State Transfer for Hypervisor-Based Proactive Recovery of Heterogeneous Replicated Services [*]

Tobias Distler[†], Rüdiger Kapitza[†], Hans P. Reiser[◇]
[†]University of Erlangen-Nürnberg, Germany, [◇]University of Lisboa, Portugal
{distler,rrkapitz}@cs.fau.de, hans@di.fc.ul.pt

Abstract: Intrusion-tolerant replication enables the construction of systems that tolerate a finite number of malicious faults. An arbitrary number of faults can be tolerated during system lifetime if faults are eliminated periodically by proactive recovery. The periodic rejuvenation of stateful replicas requires the transfer and validation of the replica state. This paper presents two novel efficient state transfer protocols for a hypervisor-based replication architecture that supports proactive recovery. Our approach handles heterogeneous replicas, and allows changing/updating the replica implementation on each recovery. We harness virtualization for an efficient state transfer between "old" and "new" replicas in virtual machines on the same physical host, and use copy-on-write disk snapshots for low-intrusive recovery of replicas in parallel with service execution. We apply the generic algorithm to a realistic three-tier application (RUBiS) and study the impact of recovery and state transfer on system performance.

1 Introduction

Dependability is an increasingly important requirement for distributed systems. In the extreme case, a system should operate correctly even if some parts of it are compromised by a malicious attacker. Byzantine fault-tolerant (BFT) replication [CL02] is a practical approach to build such systems. Typical BFT systems need $n \geq 3f + 1$ replicas in order to tolerate up to f malicious faults. Replica diversity is necessary to avoid common-mode faults in multiple replicas (achieved either by expensive N-version programming [AC77], or by opportunistic components-off-the-shelf diversity [CRL03, GP07] and system-level diversification such as address-space randomization [SBO08]).

BFT systems also need a mechanism for recovering (refreshing) faulty replicas. Otherwise, given enough time, attackers could eventually succeed in compromising more than f replicas. Proactive recovery [CL02] is a technique for refreshing replicas periodically. It enables systems to tolerate an unlimited number of faults during system lifetime as long as the number of faults remains limited within the recovery period. Recovery operations can reduce system availability, and compensating this impact by increasing the number of replicas [SNVS06] is expensive. In practical systems, a better option is to instead minimize the impact of periodic recovery actions on system operation.

[*]This work was partially supported by the German Research Council (DFG) under grant no. KA 3171/1.

The VM-FIT project [RK07] has shown that virtualization technology can be harnessed efficiently for building an intrusion-tolerant replication architecture with proactive recovery. The architecture uses a *hybrid failure model*, supporting BFT at the service-replica level, whereas the replication infrastructure is executed in a *trusted computing base* that fails only by crashing. Thus, VM-FIT can be seen as a intermediate step between simple fail-stop systems and pure BFT replication. It requires only $2f + 1$ replicas, which reduces resource requirements and makes it easier to use heterogeneous components-off-the-shelf software. Another benefit of VM-FIT is that the system support for proactive recovery has a negligible impact on system availability, without adding more replicas. However, VM-FIT misses support for state transfer of large application state, which is essential for proactively recovering real-world applications, such as a multi-tier web application.

In this paper, we define two efficient state-transfer strategies and integrate them into VM-FIT. The first approach is based on snapshots and basic conversion routines to transform the service state from/to an abstract format. The second, a log-based variant, optimizes the transfer of large application states and requires minor changes to a service. Both protocols result in a proactive recovery scheme that permits the simultaneous recovery of all replicas with low impact on service performance. We evaluate and analyze our state transfer protocols using a micro benchmark and the three-tier benchmark system RUBiS. Thereby, we apply opportunistic N-version programming using different operating systems and different application servers. Our results show that our protocols are able to recover stateful services with a downtime of a few hundred milliseconds.

The next section provides an overview of the VM-FIT architecture, and Section 3 defines the state transfer protocols. Section 4 experimentally evaluates the current prototype with a micro benchmark and RUBiS. Section 5 discusses related work, and Section 6 concludes.

