Proceedings of the
17th International Conference on Parallel and Distributed Computing Systems
San Francisco, California, pp. 241-247, September 2004.

Practical Byzantine Fault Tolerance Using Fewer than 3f+1 Active Replicas

Ming Li and Yuval Tamir
Concurrent Systems Laboratory
Computer Science Department, UCLA
Los Angeles, California 90095
{mli,tamir}@cs.ucla.edu

Abstract

Byzantine fault tolerant state machine replication (BFT-SMR) is a foundation for implementations of highly reliable services. Existing algorithms for BFT-SMR require at least $3f+1$ *active* replicas to tolerate f faulty replicas. We show that BFT-SMR can be achieved with fewer than $3f+1$ *active* replicas, as long as standby spare replicas are available, such that the number of active replicas plus the number of spares is at least $3f+1$. In particular, a single Byzantine fault can be tolerated with only three active replicas and a single standby spare. The spares are rarely used — only for reconfigurations triggered by faults or suspected faults. Each reconfiguration leads to a different subset of nodes running the active replicas. By reducing the number of active replicas, our algorithm reduces power consumption for processing and communication. It also has the potential to provide better performance. We present a 1-resilient algorithm that achieves BFT-SMR with only three active replicas and one standby spare, and show the results of preliminary performance evaluation.

1. Introduction

Reliable services can be implemented as deterministic state machines that are replicated across multiple nodes [16, 11]. Reliability is maximized if the replication algorithm can tolerate arbitrary (Byzantine) faults. Byzantine fault-tolerant state machine replication (BFT-SMR) has been shown to be feasible and practical even in asynchronous systems [4, 5, 13], subject to weak partial synchrony assumptions.

Existing BFT-SMR algorithms require at least $3f+1$ replicas to tolerate up to f faulty replicas. All replicas in those algorithms actively participate in the agreement on a total order of client requests. In most algorithms, the same $3f+1$ replicas also process the requests and generate replies. In a recently published algorithm [17], the nodes that agree on request order are separate from the nodes that actually process (execute) the requests. Only $2f+1$ execution replicas are required while the requirement for $3f+1$ agreement replicas is maintained.

In this paper, we show that the replication costs for BFT-SMR can be further reduced, using fewer than $3f+1$ active replicas for *both* agreement and execution. In addition to the active replicas, the scheme requires standby spares, such that the number of active replicas plus the number of spares is at least $3f+1$. Standby spares are involved in the algorithm only for reconfigurations when the active replicas fail to make progress (due to faulty replicas or simply due to unexpectedly long delays). Each such reconfiguration leads to a different subset of nodes running the active replicas. Eventually, the system reaches a configuration where the $2f+1$ active replicas are fault-free and thus able to make progress.

Our algorithm provides the same Byzantine resiliency and service characteristics as previous algorithms for BFT-SMR under the same synchrony assumptions. The use of fewer active replicas results in reduced power assumption for processing and communication. It also has the potential to provide better performance during normal, fault-free execution since each active replica communicates with fewer replicas.

The next section introduces the basic ideas that allow fewer than $3f+1$ active replicas to be used for BFT-SMR. The system model and key assumptions are described in Section 3. A complete algorithm that tolerates a single Byzantine fault is presented in Section 4. Tolerating multiple faults is addressed in Section 5. In Section 6, experimental results are used to compare the normal-case performance of our algorithm to that of the algorithm in [4]. Related work is discussed in Section 7.

2. BFT-SMR with Fewer than 3f+1 Active Replicas

The system implementing BFT-SMR is based on a primary replica and multiple backup replicas [4]. Clients send requests to the primary. The primary assigns a sequence number to each request and multicasts it to the backups. An agreement protocol ensures that fault-free replicas agree on the message order. Each fault-free replica then executes the request and sends a reply to the client. The client accepts the reply upon receiving at least $f+1$ consistent replies. If the client does not receive replies ''soon enough,'' it times out and broadcasts the request to all the replicas. Any replica that is not a primary and has not received the request previously, forwards it to the primary. If the primary does not multicast the request to the group, a *view change* is eventually triggered. This ensures liveness even if the primary fails [4].

An f-resilient service based on BFT-SMR requires $3f+1$ replicas to ensure both safety and liveness [4, 3]. This requirement holds even in a system where messages can be authenticated using cryptographic techniques [3, 7]. Our work is based on the observation that safety can be assured if the total number of replicas is $2f+1$ and each correct replica proceeds only after communicating with the other $2f$ replicas. Hence, in our system, there are only $2f+1$ active replicas. Thus, the system may either generate the correct result or it may fail to make progress if some of the faulty replicas do not respond.

