Byzantine Fault-Tolerance with Commutative Commands

Pavel Raykov[1], Nicolas Schiper[2], and Fernando Pedone[2]

[1] Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland
[2] University of Lugano (USI)
Lugano, Switzerland

Abstract. State machine replication is a popular approach to increasing the availability of computer services. While it has been largely studied in the presence of crash-stop failures and malicious failures, all existing state machine replication protocols that provide byzantine fault-tolerance implement some variant of atomic broadcast. In this context, this paper makes two contributions. First, it presents the first byzantine fault-tolerant generic broadcast protocol. Generic broadcast is more general than atomic broadcast, in that it allows applications to deliver commutative commands out of order—delivering a command out of order can be done in fewer communication steps than delivering a command in the same order. Second, the paper presents an efficient state machine replication protocol that tolerates byzantine failures. Our protocol requires fewer message delays than the best existing solutions under similar conditions. Moreover, processing of commutative commands on replicas requires only two MAC operations. The protocol is speculative in that it may rollback non-commutative commands.

1 Introduction

State machine replication is a popular approach to increasing the availability of computer services [1, 2]. By replicating a service on multiple machines, hardware and software failures can be tolerated. Although state machine replication has been largely studied in the presence of crash-stop failures and malicious failures, all existing protocols that provide byzantine fault-tolerance (BFT) (e.g., [3–7]) implement some variant of atomic broadcast, a group communication primitive that guarantees agreement on the set of commands delivered and on their order. In this context, this paper makes two contributions.

The first contribution of this paper is a byzantine fault-tolerant generic broadcast protocol. Generic broadcast defines a conflict relation on messages, or commands, and only orders messages that conflict. Two messages conflict if their associated commands do not commute. For instance, two increment operations of some variable x commute since the final value of x is independent of the execution order of these operations. Generic broadcast generalizes atomic broadcast—the two problems are equivalent when every two messages conflict. Previous generic

broadcast protocols appeared in the crash-stop model [8–10]; ours is the first to tolerate malicious failures. The difficulty with generic broadcast stems from the need to deliver commutative commands in two communication delays and ensure that their delivery order, with respect to non-commutative commands, is the same at all correct processes. To address this challenge under byzantine failures we define Recovery Consensus, an abstraction that ensures proper ordering between conflicting and non-conflicting messages. The proposed protocol requires $n \geq 5f + 1$ replicas to tolerate f byzantine failures. We use Recovery Consensus at the core of our generic broadcast protocol.

The second contribution of this paper is a state machine replication protocol that generalizes and improves current byzantine fault-tolerant state machine replication protocols. Our protocol builds on our generic broadcast algorithm. A naive implementation of state machine replication based on generic broadcast to propagate commands to servers would lead to a best latency of three communication delays. We rely on speculative execution to provide an efficient algorithm that executes commutative commands in two communication delays. The algorithm is speculative in that it may rollback commands in some cases (i.e., when non-commutative commands are issued). To summarize, the principal advantage of the proposed state machine replication protocol is to allow fast execution of commutative commands in two message delays. Moreover, when commands commute servers only need to execute two MAC operations per command.

The remainder of the paper is structured as follows. Section 2 defines the system model. Sections 3 and 4, respectively, present the Recovery Consensus and generic broadcast protocols. We extend our generic broadcast protocol to provide state machine replication in Section 5. Section 6 discusses related work and Section 7 concludes the paper. Correctness proofs of the protocols can be found in the appendix of the full version of this paper [11].

2 System model and definitions

We consider an asynchronous message passing system composed of n processes $\Pi = \{p_1, \ldots, p_n\}$, out of which f are *byzantine* (i.e., they can behave arbitrarily). A process that is not byzantine is *correct*. The adversary that controls byzantine processes is computationally bounded (i.e., it cannot break cryptographic primitives) and cannot change the content of messages sent by one correct process to another correct process. The network is fully connected and *quasi-reliable*: if a correct process p sends a message m to a correct process q, then q receives m.[3] We make use of public-key signatures to allow a process to sign a message m [12]. We denote message m signed by process p_i as $\langle m \rangle_{\sigma_i}$. We also use HMACs [13] to establish a bidirectional authenticated channel between any two processes p_x and p_y, with the notation $\langle m \rangle_{\sigma_{xy}}$ indicating a message m signed with a secret key shared between processes p_x and p_y.

[3] The presented algorithms can trivially be modified to tolerate *fair-lossy* links, links that may drop messages but guarantee delivery of a message m if m is repeatedly sent. We assume quasi-reliable links to simplify the presentation of the algorithms.

