

Alle Angaben ohne Gewähr. Keine Garantie auf Vollständigkeit oder Richtigkeit.

1	Introduction	3
1.1	What is a distributed system?	3
1.1.1	Characteristics by van Steen and Tanenbaum	3
1.1.2	Aspects of characteristic 1	3
1.1.3	Aspects of characteristic 2	3
1.1.4	Observations	3
1.2	What makes a distributed system a decentralized system?	3
1.2.1	Three types of Decentralization (Vitalik Buterin)	3
1.2.2	Our definition of decentralized systems	4
1.3	Reasons for decentralization	4
1.3.1	Reasons for architectural decentralization	4
1.3.2	Reasons for political decentralization	4
1.3.3	Reasons for logical decentralization	4
1.4	Challenges of decentralization	4
2	Fundamentals	4
2.1	How to model a distributed system?	4
2.1.1	Processes (Processors) and Messages	4
2.1.2	Links	5
2.1.3	Inter-Process Communication	5
2.1.4	Automata and Steps	5
2.1.5	Safety and Liveness	5
2.2	Assumptions	5
2.2.1	Why do distributed algorithms need assumptions?	5
2.2.2	Uniform/Nonuniform	5
2.2.3	Fault model	5
2.2.4	Fault tolerance	6
2.2.5	Communication: Fair-loss links	6
2.2.6	Communication: Perfect links	6
2.2.7	Timing Models	6
2.3	Time in Asynchronous Systems	6
2.3.1	Logical clocks: Lamport Clocks	6
2.3.2	Hybrid: Partial Synchrony	7
2.4	Combining Abstractions for Assumptions	7
2.4.1	Fail-stop	7
2.4.2	Fail-silent	7
2.4.3	Fail-arbitrary	7
2.5	Problem Statement: Leader Election	7
2.5.1	Problem definition	7
2.5.2	Anonymous rings	7
2.5.3	Leader Election in Anonymous Rings	7
2.5.4	Leader Election in Asynchronous Rings	7
2.6	Problem Statement: Mutual Exclusion	8
2.6.1	Problem definition	8
2.6.2	Lamport Mutual Exclusion	8
2.7	Formalization via Modules	8

2.7.1	Module: Perfect Failure Detector	9
2.7.2	Module: Eventually Perfect Failure Detector	9
2.7.3	Module: Leader Election	9
2.7.4	Module: Eventual Leader Detector	9
2.8	Quorums	10
3	Reliable Broadcast	10
3.1	Module: Best Effort Broadcast	10
3.2	Module: Reliable Broadcast	10
3.3	Byzantine Reliable Broadcast: Synchronous Case	11
3.3.1	Interactive Consistency	11
3.3.2	Challenges	11
3.3.3	Impossibility Result	11
3.3.4	Authenticators	11
3.4	Byzantine Reliable Broadcast: Asynchronous Case	11
3.4.1	Bracha's Reliable Broadcast	11

1 Introduction

1.1 What is a distributed system?

1.1.1 Characteristics by van Steen and Tanenbaum

“A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.” ([van Steen and Tanenbaum, 2016]), this results in two characteristics:

1. **“Collection of autonomous computing elements”**: *“In practice, nodes are programmed to achieve common goals, which are realized by exchanging messages with each other”* ([van Steen and Tanenbaum, 2016])
2. **“Appears as a single coherent system”**: Appears as a single large system

1.1.2 Aspects of characteristic 1

- *“we cannot assume that there is something like a global clock”* ([van Steen and Tanenbaum, 2016]), therefore the **synchronization and coordination** between participants must be worked out
- *“The fact that we are dealing with a collection of nodes implies that we may also need to manage the membership and organization of that collection”* ([van Steen and Tanenbaum, 2016]), therefore we need to think about identities and possible access-restrictions

1.1.3 Aspects of characteristic 2

- *“To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system [...] leading to what is known as middleware”* ([van Steen and Tanenbaum, 2016])

1.1.4 Observations

- Distributing tasks and aggregating a result from them is not easy at all
- The coordination of those tasks is still **centralized**