Our approach to dealing with failure of the system to make progress is to exploit the fact that, even with $3f+1$ active replicas, existing schemes already require *view changes* triggered by timeouts to ensure liveness for the case of primary replica failure [4]. While there are only $2f+1$ active replicas in our system, additional standby spares participate in view changes that are triggered by the same timeout mechanism. Each view change results in a different subset of the replicas being active. Eventually, the system can make progress after reaching a configuration where there are $2f+1$ *fault-free* active replicas. Since a view change must transform a standby spare into an active replica, the protocol includes a mechanism that allows the standby to obtain the correct state from the remaining active replicas.

3. System Model and Assumptions

The system model is the same as the one used in [4]. We consider an asynchronous distributed system consisting of nodes connected by a network. Communication is unreliable: messages may be lost, delayed, duplicated, or delivered out of order.

We do not assume a completely asynchronous system model since it is impossible to solve the consensus problem in such an asynchronous system [8]. We make the *partial synchrony* assumption that was introduced by Dwork et al. [7]: the bounds on message delay and relative speeds of different nodes exist but they are not known and they hold only after some unknown time. Our algorithm relies on partial synchrony to provide liveness. Other schemes rely on similar synchrony assumptions by relying on timeouts [4].

We assume a Byzantine failure model: faulty processes can exhibit arbitrary (Byzantine) behavior, limited only by the restriction stated below. Fault-free replicas must be deterministic — the execution of an operation in a given state and with a given set of arguments must always produce the same result [4].

We assume that cryptographic techniques are used to prevent faulty processes spoofing or replaying other processes' messages. A process can verify the content and the original sender of a message, even if the message has been relayed by other processes. Thus, the fault model we consider is the *authenticated Byzantine model* [10]. The message authentication mechanism can employ a public-key cryptosystem such as RSA [14]. Each process can digitally sign its messages with a private key. Each process obtains the public keys of other processes to verify signed messages. The signatures are unforgeable. We use a message-digest algorithm such as MD5 [15] to compress a message of arbitrary length into a fixed-length message digest. A process signs the digest of a message instead of signs the entire message.

4. A 1-Resilient BFT-SMR Algorithm Using Three Active Replicas

In this section we present an algorithm that can tolerate a single Byzantine fault using three active server replicas and one spare replica. We identify each replica using a number in $\{0,1,2,3\}$. Throughout the paper we denote a message m signed by replica i as $\langle m \rangle_i$. The algorithm provides both safety and liveness under the single fault assumption. Safety requires agreement among all correct replicas on the total ordering of client requests. Liveness requires that the algorithm eventually makes progress despite faults, i.e., clients eventually receive correct replies to their requests. The algorithm relies on partial synchrony to provide liveness.

The algorithm is based on configurations of the server replicas called *views*. A view is identified by a unique number v. View v can be succeeded by view $v+1$. The primary of view v is replica i, such that $i = v \bmod 4$; the replica identified by $(v+3) \bmod 4$ is the spare, and the remaining two replicas are backups. Thus, in every fourth view, the same configuration is reused, but with a new view number. During normal operation, the standby spare replica does not send or receive any messages and does not perform any operations. The algorithm starts with view 0. View changes are carried out by a reconfiguration protocol (Section 4.2).

Active replicas send the current view number with each message they send to the clients, so that view changes are visible to the clients. A client c sends its requests to the replica that it believes is the current primary. The request has the form $\langle request, ts, —, c \rangle_c$, where *ts* is the timestamp that is used to ensure the request is executed exactly once by the server replicas, and ''—'' represents the content of the request. Using the protocol described below, the primary atomically multicasts the request to the backups. All active replicas then execute the operation for the request and send a reply directly to the client. The client accepts the result after receiving consistent replies from two different replicas.

A client sets a timeout when it sends a request to the primary. If it does not receive valid replies before the timeout, it sends the request to all replicas. This step may indirectly trigger the reconfiguration protocol (see Section 4.2) and is critical for ensuring liveness [4].

4.1. Normal-Case Operation

The protocol for normal-case operation in our algorithm is very similar to the *three-phase* protocol in [4]. The only difference is that in our algorithm only three replicas, instead of four, participate in the protocol.

