Efficient Randomized Byzantine Fault-Tolerant Replication Based on Special Valued Coin Tossing

Junya NAKAMURA†, Nonmember, Tadashi ARARAGI††, Shigeru MASUYAMA†††, and Toshimitsu MASUZAWA†, Members

SUMMARY We propose a fast and resource-efficient agreement protocol on a request set, which is used to realize Byzantine fault tolerant server 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. This coin toss allows our protocol to reduce resource consumption and to attain faster response time than the existing representative protocols.

key words: Byzantine fault tolerance, state machine replication, Byzantine agreement, consensus, asynchronous distributed system

1. Introduction

Byzantine failure is the most malicious failure, in which a faulty system behaves in an arbitrary way deviating from the original program, including stopping failure. 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 Byzantine fault tolerance (BFT) in such asynchronous networks as the Internet is state machine replication [1]. In the replication, multiple replicas of a server system individually run on distinct hosts. Here, the system’s internal state is supposed to be determined by the sequence of requests applied to the system. In state machine replication, client systems multicast requests to all replicas. 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 requests differ among replicas, the replicas behave identically. In the setting of state machine replication for Byzantine failure, we assume that only a few replicas can be taken over by crackers and behave differently from the non-faulty replicas. However, 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 seen above, agreements are repeated among replicas many times to continue the replicated service. On the other hand, the protocols for Byzantine agreement are very costly. Therefore, the replication method has long been considered non-practical. However, Castro and Liskov presented a very efficient BFT method based on the replication under some assumptions [2]. This result stimulated the development of efficient BFT protocols.

In this paper, we propose another new efficient BFT protocol for state machine replication. Our protocol is based on randomization, and can be employed in more general settings without Castro and Liskov’s assumptions such as weak asynchrony and non-faulty clients. Compared with other randomized protocols for BFT, it is efficient enough and much more robust against congested communication. We explain our protocol’s fundamental idea and its characteristics in more detail after discussing related work.

Related work: As stated above, Byzantine agreement is a key tool to achieve BFT state machine replication. However, the termination of an agreement cannot be guaranteed by deterministic protocols in asynchronous distributed systems like the Internet, even if there is only one crash process [3]. There are two approaches to overcome this impossibility:

1. Ensure the termination deterministically under some assumptions of communication delay called weak asynchrony assumptions (Byzantine failure: [2], [4]–[6], crash failure: [7]–[9]).
2. Ensure the termination with probability 1 by using randomized methods (Byzantine failure: [10]–[12], crash failure: [13]).

In the first approach, a special replica called a primary or a rotating coordinator controls the protocol to efficiently achieve an agreement on the processing order of requests so that all non-faulty replicas can process the same requests in the same order. A representative agreement protocol for this approach is the Practical BFT protocol proposed by Castro and Liskov [2]. In the second approach, an agreement is achieved mainly by repeated coin tosses. For efficiency, the
agreed value is 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), and 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). ABC[10] and RITAS[11],[12] are representative protocols of the second approach. Our agreement protocol follows the second approach. Below, we explain the characteristics of these protocols and compare them with ours.

Castro and Liskov developed a practical BFT protocol[2] in the first approach and denoted it as PBFT. In the PBFT protocol, one special replica, called a primary, decides the order of processing the requests and broadcasts it to the other replicas. PBFT normally achieves an agreement very fast and provides a practical solution for state machine replication. When an agreement is delayed and many non-faulty replicas suspect that the current primary is faulty, the primary’s role is moved to another replica based on their agreement, which is why it is called a rotating coordinator. PBFT guarantees the termination of agreements, even if there are continuous rotations of a primary coordinator, assuming that the message delay has an asymptotic 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 is very costly.

ABC[10] proposed by Cachin et al., and RITAS[11],[12] proposed by Correia et al., take the second approach. Their protocols realize atomic broadcast. 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 \{0,1,...,v\}. 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[14], and RITAS uses Bracha’s binary Byzantine agreement protocol[15]. ABBA employs shared coin tossing[16] 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.

Our contribution: In this paper, we present a new randomized multi-valued Byzantine agreement protocol. Although we employ Bracha’s agreement protocol[15] as a framework of our protocol, we 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 and RITAS, and much less resource-consuming than these two protocols.

