State Machine Replication with Byzantine Faults

Christian Cachin

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
cachin@acm.org

26th May, 2008

Abstract

This paper gives an introduction to state machine replication in groups subject to arbitrary or "Byzantine" faults. It explains the principles of such protocols and covers the following topics: broadcast primitives, distributed cryptosystems, randomized Byzantine agreement protocols, and atomic broadcast protocols.

1 Introduction

Coordinating a group of replicas to deliver a service, while some of them are actively trying to prevent the coordination effort, is a fascinating topic. It stands at the heart of Pease et al.'s classic paper [PSL80] on reaching agreement in the presence of faults, which ignited an impressive flow of papers elaborating on this problem over the last 30 years.

In this paper, we survey the principles of atomic broadcast protocols in asynchronous networks for a group of n parties, of which up to t are subject to so-called "Byzantine" faults. No assumptions about the behavior of the faulty parties are made; they may deviate arbitrarily form the protocol, as if corrupted by a malicious adversary. Atomic broadcast guarantees that every correct party in the group receives the same sequence of broadcast requests. It is the key step for replicating a deterministic service among the group [Lam78, Sch90]. This approach allows to build highly resilient and intrusion-tolerant services on the Internet.

The model considered here is motivated by practice. The parties are connected pairwise by reliable authenticated channels. Protocols may use cryptographic methods, such as public-key encryption and digital signatures. A trusted entity takes care of initially generating and distributing private keys, public keys, and certificates, such that every party can verify signatures by all other parties, for example. The system is asynchronous: there are no bounds on the delivery time of messages and no synchronized clocks. This is an important aspect because systems whose correctness relies on timing assumptions are vulnerable to attackers that simply slow down the correct parties or delay the messages sent between them.

The paper is organized as follows. We first introduce some building blocks for atomic broadcast; they consist of two broadcast primitives, distributed cryptosystems, and randomized Byzantine agreement protocols. Then we present the structure of some recent atomic broadcast protocols. Finally, we illustrate some issues with service replication that arise specifically in the presence of Byzantine faults.

2 Building Blocks

2.1 Broadcast Primitives

We present two broadcast primitives, which are found in one way or other in all agreement and atomic broadcast protocols tolerating Byzantine faults. Such protocols invoke multiple instances of a broadcast primitive. All instances are thus parameterized by a tag ID, which is contained implicitly in every message and must be included in all signatures.

These broadcast instances have a distinguished sender, which *broadcasts* a *request* m to the group at the start of the protocol. All parties should later *deliver* m, though termination is not guaranteed with a faulty sender. To simplify the notation, we assume that all requests are unique.

We first present *consistent broadcast* for a group of n parties, called $P_1, \ldots, P_n$. A designated sender P_s starts the protocol by executing *c-broadcast* with a request m. All parties terminate the protocol by executing *c-deliver* and output a request m. Consistent broadcast ensures only that the delivered request is the same for all receivers. In particular, termination is not guaranteed with a faulty sender.

The following definition is formulated using conditions that generalize the corresponding notions for systems with crash failures [HT93].

Definition 1 (Consistent broadcast). *A protocol for consistent broadcast satisfies:*

Validity: *If a correct sender P_s* c-broadcasts *m, then all correct parties eventually* c-deliver *m.*

Consistency: *If a correct party* c-delivers *m and another correct party* c-delivers *m', then $m = m'$.*

Integrity: *For any request m, every correct party* c-delivers *m at most once. Moreover, if the sender P_s is correct, then m was previously* c-broadcast *by P_s.*

Algorithm 1 (Echo broadcast [Tou84]). All parties use secure digital signatures.

upon *c-broadcast*(m): // only P_s
  send message $(\texttt{send}, m)$ to all

upon *receiving a message* $(\texttt{send}, m)$ *from* P_s:
  compute signature σ on $(\texttt{echo}, s, m)$
  send message $(\texttt{echo}, m, \sigma)$ to P_s

upon *receiving* $\lceil \frac{n+t+1}{2} \rceil$ *messages* $(\texttt{echo}, m, \sigma_i)$ *with valid* σ_i: // only P_s
  let Σ be the list of all received signatures σ_i
  send message $(\texttt{final}, m, \Sigma)$ to all

upon *receiving a message* $(\texttt{final}, m, \Sigma)$ *from* P_s *with* $\lceil \frac{n+t+1}{2} \rceil$ *valid signatures in* Σ:
  c-deliver(m)

