Tradeoffs in Byzantine-Fault-Tolerant State-Machine-Replication Protocol Design

Michael G. Merideth

March 2008
CMU-ISR-08-110

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Many state-machine-replication protocols perform the same tasks of tolerating Byzantine faults and guaranteeing consistency in an asynchronous environment. However, each protocol seems uniquely complex in part because commonalities are lost in descriptions of the protocols. In this paper, we identify Byzantine quorum systems as a unifying factor in the design of each protocol. Leveraging this, we present a framework of high-level, logical phases, which may be optimistic or pessimistic, as a path to understanding: the number of servers required; the number of faults that can be tolerated; and the number of rounds of communication employed by each protocol. Our framework highlights a tradeoff between the number of rounds of communication required and the maximum number of faults that can be tolerated. Furthermore, it highlights an independent tradeoff between an additional round of communication and possibly unnecessary computation. We use the framework to describe three mainstream state-machine-replication protocols and their variants.

This work was partially supported by NSF grant CCF-0424422.

Keywords: Distributed systems, Byzantine fault tolerance, state machine replication, quorum systems, survey

1 Introduction

State machine replication [18] is a way to implement a fault-tolerant stateful service. It involves replicating the service (i.e., running multiple copies) while guaranteeing *consistency*—the illusion of a single centralized service. Conceptually, the operations provided by the service are mapped to the state transitions of a deterministic state machine. Each replica runs a copy of the state machine. The state-machine-replication protocol maintains consistency by ensuring that each replica processes the same requests in the same order.

Byzantine-fault-tolerant state-machine-replication protocols (e.g., [16, 8, 4, 17, 3, 20, 10, 15, 1]) are powerful because of their ability to work even if up to b of the n total replicas and any number of the clients are faulty such that they behave arbitrarily or maliciously (i.e., Byzantine faults [11]). Unfortunately, due to the Byzantine fault model, no single replica or client can be trusted independently to provide the same ordered sequence of client requests to all replicas.

Instead, the non-faulty replicas work together to choose the same request ordering and to ensure that they all receive the same requests. However, this is complicated by an *asynchronous timing model*, i.e., one in which messages might be delayed for arbitrary lengths of time. In such an environment, it is impossible to ensure consistency while simultaneously ensuring availability (i.e., allowing for operations to complete) [6].[1]

A number of state-machine-replication protocols (e.g., [4, 17, 20, 10, 15, 1]) choose to guarantee consistency in an asynchronous environment while ensuring availability only during periods of synchrony in which messages to/from non-faulty replicas are delivered in a timely fashion. These protocols maintain consistency by using Byzantine quorum systems [13], which require only any quorum (subset) of the replicas to be involved in any operation. Inherently, the use of Byzantine quorum systems allows the protocols to maintain consistency while making progress even if some replicas never receive or process requests.

Despite the commonalities of the protocols, some such as BFT [4] (and its derivatives like [17, 20, 10]) are commonly labeled atomic-broadcast protocols, others like Q/U [1] look more like Byzantine-quorum system protocols, while still others like FaB Paxos [15] seem to fall somewhere in between.

However, as we show, the common use of Byzantine quorum systems in these protocols provides a basis for their comparison. We present a framework of four high-level phases: propose; accept; update; and verify. The framework allows us to compare the designs of the protocols, and to see the tradeoffs made in terms of the number of servers required, the number of faults that can be tolerated, and the number of rounds of communication required. The accept and update phases may be either optimistic or pessimistic. An optimistic accept phase requires an opaque quorum system construction, and therefore at least $5b + 1$ servers; a pessimistic accept phase can use dissemination or masking quorum systems, and therefore as few as $3b + 1$ servers, but requires at least one additional round of communication. An optimistic update phase allows replicas to process requests that may later be given a different ordering; a pessimistic update phase requires

[1] Intuitively, this is because one cannot distinguish between a faulty replica that is not responding, and a non-faulty replica that is slow in responding or from which messages are delayed. Therefore, a non-faulty replica generally cannot wait for more than $n - b$ responses from different replicas before proceeding, or else risk waiting indefinitely.

an additional round of communication, but allows replicas to be confident that the request ordering is permanent before processing.

2 Preliminaries

We begin with: (i) the system model; (ii) a summary of the intuition behind common set sizes like $n - b$ and $2b + 1$; and (iii) a review of the three types of quorum systems related to the protocols we survey.

2.1 System Model

The system consists of a universe U of n replicas, and an arbitrary, but bounded, number of clients. There is a set $B \subset U$ that represents the b faulty replicas; the composition of B is known by the faulty clients and replicas, but not by the non-faulty ones. The remaining $n-b$ replicas, i.e., $U \setminus B$, are non-faulty. Faulty replicas and clients can behave arbitrarily (i.e., Byzantine faults [11]), but, as is typical, are computationally bound such that, e.g., they cannot subvert cryptographic primitives (e.g., cryptographic hash functions) used in the protocols. The protocols maintain consistency without assumptions about the processing rates of non-faulty clients and replicas or the message delays of the network (asynchronous timing model). However, for availability, these delays must be finite, and the processing rates must be non-zero.

