Basics of Consensus

Part I: Intro to Fault-Tolerant Distributed Algorithms

Annu Gmeiner  Igor Konnov  Ulrich Schmid
Helmut Veith  Josef Widder

VTSA 2021
Are they always working?
No... some failing systems

- **Therac-25 (1985)**
  - radiation therapy machine
  - gave massive overdoses, e.g., due to race conditions of concurrent tasks

- **Quantas Airbus in-flight Learmonth upset (2008)**
  - 1 out of 3 replicated components failed
  - computer initiated dangerous altitude drop

- **Ariane 501 maiden flight (1996)**
  - primary/backup, i.e., 2 replicated computers
  - both run into the same variable overflow

- **Netflix outages due to Amazon’s cloud (ongoing)**
  - one is not sure what is going on there
  - hundreds of computers involved
Why do they fail?

faults at design/implementation phase

approach:
find and fix faults before operation
⇒ model checking

faults at runtime
outside of control of designer/developer

⇒ e.g., to the right: crack in a diode in the data link interface of the Space Shuttle led to erroneous messages being sent

approach:
keep system operational despite faults
⇒ fault-tolerant distributed algorithms

Driscoll (Honeywell)
Josef Widder (Informal Systems)
Why do they fail?

- faults at design/implementation phase

- faults at runtime
  - outside of control of designer/developer
  - e.g., to the right: crack in a diode in the data link interface of the Space Shuttle
    ⇒ led to erroneous messages being sent

Driscoll (Honeywell)
Why do they fail?

- faults at design/implementation phase
  - approach: find and fix faults before operation
  - ⇒ model checking

- faults at runtime
  - outside of control of designer/developer
  - e.g., to the right: crack in a diode in the data link interface of the Space Shuttle
  - ⇒ led to erroneous messages being sent

Driscoll (Honeywell)
Why do they fail?

- faults at design/implementation phase
  - approach: find and fix faults before operation
  - ⇒ model checking

- faults at runtime
  - outside of control of designer/developer
  - e.g., to the right: crack in a diode in the data link interface of the Space Shuttle
  - ⇒ led to erroneous messages being sent

  - approach: keep system operational despite faults
  - ⇒ fault-tolerant distributed algorithms

Driscoll (Honeywell)
Bringing both together

Goal: automatically verified fault-tolerant distributed algorithms
Bringing both together

Goal: automatically verified fault-tolerant distributed algorithms

model checking FTDAs is a research challenge:
- computers run independently at different speeds
- exchange messages with uncertain delays
- faults
- parameterization

... fault-tolerance makes model checking harder
Lecture overview

Part I: Fault-tolerant distributed algorithms
- introduction to distributed algorithms
- details of a case study algorithm
- motivation why verification is cool (Motivating Cezara’s lecture)

Part II: General discussions + We look at Ben Or’s algorithm together

Part III: Modeling message passing + synchronous consensus
- Jennifer Welch’s slides

Part IV: General discussions

Part V: Consensus-related issues in the Cosmos Blockchain (Tendermint)
- ongoing work
Part I: Fault-Tolerant Distributed Algorithms
Distributed Systems are everywhere

What they allow to do

- share resources
- communicate
- increase performance
  - speed
  - fault tolerance

Difference to centralized systems

- independent activities (concurrency)
- components do not have access to the global state (only “local view”)
Application areas

buzzwords from the 60ies

- operating systems
- (distributed) data base systems
- communication networks
- multiprocessor architectures
- control systems

New buzzwords

- cloud computing
- social networks
- multi core
- cyber-physical systems
Major challenge

Uncertainty
- computers run independently at different speeds
- exchange messages with (unknown) delays
- faults
Major challenge

Uncertainty

- computers run independently at different speeds
- exchange messages with (unknown) delays
- faults

challenge in design of distributed algorithms

- a process has access only to its local state
- but one wants to achieve some global property
Major challenge

Uncertainty
- computers run independently at different speeds
- exchange messages with (unknown) delays
- faults

challenge in design of distributed algorithms
- a process has access only to its local state
- but one wants to achieve some global property

challenge in proving them correct
- large degree of non-determinism
  ⇒ large execution and state space
Process $P$ provides a service. We want to access it reliably but $P$ may fail.
Process $P$ provides a service. We want to access it reliably but $P$ may fail.

canonical approach: replication, i.e., several copies of $P$

Due to non-determinism, the behavior of the copies might deviate (e.g. in a replicated database, transactions are committed in different orders at different sites).
Process $P$ provides a service. We want to access it reliably but $P$ may fail.

canonical approach: replication, i.e., several copies of $P$.

Due to non-determinism, the behavior of the copies might deviate (e.g. in a replicated database, transactions are committed in different orders at different sites).

$\Rightarrow$ we have to enforce that the copies “behave as one”.

$\Rightarrow$ Consistency in a distributed system: what does it mean to behave as one.
Replication — distributed systems

\[ P \rightarrow \text{replication} \rightarrow \{P_1, P_2, P_3\} \]
Distributed message passing system

multiple distributed processes $p_i$

dots represent states

- a step of a process can be
  - a send step (a process sends messages to other processes)
  - a receive step (a process receives a subset of messages sent to it)
  - an internal step (a local computation)

- steps are the atomic (indivisible) units of computations
Types of Distributed Algorithms: Synchronous vs. Asynchronous

Synchronous
- all processes move in lock-step
- rounds
- a message sent in a round is received in the same round
- idealized view
- impossible or expensive to implement

Asynchronous
- only one process moves at a time
- arbitrary interleavings of steps
- a message sent is received eventually
Types of Distributed Algorithms:  
Synchronous vs. Asynchronous

Synchronous
- all processes move in lock-step
- rounds
- a message sent in a round is received in the same round  
- idealized view
- impossible or expensive to implement

Asynchronous
- only one process moves at a time
- arbitrary interleavings of steps
- a message sent is received eventually
- important problems not solvable (Fischer et al., 1985)!
Types of Distributed Algorithms: Synchronous vs. Asynchronous

Synchronous
- all processes move in lock-step
- rounds
- a message sent in a round is received in the same round
- idealized view
- impossible or expensive to implement

Asynchronous
- only one process moves at a time
- arbitrary interleavings of steps
- a message sent is received eventually
- important problems not solvable (Fischer et al., 1985)!

We focus on asynchronous algorithms today...
Asynchronous system

has very mild restrictions on the environment
  ■ interleaving semantics
  ■ unbounded message delays

very little can be done...