1.2 What makes a distributed system a decentralized system?

According to **ISO/TC 307**: *“distributed system wherein control is distributed among the persons or organizations participating in the operation of the system”*

1.2.1 Three types of Decentralization (Vitalik Buterin)

Vitalik Buterin defines three types of Decentralization:

- **Architectural (de)centralization**: How many **physical computers** is a system made up of? How many of those computers can it tolerate breaking down at any single time?
- **Political (de)centralization**: How many **individuals or organizations** ultimately control the computers that the system is made up of?
- **Logical (de)centralization**: Does the **interface and data structures** that the system presents and maintains look more like a single monolithic object, or an amorphous swarm? One simple heuristic is: if you cut the system in half, including both providers and users, will both halves continue to fully operate as independent units?

1.2.2 Our definition of decentralized systems

- A decentralized system has political decentralization, where multiples parties are making their own independent decisions (they can still coordinate with each other)
- If a system is architecturally but not politically decentralized, we call it a distributed system
- Decentralized systems can be **logically decentralized or centralized**
- Decentralized systems can be open systems (anybody can participate) or closed systems

1.3 Reasons for decentralization

1.3.1 Reasons for architectural decentralization

- **Latency**
- **Scalability**: Scale number of machines running the system
- Increase **fault tolerance** and **availability**, remove single point of failures
- Increase **attack resistance**, because no central points exist

1.3.2 Reasons for political decentralization

- **Collusion resistance**: It is harder for participants to collude in ways that benefit a small group at the expense of other participants
- **Power** can be distributed “equally”

1.3.3 Reasons for logical decentralization

Logical decentralization is not always possible or even wanted, especially in use cases we cover. Example: **Distributed Ledgers**, where the goal is to have one commonly agreed system state at any point in time.

1.4 Challenges of decentralization

Decentralized systems come with risks/challenges to avoid harm to the system or its participants:

- **Time and synchrony**: Do we have a global clock? Is the communication synchronous or asynchronous?
- **behavior** of nodes: Can we handle arbitrary behavior? How many faulty nodes can we tolerate?
- **Identity**: Do we have an open system? Are nodes (identities) known?

2 Fundamentals

2.1 How to model a distributed system?

We define a **distributed system** as a set of identical processes (or processors) that execute a program. Coordination between the processors is needed. The combination of processes form an application.

2.1.1 Processes (Processors) and Messages

A distributed system or algorithm consists of n **processors** (called nodes/agents/participants) p_0, \dots, p_{n-1} .

Each process runs a local process and the processors cooperate on some common task. They can communicate with each other.

Messages are uniquely identified by the sender using a sequence number of a logical clock.

2.1.2 Links

A **link** (channel) $\{p_i, p_j\}$ connects processors i and j . Links are always considered **bidirectional**.

The network is a collection of all channels, the topology is a pattern of channels (e.g. mesh, star).

2.1.3 Inter-Process Communication

Processors are communicating by **passing messages** to **in-** and **outboxes**. The address is a set of processes. Processors have access to **shared memory**.

2.1.4 Automata and Steps

We can model a distributed algorithm as a distributed collection of **automata** (one per process). Each automata is a state machine with defined states (**configurations**) and state transitions (**step**) that are triggered by an **event**.

2.1.5 Safety and Liveness

- **Safety**: “*Nothing bad has happened, yet*”: If a safety property is violated, we can point to a specific point in time where the violation occurred, **the violation cannot be undone**.
- **Liveness**: “*Eventually something good happens*”: At any time, there is the chance that the property will be satisfied at a later point in time.

2.2 Assumptions

2.2.1 Why do distributed algorithms need assumptions?

Distributed algorithms deal with a lot of uncertainty, therefore we need assumptions to describe the **uncertainty** and **guarantees** of the system. Typical are

- **Process assumptions**: Crash behavior, adherence to the protocol
- **Communication assumptions**: Topology, reliability, attackers
- **Timing assumptions**: Latency, synchrony
- **Cryptographic assumptions**: Cryptographic primitives (e.g. encryption, signatures)
- **Setup assumptions**: What information is available to the participants at the start

2.2.2 Uniform/Nonuniform

- **Uniform**: Total number of processors n is not known to the algorithm
- **Nonuniform**: Each processor knows the total number of processors n

2.2.3 Fault model

The **Fault model** abstracts faults in the processors and channels.

