SPARE: Replicas on Hold

Tobias Distler[1], Rüdiger Kapitza[1], Ivan Popov[1], Hans P. Reiser[2], Wolfgang Schröder-Preikschat[1]
[1]Friedrich-Alexander University Erlangen-Nuremberg [2]Universidade de Lisboa
{distler,rrkapitz,popov,wosch}@cs.fau.de hans@di.fc.ul.pt

Abstract

Despite numerous improvements in the development and maintenance of software, bugs and security holes exist in today's products, and malicious intrusions happen frequently. While this is a general problem, it explicitly applies to web-based services. However, Byzantine fault-tolerant (BFT) replication and proactive recovery offer a powerful combination to tolerate and overcome these kinds of faults, thereby enabling long-term service provision. BFT replication is commonly associated with the overhead of $3f + 1$ replicas to handle f faults. Using a trusted component, some previous systems were able to reduce the resource cost to $2f + 1$ replicas. In general, adding support for proactive recovery further increases the resource demand. We believe this enormous resource demand is one of the key reasons why BFT replication is not commonly applied and considered unsuitable for web-based services.

In this paper we present SPARE, a cloud-aware approach that harnesses virtualization to reduce the resource demand of BFT replication and to provide efficient support for proactive recovery. In SPARE, we focus on the main source of software bugs and intrusions; that is, the services and their associated execution environments. This approach enables us to restrict replication and request execution to only $f + 1$ replicas in the fault-free case while rapidly activating up to f additional replicas by utilizing virtualization in case of timing violations and faults. For an instant reaction, we keep spare replicas that are periodically updated in a paused state. In the fault-free case, these passive *replicas require far less resources than active replicas and aid efficient proactive recovery.*

1. Introduction

Malicious intrusions remain a problem for web-based services despite a number of improvements in the development and maintenance of software. Byzantine fault-tolerant (BFT) replication is a key technology for enabling services to tolerate faults in general and software-induced problems in particular. However, BFT replication involves high overhead, as it requires $3f + 1$ replicas to tolerate f faults [7]. Lately, a debate has been started how and if Byzantine fault tolerance and the provided results will be adopted by the industry [12, 27]. One of the outcomes was that the enormous resource demand and the associated costs seem to be too high for many scenarios to legitimate the improved fault-tolerance properties, and that this is the reason why the use of BFT is only accepted for critical infrastructures and services. This fact becomes even more severe if long-term service provision is targeted, which demands for proactive recovery to neutralize faults and tends to further increase the resource demand due to the temporary need of additional replicas during recovery [38, 43].

At the same time, we see web-based services playing a more and more important role in our everyday life. This becomes evident the minute these services are no longer accessible due to faults and, even worse, when faulty results are provided to users. Web-based services are typically confronted with much stronger economic constraints than critical infrastructures. As a consequence, resource-intensive techniques such as BFT replication are not an option. On the other hand, with economical constraints impacting software quality and provision management, web-based services are easy targets for worms, viruses and intrusions. Accordingly, our general goal is to reduce the resource costs of BFT replication to lower the entry barrier for this technique and to make it widely applicable, especially for web-based services. A pragmatic approach to achieve this goal is to make the assumption that software bugs and intrusions are limited to certain network-exposed parts of the system that are complex and subject to innovation. In contrast, the remaining subsystem is considered as intrusion-free and trusted. The utilization of a trusted subsystem allows reducing the resource requirements to as few as $2f + 1$ replicas [11, 13, 38, 49]. However, we consider this still too much to be applicable in practice.

Accordingly, we investigated the fact that there is a trend to run web-based services on top of a virtualization platform that provides basic management of virtual machines. Such an environment can be found in cloud-computing infrastructures like Amazon EC2 [1], Eucalyptus [33], OpenNebula [42], and IBM CloudBurst [22], but also in environments that are self-managed (e. g., to consolidate hardware). There is an ongoing development to make hypervisors and associated minimal runtime environments supporting virtualization small and trustworthy (e. g., using verification) [26, 29, 32, 45]. Still, software executed inside of a virtual machine can fail maliciously. This is not likely to change in the near future due to the rapid development in the application and middleware sector and the associated complexity.

Based on these observations, we present *SPARE*, a system that focuses on service-centric Byzantine fault tolerance and requires only $f+1$ active replicas during fault-free execution. This number is the minimum amount of replicas that allows the detection of up to f replica faults when utilizing a trusted subsystem. When SPARE suspects or detects faulty behavior of some of the replicas, it activates up to f additional *passive replicas*. A replica is suspected to be faulty if it either produces faulty results that are inconsistent with the results provided by other replicas, or if its response time exceeds a threshold; the latter is needed as attackers might delay replies to prevent further execution.

Virtualization provides an ideal basis for dynamically activating additional replicas, as it offers means for rapid activation and deactivation of virtual machines. In the context of SPARE, a passive replica is a paused virtual machine (i. e., it is not scheduled by the hypervisor and therefore does not consume CPU) that is periodically unpaused and updated. This saves resources as providing and applying state updates is usually less resource demanding than request execution and read-only requests only need be executed by active replicas. Albeit this already results in substantial resource savings, passive replicas provide another great benefit: If state updates are agreed on by a majority of active replicas, passive replicas can be used as a basis for resource-efficient proactive recovery. Furthermore, SPARE replicas are able to agree on updates on the granularity of a request, offering additional means for fault detection.

We implemented and evaluated SPARE using a common hypervisor and a web-based three-tier benchmark. SPARE needs 21-32% less resources (CPU, memory, and power) than a traditional trusted-subsystem approach and only demands 6-12% more than plain crash-stop replication. We consider this a substantial step towards making BFT systems affordable for a broad spectrum of web-based services.

In this paper, we present the SPARE architecture (Section 2) and introduce passive replicas in BFT (Section 3). Section 4 describes fault handling and proactive recovery in SPARE. Section 5 discusses how to integrate SPARE with an infrastructure cloud. Section 6 presents an extensive evaluation of SPARE's resource footprint. Section 7 gives some final remarks on the practicality of SPARE. Section 8 presents related work, and Section 9 concludes.

2. Architecture of SPARE

We build SPARE as a generic architecture for application-centric Byzantine fault-tolerant replication using virtualization. This section presents the system model as well as the basic architecture of SPARE.

2.1. System Model

Our architecture makes the following assumptions:

- Clients exclusively interact with the remote service using request/reply network messages. All client–service interaction can be intercepted at the network level.
- The remote service can be modeled as a deterministic state machine. This property allows replication using standard state-machine–replication techniques.
- Service replicas, including their operating system and execution environment, may fail in arbitrary (Byzantine) ways. At most $f \leq \lfloor \frac{n-1}{2} \rfloor$ of the n active and passive replicas may fail within a recovery round.
- Apart from the service replicas, the remaining system components, including the hypervisor and a trusted system domain, fail only by crashing.
- Host crashes can be reliably detected within a bounded time span in minimal configurations of $f+1$ nodes.

While the first three assumptions are common for systems featuring BFT replication and proactive recovery, the last two assumptions are specific to SPARE. Assuming that a non-trivial part of the system can only fail by crashing is critical. However, SPARE does not target to provide Byzantine fault tolerance for a whole system but only for a replicated service and its directly associated execution environment; that is, SPARE goes beyond plain crash-stop replication, but its ability to tolerate malicious faults is limited compared to pure BFT replication.

