Where should RC be implemented?

- In hardware
  - sensitive to architecture changes
- At the OS level
  - state transitions hard to track and coordinate
- At the application level
  - requires sophisticated application programmers

Hypervisor-based Fault-tolerance

- Implement RC at a virtual machine running on the same instruction-set as underlying hardware
- Undetectable by higher layers of software
- One of the great come-backs in systems research!
  - CP-67 for IBM 369 [1970]
  - Xen [SOSP 2003], VMware

The Hypervisor as a State Machine

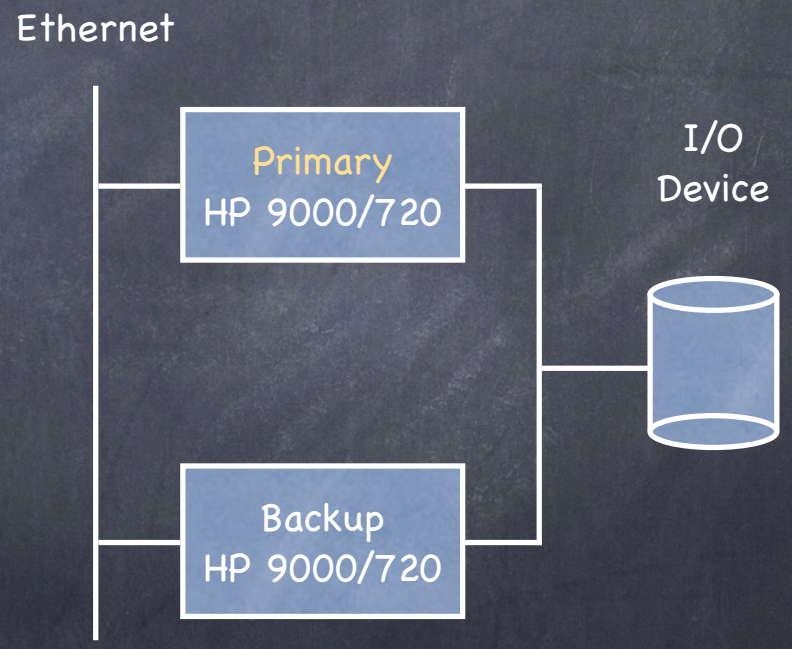
- Two types of commands
  - virtual-machine instructions
  - virtual-machine interrupts (with DMA input)
- State transition must be deterministic
  - ...but some VM instructions are not (e.g. time-of-day)
  - interrupts must be delivered at the same point in command sequence

The Architecture

- Good-ol' Primary-Backup
- Primary makes all non-deterministic choices

I/O Accessibility Assumption

Primary and backup have access to same I/O operations



Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions

Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions
- Environment Instruction Assumption

  Hypervisor captures all environment instructions, simulates them, and ensures they have the same effect at all state machines

Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions
- Environment Instruction Assumption
- VM interrupts must be delivered at same point in instruction sequence at all replicas

Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions
- Environment Instruction Assumption
- VM interrupts must be delivered at same point in instruction sequence at all replicas
- Instruction Stream Interrupt Assumption

  Hypervisor can be invoked at specific point in the instruction stream

Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions
- Environment Instruction Assumption
- VM interrupts must be delivered at same point in instruction sequence at all replicas
- Instruction Stream Interrupt Assumption
  - implemented via recovery register
  - interrupts at backup are ignored

The failure-free protocol

P0: On processing environment instruction i at pc, HV of primary p :

sends $[e_p, pc, Val_i]$ to backup b
waits for ack

P1: If p'HV receives Int from its VM:

p buffers Int until epoch ends

P2: If epoch ends at p:

p sends to b all buffered Int in e_p
p waits for ack
p delivers all VM Int in e_p
$e_p := e_p + 1$
p starts e_p

P3: If b'HV processes environment instruction i at pc

b waits for $[e_b, pc, Val_i]$ from p
returns Val_i

If b receives $[E, pc, Val]$ from p :

b sends ack to p
b buffers Val for delivery at E, pc

P4: If b'HV receives Int from its VM

b ignores Int

P5: If epoch ends at b:

b waits from p for interrupts for e_b
b sends ack to p
b delivers all VM Int buffered in e_b
$e_b := e_b + 1$
b starts e_b

If the primary fails...

