Chapter 10
Selected Results from the Latest Decade of Quorum Systems Research

Michael G. Merideth and Michael K. Reiter

Abstract Over the past decade, work on quorum systems in non-traditional scenarios has facilitated a number of advances in the field of distributed systems. This chapter surveys a selection of these results including: Byzantine quorum systems that are suitable for use when parts of the system cannot be trusted; algorithms for the deployment of quorum systems on wide area networks so as to allow for efficient access and to retain load dispersion properties; and probabilistic quorum systems that yield benefits for protocols and applications that can tolerate a small possibility of inconsistency. We also present a framework grounded in Byzantine quorum systems that can be used to explain, compare, and contrast several recent Byzantine fault-tolerant state-machine and storage protocols. The framework provides 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.

10.1 Introduction

Given a universe U of servers, a *quorum system* over U is a collection $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ such that each $Q_i \subseteq U$ and

$$|Q \cap Q'| > 0 \tag{10.1}$$

for all $Q, Q' \in \mathcal{Q}$. Each Q_i is called a *quorum*. The intersection property (10.1) makes quorum systems a useful primitive for coordinating actions in a distributed system. For example, if each write is performed at a quorum of servers, then a client who reads from a quorum will observe the last written value. Because of their utility in such applications, quorum systems have a long history in distributed computing.

In this paper, we survey a number of advances that have occurred in using quorum systems to implement distributed services in the last ten years or so. These recent advances derive primarily from the use of quorum systems in non-traditional scenarios. We begin by focusing on their use in systems that suffer Byzantine faults [15], which introduce the possibility of not being able to trust all servers in the intersec-

B. Charron-Bost, F. Pedone, and A. Schiper (Eds.): Replication, LNCS 5959, pp. 185–206, 2010.
© Springer-Verlag Berlin Heidelberg 2010

tion of two different quorums. In Section 10.2, we summarize the basic properties of Byzantine quorum systems, including the minimum universe sizes they require and their best-case load-dispersing properties. We also summarize weaker, probabilistic variants of Byzantine quorum systems that can offset these costs in some cases.

The use of distributed protocols in wide-area networks (e.g., in support of edge computing or content distribution) motivates another set of results that we summarize in Section 10.3. Namely, these results focus on how to position servers and quorums in networks so as to minimize the delays that clients suffer by accessing them. Interestingly, there can be a tension between achieving good load dispersion and having low delay for accessing quorums because, when seeking to balance the load, a client might be required to bypass a nearby but heavily loaded server for another that is further away but more lightly loaded. We briefly summarize some results that have been developed to balance this trade-off.

We then return to Byzantine quorums in Section 10.4, but with an eye toward their use in Byzantine-fault-tolerant state-machine-replication protocols. We present a framework grounded in Byzantine quorum systems that can be used to explain, compare, and contrast several recent such protocols. The Byzantine quorum systems framework particularly helps to elucidate the commonalities among these protocols, and we further believe that this framework should be useful in explaining future such protocols.

10.2 Quorum Systems for Byzantine Faults

In systems that may suffer Byzantine faults, the intersection property (10.1) is typically not adequate as a mechanism to enable consistent data access. Because (10.1) requires only that the intersection of quorums be non-empty, it could be that two quorums intersect in only a single server, for example. In a system in which up to $b > 0$ servers might suffer Byzantine faults, this single server might be faulty and, consequently, could fail to convey the last written value to a reader, for example.

For this reason, Malkhi and Reiter [18] proposed various ways of strengthening the intersection property (10.1) so as to enable quorum systems to be used in Byzantine environments. Our presentation here assumes that there is an (unknown) set B of up to b servers that are faulty. Malkhi and Reiter considered three stronger properties as alternatives. All of these alternatives require that for some $Q \in \mathcal{Q}$,

$$|B \cap Q| = 0. \tag{10.2}$$

In words, for any set of server faults B, there is a quorum Q that does not intersect B. Without this constraint, the faulty servers could prevent progress in the system by simply not responding, since a client typically requires a response from a full quorum of servers to make progress.

The first and simplest alternative to (10.1) is

$$|Q \cap Q' \setminus B| > 0 \tag{10.3}$$

for all $Q, Q' \in \mathcal{Q}$. That is, the intersection of any two quorums contains at least one non-faulty server. For example, if a non-faulty client acquires a lock by accessing Q, then any subsequent client that attempts to acquire this lock at another quorum Q' will notice that some other client already requested the lock, regardless of the behavior of the faulty servers. Quorum systems satisfying (10.3) are called *dissemination* quorum systems.

The second alternative to (10.1) is

$$|Q \cap Q' \setminus B| > |Q' \cap B| \tag{10.4}$$

for all $Q, Q' \in \mathcal{Q}$. In words, the intersection of any two quorums contains more non-faulty servers than the faulty ones in either quorum. As such, the responses from these non-faulty servers will outnumber those from faulty ones. These quorum systems are called *masking* quorum systems.

Finally, the third alternative to (10.1) is

$$|Q \cap Q' \setminus B| > |(Q' \cap B) \cup (Q' \setminus Q)| \tag{10.5}$$

for all $Q, Q' \in \mathcal{Q}$. In words, the number of non-faulty servers in the intersection of Q and Q' (i.e., $|Q \cap Q' \setminus B|$) exceeds the number of faulty servers in Q' (i.e., $|Q' \cap B|$) together with the number of servers in Q' but not Q. The rationale for this property can be seen by considering the servers in Q' but not Q as "outdated", in the sense that if Q was used to perform a write to the system, then those servers in $Q' \setminus Q$ are unaware of the write. As such, if the faulty servers in Q' behave as the outdated ones do, their behavior (i.e., their responses) will dominate that from the non-faulty servers in the intersection ($Q \cap Q' \setminus B$) unless (10.5) holds. Quorum systems satisfying (10.5) are called *opaque* quorum systems.

The increasingly stringent properties (10.3) – (10.5) come with costs in terms of the smallest system sizes that can be supported while tolerating a number b of faults, as shown in the following theorem.

