Distributed algorithms: Lesson 1 on the broadcast abstraction

Emmanuelle Anceaume

Directrice de recherche CNRS
IRISA
http://people.irisa.fr/Emmanuelle.Anceaume/
Introduction

- Back in the 1970s: computers were used in aircraft control:
  - Need to replicate services on physically distributed systems
  - The question was: how to make sure that the multiple machines share a consistent view and make consistent decisions?
- To better understand this problem, NASA sponsored the Software Implemented Fault-Tolerance (SIFT) project
  - Objective: Build a resilient aircraft control system that tolerated failures of its components
  - The famous Lamport’s « Byzantine Generals » problem came out of the project
  - This work laid the foundations of distributed computing
Since then, fault tolerant distributed algorithms have been deployed in many application settings.

In the past 20 years, companies like Google and Facebook have adopted agreement algorithms as part of their computing infrastructure (Google wallet/Google pay).

Starting in 2009, Bitcoin and various subsequent cryptocurrencies came around.

The cryptocurrencies achieved a new breakthrough in distributed computing. They showed for the first time that one may reach a pretty strong agreement in an open large-scale distributed system.

In this course, I will present you some of the theoretical foundations of agreement problems.
Distributed computing

- Solving a non trivial common problem with distinct entities that only have a local view of the system
- Uncertainty due to failures and asynchrony
- Distributed computing is significantly simplified if we encapsulate the difficulty of robust cooperation within abstractions
Goal of distributing computing

- Identifying problems that are abstractions of those that arise in a variety of distributed situations
  - Sending an information to everyone
  - Agreeing on a common value
  - Synchronizing clocks
  - ...

- Stating precisely these abstractions in terms of properties

- Designing and analyzing efficient distributed algorithms to solve them
Abstractions: which properties on may expect in terms of ...
... communication among entities

- Reliable broadcast
- Causal broadcast
- Atomic broadcast
- R/W register
... synchronization

- Reliable broadcast
- Clock synchronization
- Snapshot
- R/W register
- Causal broadcast
- Mutual exclusion
- Atomic broadcast
... agreement

- Reliable broadcast
- Causal broadcast
- Atomic broadcast
- R/W register
- Clock synchronization
- Mutual exclusion
- Snapshot
- Lattice agreement
- Consensus
- Replicated state machine
- Leader election
- Atomic broadcast
When designing an algorithm, you need to precisely state the characteristics of the system in which this algorithm will be deployed

- **Collection of computing entities**
  - \( \Pi = \{p_1, \ldots, p_n, \ldots\} \), where \( p_i \) is called an entity, a process, a party, or a player
  - Processes are uniquely identified or not?

- **Synchrony model**
  - concerns the existence and knowledge of temporal bounds

- **Failure model**
  - concerns the behaviors of the system entities

- **Communication model**
  - concerns the way entities communicate
Computing entities

- Identifying entities
  - Entities are identified with unique identifiers
  - There is a some public key infrastructure (PKI) that allows entities to sign their messages
  - Entities are anonymous

- Composition of the system $\Pi$
  - $\Pi$ is fixed before the execution of the algorithms
  - $\Pi$ evolves with time, i.e., during the execution of the algorithms
Synchrony model

- Synchronous
- Asynchronous
- Partially asynchronous
Synchronous model

- The algorithm is designed assuming that it will be executed in a synchronous distributed system.