2.2 Sizes of Sets

The meanings of quantities such as $n - b$, $b + 1$, and $2b + 1$ can be a source of confusion. This is sometimes compounded in descriptions of the protocols by substitution of quantities that are not necessarily equivalent in all contexts. For example, the quantity $2b + 1$ has been used for at least three distinct conceptual purposes (described in more detail below): (i) the maximum number of responses for which to wait given the asynchronous timing model ($n - b$); (ii) a set with a non-faulty majority ($2b + 1$); and (iii) the set size that guarantees at least one non-faulty replica in the intersection of any two sets ($\lceil (n + b + 1)/2 \rceil$). As seen in the following equations, these quantities are equivalent under the BFT assumption that $n = 3b + 1$:

$$\begin{aligned}
&\lceil (n + b + 1)/2 \rceil \\
&\quad = \lceil (3b + 1 + b + 1)/2 \rceil \\
&\quad = \lceil (4b + 2)/2 \rceil \\
&\quad = (2b + 1). \\
&(n - b) \\
&\quad = (3b + 1 - b) \\
&\quad = (2b + 1).
\end{aligned}$$

However, these quantities are not necessarily equivalent given other values for n or other types of quorum systems. Note in particular that $n - b$ and $\lceil (n + b + 1)/2 \rceil$ depend on the values of n and

b while $2b+1$ depends only on b. Because these quantities are very different conceptually, the use of a single identifier like $2b+1$ can make the description of a protocol difficult to decipher.

Guaranteed responses ($n-b$). Given an asynchronous timing model and a system that can tolerate up to b faults, a process can wait for up to, but not more than, $n-b$ responses. This is because b replicas may be faulty and may never respond, and it is impossible to distinguish between a faulty process and one that is slow.

Guaranteed non-faulty responses ($n-2b$). As described above, one can wait for only $n-b$ responses. Of these responses, b may be from faulty replicas; the b replicas not represented in the set of responses may have simply been slower. Note that in exceptional cases, this may not be hold. In particular, if one can detect that a response is from a faulty replica based on some property of the response, then one can wait for an additional response from a non-faulty replica for each response from a faulty replica.

At least one non-faulty replica ($b+1$). Because there are at most b faulty replicas, any set of $b+1$ responses contains at least one response from a non-faulty replica.

Non-faulty majority ($2b+1$). Because there are at most b faulty replicas, any set of $2b+1$ replicas necessarily contains at least $b+1$ non-faulty replicas. As such, the majority of any set of at least $2b+1$ replicas is non-faulty.

Replica in intersection ($\lceil (n+1)/2 \rceil$). Any two sets of $\lceil (n+1)/2 \rceil$ replicas chosen from the n total replicas contain at least one replica in common.

Non-faulty replica in intersection ($\lceil (n+b+1)/2 \rceil$). Any two sets of $\lceil (n+b+1)/2 \rceil$ replicas chosen from the n total replicas contain at least one non-faulty replica in common.

$c+1$ **non-faulty replicas in intersection ($\lceil (n+b+c+1)/2 \rceil$).** For any non-negative constant c, any two sets of $\lceil (n+b+c+1)/2 \rceil$ replicas chosen from the n total replicas contain at least $c+1$ non-faulty replicas in common.

2.3 Byzantine Quorum Systems

A Byzantine quorum system [13] is a set of quorums (subsets) of replicas. As seen in Table 1, the different types of threshold quorum systems make different assumptions concerning the replicas that may vote for conflicting candidates (defined below); this has implications for the number of servers required as well as the number of rounds of communication needed to ensure a successful update. Quorums are small enough to ensure that there is always an *available* quorum (one in which all replicas respond during periods of synchrony); this involves setting the size of quorums

$q \leq n - b$. The quorums are used for read and update operations. They overlap in enough non-faulty replicas to ensure that updates written to one quorum are propagated to other quorums and cannot be fabricated or corrupted by faulty replicas.

An update that is accepted by a replica yields a *candidate* at that replica. A candidate is *established* once it is accepted by all of the non-faulty replicas in some update quorum. As discussed below, in opaque quorum systems (used by protocols that have an optimistic accept phase), different non-faulty replicas may have different candidates issued by concurrent updates at a given instant. (This must be prevented by the protocol if a masking or dissemination quorum system is used.) Moreover, in either masking or opaque quorum systems, a faulty replica may try to forge a concurrent candidate. If there are multiple concurrent candidates and one is established, the others are called *conflicting*. A replica can try to *vote* for some candidate by sending a message claiming to have accepted it.

Byzantine quorum systems require that a candidate written to an update quorum be *observed* in any other quorum; this is how candidates are propagated. To be more precise, we say that a candidate is observed in a read quorum if it receives at least a threshold r of votes from different replicas. Therefore, the number of non-faulty replicas in the intersection of the update quorum and any other quorum falls in the range $[r..q]$, where r is greater than the number of replicas that could vote for a conflicting update. This requires two constraints.

