# Proactive Service Migration for Long-Running Byzantine Fault Tolerant Systems

Wenbing Zhao and Honglei Zhang

Department of Electrical and Computer Engineering

Cleveland State University, 2121 Euclid Ave., Cleveland, OH 44115

wenbing@ieee.org

## Abstract

*In this paper, we describe a proactive recovery scheme based on service migration for long-running Byzantine fault tolerant systems. Proactive recovery is an essential method for ensuring long term reliability of fault tolerant systems that are under continuous threats from malicious adversaries. The primary benefit of our proactive recovery scheme is a reduced vulnerability window under normal operation. This is achieved by two means. First, the time-consuming reboot step is removed from the critical path of proactive recovery. Second, the response time and the service migration latency are continuously profiled and an optimal service migration interval is dynamically determined during runtime based on the observed system load and the user-specified availability requirement.*

**Keywords:** Proactive Recovery, Byzantine Fault Tolerance, Service Migration, Replication, Byzantine Agreement

## 1. Introduction

We have seen increasing reliance on services provided over the Internet. These services are expected to be continuously available over extended period of time (typically 24x7 and all year long). Unfortunately, the vulnerabilities due to insufficient design and poor implementation are often exploited by adversaries to cause a variety of damages, e.g., crash of applications, leak of confidential information, modification or deletion of critical data, or injection of erroneous information into the system. These malicious faults are often modeled as Byzantine faults [13], and they are detrimental to any online service provider. Such threats can be coped with using Byzantine fault tolerance (BFT) techniques, as demonstrated by many research results [2, 3, 5, 25]. Byzantine fault-tolerance algorithms assume that only a small portion of the replicas can be faulty. When the number of faulty replicas exceeds a threshold, BFT may fail. Consequently, Castro and Liskov [2] proposed a proactive recovery scheme (for BFT) that periodically reboots replicas and refreshes their state, even before it is known that they have failed. Furthermore, they introduced a term called window of vulnerability (or vulnerability window) referring to the time window in which all replicas are proactively recovered at least once. As long as the number of compromised replicas does not exceed the threshold within the vulnerability window, the integrity of the BFT algorithm holds and the services being protected remain highly reliable over the long term.

However, the proactive recovery scheme in [2] has a number of issues. First, it assumes that a simple reboot (*i.e.,* power cycle of the computing node) can be the basis for repairing a compromised node (of course, in addition to the reboot, which could wipe out all memory-resident malware, a copy of clean executable code and the current state must be fetched and restored, and all session keys must be refreshed upon recovery), which might not be the case (*e.g.,* some attacks might cause hardware damages), as pointed out in [20]. Second, even if a compromised node can be repaired by a reboot, it is often a prolonged process (typically over $30s$ for modern operating systems). During the rebooting step, the BFT service might not be available to its clients (*e.g.,* if the rebooting node happens to be a nonfaulty replica needed for the replicas to reach a Byzantine agreement). Third, there lacks coordination among replicas to ensure that no more than a small portion of the replicas (ideally no more than $f$ replicas in a system of $3f + 1$ replicas to tolerate up to $f$ faults) are undergoing proactive recovery at any given time, otherwise, the service may be unavailable for extended period of time. The static watchdog timeout used in [2] also contributes to the problem because it cannot automatically adapt to various system loads, which means that the timeout value must be set to a conservative value based on the worst-case scenario. The staggered proactive recovery scheme in [2] is not sufficient to prevent this problem from happening if the timeout value is set too short. Recognizing these issues, a number of researchers have proposed various methods to enhance the original proactive recovery scheme.

The issue of uncoordinated proactive recovery due to system asynchrony has been studied by Sousa et al. [21, 22]. They resort to the use of a synchronous subsystem to ensure the timeliness of each round of proactive recovery. In particular, the proactive recovery period is determined *a priori* based on the worst case execution time so that even under heavy load, there

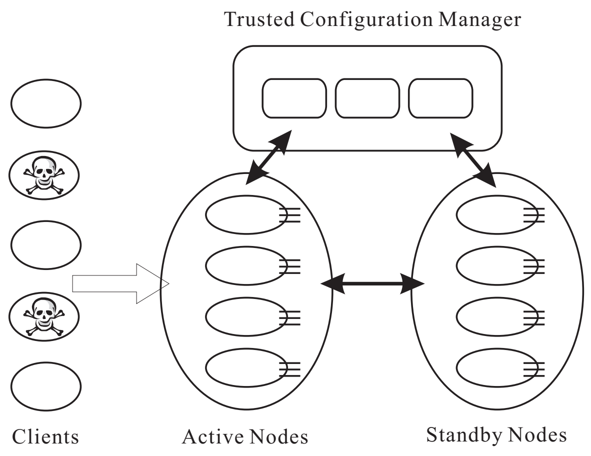

**Figure 1. Main components of the BFT system with migration-based proactive recovery.**

will be no more than $f$ replicas going through proactive recovery.

The impact of proactive recovery schemes on the system availability has also been studied by Sousa et al. [23] and by Reiser and Kapitza [19]. In the former scheme, extra replicas are introduced to the system and they actively participate message ordering and execution so that the system is always available when some replicas are undergoing proactive recovery. In the latter scheme [19], a new replica is launched by the hypervisor on the same node when an existing replica is to be rebooted for proactive recovery so that the availability reduction is minimized.

In this paper, we present an alternative proactive recovery scheme based on service migration. Similar to the work in [19, 21, 22, 23], the objective of our approach is to provide proactive recovery for long-running Byzantine fault tolerance applications without suffering from the issues of [2]. In the following, we first present an overview of our scheme and then compare with similar approaches reported in [19, 21, 22, 23].