2 The VM-FIT Infrastructure

VM-FIT [RK07] implements a generic architecture for intrusion-tolerant replication of network-based services using isolated virtual machines on top of a virtual machine monitor. It supports efficient proactive recovery by booting a new replica instance (*shadow replica*) in an additional virtual machine before shutting down the old replica (*senior replica*). Most other systems that support proactive recovery require rebooting the entire physical host from a secure read-only image [CRL03, ASH07], which leads to significantly longer replica unavailability during reboot. Client/service interaction in VM-FIT is exclusively performed via request/reply network messages that can be intercepted at the network level. The remote service is deterministic and replicated using standard state-machine replication techniques.

In VM-FIT, a replica manager runs in a *privileged domain*, whereas service replicas execute in completely separated *application domains* (see Figure 1). The replica manager communicates with clients, delivers requests to replicas in total order using a group communication system, and votes on replies; it also contains the logic for proactive recovery. The virtual machine monitor and the privileged domain with the replica manager form the trusted computing base of VM-FIT that can only fail by crashing. Service replicas, which include a separate operating system, a middleware infrastructure, and a service implementation, are placed in isolated, untrusted application domains. They may fail in arbitrary (Byzan-

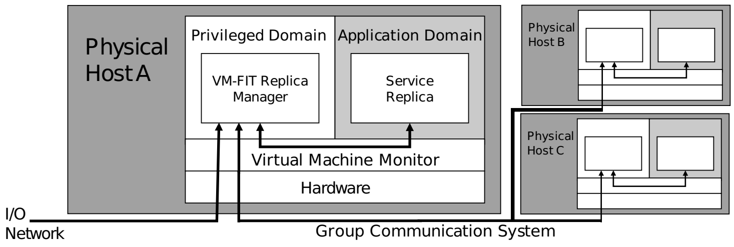


Figure 1: VM-FIT basic replication architecture

tine) ways. This *hybrid fault model* allows us to build services that tolerate any kind of failure in application domains, including malicious faults. This is a significant improvement over applications that do not tolerate intrusions at all. Furthermore, recent advances in software security (e. g., the verification of the virtual machine monitor Hyper-V [CDH+09]) show that it is becoming feasible to verify the correctness of a trusted computing base with the complexity of what we use in VM-FIT.

The basic **request processing workflow** of VM-FIT consists of the following steps: A client sends a service request, which is then received by one of the replica managers. Next, the manager transmits the request to all replica managers using the group communication system. Upon receiving the request, each replica manager passes it to the local application domain, which processes the request and returns a reply. The replica manager that is connected to the client collects the replies from the other replica managers, votes on the replies, and after having obtained $f + 1$ identical values, returns a result to the client. Note that we need only $2f + 1$ replicas to tolerate f malicious faults in the replica domains, as the group communication is part of the trusted computing base of the system. Assuming an asynchronous, reliable network, the voter eventually receives replies from all correct replicas, so it always obtains at least $f + 1$ identical replies, even if f replicas are malicious.

The replica manager provides a system component that controls **proactive recovery** in a reliable and timely way. The basic recovery strategy uses the following algorithm:

1. *Upon trigger of periodic recovery timer:*
   $\rightarrow$ Start new virtual machine *(shadow replica)* from secure image
   $\rightarrow$ After startup completion, broadcast `recovery-ready` to all replicas
2. *Upon reception of* k*th* `recovery-ready` *(*k *is the number of non-crashed nodes):*
   $\rightarrow$ Suspend request processing
   $\rightarrow$ Create local snapshot of *senior replica*
   $\rightarrow$ Resume request processing on senior replica
   $\rightarrow$ Transfer application state to new replica domain
3. *Upon completion of the state transfer:*
   $\rightarrow$ Process (buffered) client requests in shadow replica
   $\rightarrow$ Shut down senior replica

Note that as `recovery-ready` messages are totally ordered relative to client requests, all k snapshots represent the same logical point in service history. As soon as the replica manager resumes request processing, all requests are also forwarded to the shadow replica, which processes them as soon as the state transfer is completed.

The outlined algorithm delays execution of requests during snapshot generation and state transfer. For a small application state, we have shown this time to be negligible compared to replica startup [RK07]. For a large application state, however, transfer can take a significant amount of time. In this paper, we propose two optimized state transfer algorithms that tackle this problem and substitute the current simple approach.