- A synchronous distributed system is characterized by the existence and knowledge of temporal bounds:
  - On the time it takes for a message to be received by its recipient.
  - On the time it takes for a process to execute a given computation.
Synchronous model: round-based algorithm

- The progress of an algorithm is governed by a global clock (whose domain of values are integers).
- We can use this global time to design distributed algorithms as a sequence of rounds: *round-based algorithms*.
- Processes collectively execute these rounds, each round corresponding to a value of the global clock.
- In each round:
  - A process receives all the messages sent in the previous round, executes local computations, and based on these computations sends messages.
When a process is in round \( r+1 \) it has received and processed all the messages sent in round \( r \) for which it was recipient.
Asynchronous model

- The algorithm is designed assuming that it will be executed in an asynchronous distributed system.
- An asynchronous distributed system is characterized by the absence of temporal bounds.
  - Is Internet an asynchronous system?
Asynchronous model: round-based algorithm

- The progress of a process is governed by the messages it receives and its own local computations.
- A round is no more delimited by the time, but by the number of messages received.
- In each round:
  - A process receives messages, executes local computations, and possibly send messages.
Asynchronous model: round-based algorithm
The partially synchronous model aims to find a middle ground between these two models. The assumption is that there exists some known finite time $\Delta$ and a special event GST (Global Stabilization Time) such that:

- The GST event eventually happens after some unknown finite time
- Any message sent at time $t$ must be delivered by time $\Delta + \max(GST, t)$

Informally, in the partially synchronous model, the system behaves asynchronously till GST and synchronously after GST.

The GST event can be delayed by any finite amount of time.

No protocol can explicitly detect that GST has occurred: there is no external signal that tells you that GST happened.
The synchronous model seems to be good enough, does not it?

- e.g., we may assume that a message sent over a network will take say at most 2 minutes to be delivered by all the parties

- Is there some trade-off?
The asynchronous model forces protocol designers to assume nothing about network delays.

This often gives rise to robust protocols:
- Since protocols do not depend on any time bound, message delays cannot cause unexpected safety violations.
- Since protocols cannot use any fixed values for timeouts, they must inherently adapt to the actual latency of the system.

Main issue:
- Protocols in this model are often more complex and thus harder to reason about.
- There are many complexity gaps between synchronous and asynchronous model.
Partial synchrony captures the intuition that synchrony assumptions can be temporarily violated, e.g., denial of service attack.

The idea is to design protocols that are always safe (even when the system is asynchronous) but provide liveness (e.g. termination) after GST (when the system becomes synchronous).
Implementing abstractions

Asynchronous
- Reliable broadcast
- Consensus
- Causal broadcast
- Clock synchronization
- Leader election
- Lattice agreement
- Atomic broadcast
- Replicated state machine
- Mutual exclusion
- R/W register
- Nakamoto Consensus

Timing model

Synchronous
Processes can commit different types of failures

A faulty process is a process whose functionality is incorrect

There are different ways for a processor to be incorrect: from crash failures to Byzantine failures
Crash failures

- A process stops prematurely its execution
- Before crashing, it executes correctly its local algorithm
Byzantine failures

- Byzantine failures
  - A process does not follow its local algorithm
  - We cannot make any assumptions on the way it behaves
  - e.g. send $m_1 \neq m_2$ to $p_i$ and $p_j$ respectively although they should be identical messages
  - Byzantine processes can collude to mislead non-faulty processes

It is important to observe that one never knows in advance which processes will fail during the execution of the algorithm.
In each round

- Each non-faulty process executes local computational steps, sends messages to its neighbors, and delivers all the messages sent to it during the round.
- A faulty process may not complete its round in its entirety (e.g., its messages may not have been received by all its neighbors), or may behave arbitrarily.
Execution of a synchronous algorithm in presence of crash failures

Crashed processes do not execute anymore
Execution of a synchronous algorithm in presence of Byzantine failures

