Title	A Study on Asynchronous Randomized Consensus Algorithms for Byzantine Fault Tolerant Replication
Author(s)	中村, 純哉
Citation	大阪大学, 2014, 博士論文
Version Type	VoR
URL	https://doi.org/10.18910/34568
rights	
Note	

A Study on
Asynchronous Randomized Consensus Algorithms
for Byzantine Fault Tolerant Replication

Submitted to

Graduate School of Information Science and Technology

Osaka University

January 2014

Junya NAKAMURA

List of Major Publications

Journal Papers

1. Junya Nakamura, Tadashi Araragi, Toshimitsu Masuzawa, and Shigeru Masuyama, "A method of parallelizing consensuses for accelerating byzantine fault tolerance," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 1, 2014. (to appear).

2. Junya Nakamura, Tadashi Araragi, Shigeru Masuyama, and Toshimitsu Masuzawa, "Efficient randomized byzantine fault-tolerant replication based on special valued coin tossing," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 2, 2014. (to appear).

Conference Papers

3. Junya Nakamura, Tadashi Araragi, and Shigeru Masuyama, "Asynchronous byzantine request-set agreement algorithm for replication," in *Proceedings of the 1st AAAC Annual Meeting*, p. 35, 2008.

4. Junya Nakamura, Tadashi Araragi, and Shigeru Masuyama, "Acceleration of byzantine fault tolerance by parallelizing consensuses," in *Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '09, pp. 80–87, Dec. 2009.

Technical Reports

5. Junya Nakamura, Tadashi Araragi, and Shigeru Masuyama, "Byzantine agreement on the order of processing received requests is solvable deterministically in asynchronous systems," in *IEICE Technical Report*, vol. 106 of *COMP2006-35*, pp. 33–40, Oct. 2006.

6. Junya Nakamura, Tadashi Araragi, and Shigeru Masuyama, "Techniques to accelerate request processing for byzantine fault tolerance," in *IEICE Technical Report*, vol. 107 of *COMP2007-34*, pp. 13–20, Sep. 2007.

List of Related Publications

Journal Papers

7. Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Loosely-stabilizing leader election in a population protocol model," *Theoretical Computer Science*, vol. 444, pp. 100–112, Jul. 2012.

8. Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa, "A concurrent partial snapshot algorithm for large-scale and dynamic distributed systems," *IEICE Transactions on Information and Systems*, vol. E97.D, no. 1, 2014. (to appear).

Conference Papers

9. Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Loosely-stabilizing leader election in population protocol model," in *Proceedings of the 16th international conference on Structural Information and Communication Complexity*, SIROCCO'09, pp. 295–308, Springer-Verlag, 2009.

10. Yuichi Sudo, Daisuke Baba, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "An agent exploration in unknown undirected graphs with whiteboards," in *Proceedings of the Third International Workshop on Reliability, Availability, and Security*, WRAS '10, pp. 8:1–8:6, ACM, 2010.

11. Yonghwan Kim, Tadashi Araragi, Junya Nakamura, and Toshimitsu Masuzawa, "Brief announcement: a concurrent partial snapshot algorithm for large-scale and dynamic distributed systems," in *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pp. 445–446, Springer-Verlag, 2011.

Technical Reports

12. Asaha Ishii, Yonghwan Kim, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa, "Evaluation of hadoop system consisting of virtual machines on multi-core cpus (in Japanese)," in *IPSJ SIG Technical Reports*, vol. 2012-HPC-136, pp. 1–7, Sep. 2012.

13. Kim Yonghwan, Araragi Tadashi, Nakamura Junya, and Masuzawa Toshimitsu, "A distributed and cooperative namenode cluster for a highly-available hadoop distributed file

system," in *Proceedings of the 13th High Performance Computing Symposium (HPCS)*, p. 93, Jan. 2013.

Abstract

A distributed system consists of multiple processes connected by a network and the processes can communicate with each other by sending messages. The processes execute a distributed algorithm to solve a problem to provide a function, e.g. routing, an overlay network, and a distributed file system. One of the important features provided by the distributed system is *fault tolerance*. A distributed system realizes the fault tolerance by utilizing redundancy, and a target of fault tolerance, e.g., data, service, and so on, is replicated to the processes. The processes keep their replicas consistent to tolerate failures of communication links or processes.

Byzantine failure is the most malicious failure, in which a faulty system behaves in an arbitrary way deviating from the original program. Such failures are caused by software bugs, hardware problems, or cracker attacks. In particular, cracker's attacks such as infections from viruses and intrusions are serious problems that severely damage systems connected in the Internet. Therefore, a practical fault tolerant method for Byzantine failure is strongly demanded.

State machine replication is one of the main approaches to build a server system that can tolerate such Byzantine failures. In the replication, the server role is replicated to multiple replicas and the replicas process requests submitted by clients to the server. The replication guarantees that the server system can continue to process requests, even if a fraction of the replicas is Byzantine faulty because of crackers' attacks or software errors. To eliminate their malicious behaviors, a client uses a majority rule; the client collects responses from the replicas of a request and accepts only a major value of them.

The non-faulty replicas must make the same responses so that the majority rule can work correctly, and this is realized if the replicas process all requests in the same order. Since network speed is not uniform and requests may be delivered to the replicas in different orders, the replicas execute a consensus protocol to agree on processing orders of the requests.

In this dissertation, we propose two methods for the Byzantine fault tolerant (BFT) state machine replication that focus on the consensus part of the replication and improve efficiency

and practicality.

First, we define a new type of consensus problem called request set consensus problem and propose a randomized BFT protocol that solves it. The protocol is designed for an asynchronous distributed system like the Internet, and can be used to realize state machine replication. The protocol has two features to solve the problem efficiently. First, although most existing multi-valued consensus protocols take a modular approach, in which replicas repeatedly solve a binary consensus problem to reach an agreement, our consensus protocol solves the request set consensus problem directly without such repetition. Thanks to the simple structure of the protocol, we can reduce the communication steps needed to reach an agreement. Second, we introduce an efficient coin tossing scheme that enables replicas to reach an agreement in a few rounds by exploiting the structure of the BFT replication. We prove that the protocol satisfies the correctness of the request set consensus problem. Performance evaluation is conducted from two viewpoints, i.e., analytically and experimentally. The analytical evaluation shows that our protocol can reach an agreement within two rounds, even if there are many replicas to tolerate Byzantine faults. In the experimental evaluation, our protocol achieves higher throughput and shorter latency than the existing ones, especially when the number of replicas is large.

Second, we propose a method that parallelizes consensuses to determine processing order of requests. Since the BFT replication is deployed on an asynchronous network such as the Internet and every consensus protocol for the replication is a randomized one, the duration of an execution of consensuses varies every time. If some consensus takes a long time, invocations of succeeding consensuses are delayed. This causes performance degradation to the server replication, and we solve the problem by parallelizing consensus executions. However, if replicas simply process the agreed requests in their terminated orders of the consensuses, it also makes their replicated server states inconsistency, since the network is asynchronous and the terminated orders may differ among the replicas. Therefore, we introduce an extra agreement step to resolve this discrepancy. Moreover, to decrease the cost of the parallelization, we introduce a randomization technique to a consensus protocol to generate initial proposals for newly-invoked consensuses. It reduces the size of the proposals, and, as a result, the duration taken for the consensus protocol becomes shorter. We prove that the parallelizing method satisfies correctness for the replication and evaluate its performance by comparing the parallelizing method with a sequential method currently in use. The evaluation results show that the parallelization has a strong advantage in spite of requiring additional consensuses, especially, when some replicas work slowly or some requests are delivered late.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A distributed system [1] consists of multiple processes connected by a network and the processes can communicate with each other by sending messages. The processes execute a distributed algorithm [2] to solve a problem to provide a function, e.g. routing, an overlay network, and a distributed file system. Especially, one of the important features provided by the distributed system is *fault tolerance*. A distributed system realizes the fault tolerance by utilizing redundancy, and a target of fault tolerance, e.g., data, service, and so on, is replicated to the processes. The processes keep their replicas consistent to tolerate failures of communication links or processes.

Byzantine failure is the most malicious failure, in which a faulty system behaves in an arbitrary way deviating from the original program. Such failures are caused by software bugs, hardware problems, or cracker attacks. In particular, cracker's attacks such as infections from viruses and intrusions are serious problems that severely damage systems connected in the Internet. Therefore, a practical fault tolerant method for Byzantine failure is strongly demanded.

One of the main approaches to tolerate Byzantine failure in asynchronous networks as the Internet is state machine replication [3]. The state machine replication is designed for a server-client system. In the replication, the server role is replicated to multiple replicas and the replicas process requests submitted by clients to the server. In the replication, as shown in Fig. 1.1, clients multicast requests to all the replicas. Here, the system's internal state is supposed to be determined by the initial states and the sequence of requests applied to the system. The replicas make an agreement on the processing order of the received requests, and process them sequentially in the agreed order. Even if the actual orders of the deliveries of the requests differ among the replicas, the replicas process the requests in the same order and keep their replicated server states identical. In the setting of the state machine replication for Byzantine failure, we assume that
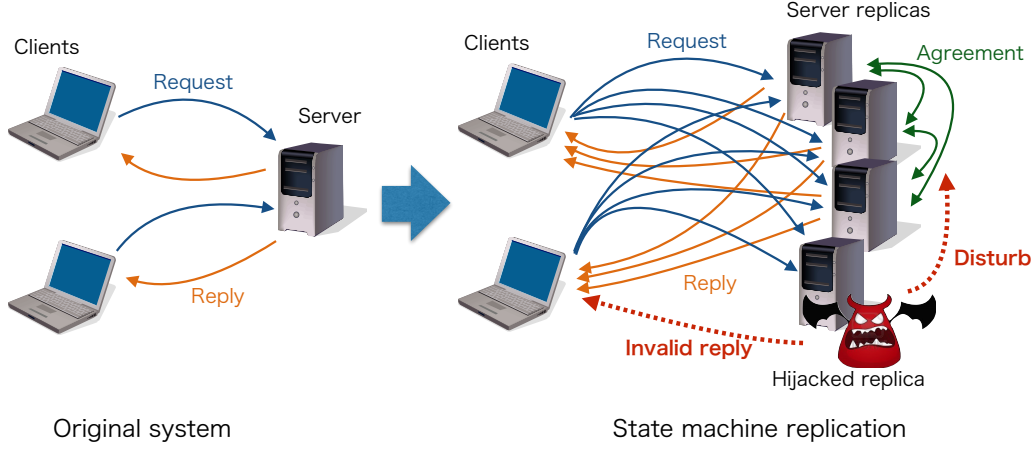
Figure 1.1: Structure of state machine replication

only a small fraction of replicas can be faulty and behave differently from the non-faulty replicas. With this assumption, if a majority of identical replies for the request is issued from non-faulty replicas, they can exclude the effect of the faulty replicas.

As stated above, in the state machine replication, *agreement* or *consensus* plays an important role, and many researches in this area were done [4, 5, 6, 7, 8, 9]. Fischer et al. proved some important result, that is, there exists no deterministic agreement protocol for an asynchronous distributed system, even if only one process may crash during an execution. Therefore, to solve the consensus and to realize the state machine replication, it is necessary to weaken the assumed model or the requirements of the consensus problem. There are two approaches to overcome this impossibility:

1. Ensure the termination deterministically under some assumptions about communication delay called *weak asynchrony assumptions* (Byzantine failure: [9, 10, 11, 12], crash failure: [8, 13, 14, 15]).

2. Ensure the termination with probability 1 by using randomized methods (Byzantine failure: [16, 17, 18, 19, 20, 21, 22], crash failure: [23]).

In the first approach, to solve the consensus problem deterministically, the asynchrony assumptions for the communication speed and the processing power of processes are weaken in the weak asynchrony model; there exist an upper bound (usually unknown) about the communication delay and a lower bound of the processing power of the processes. Although the impossibility of the asynchronous consensus problem is mainly brought by the difficulty to distinguish the crashed

process from the one just working slowly, this difficulty is not a problem in the model. A process in the model can detect another process's failure, since the process can estimate these bounds. If the process does not receive any response from the suspected process within the expected interval, the process treats the suspected process as a faulty one from now on. Consequently, a primary-backup approach is introduced to take advantage of this feature of the model. A special replica, called *primary* or *rotating coordinator*, controls the protocol's execution. When the primary receives a request from its client, the primary assigns the processing order of the request, and broadcasts the order and the request to the other replicas called *backups*. When the backups receive the request with the assigned order, they process it and reply their responses to the client. In such leader-based approach, replicas normally achieve an agreement very fast and provide a practical solution for the state machine replication. The backups also keep watching the primary's work, and if they suspect that the current primary is faulty, because the primary does not send such a assigned request within the predefined interval, they start a procedure to move the primary's role to another replica based on their agreement, which is why it is called a rotating coordinator.

A representative consensus protocol for this approach is PBFT protocol proposed by Castro and Liskov [9]. The Byzantine fault tolerant state machine replication has long been considered costly and non-practical, the efficient PBFT protocol dispelled such negative rumor and their result stimulated the development of efficient BFT protocols. PBFT guarantees the termination of agreements, even if there are continuous rotations of a primary coordinator, assuming that the message delay has an upper bound. However, this assumption can be broken if the attacker skillfully controls the flow of messages, and this can happen in an open network like the Internet. Even without such attacks, in a congested network, the primary coordinator is often changed and the efficiency is greatly reduced because each change of the primary is very costly.

In the second approach, protocols use some randomization mechanisms implicitly or explicitly. Most of protocols belonging to this approach randomized themself explicitly. An execution of the randomized consensus protocols is composed of *rounds*, and in a round, processes propose their own candidates and try to decide a common value for agreement. If they succeed to decide the common value, they will terminate. When they fail to agree with some value, they change own proposals randomly based on the messages received in this round (this random selection is usually called *coin tossing* or *coin flipping*), and proceed to the next round. This repetition continues until all non-faulty processes reach an agreement. For efficiency, in the context of the BFT replication, the proposals and the agreed value are not a single request (i.e., the request to be processed next), but a set of requests (i.e., the set of requests to be processed next). The

processing order among the requests in the set is determined by a predefined order (e.g., an order of the IDs of the clients that issued the requests).

Another type of the randomization is used by a protocol proposed in [24]. The protocol assumes randomized deliveries of messages sent in the protocol, and each process chooses a proposal of a next round based on messages received during the current round. If a sufficient number of non-faulty processes' proposals become identical by the implicit randomization, the processes agree with the proposal.

These randomization-based consensus protocols can be employed in more general environments than the first leader-based ones, since the protocols do not need any extra assumptions to solve the consensus problem correctly. Moreover, the randomization-based protocols' performance degradation during Byzantine faults occur is expected to be smaller than the leader-based ones, since the leader-based ones must take care of faulty primaries. Borran et al. analytically compared the two approaches on a partial synchronous model [25].

There also exists a hybrid solution [26], i.e., a protocol that uses both of the approaches. The processes normally agree with the processing order based on the first approach, however, if once a current leader is suspected to be faulty, replicas switch to a pessimistic mode and assign the requests based on the second approach.

1.1   Overview of This Dissertation

This dissertation focuses on the second approach, i.e., randomization-based Byzantine fault tolerant replication in which replicas execute a randomized Byzantine consensus protocol to agree with the processing order of submitted requests. We propose two methods for improving efficiency of the replication.

1.1.1   Request Set Consensus Protocol for BFT Replication

In Chapter 3, we propose a fast and resource-efficient agreement protocol on a request set, which is used to realize Byzantine fault tolerant state machine replication. Although most existing randomized protocols for Byzantine agreement exploit a modular approach, that is, a combination of agreement on a bit value and a reduction of request set values to the bit values, our protocol directly solves the multi-valued agreement problem for request sets. We introduce a novel coin tossing scheme to select a candidate of an agreed request set randomly. By specializing in the structure of the replication, the coin tossing scheme enables replicas to merge their candidates

quickly and realize fast agreement. In the performance evaluation, we analyze our protocol theoretically, and compare the protocol with the existing representative protocols in a practical environment.

1.1.2 Parallelizing Consensuses to Reduce Latency

In Chapter 3, we propose a new method that accelerates asynchronous Byzantine Fault Tolerant (BFT) protocols designed on the principle of state machine replication. State machine replication protocols ensure consistency among replicas by applying operations in the same order to all of them. A naive way to determine the application order of the operations is to repeatedly execute the BFT consensus to determine the next executed operation, but this may introduce inefficiency caused by waiting for the completion of the previous execution of the consensus protocol. To reduce this inefficiency, our method allows parallel execution of the consensuses while keeping consistency of the consensus results at the replicas. We also prove the correctness of our method and experimentally compare it with the existing method in terms of latency and throughput. The evaluation results show that our method makes a BFT protocol three or four times faster than the existing one when some machines or message transmissions are delayed.

1.2 Organization of This Dissertation

This dissertation is composed of five chapters. In Chapter 2, we define our distributed system model and state machine replication. We also describe how to realize the state machine replication by using a consensus protocol with examples. In Chapters 3 and 4, we propose two methods for the Byzantine fault tolerant replication. Our new efficient randomized consensus protocol RSC is proposed in Chapter 3. The protocol uses new randomization techniques to realize fast agreement and its correctness and performance are also shown there. Chapter 4 proposes another method that accelerates existing replication protocols, e.g., our consensus protocol RSC. The method executes a replication protocol concurrently to reduce response time of submitted requests, and also does another special type of consensus protocol to keep consistency while executing consensuses concurrently. Finally, we conclude this dissertation in Chapter 5.