Theorem 10.1 ([18]). *Let $n = |U|$, and let $\mathcal{Q}$ be a quorum system over U.*

- *If $\mathcal{Q}$ is a dissemination quorum system* (10.3)*, then $b < n/3$.*
- *If $\mathcal{Q}$ is a masking quorum system* (10.4)*, then $b < n/4$.*
- *If $\mathcal{Q}$ is an opaque quorum system* (10.5)*, then $b < n/5$.*

10.2.1 Access Strategies and Load

Naor and Wool [24] introduced the notion of an *access strategy* by which clients select quorums to access. An access strategy $p : \mathcal{Q} \to [0,1]$ is simply a probability distribution on quorums, i.e., $\sum_{Q \in \mathcal{Q}} p(Q) = 1$. Intuitively, when a client accesses the system, it does so at a quorum selected randomly according to the distribution p.

The formalization of an access strategy is useful as a tool for discussing the load dispersing properties of quorum systems. Specifically, it permits us to talk about the probability with which a server is accessed in any given quorum access, i.e.,

$$\ell_p(u) = \sum_{Q \ni u} p(Q), \tag{10.6}$$

and then the maximally loaded server under a given access strategy p, i.e.,

$$\mathscr{L}_p(\mathscr{Q}) = \max_{u \in U} \ell_p(u). \tag{10.7}$$

Finally, this enables us to define the *load* [24] of a quorum system as

$$\mathscr{L}(\mathscr{Q}) = \min_p \mathscr{L}_p(\mathscr{Q}). \tag{10.8}$$

In words, $\mathscr{L}(\mathscr{Q})$ is the probability with which the busiest server is accessed in a client access, under the best possible access strategy p.

Theorem 10.2 ([24, 19]). *Let $\mathscr{Q}$ be a quorum system of U, $n = |U|$.*

- *If $\mathscr{Q}$ is a regular quorum system* (10.1)*, then* $\mathscr{L}(\mathscr{Q}) \geq \sqrt{\frac{1}{n}}$.
- *If $\mathscr{Q}$ is a dissemination quorum system* (10.3)*, then* $\mathscr{L}(\mathscr{Q}) \geq \sqrt{\frac{b}{n}}$.
- *If $\mathscr{Q}$ is a masking quorum system* (10.4)*, then* $\mathscr{L}(\mathscr{Q}) \geq \sqrt{\frac{2b}{n}}$.
- *If $\mathscr{Q}$ is an opaque quorum system* (10.5)*, then* $\mathscr{L}(\mathscr{Q}) \geq \frac{1}{2}$.

All of these lower bounds are tight, in the sense that there are known quorum systems $\mathscr{Q}$ (of the respective types) and access strategies p that meet these lower bounds asymptotically [24, 18, 19]. The last of the results listed in Theorem 10.2 is particularly unfortunate, since it shows that systems that utilize opaque quorum systems cannot effectively disperse processing load across more servers (i.e., by increasing n). We see one way to address this in Section 10.2.2.

10.2.2 Probabilistic Quorum Systems

The lower bounds on universe size presented in Theorem 10.1 present an obstacle to the use of quorum systems in practice. Moreover, the constant lower bound on the load of opaque quorum systems imposes a theoretical limit on the scalability of systems that use them.

One way to circumvent these lower bounds is to relax the quorum intersection properties themselves, and one such way is to ask them to hold only with high probability. More specifically, we can relax any of (10.3), (10.4) and (10.5) to hold only with probability $1 - \varepsilon$, where probabilities are taken with respect to the selection of quorums according to an access strategy p [20, 22]. This technique yields masking quorum system constructions tolerating $b < 2.62/n$ and opaque quorum system constructions tolerating $b < 3.15/n$. These bounds hold in the sense that for any $\varepsilon > 0$ there is an n_0 such that for all $n > n_0$, the required intersection property ((10.4) or (10.5) for masking and opaque quorum systems, respectively) holds with probability at least $1 - \varepsilon$. Unfortunately, this technique alone does not materially improve the load of these systems [20].

An additional modification, however, can improve this situation even further. Merideth and Reiter [23] propose the use of *write markers* for further improving the resilience and load of these systems. Intuitively, in each write access to a quorum of servers, a write marker is placed at the accessed servers in order to evidence the quorum used in that access. This write marker identifies the quorum used; as such, faulty servers not in this quorum cannot respond to subsequent quorum accesses as though they were. By using this method to constrain how faulty servers can collaborate, the resilience of probabilistic masking quorum systems can be improved to $b < n/2$, and the resilience of probabilistic opaque quorum systems can be improved to $b < n/2.62$. In addition, probabilistic opaque quorum systems with load $O(b/n)$ can also be achieved via this technique, breaking the constant lower bound on load for opaque systems.

In addition to introducing a probability ε of error, probabilistic quorum systems require mechanisms to ensure that accesses are performed according to the required access strategy p, if the clients cannot be trusted to do so (e.g., see [22, 23] for such mechanisms). Moreover, the communication network must be assumed not to bias different clients' accesses toward different (and not adequately intersecting) quorums, and so these approaches require a stronger system model than do strict quorum systems.

10.3 Minimizing Delays of Quorum Accesses

Before explaining the use of quorum systems in protocols in Section 10.4, we first consider the performance impacts of quorums generically. The performance implications for a protocol utilizing quorums primarily lie in the costs of accessing a full quorum, in addition to the processing delays incurred at the servers in the accessed quorum. Most early research in quorum systems assumed an abstract setting that does not ascribe any costs or delays to quorum accesses or heterogeneous limits on the processing capabilities of different servers. Recent research, however, has made strides in taking these into account.

To frame this progress, suppose that the communication network can be represented by an undirected graph $G = (V,E)$, where each edge $e \in E$ has a positive "length" $l(e)$. This induces a distance function $d : V \times V \to \mathbb{R}^+$ obtained by setting $d(v,v')$ to be the sum of lengths of the edges comprising the path from v to v' that minimizes this sum (i.e., the shortest path). This can naturally be extended to a distance $\delta : V \times 2^V \to \mathbb{R}^+$ defined as $\delta(v,Q) = \max_{v' \in Q} d(v,v')$.