The first constraint is that a non-faulty client must observe the latest established candidate if such a candidate exists. All three types of quorum systems state it as follows (where Q is an update quorum and Q' is some other quorum):

$$\forall Q, Q' : |Q \cap Q' \setminus B| \geq r$$

The second constraint is that a conflicting candidate (which, as described above, occurs only if there is already an established candidate for the same timestamp) is not observed by any client (non-faulty or faulty). It requires that the replicas that can vote for a conflicting update are fewer than r; the number of such replicas is dependent on the restrictions of the type of quorum system (i.e., dissemination, masking, or opaque). In general, tighter restrictions that decrease this number have the benefit of allowing for smaller values of n in terms of b, but require additional rounds of communication in order to satisfy as discussed below.

Masking quorum systems. Though not used by any of the three protocols that we review in detail, masking quorum systems make relatively simple assumptions: faulty replicas can vote for conflicting candidates, but non-faulty replicas cannot.

Table 1: Threshold quorum systems.

	minimum n	used by e.g.,	conflicting candidate
opaque	$5b+1$	Q/U [1], FaB Paxos [15]	any server
masking	$4b+1$	PASIS [7]	faulty server
dissemination	$3b+1$	BFT [4], SINTRA [3]	no server

Masking quorum systems require $n > 4b$. They provide the property that any two quorums intersect in at least $b + 1$ non-faulty replicas (enough to outnumber the faulty replicas). Quorums are of size $\lceil (n + 2b + 1)/2 \rceil$.

For example, quorums are of size $3b + 1$ if $n = 4b + 1$. An update quorum of size $3b + 1$ from $4b + 1$ means that b replicas do not observe the update (it is possible that these replicas are not faulty). Further, b replicas from the $3b + 1$ may be faulty. This means that only $2b + 1$ non-faulty of the n total replicas are certain to have observed the update. In another quorum of $3b + 1$ responses, it is the case that b may be from the replicas that were not part of the update quorum; however, these replicas cannot vote for a conflicting candidate by assumption. Another b responses may be votes for a conflicting candidate from faulty replicas. Therefore, it is possible that only $b + 1$ votes are for the established candidate. However, these votes outnumber the votes for any conflicting candidate. Therefore, r can be set to $b + 1$ in order to ensure consistency.

An echo protocol like that in Rampart [16] and Phalanx [14] can be used to ensure that non-faulty replicas do not accept conflicting candidates.[2] With an echo protocol, an update requires two phases. First, the client proposes the candidate. If a replica is willing to accept the candidate upon the condition that other non-faulty replicas accept no conflicting candidate, the replica sends a tentative accept (echo) response. A quorum of echo responses proves that no quorum will accept a conflicting update. This is because every two quorums overlap in some positive number of non-faulty replicas, and no non-faulty replica sends echo messages for different conflicting candidates. In the second phase of the update, the client sends the quorum of echo messages along with the candidate. If this is accepted by a quorum of replicas, the update is complete.

Note that the echo protocol by itself does not provide a way to distinguish later between votes for established and conflicting candidates. Even though non-faulty replicas would accept no conflicting candidate, faulty replicas may still forge conflicting candidates. Even having the client digitally sign each candidate would not help, because clients may also be Byzantine-faulty and could provide multiple signed versions of the candidate (the established one to non-faulty replicas, and forged versions to faulty replicas).

Opaque quorum systems. Of the three types of quorum systems discussed here, opaque quorum systems are the only one appropriate for the situation in which some non-faulty replicas might accept candidates that conflict with an established candidate. As such, they can allow for update operations to complete in a single (optimistic) round of communication, even in the face of Byzantine and/or concurrent clients that may e.g., propose conflicting candidates to some non-faulty replicas. In order to provide consistency despite this, opaque quorum systems require that the number of non-faulty replicas in the intersection of any two quorums is larger than the remainder of either quorum.

Opaque quorum systems have the disadvantage of requiring $n > 5b$, which is larger than required by the two other types. Quorums are of size $\lceil (n + 3b + 1)/2 \rceil$ assuming $n = q + b$.

[2]Another way to ensure that non-faulty replicas do not accept conflicting candidates is to use unique, verifiable timestamps for each data item, as done in the PASIS R/W protocol [7]. However, unlike the echo protocol, this method is not used by any of the protocols in our survey.

For example, quorums are of size $4b + 1$ if $n = 5b + 1$. The non-faulty intersection of any two quorums of size $4b + 1$ is $2b + 1$, and, therefore, a majority of any quorum. An update quorum of size $4b + 1$ from $5b + 1$ means that b replicas do not observe the update (it is possible that these replicas are not faulty). Further, b replicas from the $4b + 1$ may be faulty. This means that only $3b + 1$ non-faulty replicas of the n total replicas are certain to have observed the update. In another quorum of $4b + 1$ responses, b responses may be from the replicas that did not observe the update. Furthermore, these replicas may have accepted a conflicting candidate and therefore vote for it. Also, b responses may be from faulty replicas that also vote for the conflicting candidate. Therefore, it is possible that only $2b + 1$ responses come from non-faulty replicas that have the established candidate. However, even in this case, these $2b + 1$ responses strictly outnumber the $2b$ votes for the conflicting candidate. Therefore, r can be set to $2b + 1$.

