3 Byzantine Broadcasts and Randomized Consensus

3.1 Model

The model considered here is motivated by practice. There are n parties or replicas $\mathcal{P} = \{P_1, \ldots, P_n, \}$, of which up to t are subject to *Byzantine* faults. The parties are connected pairwise by reliable authenticated channels. Protocols may use cryptographic methods, such as public-key cryptosystems 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.

3.2 Broadcasts

This section presents three *broadcast primitives*. Most published Byzantine consensus and atomic broadcast protocols implicitly or explicitly use one or more of them. As a high-level protocol usually invokes multiple instances of a broadcast primitive, every implementation-level message generated by a primitive is tagged by an identifier of the instance in practice (and where applicable, the identifier is also included in every cryptographic operation).

Every broadcast instance has a designated sender P_s, 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 matters, we assume that the sender is a member of the group and that all requests are unique.

3.2.1 Consistent broadcast

In *consistent broadcast*, the sender first executes *c-broadcast* with request m and thereby starts the protocol. All parties terminate the protocol by executing *c-deliver* with request m. Consistent broadcast ensures only that the delivered request is the same for all receivers. In particular, when the sender is faulty, it does not guarantee that *every* party terminates and delivers a request. The following definition models one instance of consistent broadcast.

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: Every correct party *c-delivers* at most one request. Moreover, if the sender P_s is correct, then the request was previously *c-broadcast* by P_s.

The following protocol, called *authenticated broadcast*, implements consistent broadcast with a *quadratic* number of messages and a latency of two message exchanges. Intuitively, the sender distributes the request to all parties and expects $\lceil \frac{n+t+1}{2} \rceil$ parties to act as witnesses for the request to the others. Every correct party witnesses for the sender's request by echoing it to all parties; this authenticates the request. When a correct party has received $\lceil \frac{n+t+1}{2} \rceil$ such echos with the same request m, then it *c-delivers* m. In all *upon* clauses of the protocol description below that involve receiving a message, only the first message from each party is considered.

Algorithm 2 (Authenticated broadcast or Srikanth-Toueg broadcast [ST87]).

upon *c-broadcast*(m): // only P_s
  send message (SEND, m) to all

upon *receiving a message* (SEND, m) *from* P_s:
  send message (ECHO, m) to all

upon *receiving* $\lceil \frac{n+t+1}{2} \rceil$ *messages* (ECHO, m):
  c-deliver(m)

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

Proof. Validity and integrity are straightforward to verify. Consistency follows from the observation that in order to *c-deliver* a request m, a correct party needs to receive $\lceil \frac{n+t+1}{2} \rceil$ ECHO messages containing m. A set that corresponds to this number of distinct parties is also called a *Byzantine quorum* [MR98]. Because there are only n distinct parties, every two Byzantine quorums overlap in at least one correct party. Hence, if some distinct party *c-delivers* a request m', it has received a quorum of ECHO messages containing m', and since the correct party in the intersection of the two quorums sent the same ECHO message to all parties, it follows $m = m'$. □

The authenticated broadcast protocol has a *latency* of two message exchanges, its *message complexity* is $O(n^2)$, and its *communication complexity* is $O(n^2|m|)$, where $|m|$ denotes the length of m.

Another well-known implementation of consistent broadcast exists under the name of *echo broadcast*. Compared to the authenticated broadcast protocol, it uses digital signatures and achieves smaller communication complexity (only a *linear* number of messages) at the cost of higher latency. Again, the idea is that the sender distributes the request to all parties and expects $\lceil \frac{n+t+1}{2} \rceil$ parties to act as witnesses. Compared to the authenticated broadcast protocol, the witnesses authenticate a request not by sending an ECHO message to all parties but by issuing a signed statement, which they disseminate through the sender.

Algorithm 4 (Echo broadcast [Rei94]). All parties use digital signatures (see Chapter 1, Section 1.2.2). This is the code for P_i.

upon *c-broadcast*(m): // only P_s
  send message (SEND, m) to all

upon *receiving a message* (SEND, m) *from* P_s:
  $\sigma \leftarrow$ *sign*$_i(\text{ECHO}\|s\|m)$
  send message (ECHO, m, σ) to P_s

upon *receiving* $\lceil \frac{n+t+1}{2} \rceil$ *messages* $(\text{ECHO}, m, \sigma_j)$ *with valid* σ_j, i.e., // only P_s
    such that *verify*$_j(\text{ECHO}\|s\|m, \sigma_j)$:
  let Σ be the list of all received signatures σ_j
  send message $(\text{FINAL}, m, \Sigma)$ to all

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

Theorem 5. *Algorithm 4 implements consistent broadcast for* $n > 3t$.

