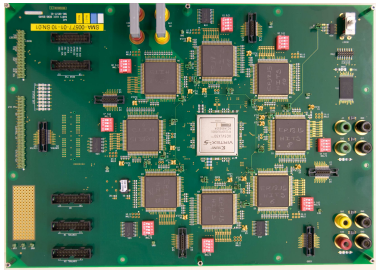
Model Checking of Fault-Tolerant Distributed Algorithms

Part I: Fault-tolerant Distributed Algorithms

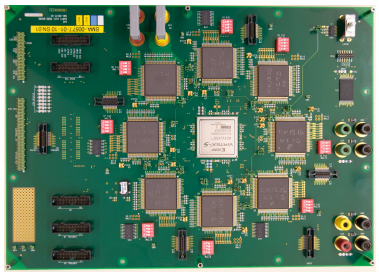
Annu Gmeiner Igor Konnov Ulrich Schmid Helmut Veith
Josef Widder

SFM-14:ESM. Bertinoro, Italy, EU

Distributed Systems



Distributed Systems



Are they always working?

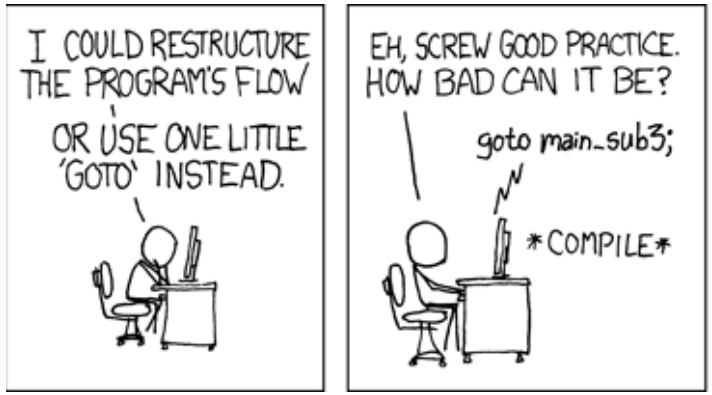
No. . . some failing systems

- Therac-25 (1985)
  - radiation therapy machine
  - gave massive overdoses, e.g., due to race conditions
- Quantas Airbus in-flight Learmonth upset (2008)
  - 1 out of 3 replicated components failed
  - computer initiated dangerous altitude drop
- Ariane 501 maiden flight (1996)
- Netflix outages due to Amazon's cloud (ongoing)

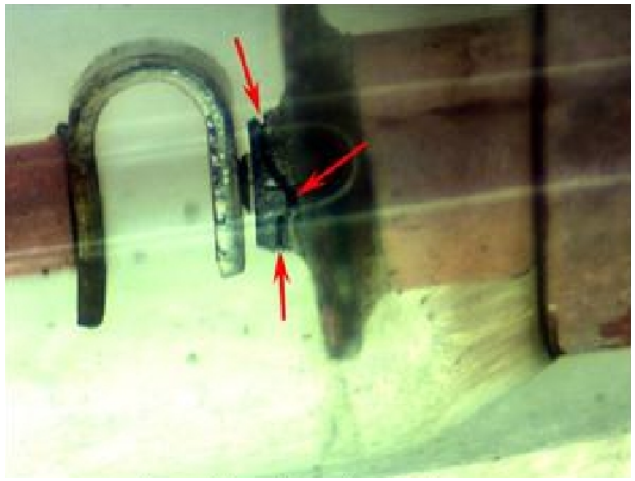
Why do they fail?

Why do they fail?

- faults at design/implementation phase



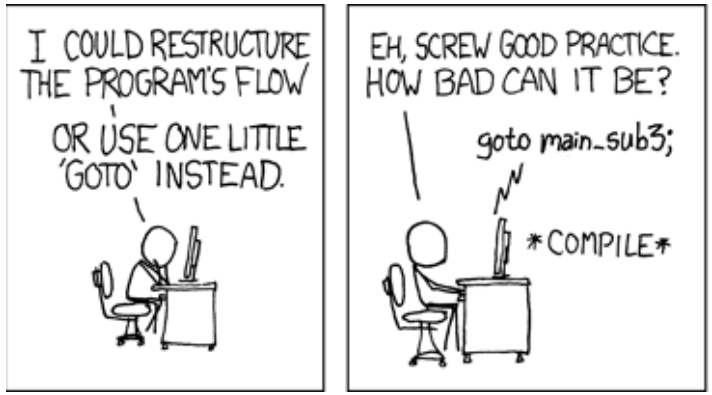
- faults at runtime
  - outside of control of designer/developer
  - power outage, hardware faults



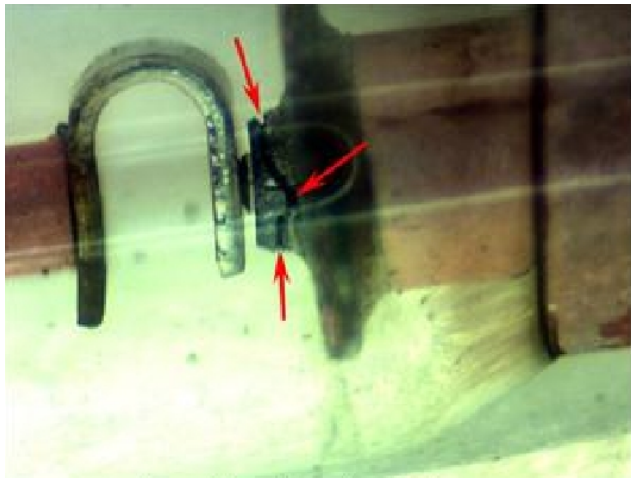
Driscoll (Honeywell)

Why do they fail?

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



- faults at runtime
  - outside of control of designer/developer
  - power outage, hardware faults



Driscoll (Honeywell)

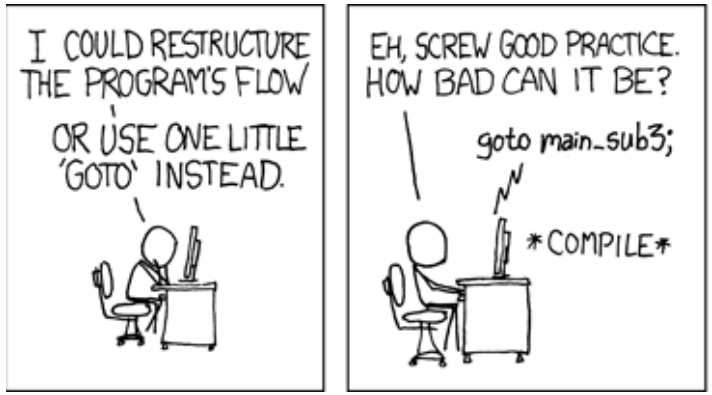
Why do they fail?