Chapter 2

Preliminary

Here, we describe our system model, state machine replication, and how the replication is realized by a protocol that solves the request set consensus problem. Finally, we formally define the request set consensus problem.

2.1 System Model

A distributed system consists of *processes* and *communication links*. We assume the followings for our system model. The system is *asynchronous*; there is no bound on time to process data or communication delays. Every pair of processes is directly connected by a communication link, and processes only exchange information by message passing. The communication links are *reliable channels*; messages sent by non-faulty processes must eventually be delivered to the destination processes, and no message is lost in the communication links. From a received message, a process can identify the sender process, and even a malicious process cannot impersonate it. Each process has a local clock, but these clocks are not synchronized among processes; they may run at different rates and indicate different times.

Some processes may fail during a protocol execution. We adopt *Byzantine* failure (also called arbitrary failure) as a failure model. A Byzantine process can behave arbitrarily deviating from the protocol specification by stopping processes, omitting messages, and submitting invalid messages, etc. The processes behaving based on the protocol specification are called *non-faulty*, and the others (i.e. Byzantine processes) are called *faulty*.

2.2   State Machine Replication

State machine replication [3] is used in the server-client model, and a server is modeled as a state machine. A state machine consists of a set of states and a set of commands, and executes a command to change its state. The next state of a state machine is determined by an executed command and its current state. The server's role is replicated to n processes called *replica* that independently operate the role on distinct hosts and interact with client processes by request and response messages. A client submits a request to all replicas to request the servers to execute commands. Although, an asynchronous system allows request messages to arrive at different replicas in different orders, the replicas must process the requests in the same order to keep consistency among the replicas. More formally, a protocol that realizes the state machine replication must satisfy the following two requirements:

Safety All correct replicas process the requests submitted by clients in the same order.

Liveness A client eventually accepts the response to any request it submitted.

To realize identical processing order of requests, the replicas execute a consensus protocol. After a replica processes a request, it replies to the client with the execution result. The client accepts the result when it receives the same result from $f + 1$ replicas. Here, f is the upper bound of the number of faulty replicas. A client can confirm that at least one correct result was received from a correct replica when it collects $f + 1$ identical results. Since n must be greater than or equal to $3f + 1$ to realize Byzantine consensus by randomized protocols [27], we assume that $f \leq \lfloor (n-1)/3 \rfloor$. (In our model, we assume that clients never fail, and only replicas may fail, because a faulty client cannot affect an execution of state machine replication based on randomized Byzantine agreement. Therefore, the assumption never reduces the generality of our protocol. On the other hand, a faulty client can do of rotating coordinator-based replication, e.g. view-change of PBFT [9]).

Figure 2.1 shows an example of state machine replication. There are two clients and four replicas, and the clients broadcast requests r_1 and r_2. Since its network is asynchronous, the arrival orders of the requests are different among the replicas that execute a Byzantine consensus protocol to agree with the processing order of the requests. As a result, the replicas agree with processing order $r_1 \rightarrow r_2$, process the requests in the order, and send their responses to the clients.
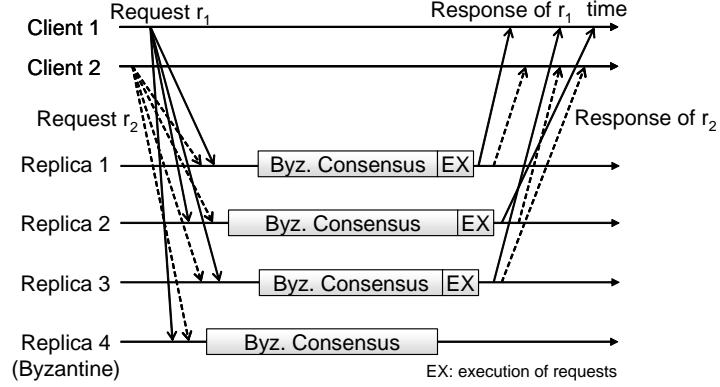
Figure 2.1: An example of state machine replication

2.3 Request Set Consensus Approach for State Machine Replication

Here, we explain how state machine replication is realized by a protocol that solves the request set consensus problem. Hereafter, we will formally define the requirements for the request set consensus problem in Sec. 3.2.

As previously explained in Sec. 2.2, when a client requests a server to execute some commands, it multicasts the requests to all the replicas. Although the requests are eventually received by all the replicas, they can be delivered in different orders among replicas. For example, in Fig. 2.1, Replica 1 receives request r_2 from client 2 first and then request r_1 from client 1, but Replica 2 may receive request r_1 from client 1 first and then r_2. If the replicas process the requests in the order they are received, the behaviors of the non-faulty replicas can be different.

To obtain a common order of processing requests among replicas, they must repeatedly execute a request set consensus protocol and arrange the requests in the agreed set as follows:

Step 1: A replica finds the set of requests it has received and has not yet processed, which is called agreement *candidate*, and executes a request set consensus protocol with the candidate as its initial value (or proposal) of the protocol. Let M be the agreed set of the execution, which is, of course, common to all non-faulty replicas.

Step 2: A replica processes the requests in M in a given deterministic order common to the replicas and returns the results to the corresponding clients.

Figure 2.2: Execution example of state machine replication by the request set consensus problem

Step 3: Return to Step 1.

This repetition is continued until the service is terminated.

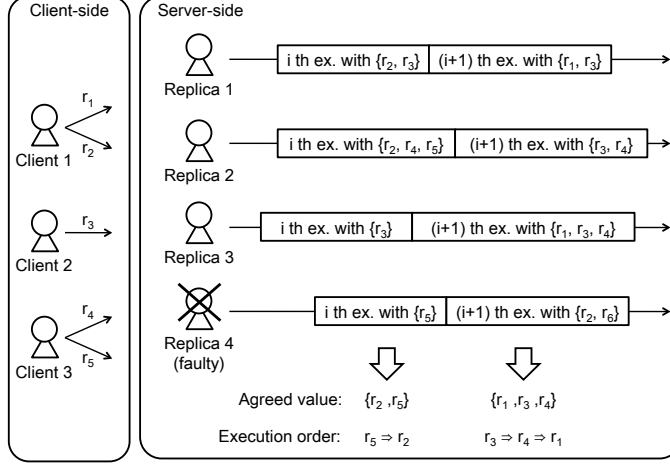Figure 2.2 illustrates the execution of state machine replication based on the request set consensus problem. Here, the request set consensus problem is solved repeatedly, and Fig. 2.2 shows the i-th and the $(i+1)$-th agreements. The distributed system in Fig. 2.2 has three clients and four replicas where Replica 4 is faulty. The clients issued five requests: $r_1, r_2, \cdots, r_5$. Before starting the i-th agreement, every replica places the set of unprocessed requests it has received to its initial value for the i-th execution of an agreement protocol. Since the order and timing of arriving requests can be different among replicas, the initial values might also be different; e.g., Replica 1 sets $\{r_2, r_3\}$ and Replica 3 sets $\{r_3\}$ to their initial values of the i-th agreement. Our agreement protocol for the request set consensus problem guarantees that every non-faulty replica obtains a common set of requests, and the agreed set is subset of the union of the initial values of the non-faulty replicas. In Fig. 2.2, the i-th agreement returns agreed value $\{r_2, r_5\}$. Then every non-faulty replica processes the elements of the agreed set in some deterministic order. If r_5 precedes r_2 in the order, the non-faulty replicas execute r_5 first and then r_2. When a replica finishes execution of the i-th agreement, it starts the $(i+1)$-th agreement. Each replica's initial value of the $(i+1)$-th agreement protocol is, as above, the sets of requests that have arrived and have not been processed yet; they are not included in the agreed sets of any preceding j-th agreement ($j < i+1$). For example, Replica 2 sets $\{r_2, r_4, r_5\}$ to the i-th initial value, and r_2 and r_5 are included in the i-th agreed set, and then r_4 is again included in the initial value of the

$(i + 1)$-th agreement. On the other hand, r_3 is received by Replica 2 during the i-th agreement protocol execution and is also included in the $(i + 1)$-th initial value. In Fig. 2.2, Replica 4, a faulty replica, is disturbing the agreement by proposing an invalid initial value for the $(i + 1)$-th agreement. Replica 4 includes r_2 in the initial value, which was already included in the i-th agreed set. The replica also includes forged request r_6 in the initial value. Even if malicious replicas act in such a way, an agreement protocol works correctly, because it does not include requests of the past agreement again or forged ones in the agreed value. The $(i + 1)$-th agreed value is $\{r_1, r_3, r_4\}$, and every replica processes the requests in some deterministic order. Replicas repeat such agreements and processing until the service is terminated.

Note that definitions of request set consensus differ between Chapters 3 and 4. Although we will focus on the state machine replication and the request set consensus approach throughout the thesis, each chapter discusses different problems under different definitions.

Chapter 3

Request Set Consensus Protocol for BFT Replication

3.1 Introduction

In this chapter, we present a new randomized multi-valued Byzantine consensus protocol for Byzantine fault tolerant state machine replication. Our protocol is based on randomization, and thus, can be employed in more general settings than leader-based consensus protocols such as Castro and Liskov's PBFT. We employ Bracha's agreement protocol [24] as a framework of our protocol, and develop a very original way of coin tossing, which is crucial to realize fast and efficient agreement. Our protocol called RSC is as efficient as existing fast randomized protocols ABC [16] and RITAS [17, 18, 19], and much less resource-consuming than these two protocols.

3.1.1 Related work

ABC [16] proposed by Cachin et al., and RITAS [17, 19, 18] proposed by Correia et al., take the second approach. Their protocols realize atomic broadcast [28]. Atomic broadcast guarantees that every non-faulty replica in a distributed system receives the same broadcast messages in the same order, and is easily transformed to the state machine replication. ABC and RITAS realize atomic broadcast by multi-valued Byzantine agreements, where replicas agree on the same value in a given set of multiple values $\{v_1, \ldots, v_n\}$. However, these multi-valued agreements are implemented by reducing them to binary Byzantine agreements, where replicas agree on the same value in the set $\{0, 1\}$. ABC uses Cachin et al's binary Byzantine agreement protocol ABBA [29],

and RITAS uses Bracha's binary Byzantine agreement protocol [24]. ABBA employs shared coin tossing [6] that minimizes the number of rounds to get the same coin value by cryptographic communication among the replicas, and Bracha's protocol simply does local coin tossing, where the replicas independently repeat the coin tossing until they happen to have the same coin value. RITAS generally needs more rounds for an agreement, while ABC is time-consuming during cryptographic communication. In addition, Moniz et al. evaluated ABBA and Bracha's consensus protocol in a real environment, and highlighted these protocols' characteristics [30].

3.1.2   Contributions

Multi-valued Byzantine agreement with special valued coin tossing: A naive multi-valued Byzantine agreement based on local coin tosses [5] is extremely inefficient, especially when the number of replicas increases. Therefore, ABC and RITAS proposed ways of reducing multi-valued agreement to binary agreement. In contrast, we employ multi-valued agreement with a new way of coin tossing. First, by exploiting the structure of replication, we introduce a special value of the coin that changes dynamically and independently on each replica but finally coincides among the replicas. We analyze how this special coin value overcomes the inefficiency of multi-valued agreement in the performance evaluation of Sec. 3.5.2. Second, our multi-valued agreement also introduces a procedure for each replica to merge the request sets proposed by other replicas during the rounds of the agreement process. This merging allows the proposed value to converge and contributes to the fast termination of the agreement. We also analyze the effect in Sec. 3.5.2.

Less resource-consumption: Owing to the simple structure of the multi-valued agreement, our protocol is much less resource-consuming than ABC and RITAS. In Sec. 3.5.3, we compare the loads of the request processing (i.e., request frequency) that reach the resource bound among ABC, RITAS, and our protocol by changing the number of replicas. We prove that our protocol is much more resource-efficient especially in a large number of replicas, while preserving fast responses. We also experimentally analyze and compare our protocol's characteristics with RITAS.

3.2   Request Set Consensus

We formally define the requirements for the request set consensus problem for replication. Note that the definition is specific to the replication rather than a general use of consensus. In particular, the agreed value is a set of requests, rather than a single numerical value, as stated in the introduction.

Let the initial value of the i-th execution of the request set consensus protocol for replica p be I_p^i, the i-th execution of the request set consensus protocol by p be $RSC(p, i, I_p^i)$ and its agreed value be V_p^i. Note that I_p^i and V_p^i are sets of requests. Let R be the set of all the requests submitted by clients and $M_p^i (\subset R)$ be the set of requests that process p has received before starting the i-th request set consensus execution. We assume that every replica eventually receives all requests in R and $I_p^i = M_p^i - \left(V_p^1 \cup V_p^2 \cup \cdots \cup V_p^{i-1} \right)$. If there is no need to distinguish replicas, we simply write V_p^i as V^i and do similarly for others.

The followings are the requirements of the request set consensus problem for replication: Agreement, Termination, Integrity, and Validity.

Agreement: Let p and q be non-faulty replicas. For any i, if $RSC(p, i, I_p^i)$ and $RSC(q, i, I_q^i)$ terminate, then $V_p^i = V_q^i$.

Termination: For any i and any non-faulty replica p, $RSC(p, i, I_p^i)$ terminates with probability 1.

Integrity: For any i and any non-faulty replica p, $V_p^i \subseteq R$.

Validity: For any $r \in R$ and any non-faulty replica p, there exists i such that $r \in V_p^i$.

Agreement ensures that all non-faulty replicas agree with the same value at each execution, and termination ensures that every execution of the request set consensus at every non-faulty replica terminates with probability 1. Integrity guarantees that no agreed value contains any forged requests, and validity guarantees that any request is eventually processed.

The Integrity and Validity above and used in the thesis are slightly different from the requirements presented below of a usual request set consensus problem.

Usual Integrity: For any i and any non-faulty replica p, $V_p^i \subseteq \bigcup_{\text{non-faulty } q} I_q^i$.

Usual Validity: For any i and any non-faulty replica p, $\bigcap_{\text{non-faulty } q} I_q^i \subseteq V_p^i$.

Our requirements are arranged for the repeated use of the request set consensus problem in replication. Our integrity requirement is induced from the usual one, and our validity requirement is also induced by the usual one on the assumption that every replica eventually receives all requests in R, as mentioned above. Note that our RSC protocol presented in the next section actually does not satisfy the usual integrity requirement.

3.3   The RSC Protocol

We propose a new efficient BFT protocol called RSC that solves the request set consensus problem, based on Bracha's binary Byzantine agreement protocol [24]. Let n be the number of replicas and f be the maximum possible number of faulty replicas. The protocol works correctly, that is, it satisfies the requirements: Agreement, Termination, Integrity, and Validity, when $f \leq \lfloor (n-1)/3 \rfloor$, i.e., $n \geq 3fgT + 1$. Since asynchronous Byzantine agreement cannot be solved deterministically even if $f = 1$ [7], our protocol uses randomized coin tossing, like Bracha's protocol.

We borrow two techniques, which we explain in Sec. 3.3.1 and Sec. 3.3.2, from Bracha's original protocol: reliable broadcast and internal message validity check. We introduce another message validity check for RSC in Sec. 3.3.2. The details of our protocol are shown in Sec. 3.3.3. In Sec. 3.3.4, we describe our characteristic coin tossing scheme that plays a key role in RSC.

3.3.1   Reliable Broadcast

Reliable Broadcast [31, 24, 16] is a broadcast primitive that guarantees the followings: (i) a message broadcast by a non-faulty replica is eventually delivered to all non-faulty replicas, and (ii) if a message is delivered to a non-faulty replica, then the same message is eventually delivered to all other non-faulty replicas. Therefore, every replica accepts at most one identical message for a given ID, and a faulty replica cannot send distinct messages with identical IDs to different replicas in a broadcast. We denote the action of sending message m by reliable broadcast by R-Broadcast(m).

3.3.2   Message Validity Check

The RSC protocol uses two validity check methods: *internal* and *external*. The internal validity check is applied to protocol messages exchanged among replicas at Steps 2 and 3, and the external validity check is done to *INITIAL* messages issued at Step 0 and received at Step 3 in our protocol as described later. The internal validity check is the same as that used in Bracha's agreement protocol [24], which prevents a faulty replica from disturbing protocol executions. The external validity check, which is original with the RSC protocol, avoids false requests forged by faulty replicas. In the following, we explain these message validity checking methods in more detail.

Protocol 3.1 Pseudo code of Protocol $\mathsf{RSC}(p, i, I_p^i)$ for replica p (Part 1)

Protocol $\mathsf{RSC}(p, i, I_p^i)$

Input:

1: p $(1 \leq p \leq n)$: process number;

2: $i \in \mathbb{N}$: execution ID of RSC protocol;

3: $I_p^i \subseteq R$: initial candidate-value;

Output:

4: $V_p^i \subseteq R$: agreed value;

Variables:

5: $rn \in \mathbb{N}$: round number, initially zero;

6: $Cand \subseteq R$: candidate of agreement, initially I_p^i;

7: $RI \subseteq R$: union of accepted initial values, initially I_p^i;

8: $isMajor \in \{\mathbf{true}, \mathbf{false}\}$: flag indicating whether $Cand$ is a majority, initially **false**;

Internal Validity Check

In internal validity check, a non-faulty replica accepts a message after confirming that it can be sent. A replica verifies that a message being checked can be sent by a sender replica after seeing the messages received from other replicas in the last step. $n - f$ messages must be sent in the last step to induce the sender replica to send the message in the current step. Messages validated by this checking are called internally-valid. By this checking, we can prevent faulty replicas from sending illegal messages. A detailed and formal explanation of the internal validity check can be found in [24].