A Byzantine process behaves arbitrarily
Implementing abstractions

Reliable broadcast
Causal broadcast
Atomic broadcast
R/W register
Clock synchronization
Leader election
Causal broadcast
Snapshot
Lattice agreement
Replicated state machine
Mutual exclusion
Atomic broadcast
Nakamoto Consensus
Byzantine Consensus
Failure model
Timing model
Asynchronous
Crash
Synchronous
Failure model
Nakamoto Consensus
R/W register
Clock synchronization
Leader election
Causal broadcast
Snapshot
Lattice agreement
Replicated state machine
Mutual exclusion
Atomic broadcast
Nakamoto Consensus
Byzantine Consensus
Failure model
Timing model
Asynchronous
Crash
Synchronous
Failure model
Nakamoto Consensus
R/W register
Clock synchronization
Leader election
Causal broadcast
Snapshot
Lattice agreement
Replicated state machine
Mutual exclusion
Atomic broadcast
Nakamoto Consensus
Byzantine Consensus
Failure model
Timing model
Asynchronous
Crash
Synchronous
Failure model
Nakamoto Consensus
R/W register
Clock synchronization
Leader election
Causal broadcast
Snapshot
Lattice agreement
Replicated state machine
Mutual exclusion
Atomic broadcast
Nakamoto Consensus
Byzantine Consensus
Failure model
Timing model
Asynchronous
Crash
Synchronous
Failure model
Nakamoto Consensus
R/W register
Clock synchronization
Leader election
Causal broadcast
Snapshot
Lattice agreement
Replicated state machine
Mutual exclusion
Atomic broadcast
Nakamoto Consensus
Byzantine Consensus
Failure model
Timing model
Asynchronous
Crash
Synchronous
Processes can communicate
- By sending and receiving messages over a communication medium
- By writing and reading shared variables
- In the first part of the course we will design algorithms in the former model, and in the second part we will consider the latter one
Implementing abstractions

- Reliable broadcast
- Causal broadcast
- Atomic broadcast
- R/W register
- Clock synchronization
- Leader election
- Causal broadcast
- Lattice agreement
- Replicated state machine
- Mutual exclusion
- Atomic broadcast
- Nakamoto Consensus
- Replicated state machine
- Lattice agreement
- Mutual exclusion
- Atomic broadcast
- Nakamoto Consensus
- Byzantine
- Failure model
- Crash
- Nakamoto Consensus
- Byzantine
- Failure model
- Crash

Communication model
- Asynchronous
- Synchronous

Timing model
- Asynchronous
- Synchronous

Failure model
- Byzantine
- Crash
For a given execution (run)

- It is not known in advance which processes will fail (in both synchronous and asynchronous models)
- It is not known in advance how long messages will take to be received (asynchronous model)

These uncertainties are modeled by an adversary

The design of a distributed algorithm can be seen as a game between the adversary and the correct processes (non-faulty processes)

- The adversary designs strategies to defeat correct processes
A distributed problem is defined by safety and liveness properties

- Any problem can be defined by two types of properties: safety and liveness properties.
- Liveness (or termination) properties state that « something happens »
  - i.e., a result is calculated.
- Safety properties state that « nothing bad happens »
  - Describe properties that must never be violated (i.e., invariants).
- The decomposition of a problem into safety and liveness properties simplifies its understanding and its proof of correctness.
The reliable broadcast communication abstraction

- Definition of the reliable broadcast communication abstraction
- How can we implement this abstraction
  - in a synchronous environment with crash failures
  - in a synchronous environment with Byzantine failures
  - in an asynchronous environment with crash failures
  - in an asynchronous environment with Byzantine failures
Definition: Reliable broadcast

- **Validity**: If the sender is correct, then all correct processes decide the sender’s value, and only this value.
- **Agreement**: All correct processes decide the same unique value.
- **Termination**: All correct processes eventually decide in a known bounded time.
Implementing the reliable broadcast abstraction in a synchronous system presence of crash failures

- Set of $n$ processes
- Processes have unique ids in the range $[1, \ldots, n]$
- Processes communicate through reliable links in a fully connected point to point network
- Synchronous system
- Up to $t$ crash failures
  - Crash failure: A process fails by halting prematurely. Until it halts, it behaves correctly
Implementing Reliable broadcast with crash: Round-based algorithm

- Processes execute in synchronous rounds
  - Processes first send messages (according to their state), wait to receive messages sent by other processes in the same round, and then change their state accordingly