3 State Transfer

The recovery of a stateful application's replica requires that the new replica instance receives the current service state. Assuming non-benign faults, the state transfer protocol must ensure transferring a non-corrupted state. Instead of copying the state of a single replica, VM-FIT transfers the state of multiple replicas in parallel and ensures the correctness by voting. Software diversity usually results in different internal state representations in each replica. Nevertheless, all correct replicas will maintain a consistent state from a logical point of view. This means that for voting all replicas need to convert their specific application state into a uniform external representation (*abstract state*), and vice versa. Such an approach requires explicit support for state transformation in the replica implementations.

A straight-forward way for transferring state across virtual machines is a black-box approach that considers the whole memory and disk image of a virtual machine as state. However, this view includes a lot of data that is irrelevant for the logical state of a replicated service, such as internal operating system variables. VM-FIT instead distinguishes between *system state* and *application state*, and transfers only the application state. The system state includes all state of operating system and middleware that requires no consistent replication; we assume that a correct system state can be restored from a secure code storage. In contrast, the application state, composed of *volatile state* (data stored in memory) and *persistent state* (data stored on disk), pertains to the replica implementation.

The efficiency of the state transfer highly influences service availability and performance during recovery. In the following, we present a *snapshot-based* state transfer protocol that exploits the locality provided by virtualization; a preliminary version of this protocol was published in [DKR08]. As most of the state data is transferred directly between the old and the new domain running on the same machine, it is suited for scenarios where large states have to be transmitted. For some applications, state conversion to and/or from the abstract state is costly. In these cases, a benefit can be obtained by differentiation techniques that reduce the amount of transferred data. Unfortunately, direct propagation of state updates in implementation-specific formats such as memory pages is impossible for heterogeneous replicas. To resolve this problem, we integrated an efficient *log-based* state transfer protocol into VM-FIT that copes with diverse replica implementations and yet only transfers state changes made since the previous recovery.

3.1 Snapshot-Based State Transfer

Virtualization can be harnessed for transferring state directly from senior replica to shadow replica on the same physical host. Remote replicas can validate the local state by calculating and transferring checksums of their local abstract state via the network. Unlike traditional reboot approaches, in which only a subset of the replicas can be restarted at a time in order

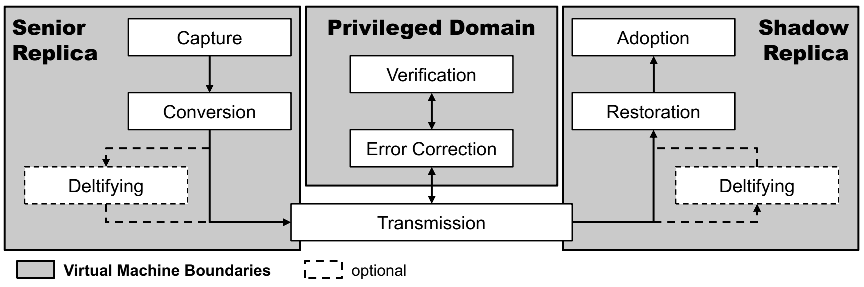


Figure 2: Conceptual phases of a state transfer

to avoid complete unavailability [SNVS06], our virtualization-based approach allows the simultaneous recovery of all replicas. The recovery process in VM-FIT starts with the creation of the shadow domain by the recovery unit (see Section 2). On each machine, the replica manager initiates and controls the subsequent snapshot-based state transfer between senior replica and shadow replica, using the following conceptual phases (see Figure 2):

Capture Phase (Capture, Conversion, optional Deltifying) Prior to the state transfer, senior replicas have to generate a consistent checkpoint. As soon as the replica manager receives a `recovery-ready` message, it suspends the distribution of client requests. After all pending requests have finished, it instructs the replica to write all replication-relevant volatile state to disk. Next, the replica manager creates a snapshot of the disk volume containing the replica application data. Thereby, volatile and persistent state of the application are both captured. After that, the replica manager attaches the snapshot to the senior domain, which now has a read-only view as well as a copy-on-write view of the same logical disk. The read-only view contains the snapshot and is used for state conversion and transfer. The copy-on-write view, which is the still attached but reconfigured replica disk, on the other hand, is used as a basis for further service execution. Thus, the senior replica can resume request processing and change the state of the copy-on-write view without affecting the snapshot. As soon as the snapshot exists, the replica manager resumes forwarding client requests. When the senior replica restarts processing, the replica manager logs these requests in order to re-execute them at the shadow replica. This way, the actual period of replica unavailability can be reduced to the process of taking the snapshot. Optionally, the senior replica can build a delta between the abstract state used for its initialization (as a result of the previous recovery cycle) and the current abstract state. Transferring the delta instead of the full abstract state reduces the data transmission time.