Theorem 2. *Algorithm 1 implements consistent broadcast for $n > 3t$.*

Proof. Validity and integrity are straightforward to verify. Consistency follows from the observation that the request m in any $\texttt{final}$ message with $\lceil \frac{n+t+1}{2} \rceil$ valid signatures in Σ is unique. To see this, consider the set of parties that issued the $\lceil \frac{n+t+1}{2} \rceil$ signatures: because there are only n distinct parties, every two sets of signers overlap in at least one correct party. Such sets are also called *Byzantine quorums* [MR98b]. □

The message complexity of echo broadcast is $O(n)$ and its communication complexity is $O(n^2(k + |m|))$, where k denotes the length of a digital signature. Using a non-interactive threshold signature scheme, the communication complexity can be reduced to $O(n(k + |m|))$ [CKPS01].

The second broadcast primitive is *reliable broadcast*. It is characterized by a *r-broadcast* event and a *r-deliver* event in the same way as consistent broadcast. Reliable broadcast additionally ensures agreement on the delivery of the request in the sense that either all correct parties deliver it or none delivers it. In the literature *consistency* and *totality* are often combined into a single condition called *agreement*. This primitive is also known as the "Byzantine generals problem."

Definition 2 (Reliable broadcast). *A protocol for reliable broadcast is a consistent broadcast protocol that satisfies also:*

Totality: *If some correct party* r-delivers *a request, then all correct parties eventually* r-deliver *a request.*

Algorithm 3 (Bracha broadcast [Bra87]).

upon *r-broadcast*(m): // only P_s

send message $(\texttt{send}, m)$ to all

upon *receiving a message* $(\texttt{send}, m)$ *from* P_s:

send message $(\texttt{echo}, m)$ to all

upon *receiving* $\lceil \frac{n+t+1}{2} \rceil$ *msgs.* $(\texttt{echo}, m)$ *and not having sent a* `ready`*-msg.:*

send message $(\texttt{ready}, m)$ to all

upon *receiving* $t + 1$ *msgs.* $(\texttt{ready}, m)$ *and not having sent a* `ready`*-msg.:*

send message $(\texttt{ready}, m)$ to all

upon *receiving* $2t + 1$ *messages* $(\texttt{ready}, m)$:

r-deliver(m)

Theorem 4. *Algorithm 3 implements reliable broadcast for* $n > 3t$.

Proof. Consistency follows from the same argument as in Theorem 2, since the request m in any `ready` message of a correct party is unique. Totality is implied by the "amplification" of `ready` messages from $t+1$ to $2t+1$ with the fourth **upon** clause of the algorithm. Specifically, if a correct party has *r-delivered* m, it has received a `ready` message with m from $2t+1$ distinct parties. Therefore, at least $t+1$ correct parties have sent a `ready` message with m, which will be received by all correct parties and cause them to send a `ready` message as well. Because $n-t \geq 2t+1$, all correct parties eventually receive enough `ready` messages to terminate. □

The message complexity of Bracha broadcast is $O(n^2)$ and its communication complexity is $O(n^2|m|)$. Unlike consistent broadcast, it does not need digital signatures, which are usually computationally expensive operations.

Several complex agreement and atomic broadcast protocols use either the consistent or the reliable broadcast primitive, and one can often substitute either primitive for the other one in these protocols, with appropriate modifications. Selecting one of these primitives involves a trade-off between computation time and message complexity. It is an interesting question to determine the experimental conditions under which either primitive is more suitable; Moniz et al. [MNCV06] present some initial answers.

2.2 Distributed Cryptography

Distributed cryptography spreads the operation of a cryptosystem among a group of parties in a fault-tolerant way [Des94]; such schemes are also called *threshold cryptosystems*. They are based on *secret sharing* methods and distributed implementations are typically known only for public-key cryptosystems because of their algebraic properties.

2.2.1 Secret sharing.

In a $(t+1)$-out-of-n *secret sharing scheme*, a secret s, element of a finite field $\mathbb{F}$ with q elements, is shared among n parties such that the cooperation of at least $t+1$ parties is needed to recover s. Any group of t or fewer parties should not get any information about s.

