Efficient State Transfer for Hypervisor-Based Proactive Recovery[*]

Tobias Distler, Rüdiger Kapitza
Dept. of Comp. Sciences, Informatik 4
University of Erlangen-Nürnberg
Germany
{distler,rrkapitz}@cs.fau.de

Hans P. Reiser
LaSIGE, Departamento de Informática
University of Lisboa
Portugal
hans@di.fc.ul.pt

ABSTRACT

Proactive recovery of replicated services is a novel approach that allows tolerating a potentially unlimited number of malicious faults during system lifetime by periodically restarting replicas from a correct state. Recovering a stateful replica requires a time-consuming transfer and verification of the state. During this time, the replica usually is unable to handle client requests. Our VM-FIT architecture harnesses virtualization to significantly reduce this service unavailability. Our approach allows recovery in parallel with service execution, and uses copy-on-write techniques and provides efficient state transfer support between virtual replicas on a host.

1. INTRODUCTION

Replication is a popular mechanism for architecting dependable distributed applications. Byzantine state machine replication [1] allows tolerating arbitrary malicious faults. However, the correctness of the algorithms used in such systems depends on tight bounds on the number of faults. Typically, a system with n replicas can tolerate up to f Byzantine faults only if $n > 3f$. Proactive recovery [2] is a technique to periodically refresh replicas and thus to eliminate faults. Systems with proactive recovery can tolerate an unlimited number of faults during system lifetime as long as the number of faults remains bounded within the recovery period.

In previous work [3,4] we have presented VM-FIT, a replication architecture based on virtualization technology. In VM-FIT, the replication logic is provided in an isolated system domain, while application replicas are executed in separated virtual machines. The architecture supports Byzantine fault tolerance and offers support for proactive recovery.

In this paper we discuss state transfer in a proactive recovery system. In practice, most distributed applications will have state, and proactive recovery requires transferring the state from existing replicas to a newly recovered one. To minimize the time needed for recovery, optimized strategies for state transfer are necessary. We have implemented such efficient strategies in an extended VM-FIT prototype. An abstract state representation allows state transfer between heterogeneous replicas. Virtualization allows parallelizing the state transfer with service execution, and thus minimizes the unavailability during recovery. We compare early results of a stream-based and a disk-remapping-based approach for state transfer between replicas in virtual machines of the same host and show that both approaches perform similarly.

In the next section, we give a short overview of the VM-FIT architecture. In Section 3 we provide a classification of application state variants, and illustrate our optimized approach. Section 4 presents an experimental evaluation of our current prototype. Section 5 discusses related work, and Section 6 concludes.

2. THE VM-FIT ARCHITECTURE

The VM-FIT prototype implements a generic architecture for intrusion-tolerant replication of network-based services using virtualization technology. In previous publications [3,4] we have presented the basic architecture and discussed the advantages for doing proactive recovery, without focussing on the efficiency of state transfer.

2.1 System and Threat Model

The VM-FIT architecture makes the following assumptions on system model and threat model:

- All client–service interaction is intercepted at the network level. Clients exclusively interact with a remote service on the basis of request/reply network messages.
- The remote service can be modelled as a deterministic state machine. This property allows replication using standard state machine replication techniques.

[*]This work was partially supported by the EU through NoE IST-4-026764-NOE (RESIST/FOREVER) and project IST-4-027513-STP (CRUTIAL), and by the FCT through the Multiannual Programme.

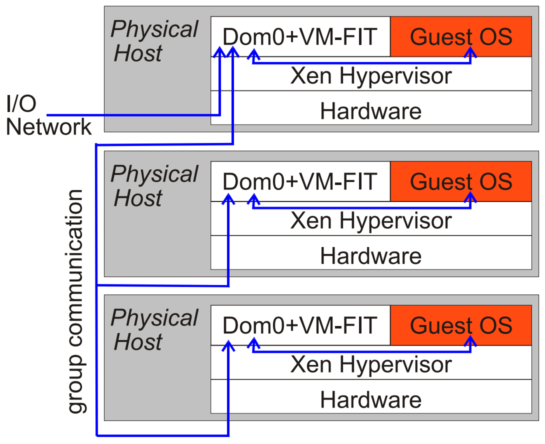

Figure 1: VM-FIT basic replication architecture

- Service replicas, including their operating system and execution environment, may fail in arbitrary (Byzantine) ways. At most $f < \lfloor \frac{n-1}{2} \rfloor$ replicas may fail within a single recovery round.
- The remaining components of the system (which include the hypervisor and a trusted system domain) fail only by crashing.