- In this algorithm, we will use a rotating coordinator
  - A subset of $t+1$ processes cyclically become coordinators for a constant number of rounds each
  - The sender is the first coordinator and its id is 1
  - When a process becomes coordinator, it determines a consistent decision value and tries to impose it on the remaining of the processes
// Initialization
\[ v_i \leftarrow m \text{ if } i \text{ is the sender (i.e., } i = 1) \text{ otherwise } v_i = \perp \]
undecided \leftarrow \text{true} \text{ (boolean value)}

For \( c \leftarrow 1, 2, \ldots t + 1 \) do { // Process \( c \) is coordinator for 3 consecutive rounds
  Round 3c-2:
  if (undecided = true) then sends request(\( v? \)) to \( c \)
  if (\( i = c \) and \( i \) does not receive any request(\( v? \)))
  then \( i \) skips rounds 3c-1 and 3c
  Round 3c-1:
  if (\( i = c \)), \( i \) sends \( v_c \) to all processes
  if (undecided = true and receive \( v_c \)) then do
  \[ v_i \leftarrow v_c \]
  Round 3c:
  if (\( i = c \)) then sends decide to all processes
  if (undecided = true and \( i \) receives decide) then do
  deliver \( v_i \)
  undecided \leftarrow \text{false}
}

Algorithm 1: Reliable broadcast in presence of crash failures
Proof of correction Algorithm 1

Theorem

Algorithm 1 implements the Reliable Broadcast specification in the presence of up to \( t \) crash failures, with \( t < n \)

Recall the definition of reliable broadcast

- **Validity**: If the sender is correct, then all correct processes decide the sender’s value, and only this value
- **Agreement**: All correct processes decide the same unique value
- **Termination**: All correct processes decide in bounded time
Proof of correction Algorithm 1

// Initialization
\[ v_i \leftarrow m \text{ if } i \text{ is the sender (i.e., } i = 1) \text{ otherwise } v_i = \perp \]
\[ \text{state}_i \leftarrow \text{undecided} \]

For \( c \leftarrow 1, 2, \ldots t + 1 \) do { // Process \( c \) is coordinator for 3 consecutive rounds
  Round 3c-2:
  All undecided processes send request(\( v? \)) to \( c \)
  If \( c \) does not receive request(\( v? \)) then it skips rounds 3c-1 and 3c
  Round 3c-1:
  \( c \) sends \( v_c \) to all processes
  All undecided processes \( i \) that receive \( v_c \) do
  \[ v_i \leftarrow v_c \]
  Round 3c:
  \( c \) sends \textit{decide} to all processes
  All undecided processes that receive a \textit{decide} do
  deliver \( v_i \)
  \[ \text{state}_i \leftarrow \text{decided} \]
}

• Termination: All correct process decide (in bounded time)
Lemma (1)

If coordinator $c$ is correct, all processes which have not crashed decide by the end of round $3c$

Proof:

- Suppose that there is some undecided process $i$, which has not crashed by the end of round $3c - 2$
- Process $i$ sends a request to coordinator $c$, which sends its estimate in round $3c - 1$
- All correct processes still undecided receive and update their value with $v_c$
- Thus all correct processes decide by round $3c$. 
Validity is satisfied. The proof is direct from Lemma 1 with $c = 1$.
Proof of correction Algorithm 1

Let $T$ be the round in which the first message $\textit{decide}$ is received by any process, and let $c$ be the coordinator that sent $\textit{decide}$, and $v_c$ be the value of $c$ sent in round $T - 1$.

Lemma (2)

At round $T - 1$, all processes $i$ which did not crash received $v_c$ and set $v_i$ to $v_c$.