Algorithm 5 (Polynomial secret sharing [Sha79]). To *share* $s \in \mathbb{F}_q$, a *dealer* $P_d \notin \{P_1, \ldots, P_n\}$ chooses uniformly at random a polynomial $f(X) \in \mathbb{F}_q[X]$ of degree t subject to $f(0) = s$, generates *shares* $s_i = f(i)$, and sends s_i to P_i for $i = 1, \ldots, n$. To recover s among a group of $t+1$ parties with indices $\mathcal{S}$, every party reveals its share and all parties together recover the secret by computing

$$s = f(0) = \sum_{i \in \mathcal{S}} \lambda_{0,i}^{\mathcal{S}} s_i,$$

where

$$\lambda_{0,i}^{\mathcal{S}} = \prod_{j \in \mathcal{S}, j \neq i} \frac{j}{j-i}$$

are the (easy-to-compute) Lagrange coefficients.

Theorem 6. *The secret sharing scheme has perfect security, i.e., the shares held by every group of t or fewer parties are statistically independent of s.*

We refer to the literature for definitions and a proof of the theorem. Secret sharing schemes do not directly give fault-tolerant replicated implementations of cryptosystems; if the secret key were reconstructed for performing a cryptographic operation, all security would be lost because the key is exposed to the faulty parties. So-called *threshold cryptosystems* perform these operations securely; as an example, threshold public-key encryption based on the ElGamal system is presented next (details can be found in textbooks on modern cryptography [Sma03]).

2.2.2 Discrete logarithm-based cryptosystems.

Let G be a group of prime order q such that g is a generator of G. The *discrete logarithm problem (DLP)* means, for a random $y \in G$, to compute $x \in \mathbb{Z}_q$ such that $y = g^x$. The *Diffie-Hellman problem (DHP)* is to compute $g^{x_1 x_2}$ from random $y_1 = g^{x_1}$ and $y_2 = g^{x_2}$.

It is conjectured that there exist groups in which solving the DLP and the DHP is *hard*, for example, the multiplicative subgroup $G \subset \mathbb{Z}_p^*$ of order q, for some prime $p = mq + 1$ (recall that q is prime). For example, this choice with $|p| = 2048$ and $|q| = 256$ is considered secure today and used widely on the Internet.

A *public-key cryptosystem* consists of three algorithms, K, E, and D. The key generation algorithm K outputs a pair of keys (pk, sk). The encryption and decryption algorithms, E and D, have the property that for all (pk, sk) generated by K and for any plaintext message m, it holds $\mathsf{D}(sk, \mathsf{E}(pk, m)) = m$.

A public-key cryptosystem is *semantically secure* if no efficient adversary A can find two messages such that A can distinguish their encryptions. Semantic security provides security against *passive* attacks, but not against *active* attacks.

2.2.3 Threshold public-key encryption.

The *ElGamal cryptosystem* is based on the DHP: K selects a random secret key $x \in \mathbb{Z}_q$ and computes the public key as $y = g^x$. The encryption of $m \in \{0,1\}^k$ under public-key y is the tuple $(A, B) = (g^r, m \oplus H(y^r))$, computed using a randomly chosen $r \in \mathbb{Z}_q$ and a hash function $H : G \to \{0,1\}^k$. The decryption of a ciphertext (A, B) is $\hat{m} = H(A^x) \oplus B$. One can easily verify that $\hat{m} = m$ because $A^x = g^{rx} = g^{xr} = y^r$, and therefore, the argument to H is the same in encryption and decryption. The cryptosystem is semantically secure under the assumption that the DHP is hard.

Algorithm 7 (Threshold ElGamal encryption). Let the secret key x be *shared* among $P_1, \ldots, P_n$ using a polynomial f of degree t over $\mathbb{Z}_q$ such that P_i holds a share $x_i = f(i)$. The public key $y = g^x$ is known to all parties. Encryption is the same as in standard ElGamal above. For decryption, a client sends a decryption request containing a ciphertext (A, B) to all parties. Upon receiving a decryption request, party P_i computes a *decryption share* $d_i = A^{x_i}$ and sends it to the client. Upon receiving decryption shares from a set of $t+1$ parties with indices $\mathcal{S}$, the client recovers the plaintext as

$$\hat{m} = H\Big(\prod_{i \in \mathcal{S}} {d_i}^{\lambda^{\mathcal{S}}_{0,i}}\Big) \oplus B.$$