- **Crash fault**: Processor works correctly until it crashes and never recovers
- **Omission fault**: Processor fails to send/receive messages it is supposed to send/receive (e.g. due to buffer overflow)
- **Crashes with recoveries**: Either the process crashes and never recovers or the process keeps crashing and recovering infinitely often
- **Byzantine fault**: Arbitrary behavior, the process can deviate from the protocol in any way

2.2.4 Fault tolerance

The **Fault tolerance** of a system is the number of faulty processes f out of n processes that the system can tolerate while still operating correctly.

2.2.5 Communication: Fair-loss links

Fair-loss links are defined by three properties:

1. *Fair-loss*: If a correct process p infinitely often sends a message m to a correct process q , then q delivers m an infinite number of times
2. *Finite duplication*: If a correct process p sends a message m a finite number of times to a process q , then m cannot be delivered an infinite number of times by q
3. *No creation*: If some process q delivers a message m with sender p , then m was previously sent to q by p

2.2.6 Communication: Perfect links

Perfect links are also defined by three properties:

1. *Reliable delivery*: If a correct process p sends a message m to a correct process q , then q eventually delivers m
2. *No duplication*: No message is delivered by a process more than once
3. *No creation*: If some process q delivers a message m with sender p , then m was previously sent to q by p

Authenticated perfect links are an extension of perfect links.

2.2.7 Timing Models

The **Timing model** describes the timing assumptions of the communication and execution behavior:

- **Synchronous model**: There is a **known upper bound** on processing delays and on message transmission delays
- **Asynchronous model**: There is **no timing assumption at all**. The execution and message delivery happens at an arbitrary speed, **but messages arrive eventually**

2.3 Time in Asynchronous Systems

2.3.1 Logical clocks: Lamport Clocks

Lamport clocks are used to **measure passage of time** in **asynchronous systems**.

- Each process p_i has a **logical clock** l_i , initially set to 0
- Upon an event (sending or receiving a message), l_i is incremented by 1
- When sending a message m , process p_i adds a timestamp $t_m = l_i$ to the message
- When receiving a message m , process p_j increases its timestamp to $\max(l_j, t_m) + 1$

With this, a **happened-before relationship** between events is established. For any two events e_1, e_2 : $e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$.

This defines a **partial order** on the events.

2.3.2 Hybrid: Partial Synchrony

A hybrid between synchrony and asynchrony is **partial synchrony**, which comes in two variants:

- **Eventually synchronous**/Global Stabilization Time (GST): An event GST occurs after some finite time, afterwards time bound Λ holds
- **Unknown Latency (UL)**: The system is always synchronous, but the delay bound Λ is unknown

Algorithms for these models typically increment their estimation of the delay bound dynamically.

2.4 Combining Abstractions for Assumptions

2.4.1 Fail-stop

- Crash faults
- Perfect links
- Perfect failure detector

2.4.2 Fail-silent

- Crash faults
- Perfect links
- No failure detector

2.4.3 Fail-arbitrary

- Byzantine faults
- Authenticated perfect links

2.5 Problem Statement: Leader Election

2.5.1 Problem definition

- Group of processors has to elect one of them as leader
- Exactly one processor enters an elected state, all others enter a non-elected state

2.5.2 Anonymous rings

- Processors have no identifiers
- Each processor has the same deterministic state machine
- Each processor is connected to two other processors

2.5.3 Leader Election in Anonymous Rings

“There is no nonuniform anonymous algorithm for leader election in synchronous rings.”

So, even with these strong assumptions, **leader election is not possible with anonymous participants.**

2.5.4 Leader Election in Asynchronous Rings

Setup assumptions:

- Assign each processor p_i a unique identifier id_i
- Label the two connected processors of p_i as *left* and *right neighbor*

Algorithm 1 Algorithm for Leader Election in Asynchronous Rings**Each processor sends its identifier to its left neighbour.****When a processor receives a termination message, it forwards it to the left neighbour and terminates as non-leader.****Upon receiving message m from right neighbour****if $m < id_i$ then****drop message****else if $m > id_i$ then****forward m to left neighbour****else if $m == id_i$ then****Send termination message to left neighbour****Terminate as leader****end if**

The algorithm sends not more than $O(n^2)$ messages. Nonuniform algorithms for synchronous rings also exist.

