From Byzantine Generals to Hackers

Leslie Lamport
Microsoft Research

Part I

Designing Computer Systems

To Fly an Airplane

The 1970s

The 1970s

Oil embargo against the U.S.

The 1970s

Oil embargo against the U.S.

Can save fuel by making planes aerodynamically unstable.

The 1970s

Oil embargo against the U.S.

Can save fuel by making planes aerodynamically unstable.

Such planes can be flown only by computer.

The 1970s

Oil embargo against the U.S.

Can save fuel by making planes aerodynamically unstable.

Such planes can be flown only by computer.

If the computer stops working for 20ms, the wings fall off.

The 1970s

Oil embargo against the U.S.

Can save fuel by making planes aerodynamically unstable.

Such planes can be flown only by computer.

If the computer stops working for 20ms, the wings fall off.

How do we make computers reliable?

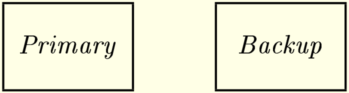
Primary-Backup

Primary-Backup

Use two computers: a primary and a backup.

Primary-Backup

Use two computers: a primary and a backup.



Primary-Backup

Use two computers: a primary and a backup.

If the primary fails, switch to the backup.

Primary-Backup

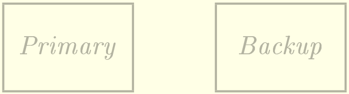
Use two computers: a primary and a backup.

If the primary fails, switch to the backup.

To handle multiple failures: have backup of backup, backup of backup of backup, . . .

Primary-Backup

Use two computers: a primary and a backup.



If the primary fails, switch to the backup.

To handle multiple failures: have backup of backup, backup of backup of backup, . . .

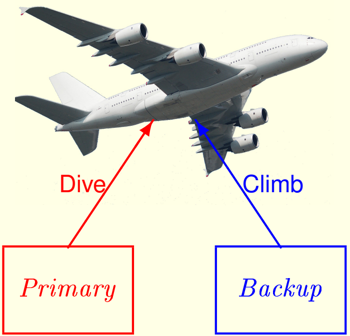
The problem with primary-backup

The problem with primary-backup

What if the primary malfunctions but keeps running?

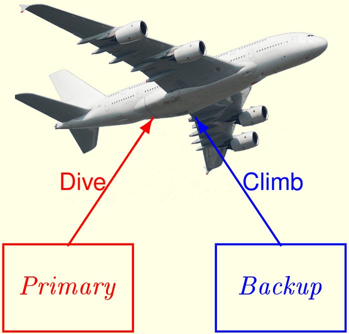
The problem with primary-backup

What if the primary malfunctions but keeps running?



The problem with primary-backup

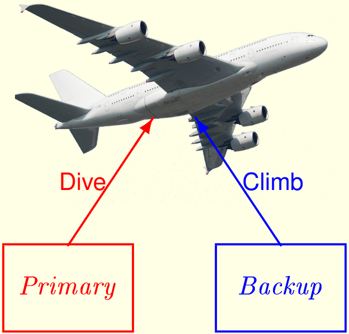
What if the primary malfunctions but keeps running?



Which one does the airplane listen to?

The problem with primary-backup

What if the primary malfunctions but keeps running?



Which one does the airplane listen to?

the actuators that control the flaps, landing gear, etc.

Simple solution: the backup turns off a bad primary.

Simple solution: the backup turns off a bad primary.

What if a bad backup turns off a good primary?

Simple solution: the backup turns off a bad primary.

What if a bad backup turns off a good primary?

A system with two computers cannot tolerate the failure of one of them.

Typical engineering solution:

Assume that such failures are highly improbable.

Typical engineering solution:

Assume that such failures are highly improbable.

Good enough for a bank's computer, but not for an airplane's.

Typical engineering solution:

Assume that such failures are highly improbable.

Good enough for a bank's computer, but not for an airplane's.

FAA's requirement:

Probability of catastrophic failure less than 10^{-10} per hour.

Typical engineering solution:

Assume that such failures are highly improbable.

Good enough for a bank's computer, but not for an airplane's.

FAA's requirement:

Probability of catastrophic failure less than 10^{-10} per hour.

That's one failure every 10 million years.

Typical engineering solution:

Assume that such failures are highly improbable.

Good enough for a bank's computer, but not for an airplane's.

FAA's requirement:

Probability of catastrophic failure less than 10^{-10} per hour.

That's one failure every 10 million years.

You can't ensure that by engineering judgement

Typical engineering solution:

Assume that such failures are highly improbable.

Good enough for a bank's computer, but not for an airplane's.

FAA's requirement:

Probability of catastrophic failure less than 10^{-10} per hour.

That's one failure every 10 million years.

You can't ensure that by engineering judgement, or by ordinary testing.

Triple Modular Redundancy

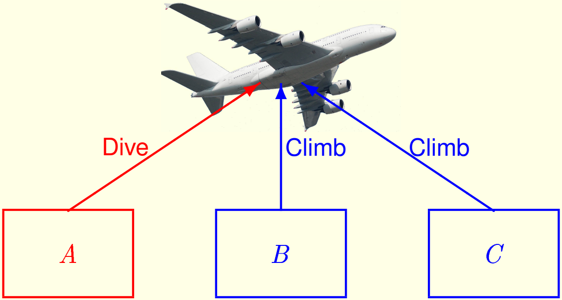
Triple Modular Redundancy

To handle one failure, use three computers.

Triple Modular Redundancy

To handle one failure, use three computers.

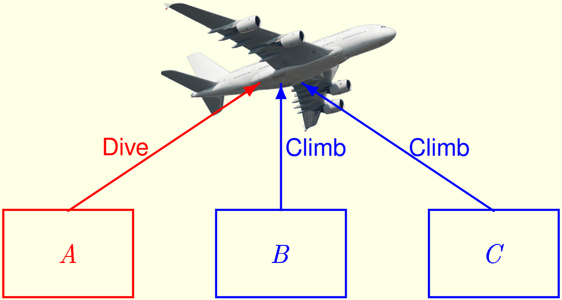
The plane listens to a majority.



Triple Modular Redundancy

To handle one failure, use three computers.

The plane listens to a majority.

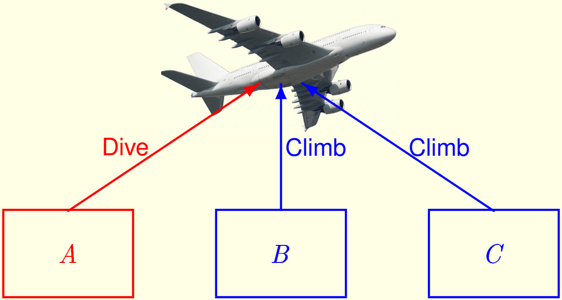


Majority voting done by actuators.

Triple Modular Redundancy

To handle one failure, use three computers.

The plane listens to a majority.



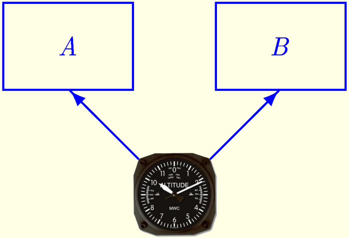
Majority voting done by actuators.

Control surface moved by three motors, any two of which can overpower the third.

One problem

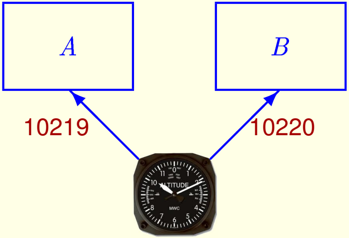
One problem

Good computers reading a sensor



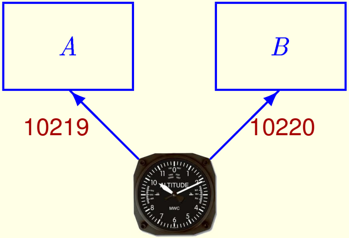
One problem

Good computers reading a sensor can get different values.



One problem

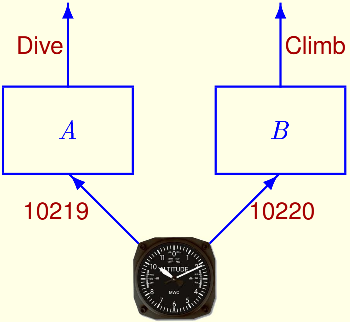
Good computers reading a sensor can get different values.



This can lead to very different decisions.

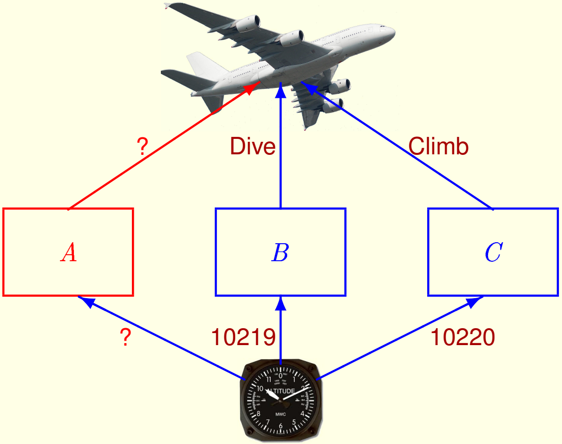
One problem

Good computers reading a sensor can get different values.



This can lead to very different decisions.

Leading to disaster.



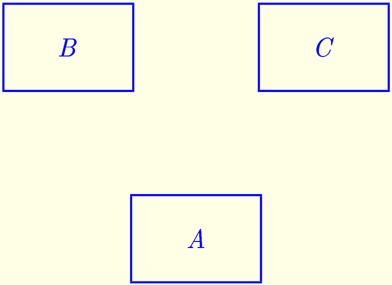
The Solution

The Solution

Each computer tells the others what inputs it read.

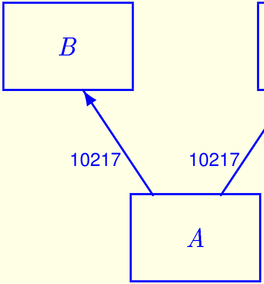
The Solution

Each computer tells the others what inputs it read.



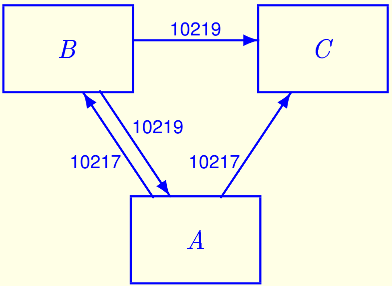
The Solution

Each computer tells the others what inputs it read.



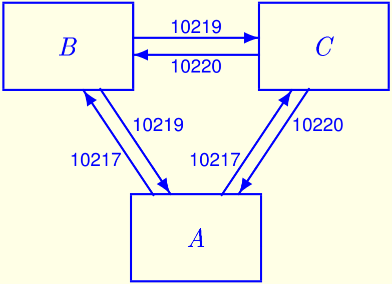
The Solution

Each computer tells the others what inputs it read.



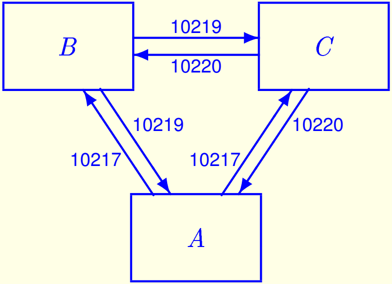
The Solution

Each computer tells the others what inputs it read.