Theorem 8. *Algorithm 7 implements a $(t+1)$-out-of-n threshold ElGamal cryptosystem that tolerates the passive corruption of $t < n/2$ parties.*

Proof. The decryption is correct because

$$\prod_{i \in \mathcal{S}} {d_i}^{\lambda^{\mathcal{S}}_{0,i}} = \prod_{i \in \mathcal{S}} A^{x_i \lambda^{\mathcal{S}}_{0,i}} = A^{\sum_{i \in \mathcal{S}} x_i \lambda^{\mathcal{S}}_{0,i}} = A^x$$

from the properties of secret sharing. The system is secure because the decryption shares ($d_i = A^{x_i}$) do not reveal anything useful about the shares of the secret key (x_i). □

This is a *non-interactive* threshold cryptosystem, as no interaction among the parties is needed. It can also be made robust such that it is secure against active attacks [SG02]. Non-interactive threshold cryptosystems can easily be integrated in asynchronous protocols, but many threshold cryptosystems require *synchronous* networks with broadcast.

2.3 Byzantine Agreement

One step up from the broadcast primitives is a protocol to reach agreement despite Byzantine faults. It is a prerequisite for implementing atomic broadcast. All atomic broadcast protocols, at least in the model with static groups considered here, either explicitly invoke an agreement primitive or implicitly contain one.

The Byzantine agreement problem is characterized by two events *propose* and *decide*; every party executes *propose*(v) to start the protocol and *decide*(v) to terminate it for a value v. In *binary* agreement, the values are bits.

Definition 3 (Byzantine agreement). *A protocol for binary Byzantine Agreement satisfies:*

Validity: *If all correct parties* propose v*, then some correct party eventually* decides v.

Agreement: *If some correct party* decides v *and another correct party* decides v'*, then* $v = v'$.

Termination: *Every correct party eventually* decides.

The result of Fischer, Lynch, and Paterson [FLP85] implies that every protocol solving Byzantine agreement has executions that do not terminate. State machine replication is also subject to this limitation. Roughly at the same time, however, randomized protocols to circumvent this impossibility have been developed. They make the probability of non-terminating executions arbitrarily small [Rab83, Ben83, Tou84]. More precisely, given a logical time measure T, such as the number of steps performed by all correct parties, define *termination with probability 1* as

$$\lim_{T \to \infty} \Pr[\text{some correct party has not } \textit{decided} \text{ after time } T] = 0.$$

Algorithm 9 (Binray Randomized Byzantine Agreement [Tou84]). Suppose a trusted dealer has *shared* a sequence $s_0, s_1, \ldots$ of random bits, called *coins*, among the parties, using $(n-t)$-out-of-n secret sharing. A party can access the coin s_r using a *recover*(r) operation, which involves a protocol that exchanges some messages, and gives the same coin value to every party. The two *upon* clauses of the algorithm below are executed concurrently.

Every party maintains a value v, called its *vote*, and the protocol proceeds in global asynchronous rounds. Every global round consists of two rounds of message exchanges. In the first round, the parties exchange their votes, and every party determines the majority vote. In the second round, every party relays the majority to all others, this time using reliable broadcast and accompanied by a set Π that serves as a *proof* for justifying the choice of the majority. After receiving reliable broadcasts from $n-t$ parties, every party determines the majority of this second vote and adopts its outcome as its vote v if the vote is unanimous; otherwise, a party sets v to the shared coin for the round. If the coin equals the outcome of the second vote, then the party decides.

upon *propose*(v):
  $r \leftarrow 0$
  loop
    send the signed message $(\texttt{1-vote}, r, v)$ to all
    receive properly signed $(\texttt{1-vote}, r, v')$ messages from $n-t$ distinct parties
    $\Pi \leftarrow$ set of received `1-vote` messages including the signatures
    $v \leftarrow$ value v' that is contained most often in Π
    r-broadcast the message $(\texttt{2-vote}, r, v, \Pi)$
    wait for *r-delivery* of $(\texttt{2-vote}, r, v', \Pi)$ messages from $n-t$ distinct senders
        with valid signatures in Π and correctly computed v'
    $b \leftarrow$ value v' that is contained most often among the r-delivered `2-vote` msgs.
    $c \leftarrow$ number of r-delivered `2-vote` messages with $v' = b$
    $s_r \leftarrow$ *recover*(r)
    if $c = n - t$ **then**
      $v \leftarrow b$
    else
      $v \leftarrow s_r$
    if $b = s_r$ **then**
      send the message $(\texttt{decide}, v)$ to all   // note that $v = s_r = b$
    $r \leftarrow r + 1$

upon *receiving* $t+1$ *messages* $(\texttt{decide}, b)$:
  if not *decided* **then**
    send the message $(\texttt{decide}, b)$ to all
    decide(b)