Our proactive recovery scheme requires the availability of a pool of standby computing nodes in addition to the active nodes where the replicas are deployed. Furthermore, we assume the availability of a trusted configuration manager. The main components for our scheme is shown in Figure 1. The basic idea is outlined below. Periodically, the replicas initiate a proactive recovery by selecting a set of active replicas, and a set of target standby nodes for a service migration. At the end of the service migration, the target nodes are promoted to the group of active nodes and the source active nodes will be put under a series of preventive sanitizing and repairing steps (such as rebooting and swapping in a clean hard drive with the original system binaries) before they are assigned to the pool of standby nodes. The sanitizing and repairing step is carried out *off the critical path of proactive recovery* and consequently, it could lead to a smaller vulnerability window under normal operation and has minimum negative impact on the availability of the services being protected. In addition, our migration-based proactive recovery scheme also ensures a coordinated periodical recovery and the dynamic adjustment of the proactive recovery period based on the synchrony of the system and the load, which prevents harmful excessive concurrent proactive recoveries.

As can be seen from previous descriptions, although our scheme largely shares the same objectives with similar approaches [19, 21, 22, 23], they differ significantly in a number of ways.

To prevent uncoordinated proactive recovery, our scheme relies on an explicit coordination mechanism, while the approach proposed by Sousa et al. [21, 22] resorts to *a priori* worst case execution time estimation. Furthermore, the proactive recovery period in [21, 22] is fixed throughout the life-cycle of the application. Inevitably, the proactive recovery period has to be set pessimistically, which would lead to a potentially large window of vulnerability. Even though we also require *a priori* worst case execution time estimation, in our scheme, the proactive recovery period can be dynamically adjusted depending on the observed system load. When the system load is light, the vulnerability window can be reduced accordingly, even if it is set to a large conservative value in the beginning of the execution. However, we should note that under certain attacks, in particular, the denial-of-service (DoS) attacks, the advantage of our scheme over other approaches disappears.

The ways to attain better availability are also very different between our scheme and others [19, 23] despite the fact that all these approaches require the availability of extra replicas. It might appear that our scheme is rather similar to that in [19] because in both schemes, a correct replica is made ready at the beginning of each round of proactive recovery. The major difference is that in our scheme, the correct replica is located on a different physical node, while in the scheme [19], the new replica is launched in a different virtual machine located in the same physical node. Apparently, in the scheme [19], the proactive recovery time could be shorter and fewer physical nodes are needed, which reduces the hardware and software costs. However, our scheme offers better fault isolation. In particular, if an attack has caused physical damage on the node that hosts the replica to be recovered, or it has compromised the hypervisor of the node [24], the new replica launched in the same node in the scheme [19] is likely to malfunction.

The difference between our scheme and that in [23] is more subtle. In our scheme, the number of active replicas remains the optimal value (*i.e.,* $3f+1$) and the standby replicas (*i.e.,* extra replicas) *do not* participate normal server processing (unless they are promoted to the active replicas pool) and they are not directly accessible from external clients. In the scheme [23], on the other hand, the recovering replicas are regarded as failed, and therefore, higher degree of replication is needed to tolerate the same number of Byzantine faults and *all the replicas* would have to participate the Byzantine agreement process. Consequently, our scheme has a number of advantages: (1) the replicated server operation could be more efficient because there are fewer multicast messages due to the use of optimal number of active replicas, (2) the standby nodes are less likely to be compromised because they are isolated from the active replicas, and (3) the standby nodes can be probed more frequently without reducing the system performance.

Finally, because of the use of a pool of standby nodes, our scheme makes it possible to carry out time-consuming repairs of faulty nodes (possibly with physical damages due to attacks), which is not addressed by the work of [19, 21, 22, 23].

## 2. System Model

The nature of this research entails a synchrony requirement on the system, *i.e.*, we assume that all message exchanges and processing related to proactive recovery can be completed within a bounded time. However, the safety property of the Byzantine agreement on all proactive recovery related decisions (such as the selection of source nodes and destination nodes for service migration) is maintained without any system synchrony requirement.

As shown in Figure 1, to enable the migration-based proactive recovery, the BFT system contains three main components: a pool of nodes for active server replicas, a pool of standby nodes, and a trusted configuration manager. Each of the three components is deployed at a separate subnet for fault isolation and they are connected by an advanced managed switch such as Cisco Catalyst 6500. Each node in the pool of active nodes and the pool of standby nodes has three network interfaces: NIC1 for connection to external clients, NIC2 for connection between the two pools of nodes, and NIC3 for connection to the configuration manager. However, only an active node (running a server replica) has all three interfaces enabled. A standby node has NIC1 disabled. The trusted configuration manager can dynamically enable and disable the NIC1 and NIC2 interfaces of any node, *e.g.*, it disables NIC1 when removing a node from the pool of active nodes and it enables NIC1 when promoting a standby node to the pool of active nodes.

We assume that there are $3f + 1$ active nodes to tolerate up to $f$ Byzantine faulty replicas [2, 11]. The pool size of standby nodes ($\geq f$) should be large enough to repair damaged nodes while enabling frequent service migration for proactive recovery. Both active nodes and standby nodes can be subject to malicious attacks (in addition to other non-malicious faults such as hardware failures). However, we assume that the far majority of malicious attacks are imposed by *external* adversaries (either malicious clients or those impersonated as clients) via malformed requests to the replicated server. In light of this assumption, the system is configured such that the standby nodes are not directly accessible by external entities (*i.e.*, NIC1 is disabled). The purpose of this configuration is to ensure that the rate of successful attacks on the standby nodes is much smaller than that on active nodes.

Similar to [22, 23], we assume a fail-stop model on the trusted configuration manager. To ensure high availability, the trusted configuration manager is replicated using the Paxos algorithm [12]. The duty of the trusted configuration manager is similar to what has been described in [20], *i.e.*, it is used to manage the pool of standby nodes, and to assist service migration. Example tasks include frequently probing and monitoring the health of each standby node, and repairing any faulty node detected.

Other assumptions regarding the system is similar to those in [2] and they are summarized here. All communicating entities (clients, replicas and standby nodes) use a secure hash function such as SHA1 to compute the digest of a message and use the message authentication codes (MACs) to authenticate messages exchanged, except for key exchange messages, which are protected by digital signatures. For point to point message exchanges, a single MAC is included in each message, while multicast messages are protected by an authenticator. Each entity has a pair of private and public keys. The active and standby nodes each is equipped with a secure coprocessor and sufficiently large read-only memory. In these nodes, the private key is stored in the coprocessor and all digital signing and verification is carried out by the coprocessor without revealing the private key. The read-only memory is used to store the execution code for the server application and the BFT framework. We do not require the presence of a hardware watchdog timer because of the coordination of migration and the existence of a trusted configuration manager. Finally, we assume that an adversary is computational bound so that it cannot break the above authentication scheme.