P6: If b receives a failure notification instead of $[e_b, pc, Val_i]$, b executes i

If in P5 b receives failure notification instead of Int (e_b is a failover epoch):

$e_b := e_b + 1$

b is promoted primary for epoch e_b

If p crashes before sending Int to b, Int is lost!

Failures and the environment

- No exactly-once guarantee on outputs
- On primary failure, avoid input inconsistencies
  - time must increase monotonically
    - at epoch boundaries, primary informs backup of value of its clock
  - interrupts must be delivered as a fault-free processor would
    - but interrupts can be lost...
    - weaken constraints on I/O interrupts

On I/O device drivers

IO1: If an I/O instruction is executed and the I/O operation performed, the issuing processor delivers a completion interrupt, unless it fails. If the processor fails, the I/O device continues as if the interrupt had been delivered.

IO2: An I/O device may cause an uncertain interrupt (indicating the operation has been terminated) to be delivered by the processor issuing the I/O instruction. The instruction could have been in progress, completed, or not even started.

On an uncertain interrupt, the device driver reissues the corresponding I/O instruction–not all devices though are idempotent or testable

Backup promotion and uncertain interrupts

P7: The backup's VM generates an uncertain interrupt for each outstanding I/O operation right before the backup is promoted primary (at the end of the failover epoch)

RC in the Hypervisor

- Nondeterministic ordinary instructions (Surprise!)

RC in the Hypervisor

- Nondeterministic ordinary instructions (Surprise!)
  - TLB replacement policy non-deterministic
  - TLB misses handled by software
  - Primary and backup may execute a different number of instructions!

HV takes over TLB replacement

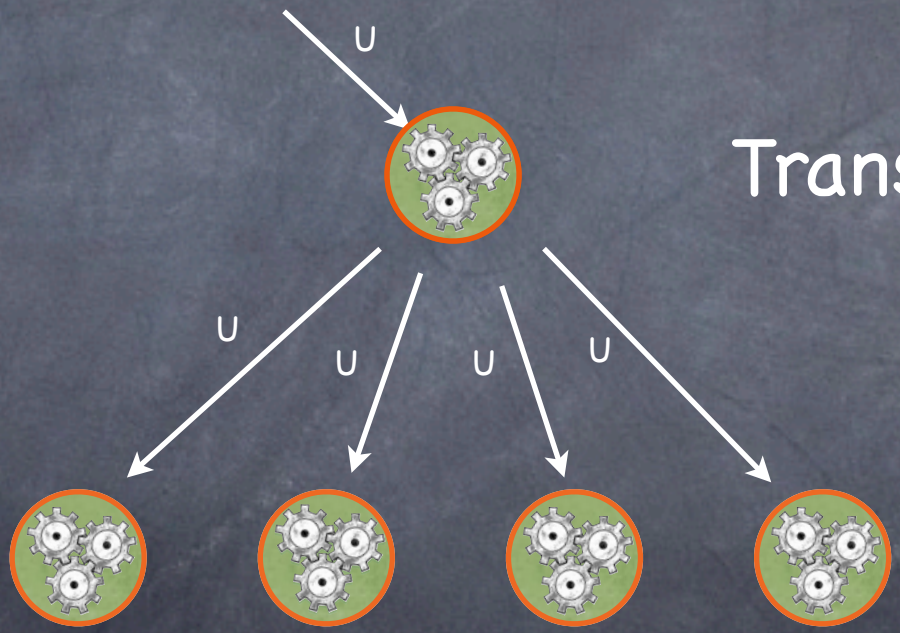
RC in the Hypervisor

- Nondeterministic ordinary instructions (Surprise!)
  - TLB replacement policy non-deterministic
  - TLB misses handled by software
  - Primary and backup may execute a different number of instructions!