Proof:

- Since $c$ sent $\textit{decide}$ in round $T$, it did not crashed in round $T - 1$.
- Since $c$ can only fail by crashing it sent $v_c$ to all processes in round $T - 1$.
- Thus all processes $i$ that did not crash by round $T - 1$ received $v_c$, and set $v_i$ to $v_c$. 
Agreement is satisfied. The proof is direct from Lemma 2
Theorem (1)

Algorithm 1 implements the Reliable Broadcast specification in the presence of up to \( f \leq t \) crash failures, with \( t < n \) and \( f \) is the actual number of crashes. Correct processes accept and become quiescent by round \( 3(f + 1) \), using \( O(nf) \) messages.

Proof

- By definition of \( f \), among \( f + 1 \) processes at least one has not crashed (i.e., is correct)
- Thus by Lemma (1), all the processes that have not crashed decide some value by round \( 3(f + 1) \)
- By round \( 3(f + 1) \), each process has either accepted or is crashed
- By Algorithm 1, only undecided processes send a request to the current coordinator
- Since by round \( 3(f + 1) \) no process will ever send a request to the current coordinator, the system becomes quiescent by round \( 3(f + 1) \)
- No more than \( 3(f + 1)n \) messages are sent.
Improving the time complexity of Algorithm 1

Algorithm 1: Reliable broadcast in presence of crash failures

```plaintext
// Initialization
vi ← m if i = 1 otherwise vi = ⊥
statei ← undecided

For c ← 1,2,...t + 1 do { // Process c becomes coordinator for 2 consecutive rounds
    Round 2c-1:
    All undecided processes i send request(v?) to c
    If c is undecided, sends vc to all
    For all undecided processes i that receive vc do
        set vi ← vc
    Round 2c:
    if c received a request(v?) in round 2c-1, then c sends decide to all processes
    For all undecided processes that receive a decide do
        accepti ← vi
        statei ← decided
}
```
Reliable broadcast in presence of Byzantine processes

- Failure model: Byzantine failures (only for processes)
Assumptions

- Each process has access to a digital signature mechanism: \( \text{sign}(m, i) \)
  is the signature by \( i \) of message \( m \)

Digital signatures provide the following properties

- Authenticity: a valid signature implies that the signer signed the associated message
- Unforgeability: only the signer can give a valid signature for the associated message
- Non-reusability: the signature of a message can not be used on another message
- Non-repudiation: the signer can not deny having signed a message that has valid signature
- Integrity: ensure the contents have not been modified
Theorem

There exists a protocol for solving reliable broadcast against any adversary controlling $t < n - 1$ processes in $t + 1$ rounds.
Let us suppose that $t = 1$, and try to implement reliable broadcast in 2 rounds.
BRB protocol (run by processes i and n ≥ 3) {

Initialization

\[ V_i = \text{emptyset} \]

round 1:

sender sends \((v, \text{sign}(v,1))\) to all processes

round 2:

if process \(i\) has received \((v, \text{sign}(v,1))\) from sender then

\[ V_i \leftarrow v \]

process \(i\) sends \((v, \text{sign}(v,1))\) to all the processes

if process \(i\) receives \((v, \text{sign}(v,1))\) from anyone then

\[ V_i \leftarrow v \]

if \(V_i\) contains a single value \(v\) then

\begin{align*}
\text{decide } v \\
\text{otherwise} \\
\text{decide } \bot
\end{align*}

}\}

Algorithm 2
Proof of Algorithm 2

- **Validity**: Recall that this property focuses executions in which the sender is correct. Due to signature authenticity\(^1\), all honest processes will see exactly a single value, and thus they will decide it.

- **Termination**: All correct processes decide at the end of round 2.

- **Agreement**:
  - If a malicious sender sends a value to some processes in round 1, all correct processes will receive it at the end of round 2, and decide this value.
  - If a malicious sender sends two (or more) different values in round 1, then all correct processes will receive these values at the end of round 2, and decide \(\bot\).

So does this protocol work?

---

1. A valid signature implies that the signer signed the associated message.
No. The problem is that Agreement does not hold. If the malicious sender does not send any value in round 1, but sends its value to some processes in round 2 then some honest will receive it and will decide it, while the others will decide ⊥.

