

# Model Checking of Fault-Tolerant Distributed Algorithms

## Part I: Fault-Tolerant Distributed Algorithms

Annu Gmeiner   Igor Konnov   Ulrich Schmid  
Helmut Veith   Josef Widder

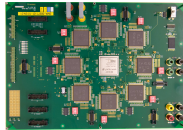


**for(sy)te,**  
Formal Methods  
in Systems Engineering

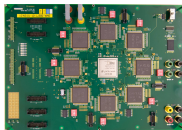
**RiSE**  
Rigorous Systems Engineering

VTSA 2014, Luxembourg

# Distributed Systems



# Distributed Systems



Are they always working?

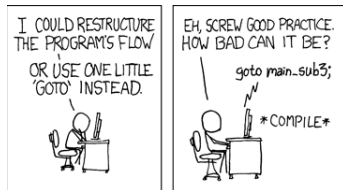
# No. . . some failing systems

- 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 integer 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?

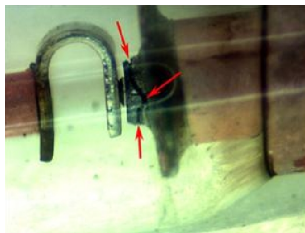
# Why do they fail?

- 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)

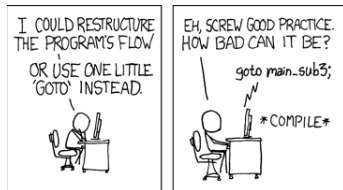
# Why do they fail?

## ■ faults at design/implementation phase

### approach:

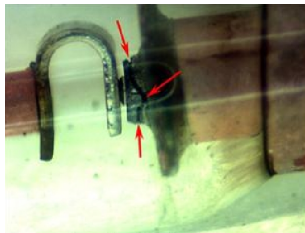
find and fix faults before operation

⇒ 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)

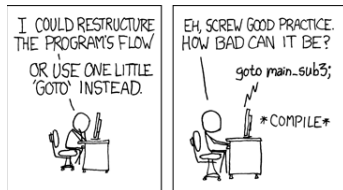
# Why do they fail?

## ■ faults at design/implementation phase

### approach:

find and fix faults before operation

⇒ model checking



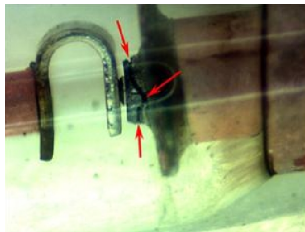
## ■ faults at runtime

- 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

## Part I: Fault-tolerant distributed algorithms

- introduction to distributed algorithms
- details of our case study algorithm
- motivation why model checking is cool

## Part II: Modeling fault-tolerant distributed algorithms

- model checking challenges in distributed algorithms
- Promela, control flow automata, etc.
- model checking of small instances with Spin

# Lecture overview cont.

## Part III: Introduction into Parameterized model checking

- the general problem statement
- well-known undecidability results

## Part IV: Parameterized model checking of FTDA's by abstraction

- parametric interval abstraction (PIA)
- PIA data and counter abstraction
- counterexample-guided abstraction refinement (CEGAR)

Igor Konnov:

## Part V: Parameterized model checking of FTDA's by BMC

- bounding the diameter
- bounded model checking as a complete method

# 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

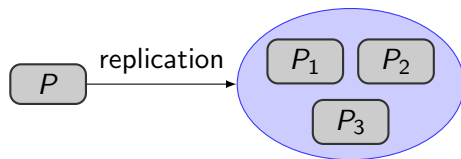
- large degree of non-determinism  
⇒ large execution and state space

# From dependability to a distributed system



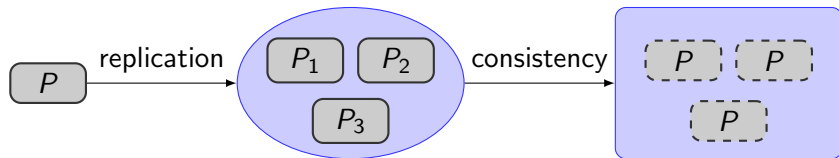
- Process  $P$  provides a service. We want to access it reliably  
but  $P$  may fail