Multi-valued Byzantine agreement with special valued coin tossing: A naive multi-valued Byzantine agreement based on local coin tosses[17] 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 Sect. 6.2.1. 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 Sect. 6.2.2.

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 Sect. 6.3, 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.

Organization: The remainder of this paper is structured as follows. In Sect. 2, we describe our distributed system and replication models, and illustrate an approach to realize replication using an agreement protocol for a request set. In Sect. 3, we formally define a problem called Request Set Consensus, which is our agreement problem for request sets. The main protocol RSC is presented in Sect. 4 and its correctness is proven in Sect. 5. The RSC performance is shown in Sect. 6, and then we conclude in Sect. 7.

2. Model

Here we describe our system model and state machine replication and how the replication is realized by a protocol that solves 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 process, 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) are called faulty.

2.2 State Machine Replication

A state machine, which consists of a set of states and a set of commands, executes a command to change its state. The next state of a state machine is determined by an executed command and its current state.

In state machine replication [1], a server is a state machine and is replicated with replicas that independently run as state machines on different hosts. A client submits a request to all the replicas to request the server to execute commands. The asynchronous system allows request messages to arrive at different replicas in different orders. However, to keep the replicas in the same state, they must process the requests in the same order. Replicas repeatedly execute an agreement protocol to determine a common order of processing requests among replicas. After a replica processes a request, it replies the result to the client, and the client accepts the result after receiving \( f + 1 \) identical results from different replicas, where \( f \) is the maximal number of replicas that may fail. The client can confirm that the result was submitted by at least one non-faulty replica, if it collects \( f + 1 \) identical results. (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 [2]).

A protocol that realizes state machine replication must satisfy the following two requirements:

**Safety**: All non-faulty replicas process the same requests submitted by clients in the same order.

**Liveness**: A client eventually accepts the result of its submitted request.

Since \( n \) must be greater than or equal to \( 3f + 1 \) to realize state machine replication [18], we assume \( f \leq \lceil (n - 1)/3 \rceil \).

2.3 Request Set Consensus Approach for State Machine Replication

In this section, we explain how state machine replication is realized by a protocol that solves the request set consensus problem. Hereafter, we will formalize the requirements for the request set consensus problem in Sect. 3 and will propose our new efficient agreement protocol RSC that satisfies the requirements in Sect. 4.

As previously explained in Sect. 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, they can be delivered in different orders among replicas. For example, Replica 1 receives request \( r_5 \) from client A first and then request \( r_6 \) from client B, but Replica 2 may receive request \( r_6 \) from client B first and then \( r_5 \). 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.

**Step 3**: Return to Step 1.

This repetition is continued until the service is terminated.

Figure 1 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. 1 shows the \( i \)-th and \((i + 1)\)-th agreements. The distributed system in Fig. 1 has three clients and four replicas where Replica 4 is faulty. The clients issued five requests: \( r_1, r_2, \ldots, r_5 \). Before starting the \( i \)-th agreement, every replica places the set of unprocessed requests it has received...
to its initial value for the $i\text{-th}$ 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 $i\text{-th}$ 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. 1, the $i\text{-th}$ agreement returns agreed value $\{r_2, r_3\}$. 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 $i\text{-th}$ agreement, it starts the $(i + 1)\text{-th}$ agreement. Each replica’s initial value of the $(i + 1)\text{-th}$ 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 $j\text{-th}$ agreement ($j < i + 1$). For example, Replica 2 sets $\{r_2, r_3, r_5\}$ to the $i\text{-th}$ initial value, and $r_2$ and $r_5$ are included in the $i\text{-th}$ agreed set, and then $r_4$ is again included in the initial value of the $(i + 1)\text{-th}$ agreement. On the other hand, $r_3$ is received by Replica 2 during the $i\text{-th}$ agreement protocol execution and is also included in the $(i + 1)\text{-th}$ initial value. In Fig. 1, Replica 4, a faulty replica, is disturbing the agreement by proposing an invalid initial value for the $(i + 1)\text{-th}$ agreement. Replica 4 includes $r_2$ in the initial value, which was already included in the $i\text{-th}$ 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 $(i + 1)\text{-th}$ 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.