2.2 Prototype

The VM-FIT prototype is based on the Xen 3.1 hypervisor. In the basic system architecture (see Figure 1), a replica manager is running within the privileged *Domain 0*, while service replicas are being executed in completely separated *application domains* (Guest OS).

The trusted replica manager includes a group communication system for consistently distributing client requests to replicas, a voter, and the logic for proactive recovery. The placement of the group communication system in the trusted domain implies that we must rely on the correctness of this part of the system. This might seem to be strong assumption, but the alternative of implementing this functionality in the untrusted domain using BFT algorithms suffers from similar problems: It would require additional means to avoid that a vulnerability in the BFT group communication can be exploited simultaneously in multiple replicas. Formal verification techniques can more easily be used for group communication in the benign trusted domain than for more complex systems in the Byzantine domain.

Service replicas, which include an operating system, a middleware infrastructure, and a service implementation, are placed in isolated application domains and can be fully heterogeneous. Diversity of replicas is essential in an intrusion-tolerant system to maintain fault-independence of the replicas. This diversity has an important impact on state transfer: It is not sufficient to simply capture the on-disk or in-memory state of a replica for transferring it to another replica. Rather, an additional conversation between diverse state representations of different replicas is required. VM-FIT simplifies the provision of diversity, as the replication logic is deployed completely separated from the replicas in an isolated domain. The replicas itself therefore do not need to implement replication mechanisms.

Another advantage of the VM-FIT architecture is its support for a hybrid fault model. The fault model for the application domain and for the Domain 0 with the replication logic can be defined independently. Assuming a Byzantine model for both domains results in a traditional BFT replication scenario. The typical configuration of VM-FIT assumes crash-stop behaviour of the replication logic and Byzantine faults in replica domains. This configuration allows coping with any kind of failure in application domains, including random non-crash faults and intentional malicious faults. On the other hand, only $2f + 1$ replicas are needed to tolerate f malicious faults with in application domains.

The previous VM-FIT prototype already supported proactive recovery by periodically triggering the replacement of the current replica with a new replica in a new virtual machine. Creating a new domain in parallel with the service execution allows a reduction of the time of unavailability during the transition from “old” to “new” replica instance.

3. STATE TRANSFER

3.1 System Model: Classification of State

An efficient state transfer mechanism has to cope with the heterogeneity of replicas and the irrelevance of large parts of internal replica state. For example, many internal variables of operating system and middleware are irrelevant for the logical state of a replicated service. We capture this situation with a system model that distinguishes system state, volatile application state, and persistent application state.

The *system state* includes all internal configuration data of operating system and middleware, while the *application state* pertains to the replica implementation. The system state requires no consistent replication mechanisms. Our architecture assumes the existence of a secure code storage from which each replica can restore a correct system state. The application state is the only relevant state that needs to be kept consistent across replicas.

We divide the application state into *volatile state* and *persistent state*. Volatile state refers to all kind of data stored in memory. Usually, only small parts of the volatile state have to be considered for state transfer after the application has reached a consistent state (e.g., no requests are currently being processed). Persistent state represents all data that is stored on disk and usually builds the main part of the application state.

The internal representation of volatile and persistent state may differ between replicas. In fact, the desired replica diversity makes it unlikely that two replica instances use identical state representations. However, it is assumed that the state is kept consistent from a logical point of view, and that all replicas are able to convert the internal application-specific state into an external representation of the logical state, in the following referred to as *abstract state*, and vice versa. Such an approach requires explicit support in the replica implementations to transform their state.

The separation of system state and application state helps to reduce the amount of state data that needs to be transferred. The complex transfer of real state can be limited to the minimally necessary application state, while the basically static system state can be transferred directly from a single code base. The secure code base can be available locally on each replica host, and thus transferring the system state does not require any network communication.

In our architecture, the replicated application is responsible for distinguishing between these state categories and transforming from and to abstract state. In practice, a wrapper approach as proposed by BASE [5] could be used for separating the transformation code from the application logic itself. This strategy would also simplify the reuse of existing code; for example, a wrapper could request a standard SQL dump from a database application and then convert the dump into a normalized format. In the scope of this paper, we only assume that the replication manager can trigger that the application transforms its state to the abstract format and back.

3.2 Basic State Transfer Techniques

Most systems supporting proactive recovery require rebooting the entire machine and relying on a secure read-only image (e.g., a cdrom). In this case, the recovering replica is unavailable during reboot and application state transfer.