The core problem is that you may learn the value in the last round but you are not sure that all other correct processes will also receive the value. You cannot forward your message because this is the last round.

Dolev and Strong show an elegant way to guarantee agreement even if the value decided is revealed in the last round of the protocol.
Algorithm 3

BRB protocol (run by processes $i$ and $n \geq 3$) {

Initialization

\[ V_i = \text{emptyset} \]

round 1:

sender sends \((v, \text{sign}(v,1))\) to all processes

round 2:

if process $i$ has received \((v, \text{sign}(v,1))\) from sender then

\[ V_i \leftarrow v \]

process $i$ sends \((v, \text{sign}(v,1), \text{sign}(v,i))\) to all the processes

if process $i$ receives \((v, \text{sign}(v,1), (\text{sign}(v,j))\) from $j \neq 1$ and $j \neq i$ then

\[ V_i \leftarrow v \]

if $V_i$ contains a single value $v$ then

decide $v$

otherwise

decide $\perp$

}

Algorithm 3
Proof of Algorithm 3

- **Validity** same as Attempt 1
- **Termination** same as Attempt 1
- **Agreement** The algorithm is resilient to 1 Byzantine process: a process decides in the last round if $t + 1 = 2$ processes signed it. If there are 2 signatures then 1 is from a correct process, so all correct processes will receive it by the end of the round.

The principle is to only accept a value in the last round if its content can certify that all processes have received this value.
The idea of Dolev and Strong is to implement this idea via signature chains.

Signature chains have two essential properties that allow processes to agree on a value at the end of $t + 1$ rounds, where $t$ is the maximal number of crash failures.

A signature chain consists of signatures from distinct processes.

- If some process receives a message with $t + 1$ signatures for $v$, then it must be the case that it contains the signature from some correct process.
- This correct process can send $v$ to all processes.

In round $r$, a process will only accept a signature chain with $r$ signatures.

- So even if a Byzantine forms a signature chain with $t$ signatures, then if it sends this chain in round $t + 1$ no correct process will accept this chain.
Messages in the protocol are signature chains where a $k$-signature chain is defined recursively:

Definition ($k$-signature chain)

- A 1-signature chain on $v$ by process $i$ is a pair $(v, sign(v, i))$
- A $k$-signature chain on $v$ by process $i$ is a pair $(m, sign(m, i))$, where $m$ is a $(k-1)$-signature chain that does not contain $i$’s signature
**Definition (validity of a $k$-signature chain)**

A $k$-signature-chain on $v$ received by process $i$ by the end of round $r = k$ is **valid** if $i$ has received a message that contains value $v$ and

- The signature chain contains $k$ signatures
- All signers in the chain are distinct
- The first signer of the signature chain is the sender
- Signature of process $i$ is not in the signature-chain
The Dolev-Strong Byzantine protocol\(^2\)

Algorithm 4

Proof of Algorithm 4 (1)

- Validity:
  - The correct sender will only sign and send a unique value $v$ to all processes in round 1.
  - By the authenticity of digital signatures, no other valid signature-chain can exist.
  - In round 2 all correct processes $i$ will consider the received message as a valid 1-signature chain.
  - They will add $v$ to $V_i$, sign it and send a 2-signature chain on $v$ to all processes.
  - $\forall$ correct $i$, $|V_i| = 1$ so $i$ will decide $v$.

3. A valid signature implies that the signer signed the associated message.
Proof of Algorithm 4 (2)

- Termination: By construction all correct processes deliver at the end of round $t + 1$
Agreement: We need to prove that if some process delivers some value \( v \) or \( \bot \) then all correct processes will deliver the same value \( v \) or \( \bot \).