# From dependability to a distributed system



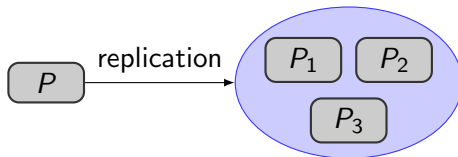
- Process  $P$  provides a service. We want to access it reliably but  $P$  may fail
- canonical approach: replication, i.e., several copies of  $P$   
Due to non-determinism, the behavior of the copies might deviate (e.g. in a replicated database, transactions are committed in different orders at different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

# From dependability to a distributed system



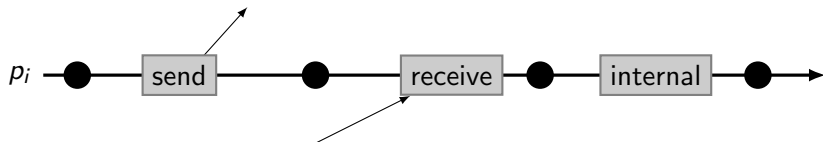
- Process  $P$  provides a service. We want to access it reliably but  $P$  may fail
- canonical approach: replication, i.e., several copies of  $P$   
Due to non-determinism, the behavior of the copies might deviate (e.g. in a replicated database, transactions are committed in different orders at different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



# 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 here...

# Asynchronous system

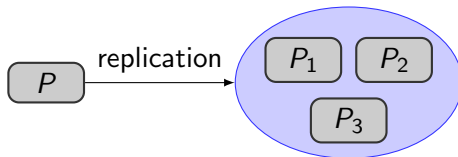
has very mild restrictions on the environment

- interleaving semantics
- unbounded message delays

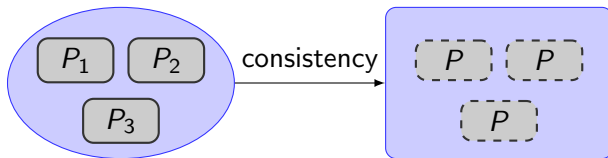
very little can be done. . .

- 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)

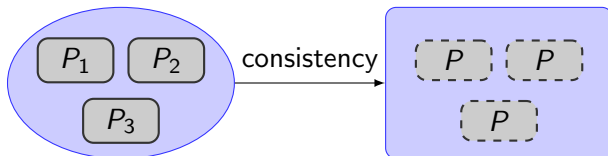
# Where we stand



## What we still need...

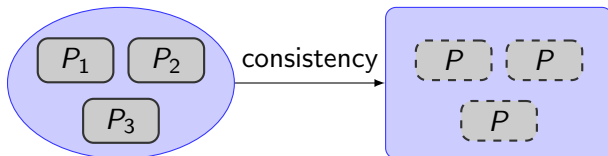


# What we still need...



- 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

# 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...

# Fault tolerance – The Byzantine generals problem

Wiktionary:

Byzantine: adj. of a devious, usually stealthy manner, of practice.



# Fault tolerance – The Byzantine generals problem

Lamport (this year's Turing laureate), 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

# Fault tolerance – The Byzantine generals problem

Lamport (this year's Turing laureate), 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

# Fault tolerance – The Byzantine generals problem

Lamport (this year's Turing laureate), 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.

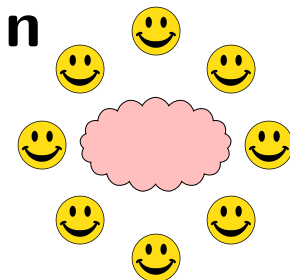
In the absence of faults it is trivial to solve:

- 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

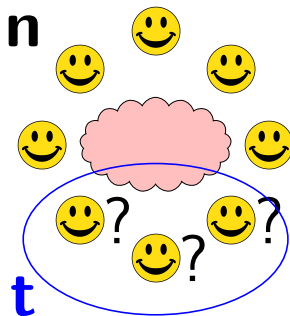
- if a process may crash, some processes may not receive messages from everyone (but some may)
- if a process may send faulty messages, contradictory information may be received, e.g.,  
“A tells B that C told A that C wants to attack, while C tells B that C does not want to attack”  
Who is lying to 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.,  $n > 3t \wedge t \geq f \geq 0$
- no masquerading: the processes know the origin of incoming messages