The assumption of a crash-only subsystem is supported by recent progress in the domain of minimal and trustworthy operating-system and virtualization support [26, 29, 45]. While the reliable detection of a crashed node is a strong assumption in the general context of distributed systems, we assume SPARE to be typically used in closely-connected environments. In this specific setting, the detection of crashed nodes within a finite time is practicable and common [14, 20]. If not stated otherwise, we present SPARE in the context of a closely-connected environment in this paper. In such an environment, the network is typically assumed to be secure. Please refer to Section 7 for a discussion of extensions and modifications (e. g., secure authenticated channels) that allow SPARE to be applied in a geographically distributed deployment; for example, across different data centers.

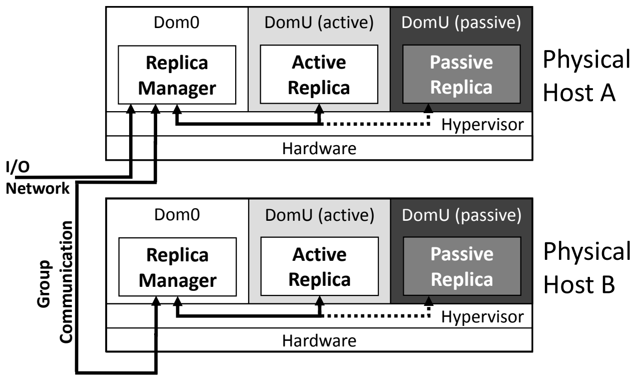


Figure 1. Minimal SPARE replication architecture featuring passive replication and proactive recovery that can tolerate a single fault.

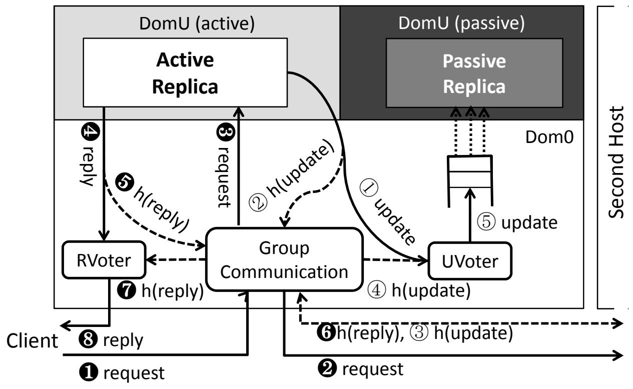


Figure 2. Detailed steps of processing a client request (❶-❽) and the corresponding state update (①-⑤) in SPARE.

2.2. Basic Architecture

SPARE is based on a hypervisor that enforces isolation between different virtual machines running on the same physical host. We distinguish between a privileged virtual machine (*Dom0*) that has full control over the hardware and is capable of starting, pausing, unpausing, and stopping all other hosted virtual machines (*DomU*s). Such a division of responsibilities is common for hypervisor-based systems [5, 45, 47]. In the context of this paper we use the terminology of Xen, as our prototype is based on this particular hypervisor. However, as the SPARE architecture is generic and does not rely on any Xen-specific functionality, other hypervisors could be used instead.

In SPARE, a *replica manager* is running within the privileged Dom0, while each service replica is executed in a completely separated application domain; that is, a DomU with guest operating system, middleware infrastructure, and service implementation (see Figure 1). The replica manager is composed of basic support for handling client connections, communication support for distributing totally-ordered requests to all replicas, and a proactive-recovery logic. In addition, the replica manager includes a custom voting component enabling on-demand activation of passive replicas, mechanisms for handling state updates (see Section 3), and support for replica cloning (see Section 4.2); the latter is needed for proactive recovery (see Section 3.3).

In combination with hypervisor and Dom0, the replica manager forms the trusted computing base of SPARE that only fails by crashing; service replicas in isolated application domains are not trusted. This *hybrid system model* allows coping with any kind of failure in application domains, including random non-crash faults and intentional malicious faults.

Figure 1 presents a minimal setting comprising two physical machines, each hosting one active and one passive replica. This configuration can tolerate one fault, either at the service replica, at the trusted-component level, or at the granularity of a whole physical machine; in general, SPARE requires $f + 1$ physical machines to tolerate f faults.

2.3. Basic Request Processing

The left half of Figure 2 shows the principle steps (❶-❽) to process a client request in SPARE. The configuration in this example comprises two physical machines and is therefore able to tolerate a single Byzantine fault in a service replica. Note that all components shown in Dom0 (i.e., `RVoter`, `UVoter`, state-update buffer, and group-communication endpoint) are subcomponents of the replica manager; for clarity, we omit the subcomponents for replica activation and proactive recovery. Furthermore, we assume that communication between clients and the service is not encrypted. If this is demanded, we expect to have a proxy in front of the service, which is transparent to our infrastructure, that handles authentication and link-level encryption.

When a client sends a request to the service, the request is received by one of the two replica managers ❶ and forwarded to the `Group Communication` ❷. As both nodes can receive client requests, the group communication imposes a total order on requests; this is necessary to preserve determinism in the service replicas. After ordering, a request is handed over to the local active replica to be processed ❸. Next, the associated reply is fed into the reply voter (`RVoter`) ❹, which decides on the result of the replicated execution. Note that this voting only takes place at the replica manager that initially received the client request, the other replica manager provides a hash over the reply of the other active replica ❺❻❼. When the local reply matches the hash (i. e., the `RVoter` has collected $f+1$ matching replies/hashes), the execution result is correct and can safely be forwarded to the client ❽. In case of a mismatch or a missing reply, additional measures are necessary to determine the correct result (see Section 4).

3 Passive Replicas

In the context of the crash-stop failure model, the use of *passive replicas* is a well-known concept to reduce replication overhead, as passive replicas only receive state updates instead of executing requests [6]. If a master replica crashes, a secondary passive replica takes over.

Common approaches for BFT replication execute all state-altering requests on every replica to ensure high availability [7, 8, 9, 13, 38, 49]. In order to reduce resource consumption in SPARE, we introduce the idea of passive replicas in the context of BFT replication. In a BFT system, we cannot rely on a single master. Instead, we need at least $f+1$ active replicas to detect up to f faults; the remaining $f+1$ replicas in SPARE are initially passive.

3.1. Activation

If at least one of the $f+1$ active replicas provides a different reply than the other active replicas (or none at all), additional (i. e., passive) replicas also execute the pending request to provide the reply voter with additional replies (see Section 4). However, passive replicas first have to be activated. To minimize service disruption, activation of passive replicas must be rapid, as the service cannot proceed as long as it takes to decide on the result of a pending request.

In SPARE, virtual machines hosting passive replicas are put into paused mode. In this mode, a passive replica can be activated instantly when needed, but until then only consumes a small amount of resources (see Section 6.4.2). Using Xen, for example, a paused virtual machine remains in memory but does not receive any processor time. On the activation of a passive replica, the replica manager unpauses the corresponding virtual machine, which only takes a few hundred milliseconds (130 ms in our testbed) for a standard Linux virtual machine and a vanilla version of Xen. Note that additional resources could be saved by booting the passive replica from scratch. However, this typically takes tens of seconds and would therefore cause a significant service downtime in case of a replica fault.

3.2. State Updates

In order to process a pending request after being activated, passive replicas must have the latest application state available. SPARE uses periodic state updates to get passive replicas up to speed. These updates have to be free of faults, otherwise they could contaminate the clean state of the passive replicas, which would render them useless for the task of providing additional replies to decide on the result of a pending request.