Dissemination quorum systems. Dissemination quorum systems are the most constrained of the three types of quorum systems. They provide the property that any two quorums overlap in at least one non-faulty replica. Since a quorum may contain more faulty replicas than this (up to b), dissemination quorum systems are suitable only for *self-verifying* data, i.e., data for which a single instance proves that it is an established candidate.

Dissemination quorum systems require $n > 3b$. Quorums are of size $\lceil (n + b + 1)/2 \rceil$.

For example, quorums are of size $2b + 1$ if $n = 3b + 1$. An update quorum of size $2b + 1$ from $3b + 1$ means that b replicas do not observe the update (it is possible that these replicas are not faulty). Further, b replicas from the $2b + 1$ may be faulty. This means that only $b + 1$ non-faulty of the n total replicas are certain to have observed the update. In any quorum of $2b + 1$ responses, b may be from the replicas that did not observe the update. Also, b replicas may be the faulty replicas. Therefore, it is possible that only one response is from a non-faulty replica with the established candidate. However, no replica can vote for a conflicting candidate, and so r can be set to 1.

Liskov and Rodrigues [12] provide a protocol that shows how to ensure self-verifying updates despite Byzantine-faulty or concurrent clients. The protocol uses a modified echo phase similar to that described above for masking quorum systems. The difference is that the quorum of echos is stored by replicas and provided to clients along with votes for a candidate. Non-faulty clients consider a vote for a candidate only if the vote is accompanied by a matching quorum of echos. Therefore, even faulty replicas cannot vote for conflicting candidates because it is impossible to gather a quorum of echos for a conflicting candidate.

3 A Framework for Protocol Operation

We present a framework for comparing and contrasting Byzantine-fault-tolerant state-machine-replication protocols that highlights a number of tradeoffs in protocol design. We introduce the framework with a high-level operational description of the way such protocols work in general.

3.1 Operational Description

Byzantine-fault-tolerant state-machine-replication protocols must ensure replica coordination of non-faulty replicas. Roughly, they do this as follows. Each request is assigned a *permanent sequence number* that exists from the time of assignment through the life of the system and is never changed.[3] We use the term sequence number to indicate that there is a totally-ordered chain of requests; however, the sequence number might be implemented as a logical timestamp [1] or other suitable device. Each permanent sequence number is assigned to a *single* request. Therefore, due to the Byzantine fault model, permanent sequence numbers cannot be assigned by a single replica or client, which might assign the same permanent sequence number to multiple requests.

In order to get a permanent sequence number, a request is first assigned a *proposed sequence number*. Unlike permanent sequence numbers, the same proposed sequence number may be assigned to multiple requests. Therefore, a proposed sequence number can be selected by a single client or replica in isolation. The assignment of permanent sequence numbers takes place by performing an update consisting of the proposed sequence number and request together as a candidate to the replicas, which act as a quorum system. Each non-faulty replica accepts the update only if it has not accepted a different (conflicting) update with the same proposed sequence number. This ensures that each sequence number is assigned only to a single request. In addition, sequence number assignments are preserved by the quorum system; therefore, they are never changed (e.g., during repair, discussed below). As such, a sequence number is permanent if and only if it has been accepted by a quorum of replicas.

The type of quorum system used for accepting proposed sequence numbers implies a lower bound on n in terms of b as discussed in Section 2.3. For example, an opaque quorum system requires at least $5b + 1$ replicas, but can accept a proposed sequence number in a single round of communication. On the other hand, dissemination and masking quorum systems need only $3b + 1$ and $4b + 1$ replicas, respectively, but require two rounds of communication (assuming an echo phase is used) in order to accept proposed sequence numbers.

A non-faulty replica executes a request only after all lower sequence numbers are assigned permanently and it has executed their corresponding requests. If the system is not making progress because a non-faulty replica is waiting to execute a request corresponding to a lower sequence number, action is taken so that the replica obtains the missing request. Individual replicas send responses to the client upon executing the request.

The client determines the correct result from the set of responses received by determining that the result is due to a permanent sequence number assignment and from at least one non-faulty replica. This works because each non-faulty replica that executes a request with a permanent sequence number returns the same, correct result to the client. However, faulty replicas and non-faulty replicas that execute requests without permanent sequence numbers (an optimization employed by some protocols) may return incorrect results.

Because of the use of the quorum system for sequence number assignments, none of the protocols surveyed in this paper become inconsistent in the face of a Byzantine-faulty proposer. How-

[3]We choose the passive voice in this description because details such as which clients/replicas are involved in assigning the sequence number are protocol-specific.

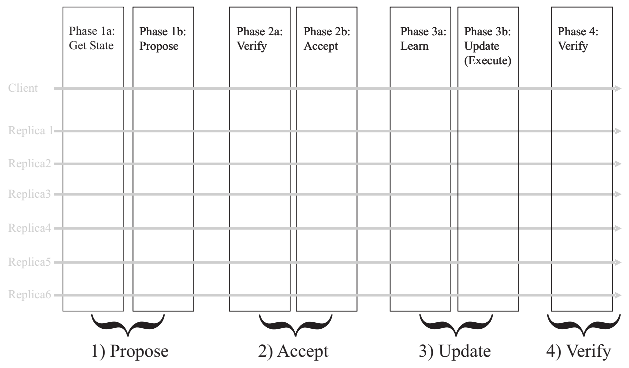


Figure 1: The stages of Byzantine-quorum state-machine-replication protocols.

ever, the protocols each require a *repair* phase in order to continue to be able to make progress.[4] The processes performing repair read from the quorum system to ensure that no permanent sequence number assignments are lost or changed. Repair is discussed further in Section 5.