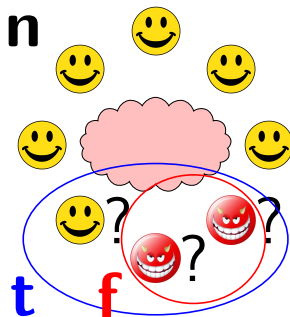
# Fault-tolerant distributed algorithms



- $n$  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.,  $n > 3t \wedge t \geq f \geq 0$
- 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.,  $n > 3t \wedge t \geq f \geq 0$
- 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:

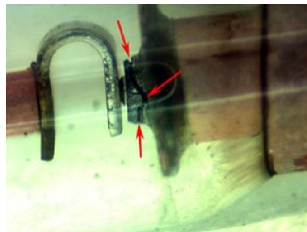


photo by Driscoll (Honeywell)

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.

## 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  $\lfloor n/2 \rfloor$  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  $\lfloor n/2 \rfloor$  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)

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:

## 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.  
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

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 **irrevocably** on some value in concordance with the following properties:

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

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

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 **irrevocably** on some value in concordance with the following properties:

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

**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 FTDAs

In this lecture we consider methods for asynchronous FTDAs that either

- solve problems that are less hard than consensus:

**reliable broadcast.** termination required only for specific initial state (Srikanth & Toueg, 1987). [Verified in Parts II, III, V]

**condition-based consensus** properties required only in runs from specific initial states (Mostéfaoui *et al.*, 2003)

[Verified in Part II]

**The Paxos idea** fault-tolerant distributed algorithms that are **safe** and make **progress** only if you are “lucky” (Lamport, 1998)

[Serious challenge]

- are asynchronous but use “information on faults” as a black box

**failure detector based atomic commitment.** distributed databases (Raynal, 1997)

[Challenge]

# Asynchronous FTDAs

In this lecture we consider methods for asynchronous FTDAs that either

- solve problems that are less hard than consensus:

**reliable broadcast.** termination required only for specific initial state (Srikanth & Toueg, 1987). [Verified in Parts II, III, V]

**condition-based consensus** properties required only in runs from specific initial states (Mostéfaoui *et al.*, 2003)

[Verified in Part II]

**The Paxos idea** fault-tolerant distributed algorithms that are **safe** and make **progress** only if you are “lucky” (Lamport, 1998)

[Serious challenge]

- are asynchronous but use “information on faults” as a black box

**failure detector based atomic commitment.** distributed databases (Raynal, 1997)

[Challenge]

We use the algorithm from (Srikanth & Toueg, 1987) as running example

# Asynchronous Reliable Broadcast (Srikanth & Toueg, 87)

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

```
1  Variables of process i
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $accept_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10    $t + 1$  distinct processes
11   and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15    $n - t$  distinct processes
16 then  $accept_i := 1$ ;
```

# Assumptions from (Srikanth & Toueg, 87)

- asynchronous interleaving
- reliable message passing (no bounds on message delays)
- at most  $t$  Byzantine faults
- resilience condition:  $n > 3t \wedge t \geq f$