State Transmission and Verification (Transmission, Verification, Error Correction) The senior replica converts the application-specific state that has been captured on the snapshot volume to the abstract format. The abstract state is transferred directly to the shadow replica as a stream of bytes. A replica wrapper (a VM-FIT component fulfilling only this task during state transfer) in the shadow domain calculates block-wise checksums over the received stream data. It then transmits them to the local replica manager, which forwards them to all other replica managers. As soon as f remote checksums that match the local checksum have been received (i. e., there are $f + 1$ identical values), the replica manager signals the shadow replica 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 correct data block is requested from a remote replica manager that supplied a correct checksum.

Finishing State Transfer (optional Deltifying, Restoration, Adoption) Finally, the shadow replica converts the verified abstract state to the local application-specific representation. If deltifying has been used, the complete abstract state has to be obtained by applying the received change set on the abstract state of the previous recovery provided by the replica manager. Afterwards, the replica manager takes a snapshot that builds the basis of the following recovery. To become up to date, the replica then re-executes all post-snapshot requests, discarding the replies, and takes over the role of the senior replica. After that, the replica manager shuts down the previous senior replica.

3.2 Log-based State Transfer

The snapshot-based state transfer process is generic. It does not require any modifications to the replicated application itself, the only replica-specific requirements are conversion routines for the state. However, state conversion and restoration can take a significant amount of time, thereby delaying replica handover. Thus, we propose another state transfer variant that performs conversion, adoption, and deltifying during normal operation mode and omits state capturing operations. It requires minor changes to the application code.

The core idea is to constantly log all state changes in the abstract state format and apply them during recovery to a pre-initialized replica. This approach requires the identification of certain locations in an application where standardized write operations are performed. For web-based multi-tier applications, an appropriate location is the transition between application layer and data layer, as most of the processing has already been performed. At this point, results are usually written to database, typically using SQL.

We do not directly log the SQL operations of the application. These operations might be crafted maliciously in order to exploit a vulnerability of one of the database replicas. Instead, we retrieve the effective write set of an operation from the database and transform it into a state update operation and append this operation to a log. The log is verified in the same way as the snapshot is verified in the snapshot-based approach. If a malicious request corrupts the state of a database replica, this problem is detected during the verification phase and subsequently corrected by transferring a validated remote log. Additionally, if the same data is changed multiple times, we use online log reduction; that is, during state transfer we only consider the latest update.

In the shadow replica, the log is re-executed on the database that has been initialized using the verified snapshot of the previous recovery, leading to a verified up-to-date replica state. After the replay, but prior to processing further requests, the replica manager takes a file-system snapshot of the current state. This snapshot forms the secure basis to initialize the shadow replica of the next recovery round. This way, we avoid costly conversion operations without changing the processes of state transmission and finishing state transfer.