When a server replica receives a client request, it checks the timestamp of the request. If it has already processed the request, it re-sends the reply to the client. If the replica is a backup and it has not received a pre-prepare message (described below) for the request from the primary yet, it relays the request to the primary.

When the primary (denoted p) receives a client request m for the first time, it starts the three-phase ordering protocol. In the first phase, the *pre-prepare* phase, the primary assigns a sequence number s to the request m and multicasts a pre-prepare message with m to all the backups. The pre-prepare message has the form

$\langle pre\text{–}prepare, v, s, d_m \rangle_p$, where v indicates the current view, and d_m is request m's digest.

After the multicast, the primary inserts the request m and the pre-prepare message into it *message log*. Each active replica maintains a message log for recording the messages it receives and it sends to others, in case they are needed for retransmission or as certificates. Messages must be kept in the log until the replica knows that the requests have been executed by all active replicas. A garbage collection mechanism is needed to keep the logs from growing without bound. A description of this mechanism is omitted due to limits on paper length.

A backup b accepts the $\langle pre\text{–}prepare, v, s, d_m \rangle_p$ message and the request m if the following conditions are all true: it is in view v; the pre-prepare message and the request m are all properly signed, by the primary of view v and by the client c, respectively; it has not yet accepted a pre-prepare message for view v with the same sequence number s but a different digest; and predicate *prepared* (m', v', s, b) (discussed below) is not true for a request m' that is different than m for any previous view $v' \leq v$.

If backup b accepts the pre-prepare message, it enters the *prepare phase* by multicasting a prepare message $\langle prepare, v, s, d_m, b \rangle_b$ to all other replicas (including the primary), then adds the request m, the pre-prepare message, and its prepare message to its message log. Otherwise, it simply discards the message.

When a replica (including the primary) receives a prepare message, it accepts and inserts the message into its log provided that the message is properly signed, the view number in the message is the same as the replica's current view.

As in [4], the predicate *prepared*(m, v, s, i) is true if and only if replica i has inserted into its log the request m, the pre-prepare message for m in view v with sequence number s and matching digest, and the prepare messages from the two backups that match the pre-prepare message (including its own prepare message if i is a backup). The pre-prepare message and prepare messages form the *prepared certificate* of m.

When *prepared*(m, v, s, i) becomes true, replica i multicasts a $\langle commit, v, s, d_m, i \rangle_i$ to all other replicas and inserts the message into its log. If a replica receives a commit message that is properly signed by the sender, and the view number v in the message is equal to its current view, it accepts the message and adds to its log.

A replica i can commit to a request m and sequence number s in view v if and only if the following conditions are all true: *prepared*(m, v, s, i) is true; it has the *committed certificate* that consists of three commit messages from different replicas (including its own) with the same sequence number s, the digest d_m of m, and the same view v; and for every request that have a lower sequence number than s, it has either committed to the request, or its state shows the request has been executed (the replica may get the state from another correct replica, as described later). Each replica can then execute the operation requested by m.

After the execution of m, the replica sends a $\langle reply, v, ts, —, i \rangle_i$ to the client, where ''—'' is the result of the execution, and *ts* is the original timestamp of m put by the client. Figure 1 shows the message exchanges in the normal-case protocol.

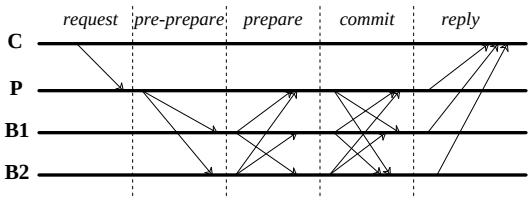

Figure 1: Normal operation. The primary is **P**, **B1** and **B2** are backups, and **C** is a client.

This algorithm provides safety: a correct replica processes a request m in the order indicated by its sequence number s only when it has received consistent commit messages for m and s from all replicas — *prepared*(m, v, s, i) is true for all replicas. At that moment, it knows that all correct replicas will not accept any different $\langle m, s \rangle$ pair in the same view. Therefore, all correct replicas execute the requests in the same order thus produce the same result.

As described so far, the algorithm does not ensure liveness. If a faulty replica sends to others incorrect m and s or simply does not send out messages, the correct replicas may not proceed. However, as mentioned earlier, our complete algorithm does ensure liveness since lack of progress eventually results in a timeout that triggers the reconfiguration protocol which is described next.