The right side of Figure 2 outlines the basic workflow for collecting state updates. Having executed a state-modifying request, the active replica provides the update voter (`UVoter`) with a state update ①; a hash over the update is propagated to the other replica manager via group communication ②③. With the other node acting accordingly, in the absence of faults, each update voter is provided ①④ with enough hashes to prove the state update correct. Note that both replica managers vote on the state update independently. When an update voter has received enough matching hashes for a state update, it enqueues the update in a temporary buffer ⑤. If an update voter receives differing hashes or is not able to collect enough hashes within a certain period of time, the replica manager activates the local passive replica and (re-)processes the specific request (see Section 4).

When the number of voted updates reaches a certain limit k (e. g., in our evaluation we use $k = 200$), the replica manager temporarily wakes up the local passive replica and applies the buffered state updates. Next, the updated replica is paused again and the buffer is emptied. Note that these periodic updates of passive replicas reduce the overhead for updating on the occurrence of faults. When a passive replica is activated to tolerate a fault, only state updates since the last periodic update have to be executed to prepare the replica.

In the context of applications addressed by SPARE, applying state updates is usually more efficient than executing the actual requests. For example, in a multi-tier web-service application, it takes longer to process a submitted form with data (request) than adding an entry in a relational database (state update). Furthermore, the creation of a state update for every modifying request is feasible as such requests only represent a small fraction of the typical workload [17]. Retrieving state updates from an active replica can be done in multiple ways, but typically some form of write-set capturing is applied. For relational databases, for example, this can be done using vendor-specific APIs, triggers, or simply additional custom-tailored queries [15, 39, 40].

3.3. Supporting Proactive Recovery

With faults accumulating over time, the number of faults may eventually exceed the fault-tolerance threshold of a BFT system. Therefore, we consider proactive recovery [8, 38] an important technique to provide long-term services. Proactive recovery periodically initializes replicas with a correct application state all non-faulty replicas have agreed on. This way, a replica is cleaned from corruptions and intrusions, even if they have not been detected. As a consequence, proactive recovery basically allows to tolerate an unlimited number of faults as long as at most f faults occur during a single recovery period.

In SPARE, passive replicas are used for proactive recovery as they already have the latest correct application state available (see Section 3.2). Instead of pausing a replica after applying the periodic state updates, the replica manager clones either the entire replica or only the relevant service state. In any case, this builds the basis for a future passive replica, while the current passive replica is promoted to be an active replica; that is, the replica manager hands over request execution to the former passive replica, and shuts down the old active replica (see Section 4.2 for details). This way, a (potentially) faulty replica is efficiently replaced with a clean one.

4 Fault Handling and Proactive Recovery

In the absence of faults, SPARE saves resources by relying on only a minimal set of $f+1$ active service replicas and by keeping all other service replicas in a resource-efficient passive mode. This section describes how SPARE makes use of those passive replicas to handle suspected and detected faults, as well as node crashes. Furthermore, passive replicas play a key role during proactive recovery.

4.1. Fault Handling

During normal-case operation, SPARE's $f+1$ active service replicas provide enough replies to prove a result correct. However, in case of faults, up to f passive replicas must step in to decide the vote. Note that, in the following, we focus on the handling of faulty service replies, faulty state updates are handled in the same manner.

4.1.1. Basic Mechanism

When a replica manager detects a situation that requires replies from additional replicas (e. g., a voting mismatch), the replica manager distributes a $\langle \texttt{HELP\_ME}, s, n \rangle$ request via group communication, indicating the number of additional replies n to be provided for the request with sequence number s. Every replica manager i that has a non-faulty passive replica available reacts by sending an $\langle \texttt{I\_DO}, s, i \rangle$ acknowledgement via group communication. If the acknowledgement of i is among the first n acknowledgements received for s, the passive replica of i has been selected to provide an additional reply; the group communication ensures that all replica managers see the acknowledgements in the same order. Note that each request for additional replies is protected by a timeout that triggers another HELP_ME request with an updated value for n, if not enough (correct) replies become available within a certain period of time.

When the local passive replica has been selected to provide an additional reply, a replica manager performs the following steps: First, it activates the passive replica by unpausing its virtual machine. Next, the replica manager applies all state updates remaining in the state-update buffer (see Section 3.2). Finally, the replica manager executes the request with sequence number s on the (now former) passive replica and forwards the reply to the replica manager that issued the HELP_ME request.

In order to save resources, replica managers usually deactivate former passive replicas after fault handling is complete. However, in some cases (e. g., node crashes), a replica manager may decide to keep the passive replica permanently activated; this decision may, for example, be based on the frequency of HELP_ME requests, using an exponential-backoff mechanism.

4.1.2. Detected Faults

A replica manager detects a faulty reply through a voting failure. If at least one of the $f+1$ replies differs from the other replies, the replica manager sends a $\langle \texttt{HELP\_ME}, s, n \rangle$ request, with $n = f + 1 - m$ and m being the maximum number of matching replies. At this point, it is most likely that those m matching replies are correct; however, this is not guaranteed. In case the m replies turn out to be incorrect, the replica manager sends additional HELP_ME requests. When the replica manager has finally collected $f+1$ identical replies, thanks to the additional replies being provided by former passive replicas, it forwards the correct result to the client. Note that using reply hashes, the replica manager may not have obtained a correct full reply. In this case, it orders the full reply from a replica manager that provided a correct reply hash.

Having learned the correct reply, the replica manager is able to determine which service replicas delivered incorrect replies; that is, which replicas are faulty. As a consequence, the replica manager distributes a $\langle \texttt{CONVICTED}, r \rangle$ notification via group communication for every faulty replica r. When a replica manager receives a CONVICTED notification for a local replica, it destroys the faulty active replica and promotes the local passive replica to be the new active service replica.

4.1.3. Timing Violations

Besides providing an incorrect result, a faulty replica may not even respond at all, leaving the replica manager with too few replies to decide the vote. To identify such situations, the replica manager starts a timer on the reception of the first reply to a request. The timer is stopped when

$f + 1$ matching replies are available. On timeout expiration, the replica manager sends a $\langle \texttt{HELP\_ME}, s, n \rangle$ request, with $n = f + 1 - a$ and a being the number of replies already available. As a consequence, the basic fault-handling mechanism leads to additional replies being provided that enable the replica manager to determine the correct result.

We assume the timeout to be application or even request dependent. In most cases, finding such a timeout is not a problem as benchmarks may give information about request durations under high load. Furthermore, if the timeout is triggered due to false prediction, this only introduces overhead but does not impact the functionality of the system.

In case of a timing violation, a replica manager does not send a CONVICTED notification. Instead, it sends a $\langle \texttt{SUSPECTED}, r \rangle$ notification for every service replica r that failed to deliver a reply; on the reception of a threshold number of SUSPECTED notifications (e. g., 100), a replica manager only pauses the affected active replica. Note that in the absence of a voting failure (i. e., a proof that a replica is actually faulty), a false suspicion could otherwise lead to the destruction of a non-faulty replica, endangering the liveness of the whole system. After being paused, the former active replica is treated like a passive replica; that is, it is periodically updated by applying state updates. Therefore, the former active replica is still able to provide replies to assist in solving (possible) future voting mismatches. However, if the former active replica is actually faulty, the next proactive recovery will cure this problem.

