the performance gain of active replication increases as the number of replicas as well as the utilisation increases. A possible reason for this is load balancing. Ignoring the groups consisting of a single replica, active replication improves performance by round about 10% to 25%.

For the sake of completeness, we state the cut off relative deviations. The relative difference of passive to active replication for Zenzi is 145.38% in case of utilisation 0.1 and 50.97% for utilisation 0.25. For the group of four replicas, the relative difference is 46.23% in case of utilisation 0.1. Figure 13 compares the relative difference of the measured round-trip delay times of passive replication with state size 0.25 MB to active replication. As we can see, active replication provides better performance in all cases. For sake of clarity, the diagram cuts off values above 45%. The improvement of active replication is in the range of 17.04% to 65.36% for Hykrion and 191.57% to 5025.54% for Zenzi. We note a performance gain of active replication in the range of 10.61% to 60.43% for the group of two replicas, 15.49% to 70.14% for the group of three replicas and 39.90% to 76.33% for the group of four replicas. Moreover, the tendency that the performance gain of active replication increases as the number of replicas increases is present. Again, a possible reason for this is load balancing.

Figure 14 compares the relative difference of the computed round-trip delay times of passive replication with state size 0.25 MB to active replication. The diagram cuts off values above 03%. In all cases, active replication provides better performance.

For the sake of completeness, we state the cut off relative deviations. The relative difference of passive to active replication for Zenzi is 423.54% in case of utilisation 0.1, 157.72% for utilisation 0.25 and 63.55% for utilisation 0.5. For Hykrion, the relative deviation is 43.48% in case of utilisation 0.1. Furthermore, we note a relative difference of 35.37% for the group of two replicas and 34.51% for the group of three replicas; both in case of utilisation 0.1. For the group of four replicas, the relative difference is 72.82% for utilisation 0.1 and 31.91% for utilisation 0.25.

7.3 Experiments with failures

This section evaluates active and passive replication in the presence of failures and the influence of the replication protocol on availability. We set the average

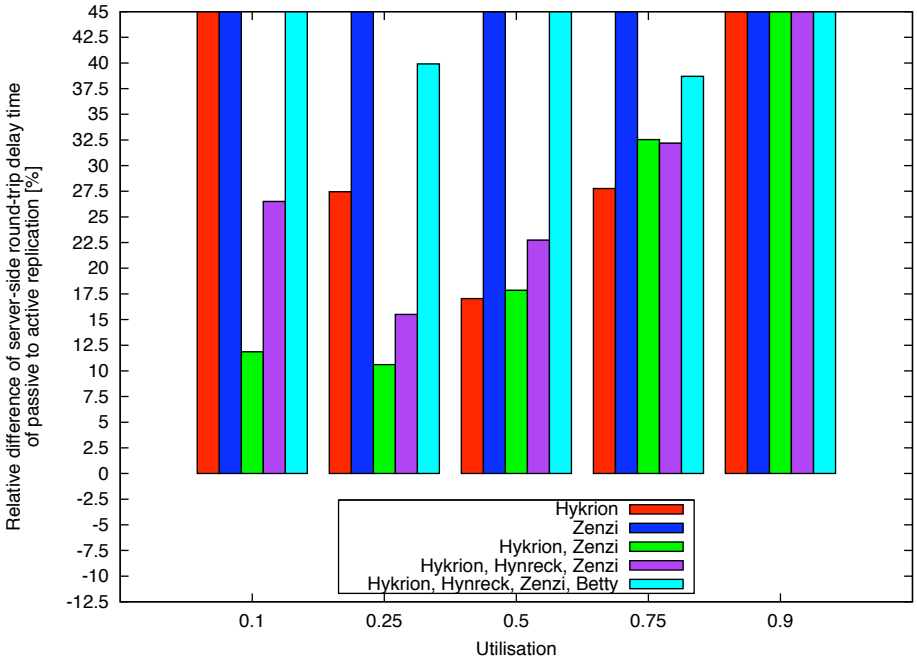


Figure 13: Relative difference of the round-trip delay times of passive replication with state size 0.25 MB to active replication in percent