■ there is no distributed algorithm that solves consensus in the presence of one faulty process
  (as we will see, consensus is the paradigm of consistency)

■ folklore explanation:
  “you cannot distinguish a slow process from a crashed one”

■ real explanation:
  see intricate proof by Fischer, Lynch, and Paterson (JACM 1985)
Where we stand

replication

P

P₁  P₂  P₃

Fault-Tolerant Distributed Algos
What we still need...

Consistency requirements have been formalized under several names, e.g., consensus, atomic broadcast, Byzantine Generals problem, Byzantine agreement, atomic commitment. Definitions are similar but may have subtle differences. We use the famous Byzantine Generals to introduce this problem domain.
What we still need...

- consistency requirements have been formalized under several names, e.g.,
  - consensus
  - atomic broadcast
  - Byzantine Generals problem
  - Byzantine agreement
  - atomic commitment

- definitions are similar but may have subtle differences
consistency requirements have been formalized under several names, e.g.,
- consensus
- atomic broadcast
- Byzantine Generals problem
- Byzantine agreement
- atomic commitment

definitions are similar but may have subtle differences

We use the famous Byzantine Generals to introduce this problem domain...
Fault tolerance – The Byzantine generals problem

Wiktionary:
Byzantine: adj. of a devious, usually stealthy manner, of practice.
Fault tolerance – The Byzantine generals problem

Lamport, Shostak, and Pease wrote in their *Dijkstra Prize in Distributed Computing* winning paper (Lamport et al., 1982):

\[\ldots\] several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. \[\ldots\] However, some of the generals may be traitors \[\ldots\]

- if the divisions of loyal generals attack together, the city falls
- if only some loyal generals attack, their armies fall
- generals communicate by obedient messengers
Fault tolerance – The Byzantine generals problem

Lamport, Shostak, and Pease wrote in their *Dijkstra Prize in Distributed Computing* winning paper (Lamport et al., 1982):

> [...] several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. [...] However, some of the generals may be traitors [...] 

- if the divisions of loyal generals attack together, the city falls
- if only some loyal generals attack, their armies fall
- generals communicate by obedient messengers

**The Byzantine generals problem:**

- the loyal generals have to agree on whether to attack.
- if all want to attack they must attack, if no-one wants to attack they must not attack
Lamport, Shostak, and Pease wrote in their *Dijkstra Prize in Distributed Computing* winning paper (Lamport et al., 1982):

> [...] several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. [...] However, some of the generals may be traitors [...]  

- if the divisions of loyal generals attack together, the city falls  
- if only some loyal generals attack, their armies fall  
- generals communicate by obedient messengers

**The Byzantine generals problem:**

- the loyal generals have to agree on whether to attack.  
- if all want to attack they must attack, if no-one wants to attack they must not attack

metaphor for a distributed system where correct processes (loyal generals) act as one in the presence of faulty processes (traitors)
Byzantine generals problem cont.

In the absence of faults it is trivial to solve:
- send proposed plan ("attack" or "not attack") to all
- wait until received messages from everyone
- if a process proposed "attack" decide to attack
- otherwise, decide to not attack
Byzantine generals problem cont.

In the absence of faults it is trivial to solve:

- send proposed plan ("attack" or "not attack") to all
- wait until received messages from everyone
- if a process proposed "attack" decide to attack
- otherwise, decide to not attack

In the presence of faults it becomes tricky

- if a process may crash, some processes may not receive messages from everyone (but some may)
- if a process may send faulty messages, contradictory information may be received, e.g.,
  "A tells B that C told A that C wants to attack, while C tells B that C does not want to attack"
Byzantine generals problem cont.

In the absence of faults it is trivial to solve:
- send proposed plan ("attack" or "not attack") to all
- wait until received messages from everyone
- if a process proposed "attack" decide to attack
- otherwise, decide to not attack

In the presence of faults it becomes tricky
- if a process may crash, some processes may not receive messages from everyone (but some may)
- if a process may send faulty messages, contradictory information may be received, e.g.,
  "A tells B that C told A that C wants to attack, while C tells B that C does not want to attack"  Who is lying to whom?
Fault-tolerant distributed algorithms

- $n$ processes communicate by messages (reliable communication)
- all processes know that at most $t$ of them might be faulty
- $f$ are actually faulty
- resilience conditions, e.g., $n > 3t \land t \geq f \geq 0$
- no masquerading: the processes know the origin of incoming messages
Fault-tolerant distributed algorithms

- $n$ processes communicate by messages (reliable communication)
- all processes know that at most $t$ of them might be faulty
- $f$ are actually faulty
- resilience conditions, e.g., $n > 3t \land t \geq f \geq 0$
- no masquerading: the processes know the origin of incoming messages
Fault-tolerant distributed algorithms

- $n$ processes communicate by messages (reliable communication)
- all processes know that at most $t$ of them might be faulty
- $f$ are actually faulty
- resilience conditions, e.g., $n > 3t \land t \geq f \geq 0$
- no masquerading: the processes know the origin of incoming messages
Fault models—abstractions of reality

- **clean crashes:** least severe
  faulty processes prematurely halt after/before “send to all”

- **crash faults:**
  faulty processes prematurely halt (also) in the middle of “send to all”

- **omission faults:**
  faulty processes follow the algorithm, but some messages sent by them might be lost

- **symmetric faults:**
  faulty processes send arbitrarily to all or nobody

- **Byzantine faults:** most severe
  faulty processes can do anything
  encompass all behaviors of above models
Fault models—the ugly truth

A photo of a Byzantine fault:

photo by Driscoll (Honeywell)
he reports Byzantine behavior on the Space Shuttle computer network

other sources of faults: bit-flips in memory, power outage, disconnection from the network, etc.
Hence, we would like the weakest assumptions possible. But there are theoretical limits on how weak assumptions can be made:

- consensus is impossible in asynchronous systems if there may be a crash fault, i.e., $t = 1$ (Fischer et al., 1985)

- consensus is possible in synchronous systems in the presence of Byzantine faults iff $n > 3t$ (Lamport et al., 1982)

- consensus is impossible in (synchronous) round-based systems if $\lfloor n/2 \rfloor$ messages can be lost per round (Santoro & Widmayer, 1989)

- fast Byzantine consensus is solvable iff $n > 5t$ (Martin & Alvisi, 2006)

- 32 different “degrees of synchrony” and whether consensus can be solved in the presence of how many faults investigated in (Dolev et al., 1987)
Model vs. reality: impossibilities

