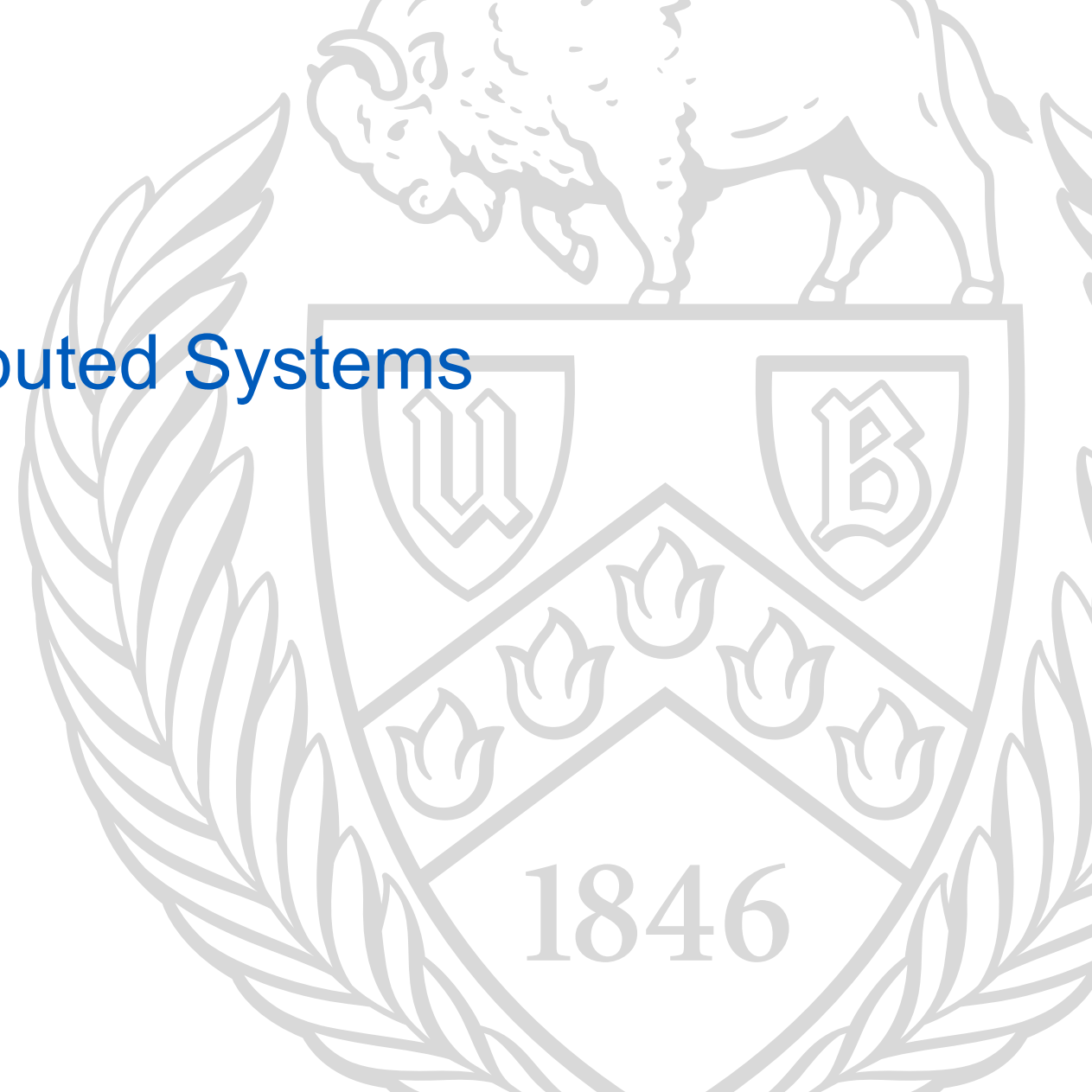


Consensus

CSE 486/586: Distributed Systems

Ethan Blanton

Computer Science and Engineering
University at Buffalo



Consensus

Consensus is an **agreement** between processes on some **state**.

Typically, the **value of a variable**.

Consensus requires that every **non-faulty process** has the same view of the state.

Faulty processes may diverge.

Impossibility of Consensus

It is **provably impossible** [2] to achieve consensus in an asynchronous system if either:

- Any process can fail
- Arbitrary messages can be lost

Nonetheless, we **often use** consensus in practice!

Using Consensus

We have already seen consensus in several protocols:

- The **message priority** in ISIS atomic broadcast
- The **elected leader** in election protocols
- The **happens before** relationship between two events in a vector clock system

We will see many more.