3. 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 $i\text{-th}$ execution of the request set consensus protocol for replica $p$ be $I'_p$, the $i\text{-th}$ execution of the request set consensus protocol by $p$ be $RS C(p, i, I'_p)$ and its agreed value be $V'_p$. Note that $I'_p$ and $V'_p$ are sets of requests. Let $R$ be the set of all the requests submitted by clients and $M'_p(\subseteq R)$ be the set of requests that process $p$ has received before starting the $i\text{-th}$ request set consensus execution. We assume that every replica eventually receives all requests in $R$ and $I'_p = M'_p - \{V'_p \cup V'_p \cup \cdots \cup V'_{p-1}\}$. If there is no need to distinguish replicas, we simply write $V'_p$ 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 $RS C(p, i, I'_p)$ and $RS C(q, i, I'_q)$ terminate, then $V'_p = V'_q$. **Termination:** For any $i$ and any non-faulty replica $p$, $RS C(p, i, I'_p)$ terminates with probability 1. **Integrity:** For any $i$ and any non-faulty replica $p$, $V'_p \subseteq R$. **Validity:** For any $r \in R$ and any non-faulty replica $p$, there exists $i$ such that $r \in V'_p$.

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 paper 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 $p$, $V'_p \subseteq \bigcup_{q \text{non-faulty}} I'_q$. **Usual Validity:** For any $i$ and any non-faulty replica $p$, $\bigcap_{q \text{non-faulty}} I'_q \subseteq V'_p$.

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.

4. 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[15]. 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 $f \leq \lceil(n - 1)/3\rceil$, i.e., $n \geq 3f + 1$. Since asynchronous Byzantine agreement cannot be solved deterministically even if $f = 1$[3], our protocol uses randomized coin tossing, like Bracha’s protocol.

We borrow two techniques, which we explain in Sect. 4.1 and Sect. 4.2.1, from Bracha’s original protocol: reliable broadcast and internal message validity check. We introduce another message validity check for RSC in Sect. 4.2.2. The details of our protocol are shown in Sect. 4.3. In Sect. 4.4, we describe our characteristic coin tossing scheme that plays a key role in RSC.

4.1 Reliable Broadcast

Reliable Broadcast[10], [15], [19] 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 \( \text{R-Broadcast}(m) \).

4.2 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 [15], 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.

4.2.1 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. 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 [15].

4.2.2 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.

4.3 Protocol

Figure 2 shows 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 \(\text{initial candidate} f_p\). The execution ID is a natural sequence number that starts from 1 and increases by one after each agreement, and RSC outputs an agreed set of requests denoted by \(V^*\).

The control variables for the protocol are \(rn\), \(\text{Cand}\), \(RI\), and \(\text{isMajor}\). A current round number is represented by \(rn\). \(\text{Cand}\) represents a tentative candidate for an agreed value. The value of \(\text{Cand}\), 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 \(I_p\) and finally becomes a common value \(\bigcup_{p} I_p\), where \(q\) ranges over the replicas that have issued an \(\text{INITIAL}\) 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 \(\text{isMajor}\) indicates whether a majority exists in the received candidates at Step 2. An \(\text{isMajor}\) value is broadcast at Step 3 with \(\text{Cand}\), and an update process of \(\text{Cand}\) at Step 3 branches based on how many received messages include \(\text{isMajor}\) as \(\text{true}\).

During the RSC execution, four kinds of messages are exchanged. An \(\text{INITIAL}\) message announces its own \(I_p\) to the other replicas. \(MSG1\), \(MSG2\), and \(MSG3\) are used to inform other replicas about \(\text{Cand}\) at Steps 1, 2, and 3, respectively. In addition, a \(MSG3\) message contains a flag \(\text{isMajor}\). All of these messages are sent by reliable broadcast, which is explained in Sect.4.1. Thus, even a faulty replica cannot send different values to different replicas in the broadcast.

**Step 0**: Reliably broadcast an \(\text{INITIAL}\) message to announce its own initial candidate \(I_p\) 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 \(\text{Cand}\) 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 \(\text{Cand}\) and \(RI\) are extended by adding the newly known unforged requests that are commonly included by at least \(f + 1\) \(\text{Cand}\) values in the messages.

**Step 2**: Reliably broadcast a \(MSG2\) message containing its own \(\text{Cand}\) 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 \((n + 1)/2\) or more \(\text{Cand}\) values have an identical value, replace the value of its own \(\text{Cand}\) with the common one, and set its own \(\text{isMajor}\) to \(\text{true}\).