Hence, we would like the weakest assumptions possible. But there are theoretical limits on how weak assumptions can be made:

- consensus is impossible in asynchronous systems if there may be a crash fault, i.e., $t = 1$ (Fischer et al., 1985)
- consensus is possible in synchronous systems in the presence of Byzantine faults iff $n > 3t$ (Lamport et al., 1982)
- consensus is impossible in (synchronous) round-based systems if $\lceil n/2 \rceil$ messages can be lost per round (Santoro & Widmayer, 1989)
- fast Byzantine consensus is solvable iff $n > 5t$ (Martin & Alvisi, 2006)
- 32 different “degrees of synchrony” and whether consensus can be solved in the presence of how many faults investigated in (Dolev et al., 1987)

arithmetic resilience conditions play crucial role!
After this excursion to faults, let’s go back to the problem of defining consistency (asynchronous systems).
Defining consistency — e.g., binary consensus

Every process has some initial value \( v \in \{0, 1\} \) and has to decide irrevocably on some value in concordance with the following properties:

1. **Agreement.** No two correct processes decide on different value.
2. **Validity.** If all correct processes have the same initial value \( v \), then \( v \) is the only possible decision value.
3. **Termination.** Every correct process eventually decides. At some point negotiations must be over.
Defining consistency — e.g., binary consensus

Every process has some initial value $v \in \{0, 1\}$ and has to decide irrevocably on some value in concordance with the following properties:

- **agreement.** No two correct processes decide on different value. Either all attack or no-one
Defining consistency — e.g., binary consensus

Every process has some initial value $v \in \{0, 1\}$ and has to decide irrevocably on some value in concordance with the following properties:

**agreement.** No two correct processes decide on different value.
- either all attack or no-one

**validity.** If all correct processes have the same initial value $v$, then $v$ is the only possible decision value
- the decision on whether to attack must be consistent with the will of at least one loyal general
Defining consistency — e.g., binary consensus

Every process has some initial value $v \in \{0, 1\}$ and has to decide irrevocably on some value in concordance with the following properties:

**agreement.** No two correct processes decide on different value. 
- either all attack or no-one

**validity.** If all correct processes have the same initial value $v$, then $v$ is the only possible decision value 
- the decision on whether to attack must be consistent with the will of at least one loyal general

**termination.** Every correct process eventually decides. 
- at some point negotiations must be over
Defining consistency — e.g., binary consensus

Every process has some initial value $v \in \{0, 1\}$ and has to decide irrevocably on some value in concordance with the following properties:

**agreement.** No two correct processes decide on different value.

**validity.** If all correct processes have the same initial value $v$, then $v$ is the only possible decision value

the decision on whether to attack must be consistent with

the will of at least one loyal general

**termination.** Every correct process eventually decides.

at some point negotiations must be over

Interplay of safety and liveness makes the problem hard...
What if only two properties have to be satisfied?

Every process has some initial value $v \in \{0, 1\}$ and has to decide **irrevocably** on some value in concordance with the following properties:

**validity.** If all correct processes have the same initial value $v$, then $v$ is the only possible decision value.

**termination.** Every correct process eventually decides.

Give an algorithm that solves validity and termination!
What if only two properties have to be satisfied?

Every process has some initial value $v \in \{0, 1\}$ and has to decide irrevocably on some value in concordance with the following properties:

**validity.** If all correct processes have the same initial value $v$, then $v$ is the only possible decision value.

**termination.** Every correct process eventually decides.

Solution: decide my own proposed value. (no need to agree)
What if only two properties have to be satisfied?

Every process has some initial value \( v \in \{0, 1\} \) and has to decide \textbf{irrevocably} on some value in concordance with the following properties:

\begin{itemize}
    \item \textbf{agreement.} No two correct processes decide on different value.
    \item \textbf{termination.} Every correct process eventually decides.
\end{itemize}

Give an algorithm that solves agreement and termination!
What if only two properties have to be satisfied?

Every process has some initial value \( v \in \{0, 1\} \) and has to decide \textbf{irrevocably} on some value in concordance with the following properties:

- \textbf{agreement}. No two correct processes decide on different value.

- \textbf{termination}. Every correct process eventually decides.

Solution: decide 0. (no relation to initial values required)
What if only two properties have to be satisfied?

Every process has some initial value $v \in \{0, 1\}$ and has to decide **irrevocably** on some value in concordance with the following properties:

**agreement.** No two correct processes decide on different value.

**validity.** If all correct processes have the same initial value $v$, then $v$ is the only possible decision value.

Give an algorithm that solves agreement and validity!
What if only two properties have to be satisfied?

Every process has some initial value \( v \in \{0, 1\} \) and has to decide **irrevocably** on some value in concordance with the following properties:

**agreement.** No two correct processes decide on different value.

**validity.** If all correct processes have the same initial value \( v \), then \( v \) is the only possible decision value.

Solution: do nothing (doing nothing is always safe)
Wrap-up: Intro to FTDAs

- distributed systems
- replication and consistency
- synchronous vs. asynchronous
- fault models
- example for an agreement problem: Byzantine Generals
Our case study...
Asynchronous Reliable Broadcast (Srikanth & Toueg, 87)

The core of the classic broadcast algorithm from the DA literature.

Variables of process $i$

1. $v_i$: $\{0, 1\}$ initially 0 or 1
2. $\text{accept}_i$: $\{0, 1\}$ initially 0

An atomic step:

3. if $v_i = 1$
   then send (echo) to all;

4. if received (echo) from at least $t + 1$ distinct processes
   and not sent (echo) before
   then send (echo) to all;

5. if received (echo) from at least $n - t$ distinct processes
   then $\text{accept}_i := 1$;
Assumptions from (Srikanth & Toueg, 87)

- asynchronous interleaving
- reliable message passing (no bounds on message delays)
- at most $t$ Byzantine faults
- resilience condition: $n > 3t \land t \geq f$
The spec of our case-study

**Unforgeability.** If \( v_i = \text{FALSE} \) for all correct processes \( i \), then for all correct processes \( j \), accept\(_j\) remains \( \text{FALSE} \) forever.

**Completeness.** If \( v_i = \text{TRUE} \) for all correct processes \( i \), then there is a correct process \( j \) that eventually sets accept\(_j\) to \( \text{TRUE} \).