The use of a hypervisor allows the coexistence of multiple replicas in separate virtual machines on a single machine. This helps to reduce the downtime of a node, as the currently active replica (*"senior replica"*) can remain available while the new replica (*"shadow replica"*) is being booted from a secure code base. It also simplifies the state transfer, as the state can be passed directly from the senior to the shadow replica. The validity of the state can be verified with identical state checksums from at least $f + 1$ replicas. Furthermore, the hypervisor-based approach enables the simultaneous recovery of all replicas. Conventionally, only parts of the replicas should be restarted at a time, in order to avoid complete unavailability [6].

Independently from using a hypervisor, the request processing during recovery can be handled in different ways: One approach is to block all replicas for the whole recovery time. Alternatively, the request processing is only blocked while creating a snapshot of the application's current state, which later on serves as a basis to initialize the newly instantiated replica. In the latter case, as service delivery is resumed after capturing the state, requests processed afterwards have to be re-executed by the new replica before it becomes fully functional. VM-FIT supports the latter variant for minimizing the time of unavailability, and is able to recover all replicas simultaneously.

3.3 Conceptual Phases of a State Transfer

The state transfer between a senior application replica and a shadow replica can be structured in several steps, as illustrated by Figure 2:

Capture: First, the state of the senior replica has to be captured into a snapshot of all application-relevant volatile and persistent state. Before creating the snapshot, the replica has to finish processing all requests in execution.

Conversion: Next, the senior replica converts the snapshot an application-specific format to an abstract format that all replica instances can interpret.

Transmission: The replica manager coordinates a direct transmission of abstract state from the senior replica to the shadow replica.

Verification: Before initializing the shadow replica with the new state, it is necessary to verify that the state is not corrupted and identical to the abstract state of all other correct replicas. Therefore, the shadow replica generates checksums over all parts of the state and forwards them to the replica manager, which broadcasts the checksums to all other replica managers. After receiving $f + 1$ identical checksums, the replica manager compares the value with the locally generated checksum. If identical, it informs the shadow replica about the correctness of the checkpoint.

Error correction: In case of wrong checksums, the replica manager has to ask another replica manager with a valid checksum for the correct state data. The correct data is then passed to the shadow replica.

Restoration: The shadow replica in turn converts each verified part of the abstract state to the implementation-specific representation.

Adoption: Finally, the shadow replica can start executing on the basis of the received state.

There are two important points to note. First, the state conversion from application-specific format to the abstract one takes place in the senior replica. This operation might be complex and intruders might try to exploit the conversion routines so this should not be handled by the replica manager or the shadow replica. Second, the abstract state is directly forwarded to the shadow replica before validation. The rational behind this is to reduce data transfer, as it is cheaper to send only the checksums to the replica manager instead of the whole data and then generate them.