- faults at design/implementation phase

  approach:

  find and fix faults before operation

  $\Rightarrow$ model checking

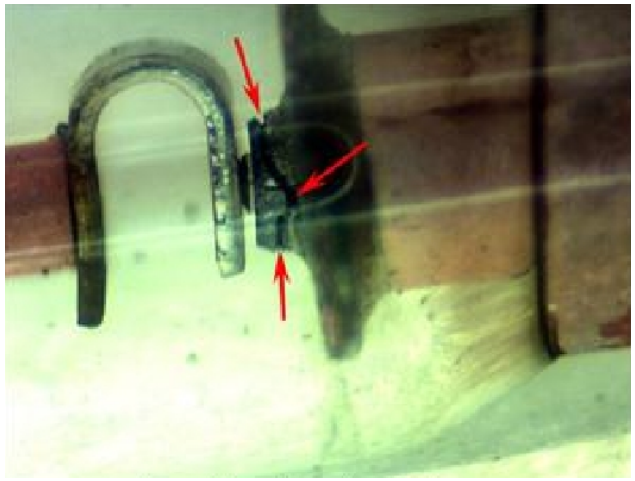


- faults at runtime
  - outside of control of designer/developer
  - power outage, hardware faults

  approach:

  keep system operational despite faults

  $\Rightarrow$ 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 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

Part III: Parameterized model checking

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

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)
- inherent uncertainty

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

- no global view available in parts of the system

Major challenge

Uncertainty

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

challenge in design of distributed algorithms

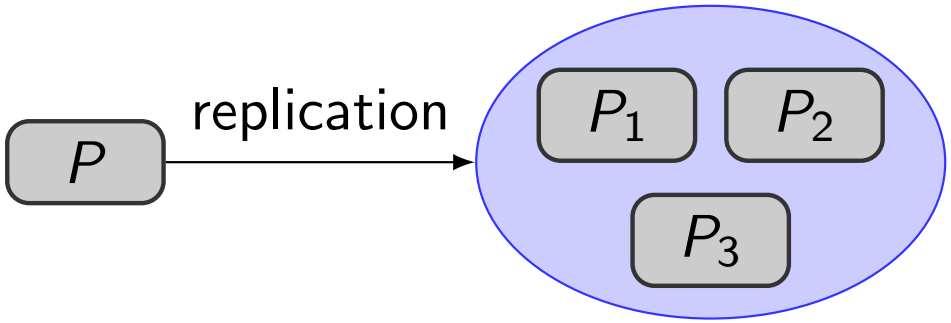
- no global view available in parts of the system

challenge in proving them correct

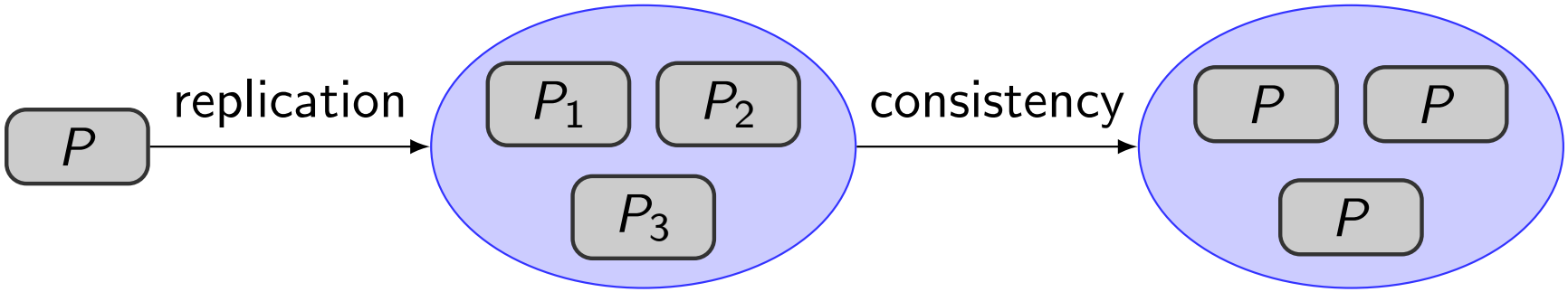
- large degree of non-determinism
  $\Rightarrow$ large execution and state space

From dependability to a distributed system — Consistency

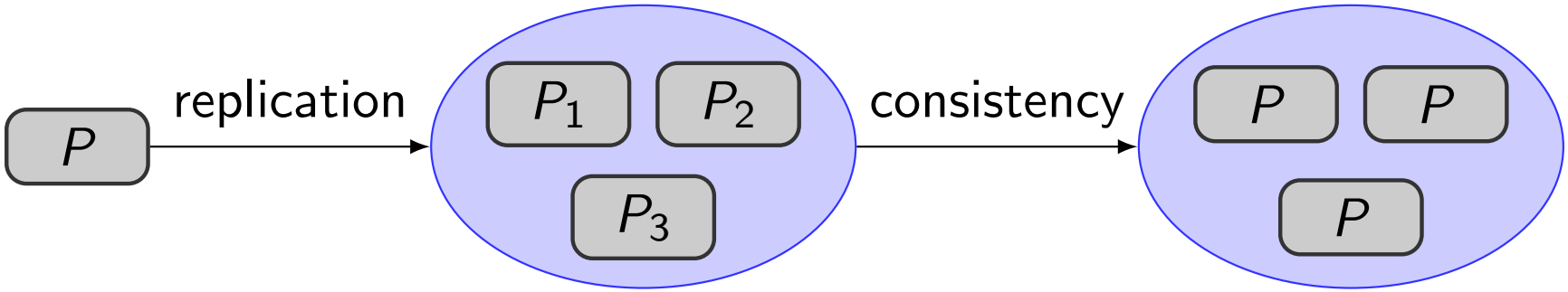
From dependability to a distributed system — Consistency