3.2 The Framework

The framework depicted in Figure 1 consists of four high-level phases totaling seven sub-phases. In phase 1, a proposed sequence number is chosen for the client's request and sent to (at least) a quorum of replicas. In phase 2, a quorum of replicas accepts the proposed sequence number (doing so might involve an echo phase as described in Section 2.3). If no quorum accepts the proposed sequence number assignment, a new proposal must be tried and the system may require repair (see Section 5). In phase 3, the request is executed according to the sequence number, and in phase 4 the client chooses the correct result. Phases 1a, 2a and 3a can be viewed as optional, as they are omitted by some protocols; however, omitting them has implications as discussed below. The remainder of this section explores each of the phases of the framework in greater detail.

Phase 1: Propose. Phase 1 is where a proposed sequence number is selected for a client request; this is done by a *proposer*, which, dependent on the protocol, is either a replica or a client. In some protocols, it is possible that the state of the system has been updated without the knowledge of the proposer (for example due to contention by multiple proposers). In this case the proposer may first need to retrieve the up-to-date state of the system, including earlier permanent sequence number assignments. This is the purpose of phase 1a. Phase 1b is where the proposed sequence number and request are sent to (at least) a quorum of replicas.

Phase 2: Accept. Phase 2 is where the proposed sequence number is either accepted or rejected. As discussed in Section 2.3, depending on the type of quorum system, this may require a round of communication (corresponding to an echo phase) for the purpose of ensuring that non-faulty

[4]In this paper, we do not classify protocols based on their repair phases. Therefore, we do not distinguish between BFT and Sintra [3], for example.

replicas do not accept different conflicting proposals for the same sequence-number. If it requires this round of communication (phase 2a), the protocol is said to employ a *pessimistic accept* phase, otherwise, it is said to employ an *optimistic accept* phase. In phase 2b, the sequence number assignment becomes permanent if and only if a quorum of replicas accepts it.

The primary benefit of an optimistic accept phase is that one round of communication (phase 2a) involving at least a quorum of replicas is avoided. The disadvantage is the need for an opaque quorum system, which requires $n > 5b$.

Phase 3: Update. Phase 3 is where the update is applied, typically resulting in the execution of the requested operation. Like phase 2, this phase can be either *pessimistic* (requiring phase 3a), or *optimistic* (omitting phase 3a). Phase 3a allows the execution replicas to learn that the sequence number assignment has been accepted by a quorum of replicas (and, as such, has become permanent) before performing the update. If this is not done, the sequence number assignment may change and the request may need to be executed again.

Since an optimistic update phase requires no additional round of communication before execution, it can lead to better performance. The disadvantages are that, as described below, clients must wait for a quorum of responses instead of just $b+1$ to ensure that the sequence number assignment is permanent, and that computation may be wasted in the case that the proposed sequence number does not become permanent.

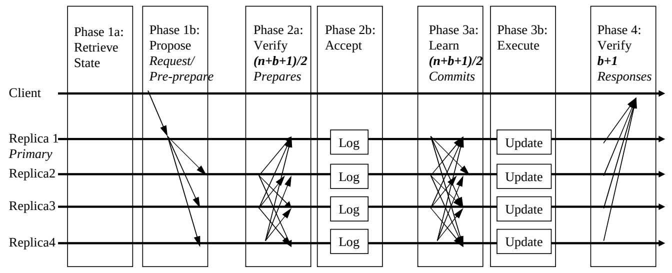
Phase 4: Verify. Phase 4 is where the client receives a set of responses. The client must verify that the update was based on a permanent sequence number assignment and performed by at least one non-faulty replica (in order to ensure that the result is correct). In general, this requires waiting for a quorum of identical responses indicating the sequence number, where the size of the quorum is dependent on the quorum system construction. However, if phase 3 is pessimistic, then no non-faulty replica will execute an operation unless the assignment is permanent. In this case, the client can rely on non-faulty replicas to verify that the sequence number is permanent, and so clients need wait for only $b + 1$ identical responses.

4 The Protocols

The protocols that we consider in this section are BFT [4], FaB Paxos [15], and Q/U [1]. As summarized in Table 2, these protocols span the four possible combinations of pessimistic and optimistic accept and update phases (i.e., phases 2 and 3). We classify the protocols accordingly.