External Validity Check

In external validity check, a non-faulty replica accepts a message including requests when it also has directly received all the included requests from clients. With this check, we exclude forged requests in the communication of the RSC protocol. Messages validated by this check are called externally-valid.

3.3.3 Protocol

Protocols 3.1 and 3.2 show the pseudo code of the RSC protocol in which we denote a set $\{1, 2, \cdots, k\}$ by $[k]$. The RSC protocol has four steps. Step 0 is executed once, and Steps 1, 2, and 3 are executed repeatedly. We call a sequence of executions from Steps 1 to 3 a *round*, and RSC repeats rounds until it reaches an agreement. The RSC input is a triplet of the identifier of replica p, execution ID i, and a value of initial candidate I_p^i. The execution ID is a natural sequence number that starts from 1 and increases by one after each agreement, and RSC outputs

---

Protocol 3.2 Pseudo code of Protocol $\mathsf{RSC}(p, i, I_p^i)$ for replica p (Part 2)

Code:

9: **[Step 0]**

10: R-Broadcast($\langle INITIAL, i, I_p^i \rangle$);

11: **[Step 1]**

12: R-Broadcast($\langle MSG1, i, rn, Cand \rangle$);

13: **Wait for** accepting $n - f$ internally-valid messages $\langle MSG1, i, rn, C_1 \rangle, \cdots, \langle MSG1, i, rn, C_{n-f} \rangle$;

14: **for all** request r contained in at least $f + 1$ candidates of $C_1, \cdots, C_{n-f}$ **do**

15:     $Cand := Cand \cup \{r\}$;

16:     $RI := RI \cup \{r\}$;

17: **end for**

18: **[Step 2]**

19: R-Broadcast($\langle MSG2, i, rn, Cand \rangle$);

20: **Wait for** accepting $n - f$ internally-valid messages $\langle MSG2, i, rn, C_1 \rangle, \cdots, \langle MSG2, i, rn, C_{n-f} \rangle$;

21: **for all** request r contained in at least $f + 1$ candidates of $C_1, \cdots, C_{n-f}$ **do**

22:     $RI := RI \cup \{r\}$;

23: **end for**

24: **if** at least $\lceil (n + 1)/2 \rceil$ of $C_1, C_2, \cdots, C_{n-f}$ are the same (denoted by C) **then**

25:     $Cand := C$;

26:     $isMajor := $ **true**;

27: **end if**

28: **[Step 3]**

29: R-Broadcast($\langle MSG3, i, rn, Cand, isMajor \rangle$);

30: **Wait for** accepting $n - f$ internally-valid messages $\langle MSG3, i, rn, C_1, b_1 \rangle, \cdots, \langle MSG3, i, rn, C_{n-f}, b_{n-f} \rangle$;

31: **for all** request r contained in at least $f + 1$ candidates of $C_1, \cdots, C_{n-f}$ **do**

32:     $RI := RI \cup \{r\}$;

33: **end for**

34: **if** there are at least $2f + 1$ *MSG3* messages $\langle MSG3, i, rn, V, \textbf{true} \rangle$ for some V **then**

35:     R-Broadcast($\langle MSG1, i, rn + 1, V \rangle$);

36:     R-Broadcast($\langle MSG2, i, rn + 1, V \rangle$);

37:     R-Broadcast($\langle MSG3, i, rn + 1, V, \textbf{true} \rangle$);

38:     **return** V;

39: **else if** there are at least $f + 1$ *MSG3* messages $\langle MSG3, i, rn, V, \textbf{true} \rangle$ for some V **then**

40:     $Cand := V$;

41: **else**

42:     **if** a message $\langle INITIAL, i, I \rangle$ has arrived but has not been accepted as an **externally-valid** message **then**

43:         Accept its message as an **externally-valid** message;

44:         $RI := RI \cup I$;

45:     **end if**

46:     $D := (\{C_1, C_2, \cdots, C_{n-f}\} \cap 2^{RI}) \cup \{RI\}$; {Here, D is a multiset, i.e., it can contain two or more identical values.}

47:     Choose candidate C from D randomly;

48:     $Cand := C$;

49: **end if**

50: $isMajor := $ **false**;

51: $rn := rn + 1$;

52: **goto Step 1**;

---

an agreed set of requests denoted by V_p^i.

The control variables for the protocol are rn, $Cand$, RI, and $isMajor$. A current round number is represented by rn. $Cand$ represents a tentative candidate for an agreed value. The value of $Cand$, which is initially the value of the initial candidate, is updated at each step based on the values collected at that step. A non-faulty replica broadcasts these values and collects them from other replicas. Roughly speaking, if most of the collected values are the same, the non-faulty replicas agree with the value. RI represents the set of requests received indirectly from a replica as its initial candidate and directly from a client after the initiation of the RSC protocol. The RI value eventually becomes common among the non-faulty replicas, although their initial values are different among replicas. In fact, the RI value is initially I_p^i and finally becomes a common value $\bigcup_q I_q^i$, where q ranges over the replicas that have issued an *INITIAL* message, which might include faulty ones. This value is used for two purposes. The first is as a special coin value used by coin tossing, and the second is to exclude forged requests in the coin toss phase (line 46). A Boolean variable $isMajor$ indicates whether a majority exists in the received candidates at Step 2. An $isMajor$ value is broadcast at Step 3 with $Cand$, and an update process of $Cand$ at Step 3 branches based on how many received messages include $isMajor$ as **true**.

During the RSC execution, four kinds of messages are exchanged. An *INITIAL* message announces its own I_p^i to the other replicas. *MSG1*, *MSG2*, and *MSG3* are used to inform other replicas about $Cand$ at Steps 1, 2, and 3, respectively. In addition, a *MSG3* message contains a flag $isMajor$. All of these messages are sent by reliable broadcast, which is explained in Sec. 3.3.1. Thus, even a faulty replica cannot send different values to different replicas in the broadcast.

Step 0: Reliably broadcast an *INITIAL* message to announce its own initial candidate I_p^i to other replicas. This message creates the special coin value RI that eventually becomes common among the replicas (line 44).

Step 1: Reliably broadcast a *MSG1* message to announce its own $Cand$ to the others and wait until $n-f$ *MSG1* messages have been received from others. During receiving, invalid messages are ignored by internal validity check, which will be explained later. With these received messages, its own $Cand$ and RI are extended by adding the newly known unforged requests that are commonly included by at least $f+1$ $Cand$ values in the messages.

Step 2: Reliably broadcast a *MSG2* message containing its own $Cand$ to the others and wait until $n-f$ internally-valid *MSG2* messages have been received from others. Add the requests contained in at least $f+1$ candidates in the messages to RI. If $\lceil (n+1)/2 \rceil$ or more $Cand$ values have an identical value, replace the value of its own $Cand$ with the common one, and set its own $isMajor$ to **true**.

Step 3: Reliably broadcast a *MSG3* message that has agreement candidate *Cand* and the flag of majority *isMajor*, whose value was set in Step 2. Wait until $n - f$ internally-valid *MSG3* messages have been received from others. Add the requests contained in at least $f + 1$ candidates in the messages to *RI*. There are three cases to update its own internal states based on the messages. (A) If there are $2f + 1$ or more messages whose *isMajor* is **true**, then *decide* the agreed value to be the *Cand* of these messages (lines 35–38). It is proven that for any two messages whose *isMajor* values are **true**, their *Cand* values are the same, even if they are sent by faulty replicas, owing to the reliable broadcast and the internal validity check. Therefore, this decision is well defined. Send messages for the next round once to other replicas that are proceeding to the next round, and then terminate. (B) If there are less than $2f + 1$ but $f + 1$ or more messages whose *isMajor* values are **true**, then replace its own *Cand* with the *Cand* of these messages (line 40). By doing this, the future decision value will be consistent with the value already decided by other replicas. (C) In the remaining case, a non-faulty replica tosses a coin (lines 42–48). The domain of the coin values consists of at most $n - f$ different *Cand* values received at Step 3 and *RI*. *RI* is extended by adding the union of the initial candidates included in the *INITIAL* messages received so far. Here, the external validity of these received *INITIAL* messages is checked, which is explained in Sec. 3.3.2, to exclude forged requests. Indirectly, *Cand* values are also checked for any forged requests by seeing that they are included in *RI*. Update its *Cand* to the coin toss result. Since the values of the non-faulty replica's *RI* eventually coincide with each other, termination is guaranteed.

3.3.4   Coin Tossing

When a non-faulty replica is not confident that a major value exists among the replica candidates, it tosses a coin. Our coin tossing is local; every replica tosses independently. The domain of our coin values is $D = \{D_1, \cdots, D_m, RI\}$, where $D_1, \cdots, D_m$ are *Cand* values that are received at Step 3 and are subsets of a local variable *RI*. We call *RI* "Special Coin Value" and *RI* plays an important role to realize an efficient agreement and to ensure the correctness of RSC protocol. A non-faulty replica randomly chooses a value from this domain, and proposes the value at the next round. Here, the domain D is a multiset, i.e., it can contain two or more identical values. Thus, the more D contains identical values, the higher the probability of choosing the value is.

We can ensure that the *RI* values of all the non-faulty replicas must eventually be stable and coincide as some value, and this is why *RI* is called as "special coin value". The reason is as follows. The value of *RI* is updated mainly when a replica receives an *INITIAL* message and

judges the message to be externally-valid. If a non-faulty replica receives an *INITIAL* message, the other non-faulty replicas also do so even if its sender is Byzantine, since all communications among replicas are done by the reliable broadcast. External validity of the message is judged based on requests received directly from clients, and these requests are eventually delivered to all replicas. Thus, if a non-faulty replica judges an *INITIAL* message to be externally-valid, the other non-faulty replicas also do so. In addition, a replica reliably broadcast an *INITIAL* message only once. Therefore, all non-faulty replicas eventually judge the same *INITIAL* messages to be externally-valid, namely, the RI values of the replicas will coincide as the union U of the request sets I_p^i sent as *INITIAL* messages. We call such a global configuration where the RI values of all the non-faulty replicas are U *SCV-full configuration*. Here SCV means the special coin value RI, and we call U a *common-value*, and the common-value ensures termination of our protocol.

3.4 Correctness

In this section, we prove that the RSC protocol satisfies the requirements for the request set consensus problem shown in Sec. 3.2: Agreement, Termination, Integrity, and Validity.

First, we prove a lemma for Theorems 1 and 2.

Lemma 1. *If all non-faulty replicas start at round r with the same value V of Cand, then they decide V in the round.*

Proof. Every replica reliably broadcasts its own $Cand$ at Step 1 of round r. At most f faulty replicas can broadcast a value other than V. Therefore, at the update of Step 1, the $Cand$ value of the non-faulty replicas remains unchanged. With this situation, at Step 2, every non-faulty replica accepts only *MSG2* messages, whose candidates are V, as internally-valid messages. Then every non-faulty replica collects $n - f$ ($\geq \lceil (n+1)/2 \rceil$) *MSG2* messages with candidate V and sets the *isMajor* flag to **true**. For the same reason as above, at Step 3, every non-faulty replica accepts only *MSG3* with a value of $Cand$ V and a **true** flag as an internally-valid message. Then every replica collects $n - f$ ($\geq 2f + 1$) such messages and decides V. $\qquad\square$

Theorem 1 (Agreement). *Let p, q be non-faulty replicas. For any i, if $RSC(p, i, I_p^i)$ and $RSC(q, i, I_q^i)$ terminate, then $V_p^i = V_q^i$.*

Proof. The main idea of the proof is identical to that for Bracha's binary Byzantine agreement protocol [24].

First, we consider the case where p and q decide at identical round r. Therefore, p and q have to accept $2f + 1$ internally-valid messages of $\langle MSG3, i, r, V_p^i, \textbf{true} \rangle$ and $\langle MSG3, i, r, V_q^i, \textbf{true} \rangle$ at

Step 3 of round r, respectively. This means that the two non-faulty replicas have to accept at least $\lceil (n+1)/2 \rceil$ $\langle MSG2, i, r, V_p^i \rangle$ messages and at least $\lceil (n+1)/2 \rceil$ $\langle MSG2, i, r, V_q^i \rangle$ messages respectively at Step 2. Since a replica cannot send two or more $MSG2$ messages in the same step even if the replica is faulty, $V_p^i = V_q^i$. Next, assume that p decides at round r, and no non-faulty replica decided in the preceding rounds. Therefore, all non-faulty replicas that do not decide at round r, one of which is q, commonly set V_p^i to their $Cand$ values at Step 3 of round r (line 40), and the replicas decide V_p^i at the next round $r + 1$ by Lemma 1.   □

Theorem 2 (Termination)**.** *For any i and any non-faulty replica p, $RSC(p, i, I_p^i)$ terminates with probability 1.*

Proof. From Lemma 1 and the fact that SCV-full configuration is eventually reached as stated in Sec. 3.3.4, it is sufficient to show that the $Cand$ values of all non-faulty replicas have the same value with some probability at the beginning of every round after a finite time when a SCV-full configuration is reached. There are three cases to update the $Cand$ values at Step 3. (a) If a non-faulty replica collects $2f + 1$ $MSG3$ messages with the same candidate and $isMajor$ is **true** (lines 35–38), all non-faulty replicas have the same candidate at the beginning of the next round, as discussed in the proof of Theorem 1. Note that the replicas collecting $2f + 1$ such $MSG3$ messages (i.e., they decide the agreed value) behave as non-faulty replicas with the candidate in the next round. (b) If there is no such non-faulty replica but only those collecting $f + 1$ or more such $MSG3$ messages (line 40), then those replicas set the candidate of the messages to their $Cand$ values. On the other hand, the replicas that do not collect more than $f + 1$ such $MSG3$ messages must accept at least one such $MSG3$ message by the validity check and toss a coin whose domain includes the candidate-value of the $MSG3$ messages. Therefore, the non-faulty replicas share the candidate-value with some probability at the beginning of the next round. (c) Lastly, if no non-faulty replica collects $f + 1$ or more such $MSG3$ messages (lines 42–48), every non-faulty replica tosses a coin whose domain includes the common-value, RI. Thus, they have a common candidate with some probability at the beginning of the next round.   □

Theorem 3 (Integrity)**.** *For any i and any non-faulty replica p, $V_p^i \subseteq R$.*

Proof. We show that no non-faulty replica contains forged requests in its $Cand$ value during the execution of the RSC protocol. Since the RSC protocol's initial candidate is a set of requests received directly from clients, there is no chance that a forged request is included. At Step 1, the $Cand$ value is modified by adding the requests that are commonly included in at least $f + 1$ internally-valid $MSG1$ messages (line 15), one of which is broadcast by a non-faulty replica.

Therefore, if the candidate-values of any non-faulty replicas at the beginning of the round do not include a forged request, the modified value does not either. At Step 2, the *Cand* value can be completely changed to a common candidate-value of at least $\lceil (n+1)/2 \rceil$ $(\geq f+1)$ *MSG2* messages (line 25). Therefore, similar to Step 1, no forged request is included in it. At Step 3, if a replica receives $f+1$ or more *MSG3* of an identical candidate and *isMajor* **true**, the replica sets its own *Cand* to the value (lines 35–37 and 40). Since at least one message is from a non-faulty replica, no forged request is included in the *Cand*. In the case of coin tossing, the coin values are subsets of RI (line 46). On the other hand, any element of RI is one included in a candidate of a non-faulty replica (lines 16, 22, and 32) or an externally-valid message (line 42), and then it is not a forged request. When a replica decides a value at Step 3, the value is a common candidate-value of at least $2f+1$ $(\geq f+1)$ *MSG2* messages. By the above discussion on the *Cand* value, no non-faulty replica contains any forged requests in its *Cand*. □

Theorem 4 (Validity). *For any $r \in R$ and any non-faulty replica p, there exists i such that $r \in V_p^i$.*

Proof. Let r be a request from a client and assume that it is never included in any agreed value. It is obvious that such a request will eventually be included in the initial values of the RSC protocol for all non-faulty replicas. That is, there is i such that for any non-faulty replica p, $r \in I_p^i$. If, at the beginning of Step 1, all non-faulty replicas include r in their candidate-values, then, after the modification of the candidate-value at Step 1, r is included in the candidate-value. Because among $n-f$ accepted messages, at least $f+1$ *MSG1* messages of them include r in their candidates. Similarly, at the modification of *Cand* at Step 2 and Cases A (lines 35–38) and B (line 40) of Step 3, the modified value is a candidate value of a non-faulty replica at the previous step and includes r. At Case C of Step 3 (lines 42–48), i.e., at the coin tossing, its domain is $\{D_1, \cdots, D_m, RI\}$, and all the values include r. Therefore, in every case, the modified candidate-value include r. With this observation that the *Cand* value always includes r through the execution, we conclude that V_p^i includes r, which contradicts the assumption. □

3.5 Performance Evaluation

3.5.1 Overview

We evaluate our RSC protocol from two points of view. First, we analyze the original features of RSC by simulation experiments. We measure the effect of the special coin value RI in SCV-full configurations (defined in Sec. 3.3.4) in Sec. 3.5.2 and Sec. 3.5.2 and how fast a system moves

from a non-SCV-full configuration to a SCV-full one in Sec. 3.5.2. We use the number of rounds needed to reach an agreement as an efficiency measure, simulate executions of the RSC protocol by replicas on a single machine, and evaluate the performance in relation to SCV-full configurations.