From dependability to a distributed system — Consistency



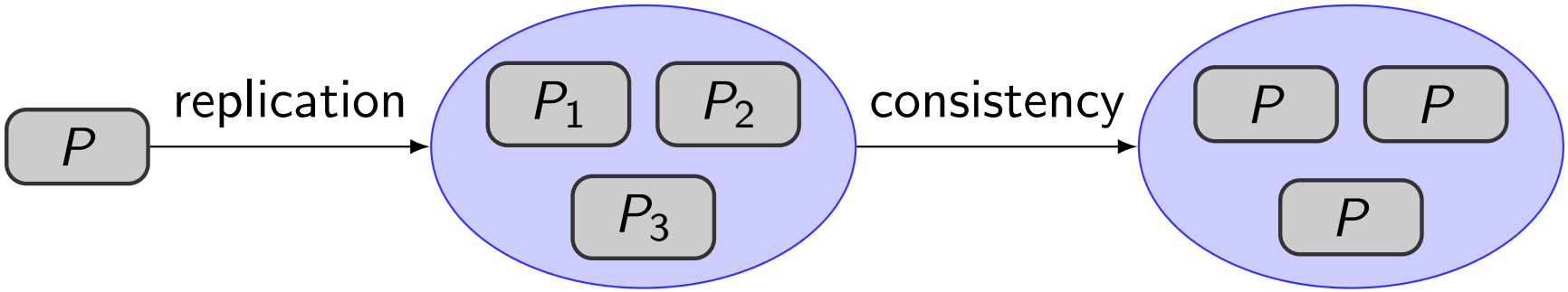
From dependability to a distributed system — Consistency



We thus discuss

- consistency in distributed systems in the presence of failures
  - how to define
  - how to achieve

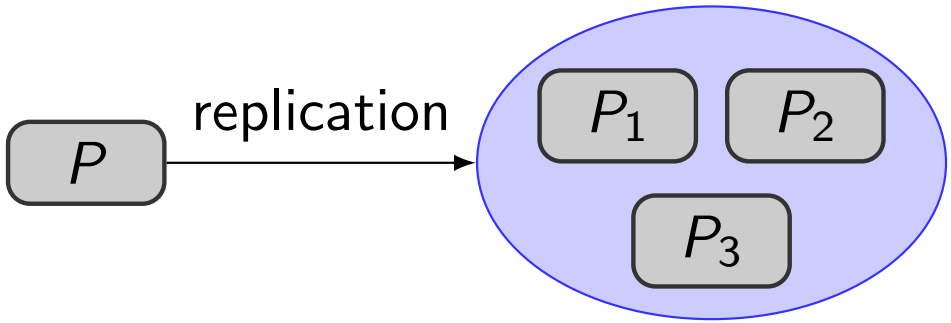
From dependability to a distributed system — Consistency



We thus discuss

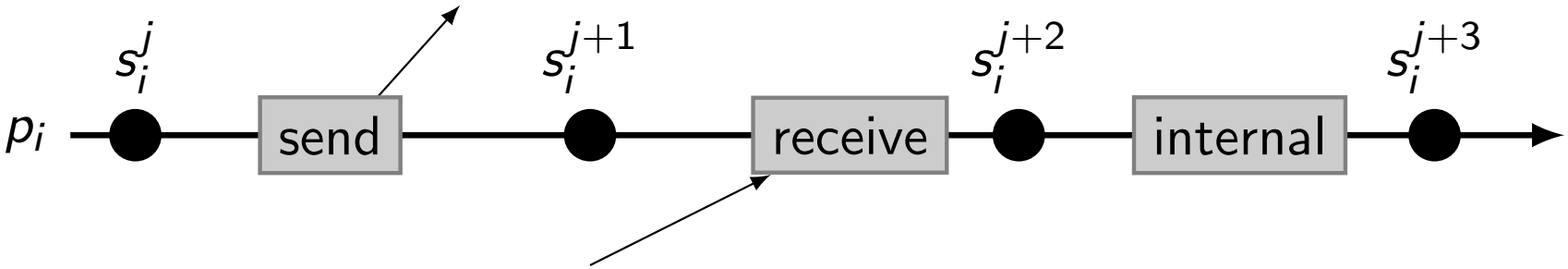
- consistency in distributed systems in the presence of failures
  - how to define
  - how to achieve
- what are the problems posed by distribution
  - asynchrony
  - failures
  - how to they play against us

Replication — distributed systems



Distributed message passing system

multiple distributed processes p_i



Distributed systems

non-empty set $P = \{p_1, \cdots p_n\}$ of n processes

each process p_i has

Distributed systems

non-empty set $P = \{p_1, \cdots p_n\}$ of n processes

each process p_i has

- a set of states S_i (subset initial states)
  - variables

Distributed systems

non-empty set $P = \{p_1, \cdots p_n\}$ of n processes

each process p_i has

- a set of states S_i (subset initial states)
  - variables
- a set of actions A_i
  - sending of a unique message
  - receive of a message
  - internal

Distributed systems

non-empty set $P = \{p_1, \cdots p_n\}$ of n processes

each process p_i has

- a set of states S_i (subset initial states)
  - variables
- a set of actions A_i
  - sending of a unique message
  - receive of a message
  - internal
- transition relation $S_i \times A_i$: which actions can execute from a given state

Distributed systems

non-empty set $P = \{p_1, \cdots p_n\}$ of n processes

each process p_i has

- a set of states S_i (subset initial states)
  - variables
- a set of actions A_i
  - sending of a unique message
  - receive of a message
  - internal
- transition relation $S_i \times A_i$: which actions can execute from a given state
- transition function $T_i : S_i \times A_i \rightarrow S_i$

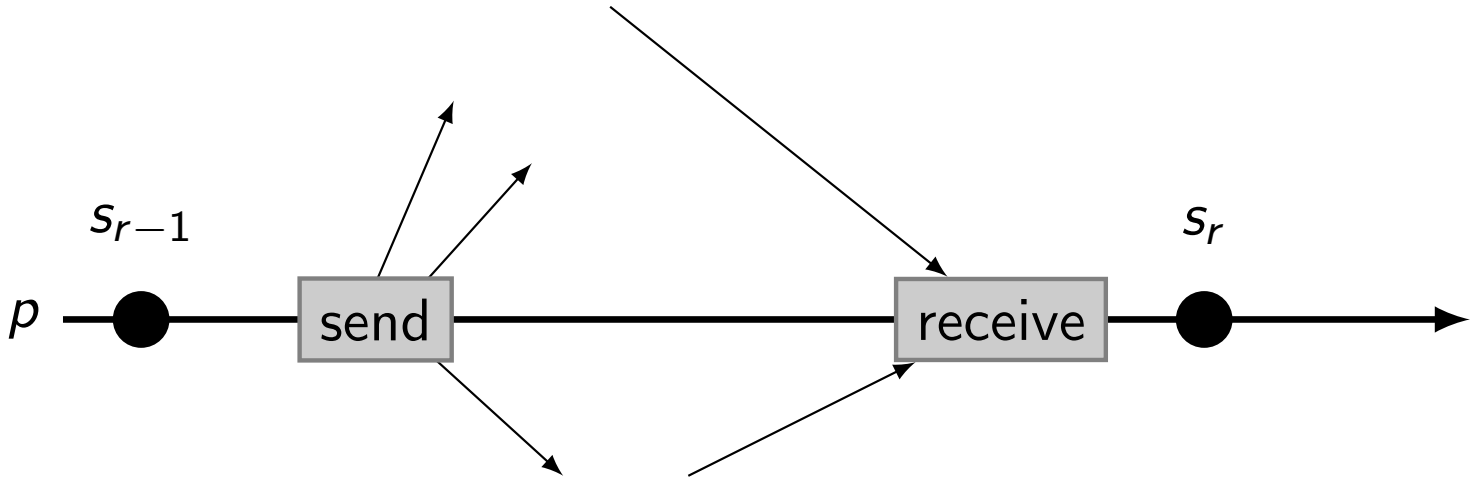
Types of Distributed Algorithms sync. vs. async

