Byzantine Generals Problem:
Solution using signed messages
The signed message model

A signed message satisfies all the conditions of oral message, plus two extra conditions:

- Signature cannot be forged. Forged messages are detected and discarded by loyal generals.
- Anyone can verify its authenticity of a signature.

Signed messages improve resilience.
Example

Using signed messages, byzantine consensus **is feasible** with 3 generals and 1 traitor. In (b) the loyal lieutenants compute the consensus value by applying some choice function on the set of values.
Signature list

0 → 1 (v{0})
1 → 7 (v{0,1})
7 → 4 (v{0,1,7})
4 → 2 (v{0,1,7,4})
Byzantine consensus:  
The signed message algorithms \( SM(m) \)

Commander \( i \) sends out a signed message \( v_i \) to each lieutenant \( j \neq i \)

Lieutenant \( j \), after receiving a message \( v_S \), append it to a set \( V_j \), only if

(i) it is not forged, and  
(ii) it has not been received before.

If the length of \( S \) is less than \( m+1 \), then lieutenant \( j \)

(i) appends his own signature to \( S \), and

(ii) sends out the signed message to every other lieutenant
whose signature does not appear in \( S \).

Lieutenant \( j \) applies a choice function on \( V_j \) to make the final decision.
Theorem of signed messages

If \( n \geq m + 2 \), where \( m \) is the maximum number of traitors, then \( \text{SM}(m) \) satisfies both IC1 and IC2.

Proof.
Case 1. Commander is loyal. The bag of each process will contain exactly one message, that was sent by the commander.

(Try to visualize this)
Proof of signed message theorem

Case 2. Commander is traitor.

• The signature list has a size \((m+1)\), and there are \(m\) traitors, so at least one lieutenant signing the message must be loyal.

• Every loyal lieutenant \(i\) will receive every other loyal lieutenant’s message. So, every message accepted by \(j\) is also accepted by \(i\) and vice versa. So \(V.i = V.j\).
With $m=2$ and a signature list of length 2, the loyal generals may not receive the same order from the commander who is a traitor. When the length of the signature list grows to 3, the problem is resolved.
Concluding remarks

• The signed message version tolerates a larger number \((n-2)\) of faults.

• Message complexity however is the same in both cases.

Message complexity = \((n-1)(n-2) \ldots (n-m+1)\)
Failure detectors
Failure detector for crash failures

- The design of fault-tolerant algorithms will be simple if processes can detect (crash) failures.

- In synchronous systems with bounded delay channels, crash failures can definitely be detected using timeouts.
Failure detectors for asynchronous systems

In asynchronous distributed systems, the detection of crash failures is imperfect. There will be false positives and false negatives. Two properties are relevant:

Completeness. Every crashed process is eventually suspected.

Accuracy. No correct process is ever suspected.
Failure Detectors

An FD is a distributed **oracle** that provides **hints** about the operational status of processes.

However:

- Hints may be **incorrect**
- FD may give **different** hints to different processes
- FD may **change its mind** (over & over) about the operational status of a process
Typical FD Behavior

FD at q: trust → suspect → trust → suspect

Process p: up → down

FD at q: suspect (permanently)
Revisit the Consensus problem

![Diagram of consensus problem]

- Input nodes: 1, 2, 3, 4
- Agreed value

- Output

---
Example

0 suspects \{1,2,3,7\} to have failed. Does this satisfy completeness? Does this satisfy accuracy?
Classification of completeness

• **Strong completeness.** Every crashed process is eventually suspected by *every correct process*, and remains a suspect thereafter.

• **Weak completeness.** Every crashed process is eventually suspected by *at least one* correct process, and remains a suspect thereafter.

*Note that we don’t care what mechanism is used for suspecting a process.*
Classification of accuracy

• **Strong accuracy.** No correct process is ever suspected.

• **Weak accuracy.** There is at least one correct process that is never suspected.
Transforming completeness

Weak completeness can be transformed into strong completeness

Program strong completeness (program for process i);
define D: set of process ids (representing the suspects);
initially D is generated by the weakly complete failure detector of i;

{program for process i}
do true →
  send D(i) to every process j ≠ i;
  receive D(j) from every process j ≠ i;
  D(i) := D(i) \ D(j);
  D(i) := D(i) \ j fi
od
Eventual accuracy

A failure detector is *eventually strongly accurate*, if there exists a time $T$ after which no correct process is suspected.

*(Before that time, a correct process be added to and removed from the list of suspects any number of times)*

A failure detector is *eventually weakly accurate*, if there exists a time $T$ after which at least one process is no more suspected.
Classifying failure detectors

**Perfect P.** (Strongly) Complete and strongly accurate

**Strong S.** (Strongly) Complete and weakly accurate

**Eventually perfect ◊P.**
(Strongly) Complete and eventually strongly strongly accurate

**Eventually strong ◊S**
(Strongly) Complete and eventually weakly accurate

Other classes are feasible: W (weak completeness) and weak accuracy) and ◊W
Motivation

The study of failure detectors was motivated by those who studied the consensus problem. Given a failure detector of a certain type, how can we solve the consensus problem?