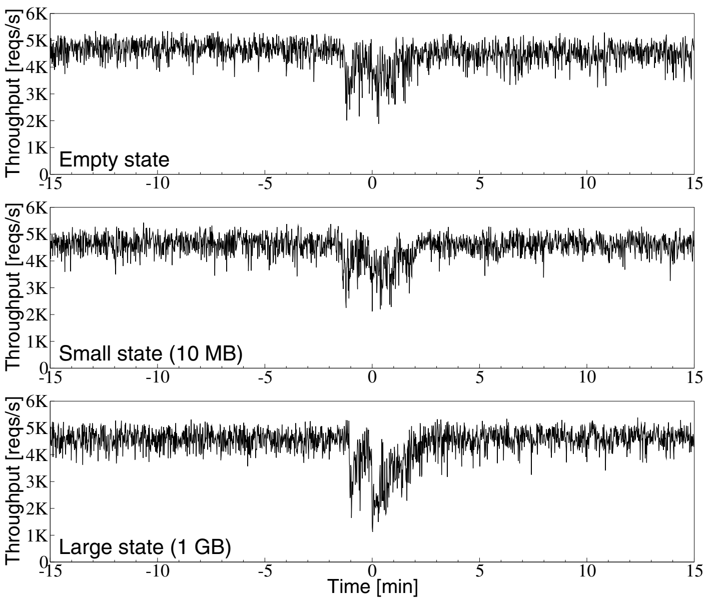


Figure 3: Micro benchmark with snapshot-based proactive recovery for different state sizes.

4 Evaluation

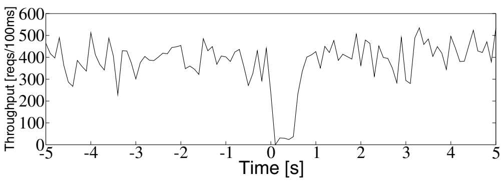
This section demonstrates that proactive recovery in VM-FIT using our state-transfer algorithms causes only negligible downtime for stateful services. We analyze the snapshot-based state transfer with a micro benchmark and the log-based approach with a complex three-tier application. The evaluation uses a cluster of four hosts (Intel Core 2 CPU, 2.4 GHz, 2 GB RAM, Xen 3.1 hypervisor), connected with 1 Gb/s Ethernet. One host simulates clients, the other three hosts execute VM-FIT with identical configurations in the privileged domain and different operating systems in the application domain (Debian Linux, NetBSD, and OpenSolaris). Snapshots of replica application states are created using LVM [LVM10], a logical volume manager for Linux.

4.1 Micro Benchmark

The micro benchmark used to evaluate the snapshot-based state transfer uses 500 clients concurrently retrieving and modifying records in a Java service that implements a simple database. We run the benchmark with empty, small (10 MB) and large (1 GB) application states. Each run takes 30 minutes and includes a recovery after 15 minutes. Furthermore, there is one comparison test run without recovery. Figures 3 and 4 present the results of the micro benchmark. All graphs are synchronized at snapshot creation time ($t = 0$). The measurements show that there are two main factors degrading performance during the recovery process: First, setting up the shadow domains puts additional load on each of the physical hosts. The complete startup usually takes about one minute and results in a 10-20% throughput decrease. This number is independent of state size.

application state size	*state transfer throughput*	*overall throughput*
empty	3459 req/s	4555 req/s
10 MB	3828 req/s	4547 req/s
1 GB	3078 req/s	4431 req/s
no recovery	-	4620 req/s

(a) Average realized throughput of a proactive-recovery micro benchmark for variable state sizes.



(b) Detailed characteristics of a proactive-recovery micro benchmark for a state size of 10 MB during snapshot creation.

Figure 4: Characteristics of a snapshot-based proactive-recovery micro benchmark

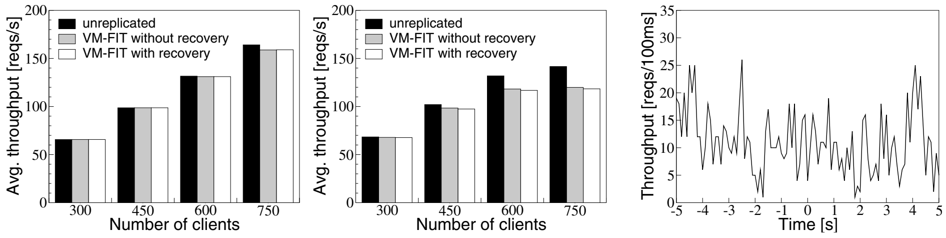
The second factor influencing throughput during recovery is state transfer. In contrast to shadow domain startup, this recovery step is dominated by the characteristics of an application. The results of the benchmark confirm that moving a larger state not only takes more time, but also impacts performance: Transferring a 10 MB state, for example, decreases throughput for 120 seconds by an average of 17%, whereas recovery with 1 GB of state leads to a temporary performance drop of 33% during 169 seconds. There are two reasons for the increase in throughput degradation: More data has to be copied and checked, and an extended transmission and verification phase leads to more requests being buffered, which then have to be re-executed afterwards. As state transfer in the empty-state scenario is limited to the short period of costly snapshot creation, a rather low throughput of 3459 requests per second can be observed during that period. Combining shadow replica startup and state transfer, the results show that proactive recovery in the micro benchmark is very cheap: For the small application state scenario, the average throughput during the test is 4547 req/s which is only 2% below the result of the no-recovery run (4620 req/s). As discussed above, transferring a bigger state further decreases performance. However, the costs for the recovery of a large state only add up to throughput degradation of 4%.