4.2. The Reconfiguration Protocol

The *reconfiguration* (or *view-change*) protocol allows the replicas to make progress despite a faulty replica, thus ensuring liveness. The protocol moves the replicas into view $v+1$ from the current view v, thus making the current primary become a standby spare, promoting a backup to be the primary, and bringing in the spare as a new active replica.

In order to initiate the reconfiguration, correct replicas must be able to find out that they are blocked. As in [4], a timeout mechanism is used. If a client does not receive valid replies within a timeout period, it multicasts its request to all replicas. Hence, if the system fails to make progress, every active replica receives the request from the client. Each active replica (*primary or backup*) starts a timer when it receives the request, if the timer is not already running. When the timeout alarm is triggered, the replica starts the reconfiguration protocol as described below. The replica restarts the timer if it still has pending requests waiting for ordering. It stops the timer only when it has committed to all the requests it has received.

Because it is impossible to identify which replica (if any) is faulty, the reconfiguration protocol always ''suspects'' the primary and removes it from the active replica set. The suspicion could be wrong, so the faulty

replica may remain in the active replica set and continue blocking the ordering protocol. However, if that happens, another reconfiguration will be triggered. Since the role of the primary rotates among the four replicas, after at most three consecutive reconfigurations, the faulty replica is removed from the active replica set. The correct replicas can then make progress.

The reconfiguration protocol must ensure correct processing of requests across views. When a spare becomes active as a result of a view change, it must obtain a state consistent with the state of the other replicas, as well as information about requests that have been prepared at correct replicas. This is achieved by having two active replicas agree on the view-change and on a consistent state, and pass the state and prepared certificates to the spare replica. The active replicas exchange view change requests and acknowledgments until two replicas reach an agreement on a common position on processing requests and on their state. Because at least one of them is non-faulty, the agreed state must be consistent with the state of other non-faulty replicas. This state is then sent to the spare replica along with messages that prove its correctness, and existing prepared certificates. Once the spare verifies and restores the state, the system can enter the new configuration.

When a timeout triggers the reconfiguration on replica i in view v, it switches its operation state (mode) to ''*reconfiguration*.'' It continues to process normal messages as in *normal operation* mode. In addition, it multicasts a view-change message to all other active replicas. The message has the form $\langle\, view\text{-}change, v+1, s_i, i \,\rangle_i$, where s_i is the sequence number of the last message that has committed at i.

When a replica j receives the $\langle\, view\text{-}change, v+1, s_i, i \,\rangle_i$ message, if it is not already in reconfiguration for view $v+1$, it discards the message. Otherwise, if the sequence number of the last message it has committed to is higher than or equal to s_i, it sends back to i a $\langle\langle\, view\text{-}change\text{-}ack, v+1, s_j, D_j, \Phi, j, \rangle_j, \Psi \rangle$ message, where s_j is the sequence number of the last message that has committed at j, and D_j is the digest or checksum of its server state after it executes that message. The set Φ contains a set Φ_m for each request m that prepared at j with a sequence number higher than s_j, i.e., request that has prepared but not committed yet at j. Each Φ_m is a set containing a valid pre-prepare message for request m and two matching, valid prepare messages signed by different replicas with the same view, sequence number and digest, i.e., the prepared certificate of m. The piggybacked Ψ is a set of Ψ_m for each request m that has a sequence number s, $s_i < s \le s_j$. Ψ could be a null set if $s_i = s_j$. Each Ψ_m is the committed certificate of m.

When replica i receives a valid $\langle\langle\, view\text{-}change\text{-}ack, v+1, s_j, D_j, \Phi, j \,\rangle_j, \Psi \rangle$ from replica j, if it has not executed a request with a higher sequence number than s_j, it adds the commit messages in each Ψ_m of Ψ to it log and executes all the requests up to sequence number s_j provided that the committed certificate is valid. It then computes the digest of its state and compares it to D_j. If the digest are the same, replica i sends a new-view message to the standby spare. The new-view message has the form $\langle\langle\, new\text{-}view, v+1, s_i, D_i, \Phi, i \,\rangle_i, \mathcal{S}, \mathcal{V} \,\rangle$, where s_i is the sequence number of the last request it has committed (should be equal to s_j), D_i is i's state digest it just computed. $\mathcal{S}$ is the complete state of i corresponding to the digest D_i. The set Φ contains prepared certificate Φ_m for each request m that prepared at i but not yet committed at i. The attached $\mathcal{V}$ is the view-change-ack message received from j, without the piggybacked Ψ. This view-change-ack message is used as a justification to the new-view message.