The Solution

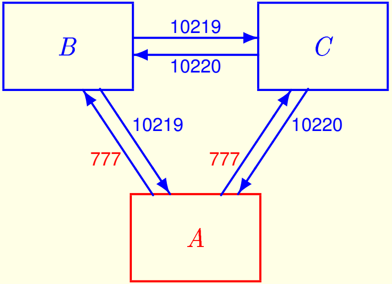
Each computer tells the others what inputs it read.



All computers get the same inputs: $\{$10217, 10219, 10220$\}$,
so they all make the same decision.

The Solution

Each computer tells the others what inputs it read.



All computers get the same inputs: {777, 10219, 10220},
so the good ones all make the same decision.

Triple modular redundancy was the state of the art in the early 1970s.

Triple modular redundancy was the state of the art in the early 1970s.

In the 1970s, NASA funded SRI to build a computer system for flying airplanes.

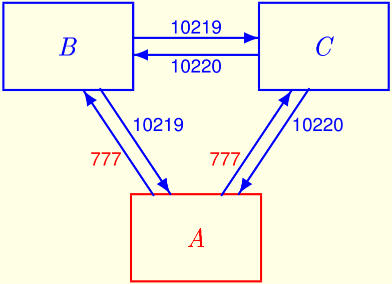
Triple modular redundancy was the state of the art in the early 1970s.

In the 1970s, NASA funded SRI to build a computer system for flying airplanes.

Someone at SRI (probably John Wensley) realized that TMR doesn't work.

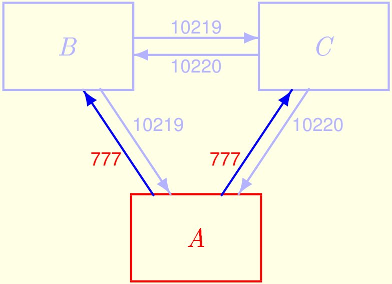
The problem with TMR

The problem with TMR

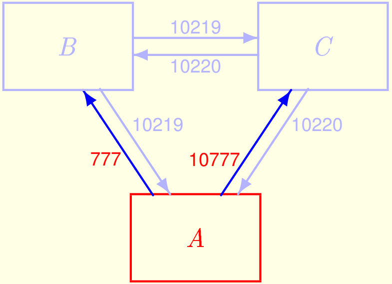


All computers get the same inputs: $\{$777, 10219, 10220$\}$,
so the good ones all make the same decision.

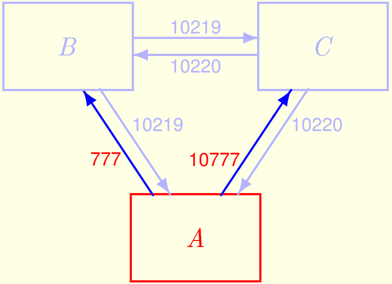
The problem with TMR



The problem with TMR



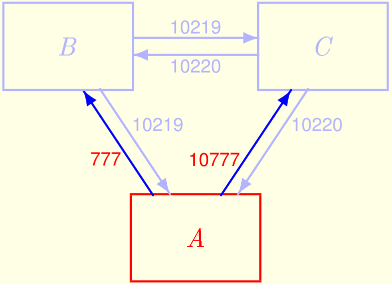
The problem with TMR



B gets $\{$777, 10219, 10220$\}$, decides Dive

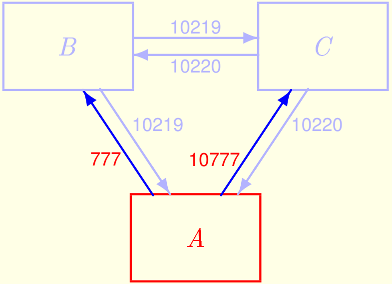
C gets $\{$10777, 10219, 10220$\}$, decides Climb

The problem with TMR



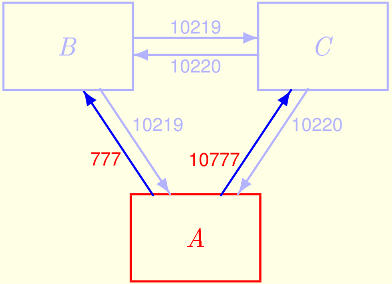
Is this highly improbable?

The problem with TMR



Is this highly improbable? Yes.

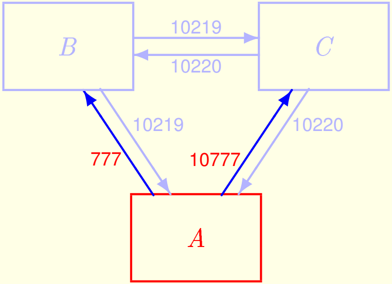
The problem with TMR



Is this highly improbable? Yes.

Is the probability less than 10^{-10} per hour?

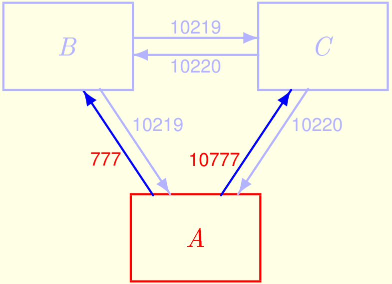
The problem with TMR



Is this highly improbable? Yes.

Is the probability less than 10^{-10} per hour? How can we tell?

The problem with TMR



It's not hard to come up with plausible failure scenarios that produce this situation.

The fundamental problem

The fundamental problem

A computer P must broadcast a value v to all computers

The fundamental problem

A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, then all nonfaulty computers get v.

The fundamental problem

A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, then all nonfaulty computers get v.
2. All nonfaulty computers get the same value.

The fundamental problem

A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, then all nonfaulty computers get v.
2. All nonfaulty computers get the same value.

2 follows from 1 if P is nonfaulty.

The Byzantine generals problem

A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, then all nonfaulty computers get v.
2. All nonfaulty computers get the same value.

The Byzantine generals problem

A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, then all nonfaulty computers get v.
2. All nonfaulty computers get the same value.

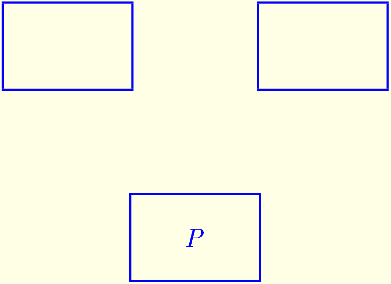
Problem described with generals, some of whom may be traitors.

A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.

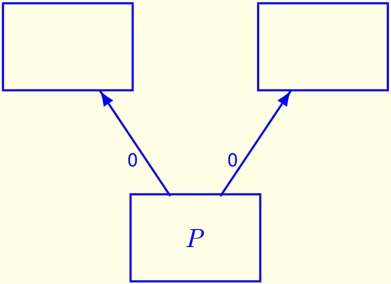
A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



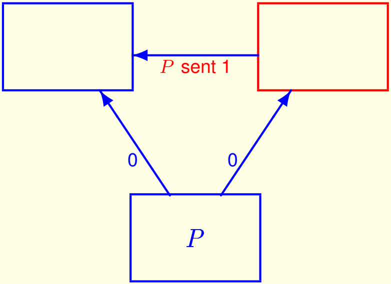
A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



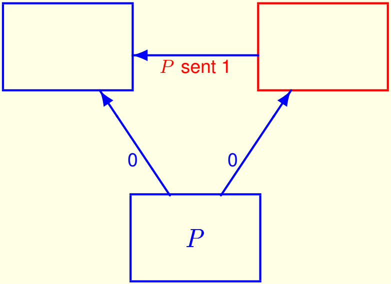
A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



A computer P must broadcast a value v to all computers such that:

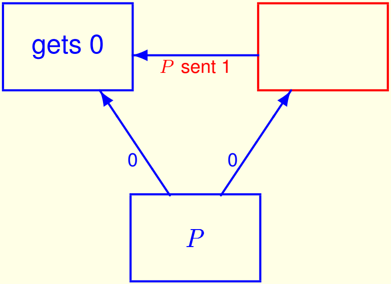
1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



By condition 1, must get 0.

A computer P must broadcast a value v to all computers such that:

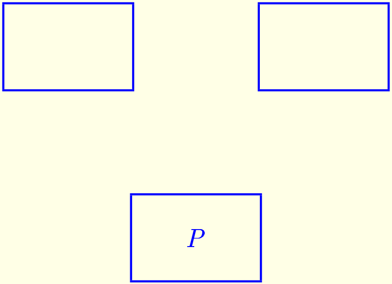
1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



By condition 1, must get 0.

A computer P must broadcast a value v to all computers such that:

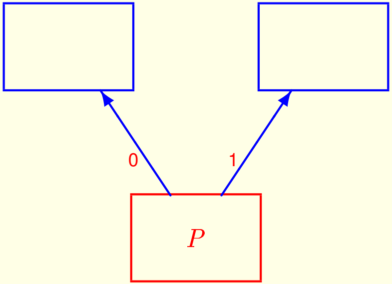
1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



A computer P must broadcast a value v to all computers such that:

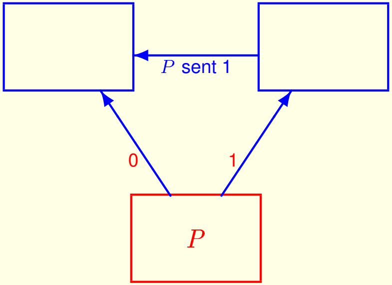
1. If P is nonfaulty, all nonfaulty computers get v.

2. All nonfaulty computers get the same value.



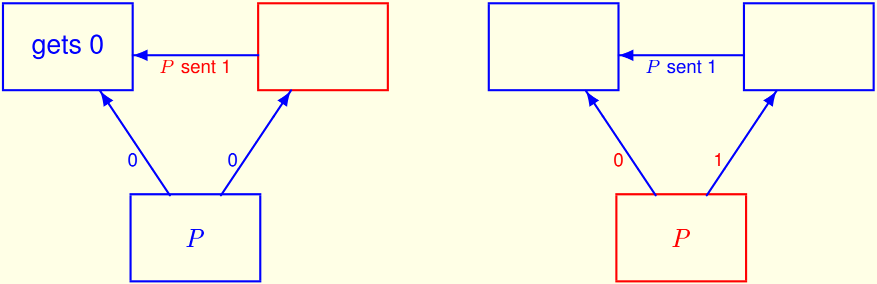
A computer P must broadcast a value v to all computers such that:

1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



A computer P must broadcast a value v to all computers such that:

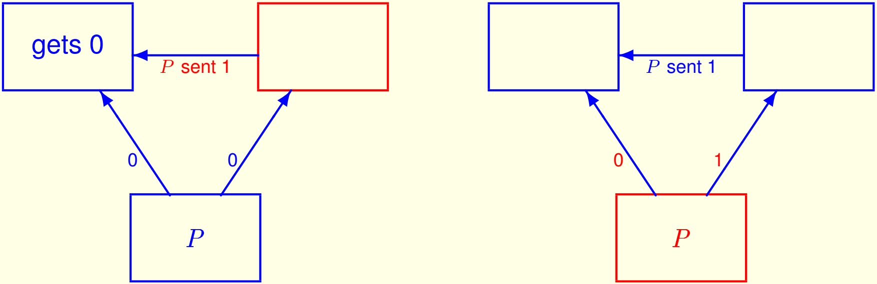
1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



Computer can't distinguish the two scenarios

A computer P must broadcast a value v to all computers such that:

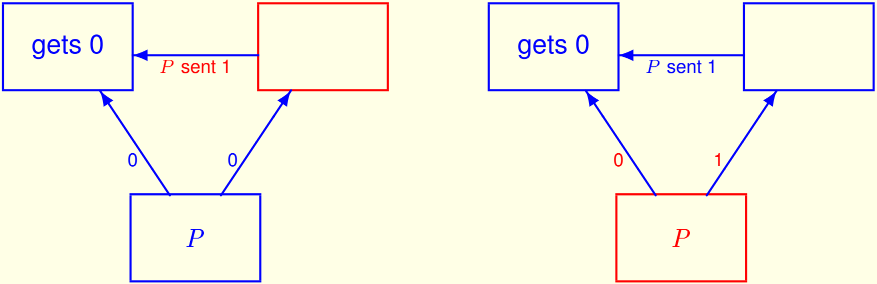
1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.