Next, we compare the latency and throughput of RSC with RITAS [19] and ABC [16] in Sec. 3.5.3. We implement request set consensus protocols by using RITAS and ABC, which are atomic broadcast protocols, in a straightforward way described in Sec. 3.5.4, and run these three protocols on multiple machines in practical settings for comparison.

In these various experiments, the efficiencies are evaluated under fault-free executions. That is, there is no Byzantine failure among the replicas. The passive reason is that there are an enormous number of ways of attacking and delaying the agreement, and it is hard to give a standard measure for the failure. The active reason is that the overhead of the agreement in the fault-free execution is more important for the replication service. Because, Byzantine faults, such as a cracker's intrusion, infection of virus, and out of order of systems, happen rarely, while the cost of operations providing against these faults is always charged, even for the case without faults.

3.5.2   Performance Evaluation on RSC Protocol Features

Evaluation in SCV-full Configuration

Evaluation environment and settings: The initial values of the RSC protocol for individual replicas, i.e., sets of requests, are set to random values so that achieving an agreement becomes hard. These values are different to each other among the replicas, and no request is included commonly in any two initial values. Note that the hardness in agreement is irrelevant to the amount of the number of requests. We implement all replicas as individual processes in a single host, and the order of receiving control messages for agreement is set to be uniformly random. We evaluate the number of rounds for termination for the values of α, 0.1, 0.5, and 0.9, varying the number of replicas n from 4 to 22. Here, α is the probability of choosing the special coin value RI in the coin tossing of the RSC protocol (line 47 of Protocol 3.2). For each case, we executed the evaluation 1,000 times, and we plot the average values in the graph of Fig. 3.1.

Result and discussion: Fig. 3.1 shows the increasing shape in the number of replicas. When α is large, the number of rounds is around 2 for any case, which is as ideally small as we expected. This is achieved by choosing a common-value with high probability at every non-faulty replica in the coin tossing of the first round. A remarkable point of this graph is that the number of rounds with small α is still reasonably small, while a naive estimation shows it needs $1/\alpha^{n-f}$
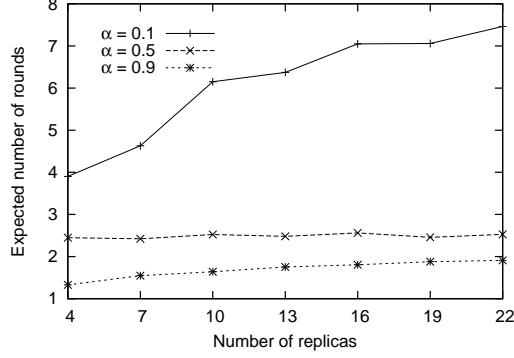
Figure 3.1: Number of rounds to reach an agreement in SCV-full configurations

rounds. With this observation, it is found that in the fault-free executions, large α gives much better performance. On the other hand, in the case of faulty replica's attacking, large α can be exploited by an attacker to delay the termination. Therefore, when we suspect such an attack, we should dynamically decrease the value of α. Thus, we are still interested in small α. In the following, we analyze the reason of this remarkable point.

Analysis of the Performance in SCV-full Configuration with Small α

First, we intuitively explain the reason of fast agreement for small α. We call a replica whose *Cand* value is the common-value defined in Sec. 3.3.4 the *candidate-full*. When α is small, the chance is small that replicas reach an agreement by coin tossing. However, if some replicas are candidate-full, others' candidate-values are more likely to be modified to the common-value in their updates at Steps 1, 2, and 3 of the RSC protocol. As a result, more replicas can have the same value in the possible coin values and the probability of getting agreement becomes high. Below, we present a simulation experiment under some model, which confirms this intuition.

Analysis model: Assume that there exists k ($\geq f + 1$) candidate-full replicas at the beginning of a round. In such a configuration, a replica can change its *Cand* in this round in the following three cases:

Case 1: At Step 1, when there is a request that is contained in at least $f + 1$ received candidates, it is added to its own *Cand*.

Case 2: At Step 2, when it receives the same value from a majority of replicas, it replaces its own candidate with the value.

Case 3: At Step 3,

1. when it receives k ($f + 1 \leq k < 2f + 1$) candidates with $isMajor = $ **true**, it replaces its own candidate with the value.

2. when it does not receive more than f candidates with $isMajor = $ **true**, it updates its own candidate randomly by a coin toss.

Following these cases, we evaluate how the expected number of candidate-full replicas changes. As a commonly used probability scheme through the evaluation, we introduce the following probability. Assume that there are y pieces of marked lots among x pieces of lots in a box. We denote $P(x, y, x', y')$ as the probability of obtaining at least y' pieces of marked lots by randomly drawing x' pieces of lots from the box. By a simple calculation, the following holds:

$$P(x, y, x', y') = \left(\sum_{i=\max\{y', x'-x-y\}}^{\min\{y, x'\}} {}_yC_i \cdot {}_{x-y}C_{x'-i} \right) / {}_xC_{x'}.$$

Using this probability, we show the evaluation.

At first, for Case 1, the k candidate-full replicas remain candidate-full. A non-candidate-full replica becomes candidate-full when it receives at least $f + 1$ common-values, and $P(n, k, n - f, f + 1)$ is the probability of this happening. There are a few other cases where non-candidate-full replicas become candidate-full, but we ignore them for simplicity. Therefore, the expected number of candidate-full replicas after Step 1 is at least

$$k_1 = k + (n - k) \cdot P(n, k, n - f, f + 1).$$

In Case 2, by the assumption that $k \geq f + 1$, a non-candidate-full replica becomes candidate-full only when it receives at least $(n + 1)/2$ common-values. Therefore, the expected number after Step 2 is at least

$$k_2 = k_1 + (n - k_1) \cdot P(n, k_1, n - f, (n + 1)/2).$$

Lastly, in Case 3, the probability that Case 3-1 happens is

$$p_1 = P(n, k_2, n - f, f + 1) - P(n, k_2, n - f, 2f + 1),$$

while that of case 3-2 is chosen is

$$p_2 = 1 - P(n, k_2, n - f, f + 1).$$

For Case 3-1, the non-candidate-full replica becomes candidate-full. For Case 3-2, the non-candidate-full replica becomes candidate-full by the coin tossing with probability
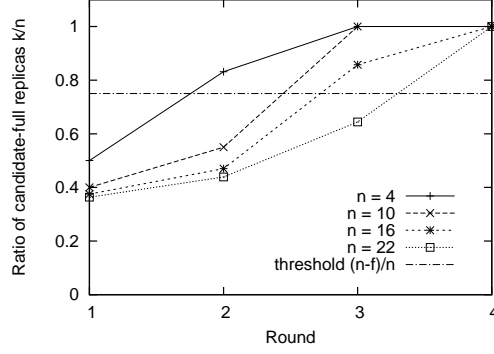
$$p_c = \alpha + (1 - \alpha) \cdot k_2/n.$$

Figure 3.2: Change of ratio of candidate-full replicas

Thus, the expected number after Step 3 is

$$k_3 = n \cdot (p_1/(p_1 + p_2) + p_2 \cdot p_c/(p_1 + p_2)).$$

Result and discussion: Next, we calculate how the ratio of candidate-full replicas changes based on the above equations, and Fig. 3.2 shows the result, where $\alpha = 0.1$ and $k = f + 1$. The four lines correspond to Cases $n = 4$, 10, 16, and 22. In each case, f is fixed to $\lfloor (n-1)/3 \rfloor$. The horizontal line is drawn at $(n-f)/n = 0.75$ as the threshold of the agreement. If the ratio exceeds this line, any non-candidate-full replica becomes candidate-full. For example, in Case n = 16, 38% of the replicas are candidate-full at the beginning of round 1, and after 2 rounds, it exceeds the threshold line. Therefore, at round 4, all the replicas become candidate-full. In any case, a configuration that all replicas become candidate-full is achieved quickly.

Evaluation of Transition Speed from non-SCV-full to SCV-full Configuration

In the previous evaluations, we showed that agreement by RSC protocol is achieved in a few rounds for any α in a SCV-full configuration. Here, we show that a SCV-full configuration is reached in a few rounds for practically reasonable sizes of n. Note that a SCV-full configuration is a configuration where the values of all non-faulty replicas' RI are the common-value. A replica's RI is updated in the following two cases:

Case 1: when there is a request contained in at least $f + 1$ received candidates at Steps 1, 2, and 3, a non-faulty replica adds the request to its RI.

Case 2: when a non-faulty replica receives a request by *INITIAL* message under external validity check, it adds the request to its RI.
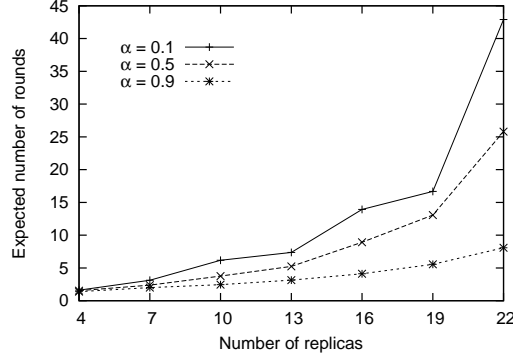
Figure 3.3: Number of rounds to become a SCV-full configuration

Although, both updates independently contribute to achieving a SCV-full configuration, the update by Case 2 depends on the communication speed on the networks between clients and replicas, which are uncontrollable in open networks like the Internet. Thus, we only evaluate the update with Case 1.

Simulation settings: We assume there are n requests and each request is included in the initial values of just $f+1$ different replicas fairly. This is the hardest setting for a request to be included in RI. Communication between replicas is simulated in the same way as Sec. 3.5.2. With this environment, we count the number of rounds to reach a SCV-full configuration or a configuration where all non-faulty replicas agree. The experiments are executed for $\alpha = 0.1$, 0.5, and 0.9, varying the number n of replicas from 4 to 22.

Result: The results are shown in Fig. 3.3 by plotting the averages of 1,000 executions for each case. A large α brings a rapid transition to the SCV-full configuration. Up to $n = 22$, the number of rounds is practically bounded.

3.5.3   Experimental Comparison with Other Protocols

Protocols to be compared: We compare our RSC protocol with two representative atomic broadcast protocols: ABC [16] and RITAS [19], which are built on binary Byzantine agreement protocols. An atomic broadcast protocol and a request set consensus protocol are equivalent in the meaning that one protocol is easily transformed to the other with an efficient procedure. The way of transformation from an atomic broadcast protocol to a request set consensus protocol is shown in Sec. 3.5.4. We use it for this experimental comparison.

RITAS internally executes Bracha's binary Byzantine agreement protocol [24] among the

replicas to guarantee agreement termination on the request sets. Bracha's protocol is itself a randomized algorithm and may possibly repeat a number of rounds for agreement, because termination depends on the probability that the values of the local coins independently tossed by replicas happen to coincide. On the other hand, since it does not employ any heavy cryptographic procedure except message authentication code (MAC), the duration of each round is very short.

ABC internally executes the binary Byzantine agreement protocol ABBA [29] O(1) times for each execution. ABBA performs shared coin tossing by using a dual threshold signature scheme. The number of rounds for the agreement is ideally small, while the duration of each round is long due to the heavy cryptographic procedures.

Experimental settings: We evaluate the latency and throughput of the replication based on each of the three protocols. The protocols were implemented by C++ language with POSIX library, and for ABC, we exploited H. Moniz's implementation of dual threshold cryptographic schemes[1]. We use seven machines as client hosts and $n = 3f + 1$ machines ($f = 1, 2, 3$) for the individual replicas, which are totally connected by one 1 Gbps network switch. The machines have a Core i3 540 3.07 GHz CPU and 2 GB RAM and run Linux 2.6.18.

Every client sends requests in a common frequency. We call the number of requests received by a replica every second *request frequency*, which we change to evaluate the latency and the throughput. We set up two models for the evaluation, and Table 3.1 depicts how requests are sent in the two models. One is a normal one, where each client multicasts the requests in the same order to the replicas. Therefore, the probability that replicas receive the requests in the same order is high. The other is a delayed one, where each client sends the requests in a different order among the replicas to represent delays of message delivery.

In RSC execution, we invoke the agreement protocol every five received requests or every millisecond. Note that because we cannot execute two agreement protocol instances in parallel for consistency, even if the scheduled timing is coming, we have to wait for the termination of the previously invoked agreement protocol. Therefore, a replica may newly receive more than five requests at the invocation and propose more than five. Through the experiments, probability α to choose the special coin value RI is set to 0.9.

Evaluation results: Figures 3.4 and 3.5 show the results of evaluating the throughput and the latency of RSC, RITAS, and ABC in the normal model, respectively. Similarly, Figs. 3.6 and 3.7 show the results in the delayed model. In each graph, the evaluation results for $n = 4$, 7 and 10 ($f = 1$, 2 and 3) are given.

In addition, Figs. 3.8 and 3.9 show the averages of the number of rounds, the duration of an

---

[1]http://sites.google.com/site/hmoniz/publications/ritas.zip

Table 3.1: Ways of sending requests in two models for $n = 4$

	1st send	2nd	3rd	4th	5th	6th	$\cdots$
Replica 1	r_1	r_2	r_3	r_4	r_5	r_6	$\cdots$
Replica 2	r_1	r_2	r_3	r_4	r_5	r_6	$\cdots$
Replica 3	r_1	r_2	r_3	r_4	r_5	r_6	$\cdots$
Replica 4	r_1	r_2	r_3	r_4	r_5	r_6	$\cdots$

(a) Normal model

	1st	2nd	3rd	4th	5th	6th	$\cdots$
Replica 1	r_1	r_2	r_3	r_4	r_5	r_6	$\cdots$
Replica 2	r_2	r_3	r_4	r_1	r_6	r_7	$\cdots$
Replica 3	r_3	r_4	r_1	r_2	r_7	r_8	$\cdots$
Replica 4	r_4	r_1	r_2	r_3	r_8	r_5	$\cdots$

(b) Delayed model

agreement, and the size of the agreed set (i.e., request set output of the agreements) by RSC and RITAS for the two models respectively, which are presented to analyze the latency results of RSC and RITAS for $n = 4$.
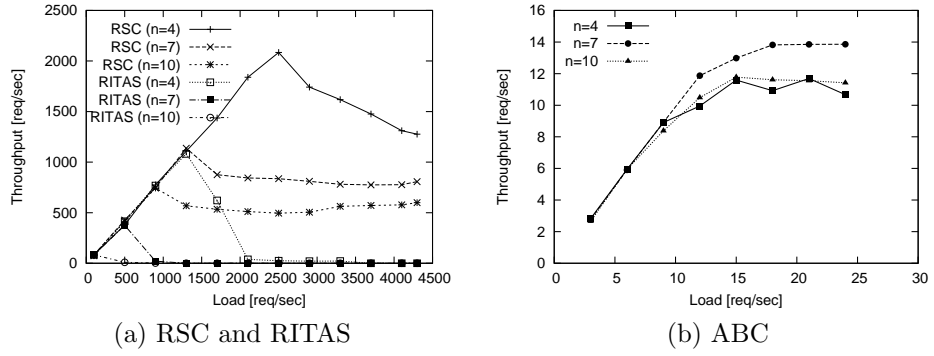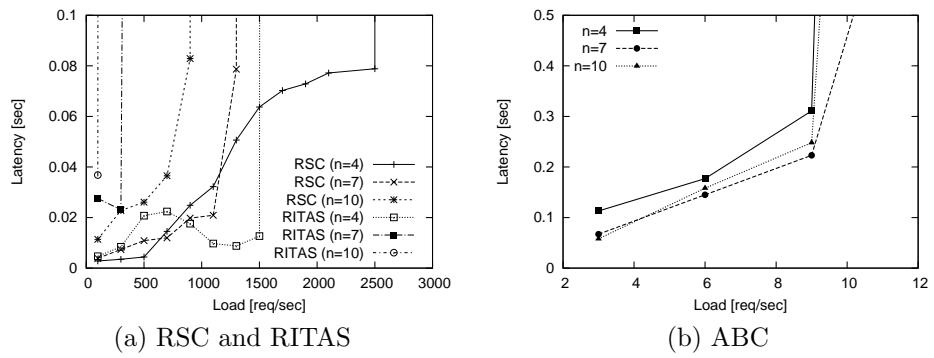
(a) RSC and RITAS   (b) ABC