4.1 Pessimistic Accept and Update

BFT (without optimizations). BFT [4] is a prominent example of a Byzantine-fault-tolerant state-machine-replication protocol that employs a pessimistic accept phase (with a dissemination quorum system) and a pessimistic update phase. As such, it involves communication in all four phases of our framework. Figure 2 shows the phases of an update request of the BFT protocol in



Phase 1a: Retrieve State
Phase 1b: Propose *Request/ Pre-prepare*
Phase 2a: Verify ***(n+b+1)/2*** *Prepares*
Phase 2b: Accept
Phase 3a: Learn ***(n+b+1)/2*** *Commits*
Phase 3b: Execute
Phase 4: Verify ***b+1*** *Responses*
Client
Replica 1 *Primary*
Replica2
Replica3
Replica4
Log
Update

Figure 2: BFT.

the fault-free case. The operation is very similar to a protocol presented by Bracha and Toueg [2] also used by other protocols, e.g., [17, 3, 20, 10].

In the common case, there is a single proposer (called the *primary*) that itself is a replica; therefore, phase 1a is unnecessary—the proposer already knows the next unused sequence number. In phase 1b, the proposer unilaterally chooses a proposed sequence number for the request (a non-faulty proposer should choose the next unassigned sequence number) and sends the request along with the proposal to the other replicas in a message called PRE-PREPARE.

The verification done in phase 2a is equivalent to an echo protocol (though the responses are sent directly to all other replicas instead of through the proposer). This guarantees that non-faulty replicas do not accept different candidates with the same proposed sequence numbers. Each replica other than the proposer sends a PREPARE (i.e., echo) message including the proposal to all other replicas. If a replica obtains a quorum of matching PREPARE and PRE-PREPARE messages (including its own), it is guaranteed that no other non-faulty replica will accept a proposal for the same sequence number but with a different request. Such a replica is called *prepared*. A prepared replica accepts the request in phase 2b, and stores the quorum of PREPARE and PRE-PREPARE messages. The sequence number assignment is permanent if and only if a quorum of replicas accept the proposed sequence number in phase 2b.

In phase 3a, prepared replicas send COMMIT messages to all other replicas. A COMMIT message includes the quorum of PREPARE and PRE-PREPARE messages (i.e., echos) so that the sequence number assignment is self-verifying (which is necessary for a dissemination quorum system as discussed in Section 2.3). Because the update phase is pessimistic, in phase 3a, replicas wait to receive a quorum of COMMIT messages to make certain that the sequence number assignment is

Table 2: Protocol Classification

	Update	
Accept	optimistic	pessimistic
optimistic (opaque quorum)	Q/U, FaB Paxos w/ TE	FaB Paxos
pessimistic (dissemination quorum)	BFT w/ TE	BFT

permanent. Having received a quorum of matching COMMIT messages for sequence number i, a replica executes the request only after executing all requests corresponding to permanent sequence number assignments $1 .. i-1$.

Since BFT employs a pessimistic update phase, the client waits for only $b+1$ identical results in phase 4. This does not require waiting for more than $2b+1$ replies.

4.2 Optimistic Accept, Pessimistic Update

One way to avoid a round of communication is to employ an optimistic accept phase (i.e., to skip phase 2a).

FaB Paxos. In relation to our framework, FaB Paxos [15], can be viewed as BFT with an optimistic accept phase (provided by the use of an opaque quorum system). It is seen in the lower half of Figure 3. Compared with BFT, FaB Paxos uses larger quorums and requires more replicas, but saves a round of communication.

4.3 Pessimistic Accept, Optimistic Update

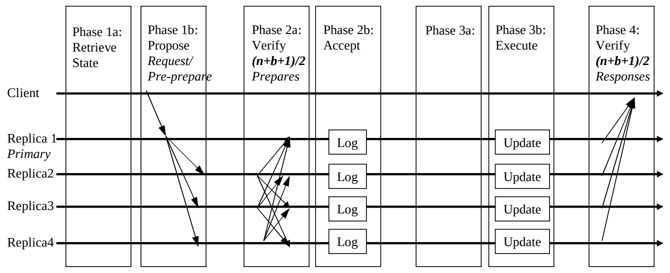
Another way to avoid a round of communication is to employ an optimistic update phase (i.e., to skip phase 3a).

BFT w/ Tentative Execution. Castro and Liskov [4] detail an optimistic update optimization for BFT called tentative execution (TE). In tentative execution, phase 3a is omitted; however, the dissemination quorum system remains the same. Compared with unoptimized BFT, tentative execution saves a round of communication. However, since a response from a non-faulty replica no longer necessarily corresponds to a permanent sequence number assignment, the client must wait for a quorum of identical responses in phase 4. In addition, replicas that execute a request corresponding to a non-permanent sequence number assignment that later changes (e.g., due to repair) may need to re-execute the request later.

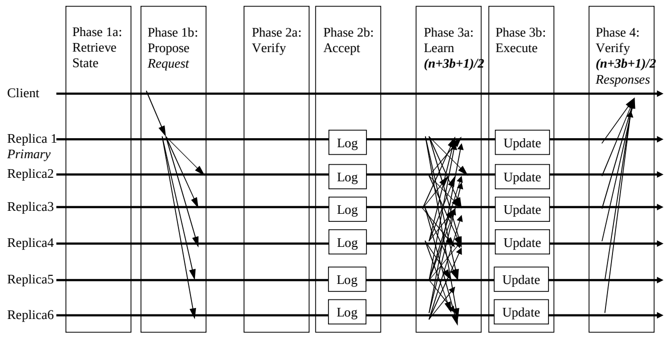
Figure 3 shows the stages of the BFT protocol with the tentative-execution optimization, compared with FaB Paxos [15].