Lemma 10. *If all correct parties start some round* r *with vote* v_0*, then all correct parties terminate round* r *with vote* v_0*.*

Proof. It is impossible to create a valid Π for a `2-vote` message with a vote $v \neq v_0$ because v must be set to the majority value in $n-t$ received `1-vote` messages and $n - t > 2t$. □

Lemma 11. *In round* $r \geq 0$*, the following holds:*

a) If a correct party sends a `decide` *message for* v_0 *at the end of round* r*, then all correct parties terminate round* r *with vote* v_0*.*

b) With probability at least $\frac{1}{2}$*, all correct parties terminate round* r *with the same vote.*

Proof. Consider the assignment of b and c in round r. If some correct party obtains $c = n - t$ and $b = v_0$, then no correct party can obtain a majority of `2-vote` messages for a value different from v_0 (there are only n `2-vote` messages and they satisfy the consistency of reliable broadcast). Those correct parties with $c = n - t$ set vote v to v_0; every other correct party sets v to s_r. Hence, if $s_r = v_0$, all correct parties terminate round r with vote v_0.

Claim *a)* now follows upon noticing that a correct party only sends a `decide` message for v_0 when $v_0 = b = s_r$.

Claim *b)* follows because the first correct party to assign b and c does so *before* anything about s_r is known (to the adversary). Thus, s_r and v_0 are independent and $s_r = v_0$ with probability $\frac{1}{2}$. □

Theorem 12. *Algorithm 9 implements binary Byzantine agreement for $n > 3t$, where termination holds with probability 1.*

The theorem follows easily from the preceding lemmas. Since the protocol reaches agreement with probability at least $\frac{1}{2}$ in every round, the expected number of rounds is 2, and the expected number of messages is $O(n^3)$.

2.3.1 Using cryptographic randomness.

The problem with Algorithm 9 is that every round in the execution uses up one shared coin in the sequence $s_0, s_1, \ldots$. As coins cannot be reused, this is a problem in practice. A solution for this is to obtain the shared coins from a threshold cryptographic function. More precisely, let the coin value s_r be the output of a distributed *pseudorandom function (PRF)*, evaluated on the round number r and the protocol instance identifier ID.

A PRF is parameterized by a secret key and maps every input-string to an output-string that looks random to anyone who does not have the secret key. A practical PRF construction is a block cipher with a secret key; distributed implementations, however, are only known for functions based on public-key cryptosystems. Cachin et al. [CKS05] describe a suitable distributed PRF based on the Diffie-Hellman problem. With their implementation of the shared coin, Algorithm 9 is practical and has expected message complexity $O(n^3)$. This can further be reduced to $O(n^2)$ expected messages [CKS05].

3 Atomic Broadcast Protocols

Atomic broadcast delivers multiple requests in the same order to all parties. Whereas instances of reliable broadcast may be independent of each other, the total order of atomic broadcast links these together and requires more complex implementations. The details of the protocols in this section are therefore omitted.

Analogously to reliable broadcast, atomic broadcast is characterized by an *a-broadcast* event, executed by the sender of a request, and an *a-deliver* event. Every party may a-broadcast multiple requests; also a-deliver generally occurs multiple times.

Definition 4 (Atomic broadcast). *A protocol for atomic broadcast is a reliable broadcast protocol that satisfies also:*

Total order: *If two correct parties P_i and P_j both* a-deliver *two requests m and m', then P_i* a-delivers *m before m' if and only if P_j* a-delivers *m before m'.*

Early atomic broadcast protocols [RB94, MR98a, PMMS01] used dynamic groups with a membership service that might evict faulty parties from the group, even if they only appear to be slow. When an attacker manages to exploit network delays accordingly, this may lead to the problematic situation where the correct parties are in a minority, and the protocol violates safety.