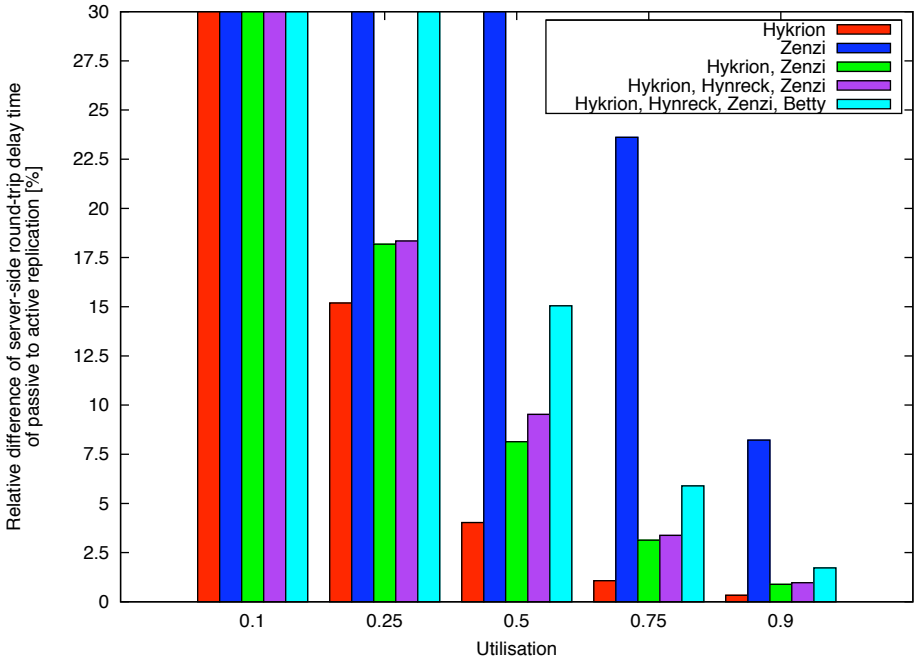


Figure 14: Relative difference of the computed round-trip delay times of passive replication with state size 0.25 MB to active replication in percent

inter-failure time τ_F to 50 seconds; that is, a replica fails on average every 500-th request. To capture a representative number of failures and recovery, we conduct this experiment for 5000 requests. Hence, we expect on average 10 failures per replica machine. The recovery rate is set to $0.167\frac{1}{s}$; that is, the recovery time is 6 seconds and represents the time after which considers a participating node to be failed. Except that, we use the same configuration of the Virtual Nodes framework as for the previous experiments but will evaluate availability only for an utilisation $\rho_{Execution}$ of 0.25 and 0.75.

A Virtual Node is up if it consists of at least one correctly working replica when using active replication or if it is has a correctly working primary replica when using passive replication. Since we are using `startNewReplica()` and `shutdownReplica()` provided by the `AdminMethods` interface of the Virtual Nodes framework to start and shut down replicas respectively, we can only determine when the invocation of theses methods returns. The major issue is that a shutdown of a replica is being executed in an asynchronous manner; that means, `shutdownReplica()` returns before the replica is actually shut down. This semantic might be acceptable for active replication but is not for passive replication because we possibly have to choose a new primary. The most reliable measure for availability is the fraction of answered requests because a Virtual Node can only answer requests if it is up. Henceforth, we report availability as the fraction of answered requests.

7.3.1 Active Replication

Tables 22 and 23 list the mean and standard deviation of availability for active replication with state size 0 and 0.25 MB respectively. Using no replication (group 1 and 2), round about 15.50% of the requests are not processed for utilisation 0.25 and both state sizes; that means, the Virtual Node answers 84.50% of the requests. The fraction of answered requests decreases slightly to 83.50% in the case of state size 0 MB and utilisation 0.75 but is still of comparable size for `Hykrion` (group 1) and `Zenzi` (group 2). Hence, it is somewhat unexpected that `Zenzi` looses about 3% more requests than `Hykrion` for state size 0.25 MB.