The results of the micro benchmark show no observable service downtime during any stage of the recovery process. Additional measurements conducted with higher sampling rates outline the actual service disruption. Figure 4b presents the throughput development around the snapshot of a 10 MB state. For an interval of 600 milliseconds the throughput drops significantly, with a maximum continuous service downtime of 130 milliseconds.

4.2 Rice University Bidding System (RUBiS)

The Rice University Bidding System (RUBiS) is a web-based auction system built to benchmark middleware infrastructures under realistic load situations [RUB10]. We replicated RUBiS in its Java servlet variant using VM-FIT. In the snapshot-based state transfer, retrieving/restoring the state from/to a database (via plain SQL interface) during the conversion/restauration phase would take several minutes. Therefore, RUBiS is an excellent candidate for the log-based state transfer, as logging state changes during normal operation circumvents the need to retrieve a complete abstract state.

We implemented our approach by intercepting calls at the JDBC layer that represents the interface between application layer and data layer. Each modifying operation is followed by



(a) Realized throughput for the plain service and VM-FIT with and without proactive recovery for database sizes of 100 MB (left) and 1 GB (right).

(b) Detailed characteristics of the state-capture phase (1 GB database).

Figure 5: Characteristics of a log-based proactive-recovery RUBiS benchmark

a read of the affected table entry whose result is then logged in plain SQL[1]. The generated log contains the (differential) abstract state to be distributed and verified during state transfer. Shadow replicas are initialized using a verified snapshot of the previous recovery. Then, they re-execute the operations from the verified log on the database. Before starting to execute client requests, the replica manager takes a file-system snapshot of the database, which serves as the verified snapshot used in the next recovery round.

Besides using different operating systems, additional heterogeneity is achieved by different web/application servers (Jetty 6.1.12rc1 on the Solaris and BSD replicas, Apache Tomcat 5.5.27 in the Debian guest domain). All clients are simulated by the RUBiS client emulator processing the default transition table (which approximates the behaviour of human clients browsing an auction website). Each test run starts with an identical initial state taken from a database dump provided on the RUBiS web site [RUB10]. It includes data sets of about one million users and more than five million bids. The total size of the application state, based on the original SQL dump, exceeds one gigabyte. In addition, we conduct a series of tests with a downscaled database of about 100 MB. Our evaluation compares the unreplicated RUBiS benchmark to VM-FIT with and without proactive recovery, varying the number of clients from 300 to 750. Clients perform a 30 minute runtime session. Proactive recovery is triggered after 15 minutes.

Figures 5a shows the overall throughput results for the RUBiS benchmark. For the small database, the average throughput of VM-FIT without recovery is within 3.2% of the throughput achieved by the unreplicated benchmark, with a maximum of 158.8 requests per second at 750 clients. The results for the tests where VM-FIT proactively recovers using the log-based state transfer are basically equal (less than 0.1% deviation). Using the large database, 750 clients saturate the RUBiS service on our machines due to database operations being more costly, leaving VM-FIT without recovery with an average throughput of 119.9 requests per second. Here, VM-FIT with recovery has a throughput of 118.4 requests per second, a decrease of 1.3%. These numbers show that the log-based proactive recovery has only little effect on the overall service performance.

[1]More efficient approaches for retrieving the write set from a database could also be used [SJPPnMK06], but this simple approach was suitable in the context of our evaluation.