Due to the impossibility to solve consensus in asynchronous systems prone to crash failures [14], it is also impossible to solve atomic broadcast and generic broadcast [15]. This impossibility is typically overcome by strengthening the model with further assumptions (e.g., [16–18]). In this paper we assume the existence of an atomic broadcast oracle [19]. Atomic broadcast is defined by the primitives A-Bcast(m) and A-Deliver(m), where m is a message. It guarantees the following properties:

- (Validity) If a correct process p A-Bcasts a message m, then p eventually A-Delivers m.
- (Agreement) If a correct process p A-Delivers a message m, then every correct process q eventually A-Delivers m.
- (Integrity) For any message m, every correct process p A-Delivers m at most once.
- (Order) If correct processes p and q both A-Deliver messages m and m', then p and q A-Deliver them in the same order.

3 Recovery Consensus

In this section we introduce Recovery Consensus, an abstraction used by generic broadcast to order messages whose associated commands do not commute, also denoted as conflicting messages. Below, we provide an implementation of Recovery Consensus that employs digital signatures for message authentication.

3.1 Problem definition

Recovery Consensus allows each process p_i to propose a set of non-conflicting messages $NCSet_i$ and a set of conflicting messages $CSet_i$. The set $NCSet_i$ is called non-conflicting since every pair of messages in it does not conflict, and the set $CSet_i$ is called conflicting since for every message $m \in CSet_i$ there is a message $m' \in NCSet_i$ such that m and m' conflict. Recovery Consensus ensures agreement on a set of non-conflicting messages $NCSet$ and on a set of conflicting messages $CSet$. Additionally, it guarantees that if $n_{chk} - f$ correct processes p_i propose a message m, i.e., m belongs to either $NCSet_i$ or $CSet_i$, where n_{chk} is a parameter of the problem, m will be part of either $NCSet$ or $CSet$.

More formally, Recovery Consensus is defined by primitives proposeRC($NCSet_i, CSet_i$) and decideRC($NCSet, CSet$). Provided that every correct process p_i invokes proposeRC($NCSet_i$, $CSet_i$) and there are no conflicting messages in $NCSet_i$, the following properties are guaranteed:

- (Termination) Every correct process eventually decides on some pair of message sets.
- (Agreement) If two correct processes decide on pairs of message sets ($NCSet_1$, $CSet_1$) and ($NCSet_2, CSet_2$), then $NCSet_1 = NCSet_2$ and $CSet_1 = CSet_2$.
- (Validity) If a correct process invokes decideRC($NCSet, CSet$), then:
  1. $NCSet \cap CSet = \varnothing$.

2. If a message m belongs to $n_{chk} - f$ $NCSet_i$ sets of correct processes, then $m \in NCSet$.
3. No two messages in $NCSet$ conflict.
4. If a message m is in $n_{chk} - f$ sets $NCSet_i \cup CSet_i$ of correct processes, then $m \in NCSet \cup CSet$.

3.2 Solving Recovery Consensus

Algorithm $\mathcal{C}_{absign}$ requires at least n_{chk} correct processes, where $n_{chk} \leq n - f$. It consists of a single task and works as follows. Each process p_i starts by atomically broadcasting the pair ($NCSet_i$, $CSet_i$) signed with p_i's signature—in the algorithm, the signed message is denoted as $\langle NCSet_i, CSet_i \rangle_{\sigma_i}$ (line 2). Process p_i then waits until it A-Delivered n_{chk} unique and valid messages, that is, messages from distinct sources that do not contain conflicting messages in $NCSet_j$ (line 3). Detecting unique messages is done with signatures: if p_i A-Delivers two messages from the same source p_j, then p_i discards both messages since p_j is byzantine. By considering the first n_{chk} unique and valid messages, this ensures that at least $n_{chk} - f$ messages A-Delivered by p_i were broadcast by correct processes.

Algorithm $\mathcal{C}_{absign}$
Process p_i Recovery Consensus algorithm with atomic broadcast and signatures

1: Procedure proposeRC($NCSet_i, CSet_i$)
2: A-Bcast($\langle NCSet_i, CSet_i \rangle_{\sigma_i}$)
3: **wait until** [$|GS| = n_{chk} : GS \stackrel{def}{=} \{(NCSet_j, CSet_j)$ | A-Delivered unique and
4: valid $\langle NCSet_j, CSet_j \rangle_{\sigma_j}$ from p_j }]
5: $NCSet \leftarrow \{m \mid \exists \lceil \frac{n_{chk}+1}{2} \rceil \ NCSet_j : (NCSet_j, \cdot) \in GS$ **and** $m \in NCSet_j\}$
6: $CSet \leftarrow (\bigcup_{(NCSet_j, CSet_j) \in GS} NCSet_j \cup CSet_j) \setminus NCSet$
7: decideRC($NCSet, CSet$)

Any message that appears in a majority of the n_{chk} $NCSet_j$ sets will appear in $NCSet$ (line 5). This guarantees the third validity property, namely that no two messages in $NCSet$ conflict, since (i) the considered $NCSet_j$ sets at line 3 do not contain conflicting messages and (ii) any message in $NCSet$ belongs to a majority of $NCSet_j$ sets.