Replication clearly improves availability in case of state size 0 MB. Here, at most 5% of the requests are not answered. In case of state size 0.25 MB, the maximum fraction of lost requests decreases only to 12.30%. Moreover, the tendency of increasing availability as the number of replicas increases is present. We note that only 1% of the requests are not answered for state size 0 MB and utilisation 0.25 if using the maximum number of four replicas (group 5). The maximum number of replicas in case of state size 0.25 MB is three replicas (group 4) because measurements for four replicas failed with an exception due to too many open network sockets. This error occurred also during the previous measurements

and indicates a timing related issue because repeating the measurements sometimes resolves the problem. Furthermore, we observed that the problem occurs more often for groups with larger size. In this case, 3.5% of the requests are not answered for utilisation 0.25. For an utilisation of 0.75, availability also increases with the number of replicas but is even for the maximum number of replicas above 10%. Somewhat unexpected is that availability increases for two replicas but decreases if adding further replicas in case of state size 0 MB and utilisation 0.75.

Group	**Utilisation**	
	0.25	0.75
1	0.84527 ± 0.00012	0.83527 ± 0.00050
2	0.84533 ± 0.00046	0.83480 ± 0.00035
3	0.97687 ± 0.00579	0.97227 ± 0.00101
4	0.98027 ± 0.00537	0.96493 ± 0.00631
5	0.98980 ± 0.00367	0.95060 ± 0.01684

Table 22: Mean and standard deviation of availability for active replication and state size 0 MB

Group	**Utilisation**	
	0.25	0.75
1	0.84533 ± 0.00031	0.82373 ± 0.02212
2	0.84600 ± 0.00000	0.79927 ± 0.00031
3	0.96180 ± 0.02463	0.87713 ± 0.04233
4	0.96567 ± 0.01864	0.89333 ± 0.01030
5	-	-

Table 23: Mean and standard deviation of availability for active replication and state size 0.25 MB

As already mentioned, the tendency of decreasing availability as the utilisation increases is present. Figure 15 compares the relative difference of availability of utilisation 0.75 to 0.25. In case of state size 0 MB, the relative difference of utilisation 0.75 to 0.25 is comparable for `Hykrion` and `Zenzi`; we note a relative difference of -1.18% and -1.25% respectively. For the group of `Hykrion` and `Zenzi`, the relative difference of availability is less than for the single machines; a possible reason for this is load balancing. Moreover, the relative difference due to higher utilisation increases with the group size. A possible reason for this is that more requests are waiting for execution due to increased group communication

costs and hence the average number of lost requests increases. For the largest group, we note a relative difference of availability of -3.96%.

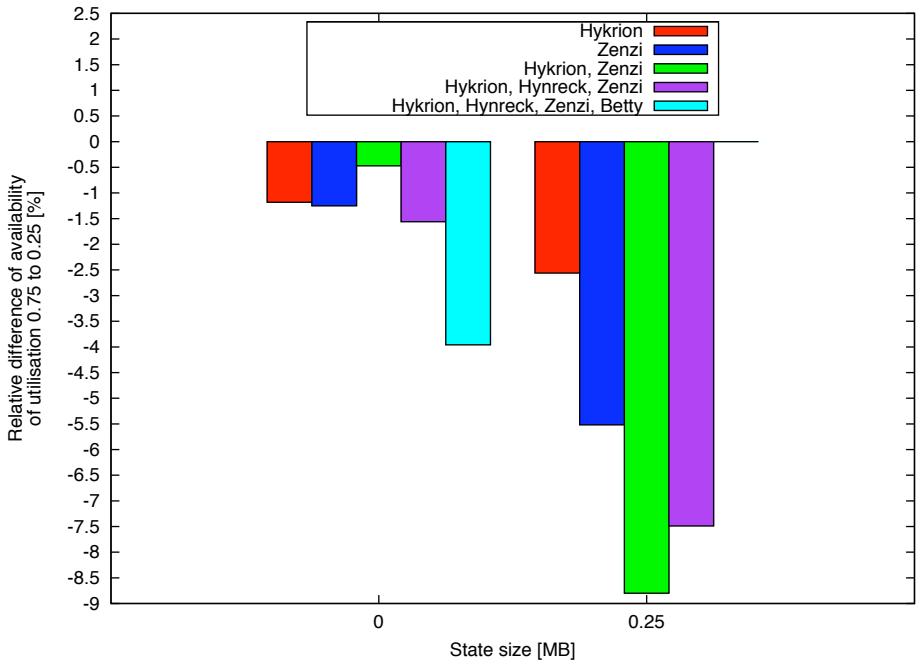


Figure 15: Relative difference of availability in relation of the utilisation for active replication in percent