4.1.4. Node Crashes

On the detection of a node crash x, each replica manager i that has a non-faulty passive replica available sends a $\langle \texttt{COMPENSATE}, x, i \rangle$ message via group communication, indicating the offer to compensate the loss of the active replica on the crashed node. The first COMPENSATE message that is delivered by the group communication determines, which node is selected to step in. At this node, the replica manager activates and prepares the passive replica (i. e., it applies the remaining state updates) and starts regular request execution. Note that in case the crashed node hosted two active service replicas, for example due to an earlier crash of another node, SPARE's mechanism for handling timing violations ensures that an additional passive replica steps in.

4.2. Proactive Recovery

In the absence of detected faults or timing violations, a passive replica R_p only receives agreed state updates generated since the last proactive-recovery round $p - 1$. Therefore, it is safe to use R_p to replace the current active replica during proactive recovery p. Prior to that, each replica manager sets up a new passive replica R_{p+1} (i. e., the next generation) and directly transfers the state between R_p and R_{p+1}.

We assume that replicas might be heterogeneous to avoid common faults (see Section 7). In this case, a BASE [9]-like approach may be implemented to transfer the state [15]. Such an approach requires replicas to not only manage the service state in their implementation-specific representation, but to also maintain the service state in an abstract format that is supported by all replicas and used for state transfer. At the technical level, the abstract state can be rapidly transferred by exchanging a file-system snapshot S.

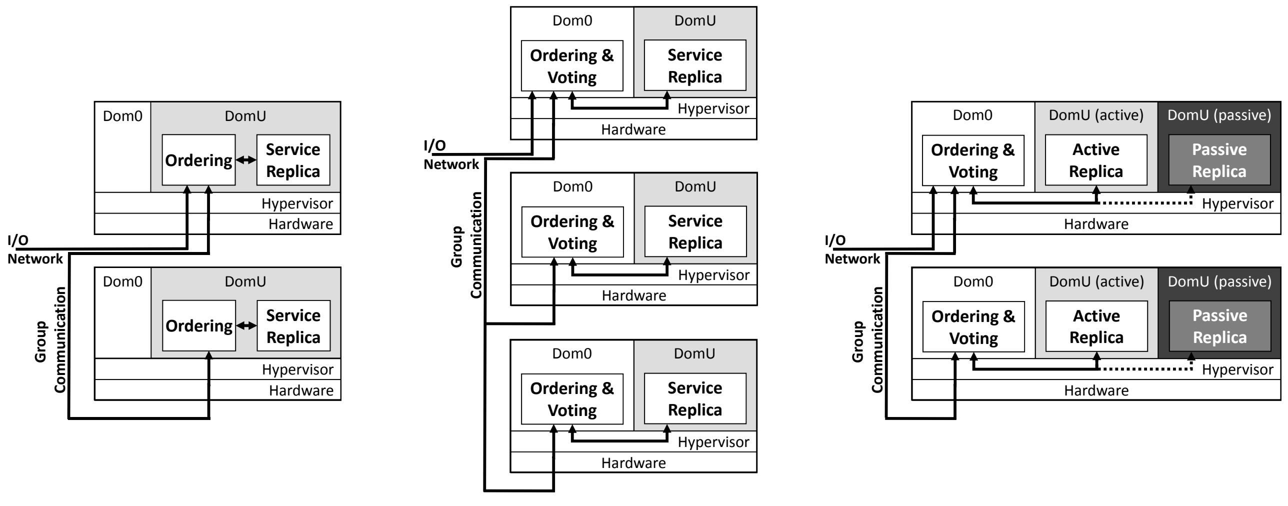
In case of detected faults or timing violations, the execution of one or more requests invalidates the clean-state assumption for the former passive replica R_p, as requests can cause intrusions or may trigger bugs when being processed. Thus, in such scenarios, a replica manager snapshots the abstract state before activating R_p. At the next proactive recovery, this snapshot S' is used to initialize the new passive replica R_{p+1}. As the snapshot S' does not contain the latest application state, all state updates generated since the creation of the snapshot also have to be applied to R_{p+1}.

5. SPARE and the Cloud

In Section 2, we described how SPARE is integrated with a common hypervisor. In this section, we investigate how current cloud infrastructures, such as Amazon's EC2 [1] and Eucalyptus [33], can be extended by SPARE. In particular, we address the delegation of management rights, explicit activation and deactivation of virtual machines, collocation and dislocation of virtual machines, as well as extended fault-detection support. Note that most of the extensions described below target to increase efficiency, they are not strictly required to safely run SPARE in the cloud.

5.1. Management of Virtual Machines

In our basic architecture (see Figure 1), we assume that both replica manager and group communication are located inside a privileged domain that has full control over the hardware, and accordingly over all virtual machines hosted by the associated physical machine. In the context of an infrastructure cloud, this is not practical as virtual machines of multiple customers might be collocated on one physical machine. Thus, we separate our base system from the hardware and the infrastructure responsible for managing virtual machines in the cloud. This can be achieved by running the replica manager in its own virtual machine and providing it with means to delegate management rights for virtual machines. In practice, a user of EC2 already possesses credentials to start and stop her virtual machines. For SPARE, those credentials need to be delegable in a way that a replica manger can be equipped with the capabilities to



(a) Crash architecture (b) AppBFT architecture (c) SPARE architecture

Figure 3. Architectures of Crash, AppBFT, and SPARE that tolerate a single fault.

start and stop its associated active and passive replicas. Besides starting and stopping replicas, for efficiently supporting SPARE, the infrastructure should also be extended by an interface that enables the replica manager to pause and unpause virtual machines. Note that such an interface offers the opportunity to reduce the resource consumption of passive replicas and therefore allows to improve the overall resource savings of SPARE, but is otherwise optional.

The replica manager may also be provided with an optional interface that comprises methods to dynamically assign memory and processor capacities to virtual machines. On the one hand, such a functionality makes it possible to keep the resource consumption of the virtual machine low while the replica is passive; on the other hand, it allows to dynamically increase the resources available to the virtual machine once the replica is activated. So far, assignment of resources to virtual machines is static in EC2 and Eucalyptus. Technically, the necessary support is already available at the virtualization layer; however, up to now, there was no need to make this feature accessible to customers. Nevertheless, even without such an interface, using passive replicas still saves costs due to EC2 mainly billing the usage of a virtual machine, besides billing its basic resource footprint (i. e., memory and number of CPUs).

5.2. Placement of Virtual Machines

During deployment, the collocation and dislocation of virtual machines is an essential requirement for SPARE. In particular, a virtual machine hosting a replica manager should be collocated with the virtual machines hosting the associated active and passive replicas. In contrast, the virtual machines hosting replica managers must be placed on separate physical machines, otherwise compromising the fault-tolerance properties of the whole system. EC2 already provides limited support for controlling the placement of virtual machines by distinguishing between different availability zones. For SPARE, this support should be extended, for example, by directly supporting SPARE and treating all virtual machines assigned to the same replica manager as a unit that can only be distributed as a whole. A more generic approach would be to use a dedicated query language to specify placement constraints as proposed in the context of distributed testbeds [35, 44].

5.3. Detection of Node Crashes

For SPARE, an infrastructure cloud needs to offer extended support for fault detection. This can be implemented in several ways, most simply, by placing replicas on directly interconnected nodes to enable a reliable fault detection in case of $f + 1$ hosts [14, 20]. If a more relaxed distribution is demanded and custom hardware settings are not an option, a witness-based approach is suitable [28, 36]. Here, additional nodes only take part in the communication to witness the occurrence of crashes and network partitions; the additional resource overhead would be minimal and the distribution requirements could be relaxed.