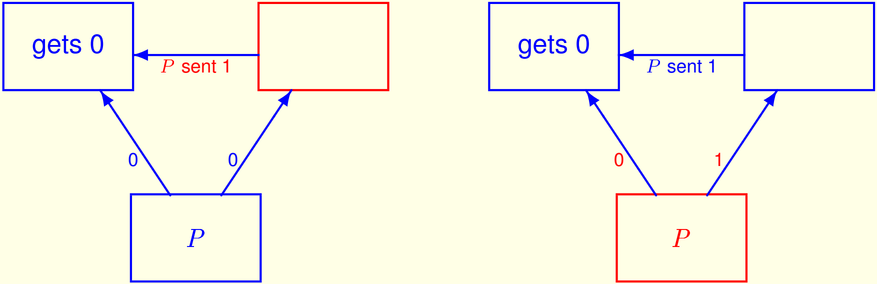
Computer can't distinguish the two scenarios ,
so it must get the same value in both.

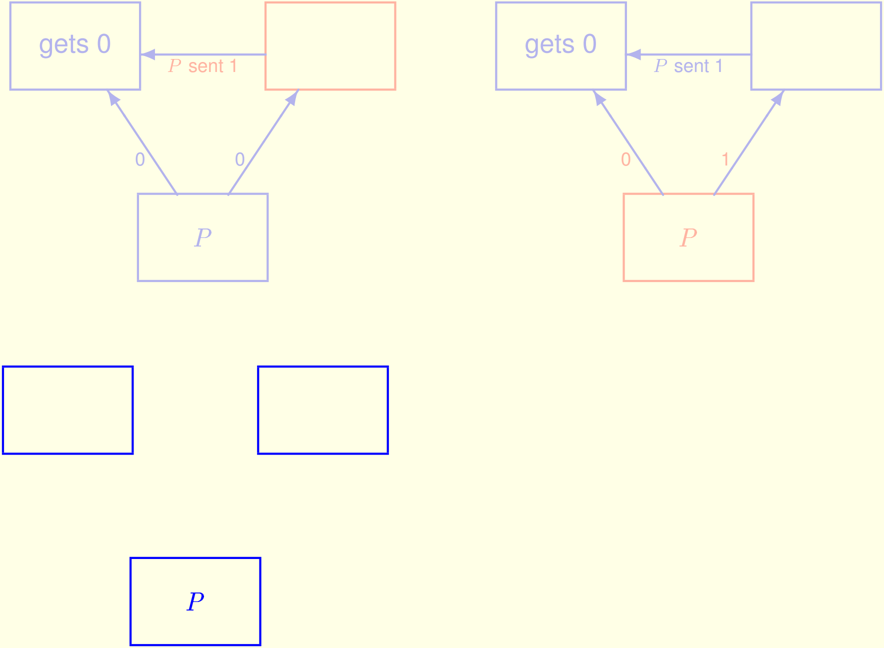
A computer P must broadcast a value v to all computers such that:

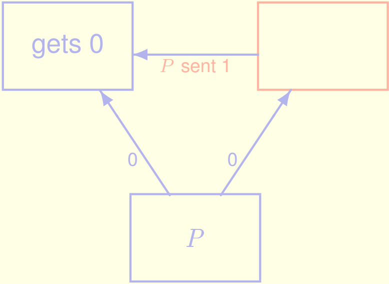
1. If P is nonfaulty, all nonfaulty computers get v.
2. All nonfaulty computers get the same value.

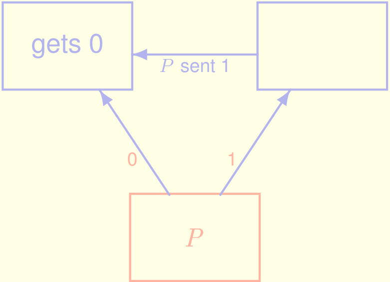


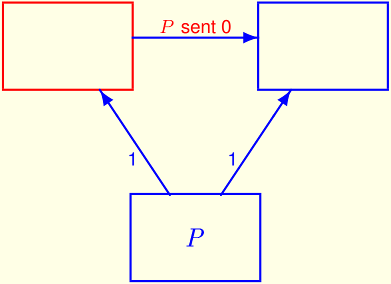
Computer can't distinguish the two scenarios ,
so it must get the same value in both.

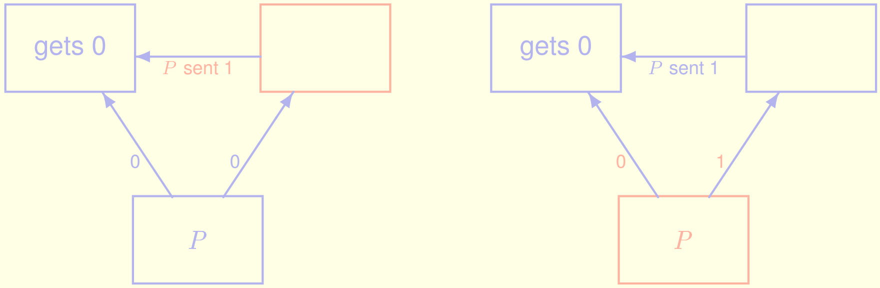

gets 0
P sent 1
0
0
P
gets 0
P sent 1
0
1
P

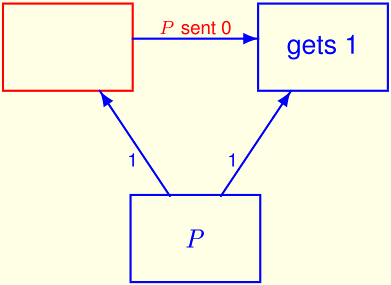

gets 0
P sent 1
0
0
P
gets 0
P sent 1
0
1
P
P


gets 0
P sent 1
0
0
P

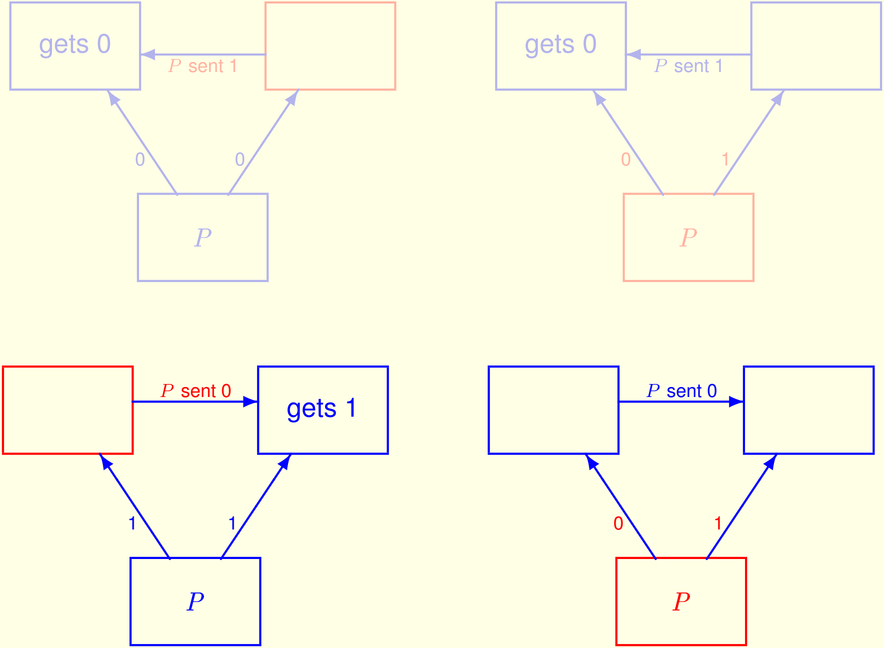

gets 0
P sent 1
0
1
P

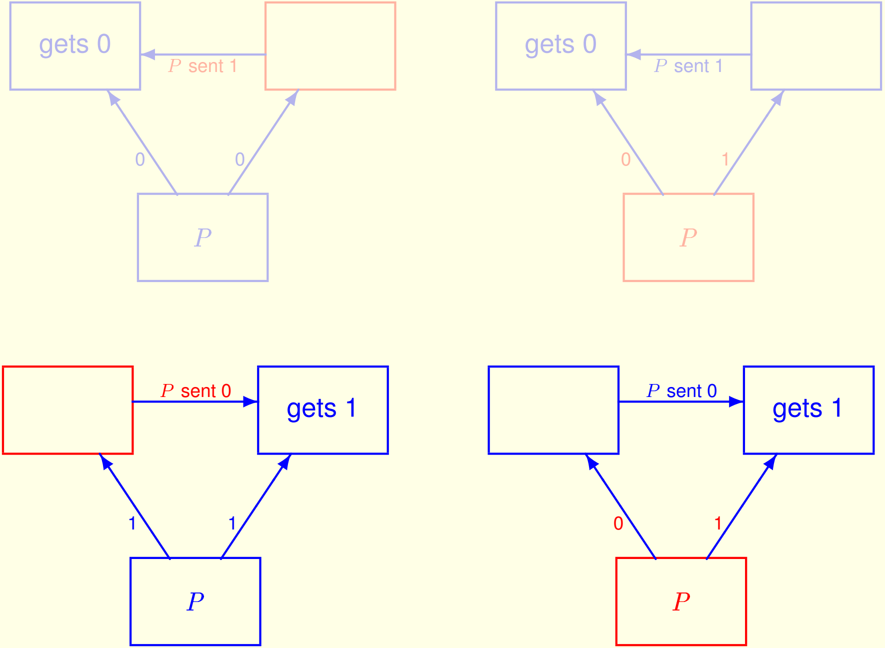

P sent 0
1
1
P

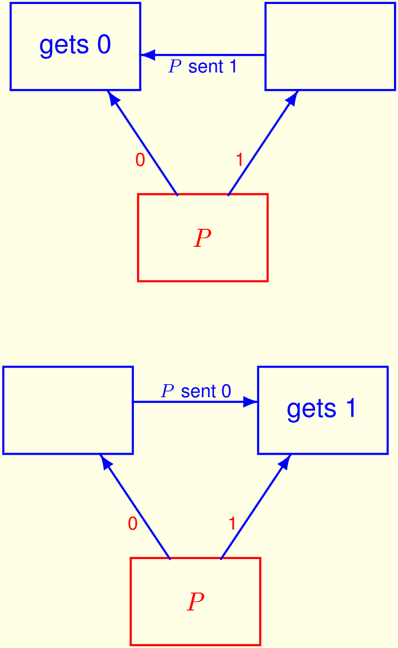




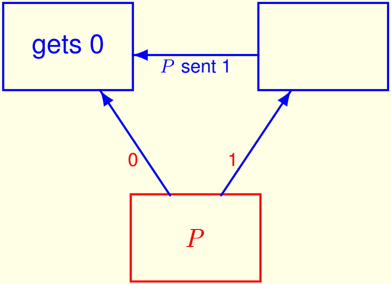
By condition 1, must get 1.