In case of state size 0.25 MB, the relative difference of availability is several times larger than for state size 0 MB. Apparent is the difference between `Zenzi` and `Hykrion`: Its availability decreases by -5.52% compared to -2.56%. For the group of `Hykrion`, `Hynreck`, `Zenzi` and `Betty` we have no comparative data.

A further tendency is that availability decreases as the state size increases. This is expected because the initial state transfer delays recovery: Since the Virtual Node operates with less replicas for a longer time period, it is more vulnerable to further failures. Figure 16 compares the relative difference of availability of state size 0.25 to 0 MB. In case of utilisation 0.25, availability increases slightly for `Hykrion` and `Zenzi`; we note an relative improvement of 0.01% and 0.08% respectively. For the groups of `Hykrion` and `Zenzi` as well as of `Hykrion`, `Hynreck` and `Zenzi` availability decreases by round about 1.5%. In case of utilisation 0.75, availability decreases for all groups. We note the minimum of -1.38% for `Hykrion` and the maximum of -9.78% for the group of `Hykrion` and `Zenzi`.

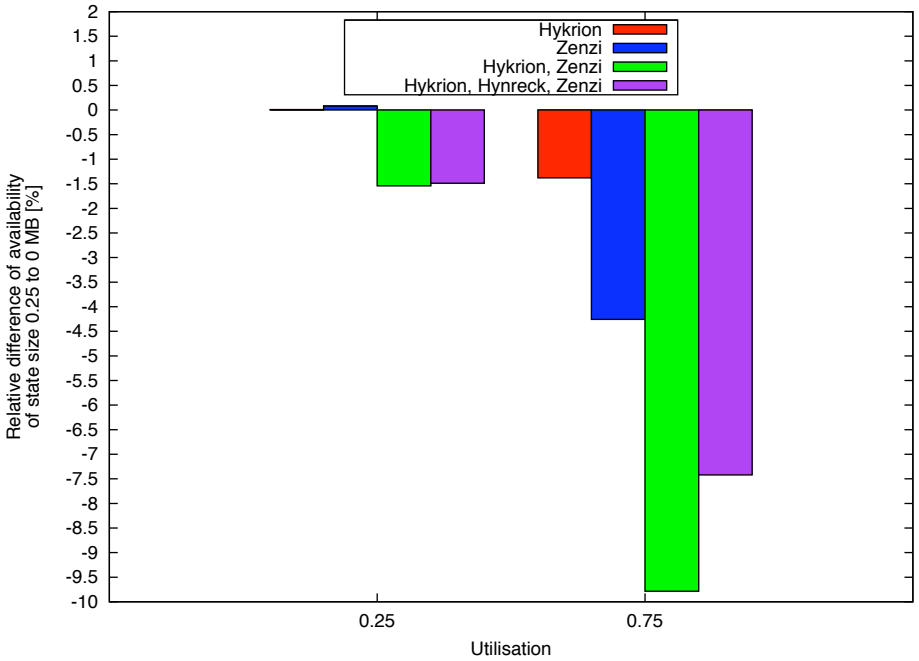


Figure 16: Relative difference of availability in relation of the state size for active replication in percent

7.3.2 Passive Replication

Tables 24 and 25 list the mean and standard deviation of availability for passive replication with state size 0 and 0.25 MB respectively. Using no replication (group 1 and 2), round about 15% of the requests are not processed for utilisation 0.25 and both state sizes; that means, the Virtual Node answers more than 84.50% of the requests. For state size 0 MB and utilisation 0.75, availability decreases slightly by -0.04% for `Hykrion` but -2.03% for `Zenzi`. In case of state size 0.25 MB, the availability of `Hykrion` increases slightly (14.99% loss) but decreases clearly for `Zenzi` (23.65% loss).