Let $Q_{n_{chk}-f}$ be a quorum of $n_{chk}-f$ correct processes that propose a message m as part of $NCSet_i$, and let $Q_{n_{chk}}$ be a quorum of n_{chk} processes, the number of unique and valid A-Delivered messages processes consider at line 3. To ensure that we include m in $NCSet$ if m belongs to $n_{chk} - f$ $NCSet_i$ sets of correct processes, the minimum size of the intersection between $Q_{n_{chk}-f}$ and $Q_{n_{chk}}$ must be $\lceil \frac{n_{chk}+1}{2} \rceil$. Hence, n_{chk} must satisfy inequality $(n_{chk}-f)+n_{chk} \geq n+\lceil \frac{n_{chk}+1}{2} \rceil$. Since $n_{chk} \leq n - f$, we conclude that $\mathcal{C}_{absign}$ requires $n > 5f$.

Finally, the set of conflicting messages $CSet$ consists of messages gathered using atomic broadcast that are not part of $NCSet$ (line 6). From the total order

property of atomic broadcast, correct processes gather the same set of pairs $(NCSet_j, CSet_j)$. Since $NCSet$ and $CSet$ are constructed from the gathered pairs using a deterministic procedure, all correct processes agree on these two sets.

4 BFT Generic Broadcast

We present a byzantine fault-tolerant generic broadcast protocol that we denote as $\mathcal{PGB}$. This protocol relies on Recovery Consensus to handle conflicting messages. We first define generic broadcast in our model and then present the algorithm.

4.1 Generic Broadcast

Generic broadcast is defined by the primitives g-Broadcast(m) and g-Deliver(m), where m is a message from the predefined set $\mathcal{M}$, to which all messages belong. We assume that each message broadcast has a unique identifier. Generic broadcast is parameterized by a symmetric relation $\sim$ on $\mathcal{M} \times \mathcal{M}$. If $(m, m') \in \sim$, or $m \sim m'$ for short, we say that m and m' *conflict* or are *conflicting messages*. If m and m' conflict, then generic broadcast will order m and m'. If m and m' do not conflict, they can be delivered in any order.

Generic broadcast guarantees the following properties, adapted from [10] to the byzantine failure model:

- (Validity) If a correct process p g-Broadcasts a message m, then p eventually g-Delivers m.
- (Agreement) If a correct process p g-Delivers a message m, then every correct process q eventually g-Delivers m.
- (Integrity) For any message m, every correct process p g-Delivers m at most once.
- (Order) If correct processes p and q both g-Deliver conflicting messages m and m' ($m \sim m'$), then p and q g-Deliver them in the same order.

As noted in [10], atomic broadcast is a special case of generic broadcast when all messages conflict with all messages, that is, $\sim = \mathcal{M} \times \mathcal{M}$. Thus, one could question the difficulty of implementing generic broadcast since we assume the existence of an atomic broadcast primitive that could be used to implement generic broadcast. The main idea of our generic broadcast protocol is that it allows *fast delivery* (i.e., in two communication delays) of non-conflicting messages, a bound that no atomic broadcast protocol can achieve in the general case [20].

4.2 Solving Generic Broadcast

The protocol is composed of two phases: an acknowledgment phase (ACK) and a check phase (CHK). Consecutive ACK and CHK phases form a "round". During the ACK phase processes g-Deliver non-conflicting messages in two message delays.