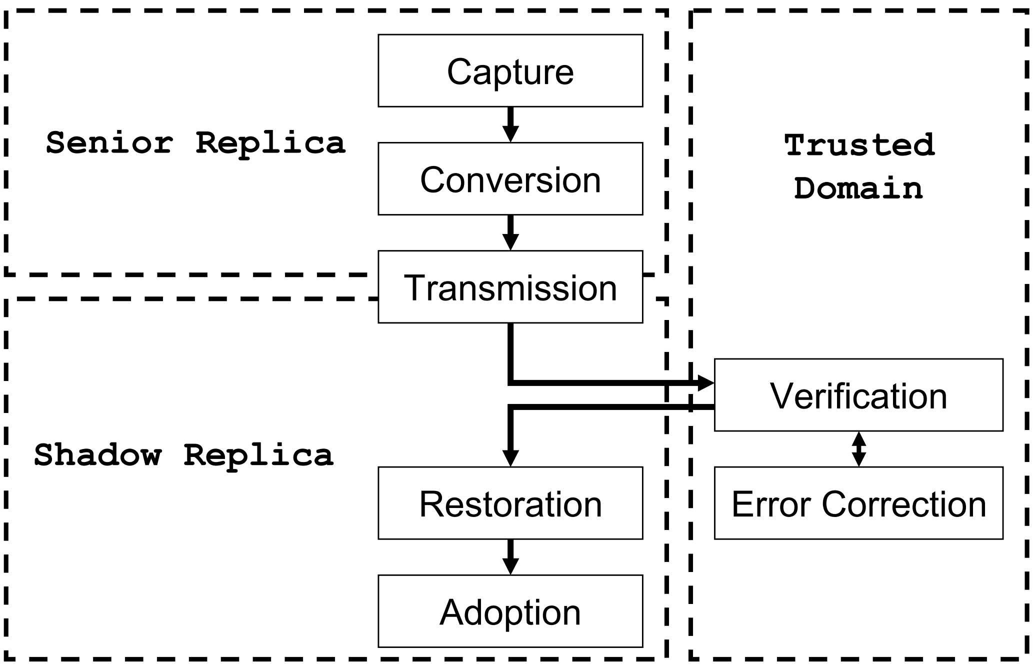

Figure 2: Conceptual phases of a state transfer

3.4 Preparing the State Transfer

Before the state transfer, the application state has to be captured by generating a consistent checkpoint. For this purpose, VM-FIT delays all subsequent client requests in the replica manager and sends a take-snapshot message to an application-specific replica wrapper, which conceptionally is part of the VM-FIT architecture. The wrapper instructs the application to write all replication-relevant volatile and persistent state after pending requests are finished.

The completion of this operation is announced to the replica manager, which in turn reacts by detaching the disk volume with the application data from the replica domain. Immediately after that it reattaches the volume twice to the same senior domain: one time as a read-only volume and another as a copy-on-write volume. The read-only volume contains the snapshot and is used for the state transfer. The copy-on-write (cow) volume on the other hand is used as a basis for further service execution.

Thus the senior replica can take over request processing and change the state of the cow volume without affecting the snapshot. When the senior replica starts processing requests, the replica manager logs these requests in order to re-execute them at the shadow replica. This way the actual duration of replica unavailability can be reduced to the tasks of taking the snapshot and reattaching the disk.

3.5 Stream-based State Transfer

In the *stream-based state transfer* approach, the senior replica reads the application-specific state from the snapshot volume and converts the state to the abstract format. The converted state is transferred to the shadow replica in parallel to its generation. At the shadow replica, the replica wrapper reads the stream block-wise and builds a checksum over each received block. The checksum is then transmitted to the local replica manager, which sends the checksum to all other replica managers using group communication. As soon as f remote checksums have been received that match the local checksum ($f+1$ identical values), the shadow replica is signalled that it can safely convert the block from the abstract format to the application-specific representation. The reception of $f + 1$ identical remote values, not matching the local value, indicates a state corruption. In this case the shadow replica has to be initialized by requesting all subsequent state blocks from a remote replica manager that supplied the correct checksum. As the trusted remote manager has already verified the checksum, no further verification of the received remote blocks is necessary.

3.6 Disk-based State Transfer

The *disk-based state transfer* approach uses the support of the hypervisor not only for the fast generation of a snapshot, but also for the actual state transfer using a dedicated transfer disk volume.

After the snapshot volume has been attached as described in Section 3.4, the senior replica generates the abstract state and writes it to the transfer volume. After completing this operation, the replica manager is informed and detaches this volume. This way, the senior replica is no longer able to modify the volume. The replica manager then maps the disk volume into the shadow replica. The shadow replica generates checksums on a per-file basis and forwards them to the replica manager, which

Domain	*Operating System*	*Kernel*
Dom0	Linux (Ubuntu)	2.6.18-xen
DomU	Linux (Debain)	2.6.19-4-generic
	Net BSD	NetBSD 4.0_RC4
	Open Solaris	SunOS 5.11

Table 1: Testbed Setting

in turn uses group communication to verify the checksum similar to the stream-based approach. As soon as a file is verified it can be converted into the service-specific representation. If the verification fails, the replica manager requests the file from a remote manager that has a valid checksum.

3.7 Finishing the State Transfer

After converting abstract state to local application-specific state, the shadow replica is ready to execute client requests. First, it has to re-execute all requests executed by the senior replica after the snapshot has been taken, discarding all client replies, until becoming up-to-date. After that, the replica manager can safely shut down the senior replica and integrate the shadow replica into the replica group as the new senior replica.

4. EXPERIMENTAL EVALUATION

In a previous publication [3] we have evaluated the basic advantage of hypervisor-based proactive recovery without considering the impact of state transfer: The VM-FIT architecture allows remaining available during proactive recovery of a node without requiring additional replicas. In a sample scenario, a traditional shutdown/reboot recovery caused downtimes of up to $40s$, while downtime with the VM-FIT approach never exceeded $1s$. Unavailability during recovery can also be avoided by adding additional replicas, but this not only increases hardware cost, but also makes it more difficult to achieve diversity of the replicas, which is a critical prerequisite for building intrusion-tolerant systems. In this paper, we focus on evaluating approaches for optimizing the transfer of large system states in the VM-FIT architecture.