One-bit Consensus

Consensus is often modeled on **a single bit**:

Every non-faulty process agrees on a value $v \in \{0, 1\}$.

This seems weak.

However, **computers only know 0 and 1**.

A sequence of such bits can agree on **any computable value**.

Consensus on Synchronous Systems

On synchronous systems, **consensus is solvable**.

Without failures, it is **trivial**.

With failures it is harder, but not much.

The model is:

- N processes, all known to each other
- At most f failures
- Processes respond within a fixed period of time
- Messages arrive within a fixed period of time
- One response time + one message transmission time = one “round”

Synchronous Consensus without Failures

If **no processes fail** in a synchronous system:

- Consensus is guaranteed
- It requires **one round** of communication

The process:

1. Each process sends its proposed value to all other processes
2. Each process decides on the consensus:
 - 1 if all proposed values are 1
 - 0 if any proposed value is 0

Synchronous Consensus with f Failures

Assume that failures are **fail-stop**.

Each process has a **starting value** of either 0 or 1.

The algorithm will tolerate **at most** f failures.

We want to maintain three properties:

- **Agreement**: All non-faulty processes decide on the same output value (safety)
- **Validity**: If any process decides on a value, then some process started with that value
- **Termination**: All non-faulty processes decide on a value in finite time (liveness)

The Algorithm

Every process p maintains a vector V every process's proposed values.

Before round 1, V contains only p 's proposed value.

In each round:

1. Sends V to all other processes
2. Adds all new values received to V

After $f + 1$ rounds, p decides on the minimum value in V .

Counting Rounds

Why does this take $f + 1$ rounds?

This is [similar to reliable broadcast](#).

Consider:

- In Round 1, p_i sends its proposal to p_j , then crashes
- Only p_j knows p_i 's proposal
- In Round 2, p_j sends its vector V containing p_i 's proposal

Thus, in 2 rounds, p_i 's proposal is known by all correct processes (1 failure, 2 rounds).

What if p_j crashes after sending 1 message in round 2?

Example Agreement

Consider $n = 5, f = 2$.

If p_1 receives 5 values in round 1, can p_1 decide?

Example Agreement

Consider $n = 5, f = 2$.

If p_1 receives 5 values in round 1, can p_1 decide?

No, what if p_1 is the **only** process with info from p_2 ?

Example Agreement

Consider $n = 5, f = 2$.

If p_1 receives 5 values in round 1, can p_1 decide?

No, what if p_1 is the **only** process with info from p_2 ?

Can p_1 decide after round 2?

Example Agreement

Consider $n = 5, f = 2$.

If p_1 receives 5 values in round 1, can p_1 decide?

No, what if p_1 is the **only** process with info from p_2 ?

Can p_1 decide after round 2?

Yes, but can p_3 ?

Example Agreement

Consider $n = 5, f = 2$.

If p_1 receives 5 values in round 1, can p_1 decide?

No, what if p_1 is the **only** process with info from p_2 ?

Can p_1 decide after round 2?

Yes, but can p_3 ?

No, p_3 doesn't know if p_1 crashed!

Example Agreement

Consider $n = 5, f = 2$.

If p_1 receives 5 values in round 1, can p_1 decide?

No, what if p_1 is the **only** process with info from p_2 ?

Can p_1 decide after round 2?

Yes, but can p_3 ?

No, p_3 doesn't know if p_1 crashed!

After round 3, can p_3 decide?

Example Agreement

Consider $n = 5, f = 2$.

If p_1 receives 5 values in round 1, can p_1 decide?

No, what if p_1 is the **only** process with info from p_2 ?

Can p_1 decide after round 2?

Yes, but can p_3 ?

No, p_3 doesn't know if p_1 crashed!

After round 3, can p_3 decide?

Yes. If any process still doesn't have all of the information, **more than two processes** must have crashed!

Correctness

Why is this correct?

Processes are **synchronous**:

If a process has a message to send in a round, it will.

Messages are **synchronous**:

If a message is sent in a round, it is received in that round.

Why could $f + 1$ failures break this?

At least **one round** must include no failures!

Defining Impossibility

The Fischer-Lynch-Paterson (FLP) result [2] says that consensus is **impossible** in an asynchronous system.

Impossible in the **theoretical sense**, however!

Not **cannot ever be achieved**, but rather:

There exists **some circumstance** where it is not achieved.

In practice, consensus is **often achievable**.

A Weak Model

The FLP model of consensus is **deliberately weak**.

If weak consensus is impossible, then surely **stronger** is, also!

It assumes:

- Messages are always delivered **correctly** and **exactly once**
- Exactly **one process** fails
- Agreement is on exactly **one bit**
- The consensus result was proposed by **at least one** process
- At least **one process** arrives at correct consensus
- Consensus can take **arbitrarily long**

However, all processes and messages are **asynchronous**.