Algorithm $\mathcal{PGB}$

Process p_i generic broadcast algorithm

1: Initialization:
2: $\quad Received \leftarrow \varnothing$, $G\_del \leftarrow \varnothing$, $pending^1 \leftarrow \varnothing$, $gAck\_del^1 \leftarrow \varnothing$, $k \leftarrow 1$
3: To execute $g\text{-}Broadcast(m)$: {Task 1}
4: $\quad$ send(m) to all
5: $g\text{-}Deliver(m)$ occurs as follows:
6: **when** receive(m) **do** {Task 2a}
7: $\quad Received \leftarrow Received \cup \{m\}$
8: **when** receive$(\langle k,\ pending_j^k,\ \text{ACK}\rangle_{\sigma_{ij}})$ **do** {Task 2b}
9: $\quad Received \leftarrow Received \cup pending_j^k$
10: **when** receive(k, S_j, CHK) **do** {Task 2c}
11: $\quad Received \leftarrow Received \cup S_j$
12: **when**$\big(Received \setminus (G\_del \cup pending^k) \neq \varnothing\big)$ **do** {Task 3}
13: $\quad$ **if** $(\forall\, m, m' \in (Received \setminus G\_del)\ :\ m \not\sim m')$ **then**
14: $\quad\quad pending^k \leftarrow Received \setminus G\_del$
15: $\quad\quad$ send$(\langle k,\ pending^k,\ \text{ACK}\ \rangle_{\sigma_{ij}})$ to all processes p_j
16: $\quad$ **else**
17: $\quad\quad$ send$(k, (Received \setminus G_del), \text{CHK})$ to all ▷ start of CHK phase
18: $\quad\quad$ proposeRC$(k, pending^k, (Received \setminus (G_del \cup pending^k)))$
19: $\quad\quad$ **wait until** decideRC$(k, NCSet^k, CSet^k)$
20: $\quad\quad$ **for each** $m \in NCSet^k \setminus (G\_del \cup gAck\_del^k)$ **do** $g\text{-}Deliver(m)$
21: $\quad\quad$ **for each** $m \in CSet^k \setminus (G\_del \cup gAck\_del^k)$ **in ID order do**
22: $\quad\quad\quad g\text{-}Deliver(m)$ } atomic
23: $\quad\quad G\_del \leftarrow G\_del \cup NCSet^k \cup CSet^k$
24: $\quad\quad k \leftarrow k+1$, $pending^k \leftarrow \varnothing$, $gAck\_del^k \leftarrow \varnothing$ ▷ end of CHK phase }
25: $\quad$ **end if**
26: **when** $\exists\, m$: [for n_{ack} processes p_j : received $\langle k,\ pending_j^k,\ \text{ACK}\rangle_{\sigma_{ij}}$ {Task 4}
27: $\quad\quad$ from p_j **and** $m \in (pending_j^k \setminus gAck\_del^k) \cap pending^k$] **do** } atomic
28: $\quad gAck\_del^k \leftarrow gAck\_del^k \cup \{m\}$
29: $\quad g\text{-}Deliver(m)$

In the CHK phase, the protocol orders conflicting messages. Notice that $\mathcal{PGB}$ does not require signatures to deliver non-conflicting messages.

Algorithm $\mathcal{PGB}$ consists of six concurrent tasks. Each line of the algorithm, lines 20–24, and lines 26–29 are executed atomically. The following variables are used by the algorithm: k defines the current round number, *Received* contains all the g-Broadcast messages that the process has received so far, G_del contains all the messages that have been g-Delivered in the previous rounds, $pending^k$ defines the set of non-conflicting messages acknowledged by the process in the current round, and $gAck\_del^k$ is the set of messages g-Delivered in the ACK phase of the current round.

When a process p wishes to g-Broadcast a message m, p sends m to all (line 4). When receiving m, a process q adds m to its *Received* set (line 7) and eventually checks whether m conflicts with any message that was received but not

delivered yet (line 13). If it is not the case, then q adds m to its $pending^k$ set and acknowledges all messages in this set by sending $pending^k$ to all (lines 14–15).[4] A process q g-Delivers m in the ACK phase when q receives n_{ack} acknowledgments for m (lines 26–29). To prevent conflicting messages from being g-Delivered in the ACK phase despite f byzantine processes, n_{ack} must be greater than $(n + f)/2$.

It is possible that q receives a message m' that conflicts with m before receiving n_{ack} acknowledgments for m. In that case, q proceeds to the CHK phase. At this point, processes start by exchanging all messages that they received but did not deliver in previous rounds (line 17). In doing so, all correct processes eventually receive m and m' despite potentially faulty senders, and enter the CHK phase. $\mathcal{PGB}$ then relies on Recovery Consensus to ensure agreement on the set of non-conflicting messages $NCSet^k$ that were potentially g-Delivered in the ACK phase, and the set $CSet^k$ of conflicting messages to deliver at the end of the current round. Correct processes invoke *proposeRC* with round number k, the set of non-conflicting messages $pending^k$, and all the other messages that were received but not delivered so far, denoted as $Received \setminus (G\_del \cup pending^k)$ (line 18), and decide on sets $NCSet^k$ and $CSet^k$ (line 19). Processes deliver non-conflicting messages in $NCSet^k$ that they had not delivered so far (line 20), and then deliver conflicting messages $CSet^k$ (line 22).