## 3. Proactive Service Migration Mechanisms

The proactive service migration mechanisms collectively ensure the following objectives: (1) to ensure that correct active replicas have a consistent membership view of the available standby nodes, (2) to determine when to migrate and how to initiate a migration, (3) to determine the set of source and target nodes for migration, (4) to transfer a correct copy of the system state to the new replicas, (5) to notify the clients the new membership after each proactive recovery.

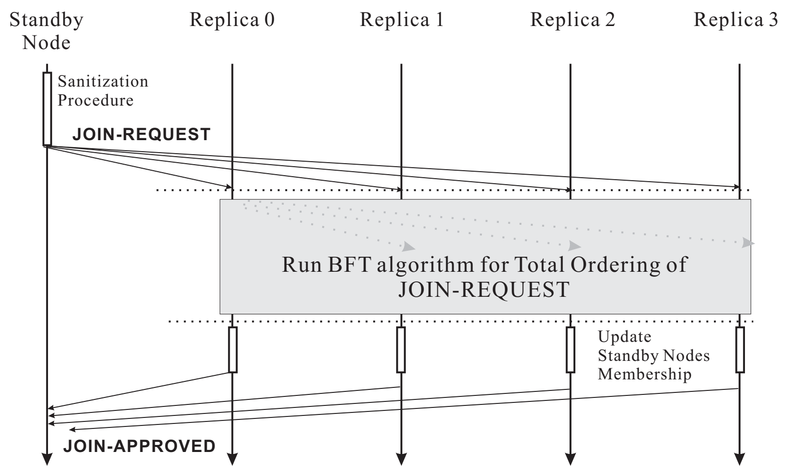

**Figure 2. The protocol used for a standby node to register with active replicas.**

### 3.1. Standby Nodes Registration

Each standby node is controlled by the trusted configuration manager and is undergoing constant probing and sanitization procedures such as reboot. If the configuration manager suspects the node to be faulty and cannot repair it automatically, a system administrator will be called in to manually fix the problem. Each time a standby node completes a sanitization procedure, it notifies the active replicas with a JOIN-REQUEST message in the form of $<$JOIN-REQUEST$, l, i_s>_{\sigma_{i_s}}$, where $l$ is the counter value maintained by the secure coprocessor of the standby node, $i_s$ is the identifier of the standby node, and $\sigma_{i_s}$ is the authenticator. The registration protocol is illustrated in Figure 2.

An active replica accepts the JOIN-REQUEST if it has not accepted one from the same standby node with the same or greater $l$. The JOIN-REQUEST message, once accepted by the primary, is ordered the same way as a regular message with a sequence number $n_r$, except that the primary also assigns a timestamp as the join time of the standby node and piggybacks it with the ordering messages. The total ordering of the JOIN-REQUEST is important so that all active nodes have the same membership view of the standby nodes. The significance of the join time will be elaborated later in this section.

When a replica executes the JOIN-REQUEST message, it sends a JOIN-APPROVED message in the form of $<$JOIN-APPROVED$, l, n_r>_{\sigma_i}$ to the requesting standby node. The requesting standby node must collect $2f + 1$ consistent JOIN-APPROVED messages with the same $l$ and $n_r$ from different active replicas. The standby node then initiates a key exchange with all active replicas for future communication.

A standby node might go through multiple rounds of proactive sanitization before it is selected to run an active replica. The node sends a new JOIN-REQUEST reconfirming its membership after each round of sanitization. The active replicas subsequently update the join time of the standby node.

It is also possible that the configuration manager deems a registered standby node as faulty and it requires a lengthy repair, in which case, the configuration manager deregisters the faulty node from active replicas by sending a LEAVE-REQUEST. The LEAVE-REQUEST is handled by the active replicas in a similar way as that for JOIN-REQUEST.

## 3.2. Proactive Service Migration

**When to Initiate a Proactive Service Migration?** The proactive service migration is triggered by the software-based migration timer maintained by each replica. The timer is reset and restarted at the end of each round of migration. An on-demand service migration may also be carried out upon the notification from the configuration manager (to be discussed later).

We require the user to specify several parameters:

1. The maximum response time to order and execute a request $T_{oe}^0$ based on the worst-case analysis. Note that $T_{oe}^0$ does not include the queueing delay for the request being ordered.

2. The minimum number of requests served $p^0$ during each proactive service migration round.

3. The maximum latency to carry out a service migration $T_s^0$, *i.e.*, the maximum time it takes to swap out an active replica and to replace it with a clean standby replica.

Based on these parameters, our proactive service migration mechanism determines an initial migration timeout value $T_w^0$ and dynamically adjusts the migration timeout value $T_w$ at runtime. The timeout value is capped by the initial timeout value to prevent an adversary from indefinitely increasing the proactive recovery period.

The initial timeout value $T_w^0$ is set to be $pT_{oe}^0$. The user provided parameters also implicitly specify a target availability $A^0$ of the system:

$$A^0 = p\frac{T_{oe}^0}{p^0 T_{oe}^0 + T_s^0} \tag{1}$$

$A^0$ corresponds to the availability under the worst-case scenario. During runtime, our mechanism continuously measures the average response time $T_{oe}$ to order and execute a request for the most recent $p^0$ requests, and the service migration latency

$T_s$. A notification is sent to the system administrator if either the response time or the service migration latency exceeds the worst-case values. The migration timeout value $T_w$ is dynamically adjusted to $pT_{oe}$, where the parameter $p$ is calculated based on the following equation

$$p = max(p^0, \frac{A^0 T_s}{(1 - A^0)T_{oe}}) \tag{2}$$

to satisfy both the requirements on the minimum number of requests served in each migration period and the target system availability.

**How to Initiate a Proactive Service Migration?** How to properly initiate a proactive service migration is quite tricky. We cannot depend on the primary to initiate a proactive recovery because it might be faulty. Therefore, the migration initiation must involve all replicas.