Synchronous

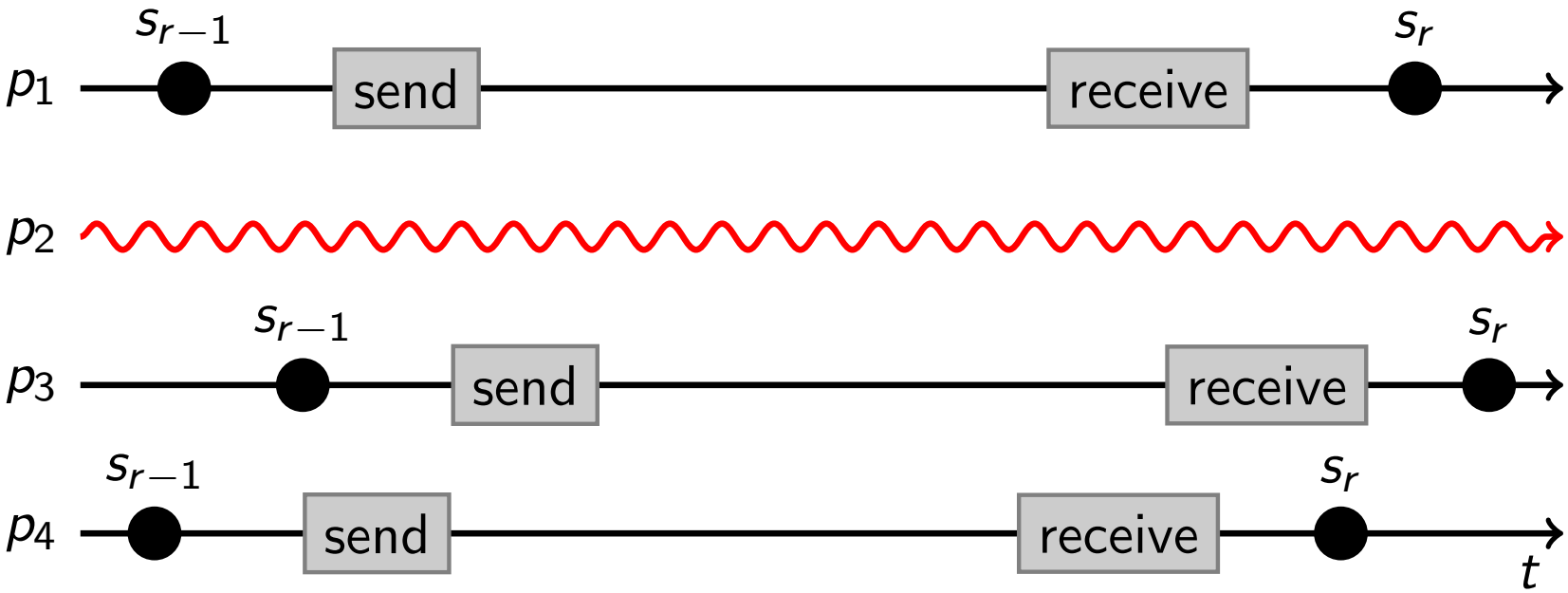
- all processes move in lock-step
- rounds
- a message sent in a round is received in the round
- idealized view
- impossible or expensive to implement

Synchronous system

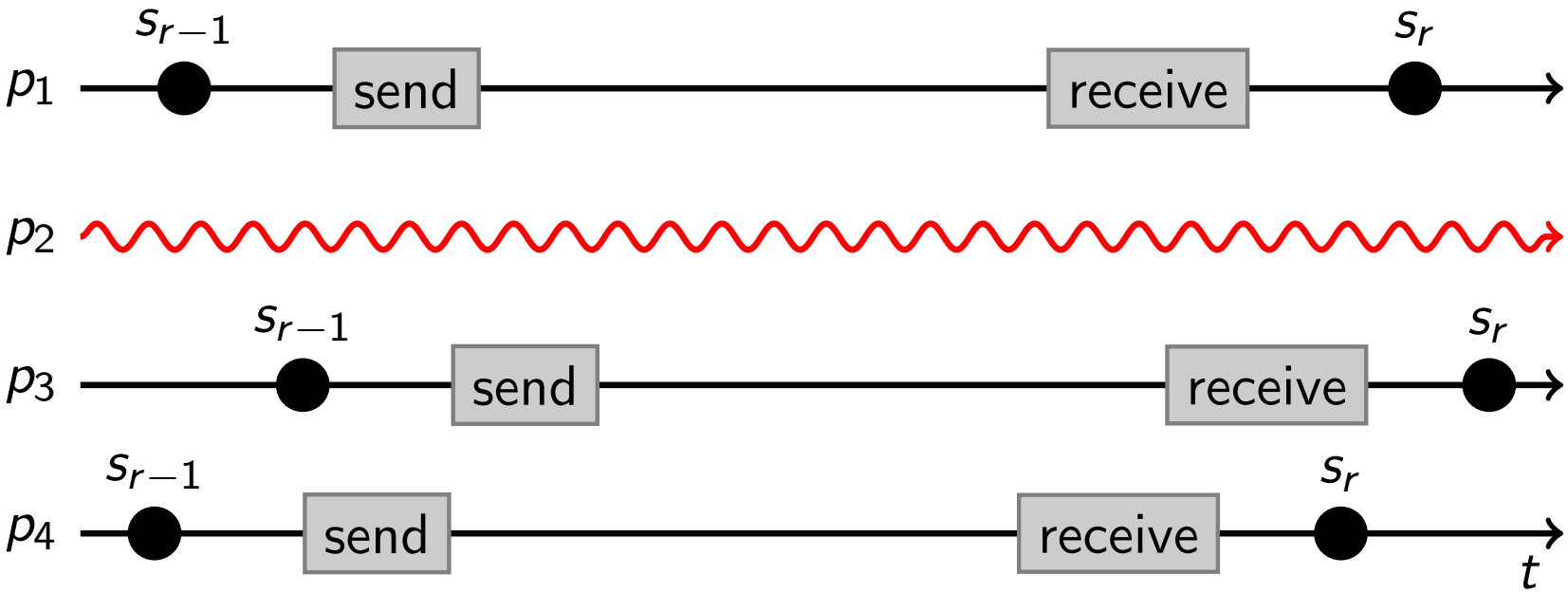
idealized view on organizing computations in rounds



Lock-step synchronous system



Lock-step synchronous system



with Byzantine process faults:

- every correct process receives all messages sent by correct processes in the round they were sent.