The Intuition

The intuition for FLP is essentially:

Suppose that p_i hears no messages from p_j .

Can p_i make a decision?

If it decides and p_j is failed: **no problem!**

If it decides and p_j has not failed: **big problem!**

What if every process that p_i heard proposed 1, and p_j proposed 0?

Therefore p_i must **wait for p_j** ... which might be failed!

Asynchrony

The FLP result essentially rests on the ambiguity of **asynchrony**.

In an asynchronous system, loss and failure **cannot be disambiguated**.

This means that any missing process **might just be slow**.

Therefore, an assumption of its failure could be **wrong**.

Using Consensus

We already said that we **use consensus!**

How, if it's **impossible?**

We either:

- **Narrow the window** of undecidability
- Change the rules (e.g., with partial synchrony)
- Tolerate occasional failures of consensus

Summary

- Deciding on **zero or one** is powerful
- Synchronous systems can decide with **an arbitrary, predefined** number of failures
- Asynchronous systems **cannot decide** ... maybe
 - **Failure is indistinguishable from delay**

Bibliography

Required Readings

- [1] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Chapter 14: 14.1–14.3, 14.5.1. Cambridge University Press, 2008.

Optional Readings

- [2] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “[Impossibility of Distributed Consensus with One Faulty Process](#)”. In: *Journal of the ACM* 32.2 (April 1985), Edited by S. L. Graham, pages 374–382.
- [3] Henry Robinson. [A Brief Tour of FLP Impossibility](#). Blog post on the Paper Trail blog. August 2008.

Discussion of FLP

The following discussion of FLP is from a previous offering.

I will not hold you responsible for it this semester.

It is left here in case you intend to do the optional readings.

Warning: Exploding Heads

The following discussion will probably make your head explode.

It might make my head explode.

You should:

1. Read FLP [2]. [Try to understand Lemma 2](#), but let it go when you can't.
2. Read the recommended reading [3].
3. Re-read FLP.

... then forgive me for what is about to happen.

Messages

FLP assumes that **all messages** are eventually delivered.

They may be delivered out of order.

It requires a model like reliable multicast:

If **any non-faulty process** receives a message, then **all non-faulty processes** receive the message.

It also requires that the following are **atomic**:

- Receipt of a message
- Processing in response to the message
- Transmission of responses to an arbitrary number of processes

Definitions

A **configuration** C is the state of all processes, plus all messages in the system.

A **step** moves from one configuration to another, and consists of one **atomic operation** (receive, process, send) in one process.

An **event** $e = \{m, p\}$ is the receipt of message m at process p , defining a step, and $e(C)$ is the configuration C after applying the event e .

A **schedule** is a finite sequence of events σ that can be applied to C , and $\sigma(C)$ is some configuration **reachable** from C .

Valence

A configuration C is **univalent** if every reachable configuration from C has the same decision.

It is **0-valent** if every decision is 0.

It is **1-valent** if every decision is 1.

A configuration C is **bivalent** if the reachable configurations from C contain both possible decisions.

Lemma 1

Lemma 1 says that **disjoint schedules** are commutative.

Given σ_1 and σ_2 , such that:

- $\sigma_1(C) = C_1$
- $\sigma_2(C) = C_2$

If the sets of processes in events in σ_1 and σ_2 are **disjoint**, then:

$$\sigma_1(C_2) = \sigma_2(C_1) = C_3$$

Lemma 2

Lemma 2 in the paper claims that **schedules matter**. [3]

It states that:

- Any starting state **is bivalent**
- The set of failures and messages in an execution from that state determines this

Thus, the initial configuration is **not enough** to determine valence.

Lemma 3

Lemma 3 claims that, starting from a bivalent configuration C [3]:

- any event e can be applied last
- There exists some sequence σ for which $e(\sigma(C))$ is bivalent

The proof for this is very confusing.

Lemma 3 Intuition

Ultimately Lemma 3 uses contradiction to show:

- There is some event $e' \neq e$ that determines whether $e(\sigma(C))$ is 0-valent or 1-valent
- If e is applied to a different process than e' , Lemma 1 says they can be applied in either order, so $e(\sigma(C))$ must be bivalent **without e'**
- If e is applied to the same process p as e' , then p can **do nothing indefinitely**; if a decision is made in this state, then p can apply either e or both e and e' , to achieve either 0-valence or 1-valence, so the decision **might be invalid**

Copyright

Copyright 2021, 2023–2026 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://cse.buffalo.edu/~eblanton/>.