In this context, several authors have made progress on placing servers U at graph nodes V and defining a quorum system $\mathcal{Q}$ over them so as to optimize the costs of clients (typically the elements of V) accessing quorums.

- Fu [5] introduced the following problem: Find a quorum system $\mathcal{Q}$ over universe V to minimize $\text{avg}_{v \in V} \min_{Q \in \mathcal{Q}} \delta(v,Q)$, i.e., the average cost for each client to reach its *closest* quorum. That work presented optimal algorithms when G has certain characteristics, e.g., G is a tree, cycle or cluster network.

- For general networks, Tsuchiya et al. [30] gave an efficient algorithm to find $\mathcal{Q}$ so as to minimize $\max_{v \in V} \min_{Q \in \mathcal{Q}} \delta(v,Q)$, i.e., the maximum cost any client pays to reach its closest quorum.
- Kobayashi et al. [12] presented a branch-and-bound algorithm to produce a quorum system $\mathcal{Q}$ to minimize $\text{avg}_{v \in V} \min_{Q \in \mathcal{Q}} \delta(v,Q)$. Their algorithm could be evaluated only on topologies with up to 20 nodes due to its exponential running time, and they conjectured that the problem of finding a quorum system to minimize $\text{avg}_{v \in V} \min_{Q \in \mathcal{Q}} \delta(v,Q)$ is NP-hard.
- Lin [16] showed that designing a quorum system $\mathcal{Q}$ to minimize $\text{avg}_{v \in V} \min_{Q \in \mathcal{Q}} \delta(v,Q)$ is indeed NP-hard, and gave a 2-approximation for the problem.

None of these works consider the load of the quorum system; indeed, Lin's 2-approximation [16] yields a quorum system with very high load: the output consists of only a single quorum containing a single node v minimizing $\sum_{v' \in V} d(v,v')$. Such a solution is not very desirable, since it eliminates the advantages (such as load dispersion and fault tolerance) of any distributed quorum-based algorithm. More generally, there is a tension between achieving low load and low quorum access delay, in that in order to reduce the load on a nearby server, it might be necessary for a client to access quorums that incur greater network latency but that have less heavily loaded servers.

Methods to balance this tension have been recently studied under the rubrics of "quorum placement" [10, 7] and "quorum deployment" [6]. In these frameworks, a quorum system $\mathcal{Q}$ over a universe of "logical" servers U is provided as an input to the problem, along with the graph $G = (V,E)$ that represents the network. The goal in these problems is to find a placement $f : U \to V$ that minimizes some notion of access delay. Gilbert and Malewicz [6] consider a variation of the problem in which $|\mathcal{Q}| = |V| = |U|$ and each client accesses only a single, distinct quorum. In this setting, they show

Theorem 10.3 ([6]). *There is a polynomial-time algorithm to compute bijections* $f : U \to V$ *and* $q : V \to \mathcal{Q}$ *that minimize* $\text{avg}_{v \in V} \gamma(v, f(q(v)))$*, where* $f(Q) = \{f(u)\}_{u \in Q}$ *and* $\gamma(v,Q) = \sum_{u \in Q} d(v,u)$.

Gupta et al. [10] consider a version of the problem in which a capacity $\mathsf{cap}(v)$ for each $v \in V$ and an access strategy p are provided as inputs. They extend the problem formulation to incorporate a placement f into the notions of load and access delay, i.e.,

$$\ell_f(v) = \sum_{u : f(u) = v} \ell_p(u),$$
$$\delta_f(v,Q) = \max_{u \in Q} d(v, f(u)).$$

They seek a placement f so that $\ell_f(v) \leq \mathsf{cap}(v)$ for all $v \in V$, and that minimizes the *expected* quorum access delay

$$\Delta_f(v) = \sum_{Q \in \mathcal{Q}} p(Q)\delta_f(v,Q),$$

averaged over all $v \in V$, i.e., that minimizes

$$\text{avg}_{v \in V} \Delta_f(v).$$

Specifically, Gupta et al. show

Theorem 10.4 ([10]). *There is a polynomial-time algorithm to compute, for any $\alpha > 1$, a placement f with $\ell_f(v) \leq (\alpha+1)\mathsf{cap}(v)$ for all $v \in V$ and for which*

$$\text{avg}_{v \in V} \Delta_f(v) \leq \frac{5\alpha}{\alpha - 1} \text{avg}_{v \in V} \Delta_{f^*}(v)$$

for any capacity-respecting solution f^* (i.e., any f^* satisfying $\ell_{f^*}(v) \leq \mathsf{cap}(v)$ for all $v \in V$).

They also provide exact polynomial-time solutions for quorum systems $\mathcal{Q}$ of certain types when the access strategy p is load-optimal for those systems, but show that solving the problem for general quorum systems and access strategies is NP-hard. Oprea and Reiter [26, 25] experiment with the algorithm specified in Theorem 10.4 in wide-area topologies, including exploring variations in which different clients can employ different access strategies. Golovin et al. [7] extend the quorum placement framework to minimize network congestion arising from quorum accesses, again while respecting capacity constraints on the processing capabilities of nodes.

10.4 Uses of Byzantine Quorums in Protocols

Byzantine variations on quorum systems have provided a basis for explaining existing agreement protocols (e.g., [29, 9]) and have contributed to the design of new ones. In this section, we present a framework by which such protocols, and specifically their use of different types of Byzantine quorum systems, can be compared.

More specifically, we consider two types of protocols in this section. The first are protocols for implementing a service offering *read* and *overwrite* operations on objects, where an overwrite operation overwrites the previous value of the object with a new value; we refer to these protocols as simple *read-overwrite* protocols.[1] The second type of protocol we consider enables the implementation of arbitrary types of operations on objects, provided that those operations are deterministic. These protocols are typically called state-machine-replication protocols, since the operations provided by the service are mapped to the state transitions of a deterministic state machine, and each service replica runs a copy of the state machine, conceptually.

Both types of protocols coordinate the treatment of client requests across multiple servers while guaranteeing *consistency*—the illusion of a single centralized service.

[1] In other contexts, such protocols are often called read-write protocols. Here, we use the term *overwrite* to distinguish from writes that occur in quorum systems.