On expiration of the migration timer, a replica chooses a set of $f$ active replicas, and a set of $f$ standby nodes, and multicasts an INIT-MIGRATION request to all other replicas in the form $<$INIT-MIGRATION$, v, l, S, D, i>_{\sigma_i}$, where $v$ is the current view, $l$ is the migration number (determined by the number of successful migration rounds recorded by replica $i$), $S$ is the set of identifiers for the $f$ active replicas to be migrated, $D$ is the set of identifiers for the $f$ standby nodes as the targets of the migration, $i$ is the sending replica id, and $\sigma_i$ is the authenticator for the message.

On receiving an INIT-MIGRATION message, a replica $j$ accepts the message and stores the message in its data structure provided that the message carries a valid authenticator, it has not accepted an INIT-MIGRATION message from the same replica $i$ in view $v$ with the same or higher migration number, and the replicas in $S$ and $D$ are consistent with the sets determined by itself according to the selection algorithm (to be introduced next).

Each replica waits until it has collected $2f+1$ INIT-MIGRATION messages from different replicas (including its own INIT-MIGRATION message) before it constructs a MIGRATION-REQUEST message. The MIGRATION-REQUEST message has the form $<$MIGRATION-REQUEST$, v, l, S, D>_{\sigma_p}$. The primary, if it is correct, should place the MIGRATION-REQUEST message at the head of the request queue and order it immediately. The primary orders the MIGRATION-REQUEST in the same way as that for a normal request coming from a client, except that (1) it does not batch the MIGRATION-REQUEST message with normal requests, and (2) it piggybacks the MIGRATION-REQUEST and the $2f+1$ INIT-MIGRATION messages (as proof of validity of the migration request) with the PRE-PREPARE message. The reason for ordering the MIGRATION-REQUEST is to ensure a consistent synchronization point for migration at all replicas. An illustration of the migration initiation protocol is

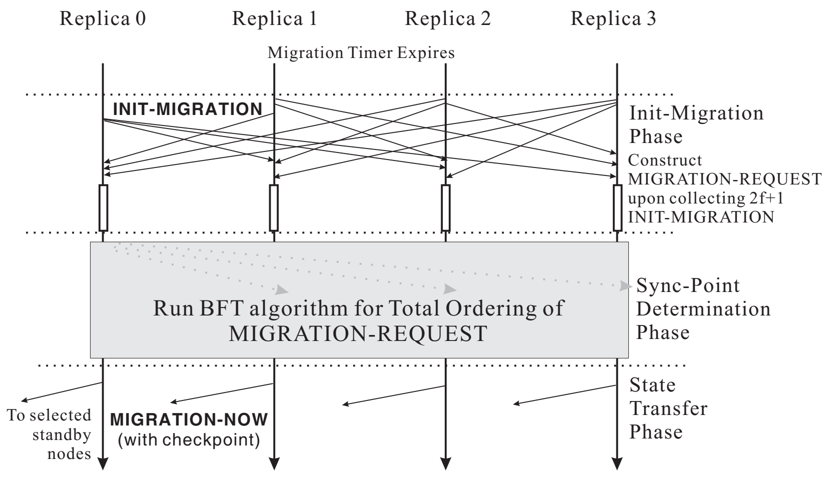
Replica 0
Replica 1
Replica 2
Replica 3
Migration Timer Expires
INIT-MIGRATION
Init-Migration Phase
Construct MIGRATION-REQUEST upon collecting 2f+1 INIT-MIGRATION
Run BFT algorithm for Total Ordering of MIGRATION-REQUEST
Sync-Point Determination Phase
State Transfer Phase
To selected standby nodes
MIGRATION-NOW (with checkpoint)

**Figure 3. The proactive service migration protocol.**

shown as part of Figure 3.

Each replica starts a view change timer when the MIGRATION-REQUEST message is constructed so that a view change will be initiated if the primary is faulty and does not order the MIGRATION-REQUEST message. The new primary, if not faulty, should continue this round of proactive migration.

**Migration Set Selection.** The selection of the set of active replicas to be migrated is relatively straightforward. It takes four rounds of migration (each round for $f$ replicas) to proactively recover all active replicas at least once. The replicas are recovered according to the reverse order of their identifiers, similar to that used in [2]. For example, for the very first round of migration, replicas with identifiers of $3f, 3f-1, ..., 2f+1$ will be migrated, and this will be followed by replicas with identifiers of $2f, 2f-1, ..., f+1$ in the second round, replicas with identifiers of $f, f-1, ..., 1$ in the third round, and finally replicas with identifiers of $0, 3f, ...2f+2$ in the fourth round. Note that only replica $0$ is required to be migrated in the fourth round. We choose to migration $f$ replicas in this round anyway because (1) it is easier to implement this selection algorithm, and (2) the cost of having $f$ parallel proactive recovery operations is not much more than that of a single replica, as analyzed in [2]. (The example assumed $f > 2$. It is straightforward to derive the selections for the cases when $f = 1, 2$.) The selection is deterministic and can be easily computed based on the migration number. Note that the migration number constitutes part of the middleware state and will be transfered to all recovering replicas. The selection is independent of the view the replicas are in.

The selection of the set of standby nodes as the target of migration is based on the elapsed time since the standby nodes were last sanitized. That is why each replica keeps track of the join time of each standby node. For each round of migration, the $f$ standby nodes with the least elapsed time will be chosen. This is because the probability of these nodes to have been compromised at the time of migration is the least (assuming brute-force attacks by adversaries).

**Migration Synchronization Point Determination.** It is important to ensure all (correct) replicas to use the same synchronization point when performing the service migration. This is achieved by ordering the MIGRATION-REQUEST message. The primary starts to order the message by sending a PRE-PREPARE message for the MIGRATION-REQUEST to all backups, as described previously.

A backup verifies the piggybacked MIGRATION-REQUEST in a similar fashion as that for the INIT-MIGRATION message, except now the replica must check that it has received all the $2f+1$ init-migration messages that the primary used to construct the MIGRATION-REQUEST, and the sets in $S$ and $D$ match those in the INIT-MIGRATION messages. The backup requests the primary to retransmit any missing INIT-MIGRATION messages. The backup accepts the PRE-PREPARE message for the MIGRATION-REQUEST provided that the MIGRATION-REQUEST is correct and it has not accepted another PRE-PREPARE message for the same sequence number in view $v$. From now on, the replicas execute according to the three-phase BFT algorithm [2] as usual until they commit the MIGRATION-REQUEST.