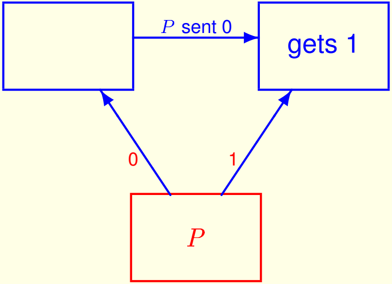

gets 0
P sent 1
0
0
P
gets 0
P sent 1
0
1
P
P sent 0
gets 1
1
1
P
P sent 0
0
1
P


gets 0
P sent 1
0
0
P
gets 0
P sent 1
0
1
P
P sent 0
gets 1
1
1
P
P sent 0
gets 1
0
1
P

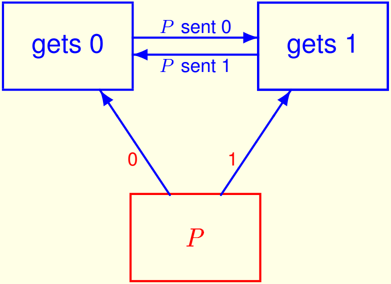

gets 0
P sent 1
0
1
P
P sent 0
gets 1
0
1
P

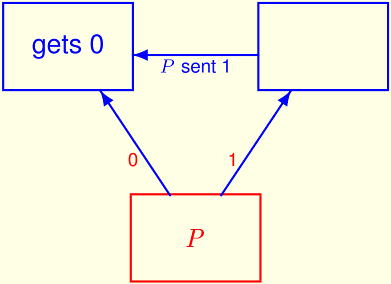
These are two views
of the same scenario.

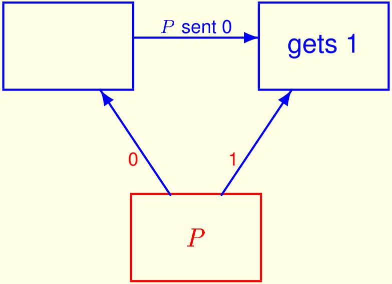




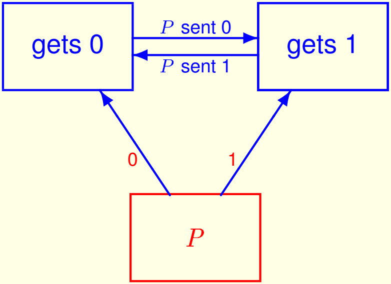
These are two views
of the same scenario.





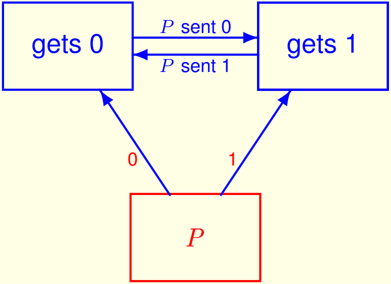


Condition 2 is violated.

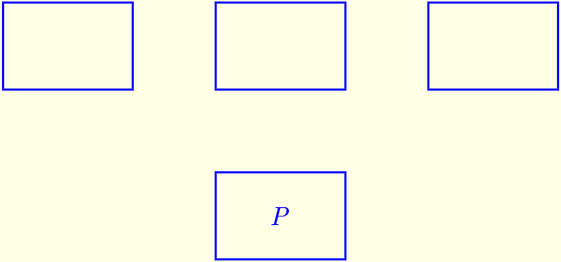


A rigorous version of this argument proves:

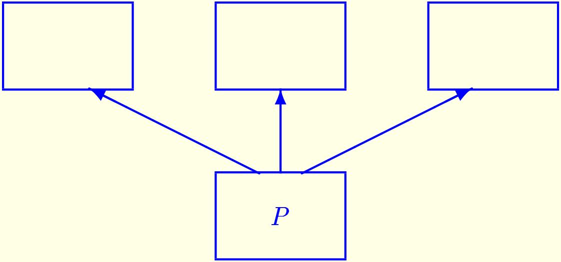
Theorem (Shostak) A solution to the Byzantine generals problem that tolerates one failure requires at least 4 computers.



A 4-computer solution

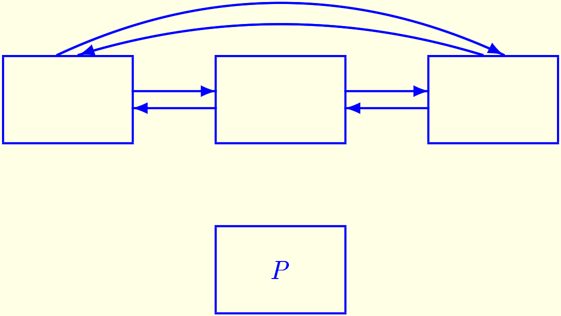


A 4-computer solution



P sends its value to the other computers.

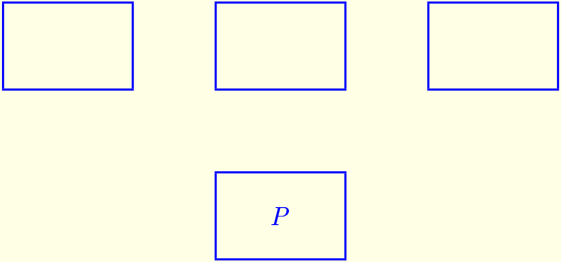
A 4-computer solution



P sends its value to the other computers.

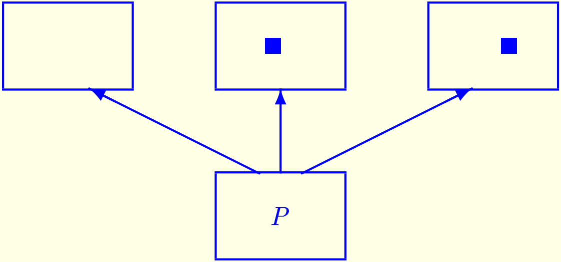
The other computers relay the value to one another.

A 4-computer solution



If P is nonfaulty, then each other computer receives at least 2 copies of P's value:

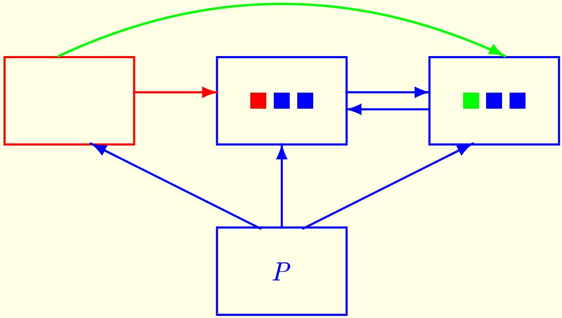
A 4-computer solution



If P is nonfaulty, then each other computer receives at least 2 copies of P's value:

It gets one directly from P.

A 4-computer solution

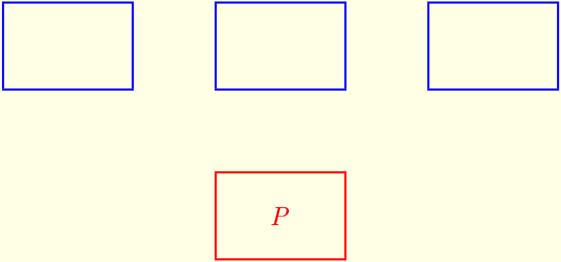


If P is nonfaulty, then each other computer receives at least 2 copies of P's value:

It gets one directly from P.

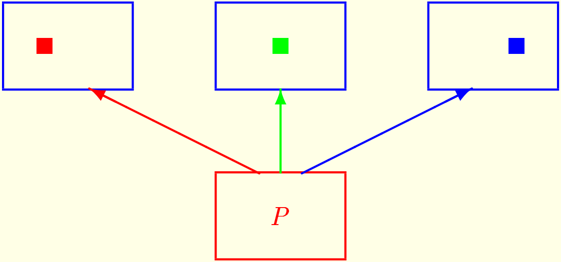
It gets one from another nonfaulty computer.

A 4-computer solution



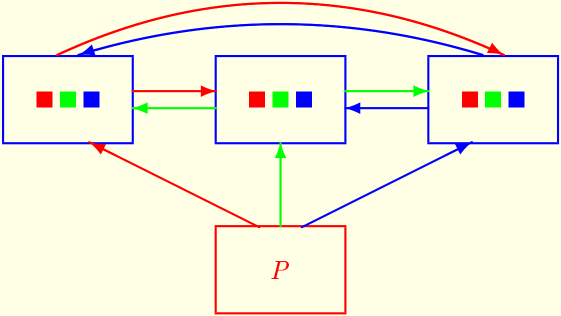
If P is faulty,

A 4-computer solution



If P is faulty,

A 4-computer solution



If P is faulty, then every other computer receives the same set of values.

A 4-computer solution

If P is nonfaulty, then each other computer receives at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

A 4-computer solution

If P is nonfaulty, then each other computer receives
at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

The Algorithm (Shostak)

A 4-computer solution

If P is nonfaulty, then each other computer receives at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

The Algorithm (Shostak)

P uses its own value.

A 4-computer solution

If P is nonfaulty, then each other computer receives at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

The Algorithm (Shostak)

P uses its own value.

For each other computer:

A 4-computer solution

If P is nonfaulty, then each other computer receives at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

The Algorithm (Shostak)

- P uses its own value.
- For each other computer:
  - If it receives 2 copies of a value, it takes that value.

A 4-computer solution

If P is nonfaulty, then each other computer receives at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

The Algorithm (Shostak)

P uses its own value.

For each other computer:

If it receives 2 copies of a value, it takes that value.

If P is nonfaulty, it sent the value.
If P is faulty, all others received those two values.

A 4-computer solution

If P is nonfaulty, then each other computer receives at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

The Algorithm (Shostak)

- P uses its own value.
- For each other computer:
  - If it receives 2 copies of a value, it takes that value.
  - Otherwise, it takes 42.

A 4-computer solution

If P is nonfaulty, then each other computer receives at least 2 copies of P's value.

If P is faulty, then every other computer receives the same set of values.

The Algorithm (Shostak)

- P uses its own value.
- For each other computer:
  - If it receives 2 copies of a value, it takes that value.
  - Otherwise, it takes 42.
    - P must be faulty, so all others will choose 42.

The General Case

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Proof: We assume an algorithm $\mathcal{A}$ that tolerates f failures with at most $3f$ computers

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Proof: We assume an algorithm $\mathcal{A}$ that tolerates f failures with at most $3f$ computers and obtain a contradiction by constructing a solution with 3 computers that tolerates 1 failure.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Proof: We assume an algorithm $\mathcal{A}$ that tolerates f failures with at most $3f$ computers and obtain a contradiction by constructing a solution with 3 computers that tolerates 1 failure.

Let the 3 actual computers simulate an execution of $\mathcal{A}$ by letting each of them simulate at most f of the computers of $\mathcal{A}$.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Proof: We assume an algorithm $\mathcal{A}$ that tolerates f failures with at most $3f$ computers and obtain a contradiction by constructing a solution with 3 computers that tolerates 1 failure.

Let the 3 actual computers simulate an execution of $\mathcal{A}$ by letting each of them simulate at most f of the computers of $\mathcal{A}$.

The one faulty actual computer simulates at most f faulty computers of $\mathcal{A}$.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Proof: We assume an algorithm $\mathcal{A}$ that tolerates f failures with at most $3f$ computers and obtain a contradiction by constructing a solution with 3 computers that tolerates 1 failure.

Let the 3 actual computers simulate an execution of $\mathcal{A}$ by letting each of them simulate at most f of the computers of $\mathcal{A}$.

The one faulty actual computer simulates at most f faulty computers of $\mathcal{A}$.