Figure 3.4: Throughput for normal model

(a) RSC and RITAS   (b) ABC
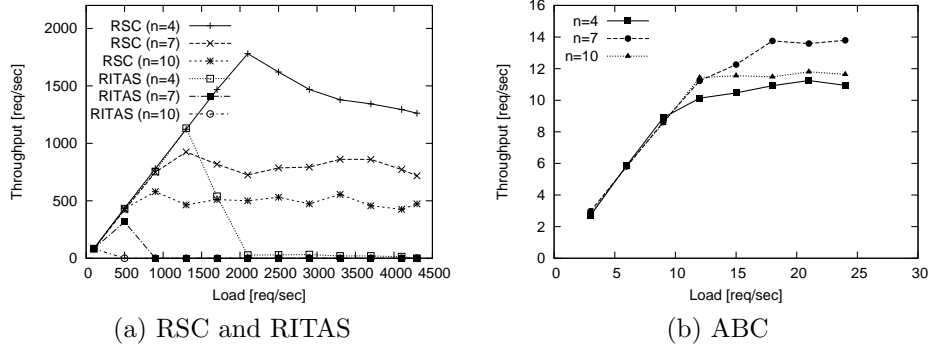
Figure 3.5: Latency for normal model

(a) RSC and RITAS

(b) ABC
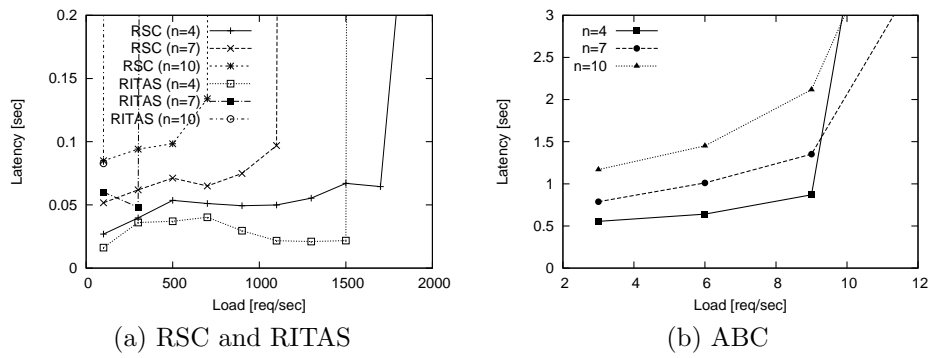
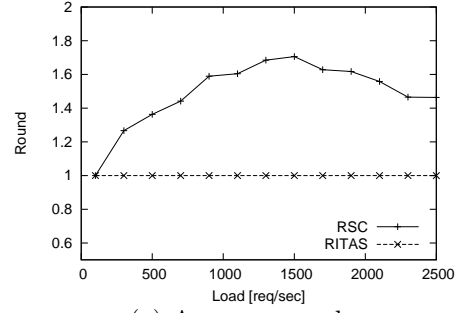Figure 3.6: Throughput for delayed model
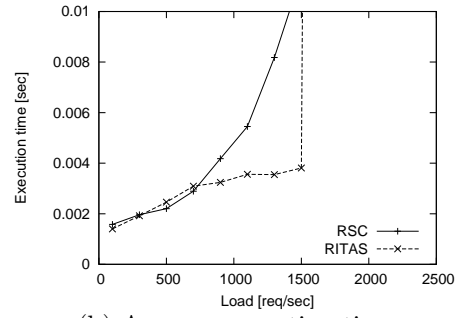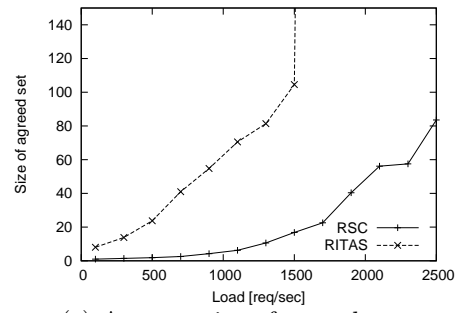

(a) RSC and RITAS

(b) ABC

Figure 3.7: Latency for delayed model
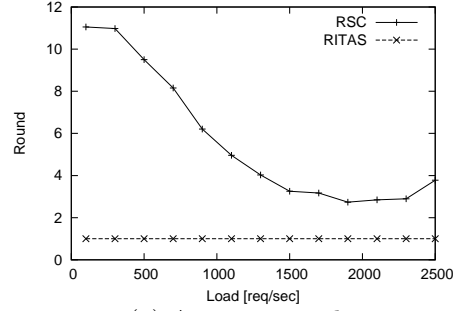
(a) Average round
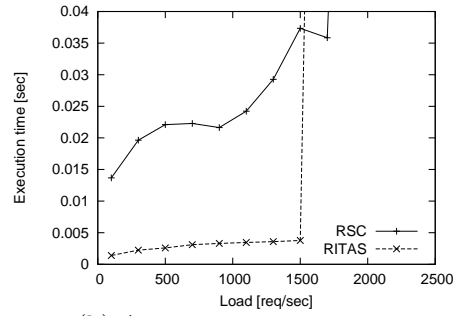

(b) Average execution time
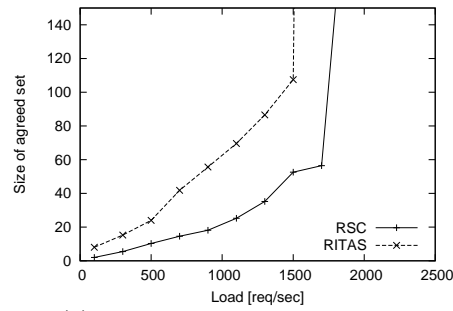

(c) Average size of agreed sets

Figure 3.8: Characteristics of agreements of RSC and RITAS for $n = 4$ in normal model

(a) Average round

(b) Average execution time

(c) Average size of agreed sets

Figure 3.9: Characteristics of agreements of RSC and RITAS for $n = 4$ in delayed model

In general, the throughput values monotonically increase and at some point begin to decrease or remain stable when the request frequency grows. At the point of the request frequency, the system reaches the resource bound, and at the same point, the latency suddenly increases markedly. Such resource bounds can be seen in the figures.

In observing Figs. 3.4 and 3.5, the ABC performance is much worse than RSC and RITAS. ABC has much longer latency than RSC and RITAS and reaches the resource bound faster. We can reason that the cryptographic primitives employed in ABC cause this. On the other hand, the latency and the request frequency of the resource bound of ABC are not much different among $n = 4, 7,$ and 10.

In comparison of RSC and RITAS, RSC reaches the resource bound later than RITAS, especially for $n = 7$ and 10. This means that RSC consumes fewer resources than RITAS does. As for latency, RSC outperforms RITAS for $n = 7$ and 10, which is affected by their resource bounds. For $n = 4$, RSC is better until 800 requests/sec, and after that, RITAS is better until the resource bound. To understand the RSC and RITAS behaviors in more detail, we consulted the result of Fig. 3.8. In RITAS, the average execution time for an agreement increased slowly, and the average size of a request set output by an agreement also increased monotonically. It is obvious that when the average execution time increases, the latency increases, and when the size of the request set increases, the latency decreases, because more requests can be processed for an execution of agreement. We can observe in Fig. 3.5 that the RITAS latency first increases, next remains stable, and then decreases. And now we can reason that the effect of increasing the average execution time for agreement is dominant first, next it is in balance with the increasing size of the agreed set, and then the increase becomes dominant. For RSC, the average size of requests is stable while the average execution time is increasing. Therefore, the latency is increasing monotonically.

Next, we consider Figs. 3.6 and 3.7 of the delayed model. ABC again shows worse performance compared with RSC and RITAS as does in the normal model. In this model, however, ABC's latencies are different among $n = 4$, 7, and 10, while their resource bounds are the same. As for throughput, RSC and RITAS show almost the same results as the normal model. The latencies of RSC and RITAS worsen as a whole, but the relation between RSC and RITAS is the same as the one in the normal model. The shape of RSC's latency is different from that in the normal model. First, it increases until 400 requests/sec and remains almost stable until the resource bound. Fig. 3.9 explains this behavior. The average execution time for an agreement increases until 400 requests/sec and then the increase slows down. On the other hand, the average size of a request set output by an agreement increases monotonically. After 400 requests/sec, the average

execution time and the average size, whose increases negatively and positively affect the latency respectively, are in balance so that the latency is stable.

Note how the results of the number of rounds in Figs. 3.8 and 3.9 clarify the differences of the characteristics between RSC and RITAS. RITAS, which keeps the number of rounds to 1 through all request frequencies for the both models, internally executes binary Byzantine agreements to terminate the agreements on a request set. Before starting the binary agreement, RITAS makes the replicas merge the requests for their proposals by communicating with each other so that the binary agreement can quickly terminate. With this device, RITAS achieves one round agreement even in the delayed model.

On the other hand, RSC executes multi-valued Byzantine agreements and needs more rounds in general. In the results, the number of rounds is small and stable in the normal model, but it is much larger and decreases as the load becomes higher in the delayed model. We can reason that the difference is caused by the external validity check. In the normal model, the validity check is accomplished quickly for every replica. On the other hand, in the delayed model, if some requests are delayed over rounds, more time is needed to finish the external validity check, and so more rounds are needed to agree with the special coin value among the replicas. However, when the load increases, the number of rounds decreases. Because, the duration of a round becomes longer, and the time needed for the validity check is encapsulated in the duration.

Summarizing the evaluation result, for the smallest $n = 4$, RSC and RITAS have similar load performances of low request frequencies and RSC accepts a higher request frequency than RITAS. For larger $n = 7$ and 10, RSC has more advantages in both response time and resource bound. ABC's performance is inferior to RSC and RITAS, but its performance does not change over the number of replicas. These characteristics can make ABC more attractive in constructing robust replication in a slower communication network like the Internet or in the future when the hardware cryptographic processing is available.

3.5.4   Request Set Consensus Using Atomic Broadcast

Here, we describe how to realize a request set consensus protocol by using an atomic broadcast protocol with a common idea described in [16]. This protocol is used in Sec. 3.5.3 to compare ABC and RITAS with RSC; since they are atomic broadcast protocols and not request set consensus ones, we cannot directly compare them with RSC.

Atomic broadcast is defined by the following four requirements. We write "a-broadcast m" for broadcasting a message m by an atomic broadcast protocol and "a-deliver m" for accepting a
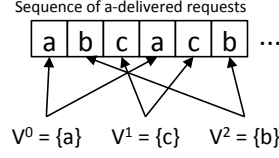
Sequence of a-delivered requests

| a | b | c | a | c | b | ⋯ |

$V^0 = \{a\}$ $V^1 = \{c\}$ $V^2 = \{b\}$

Figure 3.10: Flow of deciding agreed values

Protocol 3.3 Protocol for Request Set Consensus using Atomic Broadcast

---

```
 1: Initialization:
 2:     i := 0;
 3: When a request r is delivered from a client:
 4:     A-broadcast r;
 5: When a request r is a-delivered:
 6:     Record the reception of the request r;
 7:     if this is the (f + 1)-th a-deliver of the request r then
 8:         V_p^i := {r};
 9:         i := i + 1;
10:     end if
```

---

broadcast message m in the protocol.

Validity: If a non-faulty process a-broadcasts a message m, then some non-faulty process eventually a-delivers the message m.

Agreement: If a non-faulty process a-delivers a message m, then all other non-faulty processes eventually a-deliver m.

Integrity: Every non-faulty process a-delivers any given message m at most once, and only if m was previously a-broadcast.

Total Order: If two non-faulty processes a-deliver two messages m_1 and m_2, then they a-deliver them in the same order.

When a replica receives a request from a client, it a-broadcasts the request. When a replica has a-delivered identical request r for $f + 1$ times from different replicas, it sets agreed value V^i to $\{r\}$ (Fig. 3.10). By Total Order, the non-faulty replicas agree with the same value, and by collecting the same $f + 1$ requests, no forged request is included in the agreed value. Protocol 3.3 shows the protocol details.

3.6   Concluding Remarks

In this chapter, we proposed a randomized Byzantine fault-tolerant request set consensus protocol RSC to realize efficient state machine replication in asynchronous communication environments. In this protocol, we introduced a new method of multi-valued local coin toss for consensus. In general, multi-valued local coin toss for consensus is thought to be very inefficient. However, our RSC protocol has a property in which the domain of the coin values is dynamically narrowed. By simulation experiments, we showed that our protocol reached a consensus in fewer rounds and outperformed RITAS and ABC, two well-known randomized Byzantine fault-tolerant atomic broadcast protocols, in a network where replicas communicate with each other at high speed.

Chapter 4

Parallelizing Consensuses to Reduce Latency

4.1 Introduction

In the randomization approach, randomized actions are introduced to avoid critical damage from attackers. However, the approach is likely to be inefficient, since a number of rounds must be repeated until the correct replicas reach agreement. To improve efficiency, a request set agreement is employed rather than an agreement on a sequential number (the order to be processed) of each request. Once agreement on a request set is achieved, the requests in the set are processed in a predefined order (e.g., the order of the IDs of the clients submitting requests) among them. This request set agreement is repeated sequentially and all requests are arranged in a common order. However, if some replicas work very slowly or some requests reach very late, a request set agreement may take a long time. It seriously delays the next invocation of the consensus protocol. This chapter presents a method of solving this problem by parallelizing the request set agreements.

Next, we explain more details of the randomization approach and the involved problem. Many randomized protocols based on request set agreement have been already proposed [17, 16, 32, 33]. The consensus protocol is invoked periodically with a given time interval measured by a local clock of each replica. When an execution of the protocol is finished by agreeing on a request set, the requests in the set are arranged in a predefined order. By this series of arrangements, all the requests are arranged in a common order among the replicas. At each invocation of the consensus

protocol, each replica proposes a set of the requests that were received so far but not included in the previous agreement results. Of course, these proposals can be different among the replicas because of the delay of the request arrival or the machine behavior. However, the set agreement protocol guarantees that all non-faulty replicas agree on a subset of the union of the request sets proposed by non-faulty replicas.

The length of the local time interval between invocations of the set agreement affects the efficiency, but it is difficult to decide a suitable one. If it is short, the number of invocations of the consensus protocol will increase. If it is long, requests have to wait long for the invocation of the agreement protocol, and the agreement may take a long time because the size of the proposal grows. When an execution of the consensus protocol does not terminate within the local time interval, a big delay might occur. In this case, the invocation of the consensus protocol is kept waiting until the termination of the previous consensus, even if the local time interval passes to prevent inconsistency of the total order of requests among the replicas. Such blocking of the invocation makes the following invocations of agreement postponed. As a result, the number of unprocessed requests grows and the efficiency of the replication method is reduced. When request arrivals or machine behaviors are delayed, the validity check becomes very time consuming in the agreement, and the termination is easily delayed over the local time interval. Here, the validity check is a process in the agreement for excluding forged requests.

4.1.1 Contributions

To solve the above problem, we introduce a method that allows parallel execution of agreements so that executions of the set consensus protocol are not blocked by delayed requests or machines. Our experimental results show that our parallelization method greatly improves the efficiency compared with a sequential method, especially three or four times faster when some requests are delayed or some replicas work slowly.

We solved the following two technical issues:

Safety problem: The parallel executions of the set consensus protocol may terminate in different orders among the replicas. For example, on one replica, the execution of the agreement initiated first terminates after the one initiated second, and on another replica, the one initiated first terminates first. When the replicas are restricted to process the requests in the invocation order of the agreements, they have to wait until the delayed agreement is completed. This may reduce the efficiency achieved by parallelization. Therefore, we have to consistently arrange the outputs (or request sets) of the parallel executions among the replicas.

Liveness problem: A request contained in the proposal made by a replica is not necessarily included in the output of the corresponding agreement. Therefore, to guarantee the liveness that a request is eventually processed, a replica has to keep proposing the request until it is included in an output of the agreements. On the other hand, a Byzantine faulty replica may propose the request set that includes a forged request, which can collapse the replicated server state. To exclude such requests, a replica validates the requests contained in the proposals of other replicas. One possible way of validating a request is to wait for receiving it directly from a client [32, 33]. However, if the arrivals of the request are delayed at some correct replicas, the validation takes a long time for the clients, and the agreement is delayed. This causes a situation that a request that delays agreement can commonly be included in the proposals of the parallel executions of the agreement. This reduces the positive effects of parallelization.

To solve the safety problem, we introduce another agreement process in the replication protocol that identically arranges the output of the parallel executions among the replicas. We show that this additional agreement's overhead is small by experimentally evaluating the performance.

To solve the liveness problem, we introduce randomization to decide the proposals of each execution of the consensus protocol. The requests in the proposal are chosen randomly from the requests that have already been received but have not been processed. A request that causes a delay in a previous execution may be missed in this choice, and a new execution can have no delay. We experimentally show that this randomization brings a reasonable advantage of response time.

4.1.2 Related work

As stated above, there are two main approaches for replications based on Byzantine agreement in asynchronous distributed systems: randomization [17, 16, 32, 33] and a rotating coordinator [9, 10, 11, 34].

In the rotating coordinator approach, a special replica (a rotating coordinator) determines a sequence number (the processing order) for each received request and announces it to all the other replicas. Therefore, all the replicas can process the requests in the same order and maintain consistency. If the coordinator is suspected to be faulty, its role is taken over by another replica. From the impossibility result of Fischer et al. [7], this approach needs some assumptions on synchrony (weak synchrony) to guarantee termination. On the other hand, the randomization approach guarantees termination with probability 1 and needs no additional assumption. It is more robust but less efficient.