HV takes over TLB replacement

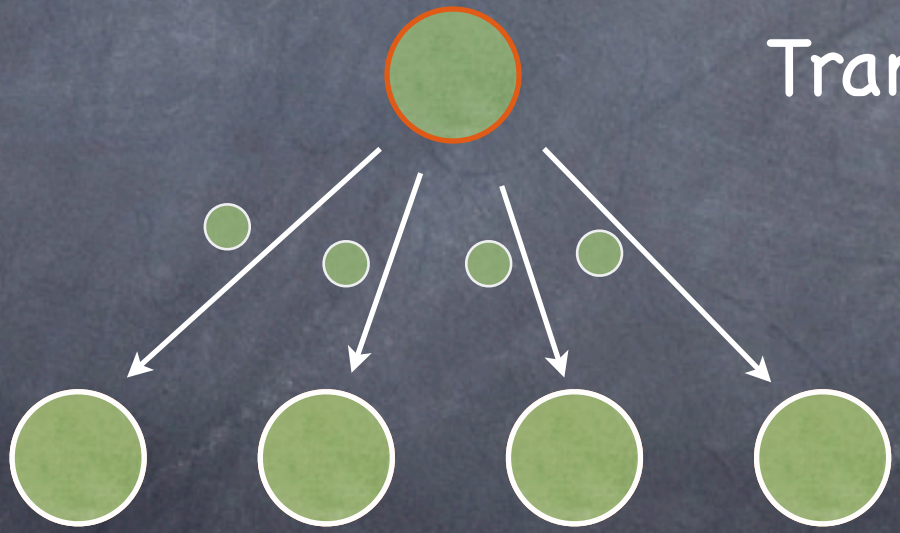
- Optimizations
  - p sends Int immediately
  - p blocks for acks only before output commit