**State Transfer.** When it is ready to execute the MIGRATION-REQUEST, a replica $i$ takes a checkpoint of its state (both the application and the BFT middleware state), and multicasts a MIGRATE-NOW message to the $f$ standby nodes selected and all replicas of the configuration manager. The MIGRATE-NOW message has the form $<$MIGRATE-NOW$, v, n, C, P, i>_{\sigma_i}$, where $n$ is the sequence number assigned to the MIGRATION-REQUEST, $C$ is the digest of the checkpoint, and $P$ contains $f$ tuples. Each tuple contains the identifiers of a source-node and target-node pair $<s, d>$. The standby node $d$, once completes the proactive recovery procedure, assumes the identifier $s$ of the active node it replaces. A replica sends the actual checkpoint (together with all queued request messages, if it is the primary) to the target nodes in separate messages.

If a replica belongs to the $f$ nodes to be migrated, it is expected to disable the NIC1 interface and stops accepting new request messages. Of course, if the replica is faulty, it might not do that. That is why the trusted configuration manager must be informed of the migration by all correct active replicas (it will take action only if it has received $f + 1$ migration-now notifications). In case the faulty replica fails to comply, the configuration manager changes the switch configuration to forcefully disable the NIC1 interface (from the switch end) and performs other sanitizing operations on the faulty node.

Before a standby node can be promoted to run an active replica, it must collect $2f+1$ consistent MIGRATE-NOW messages with the same sequence number and the digest of the checkpoint from different active replicas. Once a standby node obtains a stable checkpoint, it applies the checkpoint to its state and starts to accept clients' requests and participate the BFT algorithm as an active replica.

## 3.3. New Membership Notification

A faulty node could continue sending messages to the active replicas and the clients, even if it has been migrated, before it is sanitized by the configuration manager. It is important to inform the clients the new membership so that they can ignore such messages sent by the faulty replica. The membership information is also important for the clients to accept messages sent by new active replicas, and to send messages to these replicas. This is guaranteed by the new membership notification mechanism.

The new membership notification is performed in a lazy manner to improve the performance unless a new active replica assumes the primary role, in which case, the notification is sent immediately to all known clients (so that the clients can send their requests to the new primary). Furthermore, the notification is sent only by the existing active replicas (*i.e.*, not the new active replicas because the clients do not know them yet). Normally, the notification is sent to a client only after the client has sent a request that is ordered after the MIGRATION-REQUEST message, *i.e.*, the sequence number assigned to the client's request is bigger than that of the MIGRATION-REQUEST.

The notification message has the form $<$NEW-MEMBERSHIP$, v, n, P, i>_{\sigma_i}$ (basically the same as the MIGRATION-NOW message without the checkpoint), where $v$ is the view in which the migration occurred, and $n$ is the sequence number assigned to the MIGRATION-REQUEST, and $P$ contains the tuples of the identifiers for the replicas in the previous and the new membership.

## 3.4. On-Demand Migration

On-demand migration can happen when the configuration manager detects a node to be faulty after it has been promoted to run an active replica. It can also happen when replicas have collected solid evidence that one or more replicas are faulty, such as a lying primary. The on-demand migration mechanism is rather similar to that for proactive recovery, with only two differences: (1) The migration is initiated on-demand, rather than by a migration timeout. However, replicas still must exchange the INIT-MIGRATION messages before the migration can take place; (2) The selection procedure for the source node is omitted because the nodes to be swapped out are already decided, and the same number of target nodes are selected accordingly.

## 4. Performance Evaluation

The proactive service migration mechanisms have been implemented and incorporated into the BFT framework developed by Castro, Rodrigues and Liskov [2, 3]. Due to the potential large state, an optimization has been made, *i.e.,* instead of every replica sending its checkpoint to the target nodes of migration, only one actually sends the full checkpoint. The target node can verify if the copy of the full checkpoint is correct by comparing the digest of the checkpoint with the digests received from other replicas. If the checkpoint is not correct, the target node asks for a retransmission from other replicas.

Similar to [2], the performance measurements are carried out in general-purpose servers without hardware coprocessors. The related operations are simulated in software. Furthermore, the trusted configuration manager is not fully implemented because we currently lack the sophisticated hardware equipment that could facilitate the dynamic control of subnet formation. All the components (the configuration manager, the three pools of replicas, and the clients) are located in the same physical local area network. The motivation of the measurements is to assess the runtime performance of the proactive service migration scheme.

Our testbed consists of a set of Dell SC440 servers connected by a 100 Mbps local-area network. Each server is equipped with a single Pentium dual-core 2.8GHz CPU and 1GB of RAM, and runs the SuSE Linux 10.2 operating system. The micro-benchmarking example included in the original BFT framework is adapted as the test application. The request and reply message length is fixed at 1KB, and each client generates requests consecutively in a loop without any think time. Each server replica injects a $1ms$ processing delay (using busy waiting) to simulate some actual workload before it echoes the payload in the request back to the client.

We carry out two sets of experiments. In the first set, we profile the runtime cost of the service migration scheme with a fixed migration period, and in the second set, we characterize how the migration period can be dynamically adjusted under various conditions. In each set of experiments, we use two configurations, one to tolerate a single faulty replica in each component (referred to as the f=1 configuration where there are 4 replicas in each component, except the trusted configuration manager, which employs 3 replicas), and the other to tolerate two faulty replicas in each component (referred to as the f=2 configuration where there are 7 replicas in each component, except the trusted configuration manager, which has 5 replicas). Up to eight concurrent clients are used in each run.

## 4.1. Runtime Cost of Service Migration

To characterize the runtime cost of the service migration scheme, we measure the recovery time for a single replica with and without the presence of clients, and the impact of proactive migration on the system performance perceived by clients. In each run, the service migration interval is kept at $10s$. The recovery time is determined by measuring the time elapsed between the following two events: (1) the primary sending the PRE-PREPARE message for the MIGRATION-REQUEST, and (2) the primary receiving a notification from the target standby node indicating that it has collected and applied the latest stable checkpoint. We refer to this time interval as the service migration latency. The impact on the system performance is measured at the client by counting the number of calls it has made during a period of $50s$, which is slightly longer than four rounds of service migration (so that all replicas have been proactively recovered once), with and without proactive migration-based recovery.