6. Evaluation

In this section, we evaluate the resource footprint of SPARE for the RUBiS web service benchmark [34] and compare it to the resource footprint of the two related replicated settings *Crash* and *AppBFT* (see Figure 3).

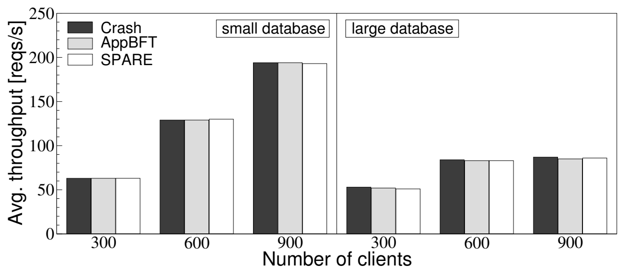


Figure 4. Throughput for Crash, AppBFT, and SPARE in dependence of the number of clients and the size of the database.

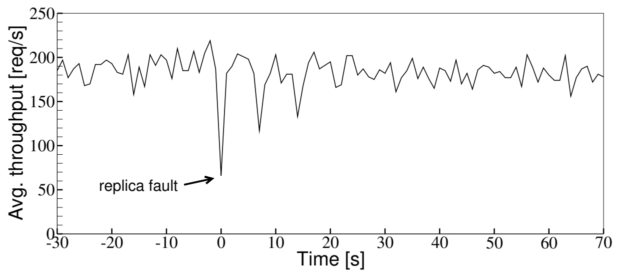


Figure 5. Impact of a replica fault on the throughput of SPARE during a test run with 900 clients, using the small database.

The characteristics of Crash and AppBFT are as follows:

- *Crash* is an actively-replicated crash-tolerant setting with $g + 1$ replicas that tolerates up to g replica faults; on a Crash host, the group communication runs in the same virtual machine (DomU) as the service replica.
- *AppBFT* is a Byzantine fault-tolerant setting that comprises a trusted ordering component and $2f + 1$ replicas to tolerate up to f Byzantine faults in the application domain.

Throughout the evaluation, we use configurations that are able to tolerate a single fault (i. e., $f = g = 1$). We also show that SPARE performs as well as Crash and AppBFT during normal-case operation, and evaluate the impact of a replica failure in SPARE. Furthermore, we discuss the resource savings that SPARE offers to web-based services and NFS. Finally, we analyze the cost-saving potential of SPARE in the cloud.

Our evaluation uses a cluster of quad-core CPU hosts (Intel Core 2 Quad CPU with 2.4 GHz, and 8 GB RAM), connected with switched 1 Gb/s Ethernet. Each replica host is running a Xen hypervisor with an Ubuntu 7.10 (2.6.24-16-xen kernel) Dom0; application domains are running Debian 4.0 (2.6.22-15-xen kernel). Clients are running on a separate machine. Our prototype relies on the Spread [4] group communication.

6.1. RUBiS

RUBiS is a web-based auction system created to benchmark middleware infrastructures under realistic workloads; it therefore is a fitting example of a multi-tier application to be typically run in an infrastructure cloud relying on SPARE. RUBiS includes a benchmarking tool that simulates the behavior of human clients browsing the auction system's web site, logging into their accounts, registering new items, and placing bids. The tool approximates a real-life usage scenario by processing a transition table that defines step-by-step probabilities for user actions such as registering users or bidding for items, including pauses between single actions ("think time").

In the configuration we use in our evaluation, a set of Java servlets is running on a Jetty web server that processes client requests. Information about registered users and item auctions is stored in a MySQL database. Each test run starts with a service state taken from a database dump published on the RUBiS web site. As database size affects the processing time of requests (i. e., the larger the database, the more records have to be searched), we vary between a small (about 100.000 users and 500.000 bids) and a large (about a million users and five million bids) initial database state.

A RUBiS replica manages all of its service state in the MySQL database that (without exception) is updated by servlets issuing SQL statements via the Java Database Connectivity (JDBC) API. To extract state updates, we created a library that analyzes those JDBC calls; for every state-modifying operation (e. g., `INSERT`, `UPDATE`), our library extracts the statement's write set, includes it in a state-update message, and forwards the state update to the local replica manager. In our library, write-set extraction is done by executing a query that reads the affected data records after the state-modifying operation completed; we found this straight-forward approach to be sufficient for RUBiS. In general, more sophisticated techniques [39, 40] should be applied to extract write sets.

6.2. Performance

We evaluate the throughput performance of the three settings (Crash, AppBFT, and SPARE) during normal-case operation, varying the number of clients from 300 to 900; 900 clients saturate the service on our machines when the large database is in use. Each client performs a six-minute runtime session. The results presented in Figure 4 show that the throughput realized for SPARE and AppBFT is within 4 % of the throughput realized for Crash. Note that this is possible, as both settings only provide BFT for the application domain, not full BFT. Therefore, they do not require a full-fledged BFT agreement protocol to order requests, which would lead to a significant drop in throughput.

6.3. Fault Handling

We evaluate the fault handling in SPARE by introducing an active-replica fault during a small-database test run with 900 clients. Note that we trigger the fault at the worst time; that is, the passive replica is in paused mode and first has to apply the maximum number of buffered state updates ($k-1 = 199$) before processing the pending client requests. Figure 5 shows the realized throughput around the time of the replica fault. When the fault is detected, the replica manager unpauses and prepares the local passive replica; in our scenario, this takes about half a second in the worst case. After that, the (now former) passive replica fully replaces the faulty active replica.

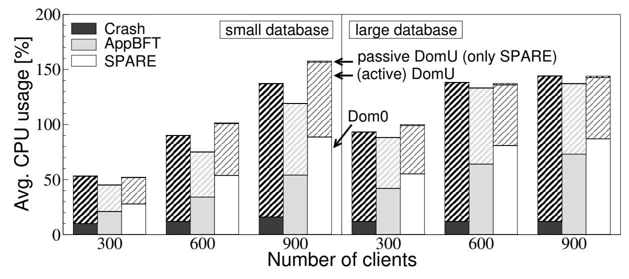
6.4. Resource Footprint

During the test runs for Section 6.2, we collected information about the CPU, memory, network, disk, and power consumption of replicas in the three settings. In this section, we assemble this data to present resource footprints for Crash, AppBFT, and SPARE. Note that we first compare the three settings using the average resource usage *per host*. Next, we address the resource footprint of a passive service replica in SPARE. Finally, we discuss overall resource footprints that take the total number of replicas into consideration; that is, they recognize that Crash requires $g + 1$ hosts, that SPARE comprises $f + 1$ hosts, and that AppBFT relies on $2f + 1$ hosts.

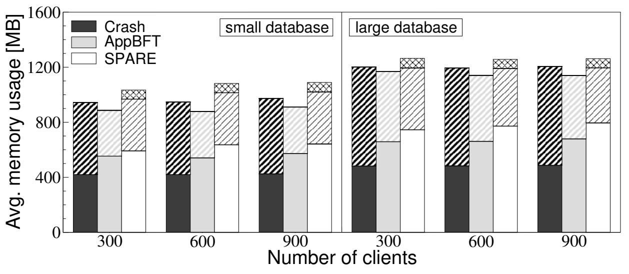
6.4.1. Host Resource Footprint

Figure 6 presents the usage of different resource types per host. The host resource footprint of Crash and AppBFT comprises the resource usage of the Dom0 and the application domain running the service replica; for SPARE, the footprint also includes the resource usage of the passive replica. Note that for disk usage, we only report the amount of data written to disk by application domains, as only those domains manage data on disk. Furthermore, we only report the Dom0 value for network transfer volume, as service replicas do not have direct access to the network. Finally, we report a combined value for power consumption.