This produces a solution that tolerates 1 failure with three computers, which is impossible.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Theorem (Fischer and Lynch) Any solution that tolerates f failures requires at least $f + 1$ rounds.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Theorem (Fischer and Lynch) Any solution that tolerates f failures requires at least $f + 1$ rounds.

Round 1: P sends its value to the other computers.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Theorem (Fischer and Lynch) Any solution that tolerates f failures requires at least $f + 1$ rounds.

Round 1: P sends its value to the other computers.

Round 2: The other computers relay the value received from P.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Theorem (Fischer and Lynch) Any solution that tolerates f failures requires at least $f + 1$ rounds.

Round 1: P sends its value to the other computers.

Round 2: The other computers relay the value received from P.

Round 3: The other computers relay the values received in Round 2.

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Theorem (Fischer and Lynch) Any solution that tolerates f failures requires at least $f + 1$ rounds.

Round 1: P sends its value to the other computers.

Round 2: The other computers relay the value received from P.

Round 3: The other computers relay the values received in Round 2.

Round 4: The other computers relay the values received in Round 3.

⋮

The General Case

Theorem (Shostak) A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

Theorem (Fischer and Lynch) Any solution that tolerates f failures requires at least $f + 1$ rounds.

There is a solution (due to Pease) that tolerates f failures with $3f + 1$ computers and takes $f + 1$ rounds.

A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

A solution to the Byzantine generals problem that tolerates f failures requires at least $3f+1$ computers.

This is true if the solution has to work in all possible cases.

A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

This is true if the solution has to work in all possible cases.

But a solution is allowed to fail with probability less than

$$\frac{10^{-10}}{\text{number of executions per hour}}$$

A solution to the Byzantine generals problem that tolerates f failures requires at least $3f + 1$ computers.

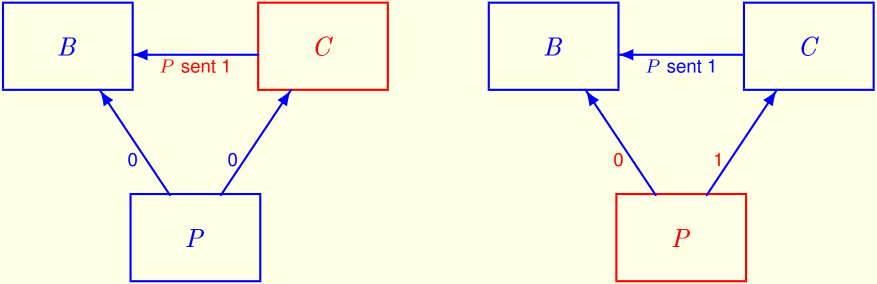
This is true if the solution has to work in all possible cases.

But a solution is allowed to fail with probability less than

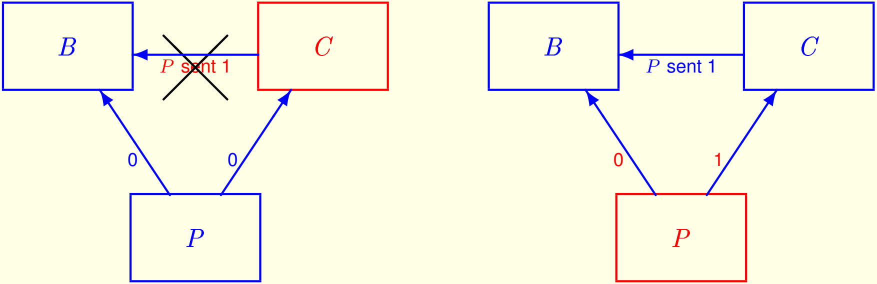
$$\frac{10^{-10}}{\text{number of executions per hour}}$$

Can we get a solution with fewer computers that works with very high probability?

The problem: B can't distinguish these two cases.



The problem: B can't distinguish these two cases.



The solution: prevent C from pretending P sent a value it didn't.

Digital Signatures

Digital Signatures

Proposed by Diffie and Hellman in mid-1970s.

Digital Signatures

Proposed by Diffie and Hellman in mid-1970s.

Few people had heard of them.

Digital Signatures

Proposed by Diffie and Hellman in mid-1970s.

Few people had heard of them. (I was one because Diffie is a friend of mine.)

Digital Signatures

Proposed by Diffie and Hellman in mid-1970s.

Few people had heard of them. (I was one because Diffie is a friend of mine.)

P can digitally sign its value; no other computer can forge its signature (with non-negligible probability).

Digital Signatures

Proposed by Diffie and Hellman in mid-1970s.

Few people had heard of them. (I was one because Diffie is a friend of mine.)

P can digitally sign its value; no other computer can forge its signature (with non-negligible probability).

Using digital signatures, a Byzantine generals solution tolerating f failures needs only $2f + 1$ computers (but still needs $f + 1$ rounds).

Digital Signatures

Proposed by Diffie and Hellman in mid-1970s.

Few people had heard of them. (I was one because Diffie is a friend of mine.)

P can digitally sign its value; no other computer can forge its signature (with non-negligible probability).

Using digital signatures, a Byzantine generals solution tolerating f failures needs only $2f + 1$ computers (but still needs $f + 1$ rounds).

First practical implementation against malicious forgery by Rivest, Shamir, and Adleman (1978).

Digital Signatures

Proposed by Diffie and Hellman in mid-1970s.

Few people had heard of them. (I was one because Diffie is a friend of mine.)

P can digitally sign its value; no other computer can forge its signature (with non-negligible probability).

Using digital signatures, a Byzantine generals solution tolerating f failures needs only $2f+1$ computers (but still needs $f+1$ rounds).

First practical implementation against malicious forgery by Rivest, Shamir, and Adleman (1978).

Easy to implement against forgery by failure.

What Do Airplane Manufacturers Use?

I don't know, but here's what I have heard.

What Do Airplane Manufacturers Use?

I don’t know, but here’s what I have heard.

Boeing: By email from a former Boeing engineer.

What Do Airplane Manufacturers Use?

I don’t know, but here’s what I have heard.

Boeing: By email from a former Boeing engineer.

“S--t! We have to use four.”

What Do Airplane Manufacturers Use?

I don't know, but here's what I have heard.

Boeing: By email from a former Boeing engineer.

"S--t! We have to use four."

Airbus: On a visit to a software group.

What Do Airplane Manufacturers Use?

I don’t know, but here’s what I have heard.

Boeing: By email from a former Boeing engineer.

“S--t! We have to use four.”

Airbus: On a visit to a software group.

Primary-backup.

What Do Airplane Manufacturers Use?

I don't know, but here's what I have heard.

Boeing: By email from a former Boeing engineer.

"S--t! We have to use four."

Airbus: On a visit to a software group.

Primary-backup.

Primary-backup can work, if each computer is carefully designed as a fault-tolerant multiprocess system.

What Do Airplane Manufacturers Use?

I don’t know, but here’s what I have heard.

Boeing: By email from a former Boeing engineer.

“S--t! We have to use four.”

Airbus: On a visit to a software group.

Primary-backup.

Primary-backup can work, if each computer is carefully designed as a fault-tolerant multiprocess system.

But engineers often think they can solve an unsolvable problem by reducing it to another unsolvable problem.

Part II

Designing Computer Systems To Run a Business

c. 1990

Why running a business is like flying an airplane

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

The computers must agree what to do next.

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

The computers must agree what to do next.

Marketing says: Raise price of widgets.

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

The computers must agree what to do next.

Marketing says: Raise price of widgets.

Customer says: Sell me a widget.

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

The computers must agree what to do next.

- Marketing says: Raise price of widgets.
- Customer says: Sell me a widget.
- All computers must agree which to do first.

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

The computers must agree what to do next.

- Marketing says: Raise price of widgets.
- Customer says: Sell me a widget.
- All computers must agree which to do first.

The basic problem: The computers must choose which one of a set of proposed commands to perform next.

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

The computers must agree what to do next.

- Marketing says: Raise price of widgets.
- Customer says: Sell me a widget.
- All computers must agree which to do first.

The basic problem: The computers must choose which one of a set of proposed commands to perform next.

Essentially the same problem as agreeing on the sensor inputs for flying an airplane.

Why running a business is like flying an airplane

Need multiple computers to tolerate failures.

The computers must agree what to do next.

Marketing says: Raise price of widgets.

Customer says: Sell me a widget.

All computers must agree which to do first.

The basic problem: The computers must choose which one of a set of proposed commands to perform next.

Essentially the same problem as agreeing on the sensor inputs for flying an airplane.

Given a solution to this problem, building a reliable system to run a business is a straightforward engineering task.

Why designing a system to run a business is easier

Why designing a system to run a business is easier

Need not be nearly as reliable.

Why designing a system to run a business is easier

Need not be nearly as reliable.

One failure every few hundred years is acceptable.

Why designing a system to run a business is easier

Need not be nearly as reliable.

One failure every few hundred years is acceptable.

Can assume that computers fail only by stopping.

Why designing a system to run a business is easier

Need not be nearly as reliable.

One failure every few hundred years is acceptable.

Can assume that computers fail only by stopping.

Need not handle malicious computers.

Why designing a system to run a business is easier

Need not be nearly as reliable.

One failure every few hundred years is acceptable.

Can assume that computers fail only by stopping.

Need not handle malicious computers.

Can tolerate occasional delays of a few seconds (or a few minutes for some businesses).

Why designing a system to run a business is easier

Need not be nearly as reliable.

One failure every few hundred years is acceptable.

Can assume that computers fail only by stopping.

Need not handle malicious computers.

Can tolerate occasional delays of a few seconds (or a few minutes for some businesses).

The business will not crash if the system stops for 20ms.

Why designing a system to run a business is harder

Why designing a system to run a business is harder

In an airplane, computers can communicate synchronously.

Why designing a system to run a business is harder

In an airplane, computers can communicate synchronously.

Messages are delivered in a (short) bounded time.

Why designing a system to run a business is harder

In an airplane, computers can communicate synchronously.

Messages are delivered in a (short) bounded time.

Messages are lost or late only if a computer fails.

Why designing a system to run a business is harder

In an airplane, computers can communicate synchronously.

In a business, computers communicate asynchronously.

Why designing a system to run a business is harder

In an airplane, computers can communicate synchronously.

In a business, computers communicate asynchronously.

Messages are delivered by complex software-controlled networks.

Why designing a system to run a business is harder

In an airplane, computers can communicate synchronously.

In a business, computers communicate asynchronously.

- Messages are delivered by complex software-controlled networks.
- Messages can be lost or take arbitrarily long to arrive.

Why designing a system to run a business is harder

In an airplane, computers can communicate synchronously.

In a business, computers communicate asynchronously.

FLP Theorem (Fischer, Lynch, and Paterson) No algorithm can ensure agreement among computers in an asynchronous system if a single computer can fail by stopping.

Overcoming FLP

Ensure that computers never disagree.

Overcoming FLP

Ensure that computers never disagree.

Ensure that the computers agree if they get lucky.

Overcoming FLP

Ensure that computers never disagree.

Ensure that the computers agree if they get lucky.

Ensure that they get lucky, except for occasional short periods.

Overcoming FLP

Ensure that computers never disagree.

Ensure that the computers agree if they get lucky.

Ensure that they get lucky, except for occasional short periods.

The system then never makes a mistake, and it keeps working except for occasional short periods of bad luck.

The Paxos Algorithm

A widely-used algorithm for reaching agreement in asynchronous computer systems.

The Paxos Algorithm

A widely-used algorithm for reaching agreement in asynchronous computer systems.

It assumes a method by which the computers select a good (non-failed) leader.