Figure 4 summarizes the measured service migration latency with respect to various state sizes and the number of concurrent clients. Figure 4(a) and (c) show the service migration latency for various state sizes (from 1MB to about 80MB) for the two configurations (labeled as “With f=1” and “With f=2”), respectively. It is not surprising to see that the cost of migration is limited by the bandwidth available (100Mbps) because in our experiment, the time it takes to take a local checkpoint (to memory) and to restore one (from memory) is negligible. This is intentional for two reasons: (1) the checkpointing and restoration cost is very application dependent, and (2) such cost is the same regardless of the proactive recovery schemes used. The migration latency is slightly larger when the replication degree is higher, especially with the presence of a client.

Furthermore, we measure the migration latency as a function of the system load in terms of the number of concurrent clients. The results for the two configurations are shown in Figure 4(b) and (d). As can be seen, the migration latency increases more significantly for larger state when the system load is higher. When there are eight concurrent clients, the migration latency for a state size of 50MB is close to $10s$. This observation suggests that if a fixed watchdog timer is used, the watchdog timeout must be set to a very conservative worst-case value. If the watchdog timeout is too short for the system to go through four rounds of proactive recovery (of $f$ replicas at a time), there will be more than $f$ replicas going through

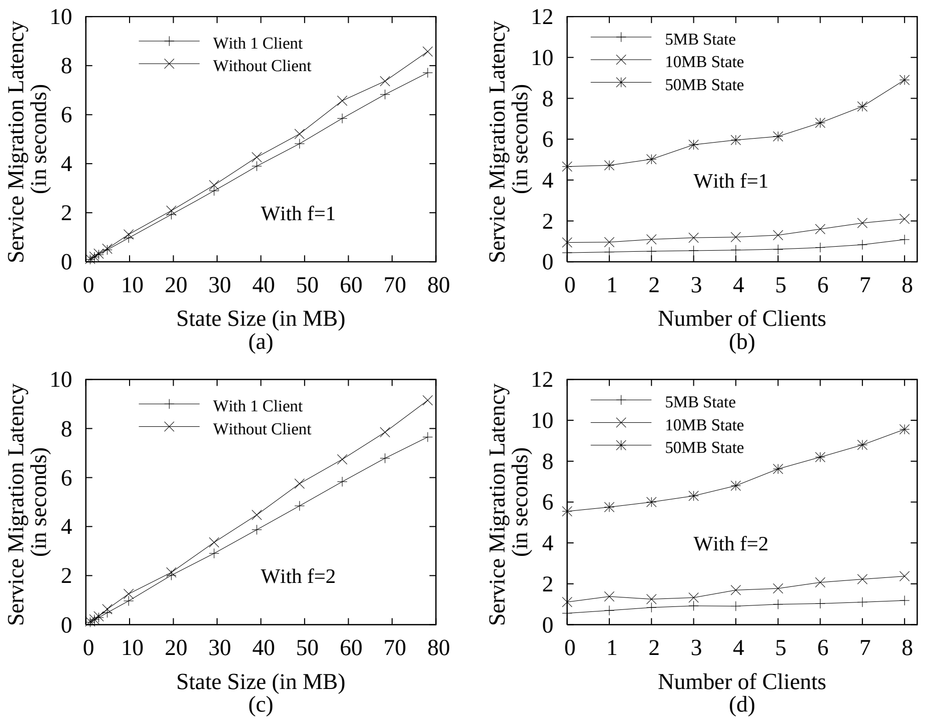

**Figure 4. (a) Service migration latency for different state sizes measured when (1) the replicas are idle (other than the service migration activity), labeled as “Without Client” and (2) in the presence of one client, for the f=1 configuration. (b) The impact of system load on the migration latency for the f=1 configuration. (c) Migration latency with respect to the state size for the f=2 configuration. (d) Migration latency with respect to the system load for the f=2 configuration.**

proactive recoveries concurrently, which will decrease the system availability, even without any fault.

Figure 5 shows the performance impact of proactive service migration as perceived by a single client in terms of the number of calls made in a 50-second interval. During this period, 4 rounds of migration would take place so that all replicas can be proactively recovered at least once. As can be seen, the impact of proactive migration on the system performance is quite acceptable. For a state smaller than 30MB, the throughput is reduced only by 10% or less comparing with the no-proactive-recovery case.

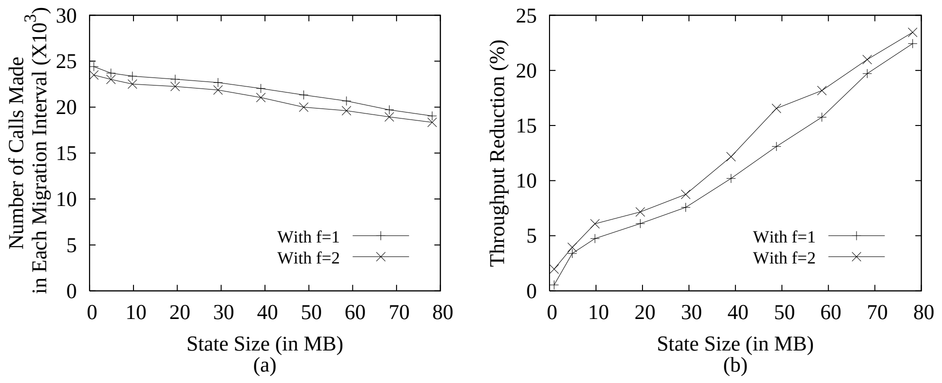

**Figure 5. The impact of service migration on the system throughput. (a) Number of calls made by a single client during a 50-second interval for various state sizes. (b) The percentage reduction in system throughput for various state sizes.**

## 4.2. Dynamic Adjustment of Migration Interval

To demonstrate the capability of dynamic adjustment of the migration interval, we carry out another set of experiments. We assume the following parameters are supplied by the user:

- Worst-case response time $T_{oe}^{0} = 5ms$,
- Worst-case migration latency $T_{s}^{0} = 10s$,
- Minimum number of requests processed in each migration round $p^{0} = 10,000$.