The more recent protocols, on which we focus here, never violate safety because of network instability. We distinguish between two kinds of atomic broadcast protocols: agreement-based and leader-based ones. We next review the principles of these protocols, starting with the historically older protocols based on agreement. A third option is to combine leader- and agreement-based protocols into hybrid atomic broadcast protocols.

3.1 Agreement-based Atomic Broadcast

The canonical implemention of atomic broadcast uses an agreement primitive to determine the next request that should be a-delivered. Such a protocol proceeds in asynchronous rounds and uses one instance of (multi-valued) Byzantine agreement in every round to agree on a set of requests, which are then delivered in a fixed order at the end of the round. The same approach has been used in the crash-failure model [HT93].

Incoming requests are buffered and proposed for delivery in the next available round. The validity notion of Byzantine agreement, however, must be amended for this to work: the standard validity condition only guarantees that a particular decision is reached when all parties make the same proposal. This will rarely be the case in practice, where every party receives different requests to a-broadcast.

A suitable notion of validity for *multi-valued Byzantine agreement* has been proposed by Cachin et al. [CKPS01]; it defines a test for determining if a proposed value is acceptable and externalizes it. A corresponding protocol implements multi-valued agreement and uses a binary agreement primitive [CKPS01]; this protocol incurs a communication overhead of $O(n^2)$ messages over the primitive for binary agreement.

With multi-valued randomized Byzantine agreement, asynchronous atomic broadcast protocols can easily be implemented; they satisfy the relaxation of Definition 4 to termination with probability 1 in the validity condition. These protocols have been prototyped in several practical systems [CP02, MNCV06, RSS07].

3.2 Leader-based Atomic Broadcast

Agreement-based protocols send all requests through Byzantine agreement to determine their order; but agreement is a rather expensive protocol. A more efficient approach is taken by the BFT protocol [CL02], which relies on a single party, called the *leader*, to determine the request order. Because the leader may be faulty, its actions must be checked by the other parties in a distributed protocol. BFT is actually a Byzantine-fault-tolerant version of the Paxos protocol [Lam98, Lam01a, Lam01b].

The BFT protocol proceeds in *epochs*, where an epoch consists of a *normal-operation phase* and *recovery phase* (also called *view change*). During every epoch, a designated party acts as the leader for the normal-operation phase, determines the delivery order of requests, and commits every request through reliable broadcast with Bracha's protocol (Algorithm 3). Because the leader runs the reliable broadcasts in a sequence, this guarantees that all correct parties receive and a-deliver the requests in the same order. This approach ensures safety even when the leader is faulty, but may violate liveness when the leader stops r-broadcasting requests.

When the leader appears faulty in the eyes of enough other parties, the protocol switches to the recovery phase. This step is based on timeouts that must occur on at least $t + 1$ parties. Once enough parties have switched to the recovery phase, the protocol aborts the still ongoing reliable broadcasts, and the recovery phase eventually starts at all correct parties. The goal of the recovery phase is to agree on a new leader for the next epoch and on the a-delivery of the requests that the previous leader may have left in an inconclusive state.

Progress during the recovery phase and in the subsequent epoch requires the timely cooperation of the new leader, since the protocol is deterministic and therefore subject to the FLP impossibility result [FLP85]. It is possible that no requests are delivered before the epoch ends again, and the protocol loses liveness. However, it is assumed that this occurs extremely rarely in practice.

Despite its inherent complexity, the recovery phase is still more efficient than one round in the agreement-based atomic broadcast protocols. The BFT protocol has message complexity $O(n^2)$, always ensures safety, and liveness during periods where the network is stable enough; it is considered practical by many system implementors.

3.3 Hybrid Atomic Broadcast

Combining the efficiency of the leader-based approach during normal operation with the strong guarantees of the agreement-based approach for recovery, protocols have been proposed that take the best features from both approaches.

The protocol of Kursawe and Shoup [KS05] is divided into epochs and uses reliable broadcast during the normal-operation phase, like the BFT protocol. For recovery, however, it employs randomized Byzantine agreement and ensures that some requests are a-delivered in any case. It therefore guarantees safety *and* liveness and has the same efficiency as the BFT protocol during stable periods.

Ramasamy and Cachin [RC06] replace the reliable broadcast primitive in the Kursawe-Shoup protocol by consistent broadcast. The resulting protocol is attractive for its low message complexity, only $O(n)$ expected messages per request, amortized over protocol executions with long periods of stability, compared to $O(n^2)$ for all other atomic broadcast protocols in the Byzantine fault model. The improvement comes at the cost of adding complexity to the recovery phase and, more importantly, by using expensive public-key operations during the normal-operation phase.