Case 1: At the end of round \( r \leq t \), some correct process \( i \) receives for the first time a valid \( r \)-signature chain on \( v \):

1. At the beginning of round \( r + 1 \leq t + 1 \), process \( i \) adds \( v \) to \( V_i \), signs the value and sends a \((r + 1)\)-signature chain on \( v \) to all.
2. At the end of round \( r + 1 \leq t + 1 \) all the correct processes \( j \) will have a \((r + 1)\)-valid signature-chain on value \( v \).
3. \( v \in V_i \)
4. If any correct has more than one value in \( V \) then all correct processes will have more than one value.
5. Thus all correct processes will deliver the same unique value.
Proof of Algorithm 4 (4)

- Agreement: We need to prove that if some process delivers some value \( v \) or \( \bot \) then all correct processes will deliver the same value.

- Case 2: at the end of round \( r = t + 1 \), some correct process \( i \) receives a valid \( (t + 1) \)-signature chain on value \( v \) (i.e., \( i \) cannot convince any other processes).
  1. It must be the case that value \( v \) has been signed by \( (t + 1) \) processes different from \( i \).
  2. By definition, no more than \( t \) processes can be Byzantine.
  3. Thus one of the signing processes \( j \) is correct.
  4. \( j \) has a valid \( t \) signature on \( v \) at round \( r = t \).
  5. Case 1 applies.
Definition: Reliable broadcast

- **Validity** If the sender is correct, then all correct processes decide the sender’s value, and only this value.

- **Agreement** If some correct process decides then all correct processes decide the same unique value.
Asynchronous algorithm

- A distributed asynchronous algorithm is an algorithm designed to be executed in an asynchronous system.
- Asynchronous system
  - No notion of time
- The progress of a process is ensured by
  - Its own computation and
  - The message it receives
- When a process receives a message
  - It processes the message, and according to its local algorithm, possibly sends messages to other processes
Absence of time

A Byzantine quorum is a set of \( \left\lfloor \frac{(n + t)}{2} \right\rfloor + 1 \) processes

**Byzantine quorum properties**

1. Any Byzantine quorum contains no more than \( n - t \) processes (liveness)
2. Any two Byzantine quorums intersect in at least one correct process (safety)

The liveness property implies a constraint on the proportion of faulty processes:

\[
\begin{align*}
    n - t & \geq \left\lfloor \frac{(n + t)}{2} \right\rfloor + 1 > \frac{(n + t)}{2} \\
    n - t & > \frac{(n + t)}{2} \\
    n & > 3t
\end{align*}
\]

Most of the Byzantine tolerant distributed algorithms impose that \( n \geq 3t + 1 \)
Byzantine quorums (2)

- A recurring tool for designing distributed systems that are tolerant to Byzantine processes is the notion of Byzantine quorum
- A Byzantine quorum is a set of $\lceil(n + t)/2\rceil + 1$ processes

**Byzantine quorum properties**

- Any Byzantine quorum contains no more than $n - t$ processes (liveness)
- Any two Byzantine quorums intersect in at least one correct process (safety)

The safety property is derived as follows:

$$|Q_1 \cup Q_2| \leq n$$
$$|Q_1 \cap Q_2| = |Q_1| + |Q_2| - |Q_1 \cup Q_2|$$
$$\geq 2\lceil(n + t)/2\rceil + 1 - n$$
$$> 2(n + t)/2 - n$$
$$> t$$
Asynchronous consistent broadcast with Byzantine processes

- Consistent broadcasts only ensure that the delivered value is the same for all correct processes.
- In particular, it does not guarantee that all correct processes will deliver a value when the sender is faulty.

Consistent broadcast definition

- **Validity**: If the sender is correct and broadcast $v$, then all correct processes eventually deliver $v$, and only $v$.
- **Consistency**: If a correct process delivers $v$, and another correct process delivers $v'$, then $v = v'$. 
An algorithm to implement consistent broadcast with Byzantine processes

This algorithm works as follows:

- The sender broadcasts its value $v$ and expects that a Byzantine quorum of processes will act as witnesses for the value to other processes.
- Each correct process witnesses the sender’s value by echoing it to all processes.
- When a correct process has received a Byzantine quorum (i.e., $\lceil (n + t)/2 \rceil + 1$) echos with the same value $v$, then it delivers $v$. 
Consistent broadcast:\n\begin{verbatim}
Consistent broadcast(v) {
  initialization
  echo = true