Primary-backup: Updates

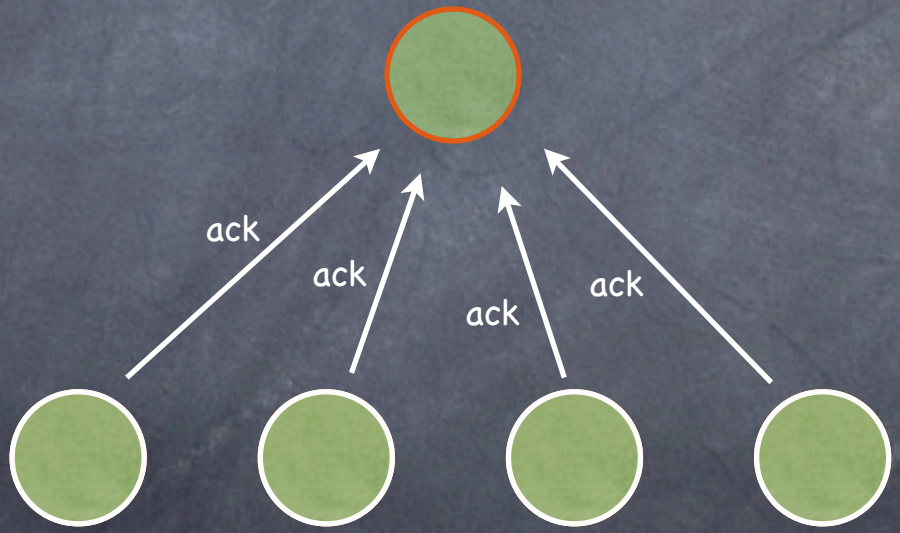


Transfer update

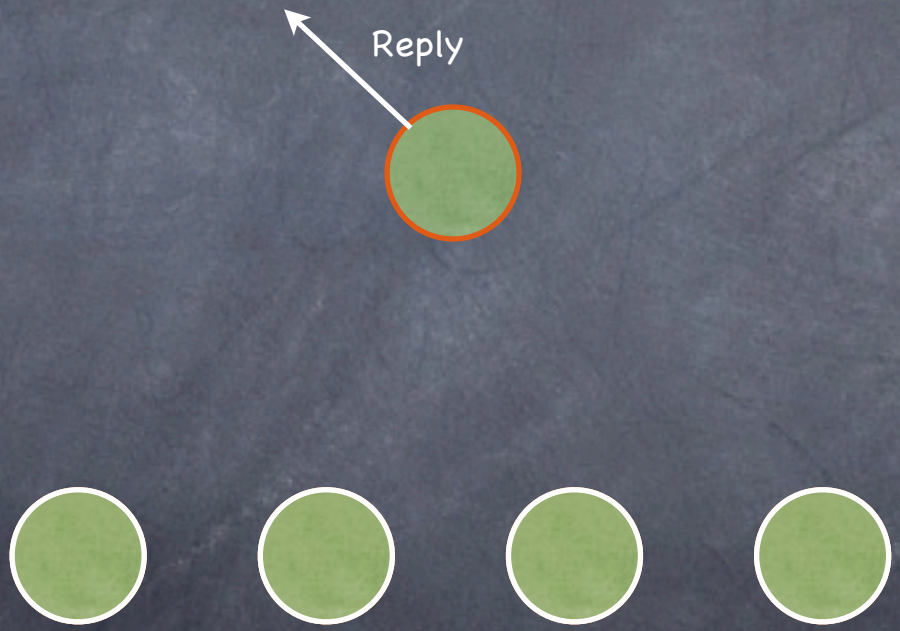
Primary-backup: Updates



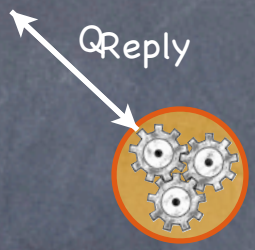
Primary-backup: Updates



Primary-backup: Updates

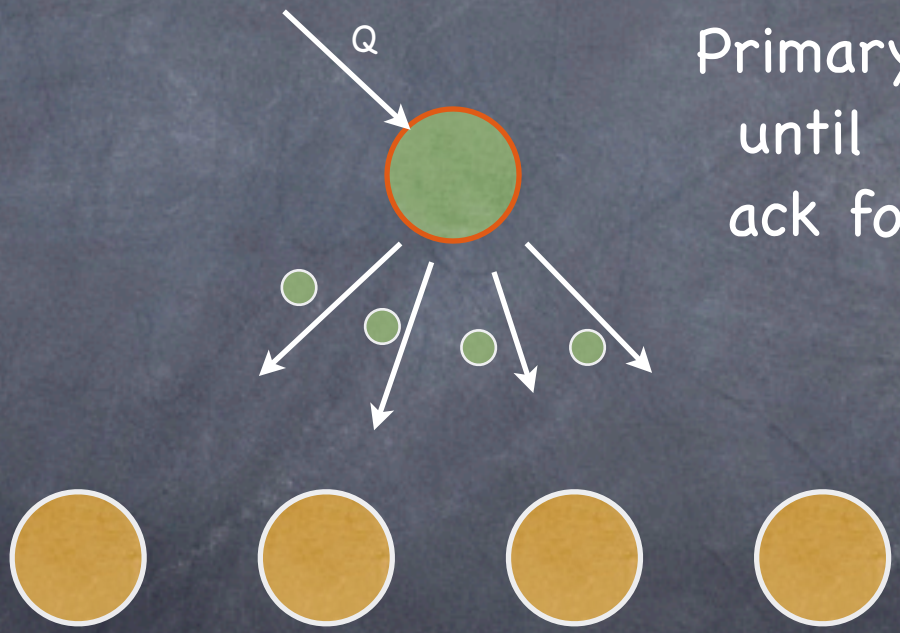


Primary-backup: Queries



However...

Primary-backup: Queries



Primary cannot respond until it has received ack for prior updates!