This gives a target availability of 83.3%. The objective of these experiments is to show how the migration interval changes under the following two scenarios: (1) different system loads due to the presence of concurrent clients, and (2) different system loads due to the state size changes.

During runtime, our mechanism profiles the average response time $T_{oe}$ and the migration latency $T_s$, and calculates the appropriate number of requests to be processed $p$ for the next round of migration based on Equation 2, which determines the migration timeout value for the next round of migration.

The results for scenario (1) with the f=1 configuration are shown in Figure 6(a) and (b), where (a) shows the $p$ values, and (b) illustrates the corresponding migration timeout values. When the replica state is kept at 5MB and 10MB, the given

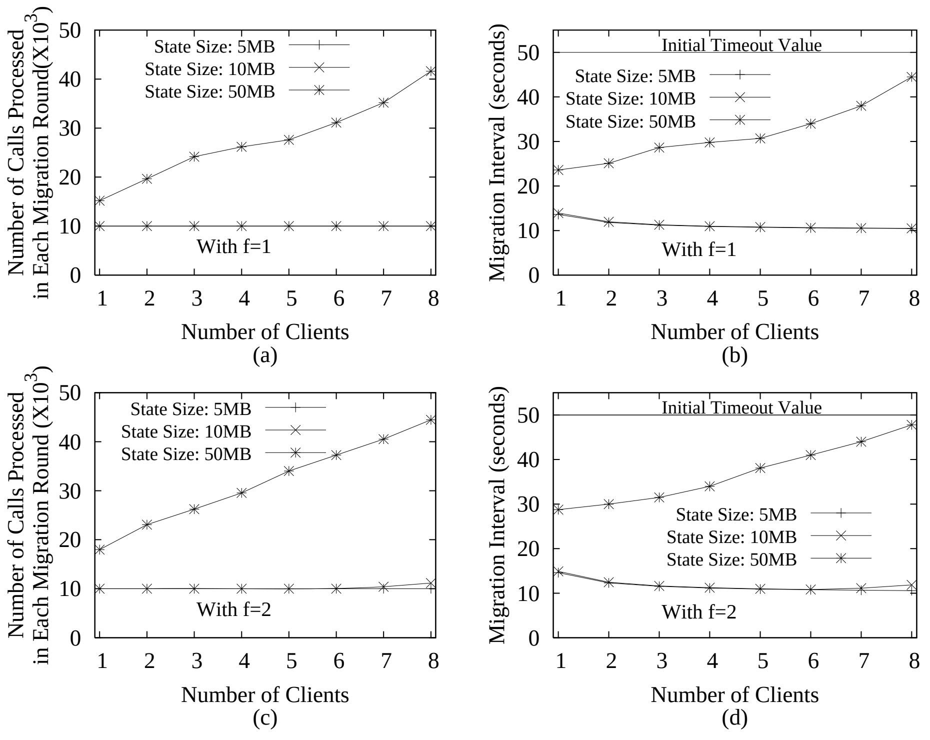

**Figure 6. Dynamic adaptation of migration interval. (a) Number of requests processed in each round of migration for 3 different state sizes in the presence of difference number of concurrent clients, for the f=1 configuration. (b) The corresponding migration interval with respect to the number of concurrent clients, for the f=1 configuration. (c) Number of requests processed in each round of migration under different number of clients, for the f=2 configuration. (d) The corresponding migration interval with respect to the number of concurrent clients, for the f=2 configuration.**

minimum number of requests $p^0 = 10,000$ is used because the calculated $p$ value to meet the target availability requirement is smaller. For the state size of 50MB, higher number of requests than $p^0$ are processed in each migration interval in order to meet the availability requirement because the migration latency is very significant. It is interesting to note that for state sizes of 5MB and 10MB, the migration timeout value actually decreases when the number of concurrent clients increases. This might appear to be counterintuitive. However, it can be easily explained. This is an artifact caused by the aggressive batching mechanism in the BFT framework [2] we used. With batching, the cost of message ordering per request is reduced. Consequently, the response time for each request is reduced, which results in a smaller migration timeout value. (Recall that

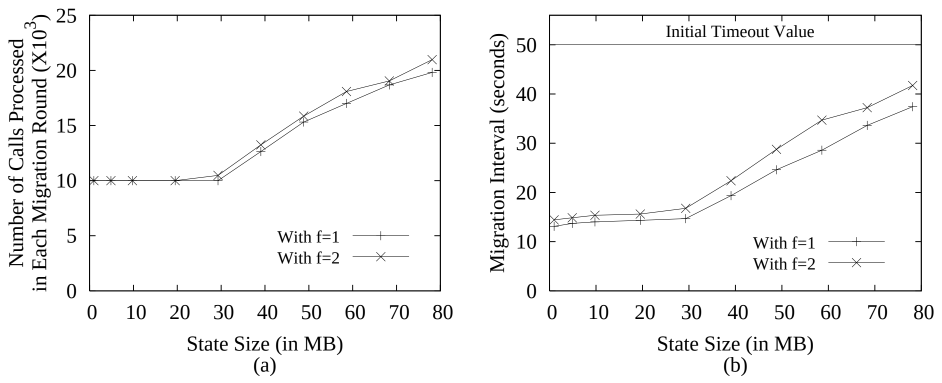

**Figure 7. Dynamic adaptation of migration interval. (a) Number of requests processed in each round of migration when the replica state changes in the presence of a single client. (b) The corresponding migration interval with respect to the state size.**

$T_{oe}$ does not include the queueing delay of the request being ordered.) Another interesting observation is that the migration timeout values determined at runtime are much smaller than the worst-case value except when the state size is large and the number of concurrent clients is significant. Figure 6(c) and (d) present similar results for the f=2 configuration. When the replication degree is higher, the migration interval is slightly larger as expected.