Again, replication clearly improves availability. Here, the loss for the group of `Hykrion` and `Zenzi` decreases to 5% for state size 0 MB and 8.68% for state size 0.25 MB in case of utilisation 0.25. For utilisation 0.75, the loss is slightly above 10% for both state sizes. Moreover, the tendency of increasing availability as the number of replicas increases is also present. In case of utilisation 0.25, we note a minimum loss rate of 1.5% for state size 0 MB and of 5% for state size 0.25 MB. Somewhat unexpected, the maximum availability is not provided by the group of four replicas but by the group of three replicas. For utilisation 0.75, the group of four replicas provides the maximum availability for both state sizes.

Group	**Utilisation**	
	0.25	0.75
1	0.84653 ± 0.00064	0.84620 ± 0.00035
2	0.84560 ± 0.00131	0.82840 ± 0.00178
3	0.95293 ± 0.02825	0.89980 ± 0.03532
4	0.98513 ± 0.00012	0.91673 ± 0.02469
5	0.97847 ± 0.00446	0.96160 ± 0.00904

Table 24: Mean and standard deviation of availability for passive replication and state size 0 MB

As already mentioned, the tendency of decreasing availability as the utilisation increases is present. Figure 17 compares the relative difference of availability of utilisation 0.75 to 0.25. In case of state size 0 MB, the availability of the Virtual Node decreases for all groups as the utilisation increases. We note the minimum decrease of availability of -0.04% for `Hykrion` (group 1) and the maximum of -6.94% for group 4.

In case of state size 0.25 MB, the relative difference of availability is several times larger than for state size 0 MB. Apparent is the sharp-cut decrease of availability of 9.93% for `Zenzi`. Somewhat unexpected is that the availability of the group of four replicas increases compared to state size 0 MB.

Group	Utilisation	
	0.25	0.75
1	0.84640 ± 0.00053	0.85013 ± 0.00031
2	0.84767 ± 0.00240	0.76353 ± 0.06698
3	0.91320 ± 0.02945	0.89660 ± 0.01196
4	0.94980 ± 0.02985	0.91807 ± 0.06106
5	0.94587 ± 0.05217	0.97233 ± 0.00991

Table 25: Mean and standard deviation of availability for passive replication and state size 0.25 MB

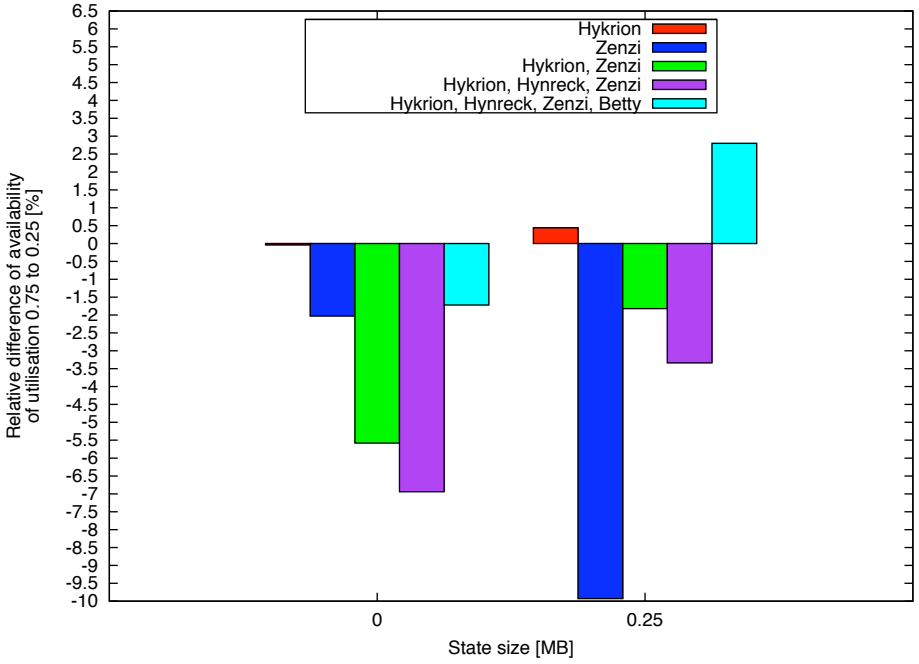


Figure 17: Relative difference of availability in relation of the utilisation for passive replication in percent