Types of Distributed Algorithms sync. vs. async

Synchronous

- all processes move in lock-step
- rounds
- a message sent in a round is received in the 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 sync. vs. async

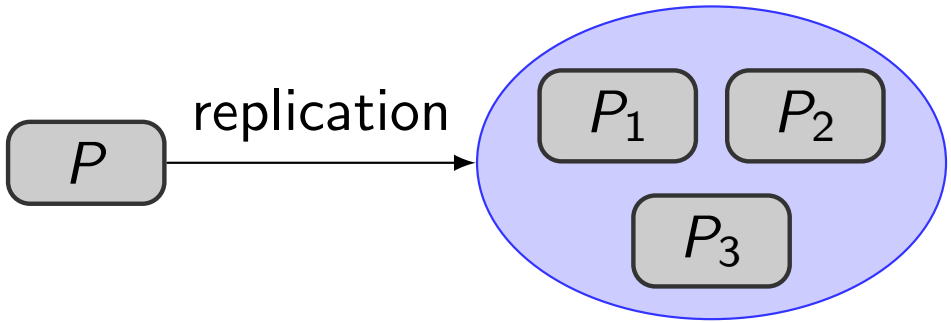
Synchronous

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

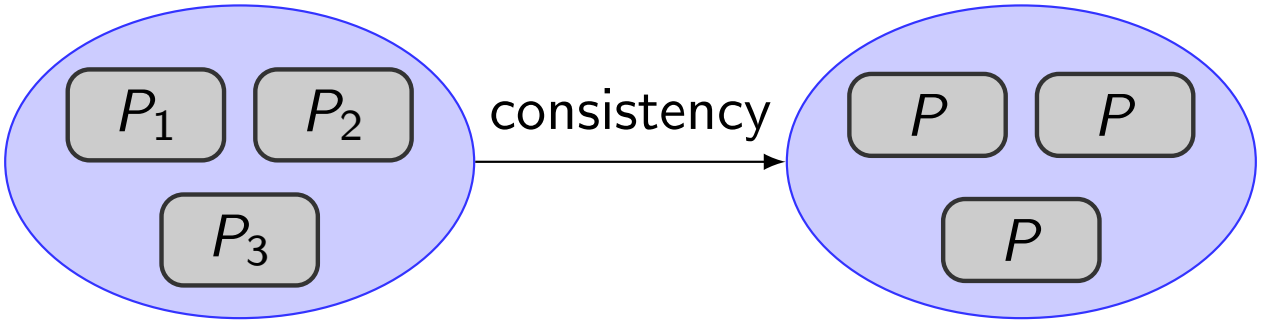
Asynchronous

- only one process moves at a time
- arbitrary interleavings of steps
- a message sent is received eventually
- important problems not solvable (FLP85)!

Where we stand



What we still need...



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 (LSP82):

> [...] *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 messengers

Fault tolerance – The Byzantine generals problem

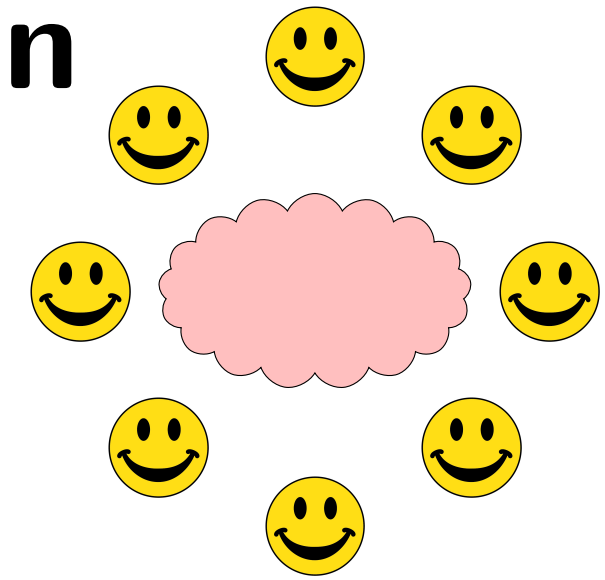
Lamport (this year's Turing laureate), Shostak, and Pease wrote in their *Dijkstra Prize in Distributed Computing* winning paper (LSP82):

> [...] *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 messengers

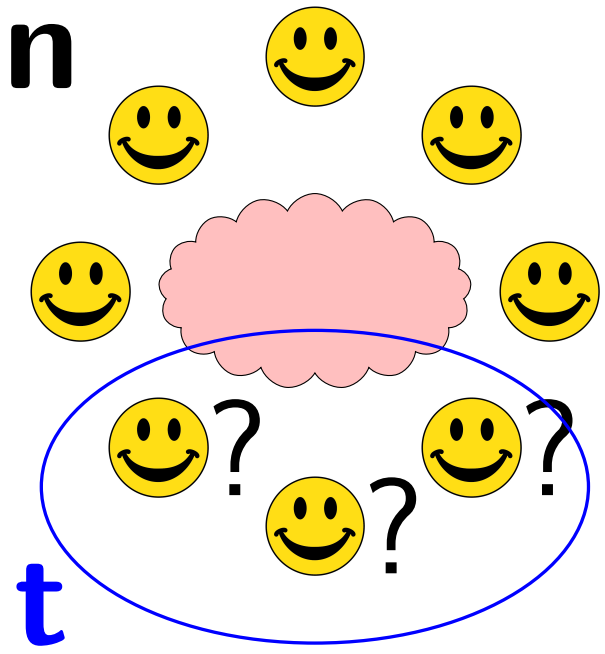
How to agree in the presence of traitors who can send different messages to different loyal generals?