Among the protocols in the coordinator approach, the Castro-Liskov protocol [9] achieves very high performance and is considered a practical replication method. Under the above assumption, it terminates in a few rounds and executions of the consensus protocol are executed in parallel. Although the original Castro-Liskov protocol executes the consensus protocol for each request, it is not hard to modify the protocol to allow each process to propose a request set like the randomization approach. However, parallel execution of the agreements for request sets in the coordinate approach is essentially different from that in the randomization approach. Actually, the modification of the Castro-Liskov protocol reduces the number of agreement executions and consequently improves efficiency in ordinary situations. However, it worsens when requests or replicas are delayed. Because of the delay, a coordinator is suspected to be faulty and coordinator alternation often happens. At each alternation, a heavy load procedure must be done to maintain this protocol's integrity.

For the existing protocols in the randomization approach, to the best of our knowledge, our parallelization proposal is the first.

4.2   Replciation by Request Set Consensus (RSC)

We introduce a state machine replication method based on Byzantine consensus on a set of requests (called *Request Set Consensus* (RSC)), which is commonly used in replications in completely asynchronous distributed systems to accelerate replication execution.

In this replication method, a replica periodically initiates RSC with a predefined interval. We denote the sequence of RSC executions by $RSC^1, RSC^2, \cdots$. A replica maintains the *arrived request set* to store the set of the requests that have already been received but have not yet been processed; a request is added to the set when it is received, and it is removed when it is processed. When a replica initiates RSC^k, its proposal is the set of the requests stored in the arrived request set. Let the output (a set of requests) of RSC^k be V_k. Requests are processed in the order of $V_1, V_2, \ldots$, and the requests in each V_i are serialized in a deterministic order shared among the replicas. In the existing methods, the initiation of RSC^{k+1} must be delayed until RSC^k is finished to maintain the consistency of the processing order of requests, even if it passes the scheduled initiation time of RSC^{k+1} (*Initiation Condition*).

To ensure the safety and liveness requirements for state machine replication, the RSC protocol must satisfy the following requirements. Hereafter we denote an execution of RSC^i at a replica with proposal v by $RSC^i(v)$ or RSC^i if the proposal does not matter.

RSC Agreement No distinct correct replicas output different sets of requests.
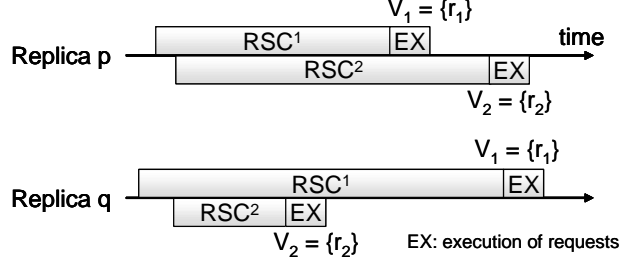
Figure 4.1: Invalid parallel executions of RSC

RSC Validity The output set is a subset of the union of the proposals of all correct replicas.

RSC Termination Every correct replica eventually outputs a set of requests.

RSC Integrity A request contained in the proposals of all correct replicas is also contained in the output.

RSC Agreement, *Validity*, and *Termination* are standard requirements for Byzantine consensus protocols. *RSC Integrity* suffices to guarantee the *Liveness requirement* of state machine replication.

4.3 Parallelizing Executions of RSC

4.3.1 Problem with Parallelization

Executions of existing replication methods can be very slow due to the initiation condition mentioned above, especially when the behaviors of some replicas are delayed or requests reach some replicas late. One idea to improve the efficiency of the replication method is parallelizing the executions of RSC by consistently removing the initiation condition. To achieve this, we have to solve the following two problems.

Safety problem: Since the delays of the communication links among replicas and clients are different from each other in asynchronous systems, the order of finishing the RSC executed in parallel can be different among the replicas. In Fig. 4.1, replica p finishes RSC^1 first, while replica q finishes RSC^2 first. If a replica immediately executes requests after the agreements, then the processing orders of the requests are not the same among replicas p and q, and the safety condition is not guaranteed.
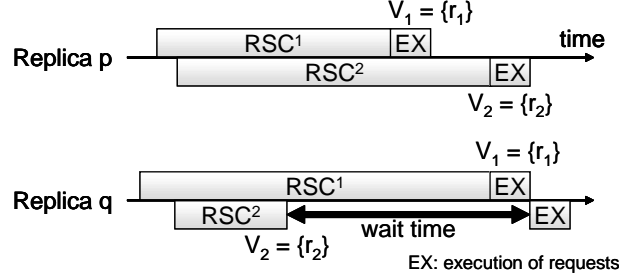
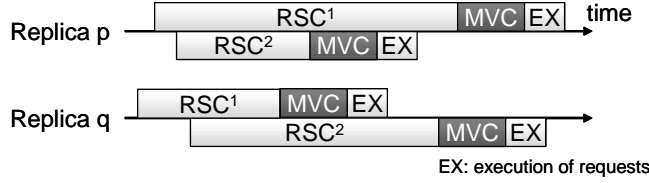Figure 4.2: Ineffective parallel executions of RSC

Figure 4.3: Execution of our proposed parallelizing method

This problem can be simply resolved by waiting for the terminations of all RSC^j $(j < i)$ before processing V_i the agreed set of requests. However, the method can cause great overhead (Fig. 4.2), where replica q has to wait for the termination of RSC^1 to process V_2. If a RSC takes a long time, all requests already agreed by the following RSCs have to wait to be processed until the previous RSC is terminated.

Liveness problem: Even if we reduce the overhead of waiting for the termination of other RSC executions, inefficiency caused by the delayed replicas or the delayed requests remains. A request included in a proposal may not be included in the output. Therefore, the replica must keep proposing the request until it is included in an output of RSC to guarantee the liveness requirement for state machine replication.

In such a naive parallelization, the proposal of RSC^{j+1} is likely to contain a request in that of RSC^j. However, if the request is greatly delayed for some replicas, the validity check in the protocol commonly takes a long time for both executions RSC^j and RSC^{j+1}. Therefore, a few delayed requests may cause big delays in the parallel execution of RSC.

4.3.2 Our Approach

To solve the safety problem, we introduce a multi-valued consensus (MVC) in the parallelization. When an execution of RSC^j is finished in a replica with output rs_j, the replica initiates MVC with proposal (j, rs_j) (Fig. 4.3). If MVC outputs agreed value (id, rs_{id}), the replicas process the requests in rs_{id} in an arbitrary predefined order. All correct replicas clearly process the same requests in the same order. Note that MVC is itself executed sequentially on each replica. An important point of this method is that the replica does not have to wait for the termination of RSC^i ($i < id$). In addition, even if RSC^{id} has not finished at the replica, it can process the requests in rs_{id} since the replica can learn the requests from the MVC output.

To solve the liveness problem, we introduce randomization for deciding the proposal. We decide the RSC proposal by probabilistically choosing requests from the set of requests already received but not yet processed. With this simple modification, we can decrease the probability that two proposals of two distinct RSC executions include the same request. Consequently, the terminations of successive RSCs executed in parallel are rarely delayed by the same request in the two proposals. At the same time, we can guarantee the liveness requirement with probability 1.

4.3.3 Multi-valued Consensus Protocol

We show the requirements for the multi-valued consensus protocol used to determine the request set to be processed first. The MVC proposal at a correct replica is a pair of ID of a terminated RSC execution and its agreed request set. The MVC protocol is, of course, the randomized protocol, because the targeted distributed system is asynchronous and Byzantine faulty. The MVC protocol must satisfy the following requirements:

MVC Agreement No distinct correct processes output different values.

MVC Validity If the proposals of all correct processes are the same, the agreed value is the proposal.

MVC Termination Every correct process eventually outputs an agreed value.

MVC Extra Validity The output of a correct process must be a proposal of some correct process.

MVC Agreement, *Validity*, and *Termination* are the common requirements for MVC in general. *MVC Extra Validity* speeds up state machine replication while avoiding forged requests explained

in Sec. 4.3.4. MVC Extra Validity is feasible using a signature scheme on an existing MVC protocol. Each replica repeatedly executes MVC, and we denote the i th execution of MVC by MVC^i.

4.3.4　Protocol

Our proposed parallelizing method is shown in Protocol 4.1. The value of *input_rs* is a set of requests given to RSC as a proposal and is modified based on *old_rs* and *new_rs*. The value of *old_rs* is a set of requests that were received before the last RSC initiation and remain unprocessed. The value of *new_rs* is a set of requests that were received after the last RSC initiation. The value of *agreed_rs* is a set of requests that belong to RSC output. *rsc_id_queue* is a queue of pairs (j, rs_j) of RSC ID j and agreed set rs_j output by the execution of the RSC with ID j. An element of the queue is a proposal of MVC. *wait_queue* is a queue of agreed request sets, and a thread $T_{process}$ processes them in order. *mvc_id* is a counter that gives a sequence number to each execution of MVC, allowing replicas to recognize a common execution of MVC.

We assume that each replica has its own special scheduler PS, which employs a local clock of the replica. PS periodically outputs positive integers 0, 1, 2, ... in this order with a predefined interval. When PS outputs number k, the replica initiates the k th execution of RSC with ID k. The shorter the PS interval is, the more frequently RSC is initiated.

We introduce *choose* function to process requests faster while keeping liveness property. The function forms a request set from the elements of *old_rs* and *new_rs*, which is used as input for newly initiated RSC. Here, *old_rs* and *new_rs* are the sets of requests that have not been included in any MVC output yet, i.e., their processing orders have not been assigned yet. The function resolves the following problems. To guarantee liveness, the requests must be included in RSC proposals repeatedly until their processing orders are assigned. A naive way to realize this is to include all the requests of *old_rs* and *new_rs* in every RSC input for the proposal. However, if a proposal contains a request that arrives late at $n - f$ replicas, the termination of the RSC execution is also delayed. To solve the problem, *choose* function chooses a part of the requests in *old_rs* and *new_rs* randomly for RSC input. As a result, we can reduce the risk that the termination of the RSC execution is delayed. There are many ways to choose the requests randomly, and we employ a simple way that chooses each request with a constant probability. By the simple way, liveness is guaranteed with probability 1. In addition, we experimentally show in Sec. 4.5.2 that *choose* function has a good effect on performance.

A replica initiates MVC with a proposal of a pair of an RSC ID and its agreed set. If MVC^j

outputs the agreed value (id, V), the replica processes V at the j th turn. The MVC proposal includes the corresponding agreed set as well as the RSC ID to improve the efficiency. If the proposal is only RSC ID, when MVC outputs RSC ID id and the replica has not finished the execution of the RSC of id, it has to wait for the termination of the RSC before processing the requests in the agreed set. With the agreed value in the output of MVC and *MVC extra validity*, which means that the agreed value is not forged, a replica can process the correct request set immediately after the MVC outputs.

Our method starts from initialization in which a replica creates a new thread $T_{process}$. $T_{process}$ dequeues a request set from *wait_queue* and processes the elements in a deterministic order shared with all replicas.

Our protocol has four *when* clauses (the line numbers at the end of each *when* clause are of Protocol 4.1):

- When a new request arrives from a client, it is added to *new_rs*. (lines 13–14)

- When scheduler PS outputs value j, first, the already agreed requests are removed from *old_rs* and *new_rs*. Next, the proposal for a new RSC is calculated using given function *choose*. The *choose* function randomly selects requests from its input *old_rs* and *new_rs* in a predefined manner. Then a new RSC with ID j is initiated, and the elements in *new_rs* are moved to *old_rs*. (lines 15–21)

- When an RSC execution with ID id is finished with output rs, a replica updates *agreed_rs* and enqueues a pair (id, rs) to *rsc_id_queue* if id is not in *agreed_rsc_id*. If a previously invoked MVC is running, it waits for the termination. Then the replica chooses the first element, a pair of an RSC ID and an agreed set (id', rs') from *rsc_id_queue* (without deleting it from the queue) and initiates a new MVC with ID *mvc_id* with the proposal (id', rs'). Lastly, the replica increments the value of *mvc_id*. (lines 22–30)

- When MVC outputs value (id, rs), a replica removes the pair whose first element is id from *rsc_id_queue* and enqueues rs into *wait_queue* and id into *agreed_rsc_id*. (lines 31–35)

4.4 Correctness

We prove that our proposed protocol, which parallelizes RSC executions, satisfies the safety and liveness requirements of state machine replication.

Protocol 4.1 Proposed parallelizing method

1: **Variables**
2:    $input\_rs := \emptyset$; {input of RSC}
3:    $old\_rs := \emptyset$; {requests received before the last RSC}
4:    $new\_rs := \emptyset$; {requests received after the last RSC}
5:    $agreed\_rs := \emptyset$; {agreed requests}
6:    $agreed\_rsc\_id := \emptyset$; {RSC IDs agreed by MVC}
7:    $prs := \emptyset$; {processed requests}
8:    $mvc\_id := 1$; {counter for MVC IDs}
9:    $rsc\_id\_queue :=$ empty; {queue of pairs of RSC ID and a set of requests }
10:    $wait\_queue :=$ empty; {queue of agreed sets waiting to be processed}
11: **Initialization**
12:    start task $T_{process}$;
13: **When** a request r arrives **do**
14:    $new\_rs := new\_rs \cup \{r\}$;
15: **When** PS outputs j **do**
16:    $old\_rs := old\_rs \setminus agreed\_rs$;
17:    $new\_rs := new\_rs \setminus agreed\_rs$;
18:    $input\_rs := choose(old\_rs, new\_rs)$;
19:    invoke $RSC^j(input\_rs)$;
20:    $old\_rs := old\_rs \cup new\_rs$;
21:    $new\_rs := \emptyset$;
22: **When** RSC^{id} outputs its agreed value rs **do**
23:    **if** $id \notin agreed\_rsc\_id$ **then**
24:      $agreed\_rs := agreed\_rs \cup rs$;
25:      enqueue (id, rs) into rsc_id_queue;
26:    **if** MVC is running **then**
27:      wait until it terminates;
28:    let (id', rs') be the first element of rsc_id_queue;
29:    invoke $MVC^{mvcid}(id', rs')$;
30:    $mvcid := mvcid + 1$;
31: **When** MVC^i outputs its agreed value (id, rs) **do**
32:    **if** rsc_id_queue contains $(id, *)$ **then**
33:      remove $(id, *)$ from rsc_id_queue;
34:    enqueue rs into $wait\_queue$;
35:    $agreed\_rsc\_id := agreed\_rsc\_id \cup \{id\}$;
36: **Task** $T_{process}$
37:    **loop**
38:      **wait until** $wait\_queue$ is not empty;
39:      dequeue rs from $wait\_queue$;
40:      **for all** $r \in (rs \setminus prs)$ in some deterministic order **do**
41:        execute r and send the result to the client;
42:      $prs := prs \cup rs$;

4.4.1 Safety

We have to show that requests are processed in the same order among the correct replicas and that no forged requests are included in them.

To show that requests are processed in the same order, it is sufficient to show that RSC outputs are enqueued to *wait_queue* in the same order among the replicas under RSC agreement, since thread *T_process* processes the requests in the order in which they are stored in *wait_queue* (line 39). On the other hand, enqueuing is executed only in the event of MVC output, and MVC is executed sequentially (lines 26–30). Therefore, the desired result follows from the MVC agreement. The non-forged requirement immediately follows from RSC validity and MVC extra validity. □

4.4.2 Liveness

Here we show that a request sent by a client will be eventually processed. Assume that there exists a request rq that is never processed. Such a request is eventually delivered to all correct replicas and stored in their *new_rs* or *old_rs*. Hence, there must be an RSC execution with some probability in which every correct replica contains rq in its proposal. This is achieved by *choose* function. Let the ID of the execution be k. By RSC termination, the execution must terminate, and by RSC integrity, agreed set V_k must contain rq. Then every correct replica enqueues (k, V_k) into *rsc_id_queue*. Assume that (k, V_k) has never been chosen as an output of any MVC execution. *rsc_id_queue* is a queue, and if (k, V_k) is not removed for a long time, (k, V_k) moves to the front of the *rsc_id_queue*. If the front of the *rsc_id_queue* of every correct replica gets (k, V_k), every correct replica proposes (k, V_k) and the agreed value of the next MVC execution must be (k, V_k) by MVC validity. The execution must terminate by MVC termination. Therefore, request rq is eventually processed, which contradicts the assumption. □

4.5 Performance Evaluation

In this section, we experimentally compare the performance of state machine replication employing our proposed parallelizing method with an existing one based on sequential agreements. In particular, we show how the delay of request message delivery and machine behavior affects the response time of the requests. Moreover, we compare scalability, i.e., the ability to process a large number of requests sent by many clients, of the two methods from the viewpoints of throughput and latency.

In this section, we compare the performance of the parallelized Byzantine fault tolerant replication proposed in this chapter and the sequential one by experiments. Although our proposed parallelized method is Byzantine fault tolerant, in the experiments, we do not model Byzantine behavior as malicious behavior of a faulty replica, where it sends invalid messages not satisfying the protocol. The reason is as follows. Most Byzantine fault tolerant protocols use reliable broadcast and validity check mechanisms [24, 33, 32, 16]. The malicious behaviors mentioned above are detected immediately by these mechanisms, and cannot have an effect on performance of the replication. Therefore, meaningful attacks that faulty replicas can do are restricted to delayed behaviors, e.g. delaying message delivery. By the reason above, we consider only delayed behaviors of faulty replicas' attacks in the experimental performance evaluation here. Moreover, delays in arrivals of requests from clients at some replicas and delays in correct replicas' actions also affect performance of the replication. To analyze how these three kinds of delay affect performances of the sequential and parallelized replications and in what situations of the delays the difference of the performances become larger, we introduce a simple model that simulates these delays. The model is controlled by the four parameters: $\#d\_req$, $\#d\_rcv$, ed_req, and ed_mac, described below. To the best of our knowledge, this is the first one that evaluates the effect of these delays in detail for randomization approach.