Again, the tendency that availability decreases as the state size increases is present for utilisation 0.25. This is expected because the initial state transfer delays recovery: Since the Virtual Node operates with less replicas for a longer time period, it is more vulnerable to further failures. Figure 18 compares the relative difference of availability of state size 0.25 to 0 MB. In case of utilisation 0.25, availability decreases slightly for `Hykrion` but increases for `Zenzi`; we note an relative decrease of 0.02% and increase of 0.24% respectively. For all other the groups, availability decreases. The minimum and maximum relative decrease are -3.33% for the group of four replicas and 4.17% for the group of `Hykrion` and `Zenzi`. In case of utilisation 0.75, availability improves by 0.46% for `Hykrion` but decreases clearly by 7.83% for `Zenzi`. For the group `Hykrion` and `Zenzi`, availability decreases slightly by -0.36% compared to state size 0 MB. Somewhat unexpected is that availability increases for the group of three and four replicas. We note an increase of availability of 0.15% for the group of three replicas and of 1.15% for the group of four replicas.

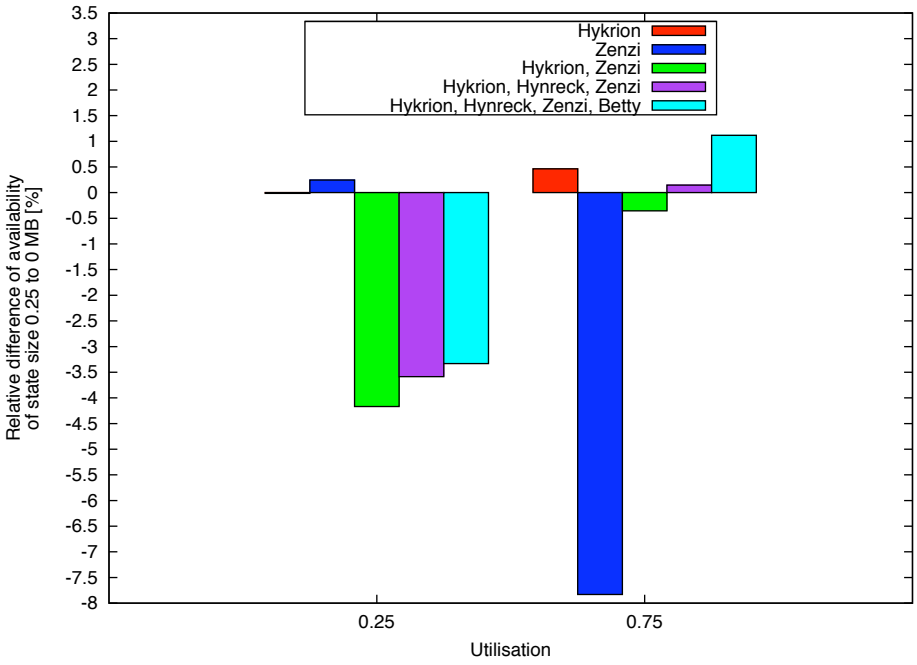


Figure 18: Relative difference of availability in relation of the state size for passive replication in percent

7.3.3 Analysis

We already analysed the availability of a Virtual Node using active and passive replication respectively. Figure 19 compares the relative difference of the measured availability of passive to active replication for state size 0 MB. For no replication, passive replication slightly improves availability by 0.15% for `Hykrion` and 0.03% for `Zenzi` in case of utilisation 0.25. In case of utilisation 0.75, passive replication improves availability by 1.31% for `Hykrion` but degrades availability by 0.77% for `Zenzi`.