4.1 Environment

All simulations presented here use four equally equipped machines (Intel Core 2 CPU with 2.4 GHz, 2 GB RAM) linked with switched 1Gb/s Ethernet. One host simulates 500 client applications using individual threads. At the beginning of a test run, each client establishes a single connection to one of the VM-FIT hosts, keeping it open from then on. After that, each client repeatedly sends service requests, waiting for a reply before sending the next request. Three hosts run the VM-FIT prototype with identical Domain 0 configurations (Table 1) using the Spread toolkit to perform group communication. Different operating systems (Debian Linux, NetBSD, and OpenSolaris) are deployed in the replica domains, in order to create a heterogeneous environment. The service component is a Java application capable of managing a trivial data base, i.e. delivering and modifying specific record sets on request. No heterogeneity is used at the application level, but in order to simulate the work load during the conversion and restoration step, the whole state is copied each time.

4.2 Transmission Alternatives

The first experiment provides a comparison of the two transmission alternatives mentioned above. A state of size 150 MB (purely persistent) is transferred twice, using a) the stream-based and b) the disk-based approach.

The measurements attained from this evaluation are presented in Figure 3. Both graphs are synchronized at the beginning of state capturing. The results show that for medium sized states the stream-based transmission is superior to the disk-based variant: In the first case, state transfer is done in 40 seconds, while in the second case it takes about twice the time (77 seconds). Both approaches add no additional service unavailability to the 3 seconds necessary during state capture. However, transmitting the state via stream affects the throughput to a greater extend as further direct inter-domain communication operations are performed. Using a transfer volume, on the other hand, only includes local disk access and therefore achieves an enhanced average performance during state transfer.

In the measurements we observed significant performance differences between Debian, BSD, and Solaris. Due to this heterogeneity, different replicas advance at different speeds, and some complete the recovery process earlier than others. In the stream-based simulation, the nodes running Debian and BSD finish recovery after 22 seconds, resulting in a visible increase in throughput. The ongoing recovery at the Solaris host causes performance degradation until $t = 40s$.

4.3 Transfer Phases

A second simulation further investigates the disk-based transmission approach with a large, purely persistent state (1 GB). The results of the simulation (Figure 4) show that the recovery process can be divided into six stages regarding the influence on the service's throughput.

During the first stage (I) each node sets up a shadow domain. The booting of a Xen guest domain consumes local resources, decreasing the remaining system's throughput by approximately 20%. With all shadow domains being set up after nearly 2.5 minutes, the state transfer process itself can be initiated. At first, the processing of client requests by application replicas is suspended during the stage of state capturing (II), causing a temporary service downtime. The copy-on-write approach