2.6 Problem Statement: Mutual Exclusion

2.6.1 Problem definition

Mutual exclusion is a known problem from concurrent computing. We need to ensure that critical sections are only accessible by one processor at a time. The desired sequence is Entry \Rightarrow Critical section \Rightarrow Exit.

The following three objectives should be satisfied:

- **Mutual exclusion:** At most one process can be in the critical section at any time (**Safety**)
- **No Deadlock:** If some processor enters an entry section, some processor will later enter a critical section.
- **No lockout (starvation):** If some processor enters an entry section, **that same processor** will later enter a critical section.

2.6.2 Lamport Mutual Exclusion

Lamport defined a mutual exclusion algorithm based on **logical clocks** [Lamport, 1978], the algorithm is not covered in this summary.

2.7 Formalization via Modules

We can now combine abstractions using **Modules**. A module has a **name**, **events** and **safety & liveness properties**.

An algorithm *implements* a module (and can build on other modules).

2.7.1 Module: Perfect Failure Detector

Events:

- **Indication:** $\langle \mathcal{P}, \text{Crash} \mid p \rangle$: Detects that process p has crashed

Properties:

- **PFD1:** *Strong completeness*: Eventually, every process that crashes is permanently detected by every correct process
- **PFD2:** *Strong accuracy*: If a process p is detected by any process, then p has crashed

Perfect Failure Detector is implemented by “Exclude on Timeout” (not covered in this summary).

2.7.2 Module: Eventually Perfect Failure Detector

Events:

- **Indication:** $\langle \diamond \mathcal{P}, \text{Suspect} \mid p \rangle$: Notifies that process p is suspected to have crashed
- **Indication:** $\langle \diamond \mathcal{P}, \text{Restore} \mid p \rangle$: Notifies that process p is not suspected anymore

Properties:

- **EPFD1:** *Strong completeness*: Eventually, every process that crashes is permanently suspected by every correct process
- **EPFD2:** *Eventual strong accuracy*: Eventually, no correct process is suspected by any correct process

Eventually Perfect Failure Detector is implemented by “Increasing Timeout” (not covered in this summary).

2.7.3 Module: Leader Election

Events:

- **Indication:** $\langle \text{le}, \text{Leader} \mid p \rangle$: Indicates that process p is elected as leader

Properties:

- **LE1:** *Eventual detection*: Either there is no correct process, or some correct process is eventually elected as the leader
- **LE2:** *Accuracy*: If a process is leader, then all previously elected leaders have crashed

Leader Election is implemented by “Monarchical Leader Election” (not covered in this summary).

2.7.4 Module: Eventual Leader Detector

Events:

- **Indication:** $\langle \Omega, \text{Trust} \mid p \rangle$: Indicates that process p is trusted to be leader

Properties:

- **ELD1:** *Eventual accuracy*: There is a time after which every correct process trusts some correct process
- **ELD2:** *Eventual agreement*: There is a time after which no two correct processes trust different processes

Eventual Leader Detector is implemented by “Monarchical Eventual Leader Detection” (not covered in this summary).

2.8 Quorums

A **Quorum** is a set of processes with special properties. They are used for fault-tolerant algorithms. Dealing with N crash-fault processes, **a quorum is any majority of processes**.

Assumption: There is a quorum of non-faulty processes (number of faulty processes $f < N/2$).

Properties:

- Two quorums intersect in at least one process
- In every quorum is at least one correct (non-faulty) process

Dealing with N arbitrary-fault processes (byzantine): To maintain the second property, a quorum needs to be a set of more than $\frac{N+f}{2}$ processes. We call a set of more than $\frac{N+f}{2}$ a **byzantine quorum**.

When the required property is that there exists a Byzantine quorum of correct processes, $3f < N$ needs to hold.