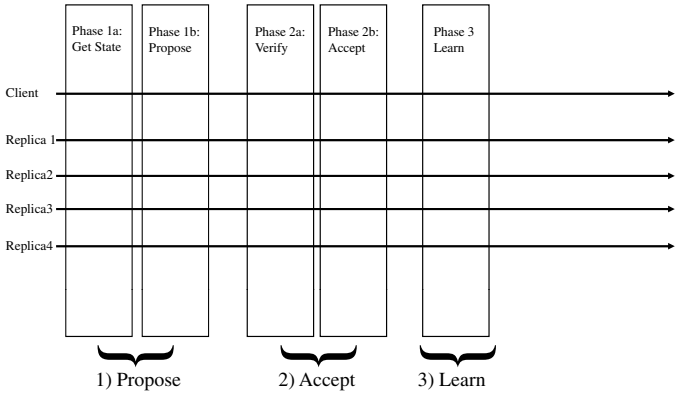

Fig. 10.1 The phases for an overwrite in a read-overwrite protocol.

To do so, these protocols order requests so that non-faulty clients perceive them to be executed in the same order. These protocols 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).

These protocols maintain consistency by using Byzantine quorum systems (see Section 10.1), 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. As we show in the rest of this section, the common use of Byzantine quorum systems in these protocols also provides a basis for their comparison. We present frameworks for both read-overwrite and state-machine-replication protocols in which we identify the roles that Byzantine quorum systems play in the protocols, and the implications that result.

10.4.1 Read-Overwrite Protocols

Figure 10.1 shows the phases for an overwrite operation in a typical read-overwrite protocol. The first phase, which we call *propose*, is where the client submits the operation. In some protocols, such as the PASIS-RW protocol [8] described below, the client must first determine the current state of the system, e.g., in order to determine what the next sequence number should be. If necessary, this is done in phase 1a. In phase 1b, the client sends the operation to at least a quorum of servers. In phase 2, servers perform any necessary verification, and accept the operation. In phase 3, the client knows that the operation is complete once it has been accepted by a quorum of servers.

Figure 10.2 shows the phases for a read operation in a typical read-overwrite protocol. To perform an operation, a client contacts at least a quorum of servers in phase 1. For the service to provide linearizable semantics [11], the client must be certain that the value it reads is the result of a complete overwrite. If not, then

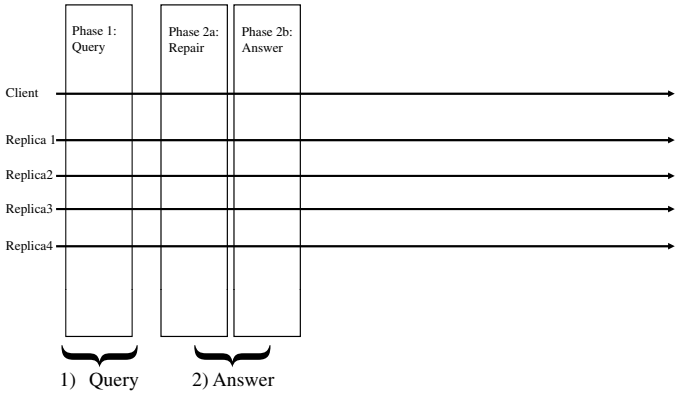

Fig. 10.2 The phases for a read in a read-overwrite protocol.

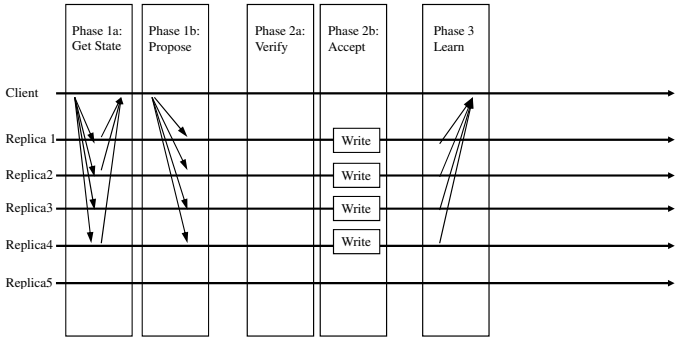

Fig. 10.3 The phases for an overwrite in the PASIS-RW protocol.

another client performing a later read might read an earlier or different value. Therefore, if the client is uncertain whether the overwrite is complete, the client repairs it, e.g., by completing it at a quorum of servers in phase 2a. Finally, if the client receives the same value from at least a quorum of servers in phase 2b, then it knows the overwrite is complete. Therefore, the client uses this value as the result of the read operation.

Use of Masking Quorum Systems

To make this more concrete, consider the PASIS-RW protocol [8]. Figures 10.3 and 10.4 show a simplified view of the protocol that omits details such as erasure coding that are discussed in [8].

The PASIS-RW protocol uses a masking quorum system that satisfies (10.4). In an overwrite operation, the client first determines the most recent timestamp by querying at least a quorum in phase 1a. Let us assume that the system does not need repair. Then, based on the retrieved state and details such as the client identifier and the request description, the client generates a unique timestamp that is greater than

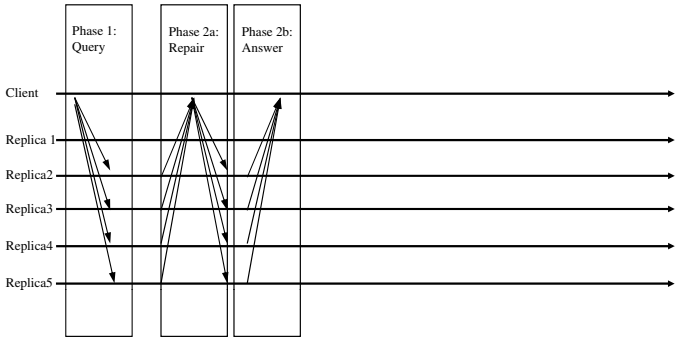

Fig. 10.4 The phases for a read in the PASIS-RW protocol.

those returned by the quorum. In PASIS-RW, timestamps must be strictly increasing so as to identify the most recent overwrite, but need not be consecutive. This fact allows timestamps to be ordered partially by the identifier of the client, and therefore to be generated without involvement of the replicas.