The standby spare in view v accepts a new-view message for view $v+1$ from any other replica provided: both the new-view message and the piggybacked view-change-ack message are properly signed and contain the same sequence number and state digest; the state $\mathcal{S}$ matches the digest. It then restores its state from $\mathcal{S}$ and go through the Φ sets in both the new-view message and the piggybacked view-change-ack message. If there is at least one valid certificate Φ_m in the two Φ sets for any request m, it adds the messages in Φ_m into its log. This ensures the predicate *prepared* to be also true at the spare for request m, its sequence number s and view v. Once it has finished, it relays the new-view message and the view-change-ack message (without the piggybacked $\mathcal{S}$) to all other replicas and installs view $v+1$.

When a replica in view v receives the relayed new-view message for $v+1$ from the spare replica, it first checks the validity of the message and the piggybacked view-change-ack message. If they are valid, the replica accepts them and installs view $v+1$.

Completion of the reconfiguration protocol with a new view relies on the correct behavior of the spare replica. The spare replica may actually be faulty and may block the reconfiguration. However, the single fault assumption means that if the spare is faulty, none of the current active replicas are faulty. In that case, the reconfiguration was ''incorrectly'' triggered by premature timeouts. Since the active replicas are not really faulty, they do eventually complete the normal case operation and pending requests are executed. When they have no request left for processing, the active replicas abort the reconfiguration and switch back to normal mode. Therefore, either the normal-case operation or the reconfiguration will eventually proceed, preventing the algorithm from blocking indefinitely.

After successfully changing to view $v+1$, the primary p of the new view first creates its own $\langle\, pre\text{-}prepare, v+1, s, d_m \,\rangle_p$ message for every request that has prepared in previous views but not yet committed at p, using the same sequence number and message digest, in consecutive order. If there is a gap in the pre-prepare message set, p creates a special pre-prepare message $\langle\, pre\text{-}prepare, v+1, s, d_{null} \,\rangle_p$, where d_{null} is the digest of special *null* request. A null request is processed like other request but invokes no operation for execution.

After this step, the primary switches back to ''*normal*'' and starts the normal-case protocol for new requests.

The primary of the previous view v becomes the spare in view $v+1$. It discards its state and cleans up its message log, then enters standby mode. Other replicas simply go back to normal state after the view change and proceed as described in Section 4.1.

After changing to view $v+1$, a backup replica may see a gap in the sequence numbers, from the last message it has committed to, to the first pre-prepare message it receives in the new view. In that case, it asks the primary to send the pre-prepare messages again for the missed sequence numbers, using the view number $v+1$.

The reconfiguration protocol ensures that non-faulty active replicas also agree on the sequence numbers of requests that commit locally in different views at different replicas. A request m commits locally at a non-faulty replica with sequence number s in view v only if $prepared(m, v, s, i)$ is true for every non-faulty active replica i in view v. The view $v+1$ will not be installed unless two active replicas in view v agree on the view-change and the standby replica receives both messages (new-view and view-change-ack) from these two replicas. At least one of the two replicas must be correct; the set Φ in its new-view or view-change-ack message then ensures that $prepared(m, v, s, k)$ is also true for the standby spare k. Therefore, the fact that m prepared at every non-faulty active replica in view v is propagated to view $v+1$.

5. Multiple Faults

Although our algorithm is described for single Byzantine fault, it can be easily extended to tolerate multiple simultaneous faults by requiring at least $2f+1$ active replicas. However, the required reconfigurations (view changes) can become expensive.

Ideally, we want to have a minimal number of active replicas in each configuration, that is, $2f+1$ active replicas. The remaining f replicas are standby spares, for a total of $3f+1$ replicas. If configurations are distinguished only by which nodes are the spares, the number of possible configurations is the binomial coefficient $\binom{3f+1}{f}$. Among all these configurations, only one can be guaranteed to consist of only non-faulty replicas as active replicas. With a configuration that has only $2f+1$ active replicas, one faulty replica can block the ordering protocol until a view change. Therefore there is only one configuration that can ensure that the replicas make progress. In the worst case, the system has to iterate through all the possible configurations via view changes to reach this ''clean'' configuration. The average time to reach it grows rapidly when f increases.

An alternative to the algorithm above is to increase the number of active replicas in each configuration to $3f$. This leads to only $3f+1$ possible configurations. Among them there is one configuration with a faulty replica being the standby spare. This configuration can assure progress because it includes at least $2f+1$ non-faulty active replicas. Therefore, the system needs at most $3f$ view changes to get to this configuration. Thus, there is a tradeoff between the number of active replicas and the amount of time a client may have to wait while the server replicas go through a sequence of view changes.