**Relay.** If a correct process \( i \) sets accept\(_i\) to \( \text{TRUE} \), then eventually all correct processes \( j \) set accept\(_j\) to \( \text{TRUE} \).
The spec of our case-study

**Unforgeability.** If \( v_i = \text{FALSE} \) for all correct processes \( i \), then for all correct processes \( j \), \( \text{accept}_j \) remains \( \text{FALSE} \) forever.

If no loyal general wants to attack, then traitors should not be able to force one.

**Completeness.** If \( v_i = \text{TRUE} \) for all correct processes \( i \), then there is a correct process \( j \) that eventually sets \( \text{accept}_j \) to \( \text{TRUE} \).

If all loyal generals want to attack, there shall be an attack.

**Relay.** If a correct process \( i \) sets \( \text{accept}_i \) to \( \text{TRUE} \), then eventually all correct processes \( j \) set \( \text{accept}_j \) to \( \text{TRUE} \).

If one loyal general attacks, then all loyal generals should attack.
The spec of our case-study

**Unforgeability.** If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, accept$_j$ remains $\text{FALSE}$ forever.

if no loyal general wants to attack, then traitors should not be able to force one.

**Completeness.** If $v_i = \text{TRUE}$ for all correct processes $i$, then there is a correct process $j$ that eventually sets accept$_j$ to $\text{TRUE}$.

If all loyal generals want to attack, there shall be an attack.

**Relay.** If a correct process $i$ sets accept$_i$ to $\text{TRUE}$, then eventually all correct processes $j$ set accept$_j$ to $\text{TRUE}$.

If one loyal general attacks, then all loyal generals should attack.

These are the specs as given in literature: they can be formalized in LTL
Reliable broadcast vs. Consensus

**Reliable broadcast: Completeness.** If $v_i = \text{TRUE}$ for all correct processes $i$, then there is a correct process $j$ that eventually sets $\text{accept}_j$ to $\text{TRUE}$.

**Consensus: Termination.** Every correct process eventually decides.

Difference:
- Completeness requires to “do something” only if $\forall i. v_i = \text{TRUE}$, i.e., only for one specific initial state
- Termination requires to “do something” in all runs (from all initial states)
- weakening of spec makes reliable broadcast solvable in async, while consensus is not solvable
Asynchronous Reliable Broadcast (Srikanth & Toueg, 87)

The core of the classic broadcast algorithm from the DA literature.

Variables of process $i$

1. $v_i$: $\{0, 1\}$ initially 0 or 1
2. $accept_i$: $\{0, 1\}$ initially 0

An atomic step:

1. if $v_i = 1$
2. then send (echo) to all;

3. if received (echo) from at least $t + 1$ distinct processes
4. and not sent (echo) before
5. then send (echo) to all;

6. if received (echo) from at least $n - t$ distinct processes
7. then $accept_i := 1$;
Threshold-Guarded Distributed Algorithms

**Standard construct: quantified guards (t=f=0)**

- **Existential Guard**
  
  if received $m$ from *some* process then ...

- **Universal Guard**

  if received $m$ from *all* processes then ...
Threshold-Guarded Distributed Algorithms

**Standard construct: quantified guards (t=f=0)**

- **Existential Guard**
  
  if received $m$ from *some* process then ...

- **Universal Guard**
  
  if received $m$ from *all* processes then ...

*what if faults might occur?* 😈
Threshold-Guarded Distributed Algorithms

**Standard construct: quantified guards** \((t=f=0)\)

- **Existential Guard**
  
  if received \(m\) from *some* process then ...

- **Universal Guard**
  
  if received \(m\) from *all* processes then ...

what if faults might occur?

**Fault-Tolerant Algorithms:** \(n\) processes, at most \(t\) are Byzantine

- **Threshold Guard**
  
  if received \(m\) from \(n-t\) processes then ...

- (the processes cannot refer to \(f\)!)

Josef Widder (Informal Systems)
Basic mechanisms used by the algorithm: thresholds

Correct processes count distinct incoming messages
Basic mechanisms used by the algorithm: thresholds

Correct processes count distinct incoming messages
Basic mechanisms used by the algorithm: thresholds

Correct processes count **distinct** incoming messages

if received $m$ from $t+1$ processes then ...

(threshold)

at least one non-faulty sent the message
Classic correctness argument — hand-written proofs
Proof: Unforgeability

If \( v_i = \text{FALSE} \) for all correct processes \( i \), then for all correct processes \( j \), accept\(_j\) remains FALSE forever.

1. **Variables of process \( i \)**
   - \( v_i: \{0, 1\} \) *initially* 0 or 1
   - \( \text{accept}_i: \{0, 1\} \) *initially* 0

2. **An atomic step:**
   - if \( v_i = 1 \)
     - then send (echo) to all;
   - if received (echo) from at least \( t + 1 \) distinct processes
     - and not sent (echo) before
     - then send (echo) to all;
   - if received (echo) from at least \( n - t \) distinct processes
     - then \( \text{accept}_i := 1; \)
Proof: Unforgeability

If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, accept$_j$ remains FALSE forever.

Variables of process $i$

1. $v_i$: \{0, 1\} initially 0 or 1
2. accept$_i$: \{0, 1\} initially 0

An atomic step:

1. if $v_i = 1$
2. then send (echo) to all;

if received (echo) from at least $t + 1$ distinct processes
   and not sent (echo) before
then send (echo) to all;

if received (echo) from at least $n - t$ distinct processes
then accept$_i := 1$;

By contradiction assume a correct process sets accept$_j = 1$.
Proof: Unforgeability

If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, $\text{accept}_j$ remains $\text{FALSE}$ forever.

1. **Variables of process $i$**
   - $v_i$: $\{0, 1\}$ initially 0 or 1
   - $\text{accept}_i$: $\{0, 1\}$ initially 0

2. **An atomic step:**
   - if $v_i = 1$
     - then send $(\text{echo})$ to all;
   - if received $(\text{echo})$ from at least $t + 1$ distinct processes
     - and not sent $(\text{echo})$ before
     - then send $(\text{echo})$ to all;
   - if received $(\text{echo})$ from at least $n - t$ distinct processes
     - then $\text{accept}_i := 1$;

   - By contradiction assume a correct process sets $\text{accept}_j = 1$
   - Thus it has executed line 16
Proof: Unforgeability

If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, accept$_j$ remains FALSE forever.