3 Reliable Broadcast

Reliable broadcast is used to **share information** among processors without losses, duplicates, etc.

3.1 Module: Best Effort Broadcast

Events:

- **Request:** $\langle \text{beb}, \text{Broadcast} \mid m \rangle$: Broadcast a message m to all processes
- **Indication:** $\langle \text{beb}, \text{Deliver} \mid p, m \rangle$: Deliver a message m broadcast by process p

Properties:

- **Liveness:**
 - *Validity*: If a correct process p broadcasts m , then every correct process eventually delivers m
- **Safety:**
 - *No duplication*: No message is delivered more than once
 - *No creation*: If a process delivers a message m with sender s , then m was prev. broadcast by s

Best Effort Broadcast is implemented by “Best Effort Broadcast Algorithm” (not covered in this summary).

3.2 Module: Reliable Broadcast

Events:

- **Request:** $\langle \text{rb}, \text{Broadcast} \mid m \rangle$: Broadcast a message m to all processors
- **Indication:** $\langle \text{rb}, \text{Deliver} \mid p, m \rangle$: Deliver a message m broadcast by processor p

Properties:

- **Validity**: If a correct processor p broadcasts m , then p eventually delivers m
- **No duplication**: No message is delivered more than once
- **No creation**: If a processor delivers m from sender s , then m was previously broadcast by s
- **Agreement**: If a message m is delivered by some correct processor, then m is eventually delivered by every correct processor

Reliable Broadcast is implemented by “Reliable Broadcast Algorithm” (not covered in this summary).

3.3 Byzantine Reliable Broadcast: Synchronous Case

Problem: Components in a distributed system don't always fail "orderly" (crash-fault), but they can fail in arbitrary and potentially malicious ways. Therefore we need to focus on **Byzantine failures**.

3.3.1 Interactive Consistency

There are $i = 1, \dots, n$ independent processors, at most f of which exhibit **arbitrary faults**. Non-faulty processors have a private value v_i .

Assumptions:

- **A1: Reliability:** Sent messages are delivered correctly
- **A2: Authenticity:** The receiver of a message knows the sender
- **A3: Synchrony:** Absence of a message is detectable

The goal is **Interactive Consistency**: Processors compute a vector such that

- Non-faulty processors compute a n -dimensional vector
- Non-faulty processors compute exactly the same vector
- The values corresponding to non-faulty processors are their private value
- The values of faulty processors can be arbitrary, but must be consistent

3.3.2 Challenges

Faulty processors can lie about themselves and others:

- **Equivocation:** A processor sends different/contradicting information to different processors
- **Relay Forgery:** A processor passes on different information than received

3.3.3 Impossibility Result

It can be shown that $3f$ processors are insufficient to overcome f faults. **Interactive consistency** between n processors can be achieved if and only if $n > 3f$ with f faults.

Oral Messages Algorithms (not covered in this summary) achieve interactive consistency.

3.3.4 Authenticators

The results seen rely on the assumptions of byzantine processors and the timing model.

Authenticators are metadata that ensure the **authenticity** and **integrity** of data (by using cryptography). They can be used to improve the results.

3.4 Byzantine Reliable Broadcast: Asynchronous Case

3.4.1 Bracha's Reliable Broadcast

Bracha's Reliable Broadcast [Bracha, 1987] is an efficient reliable broadcast algorithm for **asynchronous** systems.

Assumptions:

- System consists of n processes communicating via a *reliable message system*
- The system requires $n \geq 3f + 1$ processes to tolerate f byzantine faults

Objectives: One sender p wants to broadcast a value v . The algorithm satisfies the following properties:

- If p is correct, then all correct processes agree on v
- If p is faulty, then either all correct processes agree on the same value or none of them accepts any value from p

Bracha's Reliable Broadcast has a message complexity of $O(n^2)$. The algorithm is not covered in this summary.

References

- [Bracha, 1987] Bracha, G. (1987). Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143.
- [Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.
- [van Steen and Tanenbaum, 2016] van Steen, M. and Tanenbaum, A. S. (2016). A brief introduction to distributed systems. *Computing*, 98(10):967–1009.