4.4 Optimistic Accept and Update

Two protocols use both optimistic accept and optimistic update phases. Q/U [1] is a Byzantine-fault-tolerant state-machine-replication protocol based on opaque quorum systems. FaB Paxos, which normally employs only an optimistic accept phase as described above, can, like BFT, also employ an optimistic update optimization known as tentative execution. Since both Q/U and FaB Paxos with tentative execution skip phase 2a, neither protocol can use fewer than $5b+1$ replicas. In addition, since they also skip phase 3a, both protocols require the client to wait for a quorum of identical responses in phase 4 to make certain that the result is based on a permanent sequence


Phase 1a: Retrieve State
Phase 1b: Propose *Request/ Pre-prepare*
Phase 2a: Verify ***(n+b+1)/2*** *Prepares*
Phase 2b: Accept
Phase 3a:
Phase 3b: Execute
Phase 4: Verify ***(n+b+1)/2*** *Responses*
Client
Replica 1 *Primary*
Replica2
Replica3
Replica4
Log
Update

(BFT w/ Tentative Execution)



(FaB Paxos)

Figure 3: Optimistic update (BFT w/ tentative-execution optimization), compared to optimistic accept (FaB Paxos).

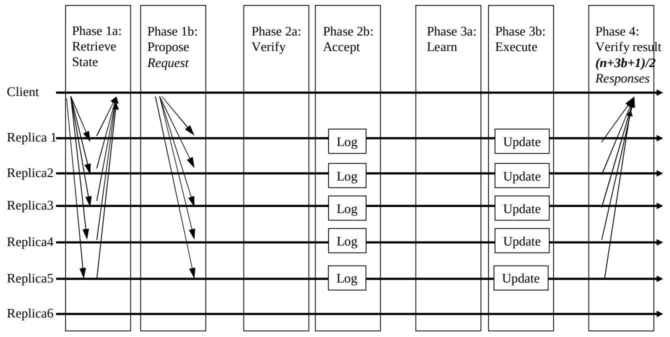
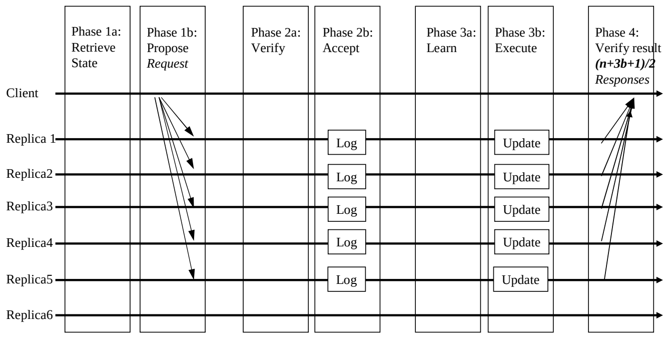


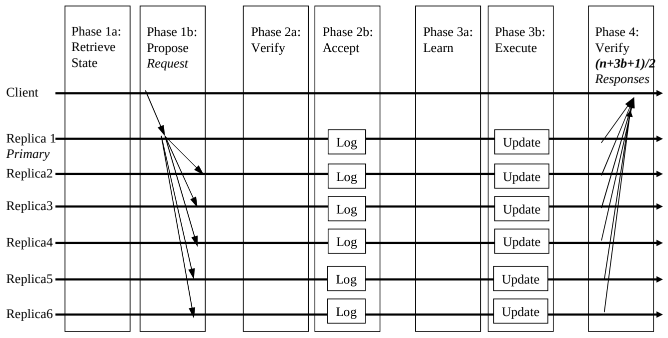
Figure 4: The Q/U protocol (optimistic accept and update).

number assignment; this quorum is larger than the quorums in systems that employ pessimistic accept.

Q/U. Figures 4 and 5 show the Q/U protocol. In phase 1, clients act as proposers and directly issue requests to the replicas. Since there are multiple proposers, a proposer may not know the next sequence number (implemented as a logical timestamp). Therefore, the client first retrieves the update history (called a replica history set) from a quorum of replicas (phase 1a). A quorum of replica history sets is called an object history set. It identifies the latest completed update, and, therefore, the sequence at which the next update should be applied. The client sends the object



(Q/U w/ Pipelined Optimization)



(FaB Paxos w/ Tentative Execution)

Figure 5: Optimistic accept and update in Q/U and FaB Paxos (both optimized).

history set along with the request to a quorum of replicas (phase 1b). In phase 2b, each replica verifies that it has not executed any operation more recent than that which is reflected in the the object history set, and then accepts the update. Having accepted the update, the acceptor executes the request (phase 3b).[5] Because Q/U is an optimistic execution protocol (it skips phase 3a), the client must wait for a quorum of responses in phase 4.

In a pipelined optimization of Q/U (shown in Figure 5), clients cache object history sets after each operation. As such, clients can avoid phase 1a if no other clients have since updated the system.