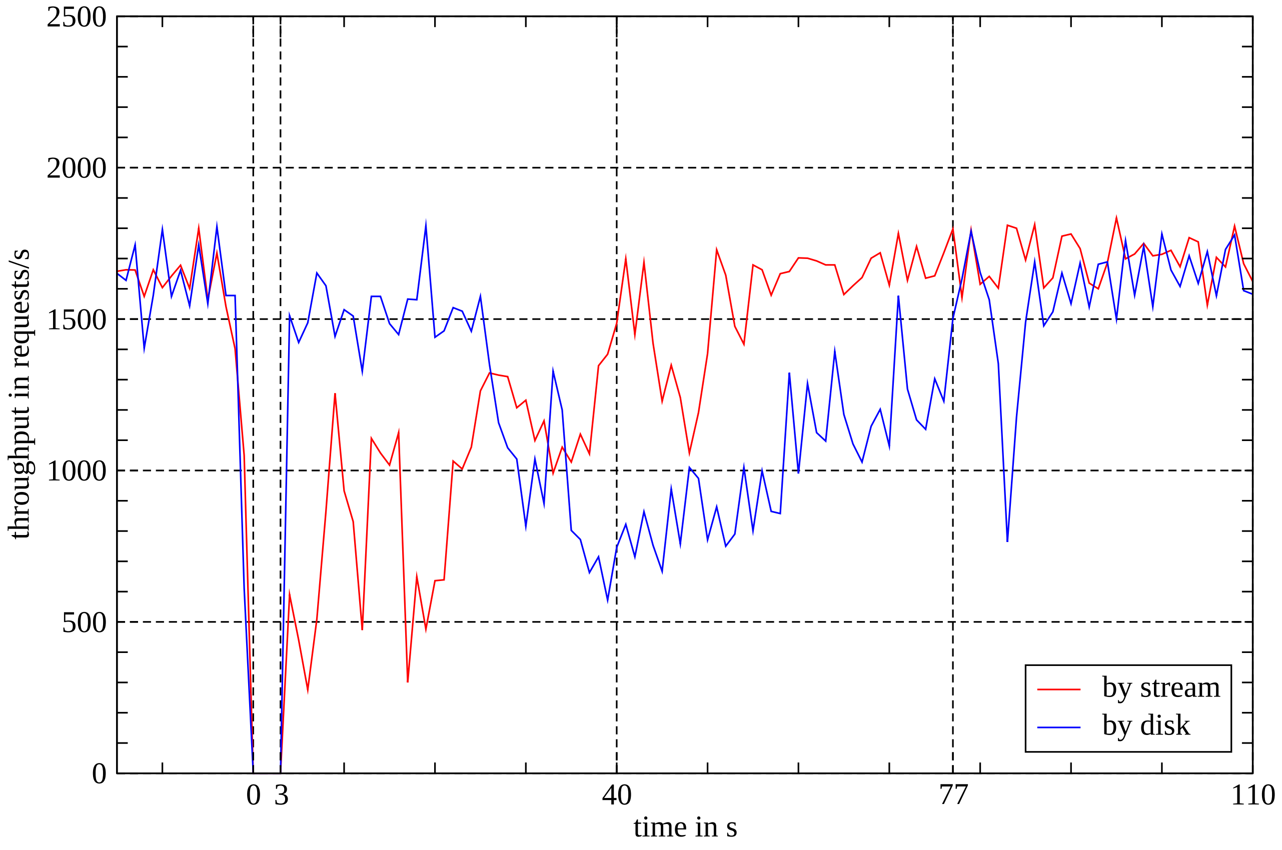

Figure 3: Comparison of state transfer approaches

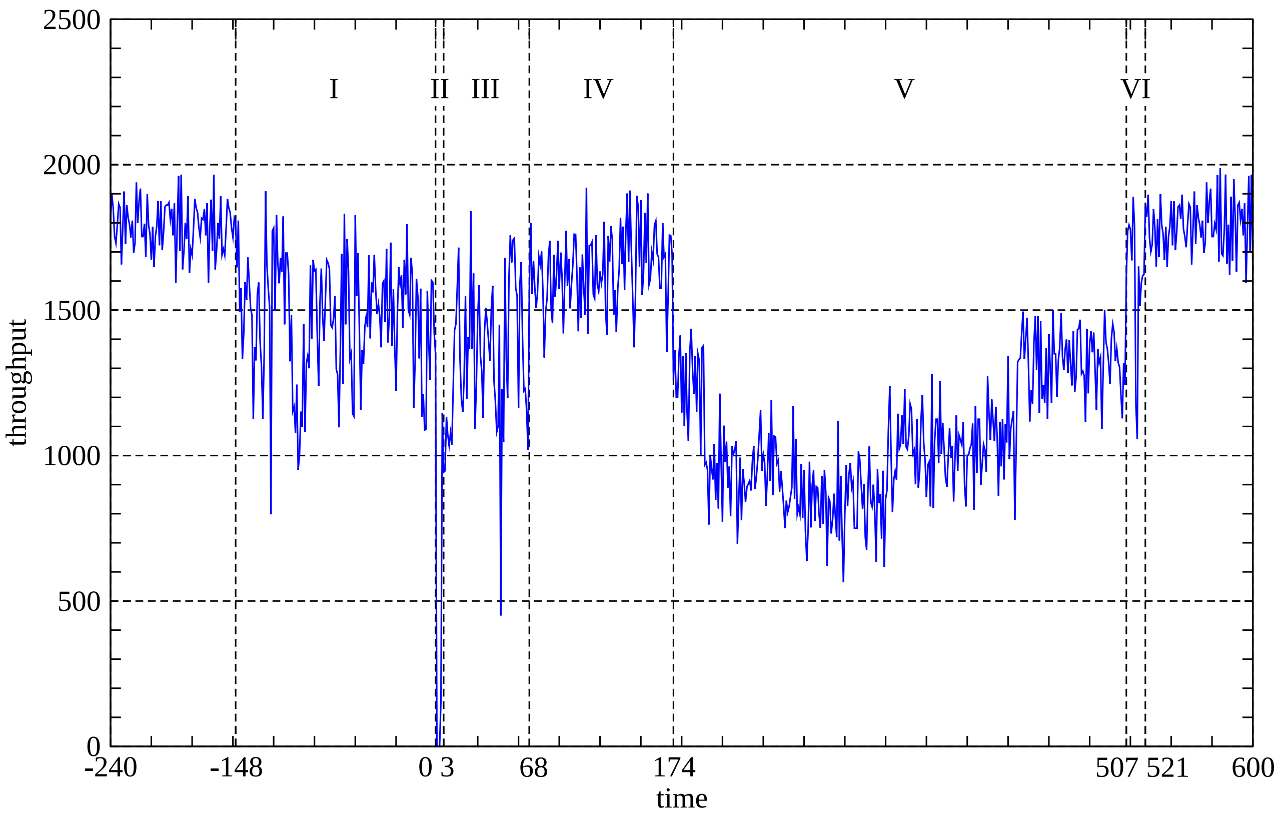

Figure 4: Transfer phases

used for state capturing in VM-FIT minimizes this period of system unavailability to an interval of about 3-4 seconds. After checkpoint creation the replicas can resume their service, and thus no additional downtime is necessary during the following stages of system recovery. The next step of state transfer is the conversion into a common abstract representation (III). This work consumes CPU and disk resources in parallel to the execution of client requests, and thus reduces service throughput. Having passed the conversion stage, all further steps, including the state's verification, restoration, and adoption, can be executed without significant influence on the senior replicas (IV). As soon as a shadow replica has been updated with the transferred state, the local replica manager starts the update procedure, i.e., it forwards the buffered client requests. This action marks the beginning of a period of high system load, as the replica manager has to cope with message traffic from two replicas processing requests in parallel. As a result, the achievable request throughput temporarily drops to about half the base level (V). After finishing the update step, the shadow replicas replace the senior replicas and the service can again be provided at full efficiency. The shutdown of the previous senior replica is visible as a short-term decrease of the global throughput (VI).

The development of the service's performance during the state update phase (V) illustrates another example of how heterogeneity of replicas affects the recovery process: In this case, the Debian as well as the BSD shadow domain start updating their states while the node running the Solaris domUs is still transferring the data. Therefore, the throughput does not reach its minimum immediately but stays temporarily on an intermediate level. An even greater impact of implementation diversity is visible at the end of the state update phase as each replica completes the process at a certain point in time. Subsequently, the throughput is not rising in one but three steps: The first improvement follows the switching of service delivery to the shadow replica on the node running the Debian domUs at $t = 338s$. The next one to finish updating is the BSD shadow replica at $t = 427s$. Finally, transferring the state between Solaris replicas in this experiment finishes at $t = 507s$.