Chain replication

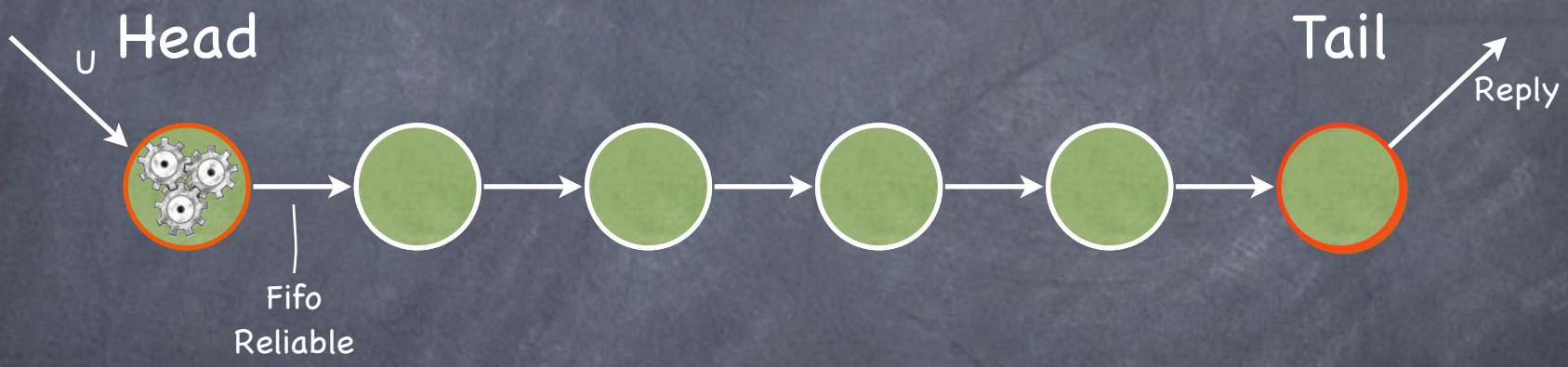
Van Renesse, Schneider, OSDI '04

Chain replication

Van Renesse, Schneider, OSDI '04

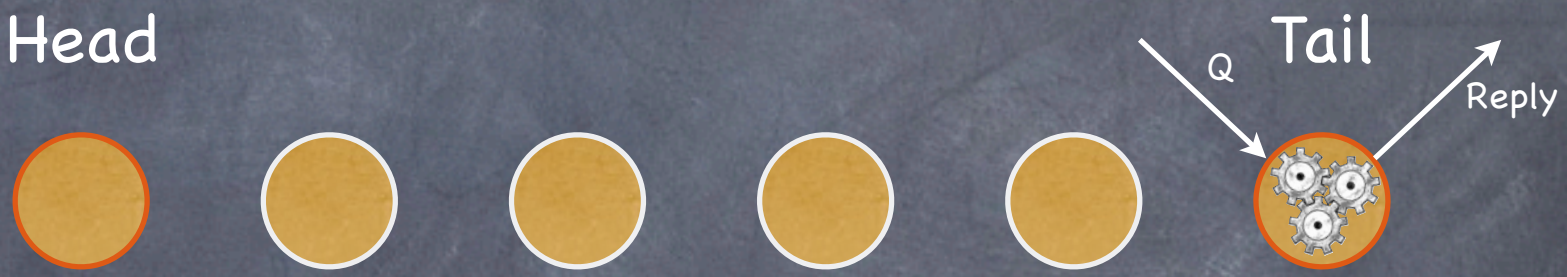
Chain replication: Updates

Van Renesse, Schneider, OSDI '04



Chain replication: Queries

Van Renesse, Schneider, OSDI '04



Furthermore...