FaB Paxos w/ Tentative Execution. The tentative-execution optimization for FaB Paxos works as it does in BFT—a replica executes the request upon accepting the sequence number assignment for it in phase 2b (assuming it has also executed the requests corresponding to all earlier permanent sequence numbers). Because this sequence number assignment may never become permanent, it may need to be rolled back. Therefore, clients must wait for a quorum of identical responses in phase 4. Figure 5 highlights the similarities between Q/U with the pipelined optimization described above and FaB Paxos with the tentative-execution optimization.

[5]Technically, the log step in phase 2b happens after the update step in phase 3b. However, as there are no messages between these steps, from an external perspective these steps can be viewed as an atomic unit here.

5 Other Tradeoffs

BFT and FaB Paxos use a single proposer (the primary), and so can omit phase 1a. On the other hand, Q/U allows clients to act as proposers, and therefore requires phase 1a. The use of a single proposer has two potential performance advantages. First, a client sends only a single request to the system (in the common case) as opposed to sending the request to an entire quorum. Therefore, since the single proposer is likely physically closer than the clients to the replicas, the use of a primary might be more efficient, e.g., on a WAN with relatively large message delays. Another advantage is that request-batching optimizations can be employed because the primary is aware of requests from multiple clients. However, the use of a primary involves an extra message delay (for the request to be forwarded to the primary).

Because a proposer may be Byzantine-faulty, a repair phase may be necessary in order for the service to make progress in the presence of faults. In systems such as BFT and FaB Paxos that use a dedicated proposer, the repair phase is used to choose a new proposer. In Q/U, this phase may also result from concurrent client updates, and is used to make sure that non-faulty replicas no longer have conflicting sequence number assignments. In BFT and FaB Paxos, repair is initiated by non-faulty replicas that have learned of some request but not have executed it after a specified length of time (*proactive repair*). In Q/U, repair is initiated by a client that has learned that the system is in a state from which no update can be completed due to conflicting proposed sequence number assignments (*need-based repair*). Because it is based on timeouts, proactive repair might sometimes be executed when it is not actually of help, e.g., when the network is being slow but the primary is not faulty.

6 Related Work

Our framework for comparing protocols finds inspiration in the framework of Wiesmann et al. [19] who compare (non-Byzantine-fault-tolerant) replication techniques for databases and distributed systems.

We distinguish state-machine-replication protocols, even those that use quorum systems, from other protocols for quorum systems [13, 7] that are designed to support read-write shared-variable semantics with idempotent operations. In protocols for read-write shared variables, updates are idempotent, and, therefore, do not necessarily all have to be executed. That is, more recent updates can be applied as soon as they arrive. If older updates arrive later, they do not need to be applied to modify the state, as the more recent updates make the older state unnecessary. In contrast, state-machine-replication protocols must guarantee that all updates are applied in order, because if a replica skips an update applied at other replicas, it might have inconsistent state as a result.

An alternative approach [16, 8] to state machine replication is to assume that all non-responsive replicas are faulty and, therefore, to remove them from the set of all replicas. However, such an approach requires the replicas to agree on the set of faulty replicas.

Martin and Alvisi [15] prove that any consensus protocol with two or fewer rounds of communication requires at least $5b + 1$ replicas. Our observations support this, and provide insight for why this is the case for Byzantine-fault-tolerant state-machine-replication protocols (which imply consensus).

7 Conclusion

We have presented a framework of high-level, logical phases for the comparison of Byzantine-fault-tolerant state-machine-replication protocols that guarantee consistency in an asynchronous environment. Our framework centers on the use of Byzantine quorum systems in each protocol, highlighting tradeoffs made by the protocols in terms of the number of replicas required, the number of faults that can be tolerated, and the number of rounds of communication required.

Since the original drafting of this document, new protocols (e.g., [5, 9]) have been introduced. We hope that our framework will facilitate understanding the intuition behind these protocols.

8 Acknowledgments

Thanks to David O'Hallaron and Michael Reiter for helpful feedback on drafts of this paper.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Symposium on Operating Systems Principles*, October 2005.

[2] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[3] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *International Conference on Dependable Systems and Networks*, pages 167–176, 2002.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, 1999.

[5] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, Nov 2006.

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

[7] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks*, June 2004.

[8] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Hawaii International Conference on System Sciences*, pages 317–326, 1998.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Symposium on Operating Systems Principles*, pages 45–58, New York, NY, USA, 2007. ACM.

[10] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks*, page 575, June–July 2004.

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

[12] B. Liskov and R. Rodrigues. Byzantine clients rendered harmless. Technical Report MIT-LCS-TR-994, MIT Laboratory for Computer Science, July 2005.

[13] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[14] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Symposium on Reliable Distributed Systems*, pages 51–58, October 1998.

[15] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[16] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Conference on Computer and Communication Security*, pages 68–80, November 1994.

[17] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Symposium on Operating Systems Principles*, 2001.

[18] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[19] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *International Conference on Distributed Computing Systems*, pages 264–274, April 2000.

[20] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Symposium on Operating Systems Principles*, pages 253–267, October 2003.