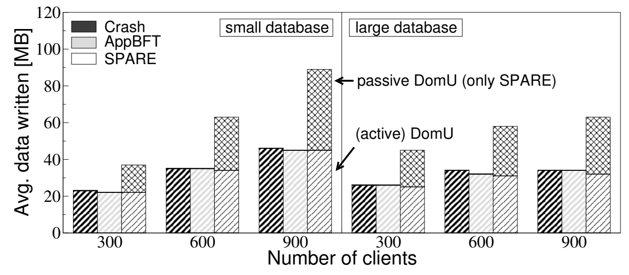
Our results show that resource types can be divided into two groups: For CPU, disk writes, and network transfer volume, the amount in use is mainly dependent on throughput. With more requests being received, distributed, and processed, as well as more replies being voted, usage of



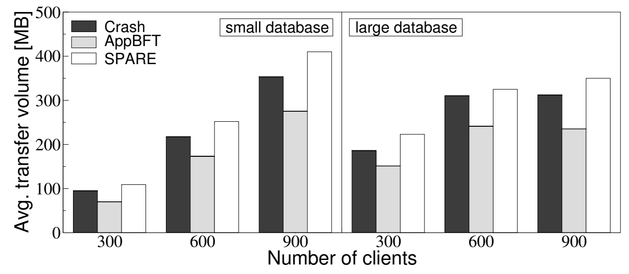
(a) CPU usage per host (aggregation)



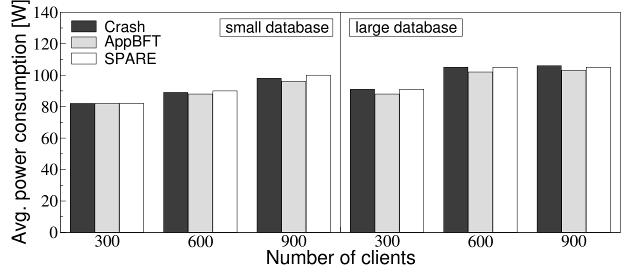
(b) Memory usage per host



(c) Data written to disk per host



(d) Network transfer volume per host



(e) Power consumption per host (idle power consumption: 73 W)

Figure 6. Host resource footprints of Crash, AppBFT, and SPARE for the RUBiS benchmark with different database sizes.

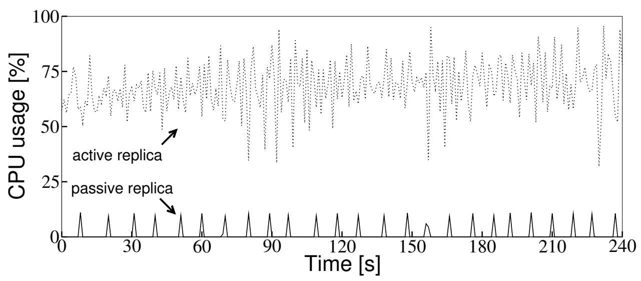


Figure 7. CPU-usage comparison between an active and a passive replica running on the same host (900 clients, small database).

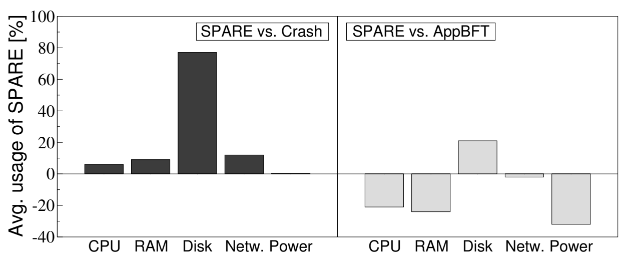


Figure 8. Resource usage of SPARE in comparison to Crash and AppBFT based on the overall resource footprints ($f = g = 1$).

those resource types significantly increases. In contrast, for memory and power consumption, the amount in use is dominated by a relatively high base level, throughput plays only a minor role.

Figure 6 shows that a SPARE host obtains more resources than an AppBFT host for all resource types. The increased resource usage stems from two factors: First, an AppBFT host is connected to only every third client, whereas a SPARE host handles half of all client connections; this factor is mainly responsible for SPARE's increased usage of CPU, memory, and network transfer volume in the Dom0. Second, managing and updating a passive replica requires additional memory and disk space. With all application-state changes of the active replica being re-applied through state updates in the passive replica, the amount of data written to disk greatly increases for SPARE.

For a comparison between SPARE and Crash only the second factor is relevant, as both settings comprise the same number of hosts. Therefore, differences in host resource footprints underline the resource overhead obtained for managing a passive service replica. Besides using a small fraction of CPU and memory, verifying state updates leads to an increase in network transfer volume, especially for scenarios with high throughput that include a large number of state changes.

6.4.2. Passive-Replica Resource Footprint

The resource footprint of a passive replica greatly differs from the resource footprint of an active replica. Figure 7 shows the CPU usage of both during a runtime session with 900 clients. In contrast to the active replica, the passive replica does not process any client requests but remains paused[1] most of the time. When the passive replica is woken up to apply state updates, it temporarily uses a small fraction of a single CPU. However, with state updates only being processed by the database layer (and not requiring operations on all layers like a state-modifying service request), CPU usage during this procedure is modest. In total, a passive replica uses 97% less CPU than an active replica; memory usage shows a similar picture (83% less). In contrast, the amount of data written to disk is nearly the same for active and passive replicas (only a 14% decrease for passive replicas), as discussed in Section 6.4.1.

6.4.3. Overall Resource Footprint

Our per-host results have shown that a SPARE host requires more resources than a host in Crash and AppBFT. To put these values into perspective, we now discuss the overall resource footprints for the three settings, which consider the overall number of machines n ($n_{Crash} = g + 1$, $n_{AppBFT} = 2f + 1$, and $n_{SPARE} = f + 1$); that is, for each resource type, we multiply the average resource usage of a single host with n. For reasons of clarity, we do not report absolute numbers, but present a summarizing comparison of the overall resource footprints for configurations that tolerate a single fault (see Figure 8).

The overall resource footprints indicate that for CPU, memory, and network transfer volume SPARE requires only 6-12% of additional resources to provide Byzantine fault tolerance in the application domain, in comparison to a crash-tolerant setting (i. e., Crash). Disk writes constitute an exception, because passive service replicas need to apply the same application-state changes (which are responsible for most of the disk writes) as active service replicas. Given that disk space is inexpensive and abundantly available in infrastructure clouds, we consider the overall resource overhead of SPARE acceptable. SPARE and Crash basically have the same power consumption.

Compared to the fully replicated AppBFT setting, however, SPARE saves 21-32% of CPU, memory, and power; again, disk writes remain an exception. Note that network

[1]Our prototype uses Xen's `pause`/`unpause` mechanism to deactivate/activate the virtual machine hosting the passive replica; both operations take about 130 ms. Although a paused virtual machine still consumes allocated memory, we decided not to use `suspend`/`resume`, which writes the status of a virtual machine to disk and removes it from memory, as suspending/resuming a virtual machine takes about 15 seconds.

Architecture	Small	Large	Extra Large	Annual Costs
Crash			2 (75%)	**7,833$**
AppBFT		6 (75%)		**11,750$**
SPARE	2 (15%)	4 (75%)		**8,393$**

Table 1. EC2 price comparison as of March 2010 for Crash, AppBFT, and SPARE for the virtual machines of a service (Linux operating system, EU region) assuming a one-year contract model.

transfer volume for SPARE and AppBFT is basically the same, that is, for this resource type, the overhead for managing state updates is comparable to the communication overhead for a third host.