1. Variables of process $i$
   - $v_i$: $\{0, 1\}$ initially $0$ or $1$
   - accept$_i$: $\{0, 1\}$ initially $0$

2. An atomic step:
   - if $v_i = 1$
     - then send (echo) to all;
   - if received (echo) from at least $t + 1$ distinct processes
     - and not sent (echo) before
     - then send (echo) to all;
   - if received (echo) from at least $n - t$ distinct processes
     - then accept$_i := 1$;

(by contradiction assume a correct process sets accept$_j = 1$

Thus it has executed line 16

Thus it has received $n - t$ messages by distinct processes
Proof: Unforgeability

If \( v_i = \text{FALSE} \) for all correct processes \( i \), then for all correct processes \( j \), \( \text{accept}_j \) remains \( \text{FALSE} \) forever.

Variables of process \( i \):
1. \( v_i: \{0, 1\} \) initially 0 or 1
2. \( \text{accept}_i: \{0, 1\} \) initially 0

An atomic step:
3. \( \text{if } v_i = 1 \)
4. \( \text{then } \text{send} \ (\text{echo}) \text{ to all}; \)
5. \( \text{if received} \ (\text{echo}) \text{ from at least} \)
6. \( t + 1 \text{ distinct processes} \)
7. \( \text{and not } \text{sent} \ (\text{echo}) \text{ before} \)
8. \( \text{then } \text{send} \ (\text{echo}) \text{ to all}; \)
9. \( \text{if received} \ (\text{echo}) \text{ from at least} \)
10. \( n - t \text{ distinct processes} \)
11. \( \text{then } \text{accept}_i := 1; \)

By contradiction assume a correct process sets \( \text{accept}_j = 1 \)
Thus it has executed line 16
Thus it has received \( n - t \) messages by distinct processes
That means messages by at \( n - 2t \) correct processes
Proof: Unforgeability

If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, accept$_j$ remains FALSE forever.

Variables of process $i$

1. $v_i$: \{0, 1\} \text{ initially } 0 \text{ or } 1
2. accept$_i$: \{0, 1\} \text{ initially } 0

An atomic step:

3. \textbf{if} $v_i = 1$
4. \textbf{then} send (echo) to all;

5. \textbf{if} received (echo) from at least $t + 1$ distinct processes
6. \textbf{and not} sent (echo) before
7. \textbf{then} send (echo) to all;

8. \textbf{if} received (echo) from at least $n - t$ distinct processes
9. \textbf{then} accept$_i := 1$;

By contradiction assume a correct process sets accept$_j = 1$

Thus it has executed line 16

Thus it has received $n - t$ messages by distinct processes

That means messages by at $n - 2t$ correct processes

Let $p$ be the first correct processes that has sent (echo)
Proof: Unforgeability

If \( v_i = \text{FALSE} \) for all correct processes \( i \), then for all correct processes \( j \), \( \text{accept}_j \) remains \( \text{FALSE} \) forever.

1. **Variables of process \( i \)**
   - \( v_i: \{0, 1\} \text{ initially } 0 \text{ or } 1 \)
   - \( \text{accept}_i: \{0, 1\} \text{ initially } 0 \)

2. **An atomic step:**
   - if \( v_i = 1 \)
     - then send (echo) to all;
   - if received (echo) from at least \( t + 1 \) distinct processes and not sent (echo) before
     - then send (echo) to all;
   - if received (echo) from at least \( n - t \) distinct processes
     - then \( \text{accept}_i := 1 \);

By contradiction assume a correct process sets \( \text{accept}_j = 1 \)
Thus it has executed line 16
Thus it has received \( n - t \) messages by distinct processes
That means messages by at \( n - 2t \) correct processes
Let \( p \) be the first correct processes that has sent (echo)
It did not send in line 7, as \( v_p = 0 \) by assumption

Josef Widder (Informal Systems) Fault-Tolerant Distributed Algos VTSA 2021 38 / 49
Proof: Unforgeability

If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, $\text{accept}_j$ remains $\text{FALSE}$ forever.

Variables of process $i$

- $v_i$: \{0, 1\} initially 0 or 1
- $\text{accept}_i$: \{0, 1\} initially 0

An atomic step:

- If $v_i = 1$
  - then send (echo) to all;
- If received (echo) from at least $t + 1$ distinct processes
  - and not sent (echo) before
  - then send (echo) to all;
- If received (echo) from at least $n - t$ distinct processes
  - then $\text{accept}_i := 1$;

By contradiction assume a correct process sets $\text{accept}_j = 1$
Thus it has executed line 16
Thus it has received $n - t$ messages by distinct processes
That means messages by at $n - 2t$ correct processes
Let $p$ be the first correct processes that has sent (echo)
It did not send in line 7, as $v_p = 0$ by assumption
Thus, $p$ sent in line 12
Proof: Unforgeability

If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, $\text{accept}_j$ remains $\text{FALSE}$ forever.

1. **Variables of process $i$**
   - $v_i$: \{0, 1\} initially 0 or 1
   - $\text{accept}_i$: \{0, 1\} initially 0

2. **An atomic step:**
   - if $v_i = 1$
     - then send (echo) to all;
   - if received (echo) from at least $t + 1$ distinct processes
     - and not sent (echo) before
     - then send (echo) to all;
   - if received (echo) from at least $n - t$ distinct processes
     - then $\text{accept}_i := 1$;

3. By contradiction assume a correct process sets $\text{accept}_j = 1$
4. Thus it has executed line 16
5. Thus it has received $n - t$ messages by distinct processes
6. That means messages by at $n - 2t$ correct processes
7. Let $p$ be the first correct processes that has sent (echo)
8. It did not send in line 7, as $v_p = 0$ by assumption
9. Thus, $p$ sent in line 12
10. Based on $t + 1$ messages, i.e., 1 sent by a correct processes
Proof: Unforgeability

If $v_i = \text{FALSE}$ for all correct processes $i$, then for all correct processes $j$, accept$_j$ remains $\text{FALSE}$ forever.

1. **Variables of process $i$**
   
   $v_i$: $\{0, 1\}$ *initially* 0 or 1
   
   accept$_i$: $\{0, 1\}$ *initially* 0

2. **An atomic step:**
   
   if $v_i = 1$
   then send (echo) to all;

3. if received (echo) from at least $t + 1$ distinct processes
   and not sent (echo) before
   then send (echo) to all;

4. if received (echo) from at least $n - t$ distinct processes
   then accept$_i$ := 1;

- By contradiction assume a correct process sets accept$_j = 1$
- Thus it has executed line 16
- Thus it has received $n - t$ messages by distinct processes
- That means messages by at $n - 2t$ correct processes
- Let $p$ be the first correct processes that has sent (echo)
- It did not send in line 7, as $v_p = 0$ by assumption
- Thus, $p$ sent in line 12
- Based on $t + 1$ messages, i.e., 1 sent by a correct processes
- contradiction to $p$ being the first one.
Proof: Completeness

If $v_i = \text{TRUE}$ for all correct processes $i$, then there is a correct process $j$ that eventually sets $\text{accept}_j$ to $\text{TRUE}$.