# The spec of our case-study

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

**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 **TRUE**.

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

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

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

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

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

```
1  Variables of process i
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $accept_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10    $t + 1$  distinct processes
11   and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15    $n - t$  distinct processes
16 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 ...

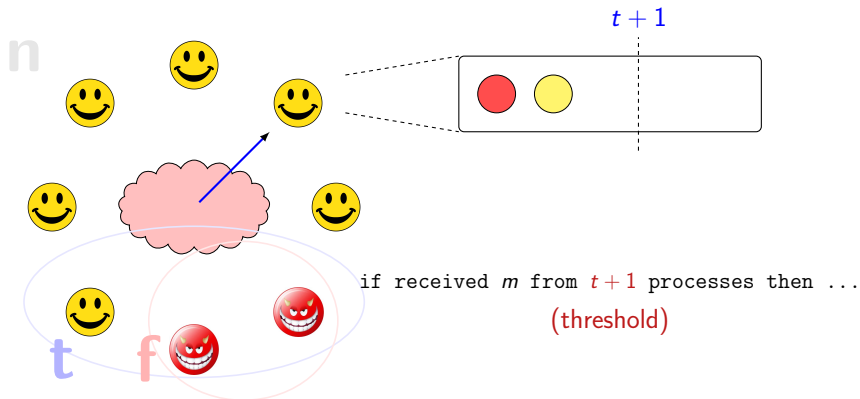
*what if faults might occur?*



## 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!*)

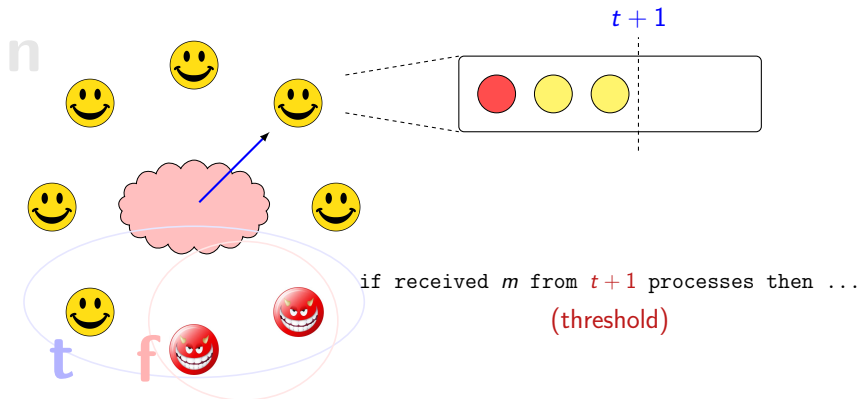
# Basic mechanisms used by the algorithm: thresholds



Correct processes count **distinct** incoming messages

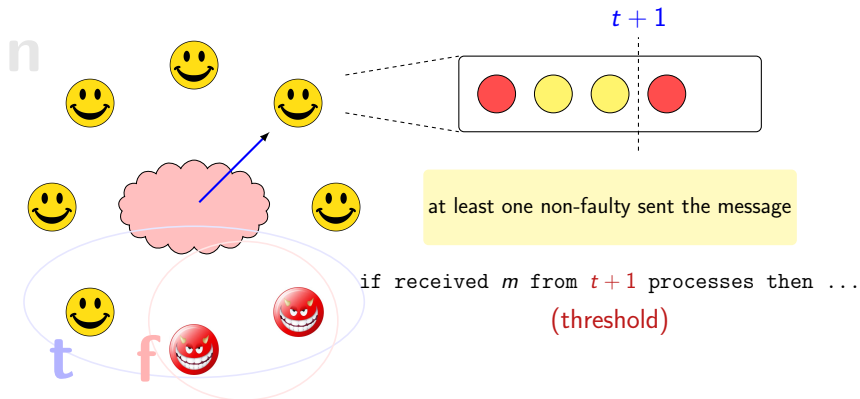


# Basic mechanisms used by the algorithm: thresholds



Correct processes count **distinct** incoming messages

# Basic mechanisms used by the algorithm: thresholds



Correct processes count **distinct** incoming messages

# Classic correctness argument — hand-written proofs

## Proof: Unforgeability

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

```
1  Variables of process  $i$ 
2   $v_i$ : {0, 1} initially 0 or 1
3   $\text{accept}_i$ : {0, 1} initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10      $t + 1$  distinct processes
11     and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15      $n - t$  distinct processes
16  then  $\text{accept}_i := 1$ ;
```

# Proof: Unforgeability

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

```
1  Variables of process  $i$ 
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10      $t + 1$  distinct processes
11     and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15      $n - t$  distinct processes
16  then  $\text{accept}_i := 1$ ;
```

- By contradiction assume a correct process sets  $\text{accept}_j = 1$

# Proof: Unforgeability

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