Using this new timestamp, the client submits the overwrite in phase 1b. A non-faulty server accepts the overwrite if it has not yet accepted one that is more recent, storing it in phase 2b. In phase 3, the client considers the operation complete upon learning that it has been accepted by at least a quorum of servers. The protocol includes mechanisms not discussed here for handling the case where the client does not receive such notification in phase 3.

In a read operation, the client submits the read request to at least a quorum of servers. If the most recent overwrite is complete, then, barring a more recent overwrite, the reading client will observe it from more than b servers. Consider the example in the figures, and assume that replica number 4 is faulty. The first client used the quorum containing replicas numbered 1 through 4 for the overwrite. The reading client queries replicas 2 through 5. Replica 5 returns a previous value, and replica 4 might forge a newer value in an attempt to hide the overwrite. However, replicas 2 and 3 each return the correct value. Since the value is returned by two replicas, which is more than b but fewer than a quorum, the client repairs the overwrite using a quorum in phase 2a. Upon learning that the value has been accepted by a quorum, the client uses the value as the result of the read in phase 2b.

Creating Self-Verifying Data

The PASIS-RW protocol just discussed uses a masking quorum system to ensure that any values generated by faulty replicas are not observed in other quorums. If the data were self-verifying, a dissemination quorum system could be used instead because replicas would be unable to generate verifiable values at all. One way to make data self-verifying is to have each client use a digital signature scheme to sign each overwrite. Unfortunately, clients may suffer Byzantine faults in our fault model, and therefore they cannot be trusted.

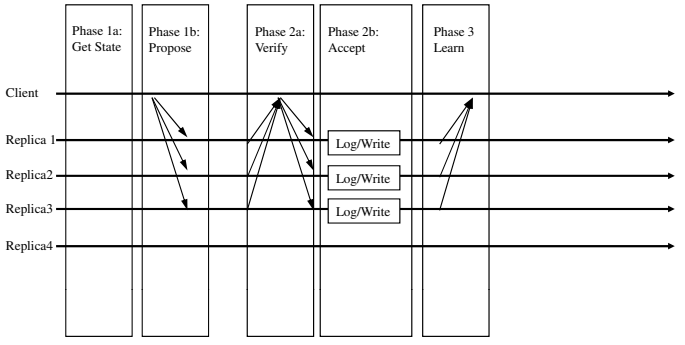

Fig. 10.5 Making data self-verifying.

One way to make an operation self-verifying without trusting clients involves using signatures from a quorum of replicas. Liskov and Rodrigues [17] show how to make operations self-verifying in a read-overwrite protocol with Byzantine clients. There are two steps: an echo protocol like that of Rampart [27] so that non-faulty servers know that other non-faulty servers are not accepting conflicting operations; and a way for clients to verify this as well during reads. For the echo protocol, the first step is to have a quorum of servers provide their signatures stating that they have *tentatively* accepted the operation. If a replica is willing to accept the operation upon the condition that other non-faulty replicas accept no conflicting operation, the replica sends a tentative accept (echo) response. Non-faulty servers only accept operations that have been tentatively accepted by a quorum of servers. Therefore, clients must send a quorum of signed echos in a second phase; in Figure 10.5, this occurs in phase 2a. Servers log the quorum of signatures along with the operation in phase 2b.

The second part of making the operation self-verifying is that the servers return the quorum of signatures to the clients during read operations. A quorum of echo responses proves that no quorum will accept a conflicting operation. This is because every two quorums overlap in some positive number of non-faulty replicas, and no non-faulty replica sends echo messages for conflicting operations. By themselves, faulty servers are too few to generate a quorum of signatures that can be verified. During a read, a client verifies the signatures before using the value, and therefore, each overwrite is self-verifying.

10.4.2 State-Machine-Replication Protocols

State-machine-replication protocols must assign a total order to requests. To minimize the amount of communication between servers, protocols like Q/U [1] and FaB Paxos [21] use opaque quorum systems [18] to order requests *optimistically*. That is, servers independently choose an ordering, without steps that would be required to reach agreement with other servers; the steps are performed only if servers choose different orderings. Under the assumption that servers independently typi-

cally choose the same ordering, the optimistic approach can provide lower overhead in the common case than protocols like BFT [4], which require that servers perform steps to agree upon an ordering *before* choosing it [1]. However, optimistic protocols have the disadvantage of requiring at least $5b+1$ servers to tolerate b server faults, instead of as few as $3b+1$ servers, and so they cannot tolerate as many faults for a given number of servers.

State-machine-replication protocols assign a total order to requests 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.[2] 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.

A quorum system is used for the assignment of permanent sequence numbers. A proposed sequence number for the request is written to the quorum system made up of the replicas. Each non-faulty replica accepts the proposed sequence number only if it has not already assigned the sequence number to a different request. A sequence number is permanent if and only if it has been accepted by a quorum of replicas. This ensures that each permanent sequence number is assigned only to a single request.

The type of quorum system used for accepting proposed sequence numbers to make them permanent implies a lower bound on n in terms of b as discussed earlier. 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 but require more rounds.

A non-faulty replica executes a request only after all lower sequence numbers are assigned permanently and it has executed their corresponding requests. Individual replicas send responses to the client upon executing the request. If a non-faulty replica is waiting to execute a request because it is unaware of the assignment of an earlier sequence number, action is taken so that the replica obtains the missing assignment.

The client determines the correct result from the set of responses it receives 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 because the service is deterministic. However, faulty replicas, as

[2] 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.