Fault-tolerant distributed algorithms



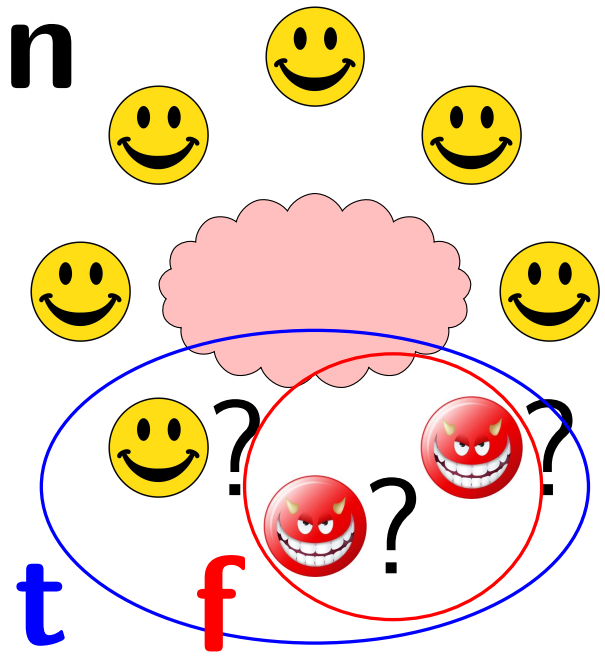
- n processes communicate by messages
- all processes know that at most t of them might be faulty
- f are actually faulty
- resilience conditions, e.g., $n > 3t \wedge t \geq f \geq 0$
- no masquerading: the processes know the origin of incoming messages

Fault-tolerant distributed algorithms



- n processes communicate by messages
- 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
- 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:
  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:
  faulty processes can do anything

Fault models — the ugly truth

A photo of a Byzantine fault:

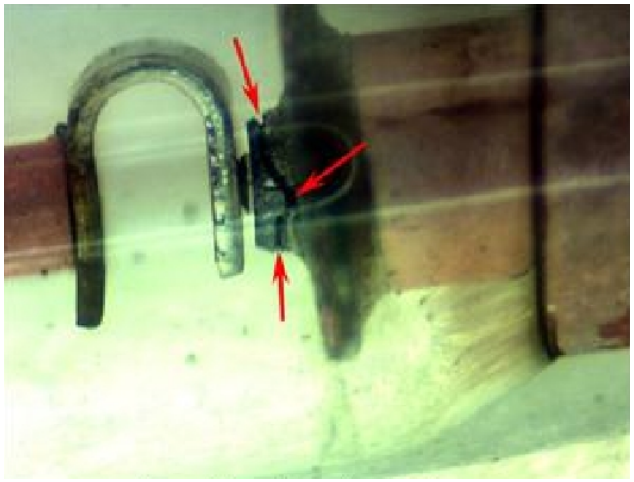


photo by Driscoll (Honeywell)
he reports Byzantine behavior on the Space Shuttle computer network

other sources of faults: bit-flips in memory, power outage, disconnection from the network, etc.

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 attack must be based on the will of 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 attack must be based on the will of 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 attack must be based on the will of one loyal general

termination. Every correct process eventually decides.
at some point negotiations must be over

satisfying only two properties?

Solving consensus

A system solves consensus if
each process's initial states are partitioned into ones that represent initial value 0 and 1. (e.g., with a binary variable `input`)

each process's states are partitioned into sets representing

- undecided
- 0-decided
- 1-decided

(e.g., with a variable `output` initially set to $\perp$)

All runs "satisfy" the properties of consensus.

- e.g. irrevocability: if in some run a process p_i's state s_i^j is a v-decided state then for all successor states, s_i^k is a v-decided state

Solving consensus

A system solves consensus if
each process's initial states are partitioned into ones that represent initial value 0 and 1. (e.g., with a binary variable `input`)

each process's states are partitioned into sets representing

- undecided
- 0-decided
- 1-decided

(e.g., with a variable `output` initially set to $\perp$)

All runs "satisfy" the properties of consensus.

- e.g. irrevocability: if in some run a process p_i's state s_i^j is a v-decided state then for all successor states, s_i^k is a v-decided state

let's try to solve it. . .

Our case study. . .

In this lecture: asynchronous FTDA

Still, problems that are solvable

- relaxations of consensus:

  reliable broadcast. relaxed termination

  condition-based consensus properties required only in runs from specific initial states

  The Paxos idea fault-tolerant distributed algorithms that are safe and make progress only if you are "lucky"

- adding information to the system

  failure detector based atomic commitment. distributed databases

  failure detector based consensus. and atomic broadcast

Our case-study problem

Unforgeability. If $v_i = \text{FALSE}$ for all correct processes i, then for all correct processes j, 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 accept_j to TRUE.

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

Our case-study problem

Unforgeability. If $v_i = \text{FALSE}$ for all correct processes i, then for all correct processes j, 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 accept_j to TRUE.

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

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

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

Our case-study problem

Unforgeability. If $v_i = \text{FALSE}$ for all correct processes i, then for all correct processes j, 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 accept_j to TRUE.

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

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

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

difference to consensus?

Our case-study problem

Unforgeability. If $v_i = \text{FALSE}$ for all correct processes i, then for all correct processes j, 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 accept_j to TRUE.

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

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

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

can be formalized in LTL

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;
```

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;
```