To ensure that if a message m was delivered in the ACK phase m will appear in set $NCSet^k$ decided by Recovery Consensus, m must be proposed by $n_{chk} - f$ correct processes in $pending^k$ at line 18. If m was delivered in the ACK phase, at least $n_{ack} - f$ correct processes propose m to Recovery Consensus. Hence, to maximize resilience we set n_{ack} equal to n_{chk}.

Note that Recovery Consensus (Algorithm $\mathcal{C}_{absign}$) runs n atomic broadcasts in parallel. Hence, when conflicting messages are issued, $\mathcal{PGB}$ has message complexity n times bigger than a usual atomic broadcast protocol. $\mathcal{PGB}$ is optimized to perform well when non-conflicting messages are broadcast and Recovery Consensus is invoked rarely.

5 State Machine Replication

5.1 A trivial algorithm

Implementing state machine replication [1, 2] using the generic broadcast algorithm of Section 4 is straightforward: each command of the state machine corresponds to a message in the set $\mathcal{M}$ and the conflict relation on messages is defined such that two messages conflict if and only if their associated commands do not commute. For instance, if replicas store bank accounts, two deposit commands on the same account commute since their execution order does not have an effect on the final state of the state machine nor on the respective outputs of these commands, which in this case only contain an acknowledgment that the

[4] To ensure that messages are not acknowledged twice and improve the efficiency of the algorithm, processes can remember the set of messages that were acknowledged and only acknowledge them once.

Algorithm $\mathcal{SMR}_{client}$
Client c algorithm

1: To execute command m:
2: send($\langle m \rangle_{\sigma_c}$) to all replicas
3: **wait until** [$\exists\, k$, s.t. received from different replicas
4: n_{ack} $\langle k, m, res(m), \text{ACK} \rangle_{\sigma_{r_i c}}$ **or** $f + 1$ $\langle k, m, res(m), \text{CHK} \rangle_{\sigma_{r_i c}}$]
5: **return** res(m)

operations were successfully executed. Clients can then directly broadcast commands to replicas using Algorithm $\mathcal{PGB}$. Once replicas deliver a command, they execute it and send back the result to the client. When $f + 1$ identical replies are received by the client, the result of the command is known. This technique guarantees a form of linearizability [21, 22].

5.2 An optimal algorithm

The above algorithm allows clients to learn the outcome of a command *cmd* in three communication delays if *cmd* commutes with concurrent commands. As we show next, a lower latency can be achieved by modifying Algorithm $\mathcal{PGB}$ and speculatively executing commands.

Before presenting the algorithm, we extend the system model of Section 2. We assume a population of n *replicas*, aforenamed processes, and a set of *clients*. Any number of clients may be byzantine and f bounds the number of faulty replicas. The latter execute a command *cmd* of the state machine by invoking *execute*(*cmd*). This invocation modifies the state of the replica and returns a result. Commands are deterministic, that is, they produce a new state and a result only based on the current state. Our protocol speculatively executes commands and may require *rolling back* some commands if their speculative order does not correspond to their definitive order. The effect of operation *rollback*(*cmd*) is such that if a sequence of commands *Seq* is executed between *execute*(*cmd*) and *rollback*(*cmd*) then the replica's state is as if only commands in sequence *Seq* were executed. Notice that although replicas may rollback some commands, clients always see the definitive result of a command (i.e., clients do not perform rollbacks).

The protocols for clients and replicas are presented in Algorithms $\mathcal{SMR}_{client}$ and $\mathcal{SMR}_{replica}$ respectively. The replica's algorithm is similar to $\mathcal{PGB}$, except for the handling of acknowledgment messages, which is moved to the client. We highlight in gray the differences between $\mathcal{SMR}_{replica}$ and $\mathcal{PGB}$. Similarly to $\mathcal{PGB}$, replicas do not need to sign any messages when clients issue commutative commands.

When a client c invokes a command m, c sends $\langle m \rangle_{\sigma_c}$ to all replicas ($\mathcal{SMR}_{client}$, line 2). A replica includes message $\langle m \rangle_{\sigma_c}$ in the *Received* set at lines 4,6,8 if m's signature is valid. When m arrives at a replica r, one of two things can happen: either (a) m does not conflict with any other command that