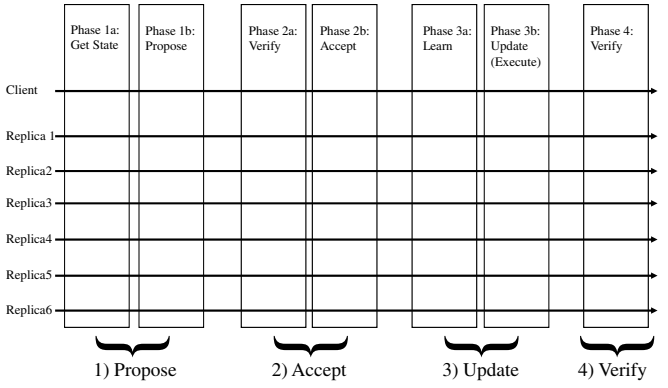

Fig. 10.6 The stages of Byzantine-quorum state-machine-replication protocols.

well as 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 become inconsistent in the face of a Byzantine-faulty proposer. However, the protocols each require a *repair* phase in order to continue to be able to make progress. 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 10.4.2.

The Framework

The framework depicted in Figure 10.6 is an extension of that in Figure 10.1. It 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. If no quorum accepts the proposed sequence number assignment, a new proposal must be tried and the system may require repair (see Section 10.4.2). 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. Depending on the type of quorum system, this may require a round of communication (corresponding to an echo phase as discussed in Section 10.4.1) for the purpose of ensuring that non-faulty 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.

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 $n > 5b$. The Zyzzyva protocol [13] discussed below avoids phase 2a in some executions (e.g., when the servers are all non-faulty and messages are delivered in a timely fashion), a case referred to as *speculative accept*, without requiring an opaque quorum system.

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). Comparing phase 3a to the third phase of an overwrite operation in a read-overwrite protocol described in Section 10.4.1, we see that phase 3a allows the execution replicas to learn that the sequence number assignment has become permanent before performing the update. If phase 3a is omitted, the sequence number assignment may change and the request may need to be executed again with the permanent sequence number.

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.

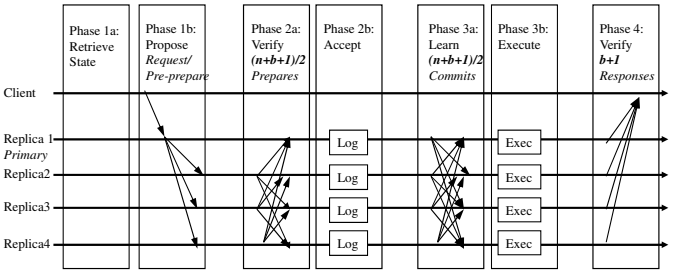

Fig. 10.7 BFT.

Use of Dissemination Quorum Systems

Pessimistic Accept and Update—BFT (without Optimizations). BFT [4] is an 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 10.7 shows the phases of an update request of the BFT protocol in 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., [28, 3, 31, 14].

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 writes (in the sense of a quorum system write, not an overwrite) 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 updates 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 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 logs the quorum of PREPARE and PRE-PREPARE messages as a form of proof. The sequence number assignment is permanent if and only if a quorum of replicas accepts 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 as discussed in Section 10.4.1. 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 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.

Pessimistic Accept, Optimistic Update—BFT w/ Tentative Execution. One way to avoid a round of communication is to employ an optimistic update phase (i.e., to skip phase 3a). 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 used to accept sequence numbers in phase 2 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 order to ensure that the sequence number assignment is indeed permanent. 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 10.8 shows the stages of the BFT protocol with the tentative-execution optimization, compared with the FaB Paxos protocol [21] that is discussed below. Note the smaller quorum size of BFT due to the use of a dissemination quorum system rather than the opaque quorum system required by FaB Paxos.

Speculative Accept, Optimistic Update—Zyzzyva. Zyzzyva [13], shown in Figure 10.9, also uses tentative execution but can save an additional round of communication in "gracious" executions by additionally omitting phase 2a in such executions. Instead of waiting for a quorum of identical responses in phase 4, the client attempts to retrieve identical responses from *all* replicas. If successful, the client knows that, in any quorum, all of the non-faulty replicas in that quorum (i.e., at least $b+1$ non-faulty replicas) will vouch for the sequence number assignment. Since these replicas are guaranteed to outnumber the faulty replicas, the sequence number assignment does not need to be self-verifying.

If the client does not receive identical responses from all servers, the execution is not gracious. In this case, phase 2a is necessary for the reasons described previously. If the client has received identical responses from at least a quorum of servers, the client executes phase 2a by sending the quorum of identical responses to the servers that each log this quorum of responses. If the client receives confirmation that at least a quorum of servers has logged this proof, then it knows that the sequence number assignment is self-verifying. Given this knowledge, the client accepts the result in phase 4.

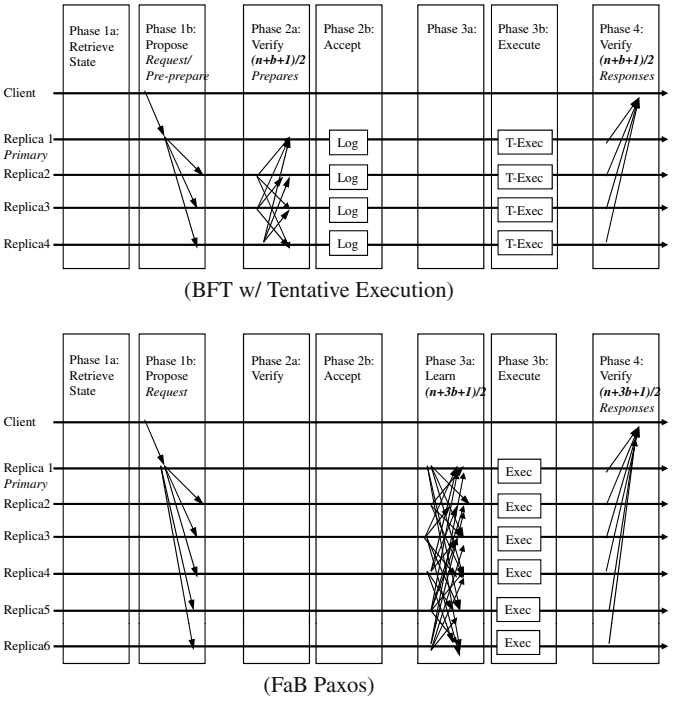

Fig. 10.8 Optimistic update (BFT w/ tentative-execution optimization), compared to optimistic accept (FaB Paxos).