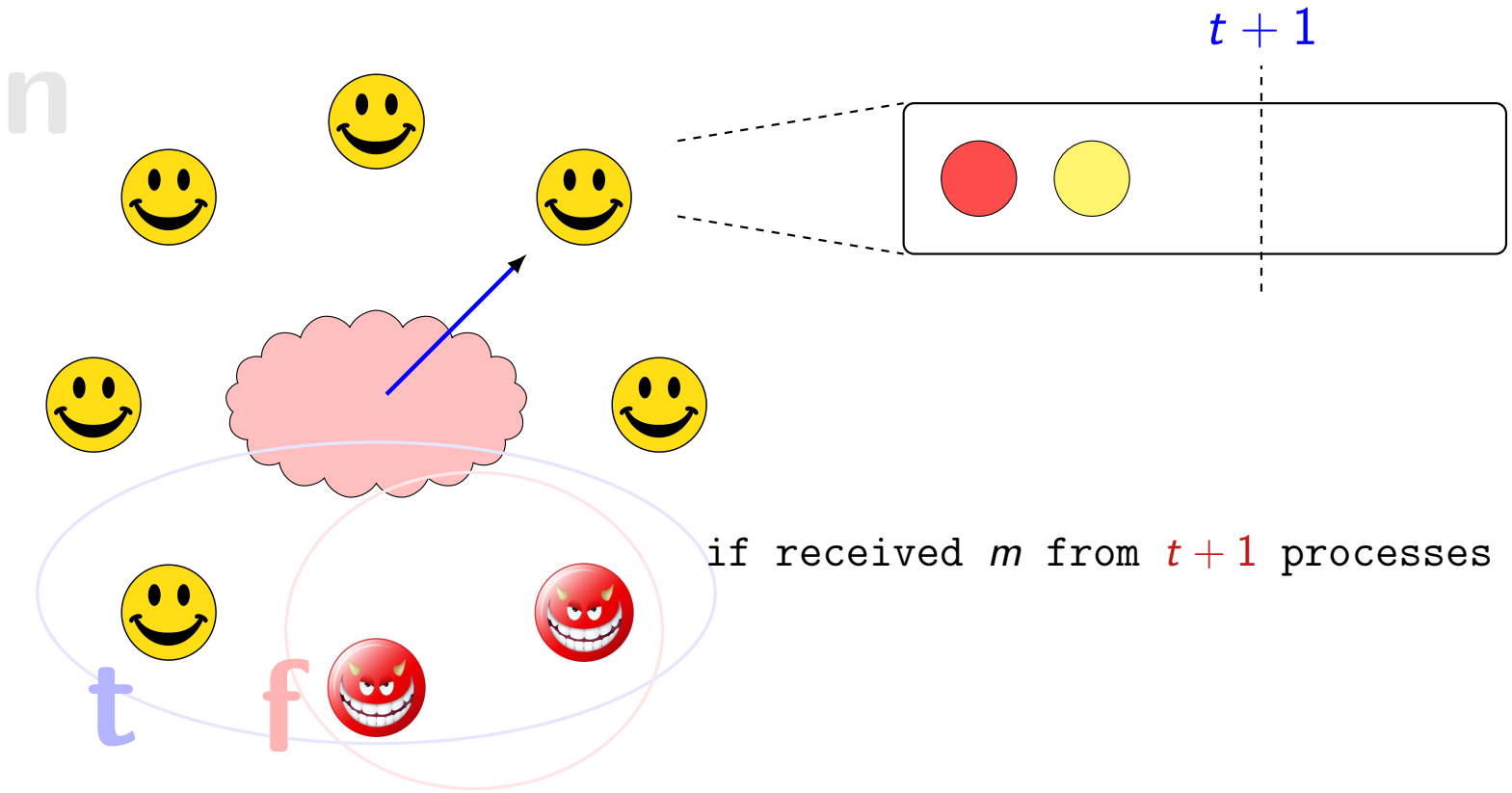
asynchronous

t Byzantine faults

correct if $n > 3t$

resilience condition RC

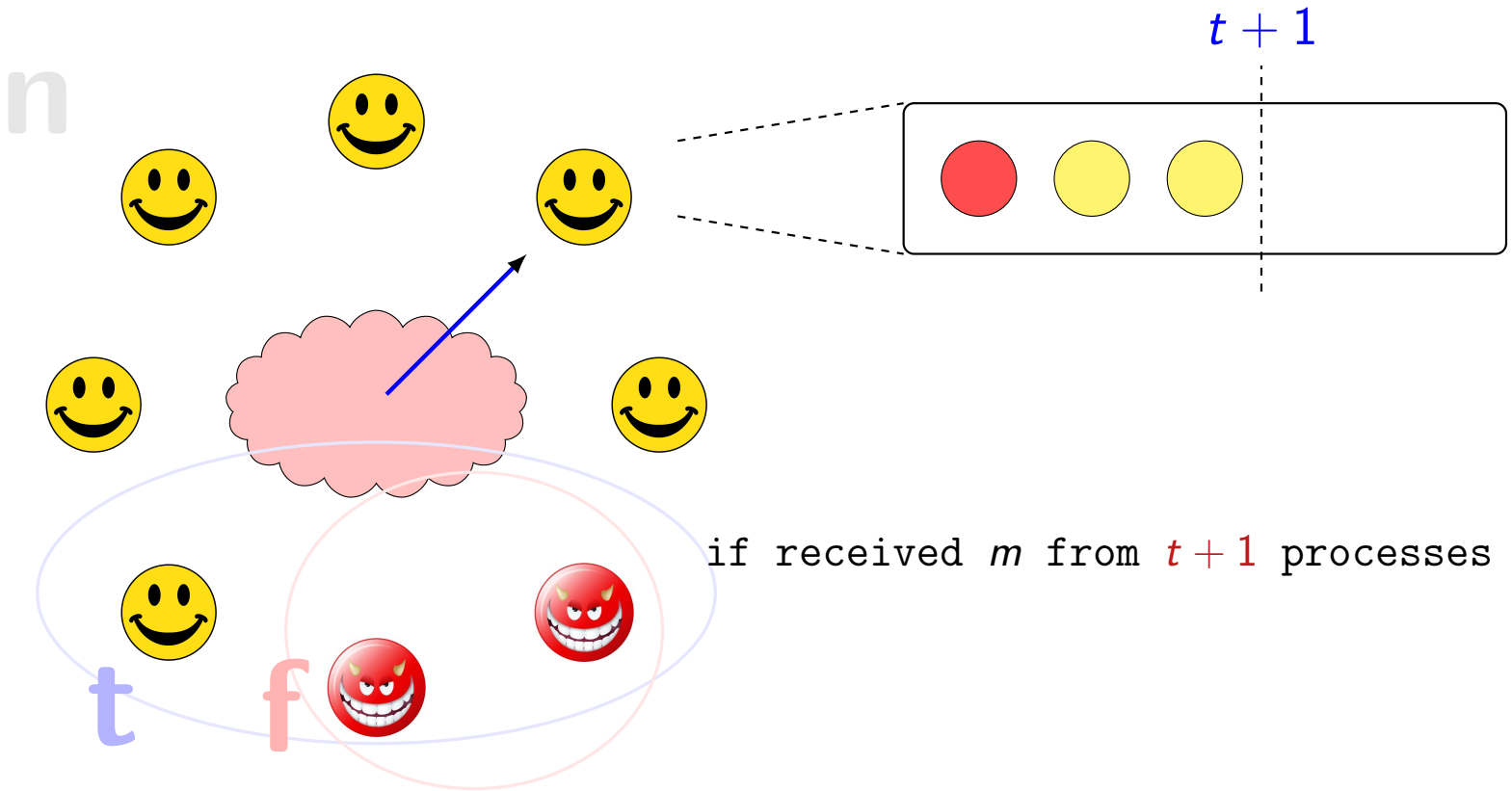
Basic mechanisms used by the algorithm



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

Correct processes count distinct incoming messages

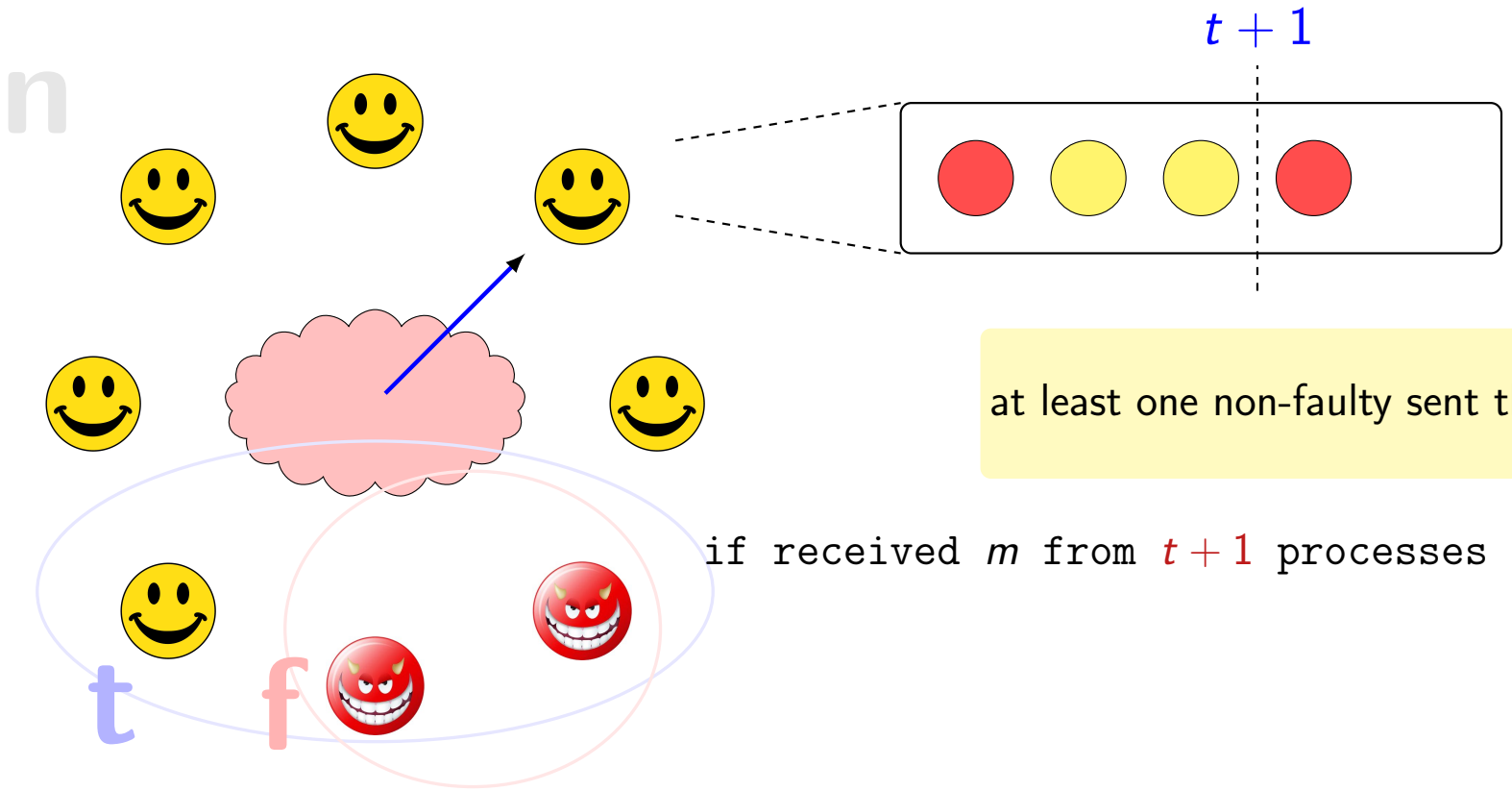
Basic mechanisms used by the algorithm



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

Correct processes count distinct incoming messages

Basic mechanisms used by the algorithm



at least one non-faulty sent the message

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

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, accept_j remains FALSE forever.

```
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;
```

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
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;
```

- 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, accept_j remains FALSE forever.

```
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;
```

- 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
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;
```

- 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, accept_j remains FALSE forever.

```
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;
```

- 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
- Because $n > 3t$, 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.

```
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;
```

- 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
- Because $n > 3t$, 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, accept_j remains FALSE forever.

```
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;
```

- 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
- Because $n > 3t$, 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, accept_j remains FALSE forever.

```
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;
```

- 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
- Because $n > 3t$, 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, accept_j remains FALSE forever.

```
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;
```

- 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
- Because $n > 3t$, 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, accept_j remains FALSE forever.

```
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;
```

- 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
- Because $n > 3t$, 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 accept_j to TRUE.

```
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;
```

Proof: Completeness

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

```
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;
```

- 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 accept_j to TRUE.

```
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;
```

- 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 accept_j to TRUE.

```
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;
```

- 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 accept_j to TRUE.

```
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;
```