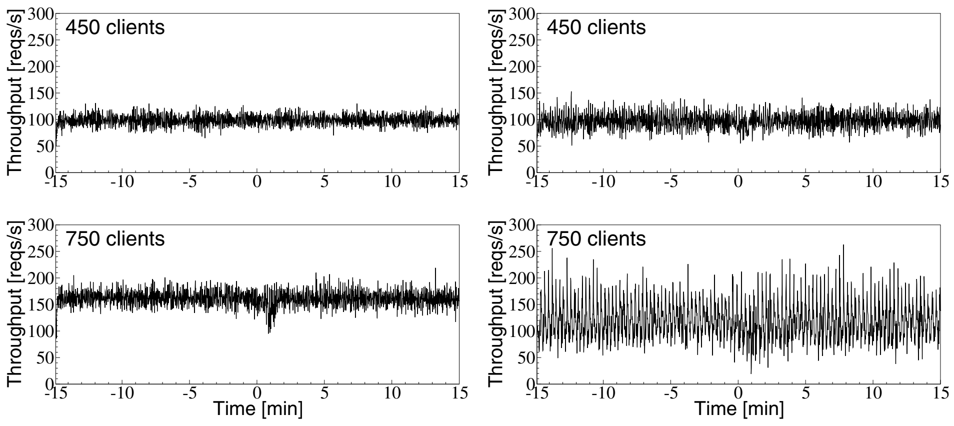


Figure 6: RUBiS benchmark with recovery for database sizes of 100 MB (left) and 1 GB (right).

Figure 6 (we omit the results for 300 and 600 clients due to limited space) gives a more detailed view of the impact of the log-based state transfer. A major difference in comparison to the micro benchmark results is that shadow domain startup has almost no influence on throughput. Note that this circumstance is due to the different application characteristics of the RUBiS service; it is not related to the log-based state transfer. However, applying the log-based approach has a great influence on other recovery phases: Instead of transmitting and verifying the whole application state of 1 GB (in case of the small database: 100 MB), only modified database entries are transferred. For 750 clients, they add up to 2.0 MB (2.1 MB); about 0.2% (2.1%) of state size. As a consequence, state transfer is done in 95 seconds (50 seconds), temporarily decreasing throughput by 11.2% (10.7%).

The most important benefit of the log-based approach is its implementation of the state capture phase. Instead of expensively creating a snapshot of the application state, the log that records state changes is just truncated. Figure 5b shows that this procedure is executed without disrupting the service.

5 Related Work

System recovery is a popular strategy to cope with bugs and intrusions. Recovery-Oriented Computing (ROC) [CBFP04] and Microreboots [CKF$^+$04] represent approaches to cure potential faults by rebooting fine-grained components. Both focus on counteracting hard-to-detect faults that over time cause service degradation, such as memory leaks. Self-Cleansing Intrusion Tolerance [ASH07] targets cleaning intrusions by rebooting services executed in a virtual machine. To prevent service disruption, a backup instance immediately takes over once the primary service instance is rebooted. Similar to this work, the Rx [QTZS07] system goes one step further by modifying the environment after a software failure. This enables Rx to handle certain kinds of deterministic bugs that otherwise would lead to repetitive reboots because of recurring faults. None of these systems is able to handle state corruptions.

The work by Sousa et al. [SNVS06] uses a trusted computing base on the basis of a virtual machine monitor and a trusted virtual machine for proactive and reactive recovery in the context of BFT replication. The trusted computing base only provides functionality for recovery; there is no support for state transfer at the virtualization layer. Our approach is similar to this architecture in the sense that it uses the virtual machine monitor to implement a trusted computing base, but we focus on stateful recovery.

The problem of state transfer has been addressed by the BFT protocol of Castro and Liskov [CL02]. They recognize that efficiency of state transfer is essential 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 [CRL03] proposes abstractions that capture parts of the state in an abstract format for use during state transfer in a heterogeneous BFT system. In addition, BASE provides wrapper functions for handling non-determinism that could also be added to our architecture. Our system also uses the concept of an abstract state, but we define two different approaches: first, a generic snapshot-based state transfer, and second, a log-based state transfer that requires slight changes to the replicated application, but reduces overhead. The approach of creating a differential state used in BASE [CRL03] is not suitable for RUBiS, as some requests involve multiple related database operations. The need to reproduce them outside the application would lead to a duplication of great parts of the database layer. Additionally, we exploit virtualization for minimizing state transfer time. Sousa et al. [SNVS06] 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 considers stateful services and reduces unavailability during recovery by performing most of the recovery in parallel to normal system operation on the basis of virtualization technology.