**Question 1.** How can we implement these classes of failure detectors in asynchronous distributed systems?

**Question 2.** What is the weakest class of failure detectors that can solve the consensus problem? (*Weakest class of failure detectors is closer to reality*)
Application of Failure Detectors

Applications often need to determine which processes are up (operational) and which are down (crashed). This service is provided by Failure Detector. FDs are at the core of many fault-tolerant algorithms and applications, like

- Group Membership
- Group Communication
- Atomic Broadcast
- Primary/Backup systems
- Atomic Commitment
- Consensus
- Leader Election
- .....
\[
\begin{align*}
\text{SLOW} & \quad \text{s} \\
p & \quad q \\
t & \quad q, s \\
r & \quad q \\
\end{align*}
\]
Consensus

Crash!
Solving Consensus

- In synchronous systems: Possible

- In asynchronous systems: Impossible [FLP83]

  even if:
  - at most one process may crash, and
  - all links are reliable
A more complete classification of failure detectors

<table>
<thead>
<tr>
<th>strong completeness</th>
<th>strong accuracy</th>
<th>weak accuracy</th>
<th>◊ strong accuracy</th>
<th>◊ weak accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect P</td>
<td>Strong S</td>
<td>◊P</td>
<td>◊S</td>
<td></td>
</tr>
<tr>
<td>weak completeness</td>
<td>Weak W</td>
<td></td>
<td>◊W</td>
<td></td>
</tr>
</tbody>
</table>
**Motivation**

**Question 1.** Given a failure detector of a certain type, how can we solve the *consensus problem*?

**Question 2.** How can we implement these classes of failure detectors in asynchronous distributed systems?

**Question 3.** What is the *weakest class of failure detectors* that can solve the consensus problem?

*(Weakest class of failure detectors is closest to reality)*
Consensus using P

\{program for process \( p \), \( t = \text{max number of faulty processes} \}\}

Initially \( V_p := (\bot, \bot, \bot, \ldots, \bot) \); \{array of size \( n \}\}

\( V_p[p] = \text{input of } p \); \( D_p := V_p \); \( r_p := 1 \)

\{\( V_p[q] = \bot \) means, process \( p \) thinks \( q \) is a suspect. Initially everyone is a suspect\}\}

\{Phase 1\} for round \( r_p = 1 \) to \( t + 1 \)

send \((r_p, D_p, p)\) to all;

wait to receive \((r_p, D_q, q)\) from all \( q \), \{or else \( q \) becomes a suspect\};

for \( k = 1 \) to \( n \) \( V_p[k] = \bot \land \exists (r_p, D_q, q): D_q[k] \neq \bot \rightarrow V_p[k] := D_q[k] \)

end for

end for

\{at the end of Phase 1, \( V_p \) for each correct process is identical\}

\{Phase 2\} Final decision value is the input from the \textit{first} element \( V_p[j] \): \( V_p[j] \neq \bot \)
Understanding consensus using $P$

Why continue $(t+1)$ rounds?

It is possible that a process $p$ sends out the first message to $q$ and then crashes. If there are $n$ processes and $t$ of them crashed, then after at most $(t+1)$ asynchronous rounds, $V_p$ for each correct process $p$ becomes identical, and contains all inputs from processes that may have transmitted at least once.
Understanding consensus using P

Well, I received D from 1, but did everyone receive it? To ensure multiple rounds of broadcasts are necessary …

Completely connected topology

Well, I received D from 1, but did everyone receive it? To ensure multiple rounds of broadcasts are necessary …
Consensus using other type of failure detectors

Algorithms for reaching consensus with several other forms of failure detectors exist. In general, the weaker is the failure detector, the closer it is to reality (a truly asynchronous system), but the harder is the algorithm for implementing consensus.
Consensus using $S$

$V_p := (⊥, ⊥, ⊥, ..., ⊥)$; $V_p[p] := \text{input of } p$; $D_p := V_p$

(Phase 1) Same as phase 1 of consensus with $P$ – it runs for $(t+1)$ asynchronous rounds

(Phase 2) send $(V_p, p)$ to all;

receive $(D_q, q)$ from all $q$;

$$\text{for } k = 1 \text{ to } n \exists V_q[k]: V_p[p] \neq ⊥ \land V_q[k] = ⊥ \rightarrow V_p[k] := D_p[k] := ⊥ \text{ end for}$$

(Phase 3) Decide on the first element $V_p[j]$: $V_p[j] \neq ⊥$
Consensus using $S$: example

Assume that there are six processes: 0,1,2,3,4,5. Of these 4, 5 crashed. And 3 is the process that will never be suspected. Assuming that $k$ is the input from process $k$, at the end of phase 1, the following is possible:

- $V_0 = (0, \bot, 2, 3, \bot, \bot)$
- $V_1 = (\bot, 1, \bot, 3, \bot, \bot)$
- $V_2 = (0, 1, 2, 3, \bot, \bot)$
- $V_3 = (\bot, 1, \bot, 3, \bot, \bot)$

At the end of phase 3, the processes agree upon the input from process 3.
Conclusion

Asynchronous system

Consensus Problem

Cannot solve consensus

Can solve consensus

P

◊
P

S

◊
S

◊
W

◊
W