**Step 3**: Reliably broadcast a \(MSG3\) message that has agreement candidate \(\text{Cand}\) and the flag of majority \(\text{isMajor}\), 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 \(\text{isMajor}\) is \(\text{true}\), then decide the agreed value to be the \(\text{Cand}\) of these messages (lines 35–38). It is proven that for any two messages whose \(\text{isMajor}\) values are \(\text{true}\), their \(\text{Cand}\) 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 \(\text{isMajor}\) values are \(\text{true}\), then replace its own \(\text{Cand}\) with the \(\text{Cand}\) 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 \(\text{Cand}\) values received at Step 3 and \(RI\). \(RI\) is extended by adding the union of the initial candidates included in the \(\text{INITIAL}\) messages received so far. Here, the external validity of these received \(\text{INITIAL}\) messages is checked, which is explained in Sect. 4.2.2, to exclude forged requests. Indirectly, \(\text{Cand}\) values are also checked for any forged requests by seeing that they are included in \(RI\). Update its \(\text{Cand}\) to the coin toss result. Since the values of the non-faulty replica’s \(RI\) eventually coincide with each other, termination is guaranteed.

### 4.4 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 \(\text{Cand}\) 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 broadcasts 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 $I_p^r$ 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.

5. Correctness

In this section, we prove that the RSC protocol satisfies the requirements for the request set consensus problem shown in Sect. 3: 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 $V$. □

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

**Proof.** The main idea of the proof is identical to that for Bracha’s binary Byzantine agreement protocol [15].

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 ($MSG3$, $i$, $r$, $V_p^r$, true) and ($MSG3$, $i$, $r$, $V_q^r$, true) at Step 3 of round $r$, respectively. This means that the two non-faulty replicas have to accept at least $[(n + 1)/2]$ ($MSG2$, $i$, $r$, $V_p^r$) messages and at least $[(n + 1)/2]$ ($MSG2$, $i$, $r$, $V_q^r$) 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, $V_p^r = V_q^r$. 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 $V_p^r$ to their Cand values at Step 3 of round $r$ (line 40), and the replicas decide $V_p^r$ at the next round $r + 1$ by Lemma 1. □

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

**Proof.** From Lemma 1 and the fact that SCV-full configuration is eventually reached as stated in Sect. 4.4, 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 $p$, $V_p^r \subseteq R$.

**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 $[(n + 1)/2]$ ($\geq f + 1$) $MSG2$ 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
(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 \( r \in V_i^p \).
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 \( p, r \in I_i^p \). 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 includes \( r \). With this observation that
the \( Cand \) value always includes \( r \) through the execution, we
conclude that \( V_i^p \) includes \( r \), which contradicts the assumption.

6. Performance Evaluation

6.1 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 Sect. 4.4)
in Sect. 6.2.1 and Sect. 6.2.2 and how fast a system moves
from a non-SCV-full configuration to a SCV-full one in
Sect. 6.2.3. 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 [12] and ABC [10] in Sect. 6.3. We implement
request set consensus protocols by using RITAS and ABC,
which are atomic broadcast protocols, in a straightforward
way described in Sect. 6.4, 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 enormous numbers 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 pro-
viding against these faults is always charged, even for the
case without faults.

6.2 Performance Evaluation on RSC Protocol Features

6.2.1 Evaluation in SCV-full Configuration

Evaluation environment and settings: The initial values
of the RSC protocol for individual replicas, i.e., sets of re-
quests, are set to random values so that achieving an agree-
ment 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 agree-
ment 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 agree-
ment is set to be uniformly random. We evaluate the number
of rounds for termination for the values of \( \alpha \), 0.1, 0.5, and
0.9, varying the number of replicas \( n \) from 4 to 22. Here,
\( \alpha \) is the probability of choosing the special coin value \( RI \)
in the coin tossing of the RSC protocol (line 47 of Fig. 2). For
each case, we executed the evaluation 1,000 times, and we
plot the average values in the graph of Fig. 3.