6. Evaluation

With fewer active replicas during normal operation there is a reduction in computation and communication overhead. Although the required number of replicas is $3f+1$, as in existing schemes [4], the standby replicas stay dormant most of the time. Thus, the machines they reside on can be used for other applications. Alternatively, the hardware components (processors, memory, disks, etc.) can be turned off to reduce power consumption through dynamic power management (DPM) [1]. Since the spare replicas are activated only on reconfigurations, which do not occur often, there would be little overhead activating and deactivating the spare replicas.

Another overhead reduction with our scheme is that with fewer active replicas, each replica sends and receives fewer messages during normal operation. This leads to reduced power consumption for communication and may result in better performance for the normal case. To evaluate this adventage, we used a simplified implementation (emulation) to compare the normal-case performance of our algorithm to the algorithm described in [4]. In both cases a single faulty replica can be tolerated and the total number of replicas is four. The evaluation was performed on a network of 350MHz Pentium-II PCs, running Solaris 8, interconnected by a 100Mb switched Ethernet, using TCP/IP for communication. The service operation is a computation that executes in a set amount of time.

For the two algorithms we measured the average response time for requests and the throughput of the service under fault-free operation. The response time is the time interval from when a client sends its request to the primary replica to the time when it receives replies from two different replicas. The response time was measured with the system processing only one request at a time. The throughput is the number of requests per second the replicas are able to process. To show the overhead of the replication algorithms, the results are normalized to the results we measured for the same unreplicated service.

The message authentication in our experiments is based on 512-bit RSA moduli and MD5 digests using the OpenSSL 0.9.7b package. On our PCs, it takes 6.2 milliseconds to generate an RSA signature of an MD5 digest and 0.5 milliseconds to verify a signature. The generation of MD5 digest of a 1KB message takes 30 microseconds.

Figure 2 shows the results for a service that takes 1 millisecond to execute each request. The results show that with the full cost of authentication (''1x''), there is not much difference in response time overhead between the two replication algorithms. The reason for this is the

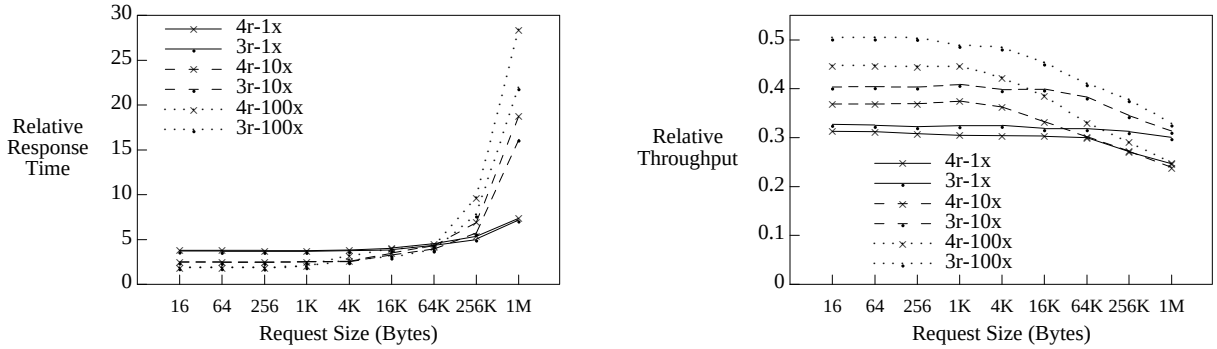

Figure 2: Response time and throughput relative to the unreplicated case, as request size and message authentication overhead vary. ‘‘4r’’ is the algorithm that uses four active replicas [4] while ‘‘3r’’ is our algorithm with three active replicas. Execution time for each request is 1 millisecond. Authentication overhead with speedup of 1x, 10x, or 100x relative to measured overhead.