This work expands on preliminary results proposing efficient state transfer for the proactive recovery of stateful services published at the WRAITS workshop [DKR08] by proposing log-based state transfer, a more detailed design as well as extended evaluation results.

6 Conclusion

We have presented two protocols for efficient state transfer between heterogeneous replicas in an intrusion-tolerant replication system with proactive recovery. Snapshot-based state transfer only requires the provision of conversion routines from/to an abstract state format. Creating a new replica instance using virtualization technologies and utilizing copy-on-write techniques for fast generation of atomic state checkpoints enables to continue service provision while the state is transferred. Additionally, we proposed log-based state transfer as a further extension for scenarios where enough knowledge of the service is available to employ a differential state transfer. The results of the micro benchmark and the RUBiS use-case illustrate the efficiency of proactive recovery in VM-FIT. The evaluation shows that there is only a negligible service disruption during recovery. This allows building BFT systems which continuously run stateful applications without needing additional replicas during proactive recovery. Although state size is not a crucial factor regarding service availability, it nevertheless influences performance. Therefore, the abstract state transferred should be limited to information that is required to initialize the application in the shadow replica. As shown with the RUBiS evaluation, log-based state transfer is an excellent way to approach this task as it reduces the transferred abstract state to the essential updates. This way, we can by simultaneously recovering all replicas of a complex three-tier application with only 2-6% decrease in the overall throughput.

References

[AC77] A. Avižienis and L. Chen. On the Implementation of N-Version Programming for Software Fault Tolerance During Execution. In *Proc. of the IEEE Computer Software and Applications Conf.*, pages 149–155, 1977.

[ASH07] D. Arsenault, A. Sood, and Y. Huang. Secure, Resilient Computing Clusters: Self-Cleansing Intrusion Tolerance with Hardware Enforced Security (SCIT/HES). In *Proc. of the 2nd Int. Conf. on Availability, Reliability, and Security*, pages 343–350, 2007.

[CBFP04] G. Candea, A. B. Brown, A. Fox, and D. Patterson. Recovery-Oriented Computing: Building Multitier Dependability. *Computer*, 37(11):60–67, 2004.

[CDH$^+$09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd Int. Conf.*, pages 23–42, 2009.

[CKF$^+$04] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation*, pages 31–44, 2004.

[CL02] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[CRL03] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Trans. on Computer Systems*, 21(3):236–269, 2003.

[DKR08] T. Distler, R. Kapitza, and H. P. Reiser. Efficient State Transfer for Hypervisor-Based Proactive Recovery. In *Proc. of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems*, pages 7–12, 2008.

[GP07] I. Gashi and P. Popov. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Trans. Dep. Secure Comp.*, 4(4):280–294, 2007.

[LVM10] LVM: Logical Volume Manager. http://sourceware.org/lvm2/, 2010.

[QTZS07] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Trans. on Computer Systems*, 25(3):7, 2007.

[RK07] H. P. Reiser and R. Kapitza. Hypervisor-Based Efficient Proactive Recovery. In *Proc. of the 26th IEEE Int. Symp. on Reliable Distributed Systems*, pages 83–92, 2007.

[RUB10] RUBiS: Rice University Bidding System. http://rubis.ow2.org/, 2010.

[SBO08] P. Sousa, A. N. Bessani, and R. R. Obelheiro. The FOREVER Service for Fault/Intrusion Removal. In *Proc. of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems*, pages 13–18, 2008.

[SJPPnMK06] J. Salas, R. Jiménez-Peris, M. Patiño Martínez, and B. Kemme. Lightweight Reflection for Middleware-based Database Replication. In *Proc. of the 25th IEEE Int. Symp. on Reliable Distributed Systems*, pages 377–390, 2006.

[SNVS06] P. Sousa, N. Neves, P. Veríssimo, and W. H. Sanders. Proactive Resilience Revisited: The Delicate Balance Between Resisting Intrusions and Remaining Available. In *Proc. of the 25th IEEE Symp. on Reliable Distributed Systems*, pages 71–82, 2006.