Result and discussion: Fig. 3 shows the increasing shape
in the number of replicas. When \( \alpha \) 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 \( \alpha \) is still reason-
ably small, while a naive estimation shows it needs \( 1/\alpha^{n-f} \)
rounds. With this observation, it is found that in the fault-
free executions, large \( \alpha \) gives much better performance. On
the other hand, in the case of faulty replica’s attacking, large \( \alpha \) can be exploited by an attacker to delay the termination. Therefore, when we suspect such an attack, we should dynamically decrease the value of \( \alpha \). Thus, we are still interested in small \( \alpha \). In the following, we analyze the reason of this remarkable point.

6.2.2 Analysis of the Performance in SCV-full Configuration with Small \( \alpha \)

First, we intuitively explain the reason of fast agreement for small \( \alpha \). We call a replica whose \textit{Cand} value is the common-value defined in Sect. 4.4 the \textit{candidate-full}. When \( \alpha \) 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 \textit{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 \textit{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 \( \text{isMajor} = \text{true} \), it replaces its own candidate with the value.
2. when it does not receive more than \( f \) candidates with \( \text{isMajor} = \text{true} \), 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:

\[
P(x, y, x', y') = \sum_{i=\max(y, x'-x-y)}^{\min(y, x')} \binom{x}{i} \binom{x-y}{x'-i} / \binom{x}{x'}.
\]

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

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

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

\[
p_1 = P(n, k, n - f, f + 1) - P(n, k_2, n - f, 2f + 1),
\]

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

\[
p_{c} = \alpha \cdot k_2 / n.
\]

Thus, the expected number after Step 3 is

\[
k_3 = n \cdot (p_1/(p_1 + p_2) + p_2 \cdot p_3/(p_1 + p_2)).
\]

**Result and discussion:** Next, we calculate how the ratio of candidate-full replicas changes based on the above equations, and Fig. 4 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.

6.2.3 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 $\alpha$ 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 $RI$.

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 Sect. 6.2.1. 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. 5 by plotting the averages of 1,000 executions for each case. A large $\alpha$ brings a rapid transition to the SCV-full configuration. Up to $n = 22$, the number of rounds is practically bounded.

6.3 Experimental Comparison with Other Protocols

**Protocols to be compared:** We compare our RSC protocol with two representative atomic broadcast protocols: ABC[10] and RITAS[12], 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 Sect. 6.4. We use it for this experimental comparison.

RITAS internally executes Bracha’s binary Byzantine agreement protocol[15] 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[14] $O(1)$ 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†. 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 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

| Table 1 | Ways of sending requests in two models for $n = 4$. |
|---------|-----------------------------------------------|
|         | 1st send 2nd 3rd 4th 5th 6th ...             |
| Replica 1 | $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$ ...   |
| Replica 2 | $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$ ...   |
| Replica 3 | $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$ ...   |
| Replica 4 | $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$ ...   |
| (a) Normal model | |
| Replica 1 | $r_1$ $r_2$ $r_3$ $r_4$ $r_5$ $r_6$ ...   |
| Replica 2 | $r_2$ $r_3$ $r_4$ $r_1$ $r_6$ $r_5$ ...   |
| Replica 3 | $r_3$ $r_4$ $r_1$ $r_2$ $r_5$ $r_6$ ...   |
| Replica 4 | $r_4$ $r_1$ $r_2$ $r_3$ $r_6$ $r_5$ ...   |
| (b) Delayed model | |

†http://sites.google.com/site/hmoniz/publications/ritas.zip
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 $\alpha$ to choose the special coin value $R_I$ is set to 0.9.

**Evaluation results:** Figs. 6 and 7 show the results of evaluating the throughput and the latency of RSC, RITAS, and ABC in the normal model, respectively. Similarly, Figs. 8 and 9 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. 10 and 11 show the averages of the number of rounds, the duration of an 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 $n = 4$.

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 re-
quest 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. 6 and 7, 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. 10. 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. 7 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. 8 and 9 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. Figure 11 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. 10 and 11 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.

6.4 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 [10]. This protocol is used in Sect. 6.3 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 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. 12). 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. Figure 13 shows the protocol details.

7. Conclusion

In this paper, 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.

As a future work, we should extend our replication model. In this paper, we assume that the server is a state machine. That is, behavior of a server is deterministic when the processing order of received requests is fixed. On the other hand, in a real system, there is nondeterminism caused by applications or operating systems. We should extend our consensus to cover this nondeterminism.

Acknowledgements

The authors would like to thank Profs. Kakugawa and Ooshita of Osaka University for their helpful discussions about improving this paper.