1. **Variables of process $i$**
   2. $v_i$: \{0, 1\} \text{ initially} 0 or 1
   3. $\text{accept}_i$: \{0, 1\} \text{ initially} 0

4. **An atomic step:**
   5. \textbf{if} $v_i = 1$
   6. \textbf{then} send \((\text{echo})\) to all;

7. \textbf{if} received \((\text{echo})\) from at least
   8. $t + 1$ distinct processes
   9. \textbf{and not} sent \((\text{echo})\) before
   10. \textbf{then} send \((\text{echo})\) to all;

11. \textbf{if} received \((\text{echo})\) from at least
   12. $n - t$ distinct processes
   13. \textbf{then} $\text{accept}_i := 1$;
Proof: Completeness

If $v_i = \text{TRUE}$ for all correct processes $i$, then there is a correct process $j$ that eventually sets $\text{accept}_j$ to $\text{TRUE}$.

1. Variables of process $i$
   
   $v_i: \{0, 1\}$ initially 0 or 1

   $\text{accept}_i: \{0, 1\}$ initially 0

2. An atomic step:

   if $v_i = 1$
   then send (echo) to all;

   if received (echo) from at least $t + 1$ distinct processes
   and not sent (echo) before
   then send (echo) to all;

   if received (echo) from at least $n - t$ distinct processes
   then $\text{accept}_i := 1$;
Proof: Completeness

If $v_i = \text{TRUE}$ for all correct processes $i$, then there is a correct process $j$ that eventually sets $\text{accept}_j$ to $\text{TRUE}$.

1  **Variables of process $i$**
2  $v_i$: $\{0, 1\}$ *initially* $0$ or $1$
3  $\text{accept}_i$: $\{0, 1\}$ *initially* $0$

4  **An atomic step:**
5  if $v_i = 1$
6  then send (echo) to all;
7
8  if received (echo) from at least
9     $t + 1$ distinct processes
10    and not sent (echo) before
11  then send (echo) to all;
12
13  if received (echo) from at least
14     $n - t$ distinct processes
15  then $\text{accept}_i := 1$;

- all, i.e., at least $n - t$ correct processes execute line 7
- by reliable communication all correct processes receive all messages sent by correct processes
Proof: Completeness

If $v_i = \text{TRUE}$ for all correct processes $i$, then there is a correct process $j$ that eventually sets $\text{accept}_j$ to $\text{TRUE}$.

1. Variables of process $i$
   \begin{align*}
   v_i : \{0, 1\} & \text{ initially 0 or 1} \\
   \text{accept}_i : \{0, 1\} & \text{ initially 0}
   \end{align*}

2. An atomic step:
   \begin{align*}
   \text{if } v_i = 1 \\
   \text{then send (echo) to all;}
   \end{align*}

3.\hspace{1em} if received (echo) from at least $t + 1$ distinct processes and not sent (echo) before \\
   \hspace{1em} then send (echo) to all;

4.\hspace{1em} if received (echo) from at least $n - t$ distinct processes \\
   \hspace{1em} then $\text{accept}_i := 1$;

   - all, i.e., at least $n - t$ correct processes execute line 7
   - by reliable communication all correct processes receive all messages sent by correct processes
   - Thus, a correct process receives $n - t$ (echo) messages
Proof: Completeness

If \( v_i = \text{TRUE} \) for all correct processes \( i \), then there is a correct process \( j \) that eventually sets \( \text{accept}_j \) to \( \text{TRUE} \).

Variables of process \( i \)

- \( v_i: \{0, 1\} \) initially 0 or 1
- \( \text{accept}_i: \{0, 1\} \) initially 0

An atomic step:

- if \( v_i = 1 \) then send (echo) to all;
- if received (echo) from at least \( t + 1 \) distinct processes and not sent (echo) before then send (echo) to all;
- if received (echo) from at least \( n - t \) distinct processes then \( \text{accept}_i := 1 \);

- all, i.e., at least \( n - t \) correct processes execute line 7
- by reliable communication all correct processes receive all messages sent by correct processes
- Thus, a correct process receives \( n - t \) (echo) messages
- Thus, a correct process executes line 16
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to TRUE, then eventually all correct processes $j$ set $\text{accept}_j$ to TRUE.

1. **Variables of process $i$**
   
   $v_i$: \{0, 1\} \textit{initially} 0 or 1

   $\text{accept}_i$: \{0, 1\} \textit{initially} 0

2. **An atomic step:**
   
   if $v_i = 1$
   
   then send (echo) to all;

3. if received (echo) from at least $t + 1$ distinct processes
   
   and not sent (echo) before
   
   then send (echo) to all;

4. if received (echo) from at least $n - t$ distinct processes
   
   then $\text{accept}_i := 1$;
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to $\text{TRUE}$, then eventually all correct processes $j$ set $\text{accept}_j$ to $\text{TRUE}$.

1. **Variables of process $i$**
   
   $v_i$: $\{0, 1\}$ *initially* 0 or 1
   
   $\text{accept}_i$: $\{0, 1\}$ *initially* 0

2. **An atomic step:**
   
   if $v_i = 1$
   
   then send \((echo)\) to all;

3. if received \((echo)\) from at least
   
   $t + 1$ distinct processes
   
   and not sent \((echo)\) before
   
   then send \((echo)\) to all;

4. if received \((echo)\) from at least
   
   $n - t$ distinct processes
   
   then $\text{accept}_i := 1$;

Correct process executes line 16

Thus it has received $n - t$ messages by distinct processes

That means messages by $n - 2t$ correct processes

By the resilience condition $n > 3t$, we have $n - 2t \geq t + 1$

Thus at least $t + 1$ correct processes have sent \((echo)\)

By reliable communication, these messages are received by all correct processes

Thus, all correct processes send \((echo)\) in line 12

There are at least $n - t$ correct processes

Thus, all correct processes eventually execute line 16
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to TRUE, then eventually all correct processes $j$ set $\text{accept}_j$ to TRUE.