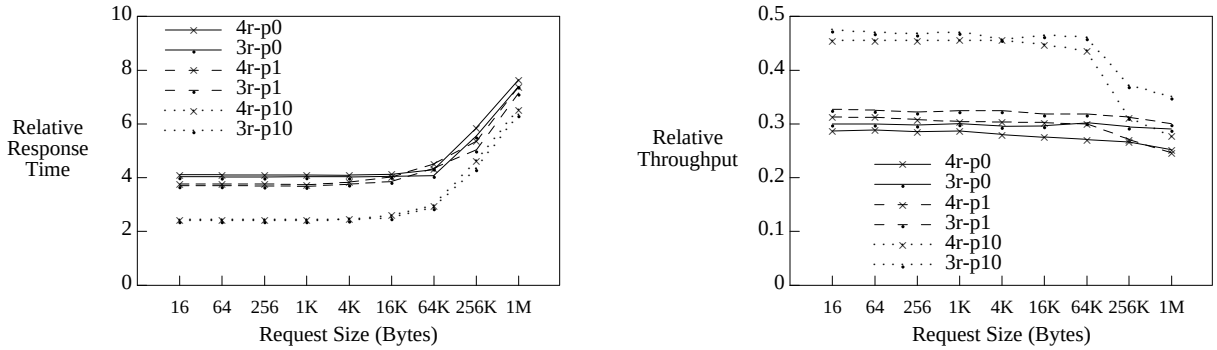

Figure 3: Response time and throughput relative to the unreplicated case, as request size and execution time for requests vary. ‘‘4r’’ is the algorithm that uses four active replicas while ‘‘3r’’ is our algorithm with three active replicas. The message authentication overhead is the actual measured overhead on our system. The execution time in the figure is in milliseconds, e.g., "-p1" means the execution time for each request is 1 millisecond.

high overhead of generating RSA signatures. Although each replica in our algorithm sends and receives fewer messages per request, it performs the same number of multicasts as in the 4-replica algorithm. RSA signatures are only generated once for each multicast and for the reply to the client. Hence, the two algorithms have the same overhead for signing messages. Although the 3-replica algorithm reduces the overhead for sending, receiving and verifying messages, the benefit is insignificant compared to the overhead of signing. In terms of throughput, with the full cost of authentication, our algorithm results in slightly higher performance due to the reduction in the number of messages exchanged.

If fast hardware implementations of the RSA algorithm [2] is used, or faster but less secure cryptography can satisfy the requirement of the service, the overhead for message authentication is reduced. Under these conditions the overhead of signing messages is no longer the dominant factor and there is more of a benefit to using fewer active replicas. This is shown in Figure 2 that present response time and throughput results with authentication that is ten and a hundred times faster.

Figure 3 shows a performance comparison between the two algorithms with various request execution times. The ‘‘-p0’’ results are presented to simulate the cases that the request execution time is insignificant. It would also represent the scenario that the execution is performed on other nodes, separated from the agreement. As the request execution time increases, the performance advantage of the 3-replica algorithm is reduced.

7. Related Work

Consensus and state machine replication in an asynchronous and Byzantine environment has be studied in both theoretical and practical settings [3, 7, 9, 6, 4, 13]. In order to tolerate up to f faulty replicas, all previous algorithms require $3f+1$ *active* replicas for at least part of the processing (agreement on message ordering). The work presented in this paper is derived from the work by Castro and Liskov [4], which provides a complete BFT-SMR solution.

In [4], the view-change procedure is invoked only when the primary fails. With our algorithm, there will be more view changes since any faulty replica may cause the invocation of a view change. However, since, in most environments, faults are rare, the replicas can be expected to spend nearly all their time in normal-case operation. Hence, there is a clear benefit in reducing the overhead

for fault-free execution, even of the cost of increased overhead when there is a fault.

A step towards reducing the replication costs was taken by Yin et al [17]. Their optimization is achieved by separating the agreement protocol that orders requests from the execution (processing) of the requests. The number of replicas that execute the service operation and reply to client requests is reduced to $2f+1$. However, the scheme requires $3f+1$ agreement replicas that actively participate in the agreement protocol for ordering requests. If there is no need for the *privacy firewall* described in [17], the scheme can be executed on $3f+1$ nodes, where $2f+1$ nodes participate in both agreement and execution, while f nodes only participate in the agreement protocol.

The algorithm presented in this paper is a further step in reducing the replication cost by reducing the number of active agreement replicas as well. This may be a significant improvement in cases where the cost of processing a request is small compared to the cost of the agreement protocol. For example, this could be the case with a service where most results are cached and reused to respond to client requests.

The scheme presented in this paper could be integrated with the scheme in [17] to reduce their overhead for reaching agreement. Our algorithm could be used for the agreement phase in [17], with their inter-cluster protocol to drive $2f+1$ execution replicas.