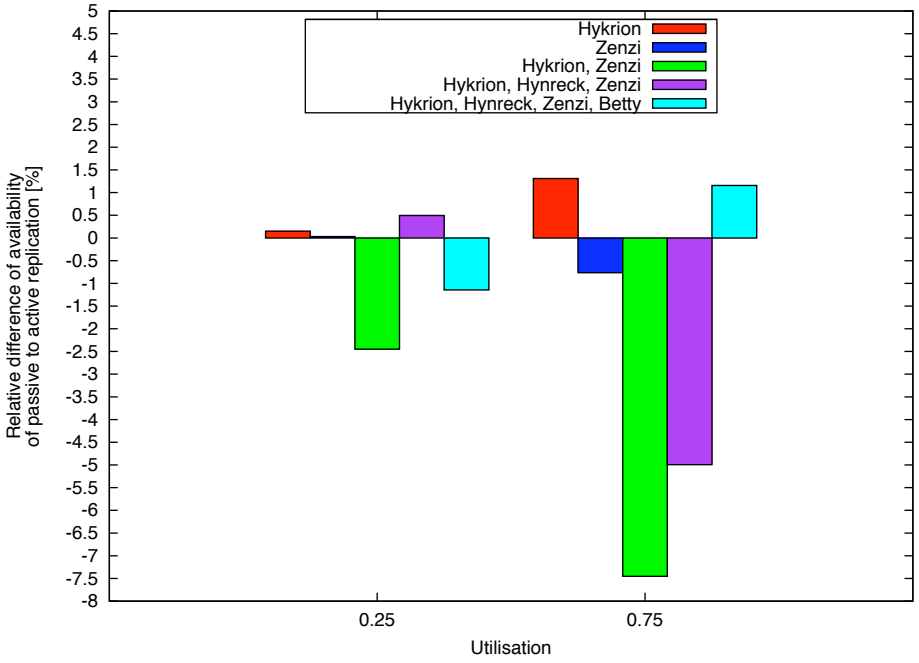


Figure 19: Relative difference of availability of passive to active replication for state size 0 MB

For replication, active replication provides better availability in two of three cases. In case of utilisation 0.25, availability of passive replication decreases by 2.45% for the group of two replicas and 1.15% for the group of four replicas but increases by 0.50% for the group of three replicas. In case of utilisation 0.75, we note a relative difference of availability of passive to active replication of -7.45% for the group of two replicas and -5.00% for the group of three replicas. For the group of four replicas, passive replication improves availability by 1.16%.

Figure 20 compares the relative difference of the measured availability of passive to active replication for state size 0.25 MB. We restrict this to groups of at most three replicas because we have no comparative data for active replica-

tion with this state size for the group of four replicas. Again, passive replication improves slightly availability for the groups consisting of one replica only and utilisation 0.25; the improvement is 0.13% for `hykrion` and 0.20% for `Zenzi`. In case of utilisation 0.75, passive replication improves availability by 3.20% for `Hykrion` but degrades availability by 4.47% for `Zenzi`.