1. **Variables of process $i$**
   
   $v_i$: \{0, 1\} \text{ initially } 0 \text{ or } 1
   
   $\text{accept}_i$: \{0, 1\} \text{ initially } 0

2. **An atomic step:**
   
   if $v_i = 1$
   
   then send (echo) to all;

   if received (echo) from at least $t + 1$ distinct processes
   
   and not sent (echo) before
   
   then send (echo) to all;

   if received (echo) from at least $n - t$ distinct processes
   
   then $\text{accept}_i := 1$;

   Correct process executes line 16
   
   Thus it has received $n - t$ messages by distinct processes

Thus, all correct processes eventually execute line 16.
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to TRUE, then eventually all correct processes $j$ set $\text{accept}_j$ to TRUE.

1. **Variables of process $i$**
   
   $v_i$: \{0, 1\} \text{ initially 0 or 1}

   $\text{accept}_i$: \{0, 1\} \text{ initially 0}

2. **An atomic step:**
   
   if $v_i = 1$
   
   then send (echo) to all;

3. if received (echo) from at least $t + 1$ distinct processes
   
   and not sent (echo) before
   
   then send (echo) to all;

4. if received (echo) from at least $n - t$ distinct processes
   
   then $\text{accept}_i := 1$;

- Correct process executes line 16
- Thus it has received $n - t$ messages by distinct processes
- That means messages by $n - 2t$ correct processes
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to TRUE, then eventually all correct processes $j$ set $\text{accept}_j$ to TRUE.

Variables of process $i$

$v_i: \{0, 1\}$ initially 0 or 1

$\text{accept}_i: \{0, 1\}$ initially 0

An atomic step:

if $v_i = 1$
then send (echo) to all;

if received (echo) from at least $t + 1$ distinct processes
and not sent (echo) before
then send (echo) to all;

if received (echo) from at least $n - t$ distinct processes
then $\text{accept}_i := 1$;

- Correct process executes line 16
- Thus it has received $n - t$ messages by distinct processes
- That means messages by $n - 2t$ correct processes
- By the resilience condition $n > 3t$, we have $n - 2t \geq t + 1$
Proof: Relay

If a correct process $i$ sets $accept_i$ to TRUE, then eventually all correct processes $j$ set $accept_j$ to TRUE.

```plaintext
Variables of process $i$
$v_i$: \{0, 1\} \(\text{initially} \ 0 \ \text{or} \ 1\)
$accept_i$: \{0, 1\} \(\text{initially} \ 0\)

An atomic step:
if $v_i = 1$
then send (echo) to all;

if received (echo) from at least $t + 1$ distinct processes
and not sent (echo) before
then send (echo) to all;

if received (echo) from at least $n - t$ distinct processes
then $accept_i := 1$;
```

- Correct process executes line 16
- Thus it has received $n - t$ messages by distinct processes
- That means messages by $n - 2t$ correct processes
- By the resilience condition $n > 3t$, we have $n - 2t \geq t + 1$
- Thus at least $t + 1$ correct processes have sent (echo)
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to TRUE, then eventually all correct processes $j$ set $\text{accept}_j$ to TRUE.

1. **Variables of process $i$**
   - $v_i$: \{0, 1\} initially 0 or 1
   - $\text{accept}_i$: \{0, 1\} initially 0

2. **An atomic step:**
   - if $v_i = 1$
     - then send (echo) to all;

   - if received (echo) from at least $t + 1$ distinct processes
     - and not sent (echo) before
     - then send (echo) to all;

   - if received (echo) from at least $n - t$ distinct processes
     - then $\text{accept}_i := 1$;

- Correct process executes line 16
- Thus it has received $n - t$ messages by distinct processes
- That means messages by $n - 2t$ correct processes
- By the resilience condition $n > 3t$, we have $n - 2t \geq t + 1$
- Thus at least $t + 1$ correct processes have sent (echo)
- By reliable communication, these messages are received by all correct processes
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to TRUE, then eventually all correct processes $j$ set $\text{accept}_j$ to TRUE.

1. **Variables of process $i$**
   
   $v_i$: \{0, 1\} initially 0 or 1

   $\text{accept}_i$: \{0, 1\} initially 0

2. **An atomic step:**

   - if $v_i = 1$
   - then send (echo) to all;

   - if received (echo) from at least $t + 1$ distinct processes
     and not sent (echo) before
   - then send (echo) to all;

   - if received (echo) from at least $n - t$ distinct processes
     then $\text{accept}_i := 1$;

   - Correct process executes line 16
   - Thus it has received $n - t$ messages by distinct processes
   - That means messages by $n - 2t$ correct processes
   - By the resilience condition $n > 3t$, we have $n - 2t \geq t + 1$
   - Thus at least $t + 1$ correct processes have sent (echo)
   - By reliable communication, these messages are received by all correct processes
   - Thus, all correct processes send (echo) in line 12
Proof: Relay

If a correct process \( i \) sets \( \text{accept}_i \) to \text{TRUE}, then eventually all correct processes \( j \) set \( \text{accept}_j \) to \text{TRUE}.

1. Variables of process \( i \)
   
   \( v_i: \{0, 1\} \text{ initially } 0 \text{ or } 1 \)

2. \( \text{accept}_i: \{0, 1\} \text{ initially } 0 \)

3. An atomic step:
   
   if \( v_i = 1 \)
   then send (echo) to all;

4. if received (echo) from at least \( t + 1 \) distinct processes
   and not sent (echo) before
   then send (echo) to all;

5. if received (echo) from at least \( n - t \) distinct processes
   then \( \text{accept}_i := 1 \);

6. Correct process executes line 16
7. Thus it has received \( n - t \) messages by distinct processes
8. That means messages by \( n - 2t \) correct processes
9. By the resilience condition \( n > 3t \), we have \( n - 2t \geq t + 1 \)
10. Thus at least \( t + 1 \) correct processes have sent (echo)
11. By reliable communication, these messages are received by all correct processes
12. Thus, all correct processes send (echo) in line 12
13. There are at least \( n - t \) correct
Proof: Relay

If a correct process $i$ sets $\text{accept}_i$ to TRUE, then eventually all correct processes $j$ set $\text{accept}_j$ to TRUE.