4.5.1　Experiment environment

For our experiments, we use five machines completely connected by one network switch. Four of them are used to run replicas and the other one is used to simulate clients. On each of the four machines, a replica is running individually. On the other machine, several clients are simulated, and their requests are issued from it. Each machine has a Core i3 540 3.07 GHz CPU and 2 GB RAM and runs Linux 2.6.18. The network is 1 Gbps LAN.

Through the experiments, we fix the *choose* function so that it uniformly chooses every element as an element of a proposal with 0.25 probability. This value is empirically preferable for the parallelization as shown in Sec. 4.5.2.

We used the RSA protocol proposed in [32, 33] as an underlying RSC protocol and the M_V_Consensus protocol proposed in [19] as the MVC protocol. These protocols and our proposed parallelizing method were implemented by C++ language with POSIX socket library for the evaluation. Note that the M_V_Consensus protocol may output a special value, $\perp$, which is different from any proposed value. To cope with this exceptional value, we slightly modified our protocol. When this value is output, we reinvoke M_V_Consensus protocol with a different

proposal: the element of *rsc_id_queue* whose RSC ID is the smallest. If the repetition of this reinovocation continues, the proposals finally coincide among the replicas, and the invocation terminates by outputting the proposal of a normal value by the M_V_Consensus property stated in Theorem 3 in [19]. Then, the repetition is finished.

4.5.2 Latency

Evaluation model

To evaluate latency of the proposed method, we measure the response time, which takes at a client until accepting the response after sending a request. It should be noted that a client accepts a response when it receives $f + 1$ identical responses from different replicas. In the setting of our experiments, each request is issued to replicas every 100 ms. The local time interval for invoking RSC is 100 ms. For each combination of the parameter values described below, we execute the experiments 50 times and average the response times.

From the machine that simulates clients, 50 requests are multicast to the replicas in total. Let $r_1, r_2, \ldots, r_{50}$ be the requests issued from the clients. To realize delayed delivery of the requests, we change the order of sending the requests. For example, if the delivery of request r_1 is delayed for replica R_1, we send the requests to the replicas other than R_1 in the order $r_1, r_2, \ldots, r_{50}$ and the requests to R_1 in the order $r_2, r_3, \ldots, r_{25}, r_1, r_{26}, \ldots, r_{50}$. To realize delayed behavior of the replicas, we delay the timing to start sending the requests to replicas. For example, if the behavior of replica R_1 is delayed, we start sending the requests to R_1 after sending 25 requests to the other replicas.

We introduce the following parameters and values to configure this model:

#*d_req*: the number of delayed requests, which is chosen from {1,2,3}.

#*d_rcv*: the number of machines that receive delayed requests. Their values are chosen from {2*,3*}, where we attach "*" to distinguish them from the values of #*d_req*.

ed_req: extent of how much requests are delayed. The values are chosen from {*middle*, *end*}, in short, {m, e}.

ed_mac: extent of how much a machine's behavior is delayed. The values are chosen from {0%, 50%, 100%}.

The values, *middle* and *end*, of *ed_req* mean that the first requests are moved afterward to the middle and to the end of the order of the sequence of requests, respectively. For example, if

$\#d\_req = 2$ and $ed\_req = middle$, r_1 and r_2 are moved between r_{25} and r_{26}, and if $ed\_req = end$, they are moved after r_{50}. We assume that at most one machine can be delayed, which is called a *delayed replica*. The value of 0% of ed_mac means that there is no machine delay. 50% and 100% mean that the sending of the requests to the delayed replica starts when the sending of the requests for the other machines has progressed 50% and 100%, respectively. Machine delay ed_mac implies delays of all the requests, and request delay ed_req does delay some requests.

Now, we explain the adequacy of our model. As described above, the purpose of the experiments is to evaluate how the delayed requests and the delayed behavior of replicas (including faulty one) affect the performance of replication. Since we are assuming $f = 1$ throughout the experiments, the number of replicas n is limited to four. A factor that affects directly to the performance of replication, e.g., throughput and latency, is the length of time for reaching an agreement. The length for RSC depends on the number of the different values of the values that replicas propose for agreement. If every correct replica happens to propose the same value, the RSC execution reaches an agreement immediately. On the other hand, it takes long time to reach an agreement when all the proposals from replicas are different to each other. By choosing the number of delayed requests $\#d\_req$ from 1, 2, and 3, we can enumerate three situations where the numbers of the different values that replicas propose are 2, 3, or 4 respectively. Thus, we can cover all the possible situations for the number of the different proposals in the case of $n = 4$. On the other hand, since we are targeting an asynchronous distributed system, the delayed behavior of correct replicas also affects the length to reach an agreement. The delayed behavior of a correct replica does not affect the performance when the faulty replica works correctly, because the underlying Byzantine request set consensus protocol we use here is designed to be able to reach an agreement with three correct replicas. Note that the case where a faulty replica and a correct replica behave slowly can be simulated by the case where two correct replicas behave slowly. Consequently, it is sufficient to consider the two cases of the delayed behavior: two replicas behave slowly and three ones does. In our model, a delayed behavior of correct replicas is realized by delayed reception of requests. Then, all substantial patterns of delayed behaviors of replicas is covered by the values of ed_mac and $\#d\_rcv$ chosen from $\{2, 3\}$ (obviously, delayed behaviors of all the four replicas is not meaningful). Moreover, to model the degrees of the delays of requests and a faulty replica, we introduce the parameters ed_req and ed_mac respectively.

Experimental results and analysis

The average response times of the sequential and parallel executions for each parameter configuration are shown in Fig. 4.4.
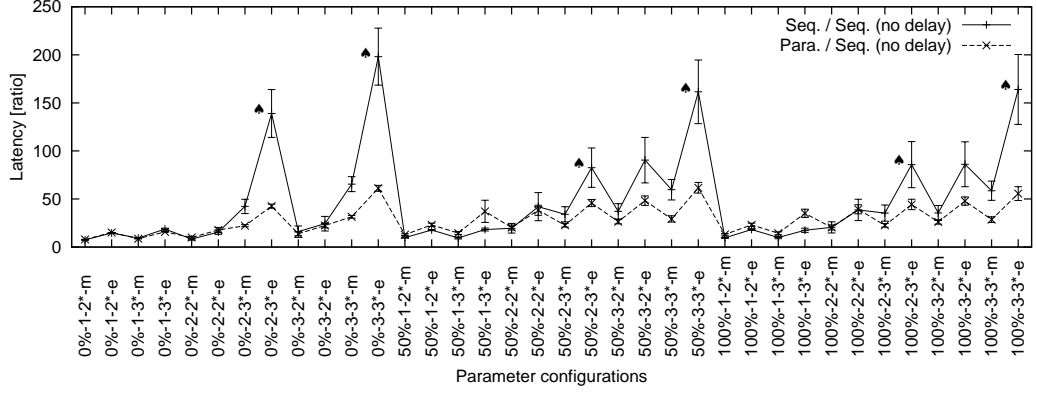
Figure 4.4: Average response times of sequential and parallel executions for individual parameter configurations

On the horizontal axis, each configuration is depicted in the form $x1$-$x2$-$x3$-$x4$, meaning that the values of *ed_mac*, *#d_rcv*, *#d_req*, and *ed_req* are $x1$, $x2$, $x3$, and $x4$, respectively. On the vertical axis, the average response times are measured in the ratio to the average response time of the sequential executions with *no delay* of the delivery of requests or the behavior of replicas.

We clearly observed that at configurations of *#d_rcv* = 3* and *ed_req* = *end*, (i.e., when the number of replicas receiving delayed requests is large and these requests arrive very late, the peaks are marked with ♠ in Fig. 4.4), the response time of the sequential executions is 150 or 200 times longer than the no delay case, and the efficiency becomes very low. On the other hand, the response time of the parallel executions is at most around 50 times longer than the no delay case. Especially, when the efficiency of the sequential executions is terrible, the good effect of parallel executions is remarkable for the following reason. Multiple replicas that receive many delayed requests cannot indirectly verify the validity of the requests received from other replicas until they receive them directly from clients. This greatly delays the termination of the involved agreement and shifts the following agreements afterward. However, in parallel executions, a new RSC can be started without waiting for termination of the agreement, and the delayed messages have no effect on the following agreements.

Although at configurations of 50%-1-3*-e or 100%-1-3*-e the efficiency of the parallel executions is worse than that of the sequential executions, the difference is small. This means that the overhead of additional MVC in parallel executions does not have much effect on the whole response time.

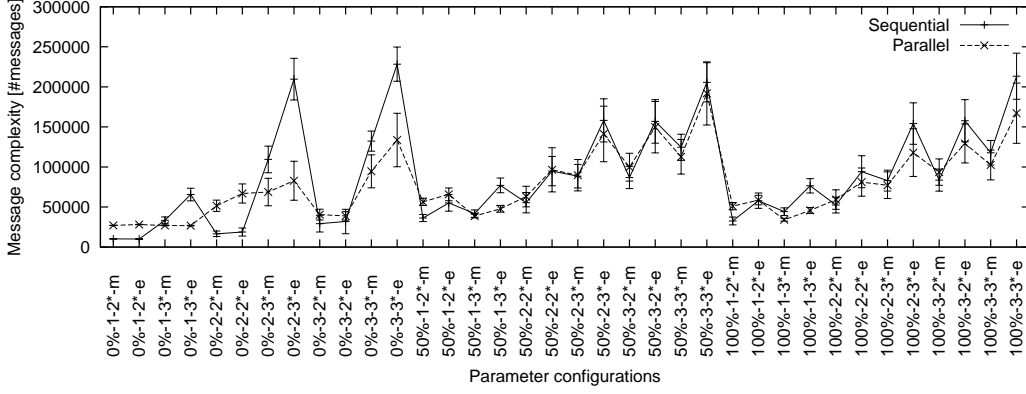Figure 4.5 shows the message complexity of sequential and parallel executions on each config-

Figure 4.5: Message complexity of sequential and parallel executions for individual parameter configurations

uration. The vertical axis of Fig. 4.5 indicates the average number of messages sent by a replica during an execution on each configuration. Since parallel executions run many RSC instances, their message complexity could naively be presumed to be high. However, the message complexities of the parallel executions and the sequential ones are actually almost the same. Especially, the parallel executions realize the lower latency with fewer messages than the sequential executions in the configurations 0%-2-3*-e and 0%-3-3*-e. The reason is considered that the RSC executions could reach agreements with small rounds thanks to the parallelization. On the other hand, it should be noted that lower latency does not always mean small message complexity, as seen in the configurations 50%-3-3*-e and 100%-3-3*-e.

Next, we focus on the effect of randomization of the RSC proposal. Fig. 4.6 shows the average response times of parallel executions with different probabilities employed in the *choose* function: 0.25, 0.5, and 1.0. Similarly to Fig. 4.4, the response times are plotted as the ratio to the average response time of the sequential executions with no delay. The case of probability 1.0 corresponds to the naive approach without randomization in RSC proposals. As we presumed, the response time is almost the same as the sequential executions, and no advantage of parallelization appears. On the other hand, probabilities 0.25 and 0.5 equally and positively affect parallelization, which proves the usefulness of our idea of randomization.
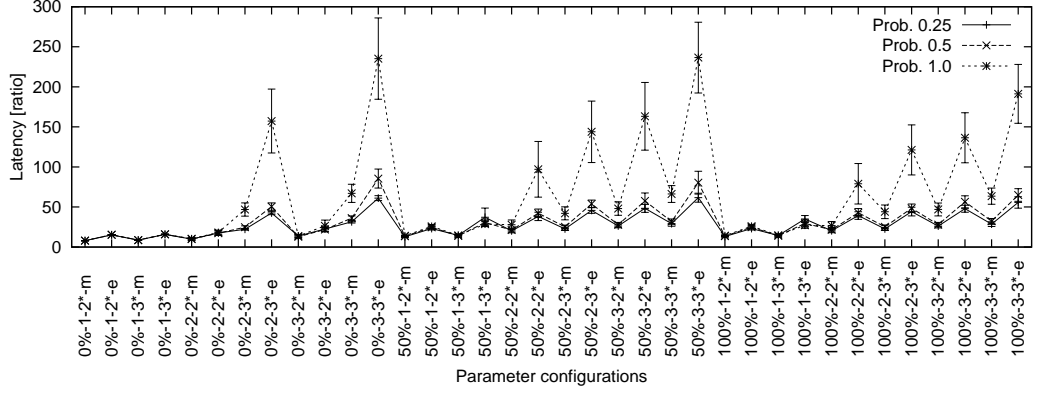
Figure 4.6: Average response times of parallel executions with probabilities: 0.25, 0.5 and 1.0

4.5.3 Scalability

Here, we conduct experiments to evaluate scalability, i.e., the ability to process a large number of requests, of the parallelizing method. In the experiments, we measure latencies and throughputs of the sequential and the parallelizing methods. The experiments are done in an environment where there is no delay on the delivery of requests or the behavior of replicas, because the delay does not have an essential effect on the performance in processing a large number of requests.

In Figs. 4.7 and 4.8, we show the throughput and the latency of the sequential and the parallel executions. Hereafter, we compare resource bounds of the sequential and the parallel executions, first based on their throughputs.

First, we explain how we evaluate the throughput because reasonable evaluation of throughput is a subtle problem at loads exceeding the resource bound of systems. To evaluate the throughput at a given request frequency (or at a given load of requests), we execute the protocol for 25 seconds at the load. Here, *request frequency* means the number of requests received by a replica every second. Then we divide the execution into five successive sections of five-second long intervals. For each section, we calculate a tentative throughput that is the average of processed requests per second. Finally, we choose the maximum value among the five tentative throughput values as the throughput value at the load. If the load does not exceed the resource bound, then the tentative throughput value increases and becomes stable. On the other hand, if it exceeds the resource bound, the value first increases and then decreases. Thus, we choose the maximum of the tentative values as the estimated throughput for both cases. The result for each request frequency listed below is an average value of ten executions.

In the throughput graph, the request frequency at which the throughput peaks corresponds to the load where the system reaches the resource bound. By our calculation, the angle of inclination after the peak shows how fast the resource will be exhausted after reaching the resource bound. A larger angle means faster exhaustion.

In Fig. 4.7, parallel executions are controlled by restricting the number of parallel agreements at a time, denoted by $\#para$. For example, $\#para = 2$ means that if two RSC are being executed in parallel and timing for a new RSC invocation is reached, the invocation is postponed until one of the executions is terminated. In Fig. 4.7, when the value of $\#para$ is large, the parallel execution reaches the resource bound with a smaller load. At loads before reaching the resource bound, parallel executions show the same throughput values as the sequential execution. At loads beyond the resource bound, parallel executions exhaust the resource more rapidly.

In Fig. 4.8, to improve the scalability of the parallel execution, we add another restriction on the frequency of the parallel executions of RSC, denoted by $freq$. For example, $\#para = 2$ and $freq = 5$ mean that if two RSCs are executed in parallel and one terminates, parallel RSC execution is not allowed until five newly invoked sequential executions of RSC have been completed. In Fig. 4.8, if we control the frequency of the parallel executions of RSC, the resource consumption is greatly reduced for $\#para = 2$.

In Figs. 4.7 and 4.8, the graphs of throughput and latency show the same characteristics of resource bound. When the executions reach the resource bound, the latency of each execution becomes high, and the throughput does low. However, the effect of resource consumption appears in lighter load for latency than throughput. Latency begins increasing before the execution reaches its resource bound.

From these observations, we conclude that parallel executions consume resources in proportion to the number of consensus protocol instances executed in parallel. When we restrict the number, the executions still exhaust the resources rapidly when the load exceeds the bound. The speed slows down when we restrict the frequency of RSC because time is required for parallel executions to release the resource. For the practical use of the parallelizing method, when the load is heavy, we should dynamically control the number of parallel executions and their frequency to avoid resource exhaustion.

4.6   Concluding Remarks

In this chapter, we proposed a method to accelerate state machine replication for Byzantine fault tolerance by parallelizing the executions of request set consensus and adding an extra multi-valued

consensus for deciding the processing order of agreed sets. We also show the correctness of the protocol for parallelizing agreements. Parallelism has a strong advantage in spite of requiring an additional agreement, especially when some replicas work slowly or some requests are delivered late. We showed this property by an experimental evaluation. In this evaluation, our parallelizing method accelerates the latency of replication three or four times more than the existing sequential method in delayed situations.