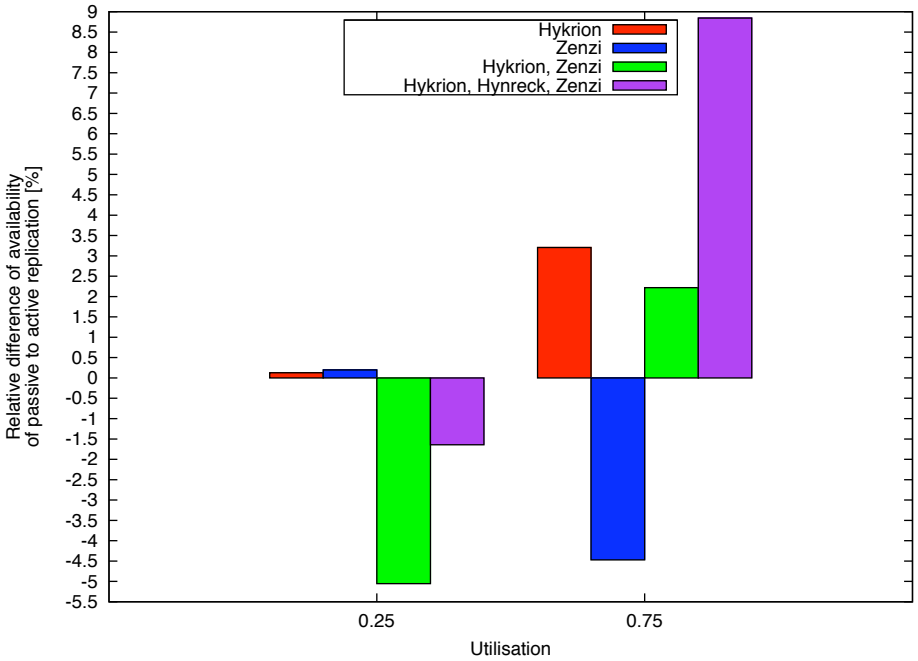


Figure 20: Relative difference of availability of passive to active replication for state size 0.25 MB

For replication, active replication provides better availability for utilisation 0.25. We note a decrease of availability if using passive replication of -5.05% for the group of two replicas and -1.64% for the group of three replicas. Hence, it is somewhat unexpected that passive replication provides better availability in case of utilisation 0.75. Here, passive replication improves availability by 2.22% in case of two replicas and by 8.84% in case of three replicas.

7.4 Summary

The analysis of the experiments without failures aimed at several things. One goal was to basically evaluate the performance of the Virtual Nodes replication framework and the influence of the server-side configurations options on availability as well as performance. That is, the differences of active and passive replication. In our setup, active replication provides better performance in most of the cases.

Furthermore, we saw that the sequential scheduling policies play a dominant role in the round-trip delay times.

The goal of the experiments with failures was to measure the service availability depending on the degree of replication. To name the replication protocol that provides better availability is more difficult. In some cases, active replication provides better availability and in other cases passive replication provides better availability. We have to further refine those results in order to make stronger and more reliable suggestions regarding when to use which replication protocol.

8 Conclusions

In this section we proved the usability of Virtual Nodes. First, we presented the design and layout of a replicated POP3 server. Afterwards, we showed our integration efforts in order to connect Virtual Nodes to other parts of XtreemOS. In particular, we are working on integrating them into the Application Execution Management and the DIXI communication infrastructure. Furthermore, we are merging them with Distributed Servers. This provides the following benefits to the project. First of all, job management becomes fault-tolerant so that information about jobs is not lost if the responsible core node fails during job execution. Second, having a DIXI front-end to Virtual Nodes opens replication facilities to all XtreemOS services that are implemented in Java and use DIXI. Third, the integration with Distributed Servers allows to provide replication without having to change any client-side software.

Apart from integration we have proven the correctness of the scheduling algorithms being shipped with the Virtual Nodes release using a SAT solver and a model checker. Finally, we presented an extensive performance and reliability evaluation of Virtual Nodes.

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

[3] The XtreemOS Consortium. D3.2.2: First prototype version of ad hoc distributed server, December 2007.

[4] The XtreemOS Consortium. D3.2.5: Design and specification of a virtual node system, December 2007.

[5] The XtreemOS Consortium. D3.3.3: Basic services for job submission, control and monitoring, December 2007.

[6] The XtreemOS Consortium. D3.3.4: Basic services for resource selection, allocation and monitoring, December 2007.

[7] The XtreemOS Consortium. D3.2.10: On the feasibility of integration between distributed servers and virtual nodes, December 2008.

[8] The XtreemOS Consortium. D3.2.9: Reproducible evaluation of a virtual node system, December 2008.

[9] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition)*. Addison Wesley, 2005.

[10] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[11] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[12] Martin Pfeil. Optimising and Self-adaptive Strategy Selection in a Replication Framework. master thesis VS-D07-2009, Institute of Distributed Systems, Ulm University, Germany, 2009.

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

[14] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.