4 Service Replication

A fault-tolerant service implemented using replication should present the same interface to its clients as when implemented using a single server. Sending requests to the replicated (deterministic) service via atomic broadcast enables the replicas to process the same sequence of requests and to maintain the same state [Sch90]. If failures are limited to benign crashes, the client may obtain the correct service response from any replica.

When the replicas are subject to Byzantine faults, additional concerns arise: First, services involving cryptographic operations and secret keys must remain secure despite the leakage of keys from corrupted replicas; second, clients must not rely on the response message from any single replica because the replica may be faulty and give a wrong answer; and third, faulty replicas may violate the causality between requests sent to the replicated service. We review methods to address each of these concerns next.

4.1 Replicating Cryptographic Services

The service may involve cryptographic operations with keys that should be protected, for example, when the service receives requests that are encrypted with a service-specific key, or when it signs responses using digital signatures. In this case, a break-in to single replica will leak all secrets to the adversary. To protect against this attack, the cryptographic operations of the service should be implemented using threshold cryptography. This leaves the service interface for clients unchanged and hides the distributed implementation of the service. An important example of such a service is a certification authority (CA), which binds public keys to names and asserts this with its digital signature. Since CAs often serve as the root of trust for large systems, implementing them in an intrusion-tolerant way is a good method to protect them. This principle has been demonstrated in prototype systems [ZSvR02, CS04].

4.2 Handling Responses Securely

As the response from any single replica may be forged, clients must generally receive at least $t + 1$ responses and infer the service response from them. If all $t + 1$ responses are equal, then at least one of them was sent by a correct party, which ensures that the response is correct. Collecting responses and deciding for a correct one involves a modification of the client-side service interface. Usually this modification is simple and can be hidden in a library. But if no such modification is possible, there is an alternative for services that rely on cryptographically protected responses: use threshold cryptography to authenticate the response, for example, with a digital signature. Then it is sufficient that the client verifies the authenticity of the response once because it carries the approval of at least $t + 1$ parties that executed the request [RB94]. In this context, it is interesting to mention the result of Yin et al. [YMAD03] that only $2t + 1$ parties need to maintain the state of the service instead of all n parties.

4.3 Preserving Causality of Requests

When a client atomically broadcasts a request to the replicated service, the faulty replicas may be able to create a derived request that is a-delivered and executed *before* the client's request. This violates the safety of the service, more precisely, the causal order among requests. For example, consider a service that registers names in a directory on a first-come, first-served basis. When a faulty party peeks inside the the atomic broadcast protocol and observes that an interesting name is being registered, it may try to quickly register the name for one of its friends.

One can ensure a causal order among the requests to the service with the following protocol [RB94]. To a-broadcast a request, the client first encrypts it with a $(t + 1)$-out-of-n threshold public-key cryptosystem under the public key of the service. Then, it a-broadcasts the resulting ciphertext. Upon a-delivery of a ciphertext, a replica first computes a decryption share for the ciphertext, using its share of the corresponding decryption key, and sends the decryption share to all replicas. Then it waits for $t + 1$ decryption shares to arrive, recovers the original request, and a-delivers it.

This protocol can be seen as an atomic broadcast protocol that respects causal order in the Byzantine-fault model [CKPS01].

5 Conclusion

In the recent years, we have seen a revival of the research on protocols for Byzantine agreement and atomic broadcast subject to Byzantine faults. This is because such protocols appear to be much more practical nowadays and because there is demand for realizing intrusion-tolerant services on the Internet. This paper has presented the building blocks for such protocols, some 25 years old, and some very recent, and shown how they fit together for securing distributed on-line services.

References

[Ben83] M. Ben-Or, *Another advantage of free choice: Completely asynchronous agreement protocols*, Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC), 1983, pp. 27–30.

[Bra87] G. Bracha, *Asynchronous Byzantine agreement protocols*, Information and Computation **75** (1987), 130–143.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, *Secure and efficient asynchronous broadcast protocols (extended abstract)*, Proc. CRYPTO 2001 (J. Kilian, ed.), Lecture Notes in Computer Science, vol. 2139, Springer, 2001, pp. 524–541.