  // only the sender executes the following
  if i = sender then
    send (init,v) to all processes

  // process i (including the sender)
  on receiving (init,v) from sender do
    if (echo = true) then
      send (echo,v) to all processes
    echo = false

  on receiving (echo,v) from ⌊(n+t)/2⌋ + 1 distinct processes
    deliver (v)
}
\end{verbatim}
Algorithm 5

Proof of Algorithm 5 (1)

Theorem

Algorithm 5 implements consistent broadcast for $n \geq 3t + 1$

Proof (1):

- **Validity:**
  - The sender is correct
  - Suppose that $t$ Byzantine processes pretend to have received $v'$ from the sender and echo $v'$
  - Then these $t$ echos are not sufficiently many to ever convince a correct process (Line 12)
    - Since we have $\lceil(n + t)/2\rceil + 1 > t$
  - On the other hand, all correct processes will echo the value $v$ of the sender (direct link from all processes and in particular from the sender)
  - All correct processes will receive sufficiently many echos (i.e., a Byzantine quorum) witnessing the value $v$ and will deliver it
    - A Byzantine quorum contains no more than $n - t$ responses
Proof of Algorithm 5 (2)

**Theorem**

*Algorithm 5 implements consistent broadcast for $n \geq 3t + 1***

**Proof (2):**

- **Consistency:**
  - Observe that to deliver a value $v$, a correct process $i$ must have received $\lfloor (n + t)/2 \rfloor + 1$ echos of $v$.
  - If some other correct process delivers another value $v'$, it should have received $\lfloor (n + t)/2 \rfloor + 1$ echos of $v'$.
  - By construction of the algorithm, correct process send an echo at most once.
  - By definition, any two Byzantine quorums intersect at at least one correct process.
  - Thus $v = v'$.

Algorithm 5 has a latency of 2 message exchanges, its message complexity is $O(n^2)$. 
Consistent broadcast is weaker than reliable broadcast
Consider Algorithm 5

- Suppose that $n = \{p_1, p_2, p_3, p_4\}$ and $p_1$ is Byzantine.
- $p_1$ sends (init, $v$) to $p_1$, $p_2$ and $p_3$ and sends $v'$ to $p_4$.
- $p_2$ and $p_3$ send (echo,$v$) to all processes but $p_1$ sends an echo to only $p_2$ and $p_3$.
- $p_4$ sends (echo,$v'$) to all processes.
- $p_1$, $p_2$ and $p_3$ receive (echo,$v$) from a Byzantine quorum of processes and thus deliver $v$ to their application.
- $p_4$ will never be able to receive any Byzantine quorum witnessing a value and thus will never deliver any value to its application.
- Agreement is not satisfied.
Asynchronous reliable broadcast with Byzantine processes

Definition: Reliable broadcast

- **Validity**: If the sender is correct, then all correct processes decide the sender’s value, and only this value.

- **Agreement**: If some correct process decides then all correct processes decide the same unique value.
Bracha’s Byzantine reliable broadcast\(^5\)

Algorithm 7

Proof of Algorithm 7

Theorem

Algorithm 7 implements Byzantine reliable broadcast for $n \geq 3t + 1$

Proof:

- Validity: idem as proof of Algorithm 5
- Agreement:
  - If a correct process deliver $v$, it must have received a vote for $v$ from $2t + 1$ processes
  - Therefore, at least $t + 1$ correct processes have sent a vote message with $v$
  - All these vote messages with $v$ will be received by all correct processes
  - This causes them to send a vote message too with $v$
  - Because $2t + 1 \leq n - t$, all correct processes eventually receive enough vote messages to deliver $v$

Algorithm 7 has a latency of 3 message exchanges, its message complexity is $O(n^2)$. 