6.5. Estimated Resource Savings for Other Services

In addition to measuring the resource savings for RUBiS, we analyze the potential for other applications and services to save resources using SPARE. In this context, the fraction of read-only requests is of particular interest, as these requests do not affect the resource usage of passive replicas. Due to lack of large traces, we investigated the share of read-only requests for the multi-tier TPC-W [17] benchmark that is designed to emulate real internet-commerce workloads. Just like RUBiS (85% read-only requests), TPC-W assumes high rates of browsing-related requests (e. g., 80% for the shopping mix) that do not change the application state. Thus, we expect the resource-usage characteristics and resource savings for TPC-W to be similar to our evaluation results for the RUBiS benchmark.

Besides multi-tier applications, we investigated the possible resource savings of distributed file systems, another example of a service that is often considered critical. We estimate the resource savings of passive replication for network file systems based on the results of two recent studies [16, 30]. Ellard et al. [16] investigated NFS traces in an university environment. For a central email and service system, they concluded that read operations outnumber write operations by a factor of 3. The same applies to the generated I/O. Leung et al. [30] conducted a study on a large enterprise data center. There, read operations outnumber writes by a factor of at least 2.3, causing at least two thirds of the total I/O. Based on these results, it is reasonable to assume that a passive replica saves at least 66% of CPU usage compared to an active replica and also provides resource savings in terms of RAM, network transfer volume, and power consumption.

6.6. Estimated Cost Savings

While the resource savings of SPARE per se are very promising, another important question is how they translate to cost reduction and greening the IT infrastructure; here, the actual energy savings are of key interest. We assume a self-hosted environment, an electricity price of 20 c/kWh [24], and the use of standard server hardware (consuming 100 W per node, see Figure 6e). Then, the reduction from three to two machines to provide application-level BFT in SPARE saves about 175 $ per year. This calculation only considers energy costs, acquisition and maintenance costs for the additional machine are not included.

To estimate the cost savings for running SPARE on top of an infrastructure cloud, we use Amazon's EC2 Cost Comparison Calculator [2]; Table 1 summarizes the key parameters. Amazon offers virtual compute nodes at various sizes: small, large, and extra large. For SPARE, we need four large instances and two small instances. The four large nodes represent the two active service replicas and their associated replica managers; the two small instances resemble the passive service replicas. Furthermore, we assume that the large instances have an average usage of 75%, whereas the small passive instances are only utilized 15% on average. In contrast, an AppBFT setting requires six large nodes, each having an average usage of 75%. In a Crash setting, group communication and service replicas are collocated on one node. Thus, we only need two nodes but, due to the increased resource demand, of size extra large. Note that we neglect network traffic, as it highly depends on the usage profile of a service. In total, SPARE causes additional costs of 560 $ (7%) compared to Crash, but saves 3.357 $ (29%) compared to AppBFT.

7. Discussion

In this section, we discuss the rational behind SPARE's focus on application-level Byzantine fault tolerance and argue for the use of a trusted subsystem. Furthermore, we address the need for fault-independent service replicas and outline two extensions that are required to deploy SPARE in a wide area environment.

7.1. Using a Trusted Subsystem

SPARE offers a higher degree of fault tolerance (i. e., application-level BFT) compared to pure fail-stop replicated applications, while consuming less resources than traditional "$3f+1$"-BFT replication and "$2f+1$"-approaches that (similar to SPARE) are based on a trusted subsystem. Although a trusted subsystem might be too risky for high-value targets, we consider this separation justified, based on the resource savings and improved fault tolerance in comparison to fail-stop replication. We see our work and related approaches as a general trend towards partitioned system architectures, as practical code-verification approaches make constant progress, starting at the fundamental system layers, such as the operating system and the hypervisor.

However, a whole system verification is still out of reach as such approaches are very time consuming. Furthermore, rapid and constant system evolution, especially at the middleware and application layer, make the verification of whole systems a fast moving target. Accordingly, a combined architecture, built of trusted long-lasting components and more flexible and fast moving but appropriately replicated parts, can foster cost-efficient and fault-tolerant systems. Our prototype is only an example of a possible SPARE implementation; the concept behind SPARE could also be implemented on top of other hybrid architectures that place a smaller part of the message-ordering support in the trusted subsystem [11, 13].

7.2. Fault-independent Service Replicas

SPARE is resilient against faults and intrusions as long as at most f service replicas are faulty at the same time. As a direct consequence of this upper-bound assumption, service replicas must be fault independent; that is, their implementations must be heterogeneous. If the implementations of the service replicas were homogeneous, a single malicious request sent by an attacker could, for example, compromise all replicas at once, as sharing the same software stack also means sharing the same vulnerabilities. In SPARE, fault independence of service replicas can be achieved by introducing diversity at the operating system, middleware, and application level.

Virtualization offers an excellent basis for hosting different operating systems on the same physical machine [10]. A study examining diverse database replication has shown that using off-the-shelf software components can be an effective means to achieve fault independence [18]. Lately, even off-the-shelf operating systems like Linux [46] and Windows [21] possess features for address-space randomization that make stack exploits more difficult, especially in a replicated setting.

In sum, diverse replication using off-the-shelf components, thereby avoiding the overhead of N-version programming, can be very effective and is easily possible; in [15], for example, we presented a heterogeneous implementation of the RUBiS benchmark that comprises only off-the-shelf components. In general, SPARE does not make any assumptions about the degree and level of diversity implemented in the service replicas. This way, replica implementations can be tailored to the protection demands of the specific replicated application.

7.3. Extensions for Use in Wide Area Networks

We assume SPARE to be typically used in a closely-connected environment, where node crashes can be detected within a finite time interval. In wide area environments where, for example, the service is composed from distinct cloud computing providers, this assumption does not hold. In this case, additional f nodes are required to witness the occurrence of node crashes [28, 36]. Note that this requirement is not specific to SPARE, but also applies to crash-tolerant systems. However, the additional nodes are only needed to participate in case of suspected crashes and can otherwise be used for other tasks.

Distributing SPARE across different data centers or different organizational domains also creates the need for secure authenticated communication links between replica managers. In such a case, a secure group communication like SecureSpread [3] may be used for message ordering in SPARE. Again, note that this modification would also be necessary for Crash and AppBFT.

8. Related Work

In the context of database replication there have been several works using a group communication for efficient recovery of failed replicas [23, 25, 31, 37]. Jiménez-Peris et al. [23] showed that replicas can be recovered in parallel by receiving state updates from the remaining operational nodes. Liang and Kemme [31] proposed a slightly more adaptive approach that enables a recovering replica to either receive data items or recent updates. All these works assume a crash-stop failure model and accordingly utilize operational replicas to initialize recovering nodes. The latter is also done in the context of Byzantine fault-tolerant replication (i. e., BASE [9]) by taking snapshots and a log of recent operations. In addition to the fail-stop failure model, the input for a recovering replica has to be validated through agreement. In SPARE, state updates can be seen as an extension to replies of client requests that are processed during normal operation; due to the virtualization-based architecture, this can be done efficiently. Our approach also aids a fast detection of faults and supports rapid recovery. Thereby, we avoid the non-trivial task of taking implementation-neutral snapshots in a non-intrusive way by implementing an abstract shadow state that has to be maintained by every replica, as proposed by BASE.