```
1  Variables of process  $i$ 
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10      $t + 1$  distinct processes
11     and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15      $n - t$  distinct processes
16  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  $j$ ,  $\text{accept}_j$  remains FALSE forever.

```
1  Variables of process i
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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

# Proof: Unforgeability

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

```
1  Variables of process  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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 least  $n - 2t$  correct processes



# Proof: Unforgeability

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

```
1  Variables of process  $i$ 
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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)

# Proof: Unforgeability

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

```
1  Variables of process  $i$ 
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10      $t + 1$  distinct processes
11     and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15      $n - t$  distinct processes
16  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

# Proof: Unforgeability

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

```
1  Variables of process  $i$ 
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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 FALSE forever.

```
1  Variables of process  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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
- 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  $j$ ,  $\text{accept}_j$  remains FALSE forever.

```
1  Variables of process  $i$ 
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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
- 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 TRUE.

```
1  Variables of process  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10      $t + 1$  distinct processes
11     and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15      $n - t$  distinct processes
16  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 TRUE.

1 *Variables of process  $i$*

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

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

4

5 *An atomic step:*

6 **if**  $v_i = 1$

7 **then** send (echo) to all;

8

9 **if** received (echo) from at least

10  $t + 1$  distinct processes

11 **and not** sent (echo) before

12 **then** send (echo) to all;

13

14 **if** received (echo) from at least

15  $n - t$  distinct processes

16 **then**  $\text{accept}_i := 1$ ;

- all, i.e., at least  $n - t$  correct processes execute line 7

# 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 TRUE.

1 *Variables of process  $i$*

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

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

4

5 *An atomic step:*

6 **if**  $v_i = 1$

7 **then** **send** (echo) **to all**;

8

9 **if** **received** (echo) from at least

10  $t + 1$  **distinct** processes

11 **and not** **sent** (echo) **before**

12 **then** **send** (echo) **to all**;

13

14 **if** **received** (echo) from at least

15  $n - t$  **distinct** processes

16 **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 TRUE.

1 *Variables of process  $i$*

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

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

4

5 *An atomic step:*

6 **if**  $v_i = 1$

7 **then** **send** (echo) **to all**;

8

9 **if** **received** (echo) from at least

10  $t + 1$  **distinct** processes

11 **and not** sent (echo) before

12 **then** **send** (echo) **to all**;

13

14 **if** **received** (echo) from at least

15  $n - t$  **distinct** processes

16 **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 TRUE.

```
1  Variables of process  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 then  $\text{accept}_i := 1$ ;
```

## 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  $i$*

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

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

4

5 *An atomic step:*

6 **if**  $v_i = 1$

7 **then** send (echo) to all;

8

9 **if** received (echo) from at least

10  $t + 1$  distinct processes

11 **and not** sent (echo) before

12 **then** send (echo) to all;

13

14 **if** received (echo) from at least

15  $n - t$  distinct processes

16 **then**  $\text{accept}_i := 1$ ;

■ 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
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 then  $\text{accept}_i := 1$ ;
```

- Correct process executes line 16
- Thus it has received  $n - t$  messages by distinct 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  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10      $t + 1$  distinct processes
11     and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15      $n - t$  distinct processes
16  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.

```
1  Variables of process  $i$ 
2   $v_i: \{0, 1\}$  initially 0 or 1
3   $\text{accept}_i: \{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10    $t + 1$  distinct processes
11   and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15    $n - t$  distinct processes
16  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  $\text{accept}_i$  to TRUE, then eventually all correct processes  $j$  set  $\text{accept}_j$  to TRUE.

```
1  Variables of process  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10    $t + 1$  distinct processes
11   and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15    $n - t$  distinct processes
16  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)



## 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  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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 TRUE, then eventually all correct processes  $j$  set  $\text{accept}_j$  to TRUE.