Use of Opaque Quorum Systems

Another way to avoid a round of communication is to employ an optimistic accept phase (i.e., to skip phase 2a). Two of the protocols that we survey 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, can, like the BFT variant described above, also employ an optimistic update optimization known as tentative execution. Since both Q/U and FaB Paxos with tentative execution always skip phase 2a (i.e., regardless of whether the execution is gracious), 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 number assignment; this quorum is larger than quorums in systems that employ dissemination or masking quorum systems.

Optimistic Accept, Pessimistic Update—FaB Paxos. In relation to our framework, FaB Paxos [21], can be viewed as BFT with an optimistic accept phase (pro-

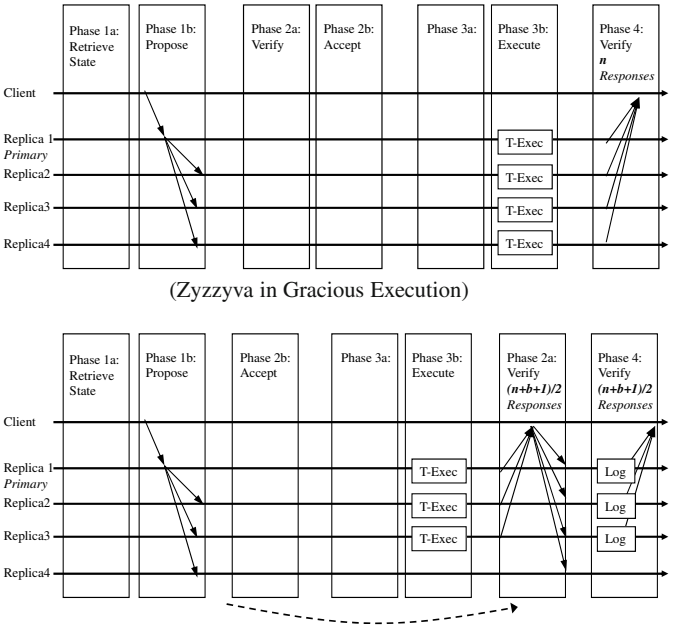

Fig. 10.9 Zyzzyva.

vided by the use of an opaque quorum system). It is seen in the lower half of Figure 10.8. Compared with BFT, FaB Paxos must use larger quorums, and therefore requires more replicas, in order to save this round of communication. While BFT with tentative execution and FaB Paxos (without tentative execution) require the same number of message delays, they have different properties. In BFT, the execution may need to be rolled back and redone if a different sequence number assignment becomes permanent. In FaB Paxos, the execution is not tentative. However, because accept is optimistic, individual servers have no proof that the sequence number assignment is permanent, and so an opaque quorum system is necessary to ensure that permanent sequence numbers are observed later, e.g., in repair.

Optimistic Accept, Optimistic Update—Q/U Protocol. Figures 10.10 and 10.11 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 number at which the next update should be applied. The client sends the object history set along with the request to a quorum of replicas (phase 1b). In phase 2b, each replica

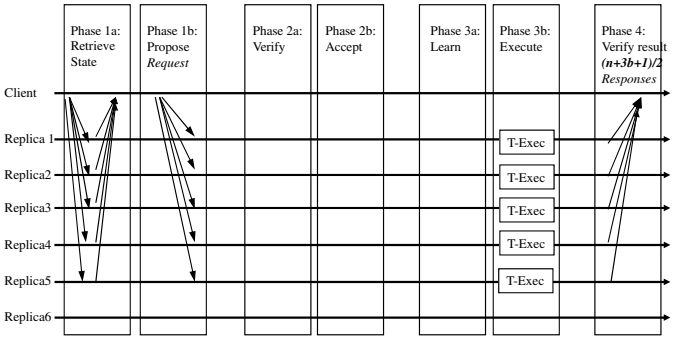

Fig. 10.10 The Q/U protocol (optimistic accept and update).

verifies that it has not executed any operation more recent than that which is reflected in the object history set, and then accepts the update. Having accepted the update, the acceptor executes the request (phase 3b). 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 10.11), clients cache object history sets after each operation. As such, clients can avoid phase 1a if no other clients have since updated the object.

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 10.11 highlights the similarities between Q/U with the pipelined optimization described above and FaB Paxos with the tentative-execution optimization.

Other Trade-offs

BFT, Zyzzyva, 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 (though it can be avoided in some cases with the pipelined optimization). The use of a single proposer has potential 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, if the single proposer is physically closer than the clients to the replicas, then 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. Furthermore, use of a primary can mitigate the impact of client contention. However, the use of a primary: (i) involves an extra

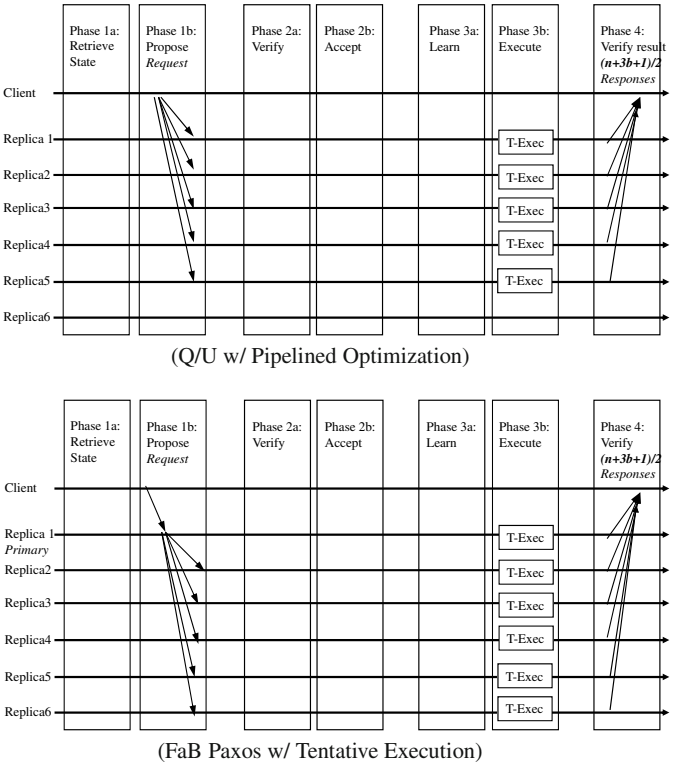

Fig. 10.11 Optimistic accept and update in Q/U and FaB Paxos (both optimized).

message delay (for the request to be forwarded from the client to the primary); and (ii) may allow a faulty primary to slow progress without being detected.