4.4 Conclusions

Our measurements show that even for large system state, a hypervisor-based recovery approach achieves a small recovery-induced downtime with a minimal number of replicas. Suspending a service for 3 seconds on each recovery is acceptable for many types of applications, e.g. web servers. Assuming a recovery period of, for example, 10 minutes it guarantees an availability of over 99%.

For some services such a delay might not be acceptable. In this case, our approach could be combined with the addition of new replicas, which would allow completely hiding the service downtime from clients. Alternatively, the current prototype offers potential for optimizations. The state-capturing process can be improved by enhancing the performance of system-inherent operations like attaching and detaching a block device to a Xen domain or reconfiguring a logical volume for copy-on-write usage. Besides that, well-known techniques for Xen migration [7, 8] could be integrated into our approach to create a service mirror within milliseconds, with the original image immediately continuing to serve requests and the mirror handling the state transformation.

5. RELATED WORK

Silva et al. [9] propose a proactive recovery approach similar to VM-FIT for software rejuvenation, but mainly focus on recovering from error situations caused by "software ageing". Ramasamy and Schunter [10] use combinatorial modelling to analyse how the use of virtualization can affect system dependability. Such a careful analysis allows a better judgement on the conditions that are necessary to make virtualization-based replication more reliable than non-replicated one.

Several authors have previously used proactive recovery in Byzantine fault tolerant systems [2, 11–13]. Sousa et al. [6] define requirements on the number of replicas that avoid potential periods of unavailability given maximum numbers of simultaneously faulty and recovering replicas. Our approach instead reduces unavailability during recovery by performing most of the recovery in parallel to normal system operation in the basis of virtualization technology.