Algorithm $\mathcal{SMR}_{replica}$
Replica r algorithm (the differences with Algorithm $\mathcal{PGB}$ are highlighted in gray)

```
1: Initialization:
2:     Received ← ∅, G_del ← ∅, pending^1 ← ∅, k ← 1, Res ← ∅
3: when receive(⟨m⟩_σc) do                                              {Task 1a}
4:     Received ← Received ∪ {⟨m⟩_σc}
5: when receive(k, pending_j^k, ACK) do                                 {Task 1b}
6:     Received ← Received ∪ pending_j^k
7: when receive(k, S_j, CHK) do                                         {Task 1c}
8:     Received ← Received ∪ S_j
9: when (Received ∖ (G_del ∪ pending^k) ≠ ∅) do                          {Task 2}
10:    if (∀ m, m' ∈ (Received ∖ G_del) : m ⇝ m') then
11:        for each m ∈ (Received ∖ (G_del ∪ pending^k)) do
12:            res(m) ← execute(m), Res ← Res ∪ (m, res(m))
13:            send(⟨k, m, res(m), ACK⟩_σrc) to client(m)
14:        pending^k ← Received ∖ G_del
15:        send(k, pending^k, ACK) to all replicas
16:    else
17:        send(k, (Received ∖ G_del), CHK) to all replicas          ▷ start of CHK phase
18:        proposeRC(k, pending^k, (Received ∖ (G_del ∪ pending^k)))
19:        wait until decideRC(k, NCSet^k, CSet^k)
20:        for each m ∈ pending^k ∖ NCSet^k do
21:            rollback(m), remove(m, res(m)) from Res
22:        for each m ∈ NCSet^k ∖ G_del do
23:            if m ∉ pending^k then res(m) ← execute(m)
24:            send(⟨k, m, res(m), CHK⟩_σrc) to client(m)    ▷ res(m) is retrieved from
   Res if needed
25:        in ID order: for each m ∈ CSet^k ∖ G_del do
26:            res(m) ← execute(m), send(⟨k, m, res(m), CHK⟩_σrc) to client(m)
27:        G_del ← G_del ∪ NCSet^k ∪ CSet^k
28:        k ← k + 1, pending^k ← ∅, Res ← ∅                          ▷ end of CHK phase
29:    end if
```

r received in the current round or (b) m conflicts with a command received in the same round.

In case (a), r speculatively executes m, stores the result in set Res, and sends the result back to the client as an acknowledgment message ($\mathcal{SMR}_{replica}$, lines 10–12). We use a function client(m) defining for a given message m the client that issued m. If client c receives n_{ack} identical acknowledgment messages for m, c learns the result of command m ($\mathcal{SMR}_{client}$, lines 3–5)—this is a similar condition under which a process can g-Deliver a message in the ACK phase of Algorithm $\mathcal{PGB}$.

In case (b), command m conflicts with a command received in the current round. Similarly to $\mathcal{PGB}$, each replica r uses Recovery Consensus to order these commands. For each command m' that was received by r in the ACK phase but that does not appear in the decided $NCSet$, r rollbacks m' and deletes

the corresponding entry from Res—the speculative execution order of m' differs from its final execution order ($\mathcal{SMR}_{replica}$, line 21).

Then, commands in $NCSet$ are executed if they were not acknowledged in the ACK phase, and the results of these commands are sent to the corresponding clients (lines 22–24). Similar actions are done for the conflicting commands of $CSet$ (lines 25–26). A client learns the result of a command m that was executed in the CHK phase after receiving $f+1$ identical replies for m ($\mathcal{SMR}_{client}$, line 4).

5.3 Optimizations

We briefly discuss two optimizations allowing Algorithms $\mathcal{SMR}_{client}$ and $\mathcal{SMR}_{replica}$ (a) to achieve the optimal latency of two communication delays in executions without *contention*, defined next, and (b) to avoid message signing by clients.

No contention. Assume that pending sets contain the order in which commands were received and executed by the replicas; essentially, a pending set becomes a command sequence. We say that two pending sets conflict if they contain two conflicting messages executed in a different order. When no pending sets conflict, we say that there is no *contention*.

The main idea behind this optimization is that now we consider the conflicts between pending sets instead of the conflicts between individual messages. In the optimized Algorithm $\mathcal{SMR}_{client}$, a client c learns the result $res(m)$ of the execution of command m if: (1) c received n_{ack} *non-conflicting* pending sets with $res(m)$ or (2) c received $f+1$ CHK messages with $res(m)$. A replica enters the CHK phase if: (1) it has received two conflicting *pending* sets or (2) it has received a CHK message indicating that some other replica entered the CHK phase in the current round.

Since conflicting commands can be executed in the ACK phase, provided that they are executed in the same order, replicas include the execution order of commands in sets $NCSet_i$ proposed to Recovery Consensus. Hence, the Recovery Consensus algorithm must be modified and $NCSet$ essentially contains commands proposed by $\lceil \frac{n_{chk}+1}{2} \rceil$ replicas r_i as part of $NCSet_i$, such that no two pending sets containing m include two non-commutative commands m_1 and m_2 that were executed before m and in different orders.