Over the last few years, there has been a rapid development from making Byzantine fault tolerance practical requiring at least $3f + 1$ replicas [7], to hybrid systems that separate agreement and execution comprising $3f + 1$ agreement nodes and $2f + 1$ execution nodes [49], to systems assuming a hybrid fault model [11, 13, 38]. The latter group requires that some components (e. g., a counter [11], a hypervisor, or a whole operating-system instance [13, 38]) are trusted and can only fail by crashing. Besides improving performance, reducing the resource footprint is one of the key reasons for this evolution. SPARE contributes to this by proposing a hybrid system that comprises the minimal set of execution replicas ($f + 1$) that enables to detect visible faults, and reactively offers fault tolerance by activating additional replicas. However, the offered BFT capability is limited to a replicated service and its associated execution environment, similar to other hybrid architectures [10, 13, 38].

Closest to our approach is ZZ, a BFT system that also builds on the idea of having $f + 1$ active replicas [48]. In ZZ, spare virtual machines can be dynamically dedicated to a certain application by on-demand fetching and verifying a file-system snapshot. Whereas this offers a resource-efficient solution, as spare machines might be acquired from a set of virtual machines that are shared among different applications, the actual initialization of the replicas requires additional work and time. SPARE reduces this overhead by binding passive replicas to applications and by periodically updating them, thereby also providing the basis for proactive recovery; ZZ does not consider proactive recovery. ZZ features a separation of agreement and execution that requires $3f + 1$ agreement nodes and up to $2f + 1$ replicas. As a result, ZZ has a larger resource footprint than SPARE. However, similar to SPARE, it relies on the trustworthiness of the underlying virtualization infrastructure and its managing operating system if more than one replica is placed on the same physical machine.

If approaches such as SPARE and ZZ are still too resource demanding and the focus is on malicious faults, one can resort to detecting misbehavior. Examples for this direction of research are PeerReview [19] and Venus [41]. While PeerReview enables accountability in a large distributed system, Venus ensures the storage integrity and consistency of a single server.

9. Conclusion

SPARE is a novel virtual-machine–based approach to provide Byzantine fault tolerance at the application level, at minimal resource cost. A SPARE configuration that tolerates up to f faults comprises only $f + 1$ active service replicas during normal-case operation. In case of faulty or slow replicas, up to f additional passive replicas are dynamically activated. Our evaluation shows that SPARE requires 21-32% less resources than a fully replicated system.

Acknowledgements

This work was partially supported by the German Research Council (DFG) under grant no. KA 3171/1, and by FCT, through project PTDC/EIA-CCO/108299/2008, through LASIGE (U.408) Multiannual Funding Programme and through the Carnegie Mellon Portugal Program.

References

[1] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] Amazon. User guide: Amazon EC2 cost comparison calculator. `http://aws.amazon.com/economics/`.

[3] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 330–343, 2000.

[4] Y. Amir and J. Stanton. The Spread wide area group communication system. Technical Report CNDS-98-4, Center for Networking and Distributed Systems, Johns Hopkins University., 1998.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *Distributed systems (2nd Ed.)*, chapter The primary-backup approach, pages 199–216. Addison-Wesley, 1993.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, 1999.

[8] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 19–34, 2000.

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

[10] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine Byzantine-fault tolerance. In *USENIX Annual Technical Conference*, pages 287–292, 2008.

[11] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 189–204, 2007.

[12] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: the time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–4, 2008.

[13] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183, 2004.

[14] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.

[15] T. Distler, R. Kapitza, and H. P. Reiser. State transfer for hypervisor-based proactive recovery of heterogeneous replicated services. In *Proceedings of the 5th Sicherheit Conference*, pages 61–72, 2010.

[16] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings the 2nd USENIX Conference on File and Storage Technologies*, pages 203–216, 2003.

[17] D. F. García and J. García. TPC-W e-commerce benchmark evaluation. *Computer*, 36(2):42–48, 2003.

[18] I. Gashi, P. Popov, and L. Strigini. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing*, 4(4):280–294, 2007.

[19] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 175–188, 2007.

[20] Hewlett-Packard. HP Integrity NonStop computing. `http://h20341.www2.hp.com/NonStopComputing/cache/76385-0-0-0-121.html`, 2010.

[21] M. Howard and M. Thomlinson. Windows Vista ISV security. `http://msdn.microsoft.com/en-us/library/bb430720.aspx`, 2010.

[22] IBM. Smart Business Development and Test Cloud. Technical report, IBM, 2010.

[23] R. Jiménez-Peris, M. Patiño Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, page 150, 2002.

[24] N. Joukov and J. Sipek. GreenFS: making enterprise computers greener by protecting them better. In *Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems*, pages 69–80, 2008.

[25] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the 2001 Conference on Dependable Systems and Networks*, pages 117–130, 2001.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, pages 207–220, 2009.

[27] P. Kuznetsov and R. Rodrigues. BFTW3: Why? When? Where? Workshop on the Theory and Practice of Byzantine Fault Tolerance. *SIGACT News*, 40(4):82–86, 2009.

[28] L. Lamport and M. Massa. Cheap Paxos. In *Proceedings of the 2004 Conference on Dependable Systems and Networks*, pages 307–314, 2004.

[29] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *Proceedings of the 2nd World Congress on Formal Methods*, pages 806–809, 2009.

[30] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 213–226, 2008.

[31] W. Liang and B. Kemme. Online recovery in cluster databases. In *Proceedings of the 11th Conference on Extending Database Technology*, pages 121–132, 2008.

[32] D. G. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *Proceedings of the 1st European Workshop on System Security*, pages 40–46, 2008.

[33] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, 2009.

[34] ObjectWeb Consortium. RUBiS: Rice University Bidding System. `http://rubis.ow2.org/`.

[35] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. Service placement in shared wide-area platforms. In *Proceedings of the USENIX Annual Technical Conference*, pages 273–288, 2006.

[36] J.-F. Paris. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 606–612, 1986.

[37] F. Pedone and S. Frølund. Pronto: High availability for standard off-the-shelf databases. *Journal of Parallel and Distributed Computing*, 68(2):150–164, 2008.

[38] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proceedings of the 26th IEEE Symposium on Reliable Distributed Systems*, pages 83–92, 2007.

[39] J. Salas, R. Jiménez-Peris, M. Patiño Martínez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 377–390, 2006.

[40] R. Salinas-Monteagudo and F. D. Munoz-Escoi. Almost triggerless writeset extraction in multiversioned databases. In *Proceedings of the 2nd International Conference on Dependability*, pages 136–142, 2009.

[41] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: verification for untrusted cloud storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, pages 19–30, 2010.

[42] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13:14–22, 2009.

[43] P. Sousa, N. F. Neves, and P. Veríssimo. Hidden problems of asynchronous proactive recovery. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability*, 2007.

[44] D. Spence, J. Crowcroft, S. Hand, and T. Harris. Location based placement of whole distributed systems. In *Proceedings of the 1st Conference on Emerging Network Experiments and Technologies*, pages 124–134, 2005.

[45] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th ACM SIGOPS European Conference on Computer Systems*, pages 209–222, 2010.

[46] The PaX Project. `http://pax.grsecurity.net`.

[47] A. Velte and T. Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., 2010.

[48] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical BFT using virtualization. Technical Report TR14-08, University of Massachusetts, 2008.

[49] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. *SIGOPS Operating Systems Review*, 37(5):253–267, 2003.