- 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 accept_i to TRUE, then eventually all correct processes j set accept_j to TRUE.

```
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;
```

Proof: Relay

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

```
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;
```

- Correct process executes line 16

Proof: Relay

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

```
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;
```

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

Proof: Relay

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

```
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;
```

- 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 accept$_i$ to TRUE, then eventually all correct processes j set accept$_j$ to TRUE.

```
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;
```

- 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.

```
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;
```

- 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 accept$_i$ to TRUE, then eventually all correct processes j set accept$_j$ to TRUE.

```
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;
```

- 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 accept$_i$ to TRUE, then eventually all correct processes j set accept$_j$ to TRUE.

```
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;
```

- 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 accept$_i$ to TRUE, then eventually all correct processes j set accept$_j$ to TRUE.

```
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;
```

- 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 $accept_i$ to TRUE, then eventually all correct processes j set $accept_j$ to TRUE.

```
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;
```

- 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*) ≥ 1 or
  msg_count(*echo*, *i*) > (*n* + *k*)/2 or
  msg_count(*ready*, *i*) ≥ *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*) ≥ *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*) ≥ 2*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
decide i

FIG. 3. An asynchronous Byzantine Agreement protocol.

Condition-based consensus

Function *Consensus*(v_i)

(1) **foreach** $j \in [1..n]$ **do** $V_i[j] \leftarrow \perp$ **enddo**; % Intialization%

%——————————————————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 ($f < n/2$)

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 (LR93)
  - clock synchronization algorithm (MS06)

End of Part I

References I

Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson.
Impossibility of distributed consensus with one faulty process.
J. ACM, 32(2):374–382, 1985.
http://doi.acm.org/10.1145/3149.214121.

P. Lincoln and J. Rushby.
A formally verified algorithm for interactive consistency under a hybrid fault model.
In *FTCS-23*, pages 402–411, 1993.
http://dx.doi.org/10.1109/FTCS.1993.627343.

Leslie Lamport, Robert E. Shostak, and Marshall C. Pease.
The byzantine generals problem.
ACM Trans. Program. Lang. Syst., 4(3):382–401, 1982.

Mahyar R. Malekpour and Radu Siminiceanu.
Comments on the byzantine self-stabilizing pulse synchronization protocol: Counterexamples.
Technical Report TM-2006-213951, NASA, 2006.