[CKS05] C. Cachin, K. Kursawe, and V. Shoup, *Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography*, Journal of Cryptology **18** (2005), no. 3, 219–246.

[CL02] M. Castro and B. Liskov, *Practical Byzantine fault tolerance and proactive recovery*, ACM Transactions on Computer Systems **20** (2002), no. 4, 398–461.

[CP02] C. Cachin and J. A. Poritz, *Secure intrusion-tolerant replication on the Internet*, Proc. International Conference on Dependable Systems and Networks (DSN-2002), June 2002, pp. 167–176.

[CS04] C. Cachin and A. Samar, *Secure distributed DNS*, Proc. International Conference on Dependable Systems and Networks (DSN-2004), June 2004, pp. 423–432.

[Des94] Y. Desmedt, *Threshold cryptography*, European Transactions on Telecommunications **5** (1994), no. 4, 449–457.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM **32** (1985), no. 2, 374–382.

[HT93] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*, Distributed Systems (S. J. Mullender, ed.), ACM Press & Addison-Wesley, 1993.

[KS05] K. Kursawe and V. Shoup, *Optimistic asynchronous atomic broadcast*, Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science, vol. 3580, 2005, pp. 204–215.

[Lam78] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Communications of the ACM **21** (1978), no. 7, 558–565.

[Lam98] ———, *The part-time parliament*, ACM Transactions on Computer Systems **16** (1998), no. 2, 133–169.

[Lam01a] ———, *Paxos made simple (Distributed Computing Column)*, SIGACT News **32** (2001), no. 4, 51–58.

[Lam01b] B. Lampson, *The ABCD's of Paxos*, Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC), 2001.

[MNCV06] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo, *Randomized intrusion-tolerant asynchronous services*, Proc. International Conference on Dependable Systems and Networks (DSN-2006), 2006, pp. 568–577.

[MR98a] D. Malkhi and M. Reiter, *Secure and scalable replication in phalanx*, Proc. 17th IEEE Symposium on Reliable Distributed Systems, 1998.

[MR98b] D. Malkhi and M. K. Reiter, *Byzantine quorum systems*, Distributed Computing **11** (1998), no. 4, 203–213.

[PMMS01] K. Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, *The SecureRing group communication system*, ACM Transactions on Information and System Security **4** (2001), no. 4, 371–406.

[PSL80] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, Journal of the ACM **27** (1980), no. 2, 228–234.

[Rab83] M. O. Rabin, *Randomized Byzantine generals*, Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS), 1983, pp. 403–409.

[RB94] M. K. Reiter and K. P. Birman, *How to securely replicate services*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 3, 986–1009.

[RC06] H. V. Ramasamy and C. Cachin, *Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast*, Proc. OPODIS 2005 — 9th Intl. Conference on Principles of Distributed Systems (J. H. Anderson, G. Prencipe, and R. Wattenhofer, eds.), Lecture Notes in Computer Science, no. 3974, Springer, December 2006, pp. 88–102.

[RSS07] H. V. Ramasamy, M. Seri, and W. H. Sanders, *Brief announcement: The CoBFIT toolkit*, Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC), 2007, pp. 350–351.

[Sch90] F. B. Schneider, *Implementing fault-tolerant services using the state machine approach: A tutorial*, ACM Computing Surveys **22** (1990), no. 4, 299–319.

[SG02] V. Shoup and R. Gennaro, *Securing threshold cryptosystems against chosen ciphertext attack*, Journal of Cryptology **15** (2002), no. 2, 75–96.

[Sha79] A. Shamir, *How to share a secret*, Communications of the ACM **22** (1979), no. 11, 612–613.

[Sma03] N. Smart, *Cryptography — an introduction*, McGraw-Hill, London, 2003.

[Tou84] S. Toueg, *Randomized Byzantine agreements*, Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC), 1984, pp. 163–178.

[YMAD03] J. Yin, J.-P. Martin, A. V. L. Alvisi, and M. Dahlin, *Separating agreement from execution in Byzantine fault-tolerant services*, Proc. 19th ACM Symposium on Operating System Principles (SOSP), 2003, pp. 253–268.

[ZSvR02] L. Zhou, F. B. Schneider, and R. van Renesse, *COCA: A secure distributed online certification authority*, ACM Transactions on Computer Systems **20** (2002), no. 4, 329–368.