References

[1] F.B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” ACM Comput. Surv., vol.22, no.4, pp.299–319, 1990.

[2] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” OSDI ’99: Proc. Third Symposium on Operating Systems Design and Implementation, pp.173–186, Berkeley, CA, USA, 1999.
[3] 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.
[4] J. Martin and L. Alvisi, “Fast byzantine consensus,” Proc. International Conference on Dependable Systems and Networks, 2005. DSN 2005, pp.402–411, 2005.
[5] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative byzantine fault tolerance,” SOSP ’07: Proc. Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp.45–58, 2007.
[6] K. Kilhøstam, L. Moser, and P. Melliar-Smith, “Byzantine fault detectors for solving consensus,” The Computer Journal, vol.46, no.1, pp.16–35, 2003.
[7] T.D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” J. ACM, vol.43, no.2, pp.225–267, 1996.
[8] F. Pedone and A. Schiper, “Optimistic atomic broadcast,” Distributed Computing, pp.318–332, 1998.
[9] M. Kawazoe Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, “Thrifty generic broadcast,” Distributed Computing, pp.69–83, 2000.
[10] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, “Secure and efficient asynchronous broadcast protocols,” Lect. Notes Comput. Sci., vol.2139, pp.524–541, 2001.
[11] H. Moniz, N.F. Neves, M. Correia, and P. Verissimo, “Randomized intrusion-tolerant asynchronous services,” DSN ’06: Proc. International Conference on Dependable Systems and Networks, pp.568–577, 2006.
[12] 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.
[13] P. Erhilchelevan, A. Mostefaoui, and M. Raynal, “Randomized multi-valued consensus,” Proc. Fourth IEEE International Symposium on, Object-Oriented Real-Time Distributed Computing, 2001. ISORC-2001, pp.195–200, 2001.
[14] C. Cachin, K. Kursawe, and V. Shoup, “Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography,” PODC ’00: Proc. Nineteenth Annual ACM Symposium on Principles of Distributed Computing, pp.123–132, 2000.
[15] G. Bracha, “An asynchronous (n−1)/3-resilient consensus protocol,” PODC ’84: Proc. Third Annual ACM Symposium on Principles of Distributed Computing, pp.154–162, 1984.
[16] M.O. Rabin, “Randomized byzantine generals,” Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS), pp.403–409, 1983.
[17] M. Ben-Or, “Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols,” PODC ’83: Proc. Second Annual ACM Symposium on Principles of Distributed Computing, pp.27–30, 1983.
[18] G. Bracha and S. Toueg, “Asynchronous consensus and broadcast protocols,” J. ACM, vol.32, no.4, pp.824–840, 1985.
[19] V. Hadzilacos and S. Toueg, “A modular approach to fault-tolerant broadcasts and related problems,” Tech. Rep., Cornell University, Ithaca, NY, USA, 1994.

Shigeru Masuyama is a Professor at the Department of Computer Science and Engineering, Toyohashi University of Technology. He received the B.E., M.E. and D.E. degrees in Mathematics from Tokyo University, in 1985 and 1987, respectively, and D.E. degree in Informatics from Kyoto University in 2006. He joined Communications and Information Processing Labs. of NTT, Yokosuka, Japan in 1987. His research interest includes formal verification, analysis of security protocols and fault tolerance of distributed systems. Dr. Masuyama is a member of IEEE Society.

Junya Nakamura received B.E. and M.E. degrees from the Department of Knowledge-Based Information Engineering, Toyohashi University of Technology, Japan, in 2006 and 2008 respectively. He is currently a Ph.D student at Graduate School of Information Science and Technology, Osaka University. His research interests include distributed algorithms and dependability of distributed systems, especially Byzantine fault tolerance.

Tadashi Araragi is a Senior Research Scientist at NTT Communication Science Laboratories, NIPPON TELEGRAPH AND TELEPHONE CORPORATION. He received the B.E. and M.E. degrees in Mathematics from Tokyo University, in 1985 and 1987, respectively, and D.E. degree in Informatics from Kyoto University in 2006. He joined Communications and Information Processing Labs. of NTT, Yokosuka, Japan in 1987. His research interest includes formal verification, analysis of security protocols and fault tolerance of distributed systems. Dr. Araragi is a member of IEEE Society.

Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993–1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, IEICE and IPSJ.