Proof. The difference to Algorithm 2 lies only in the use of digital signatures. Consistency follows from the same observation as before: the request m in any FINAL message with $\lceil \frac{n+t+1}{2} \rceil$ valid signatures in Σ is unique. Because there are only n distinct parties, every two Byzantine quorums of signers overlap in at least one correct party. □

The latency of the echo broadcast protocol is three message exchanges. Its message complexity 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 [Sho00], the communication complexity can be reduced to $O(n(k + |m|))$ bits [CKPS01].

3.2.2 Reliable broadcast

Reliable broadcast is characterized by an *r-broadcast* event and an *r-deliver* event analogous to consistent broadcast. Reliable broadcast additionally ensures agreement on the delivery of the request in the sense that either all correct parties deliver some request or none delivers any request; this property has been called *totality* [CKPS01]. 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 6 (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.

The classical implementation of reliable broadcast by Bracha [Bra87] uses two rounds of message exchanges among all parties. Intuitively, it works as follows. After receiving the request from the sender, every party echoes the request to all. After receiving such echos from a Byzantine quorum of parties, a party indicates to all others that it is ready to deliver the request. When a party receives a sufficient number of those indications, it delivers the request.

Algorithm 7 (Bracha broadcast [Bra87]).

upon *r-broadcast*(m): // only P_s
  send message (SEND, m) to all

upon *receiving a message* (SEND, m) *from* P_s:
  send message (ECHO, m) to all

upon *receiving* $\lceil \frac{n+t+1}{2} \rceil$ *messages* (ECHO, m) *and not having sent a* READY *message:*
  send message (READY, m) to all

upon *receiving* $t+1$ *messages* (READY, m) *and not having sent a* READY *message:*
  send message (READY, m) to all

upon *receiving* $2t+1$ *messages* (READY, m):
  r-deliver(m)

Theorem 8. *Algorithm 7 implements reliable broadcast for* $n > 3t$.

Proof. Consistency follows from the same argument as in Theorem 3, 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 latency of the Bracha broadcast protocol is three messages, its message complexity is $O(n^2)$, and its communication complexity is $O(n^2|m|)$. It does not need digital signatures, which are usually computationally expensive operations, but incurs $O(n^2)$ messages.

Several complex consensus and atomic broadcast protocols use one of the three broadcast primitives, and one can often substitute either primitive for another one in these protocols, with appropriate modifications. Selecting one of these primitives for an implementation 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.

3.3 Randomized Byzantine Consensus

A more involved primitive is a protocol to reach consensus on a common value despite Byzantine faults. It is a prerequisite for implementing atomic broadcast, which provides a total order on multiple requests delivered using reliable broadcast.

The *Byzantine consensus* problem, also called *Byzantine agreement*, is characterized by two events *propose* and *decide*; every party executes *propose*(v) to start the protocol and *decide*(v) to terminate it with a value v. We consider *binary* consensus, where the values are bits.

Definition 9 (Byzantine consensus). A protocol for binary Byzantine consensus 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 asynchronous protocol solving Byzantine consensus has executions that do not terminate. State machine replication in asynchronous networks is also subject to this limitation. Roughly at the same time, however, randomized protocols to circumvent this impossibility were developed [Rab83, Ben83, Tou84]. They make the probability of non-terminating executions arbitrarily small. 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 10 is a consensus protocol that terminates with probability 1. It assumes that a trusted dealer has *shared* a sequence $s_0, s_1, \ldots$ of random bits, called *coins*, among the parties, using $(t+1)$-out-of-n secret sharing (see Chapter 2). A party can access the coin s_r using a *recover*(r) operation, which involves a protocol that exchanges some messages to reveal the shares to all parties, and gives the same coin value to every party.