Avoiding message signing by clients. Digital signatures based on asymmetric cryptography can be expensive to generate or verify, let alone the problem of distributing and refreshing key pairs. Instead of signing a message m, clients can use an authenticator (a list of HMACs) to authenticate m [5].

We modify Algorithm $\mathcal{SMR}_{replica}$ as follows: (1) during the ACK phase replica r_j puts message m at lines 4,6,8 in the *Received* set only if m's authenticator contains a valid HMAC entry for r_j; (2) during the CHK phase, we change the way $CSet$ is built in the underlying protocol $\mathcal{C}_{absign}$: message m is included in $CSet$ only if it belongs to $f+1$ different $NCSet_j \cup CSet_j$. This

Protocol	PBFT [5]	Zyzzyva [3]	HQ [6]	Q/U [4]	Aliph [7]	this paper
Resilience	$f < n/3$	$f < n/3$	$f < n/3$	$f < n/5$	$f < n/3$	$f < n/5$
Best-case latency	4	3	4	2	2	2
Best-case latency in the absence of...	bf	bf slow links	bf contention	bf contention	bf contention slow links	bf contention
MAC operations at bottleneck server	2+8f	2+3f	2+4f	2+4f	2[5]	2
Command classification	read-only/ mutative	read-only/ mutative	read-only/ mutative	read-only/ mutative	none	by conflict relation $\sim$
Client-based recovery	no	yes	yes	yes	yes	no

Table 1. Byzantine fault-tolerant replication protocols ("bf":"byzantine failures").

guarantees that only client c can issue commands with c's identifier, i.e., it is impossible to impersonate client c.

Unfortunately these modifications are more difficult to apply in Recovery Consensus. To avoid expensive signing during Recovery Consensus one could use matrix signatures [23] or employ the approach described in [6] for signing certificates, both of which essentially trade off signatures for additional network delays.

Digital signatures scale better than authenticators, whose size grows linearly with the number of replicas, so deciding which technique to apply depends on the specific system settings. In any case, we note that by design, Algorithms $\mathcal{SMR}_{replica}$ and $\mathcal{SMR}_{client}$ optimize the ACK phase, since this is the case we expect to happen more often.

6 Related work

In the following we compare our BFT state machine replication protocol to the related work (see Table 1). To the best of our knowledge, this paper is the first to present an implementation of byzantine generic broadcast. All BFT state machine replication protocols we are aware of have a "fast mode"—analogous to the ACK phase, where messages are delivered fast under certain assumptions (also called "best-case"), and a recovery mechanism to switch to a "slow mode"—analogous to the CHK phase that resolves possible problems, usually contention or failures. Despite these similarities, existing protocols differ from each other in a number of aspects:

– PBFT was the first practical work on BFT state-machine replication. The best-case latency of four message delays is achieved when there are no byzantine failures. For read-only operations, the protocol can be optimized to achieve a latency of two message delays.

[5] In Table 2 of paper [7], Aliph's latency and throughput represent two different subprotocols: *Chain* and *Quorum*. We here show the number of MAC operations that *Quorum* uses since only *Quorum* achieves the best-case latency of two network delays.

- Zyzzyva employs tentative execution to improve the best-case delay of PBFT. It executes commands in three network delays when there are no byzantine failures and links are timely. Otherwise, the protocol requires five network delays. Like PBFT, it can be optimized to execute read-only operations in two message delays.
- HQ, a descendant of PBFT, is optimized to execute read-only commands in two message delays and update commands in four message delays when the execution is contention-free.
- Q/U was the first protocol to achieve the best-case latency of two network delays for all commands when replicas are failure-free and updates do not access the same object concurrently.
- In [7], the authors propose a modular approach to build BFT services based on the concept of abstract instances. An abstract instance is a BFT replication protocol optimized for specific system conditions that can abort commands. In this context, the authors propose Aliph, a composition of three abstract instances: *Quorum*, *Chain*, and *PBFT*. Quorum is optimized for latency and allows command execution in two network delays when links are timely and the execution is contention- and failure-free. Chain, on the other hand, is optimized for throughput and achieves a latency of $f + 2$ network delays when there are no failures.

The protocol presented in this paper is the first to achieve a latency of two network delays when the execution is failure-free but concurrent commutative commands are submitted. Under the same conditions, PBFT and Zyzzyva achieve latency of four and three message delays respectively, while HQ, Q/U and Aliph run an additional protocol to resolve contention.