The Paxos Algorithm

A widely-used algorithm for reaching agreement in asynchronous computer systems.

It assumes a method by which the computers select a good (non-failed) leader.

This reduces the unsolvable problem of reaching agreement to the unsolvable problem of reaching agreement on who the leader is.

The Paxos Algorithm

A widely-used algorithm for reaching agreement in asynchronous computer systems.

It assumes a method by which the computers select a good (non-failed) leader.

This reduces the unsolvable problem of reaching agreement to the unsolvable problem of reaching agreement on who the leader is.

But the leader is not needed to prevent disagreement.

The Paxos Algorithm

A widely-used algorithm for reaching agreement in asynchronous computer systems.

It assumes a method by which the computers select a good (non-failed) leader.

This reduces the unsolvable problem of reaching agreement to the unsolvable problem of reaching agreement on who the leader is.

But the leader is not needed to prevent disagreement.

Computers never disagree even if there is no leader or several leaders.

The Paxos Algorithm

A widely-used algorithm for reaching agreement in asynchronous computer systems.

It assumes a method by which the computers select a good (non-failed) leader.

This reduces the unsolvable problem of reaching agreement to the unsolvable problem of reaching agreement on who the leader is.

But the leader is not needed to prevent disagreement.

Computers never disagree even if there is no leader or several leaders.

A unique, good leader is required only to reach a decision.

In Paxos, *getting lucky* means having a unique, good leader.

In Paxos, *getting lucky* means having a unique, good leader.

In a well-engineered system, it's easy to design a leader-selection algorithm that is lucky except during occasional short periods.

In Paxos, *getting lucky* means having a unique, good leader.

In a well-engineered system, it's easy to design a leader-selection algorithm that is lucky except during occasional short periods.

Paxos uses $2f + 1$ computers and can choose a command if at most f of them fail.

The Algorithm

The Algorithm

The leader-selection algorithm tries to keep a unique, good leader—selecting a new leader if the current leader fails.

The Algorithm

The leader-selection algorithm tries to keep a unique, good leader—selecting a new leader if the current leader fails.

When a computer L believes that it has just become the leader, it starts a *ballot* to try to get a command chosen.

The Algorithm

The leader-selection algorithm tries to keep a unique, good leader—selecting a new leader if the current leader fails.

When a computer L believes that it has just become the leader, it starts a *ballot* to try to get a command chosen.

The command is chosen if a majority of the computers *vote* for it in the ballot.

The Algorithm

The leader-selection algorithm tries to keep a unique, good leader—selecting a new leader if the current leader fails.

When a computer L believes that it has just become the leader, it starts a *ballot* to try to get a command chosen.

The command is chosen if a majority of the computers *vote* for it in the ballot.

Here's what happens if we're lucky and L is the unique leader.

L sends a *start ballot* message to the other computers.

L sends a *start ballot* message to the other computers.

The other computers reply with information about any votes they've cast in other ballots.

L sends a *start ballot* message to the other computers.

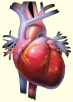
The other computers reply with information about any votes they've cast in other ballots.

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.

L sends a *start ballot* message to the other computers.

The other computers reply with information about any votes they've cast in other ballots.

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.



L sends a *start ballot* message to the other computers.

The other computers reply with information about any votes they've cast in other ballots.

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.

It votes for c and sends the message $vote\ for\ c$ to the other computers.

L sends a *start ballot* message to the other computers.

The other computers reply with information about any votes they've cast in other ballots.

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.

It votes for c and sends the message $vote\ for\ c$ to the other computers.

Each other computer that receives the $vote\ for\ c$ message votes for c and sends L a message saying that it has.

L sends a *start ballot* message to the other computers.

The other computers reply with information about any votes they've cast in other ballots.

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.

It votes for c and sends the message $vote\ for\ c$ to the other computers.

Each other computer that receives the $vote\ for\ c$ message votes for c and sends L a message saying that it has.

When L learns that at least f other computers have voted for c, then it knows that c has been chosen and informs the other computers.

This is what happens if the system is lucky.

This is what happens if the system is lucky.

If it's unlucky, some other process M will have started a competing ballot, and the other computers will start responding to M and ignoring L.

This is what happens if the system is lucky.

If it's unlucky, some other process M will have started a competing ballot, and the other computers will start responding to M and ignoring L.

If L still thinks that it is the leader, then it will start another ballot.

This is what happens if the system is lucky.

If it's unlucky, some other process M will have started a competing ballot, and the other computers will start responding to M and ignoring L.

If L still thinks that it is the leader, then it will start another ballot.

The algorithm guarantees that if two different ballots choose commands, then they both choose the same command.

This is what happens if the system is lucky.

If it's unlucky, some other process M will have started a competing ballot, and the other computers will start responding to M and ignoring L.

If L still thinks that it is the leader, then it will start another ballot.

The algorithm guarantees that if two different ballots choose commands, then they both choose the same command.

Eventually, the system will get lucky, and the leader-selection algorithm will choose a single good leader.

This is what happens if the system is lucky.

If it's unlucky, some other process M will have started a competing ballot, and the other computers will start responding to M and ignoring L.

If L still thinks that it is the leader, then it will start another ballot.

The algorithm guarantees that if two different ballots choose commands, then they both choose the same command.

Eventually, the system will get lucky, and the leader-selection algorithm will choose a single good leader.

If at most f computers have failed, that leader will start a ballot that completes and chooses a command.

Paxos will not fly an airplane

Paxos will not fly an airplane, but it is in the clouds.

Paxos will not fly an airplane, but it is in the clouds.

The data center doing your cloud computing is probably using Paxos.

Paxos will not fly an airplane, but it is in the clouds.

The data center doing your cloud computing is probably using Paxos.

It’s certainly using Paxos if it’s run by Microsoft or Google.

Part III

Designing Computer Systems
To Run a Byzantine Business

the 2000s

The Problem

The Problem

Run a business with computers that may have Byzantine (malicious) failures.

The Problem

Run a business with computers that may have Byzantine (malicious) failures.

Why?

The Problem

Run a business with computers that may have Byzantine (malicious) failures.

Why? For now, because it's fun.

The Problem

Run a business with computers that may have Byzantine (malicious) failures.

Why? For now, because it's fun.

The same basic problem: The computers must choose which one of a set of proposed commands to perform next.

The Solution

The Solution

Byzantine Paxos (Castro and Liskov) [they called it something else].

The Solution

Byzantine Paxos (Castro and Liskov) [they called it something else].

A new algorithm inspired by Paxos.

The Solution

Byzantine Paxos (Castro and Liskov) [they called it something else].

A new algorithm inspired by Paxos.

Uses $3f+1$ computers to tolerate the malicious failure of up to f of them.

The Solution

Byzantine Paxos (Castro and Liskov) [they called it something else].

A new algorithm inspired by Paxos.

Uses $3f+1$ computers to tolerate the malicious failure of up to f of them.

This is optimal. (Bracha and Toueg)

Another Approach: Byzantizing Paxos

Another Approach: Byzantizing Paxos

Can derive a class of algorithms that tolerate malicious failures

Another Approach: Byzantizing Paxos

Can derive a class of algorithms that tolerate malicious failures—including the Castrol-Liskov algorithm.

Another Approach: Byzantizing Paxos

Can derive a class of algorithms that tolerate malicious failures—including the Castrol-Liskov algorithm.

The idea: Have $2f + 1$ good computers execute Paxos, while f malicious computers try to foil them.

Another Approach: Byzantizing Paxos

Can derive a class of algorithms that tolerate malicious failures—including the Castrol-Liskov algorithm.

The idea: Have $2f + 1$ good computers execute Paxos, while f malicious computers try to foil them.

A good computer doesn't know which of the other computers are malicious.

The Key Idea

The Key Idea

Paxos works by sending messages.

The Key Idea

Paxos works by sending messages.

We require that each message M be accompanied by a proof that the Paxos algorithm being executed by the good computers allows M to be sent.

The Key Idea

Paxos works by sending messages.

We require that each message M be accompanied by a proof that the Paxos algorithm being executed by the good computers allows M to be sent.

In most cases, M can be sent only if the sender has received some set S of messages.

The Key Idea

Paxos works by sending messages.

We require that each message M be accompanied by a proof that the Paxos algorithm being executed by the good computers allows M to be sent.

In most cases, M can be sent only if the sender has received some set S of messages.

The fact that the messages in S were sent constitutes a proof that Paxos allows sending M.

The Key Idea

Paxos works by sending messages.

We require that each message M be accompanied by a proof that the Paxos algorithm being executed by the good computers allows M to be sent.

In most cases, M can be sent only if the sender has received some set S of messages.

The fact that the messages in S were sent constitutes a proof that Paxos allows sending M.

Have computers digitally sign the messages they send.
(Improperly signed messages are ignored.)

The Key Idea

Paxos works by sending messages.

We require that each message M be accompanied by a proof that the Paxos algorithm being executed by the good computers allows M to be sent.

In most cases, M can be sent only if the sender has received some set S of messages.

The fact that the messages in S were sent constitutes a proof that Paxos allows sending M.

Have computers digitally sign the messages they send.
(Improperly signed messages are ignored.)

Copies of the messages in S then prove those messages were sent

The Key Idea

Paxos works by sending messages.

We require that each message M be accompanied by a proof that the Paxos algorithm being executed by the good computers allows M to be sent.

In most cases, M can be sent only if the sender has received some set S of messages.

The fact that the messages in S were sent constitutes a proof that Paxos allows sending M.

Have computers digitally sign the messages they send.
(Improperly signed messages are ignored.)

Copies of the messages in S then prove those messages were sent, so they prove that Paxos allows sending M.

Paxos With Proofs

Paxos With Proofs

L sends a *start ballot* message to the other computers.

Paxos With Proofs

L sends a *start ballot* message to the other computers.

Allowed at any time, no proof needed.

Paxos With Proofs

The other computers reply with information about any votes they've cast in other ballots.

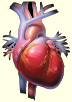
Paxos With Proofs

The other computers reply with information about any votes they've cast in other ballots.

They send proofs that each of their votes was allowed by a leader's $vote\ for$ message.

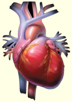
Paxos With Proofs

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.



Paxos With Proofs

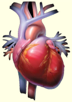
When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.



L has to receive messages with proofs from $2f$ other computers, so it has them from at least f good ones.

Paxos With Proofs

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.



L has to receive messages with proofs from $2f$ other computers, so it has them from at least f good ones.

The messages from any f of those other computers must, by themselves, prove that L is allowed to choose c.

Paxos With Proofs

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.

L has to receive messages with proofs from $2f$ other computers, so it has them from at least f good ones.

The messages from any f of those other computers must, by themselves, prove that L is allowed to choose c.

L votes for c and sends the message $vote\ for\ c$ to the other computers.

Paxos With Proofs

When L has heard from at least f other computers, either (a) it learns that a particular command c might already have been chosen, or else (b) it lets c be any proposed command.

L has to receive messages with proofs from $2f$ other computers, so it has them from at least f good ones.

The messages from any f of those other computers must, by themselves, prove that L is allowed to choose c.

L votes for c and sends the message $vote\ for\ c$ to the other computers.

L sends a proof for this message consisting of all the messages and their proofs it received from those $2f$ other computers.

Paxos With Proofs

Each other computer that receives the $vote\ for\ c$ message votes for c and sends L a message saying that it has.

Paxos With Proofs

Each other computer that receives the $vote\ for\ c$ message votes for c and sends L a message saying that it has.

What if a malicious L tells different computers to vote for different commands c?

Paxos With Proofs

Each other computer that receives the $vote\ for\ c$ message votes for c and sends L a message saying that it has.