For many applications, their state size might gradually increase over time as they process more application requests. A larger state would mean larger migration latency, which normally would lead to a larger $p$ value, as indicated in Equation 2. Figure 7 shows the results of the dynamic adaptation of migration interval in the presence of a single client, for both the f=1 and f=2 configurations. As expected, when the state size is relatively small (20MB or below), $p^0$ is used because the migration latency is small and the number of requests needed to meet the availability requirement is smaller than $p^0$. As the state size increases further, larger $p$ value is needed to meet the availability requirement. Again, we show that the migration interval dynamically determined are much smaller than the worst-case value except when the state size is very large.

## 5. Related Work

Ensuring Byzantine fault tolerance for long-running systems is an extremely challenging task. Proactive recovery [17] is regarded as a fundamental technique to defend against mobile attackers, which might compromise multiple servers over time. The pioneering work in the context of Byzantine fault tolerance is carried out by Castro and Liskov [2, 20]. Our work is inspired by their work. The comparison of our scheme and closely related work has been provided in introduction section.

Proactive recovery for intrusion tolerance has been studied in [15, 26] with the emphasis of confidentiality protection using proactive threshold cryptography [1]. Reboot is also used as the basis to recover compromised replicas, which suggests such schemes may also suffer from similar problems as those in [2].

The idea of moving expensive operations off the critical execution path is a well-known system design strategy, and it has been exploited in other fault-tolerant systems, such as [14, 18, 19]. In our scheme, this principle is used to reduce the vulnerability window.

The reliance on extra nodes beyond the $3f+1$ active nodes in our scheme may somewhat relate to the use of $2f$ additional witness replicas in the fast Byzantine consensus algorithm [16]. However, the extra nodes are needed for completely different purposes. Nevertheless, one might suggest that we should utilize the extra standby nodes (if there are $2f$ or more of such nodes) and apply the fast Byzantine consensus algorithm for fast response time. While it is certainly possible to do so, it is effective only if the vulnerability window is very large because the sanitizing operations (such as reboot), which can be time-consuming, would now be in the critical path of application requests handling. This appears to be orthogonal to the objective of this research.

Finally, other researchers have done substantial work on the availability and reliability analysis of fault-tolerant systems, such as [6, 7, 8, 9, 10]. These results could potentially be used to enhance the migration interval determination algorithm of our scheme.

## 6. Conclusion

In this paper, we presented a novel proactive recovery scheme based on service migration for long-running Byzantine fault tolerant systems. We described in detail the challenges and mechanisms needed for our migration-based proactive recovery to work. The primary benefit of our migration-based recovery scheme is a smaller vulnerability window during normal operation. When the system load is light, the migration interval can be dynamically adapted to a smaller value from the initial conservative value, which is usually set based on the worst-case scenario, and hence, resulting in a smaller vulnerability window. Our scheme also shifts the time-consuming repairing step out of the critical execution path, which also contributes to a smaller vulnerability window. We demonstrated the benefits of our scheme experimentally with a working prototype.

For future work, we plan to enhance the features of the trusted configuration manager, in particular, the incorporation of the code attestation methods [4] into the fault detection mechanisms, and the application of the migration-based recovery scheme to practical systems such as networked file systems.

## Acknowledgements

We sincerely thank the anonymous reviewers for their invaluable comments and suggestions. This research has been supported in part by the US NSF grant CNS 08-21319, and by a Faculty Research Development award from Cleveland State University.

## References

[1] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, Washington, DC, 2002.

[2] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

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

[4] B. Chen and R. Morris. Certifying program execution with secure processors. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 133–138, Lihue, HI, May 2003.

[5] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations*, pages 177–190, Seattle, WA, November 2006.

[6] Y. Dai, G. Levitin, and K. Trivedi. Performance and reliability of tree-structured grid services considering data dependence and failure correlation. *IEEE Transactions on Computers*, 56(7):925–936, 2007.

[7] Y. Dai, Y. Pan, and X. Zou. A hierarchical modeling and analysis for grid service reliability. *IEEE Transactions on Computers*, 56(5):681–691, 2007.

[8] Y. Dai, M. Xie, Q. Long, and S. Ng. Uncertainty analysis in software reliability modeling by bayesian analysis with maximum-entropy principle. *IEEE Transactions on Software Engineering*, 33(11):781–795, 2007.

[9] Y. Dai, M. Xie, and K. Poh. Modeling and analysis of correlated software failures of multiple types. *IEEE Transactions on Reliability*, 54(1):100–106, 2005.

[10] Y. Dai, M. Xie, and X. Wang. Heuristic algorithm for reliability modeling and analysis of grid systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 37(2):189–200, 2007.

[11] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, April 1985.

[12] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32:18–25, 2001.

[13] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[14] M. Malek, A. Polze, and W. Werner. A framework for responsive parallel computing in network-based systems. In *Proceedings of International Workshop on Advanced Parallel Processing Technologies*, pages 335–343, Bejing, China, September 1995.

[15] M. A. Marsh and F. B. Schneider. Codex: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January 2004.

[16] J. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July-September 2006.

[17] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 51–59, Montreal, Quebec, Canada, 1991.

[18] A. Polze, J. Schwarz, and M. Malek. Automatic generation of fault-tolerant corba-services. In *Proceedings of Technology of Object-Oriented Languages and Systems*, pages 205–213, Santa Barbara, CA, 2000. IEEE Computer Society Press.

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

[20] R. Rodrigues and B. Liskov. Byzantine fault tolerance in long-lived systems. In *Proceedings of the 2nd Workshop on Future Directions in Distributed Computing*, June 2004.

[21] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *Proceedings of the IEEE Pacific Rim Dependable Computing Conference*, pages 373–380, 2007.

[22] P. Sousa, N. F. Neves, and P. Verissimo. Proactive resilience through architectural hybridization. In *ACM Symposium on Applied Computing*, pages 686–690, Dijon, France, 2006.

[23] P. Sousa, N. F. Neves, P. Verissimo, and W. H. Sanders. Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 71–82, 2006.

[24] S. J. Vaughan-Nichols. Virtualization sparks security concerns. *Computer*, 41(8):13–15, August 2008.

[25] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 253–267, Bolton Landing, NY, 2003.

[26] L. Zhou, F. Schneider, and R. van Renesse. Coca: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.