The standby spares in our algorithm are similar to the *witnesses* in the Harp file system [12]. In Harp, a witness is a server replica that does not store actual copies of the file. The witness is only used to facilitate recovery if one of the full replicas fails or if the full replicas are temporarily unable to communicate with each other. Harp was designed to tolerate fail-stop failures only. Our algorithm can be viewed as a Byzantine fault-tolerant extension of the replication method in Harp.

8. Conclusions and Future Work

We presented a new algorithm for Byzantine fault-tolerant state machine replication that reduces replication costs by using fewer active replicas than previous algorithms. Additional standby spare replicas are used only for reconfigurations triggered by faults or suspected faults. This leads to reduced computation and communication resources, resulting in reduced power consumption and, in some circumstances, improved performance.

Our algorithm trades off increased overhead when a replica fails (or appears to fail) for reduced overhead during normal operation. The benefits of our algorithm are most pronounced for handling requests that require relatively little processing. The algorithm can be used to minimize the overhead for agreement in conjunction with an existing scheme [17] that separates agreement from execution for fault-tolerant state machine replication.

We are investigating the feasibility of using symmetric cryptography for message authentication with our algorithm as well as extending the algorithm to tolerate any number of faults over the lifetime of the system provided no more than f replicas are faulty within a window of vulnerability [5].

Acknowledgements

This work was supported by the Jet Propulsion Laboratory under NASA's New Millennium Program.

References

[1] L. Benini, A. Bogliolo, and G. De Micheli, ''A Survey of Design Techniques for System_level Dynamic Power Management,'' *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol.8, no.3 (June 2000).

[2] T. Blum and C. Paar, ''High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware,'' *IEEE Transactions on Computers*, vol.50, no.7, pp. 759-764 (July 2001).

[3] G. Bracha and S. Toueg, ''Asynchronous Consensus and Broadcast Protocols,'' *Journal of the ACM*, vol.32, no.4, pp. 824-840 (October 1985).

[4] M. Castro and B. Liskov, ''Practical Byzantine Fault Tolerance,'' *the 3rd Symposium on Operating Systems Design and Implementation*, pp. 173-186 (February 1999).

[5] M. Castro and B. Liskov, ''Proactive Recovery in a Byzantine-Fault-Tolerant System,'' *the 4th Symposium on Operating Systems Design and Implementation*, pp. 273-287 (October 2000).

[6] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, ''Muteness Failure Detectors: Specification and Implementation,'' *the 3rd European Dependable Computing Conference* , pp. 71-87 (15-17 September 1999).

[7] C. Dwork, N. A. Lynch, and L. Stockmeyer, ''Consensus in the Presence of Partial Synchrony,'' *Journal of the ACM*, vol.35, no.2, pp. 288-323 (April 1988).

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, ''Impossibility of Distributed Consensus with One Faulty Process,'' *Journal of the ACM*, vol.32, no.2, pp. 374-382 (April 1985).

[9] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, ''Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector,'' *the International Conference on Principles of Distributed Systems*, pp. 61-75 (December 1997).

[10] L. Lamport, R. Shostak, and M. Pease, ''The Byzantine Generals Problem,'' *ACM Transactions on Programming Languages and Systems*, vol.4, no.3, pp. 382-401 (July 1982).

[11] L. Lamport, ''Time, Clocks, and the Ordering of Events in a Distributed System,'' *Communications of the ACM*, vol.21, no.7, pp. 558-565 (July 1982).

[12] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams, ''Replication in the Harp File System,'' *the 13th ACM Symposium on Operating Systems Principles*, pp. 226-238 (October 1991).

[13] M. K. Reiter, ''Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart,'' *the 2nd ACM Conference on Computer and Communications Security*, pp. 68-80 (November 1994).

[14] R. L. Rivest, A. Shamir, and L. M. Adleman, ''A Method for Obtaining Digital Signatures and Public-key Cryptosystems,'' *Communications of the ACM*, vol.21, no.2, pp. 120-126 (Feburary 1978).

[15] R. L. Rivest, ''The MD5 Message-Digest Algorithm,'' *Internet RFC-1321* (April 1992).

[16] F. B. Schneider, ''Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial,'' *ACM Computing Surveys*, vol.22, no.4, pp. 299-319 (December 1990).

[17] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, ''Separating Agreement from Execution for Byzantine Fault Tolerant Services,'' *the 19th ACM Symposium on Operating Systems Principles*, pp. 253-267 (October 2003).