Q/U [4] and HQ [6] proposed a simplified version of the conflict relation: all commands are either *reads* or *writes* [6] (respectively, *queries* and *updates* in [4]); reads do not conflict with reads, and writes conflict with reads and writes. This is more restrictive than a conflict relation, as mutative operations on the same object do not necessarily conflict (e.g., incrementing a variable).

Zyzzyva, Q/U, and Aliph, more specifically the *Quorum* instance, do not use inter-replica communication to agree on the order of commands; instead they assume that it is the client's responsibility to resolve contention by collecting authenticated responses from replicas and distributing a valid certificate to the replicas. PBFT and the state machine replication protocol presented in this paper rely on inter-replica communication to serialize commands, which allows a lightweight protocol for clients. HQ uses a hybrid approach: it uses inter-replica communication only when clients demand to resolve contention explicitly, while in the "fast case" clients coordinate the execution.

Finally, we note that although [24] executes commutative commands in parallel, all commands are totally ordered using PBFT, resulting in a higher latency than our protocol in the aforementioned scenario.

7 Final remarks

This paper introduces the first generic broadcast algorithm that tolerates byzantine failures. Generic broadcast is based on message conflicts, a notion that is more general than read and write operations [4, 6]. The proposed algorithm is modular and relies on an abstraction called Recovery Consensus, used to ensure that correct processes (i) deliver the same set of non-conflicting messages in each round, and (ii) agree on the delivery order of conflicting messages. A modular approach facilitates the understanding of the ACK phase and CHK phase of Algorithm $\mathcal{PGB}$, and allows to explore various implementations of Recovery Consensus. We provided an implementation of Recovery Consensus that is based on atomic broadcast and digital signatures and requires at least $5f+1$ processes. Finally, we extended the proposed generic broadcast algorithm to provide state machine replication. The resulting protocol, with its optimizations, can execute commands in two message delays under weaker assumptions than state-of-the-art algorithms.

The Aliph protocol [7] opened new directions in the development of state machine replication protocols: it is now possible to combine different protocols in one that switches through given implementations of state machine replication under certain policies to speed up the execution. Hence, it could be an interesting task to implement the protocol proposed in this paper as an `Abstract` instance (*Generic*) and see the behavior of the resulting algorithm (e.g., *Generic-Chain-Quorum-Backup*).

References

1. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (1978) 558–565
2. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. **22** (1990) 299–319
3. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. ACM Transactions on Computer Systems **27** (2009) 1–39
4. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable byzantine fault-tolerant services. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, New York, NY, USA, ACM (2005) 59–74
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems **20** (2002) 398–461
6. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: a hybrid quorum protocol for byzantine fault tolerance. In: OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation, Berkeley, CA, USA, USENIX Association (2006) 177–190
7. Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M.: The next 700 bft protocols. In: EuroSys '10: Proceedings of the 5th European conference on Computer systems, New York, NY, USA, ACM (2010) 363–376

8. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Thrifty generic broadcast. In: Proceedings of DISC'00, Springer-Verlag (2000) 268–283
9. Lamport, L.: Generalized consensus and paxos. Technical report, Microsoft Research MSR-TR-2005-33 (2005)
10. Pedone, F., Schiper, A.: Handling message semantics with generic broadcast protocols. Distributed Computing **15** (2002) 97–107
11. Raykov, P., Schiper, N., Pedone, F.: Byzantine fault-tolerance with commutative commands. Technical report, University of Lugano, http://www.inf.usi.ch/faculty/pedone/Paper/2011/2011OPODIS-full.pdf (2011)
12. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **26** (1983) 96–99
13. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology. CRYPTO '96, London, UK, Springer-Verlag (1996) 1–15
14. Fischer, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
15. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
16. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1983) 27–30
17. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35** (1988) 288–323
18. Toueg, S.: Randomized byzantine agreements. In: PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1984) 163–178
19. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: Advances in Cryptology — CRYPTO 2001. Volume 2139 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2001) 524–541
20. Lamport, L.: Lower bounds for asynchronous consensus. Distributed Computing **19** (2006) 104–125
21. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12** (1990) 463–492
22. Malkhi, D., Reiter, M., Lynch, N.: A correctness condition for memory shared by byzantine processes. Unpublished manuscript (1998)
23. Aiyer, A.S., Alvisi, L., Bazzi, R.A., Clement, A.: Matrix signatures: From macs to digital signatures in distributed systems. In: DISC '08: Proceedings of the 22nd international symposium on Distributed Computing, Berlin, Heidelberg, Springer-Verlag (2008) 16–31
24. Kotla, R., Dahlin, M.: High-throughput byzantine fault tolerance. In: International Conference on Dependable Systems and Networks (DSN). (2004)