1. Variables of process $i$
   - $v_i$: $\{0, 1\}$ initially 0 or 1
   - $\text{accept}_i$: $\{0, 1\}$ initially 0

2. An atomic step:
   - if $v_i = 1$
     - then send (echo) to all;
   - if received (echo) from at least $t + 1$ distinct processes and not sent (echo) before
     - then send (echo) to all;
   - if received (echo) from at least $n - t$ distinct processes
     - then $\text{accept}_i := 1$;

3. Correct process executes line 16
4. Thus it has received $n - t$ messages by distinct processes
5. That means messages by $n - 2t$ correct processes
6. By the resilience condition $n > 3t$, we have $n - 2t \geq t + 1$
7. Thus at least $t + 1$ correct processes have sent (echo)
8. By reliable communication, these messages are received by all correct processes
9. Thus, all correct processes send (echo) in line 12
10. There are at least $n - t$ correct
11. Thus, all correct processes eventually execute line 16
Problems with hand-written proofs

- code inspection becomes confusing for larger algorithms
Bracha & Toueg’s algorithm (JACM 1985)

msg\_count: array of [types: 0..1] of integer
msg: record of type: (initial, echo, ready)
value: integer

while (there is no i such that
    msg\_count(initial, i) \geq 1 or
    msg\_count(echo, i) > (n + k)/2 or
    msg\_count(ready, i) \geq k + 1)
    receive(msg)
    if it is the first message received from the sender
    with these values of msg.type, msg.from
    then msg\_count(msg.type, msg.value) = msg\_count(msg.type, msg.value) + 1
end
for all q, send(\text{echo}, i)

while (there is no i such that
    msg\_count(echo, i) \geq (n + k)/2 or
    msg\_count(ready, i) \geq k + 1)
    receive(msg)
    if it is the first message received from the sender
    with these values of msg.type, msg.from
    then msg\_count(msg.type, msg.value) = msg\_count(msg.type, msg.value) + 1
end
for all q, send(\text{ready}, i)

while (there is no i such that
    msg\_count(ready, i) \geq 2k + 1)
    receive(msg)
    if it is the first message received from the sender
    with these values of msg.type, msg.from
    then msg\_count(msg.type, msg.value) = msg\_count(msg.type, msg.value) + 1
end
decide i

Fig. 3. An asynchronous Byzantine Agreement protocol.
Problems with hand-written proofs

- code inspection becomes confusing for larger algorithms
- hidden assumptions
  - resilience condition
  - reliable communication (fairness)
  - non-masquerading
  - failure model

If I cannot prove it correct, that does not mean the algorithm is wrong. How to come up with counterexamples?

Ultimate goal: verify the actual source code.

Josef Widder (Informal Systems)
Fault-Tolerant Distributed Algos
Problems with hand-written proofs

- code inspection becomes confusing for larger algorithms
- hidden assumptions
  - resilience condition
  - reliable communication (fairness)
  - non-masquerading
  - failure model
- re-using proofs if one of the ingredients changes?
- if I cannot prove it correct, that does not mean the algorithm is wrong
  \(\ldots\) how to come up with counterexamples?
- ultimate goal: verify the actual source code.
  \(\ldots\) it is not realistic that developers do mathematical proofs.
We have convinced a human, . . .

. . . why should we convince a computer?

- it is easy to make mistakes in proofs

- it is easier to overlook mistakes in proofs

- distributed algorithms require "non-centralized thinking" (untypical for the human mind)

- many issues to consider at the same time (interleaving of steps, faults, timing assumptions)

- people who tried to convince computers found bugs in published . . .

- Byzantine agreement algorithm (Lincoln & Rushby, 1993)

- clock synchronization algorithm (Malekpour & Siminiceanu, 2006)
We have convinced a human, . . .

. . . why should we convince a computer?

- it is easy to make mistakes in proofs
- it is easier to overlook mistakes in proofs
  - distributed algorithms require “non-centralized thinking”
    (untypical for the human mind)
  - many issues to consider at the same time
    (interleaving of steps, faults, timing assumptions)
We have convinced a human, . . .

. . . why should we convince a computer?

- it is easy to make mistakes in proofs
- it is easier to overlook mistakes in proofs
  - distributed algorithms require “non-centralized thinking”
    (untypical for the human mind)
  - many issues to consider at the same time
    (interleaving of steps, faults, timing assumptions)
- people who tried to convince computers found bugs in published . . .
  - Byzantine agreement algorithm (Lincoln & Rushby, 1993)
  - clock synchronization algorithm (Malekpour & Siminiceanu, 2006)
End of Part I
On the minimal synchronism needed for distributed consensus.
*J. ACM*, 34, 77–97.
http://doi.acm.org/10.1145/7531.7533.

Impossibility of Distributed Consensus with one Faulty Process.
*J. ACM*, 32(2), 374–382.
http://doi.acm.org/10.1145/3149.214121.

The part-time parliament.
http://doi.acm.org/10.1145/279227.279229.

The Byzantine Generals Problem.
A formally verified algorithm for interactive consistency under a hybrid fault model.
http://dx.doi.org/10.1109/FTCS.1993.627343.

Malekpour, Mahyar R., & Siminiceanu, Radu. 2006.
Comments on the Byzantine Self-Stabilizing Pulse Synchronization Protocol: Counterexamples.
Tech. rept. TM-2006-213951. NASA.

Fast Byzantine Consensus.

Evaluating the Condition-Based Approach to Solve Consensus.
Pages 541–550 of: DSN.

Pages 209–214 of: HASE.
Time is Not a Healer.
*Pages 304–313 of: STACS.*
LNCS, vol. 349.
http://dx.doi.org/10.1007/BFb0028994.

Srikanth, T. K., & Toueg, Sam. 1987.
Optimal clock synchronization.
*J. ACM, 34,* 626–645.
http://doi.acm.org/10.1145/28869.28876.
Model vs. reality: assumption coverage

Every assumption has a probability that it is satisfied in the actual system:

- $n > 3t$
  less likely than $n > t$

- every message sent is received within bounded time
  less likely than that it is eventually received

- processes fail by crashing
  less likely than they deviate arbitrarily from the prescribed behavior

- non-masquerading
  less likely than processes that can pretend to be someone else
Model vs. reality: assumption coverage

Every assumption has a probability that it is satisfied in the actual system:

- $n > 3t$
  - less likely than $n > t$
- every message sent is received within bounded time
  - less likely than that it is eventually received
- processes fail by crashing
  - less likely than they deviate arbitrarily from the prescribed behavior
- non-masquerading
  - less likely than processes that can pretend to be someone else

To use a distributed algorithm in practice:

- one must ensure that an assumption is suitable for a given system
- the probability that the system is working correctly is the probability that the assumptions hold
  (given that the distributed algorithm actually is correct)