What if a malicious L tells different computers to vote for different commands c?

Paxos does not allow different computers to vote for different commands in the same ballot.

Paxos With Proofs

L and each other computer that receives the $vote\ for\ c$ message (with proof) sends an $ok\ to\ vote\ for\ c$ message to all computers

Paxos With Proofs

L and each other computer that receives the $vote\ for\ c$ message (with proof) sends an $ok\ to\ vote\ for\ c$ message to all computers (including itself).

Convenient to pretend computers send themselves messages.

Paxos With Proofs

L and each other computer that receives the $vote\ for\ c$ message (with proof) sends an $ok\ to\ vote\ for\ c$ message to all computers (including itself).

Every computer that receives $ok\ to\ vote\ for\ c$ messages from $2f+1$ computers votes for v and sends a message to all computers saying it has.

Paxos With Proofs

L and each other computer that receives the $vote\ for\ c$ message (with proof) sends an $ok\ to\ vote\ for\ c$ message to all computers (including itself).

Every computer that receives $ok\ to\ vote\ for\ c$ messages from $2f+1$ computers votes for v and sends a message to all computers saying it has.

A good computer will send only one $ok\ to\ vote\ for\ c$ message in any ballot

Paxos With Proofs

L and each other computer that receives the $vote\ for\ c$ message (with proof) sends an $ok\ to\ vote\ for\ c$ message to all computers (including itself).

Every computer that receives $ok\ to\ vote\ for\ c$ messages from $2f+1$ computers votes for v and sends a message to all computers saying it has.

A good computer will send only one $ok\ to\ vote\ for\ c$ message in any ballot; with $2f+1$ good computers, it's impossible for two good computers to vote for different commands.

Paxos With Proofs

L and each other computer that receives the $vote\ for\ c$ message (with proof) sends an $ok\ to\ vote\ for\ c$ message to all computers (including itself).

Every computer that receives $ok\ to\ vote\ for\ c$ messages from $2f+1$ computers votes for v and sends a message to all computers saying it has.

Every computer that learns $2f+1$ computers voted for c knows that c was chosen.

Paxos With Proofs

L and each other computer that receives the $vote\ for\ c$ message (with proof) sends an $ok\ to\ vote\ for\ c$ message to all computers (including itself).

Every computer that receives $ok\ to\ vote\ for\ c$ messages from $2f + 1$ computers votes for v and sends a message to all computers saying it has.

Every computer that learns $2f + 1$ computers voted for c knows that c was chosen.

Because $f + 1$ are good computers executing Paxos, and a command is chosen in Paxos if $f + 1$ computers vote for it.

The Fine Print

This doesn’t quite work.

To make it work, we need to Byzantize a variant of the Paxos algorithm that differs slightly from the standard algorithm that I showed you.

Handling Malicious Leaders

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen, despite f malicious computers—including leaders.

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen, despite f malicious computers—including leaders.

Malicious computers can keep a value from being chosen.

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen, despite f malicious computers—including leaders.

Malicious computers can keep a value from being chosen.

A malicious leader can do nothing.

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen, despite f malicious computers—including leaders.

Malicious computers can keep a value from being chosen.

A malicious leader can do nothing.

A malicious computer can pretend it's the leader and keep starting new ballots, preventing the real leader's ballots from succeeding.

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen, despite f malicious computers—including leaders.

Malicious computers can keep a value from being chosen.

A malicious leader can do nothing.

A malicious computer can pretend it's the leader and keep starting new ballots, preventing the real leader's ballots from succeeding.

Castro-Liskov solution:

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen, despite f malicious computers—including leaders.

Malicious computers can keep a value from being chosen.

A malicious leader can do nothing.

A malicious computer can pretend it's the leader and keep starting new ballots, preventing the real leader's ballots from succeeding.

Castro-Liskov solution:

– Select leader by rotating through computers in a fixed order.

Handling Malicious Leaders

Adding proofs and the $ok\ to\ vote\ for\ c$ messages prevents good computers from disagreeing on what command is chosen, despite f malicious computers—including leaders.

Malicious computers can keep a value from being chosen.

A malicious leader can do nothing.

A malicious computer can pretend it's the leader and keep starting new ballots, preventing the real leader's ballots from succeeding.

Castro-Liskov solution:

– Select leader by rotating through computers in a fixed order.

– Use timeouts to switch to next leader if command not chosen.

Another Approach

Another Approach

Synchronous algorithms to fly an airplane don't need a leader.

Another Approach

Synchronous algorithms to fly an airplane don't need a leader.

They can't run a business because they fail if messages are lost or delivered too late.

Another Approach

Synchronous algorithms to fly an airplane don't need a leader.

They can't run a business because they fail if messages are lost or delivered too late.

To run a business, only need to choose a command when lucky.

Another Approach

Synchronous algorithms to fly an airplane don't need a leader.

They can't run a business because they fail if messages are lost or delivered too late.

To run a business, only need to choose a command when lucky.

Let "being lucky" mean messages are *not* lost or delivered late.

Use a synchronous algorithm to implement a single *virtual* leader.

Use a synchronous algorithm to implement a single *virtual* leader.

When system is lucky, the algorithm works, the virtual leader is good, and Byzantine Paxos chooses a command.

Use a synchronous algorithm to implement a single *virtual* leader.

When system is lucky, the algorithm works, the virtual leader is good, and Byzantine Paxos chooses a command.

When system is unlucky, the virtual leader may do nothing or be malicious

Use a synchronous algorithm to implement a single *virtual* leader.

When system is lucky, the algorithm works, the virtual leader is good, and Byzantine Paxos chooses a command.

When system is unlucky, the virtual leader may do nothing or be malicious, but this cannot cause disagreement.

Use a synchronous algorithm to implement a single *virtual* leader.

When system is lucky, the algorithm works, the virtual leader is good, and Byzantine Paxos chooses a command.

When system is unlucky, the virtual leader may do nothing or be malicious, but this cannot cause disagreement.

Byzantine Paxos tolerates a malicious leader.

Use a synchronous algorithm to implement a single *virtual* leader.

When system is lucky, the algorithm works, the virtual leader is good, and Byzantine Paxos chooses a command.

When system is unlucky, the virtual leader may do nothing or be malicious, but this cannot cause disagreement.

Eventually, the system will be lucky, the virtual leader will be good, and a command will be chosen.

Eliminating Digital Signatures

Digital signatures are used to send a proof.

Eliminating Digital Signatures

Digital signatures are used to send a proof.

They require too much computation for running some businesses.

Eliminating Digital Signatures

Digital signatures are used to send a proof.

They require too much computation for running some businesses.

I know two other methods of sending a proof.

Eliminating Digital Signatures

Digital signatures are used to send a proof.

They require too much computation for running some businesses.

I know two other methods of sending a proof.

I can describe them at the end if you're interested.

What Good is Byzantine Paxos?

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

Proposed by Castro and Liskov.

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

Proposed by Castro and Liskov.

How do you prevent a hacker from taking over all the machines?

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

Proposed by Castro and Liskov.

How do you prevent a hacker from taking over all the machines?

Need to use different operating systems on different computers.

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

- Proposed by Castro and Liskov.
- How do you prevent a hacker from taking over all the machines?
- Need to use different operating systems on different computers.

A system run by different organizations that don't trust each other?

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

- Proposed by Castro and Liskov.
- How do you prevent a hacker from taking over all the machines?
- Need to use different operating systems on different computers.

A system run by different organizations that don't trust each other?

A Napster-like service run on users' computers?

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

- Proposed by Castro and Liskov.
- How do you prevent a hacker from taking over all the machines?
- Need to use different operating systems on different computers.

A system run by different organizations that don't trust each other?

A Napster-like service run on users' computers?

- How do you avoid choosing too many malicious users' computers?

What Good is Byzantine Paxos?

Protect against hackers taking over some machines in a data center?

- Proposed by Castro and Liskov.
- How do you prevent a hacker from taking over all the machines?
- Need to use different operating systems on different computers.

A system run by different organizations that don't trust each other?

A Napster-like service run on users' computers?

- How do you avoid choosing too many malicious users' computers?
- Is random choice from a large number of mostly non-malicious users good enough?

I don’t know if anyone will ever run a Byzantine business.

I don’t know if anyone will ever run a Byzantine business.

Algorithms to do it may lead to new techniques for tolerating faults

I don't know if anyone will ever run a Byzantine business.

Algorithms to do it may lead to new techniques for tolerating faults and foiling hackers.

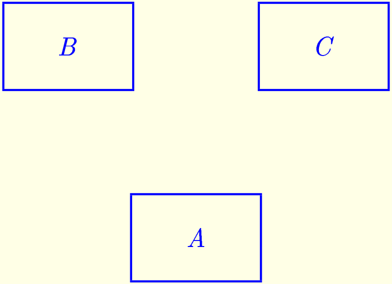
Thank you.

Addendum

Two Methods of Eliminating Digital Signatures

Method 1 (Castro and Liskov)

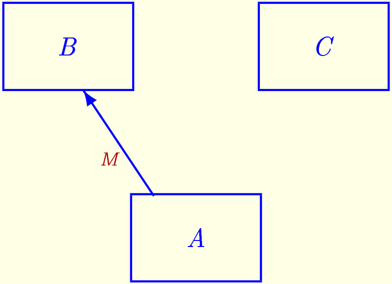
How digital signatures are used.



Method 1 (Castro and Liskov)

How digital signatures are used.

Message M is signed by A.

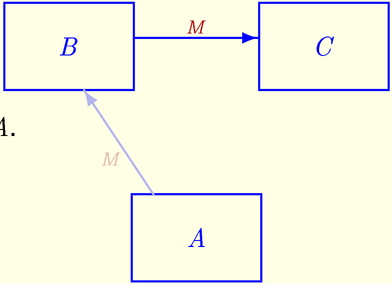


Method 1 (Castro and Liskov)

How digital signatures are used.

Message M is signed by A.

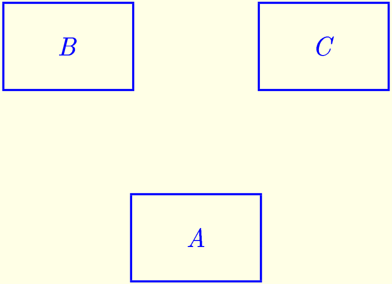
B knows C will know M sent by A.



Method 1 (Castro and Liskov)

Method 1 (Castro and Liskov)

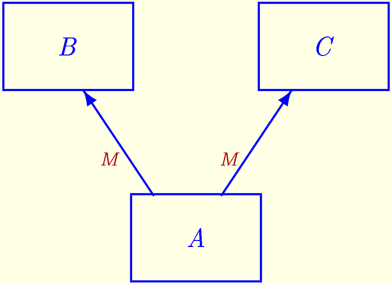
Without digital signatures.



Method 1 (Castro and Liskov)

Without digital signatures.

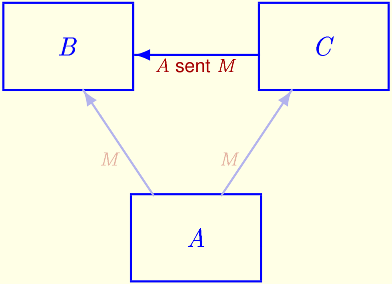
Message M unsigned.



Method 1 (Castro and Liskov)

Without digital signatures.

Message M unsigned.