```
1  Variables of process  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10    $t + 1$  distinct processes
11   and not sent (echo) before
12  then send (echo) to all;
13
14  if received (echo) from at least
15    $n - t$  distinct processes
16  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

# 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  $i$ 
2   $v_i$ :  $\{0, 1\}$  initially 0 or 1
3   $\text{accept}_i$ :  $\{0, 1\}$  initially 0
4
5  An atomic step:
6  if  $v_i = 1$ 
7  then send (echo) to all;
8
9  if received (echo) from at least
10  $t + 1$  distinct processes
11 and not sent (echo) before
12 then send (echo) to all;
13
14 if received (echo) from at least
15  $n - t$  distinct processes
16 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
- 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(echo, i)  
while(there is no i such that  
    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(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
```

Part II, Part V

FIG. 3. An asynchronous Byzantine Agreement protocol.

# Condition-based consensus (Mostéfaoui *et al.*, 2003)

## Function *Consensus*( $v_i$ )

```
(1) foreach  $j \in [1..n]$  do  $V_i[j] \leftarrow \perp$  enddo; % Initialization%  
    %-----Phase 1-----  
(2) UR_Broadcast PHASE1( $v_i, i$ );  
(3) wait until (PHASE1( $-, -$ ) messages have been delivered from at least  $(n - f)$  processes);  
(4) foreach  $j \in [1..n]$  do if (PHASE1( $v_j, j$ ) has been delivered) then  $V_i[j] \leftarrow v_j$  endif enddo;  
(5)  $w_i \leftarrow S(V_i)$ ; % Estimate of the decision %  
    %-----Phase 2-----  
(6) UR_Broadcast PHASE2( $v_i, w_i, i$ );  
(7) repeat wait until (a new PHASE2( $v_j, w_j, j$ ) message has been delivered);  
(8)     if ( $V_i[j] = \perp$ ) then  $V_i[j] \leftarrow v_j$  endif;  
(9)     if (PHASE2( $-, w, -$ ) msgs with same  $w$  delivered from a majority of proc.)  
(10)      then return( $w$ ) endif  
(11) until (a PHASE2( $-, -, -$ ) message has been delivered from each process) endrepeat;  
(12) return (a deterministically chosen value of  $V_i$ )
```

Figure 1. A Condition-Based Message Passing Consensus Protocol ( $f < n/2$ )

## Part II, Part V

# Problems with hand-written proofs

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



# 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  
... how to come up with counterexamples?
- ultimate goal: verify the actual source code.  
... 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

# 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

# References I

- Dolev, Danny, Dwork, Cynthia, & Stockmeyer, Larry. 1987.  
On the minimal synchronism needed for distributed consensus.  
*J. ACM*, **34**, 77–97.  
<http://doi.acm.org/10.1145/7531.7533>.
- Fischer, Michael J., Lynch, Nancy A., & Paterson, M. S. 1985.  
Impossibility of Distributed Consensus with one Faulty Process.  
*J. ACM*, **32**(2), 374–382.  
<http://doi.acm.org/10.1145/3149.214121>.
- Lamport, Leslie. 1998.  
The part-time parliament.  
*ACM Trans. Comput. Syst.*, **16**, 133–169.  
<http://doi.acm.org/10.1145/279227.279229>.
- Lamport, Leslie, Shostak, Robert E., & Pease, Marshall C. 1982.  
The Byzantine Generals Problem.  
*ACM Trans. Program. Lang. Syst.*, **4**(3), 382–401.

# References II

Lincoln, P., & Rushby, J. 1993.

A formally verified algorithm for interactive consistency under a hybrid fault model.

*Pages 402–411 of: FTCS-23.*

<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.

Martin, Jean-Philippe, & Alvisi, Lorenzo. 2006.

Fast Byzantine Consensus.

*IEEE Trans. Dependable Sec. Comput.*, 3(3), 202–215.

Mostéfaoui, Achour, Mourgaya, Eric, Parvédy, Philippe Raipin, & Raynal, Michel. 2003.

Evaluating the Condition-Based Approach to Solve Consensus.

*Pages 541–550 of: DSN.*

Raynal, Michel. 1997.

A Case Study of Agreement Problems in Distributed Systems: Non-Blocking Atomic Commitment.

*Pages 209–214 of: HASE.*

# References III

Santoro, Nicola, & Widmayer, Peter. 1989.

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)