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.[3] In systems such as BFT and FaB Paxos that use a dedicated proposer, the repair phase is used in part 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.

[3] We do not classify protocols based on their repair phases. Therefore, we do not distinguish between BFT and SINTRA [3], for example.

10.5 Conclusion

In this paper we have highlighted a number of advances in both the theory and practice of distributed systems that were facilitated by, or an outgrowth of, work on quorum systems, particularly of a Byzantine-fault-tolerant variety. We have provided a summary of Byzantine quorum systems and several results about the universe sizes they require and their load-dispersing capabilities. We have also discussed probabilistic variations of them. We have presented techniques that have been recently developed to place quorums in networks for efficient access, in some cases while attempting to retain their load-dispersing properties.

We have also presented a framework consisting of logical phases for the comparison of Byzantine-fault-tolerant read-overwrite and state-machine-replication protocols. Our framework centers on the use of Byzantine quorum systems in each protocol, highlighting trade-offs 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.

References

1. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable byzantine fault-tolerant services. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles, pp. 59–74 (2005)
2. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM 32(4), 824–840 (1985)
3. Cachin, C., Poritz, J.A.: Secure intrusion-tolerant replication on the Internet. In: International Conference on Dependable Systems and Networks, pp. 167–176 (2002)
4. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems 20(4), 398–461 (2002)
5. Fu, A.W.: Delay-optimal quorum consensus for distributed systems. IEEE Transactions on Parallel and Distributed Systems 8(1), 59–69 (1997)
6. Gilbert, S., Malewicz, G.: The quorum deployment problem. In: Proceedings of the 8th International Conference on Principles of Distributed Systems (2004)
7. Golovin, D., Gupta, A., Maggs, B.M., Oprea, F., Reiter, M.K.: Quorum placement in networks: Minimizing network congestion. In: Proceedings of the 25th ACM Symposium on Principles of Distributed Computing, pp. 16–25 (2006)
8. Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient Byzantine-tolerant erasure-coded storage. In: International Conference on Dependable Systems and Networks (2004)
9. Guerraoui, R., Vukolic, M.: Refined quorum systems. In: Symposium on Principles of Distributed Computing, pp. 119–128 (2007)
10. Gupta, A., Maggs, B.M., Oprea, F., Reiter, M.K.: Quorum placement in networks to minimize access delays. In: Proceedings of the 24th ACM Symposium on Principles of Distributed Computing, pp. 87–96 (2005)
11. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
12. Kobayashi, N., Tsuchiya, T., Kikuno, T.: Minimizing the mean delay of quorum-based mutual exclusion schemes. Journal of Systems and Software 58(1), 1–9 (2001)
13. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative Byzantine fault tolerance. In: Symposium on Operating Systems Principles, pp. 45–58. ACM Press, New York (2007), http://doi.acm.org/10.1145/1294261.1294267
14. Kotla, R., Dahlin, M.: High throughput Byzantine fault tolerance. In: International Conference on Dependable Systems and Networks, p. 575 (2004)

15. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (1982)
16. Lin, X.: Delay optimizations in quorum consensus. In: Eades, P., Takaoka, T. (eds.) ISAAC 2001. LNCS, vol. 2223, pp. 575–586. Springer, Heidelberg (2001)
17. Liskov, B., Rodrigues, R.: Tolerating Byzantine faulty clients in a quorum system. In: International Conference on Distributed Computing Systems (2006)
18. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing 11(4), 203–213 (1998)
19. Malkhi, D., Reiter, M.K., Wool, A.: The load and availability of Byzantine quorum systems. SIAM Journal of Computing 29(6), 1889–1906 (2000)
20. Malkhi, D., Reiter, M.K., Wool, A., Wright, R.N.: Probabilistic quorum systems. Information and Computation 170(2), 184–206 (2001)
21. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Transactions on Dependable and Secure Computing 3(3), 202–215 (2006)
22. Merideth, M.G., Reiter, M.K.: Probabilistic opaque quorum systems. In: Pelc, A. (ed.) DISC 2007. LNCS, vol. 4731, pp. 403–419. Springer, Heidelberg (2007)
23. Merideth, M.G., Reiter, M.K.: Write markers for probabilistic quorum systems. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) OPODIS 2008. LNCS, vol. 5401, pp. 5–21. Springer, Heidelberg (2008)
24. Naor, M., Wool, A.: The load, capacity and availability of quorum systems. SIAM Journal of Computing 27(2), 423–447 (1998)
25. Oprea, F.: Quorum placement on wide-area networks. Ph.D. thesis, Electrical & Computer Engineering Department, Carnegie Mellon University (2008)
26. Oprea, F., Reiter, M.K.: Minimizing response time for quorum-system protocols over wide-area networks. In: Proceedings of the 37th International Conference on Dependable Systems and Networks, pp. 409–418 (2007)
27. Reiter, M.K.: Secure agreement protocols: Reliable and atomic group multicast in Rampart. In: Conference on Computer and Communication Security, pp. 68–80 (1994)
28. Rodrigues, R., Castro, M., Liskov, B.: BASE: Using abstraction to improve fault tolerance. In: Symposium on Operating Systems Principles (2001)
29. Song, Y.J., van Renesse, R., Schneider, F.B., Dolev, D.: The building blocks of consensus. In: Proceedings of the 9th International Conference on Distributed Computing and Networking (2008)
30. Tsuchiya, M., Yamaguchi, M., Kikuno, T.: Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes. IEEE Transactions on Parallel and Distributed Systems 10(4), 337–345 (1999)
31. Yin, J., Martin, J.P., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating agreement from execution for Byzantine fault tolerant services. In: Symposium on Operating Systems Principles, pp. 253–267 (2003)