Clement et al. experimentally compared the performances in such delay situations among representative protocols based on rotating coordinator approach [35]. They showed that Castro-Liskov protocol [9], which is known to be practically very fast in the normal situation, degrades the performance in the delayed situations. For randomization approach on which this chapter focuses, Moniz et al. evaluated RITAS in the WAN environment where communication speeds between processes are not uniform [17]. However, the evaluation models were not so detailed as the one of this chapter. To the best of our knowledge, for randomization approach, this is the first one that evaluated in such detail the effect caused by the delay of message delivery and slow behavior of replicas.
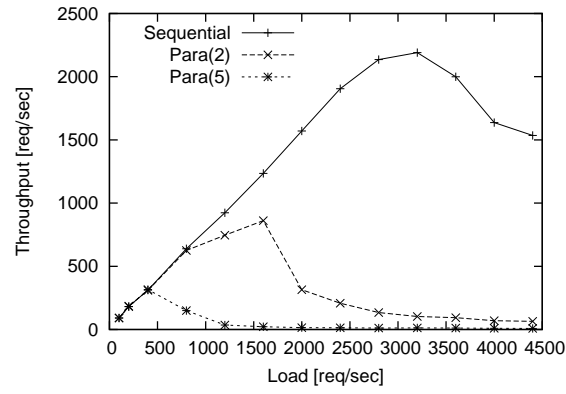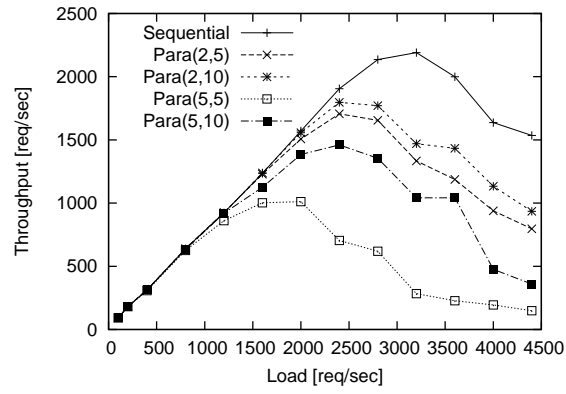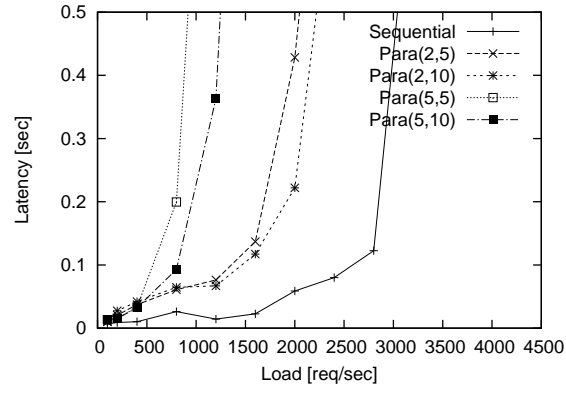
(a) Throughput
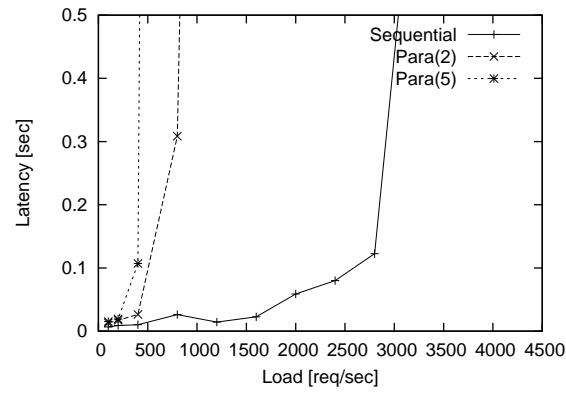
(b) Latency

Figure 4.7: Results with restriction on number of the parallel RSC executions. $Para(x)$ means "Parallel execution with $\#para = x$".

(a) Throughput

(b) Latency

Figure 4.8: Results with additional restriction on frequency of the parallel RSC executions. Para(x,y) means "Parallel execution with $\#para = x$ and $freq = y$".

Chapter 5

Conclusion

5.1   Summary of the Results

In this dissertation, we focused on Byzantine fault tolerant state machine replication, and proposed two methods that make the replication more efficient and can motivate system administrators to utilize the replication to improve reliability of server systems.

In Chapter 3, our new randomized Byzantine consensus protocol for a set of requests, RSC, was proposed. It was designed for an asynchronous distributed system like the Internet, and can be used to realize state machine replication. The protocol has two features. First, it directly solves the request set consensus problem, differently from existing protocols. Most existing protocols take a modular approach, in which replicas repeatedly solve a binary consensus problem to solve the request set consensus problem. In contrast, RSC can solve the problem without such repetition and can reach an agreement efficiently. Second, it has an efficient local coin tossing scheme. Although the local coin tossing approach was broadly known as inefficient, by exploiting the structure of the BFT replication, we can guarantee that the candidate values of the non-faulty replicas become identical with high probability. Thanks to the scheme, it can reach an agreement within a few rounds. Performance evaluation was conducted from two viewpoints, i.e., analytically and experimentally. The analytical evaluation showed that our protocol can reach an agreement within two rounds in some specific configurations, even if there are many replicas to tolerate Byzantine faults. In addition, it reaches the specific configurations within 10 rounds from initial configurations. Therefore, we could conclude that our protocol RSC has high scalability with respect to the number of replicas. The experimental performance was evaluated on a real distributed system composed of 10 physical machines and compared with that of two existing

protocols. In the experimental evaluation, our protocol achieved higher throughput and shorter latency than the existing ones, especially when the number of replicas is large.

In Chapter 4, we proposed another method that accelerates the Byzantine fault replication by parallelizing consensuses to determine processing orders of requests. Since the BFT replication is deployed on an asynchronous network, these concurrently executed consensuses possibly terminate in different orders at different replicas. If the replicas simply process the requests in the agreed sets in their terminated order, their replicated server states reach to distinct states and the replication collapses. Therefore, when a consensus execution terminates, our method invokes a multi-valued consensus protocol to determine which agreed set the replicas process next. However, there still existed a problem about this parallelization, i.e. its computation cost of multiple consensuses. To reduce the cost, we modified a procedure for creating an initial proposal for request set consensus. Its original procedure created a set of all the requests that had received but not been processed yet as the proposal. Since the size of the initial proposal affects both durations of the consensus and processing times of the consensus protocol's messages, we modified the procedure so that it randomly removes requests from the set. We proved that the parallelizing method satisfies correctness for the state machine replication, and evaluated its performance in comparison with an existing sequential method experimentally. The evaluation results showed that the parallelization has a strong advantage in spite of requiring an additional consensus, especially, when some replicas work slowly or some requests are delivered late.

5.2   Future Directions

As summarized above, we aimed to improve practicality of the Byzantine fault tolerant replication by proposing the two methods. However, many research themes that should be resolved still remain, and we briefly discuss them here.

Dynamic Change of a Replication's Configuration   When considering the practicality of the Byzantine fault tolerant replication, its replication cost is a serious problem. To tolerate f faulty replicas, Byzantine fault tolerant replication requires $3f + 1$ replicas in total, although the replication that can tolerate only crash failures does $2f + 1$ ones. The Byzantine replication needs 50% more replicas, and it is difficult to pay such cost always. Yin et al. proposed an algorithm that reduce the number of replicas that process requests from $3f + 1$ to $2f + 1$ [36], although its configuration is static and fixed. If a user can change a configuration of the replication dynamically during its execution, the user will be able to take a next scenario; When a threat of attacks is

expected to be relatively weak, the system administrator keeps the total number of the replicas small as possible as the user can. If the expected threat becomes strong, the user increases a scale of the replication and prepares for attacks. When the degree of the expected threat decreases, the total number of the replicas also decreases. Bortnikov et al. proposed a prototype of such system [37]. If such control is possible, the BFT replication will become more practical.

Support for Nondeterministic Applications   Byzantine fault tolerant replication protocols usually assume that a replication-target service has only deterministic operations and its state is determined only by the processing order of requests. However, in practical systems, there is nondeterminism caused by applications and/or operating systems, and the nondeterminism also should be supported by replication protocols. For leader-based replication approach, Zhao introduced a classification of common types of nondeterminism present in many applications and proposed mechanisms needed to handle these types [38]. On other hand, for randomization-based approach, more work is demanded.

Focus on Virtual Machine and Cloud Infrastructure   Recently, virtual machines and cloud infrastructures appear as a new execution environment of distributed algorithms. These environments have new properties that traditional environments, e.g. physical machines connected with LAN, do not have, and it is expected to realize more efficient replication by exploiting the properties.

First, let us consider an environment composed of a single physical machine equipping some virtual machine monitor (also known as hypervisor) and multiple virtual machines are deployed and running in the physical machine. Chun et al. investigated such environment, and, based on the leader-based approach, they discussed what software/hardware functions will be important [39]. They also showed two new lightweight BFT protocols exploiting hardwares shared by all replicas on the virtual machines. In the randomization approach, which we focus on the dissertation, there also exist many opportunities to improve its efficiency. The replicas can share many devices, e.g. bus, clock, random number generator, and so on. For example, shared coin tossing is known to be slow due to cryptographic schemes it uses. Although the scheme is necessary to share secrets without revealing them by attackers, in the above virtual machine environments, the cryptographic scheme will become lighter, since malicious replicas must not be able to impersonate other replicas and steal messages sent between other replicas.

Second, let us discuss about cloud infrastructures. In the cloud infrastructure, many virtual machines are deployed in regions. A region is composed of physical machines connected by LAN.

Regions are also connected by WAN. A virtual machine can migrate from a physical machine to another one, possibly over two regions because of some reasons, e.g. stabilizing performance of physical machines. This research area is novel, and there are only a few results [40, 41, 42]. One of the features of the cloud infrastructures is extensive variety of quality of communications. Assume that two replicas on two virtual machines want to communicate with each other. If the two virtual machines are deployed on the same physical machine, they can communicate very efficiently. Even if the two virtual machines are deployed on two distinct physical machines on the same region, the communication is still efficient. However, if the replicas communicate over their regions, the communication is slow. Moreover, in typical cloud infrastructures such as Amazon AWS [43] or Google Compute Engine [44], data transmission to WAN is charged based on the amount of transferred data. Therefore, new distributed algorithms designed for the cloud infrastructures are demanded. For instance, it is useful if there exists an algorithm that collects statistical information about data transfer and autonomously migrates replicas to optimize its distributed system for some predefined goal, e.g., minimizing data transfer charge, or maximizing performance of the replication. A prototype of another optimization to the cloud infrastructure was proposed by Amir et al. [45]. Their algorithm constructs a layered structure of the replication and realizes the BFT replication on WAN. In the algorithm, since communications over regions are costly, replicas in the same region agree with the contents of messages to other regions, and only a leader of the region communicates with the other regions. Although the algorithm was not specialized for the cloud infrastructure, such idea of the layered structure will be useful to consider algorithms fitted with the cloud.

Standard Framework for BFT replication   Standard framework for Byzantine fault tolerant state machine replication is needed. Although there were many implementations of protocols for the BFT replication, most of them were used once for the performance evaluation of the research papers and never used again. If there exists a standard framework for the replication that defines programming interfaces between a replication protocol and a replication-targeted service, how to retrieve information of a running distributed system, and so on, we could re-use them easily. Moreover, this enables easy performance tuning of a user's replication system; the user just tries many prepared replication protocols and chooses the best one. Although there exists some attempts to build a framework/middleware for Byzantine fault tolerance [46, 47], their main purpose was to make building a replicated service easy, and not interested in re-usability of the implementations of algorithms. By combining with virtual machine management tools like Chef [48] or Docker [49], automated deployment and management of replicas will be possible. There

are many advantages for the existence of the standard framework as see above, and thus, many discussion about the standard framework should be done.

Acknowledgements

I could not finish this dissertation if I were not encouraged by my best friends. It is priceless and precious experience for me to meet and share the same time with the friends. Especially, I wish to express my deep gratitude to Taro Nakazawa, Teru Ito, and Yonghwan Kim.

Finally, I strongly appreciate my parents, Dr. Kazuo Nakamura and Miyuki Nakamura, and all of my family for their supports and kindness during my life.

Bibliography

[1] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2006.

[2] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[4] 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, 1982.

[5] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, PODC '83, pp. 27–30, ACM Press, 1983.

[6] M. O. Rabin, "Randomized byzantine generals," in *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, SFCS '83, pp. 403–409, IEEE Computer Society, 1983.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[8] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[9] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pp. 173–186, USENIX Association, 1999.

[10] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pp. 45–58, ACM, 2007.

[12] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Byzantine fault detectors for solving consensus," *The Computer Journal*, vol. 46, no. 1, pp. 16–35, 2003.

[13] F. Pedone and A. Schiper, "Optimistic atomic broadcast," in *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, pp. 318–332, Springer-Verlag, 1998.

[14] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Thrifty generic broadcast," in *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pp. 268–282, Springer-Verlag, 2000.

[15] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 41st International Conference on Dependable Systems Networks*, DSN '11, pp. 245–256, 2011.

[16] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pp. 524–541, Springer-Verlag, 2001.

[17] H. Moniz, N. F. Neves, M. Correia, and P. Verissimo, "Ritas: Services for randomized intrusion tolerance," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 1, pp. 122–136, 2011.

[18] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo, "Randomized intrusion-tolerant asynchronous services," in *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pp. 568–577, 2006.

[19] M. Correia, N. F. Neves, and P. Verissimo, "From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures," *The Computer Journal*, vol. 49, no. 1, pp. 82–96, 2006.

[20] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pp. 59–74, ACM, 2005.

[21] Y. J. Song and R. Renesse, "Bosco: One-step byzantine asynchronous consensus," in *Proceedings of the 22nd International Symposium on Distributed Computing*, DISC '08, pp. 438–450, Springer-Verlag, 2008.

[22] C. Cachin and J. A. Poritz, "Secure intrusion-tolerant replication on the internet," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pp. 167–176, IEEE Computer Society, 2002.

[23] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, "Randomized multivalued consensus," in *In Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '01, pp. 195–200, IEEE, 2001.

[24] G. Bracha, "An asynchronous [(n - 1)/3]-resilient consensus protocol," in *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pp. 154–162, ACM Press, 1984.

[25] F. Borran, M. Hutle, and A. Schiper, "Timing analysis of leader-based and decentralized byzantine consensus algorithms," *Journal of the Brazilian Computer Society*, vol. 18, no. 1, pp. 29–42, 2012.

[26] 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 7th symposium on Operating systems design and implementation*, pp. 177–190, USENIX Association, 2006.

[27] G. Bracha and S. Toueg, "Resilient consensus protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, PODC '83, pp. 12–26, ACM Press, 1983.

[28] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2004.

[29] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[30] H. Moniz, N. F. Neves, M. Correia, and P. Verissimo, "Experimental comparison of local and shared coin randomized consensus protocols," in *Proceedings of the 27th IEEE Symposium on Reliable Distributed Systems*, SRDS '06, 2006.

[31] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," tech. rep., Cornell University, 1994.

[32] J. Nakamura, T. Araragi, and S. Masuyama, "Asynchronous byzantine request-set agreement algorithm for replication," in *Proceedings of the 1st AAAC Annual Meeting*, p. 35, 2008.

[33] J. Nakamura, T. Araragi, S. Masuyama, and T. Masuzawa, "Efficient randomized byzantine fault-tolerant replication based on special valued coin tossing," *IEICE Transactions on Information and Systems*, vol. E97-D, no. 2, 2014. (to appear).

[34] L. Lamport, "Byzantizing paxos by refinement," in *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pp. 211–224, Springer-Verlag, 2011.

[35] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pp. 153–168, USENIX Association, 2009.

[36] 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 19th ACM symposium on Operating systems principles*, SOSP '03, pp. 253–267, 2003.

[37] V. Bortnikov, G. Chockler, D. Perelman, A. Roytman, S. Shachor, and I. Shnayderman, "Frappé: Fast replication platform for elastic services," in *Proceedings of the 5th Workshop on Large Scale Distributed Systems and Middleware*, 2011.

[38] W. Zhao, "Byzantine fault tolerance for nondeterministic applications," in *Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, DASC '07, pp. 108–118, IEEE Computer Society, 2007.

[39] B.-G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance," in *Proceedings of USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pp. 287–292, USENIX Association, 2008.

[40] Y. Zhang, Z. Zheng, and M. R. Lyu, "Bftcloud: A byzantine fault tolerance framework for voluntary-resource cloud computing," in *Proceedings of the 4th International Conference on Cloud Computing*, CLOUD '11, pp. 444–451, IEEE Computer Society, 2011.

[41] P. Garraghan, P. Townend, J. Xu, X. Yang, and P. Sui, "Using byzantine fault-tolerance to improve dependability in federated cloud computing," *International Journal of Software and Informatics*, vol. 7, no. 2, pp. 221–237, 2013.

[42] H. Liu, H. Jin, X. Liao, C. Yu, and C.-Z. Xu, "Live virtual machine migration via asynchronous replication and state synchronization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 12, pp. 1986–1999, 2011.

[43] Amazon Web Services, Inc., "Amazon Web Services, Cloud Computing: Compute, Storage, Database." `http://aws.amazon.com`, 2013. [Online; accessed December 11, 2013].

[44] Google Inc., "Home Google Cloud Platform." `https://cloud.google.com`, 2013. [Online; accessed December 11, 2013].

[45] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2010.

[46] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware forweb-service applications," in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, pp. 131–142, IEEE Computer Society, 2005.

[47] W. Zhao, "Design and implementation of a byzantine fault tolerance framework for web services," *Journal of Systems and Software*, vol. 82, no. 6, pp. 1004 – 1015, 2009.

[48] Chef, "Chef — Chef." `http://www.getchef.com/chef/`, 2013. [Online; accessed December 11, 2013].

[49] Docker Inc., "Homepage - Docker: the Linux container engine." `https://www.docker.io/`, 2013. [Online; accessed December 11, 2013].