The protocol works as follows. Every party maintains a value v, called its *vote*, and the protocol proceeds in global asynchronous rounds. Every round consists of two voting steps among the parties with all-to-all communication. In the first voting step, the parties simply exchange their votes, and every party determines the majority of the received votes. In the second voting step, every party relays the majority vote 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. The set Π contains messages and signatures from the first voting step. 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 tally 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.

Algorithm 10 (Binary randomized Byzantine consensus [Tou84].). The two *upon* clauses below are executed concurrently.

upon *propose*(v):
  $r \leftarrow 0$
  $decided \leftarrow$ FALSE
  loop
    send the signed message $(1\text{-VOTE}, r, v)$ to all
    wait for receiving properly signed $(1\text{-VOTE}, r, v')$ 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 $(2\text{-VOTE}, r, v, \Pi)$
    wait for *r-delivery* of $(2\text{-VOTE}, r, v', \Pi)$ messages from $n - t$ distinct
        senders with valid signatures in Π and correctly computed v'
    $b \leftarrow$ value v' 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 (DECIDE, v) to all // note that $v = s_r = b$
    $r \leftarrow r + 1$

upon *receiving* $t + 1$ *messages* (DECIDE, b):
  if $decided =$ FALSE **then**
    send the message (DECIDE, b) to all
    $decided \leftarrow$ TRUE
    decide(b)

Lemma 11. *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 12. *In round $r \geq 0$, the following holds:*

1. *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.*
2. *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 their 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 1 now follows upon noticing that a correct party only sends a DECIDE message for v_0 when $v_0 = b = s_r$.

Claim 2 follows because the first correct party to assign b and c does so *before* any information about s_r is known (to the adversary). To see this note that at least $t + 1$ shares are needed for recovering s_r, but a correct party only reveals its share *after* assigning b and c. Thus, s_r and v_0 are statistically independent and $s_r = v_0$ holds with probability $\frac{1}{2}$. □

Theorem 13. *Algorithm 10 implements binary Byzantine consensus for $n > 3t$, where termination holds with probability 1.*

The theorem follows easily from the two lemmas. The protocol achieves optimal resilience because reaching agreement in asynchronous networks with $t \geq n/3$ Byzantine faults is impossible, despite the use of digital signatures [Tou84]. Since Algorithm 10 terminates with probability at least $\frac{1}{2}$ in every round, the expected number of rounds is two, and the expected number of messages is $O(n^3)$.

Using cryptographic randomness. The problem with Algorithm 10 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. Malkhi and Reiter [MR00] observe that a non-interactive and deterministic threshold signature scheme already yields unpredictable bits.

More generally, one may obtain the coin value s_r from the output of a distributed *pseudorandom function (PRF)* [KL07] evaluated on the round number r and the protocol instance identifier. 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 10 is quite practical and has expected message complexity $O(n^3)$. It can further be improved to a randomized asynchronous Byzantine consensus protocol with $O(n^2)$ expected messages by replacing the reliable broadcasts with a two-phase voting structure that uses *justifications* for each vote obtained through threshold signatures on earlier votes [CKS05].

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)*, Advances in Cryptology: 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.

[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.

[KL07] J. Katz and Y. Lindell, *Introduction to modern cryptography: Principles and protocols*, Chapman & Hall/CRC, 2007.

[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-DCCS), 2006, pp. 568–577.

[MR98] D. Malkhi and M. K. Reiter, *Byzantine quorum systems*, Distributed Computing **11** (1998), no. 4, 203–213.

[MR00] ———, *An architecture for survivable coordination in large distributed systems*, IEEE Transactions on Knowledge and Data Engineering **12** (2000), no. 2, 187–202.

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

[Rei94] M. K. Reiter, *Secure agreement protocols: Reliable and atomic group multicast in Rampart*, Proc. 2nd ACM Conference on Computer and Communications Security (CCS), 1994, pp. 68–80.

[Sho00] V. Shoup, *Practical threshold signatures*, Advances in Cryptology: EUROCRYPT 2000 (B. Preneel, ed.), Lecture Notes in Computer Science, vol. 1087, Springer, 2000, pp. 207–220.

[ST87] T. K. Srikanth and S. Toueg, *Simulating authenticated broadcasts to derive simple fault-tolerant algorithms*, Distributed Computing **2** (1987), 80–94.

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