To avoid long service interruptions during state transfer, group communication systems such as Eternal [14] and work by Birman [15] provide solutions in which the system or parts of the system are blocked only for the time of the state acquisition. However, both approaches do not consider Byzantine faults.

The problem of state transfer has previously been addressed by the BFT protocol of Castro and Liskov [2]. They recognize that efficiency of state transfer is essential in proactive recovery systems and propose a solution that creates a hierarchical partition of the state in order to minimize the amount of data to transfer. BASE [5] proposes abstraction for state transfer in a heterogeneous, Byzantine fault tolerant system. In addition, BASE provides wrapper functions for handling non-determinism, a technique that could also be added to our architecture. VM-FIT uses a similar abstraction approach, but also parallelizes state transfer and recovery with normal operation, as well as exploits virtualization for minimizing state transfer time.

Clark et al. [7] describe a technique for migrating Xen-based virtual machines between hosts. Their pre-copy approach is very efficient (full migration of virtual machine on LAN within milliseconds), but only transfers the memory image, not the persistent on-disk state. Bradford et al. [8] extend this approach to transfer persistent disk state as well. Both approaches assume homogeneous nodes and do not consider state verification. It might be feasible to add state verification to these approaches and thus apply them to BFT replication. However, the main advantage of the VM-FIT approach is its support for heterogeneous systems, which not only includes heterogeneous hardware, but also diversity at the software level.

6. SUMMARY

In this paper we have presented approaches for efficient state transfer between heterogeneous replicas in an intrusion-tolerant replication system with proactive recovery. Our approach creates a new replica instance using virtualization technologies and creates atomic state checkpoints using hypervisor-based copy-on-write techniques. Replicas continue to handle client requests while state is transferred to a new replica instance, thus maintaining system availability during state transfer. For the moment, measurements attained from evaluations conducted with our prototype allow no final conclusions whether the stream-based or the disk-based state transfer alternative is more efficient.

We would like to thank the anonymous reviewers for their encouraging comments that helped to improve the final version of this paper.

7. REFERENCES

[1] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI '99: Proc. of the third Symp. on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[2] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Fourth Symp. on Operating Systems Design and Implementation (OSDI)*, San Diego, USA, October 2000.

[3] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proc. of the 26th IEEE Symp. on Reliable Distributed Systems - SRDS'07 (Oct 10-12, 2007, Beijing, China)*, pages 83–92, 2007.

[4] H. P. Reiser and R. Kapitza. VM-FIT: supporting intrusion tolerance with virtualisation technology. In *Proc. of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems (Lisbon, Portugal, March 23, 2007)*, pages 18–22, 2007.

[5] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proc. of the 18th ACM Symp. on Operating System Principles*, pages 15–28, Banff, Canada, October 2001.

[6] P. Sousa, N. F. Neves, P. Verissimo, and W. H. Sanders. Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available. In *SRDS '06: Proc. of the 25th IEEE Symp. on Reliable Distributed Systems (SRDS'06)*, pages 71–82, 2006.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, May 2005.

[8] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proc. of the 3rd Int. Conf. on Virtual Execution Environments*, pages 169–179. ACM, 2007.

[9] L. M. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak. Using virtualization to improve software rejuvenation. In *Proc. of the 6th IEEE Int. Symp. on Network Computing and Applications (NCA 2007)*, pages 33–44, 2007.

[10] H. V. Ramasamy and M. Schunter. Architecting dependable systems using virtualization. In *Workshop on Architecting Dependable Systems: Supplemental Volume of the 2007 International Conference on Dependable Systems and Networks (DSN-2007)*, 2007.

[11] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *CCS '99: Proc. of the 6th ACM conference on Computer and communications security*, pages 18–27, New York, NY, USA, 1999. ACM Press.

[12] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS '02: Proc. of the 9th ACM conf. on Computer and communications security*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[13] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *PODC '91: Proc. of the 10h annual ACM Symp. on Principles of Distributed Computing*, pages 51–59, 1991.

[14] P. Narasimhan, L. Moser, and P. M. Melliar-Smith. State synchronization and recovery for strongly consistent replicated CORBA objects. In *DSN '01: Proc. of the 2001 Int. Conf. on Dependable Systems and Networks*, pages 261–270. IEEE Computer Society, 2001.

[15] K. P. Birman. *Building secure and reliable network applications.* Manning Publications Co., Greenwich, CT, USA, May 1997.