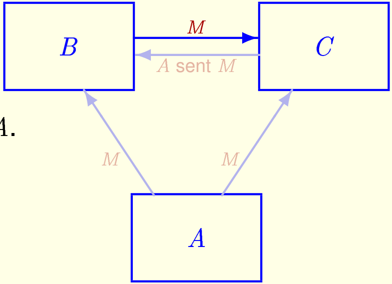


Method 1 (Castro and Liskov)

Without digital signatures.

Message M unsigned.

B knows C will know M sent by A.



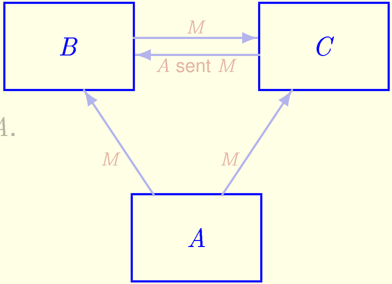
Method 1 (Castro and Liskov)

Without digital signatures.

Message M unsigned.

B knows C will know M sent by A.

Assumes a computer knows the immediate sender of a message.



Method 2 Vectors of Message Authenticators

Method 2 Vectors of Message Authenticators

A message authenticator M_{AB} proves to B that A sent M.

Method 2 Vectors of Message Authenticators

A message authenticator M_{AB} proves to B that A sent M.

Meaningless to any other computer.

Method 2 Vectors of Message Authenticators

A message authenticator M_{AB} proves to B that A sent M.

Meaningless to any other computer.

Requires much less computation than a digital signature.

Method 2 Vectors of Message Authenticators

A message authenticator M_{AB} proves to B that A sent M.

Meaningless to any other computer.

Requires much less computation than a digital signature.

Let M_A be the vector of authenticators $\langle M_{AB},\, M_{AC}, \ldots \rangle$, one for every other computer.

Method 2 Vectors of Message Authenticators

A message authenticator M_{AB} proves to B that A sent M.

Meaningless to any other computer.

Requires much less computation than a digital signature.

Let M_A be the vector of authenticators $\langle M_{AB}, M_{AC}, \ldots \rangle$, one for every other computer.

M_A is not as good as a digitally signed message M.

Method 2 Vectors of Message Authenticators

A message authenticator M_{AB} proves to B that A sent M.

Meaningless to any other computer.

Requires much less computation than a digital signature.

Let M_A be the vector of authenticators $\langle M_{AB}, M_{AC}, \ldots \rangle$, one for every other computer.

M_A is not as good as a digitally signed message M.

It proves to B that A sent M

Method 2 Vectors of Message Authenticators

A message authenticator M_{AB} proves to B that A sent M.

Meaningless to any other computer.

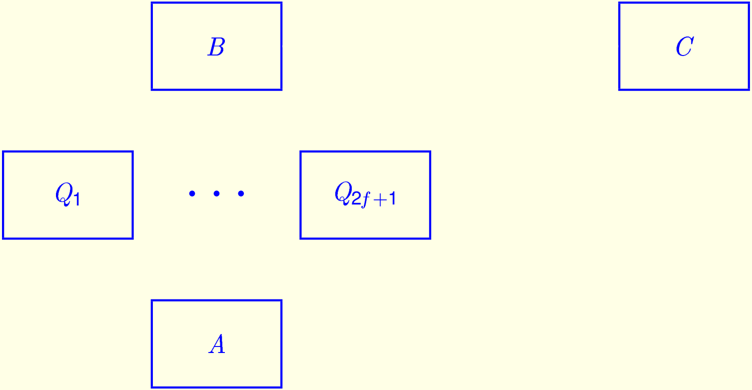
Requires much less computation than a digital signature.

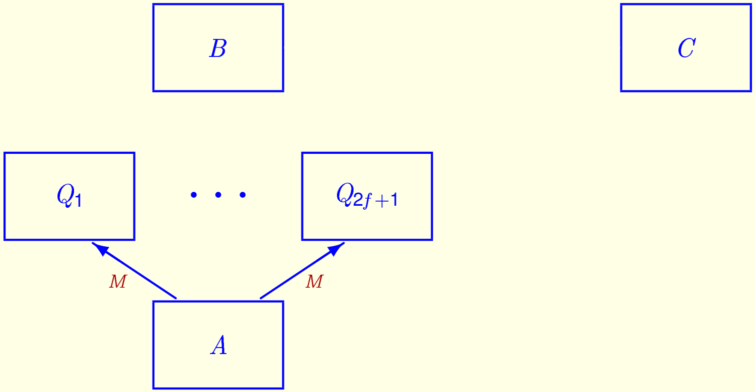
Let M_A be the vector of authenticators $\langle M_{AB}, M_{AC}, \ldots \rangle$, one for every other computer.

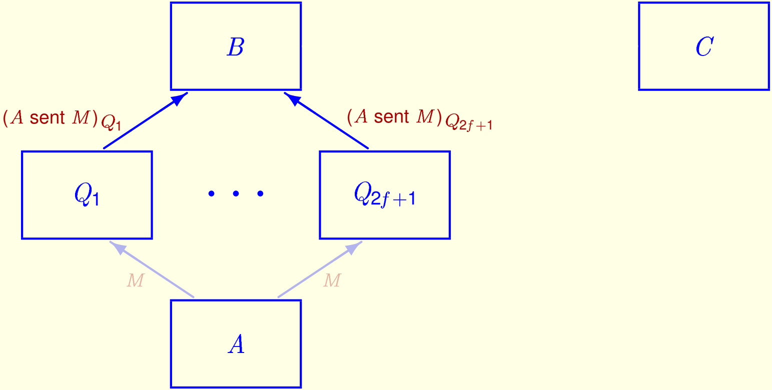
M_A is not as good as a digitally signed message M.

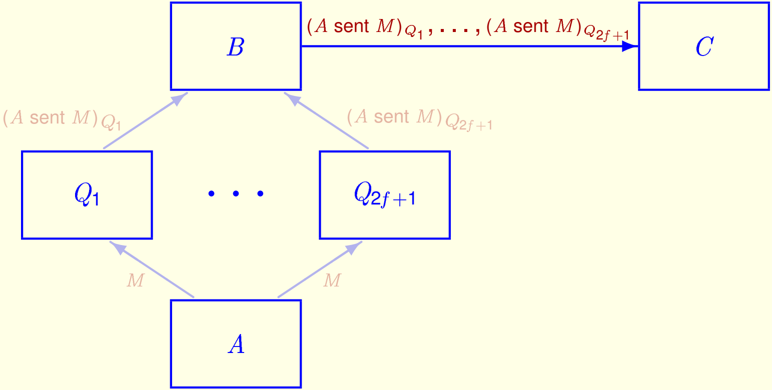
It proves to B that A sent M, but B doesn't know if it proves that to C because A could be malicious and the M_{AC} entry in the vector could be garbage.

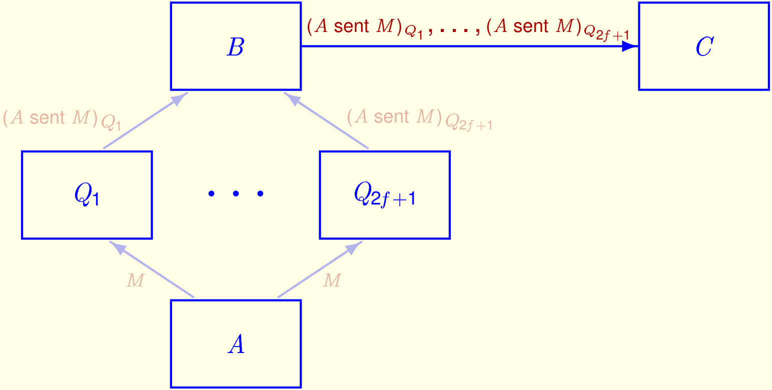
But vectors of authenticators are good enough.


B
C
Q_1
$\cdots$
Q_{2f+1}
A

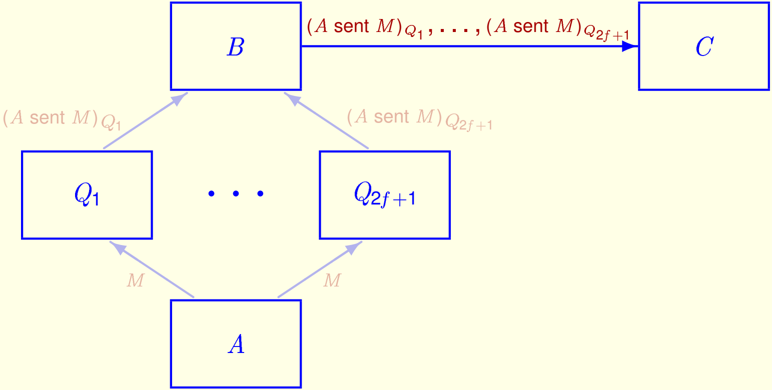

B
C
Q1
. . .
Q2f+1
M
M
A


B
C
(A sent M)$_{Q_1}$
(A sent M)$_{Q_{2f+1}}$
Q_1
. . .
Q_{2f+1}
M
M
A


B
$(A \text{ sent } M)_{Q_1}, \ldots, (A \text{ sent } M)_{Q_{2f+1}}$
C
$(A \text{ sent } M)_{Q_1}$
$(A \text{ sent } M)_{Q_{2f+1}}$
Q_1
$\cdots$
Q_{2f+1}
M
M
A

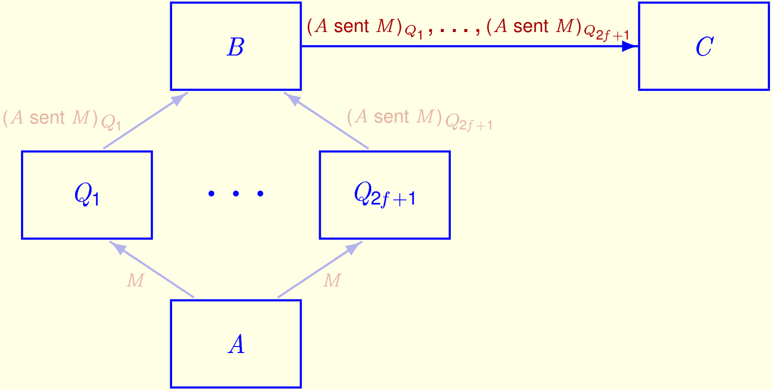


B knows C will know M sent by A



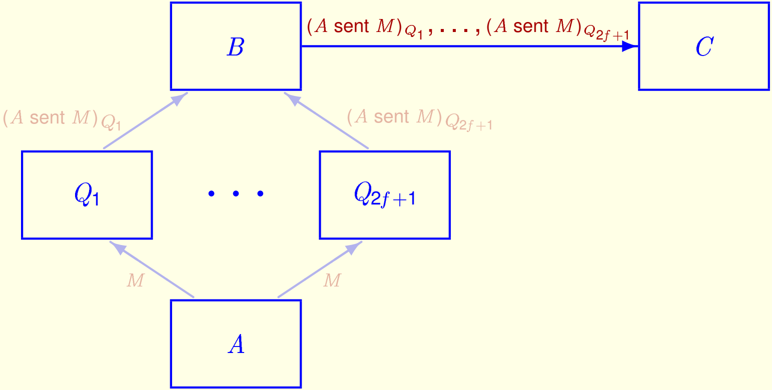
B knows C will know M sent by A

Because at most f of the Q_i are malicious



B knows C will know M sent by A

Because at most f of the Q_i are malicious, so C will receive properly authenticated messages from $f + 1$ computers saying A sent M



B knows C will know M sent by A

Because at most f of the Q_i are malicious, so C will receive properly authenticated messages from $f + 1$ computers saying A sent M, and C knows that at least one of those computers must be good.