Chain replication: Queries

Van Renesse, Schneider, OSDI '04

Tail can respond immediately,
without waiting for the new update

Some like it hot

- Hot Backups process information from the primary as soon as they receive it
- Cold Backups log information received from primary, and process it only if primary fails
- Rollback Recovery implements cold backups cheaply:
  - the primary logs directly to stable storage the information needed by backups
  - if the primary crashes, a newly initialized process is given content of logs—backups are generated "on demand"

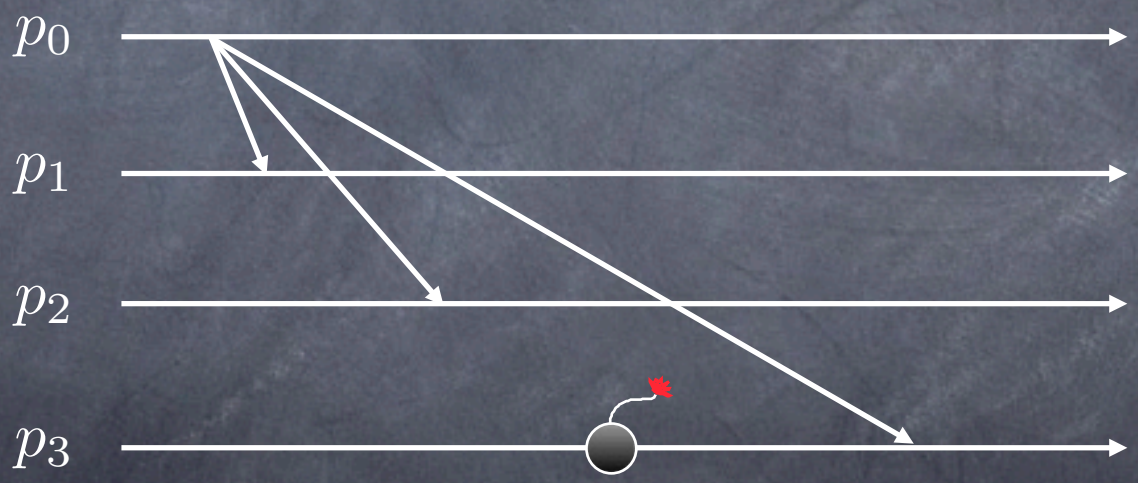
Consensus and Reliable Broadcast

Broadcast

- If a process sends a message m, then every process eventually delivers m

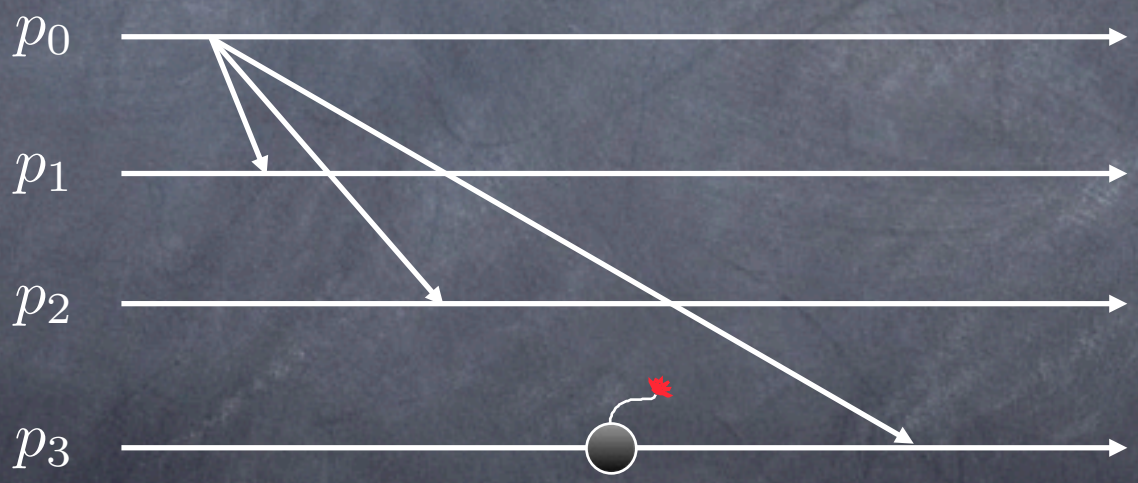
Broadcast

- If a process sends a message m, then every process eventually delivers m



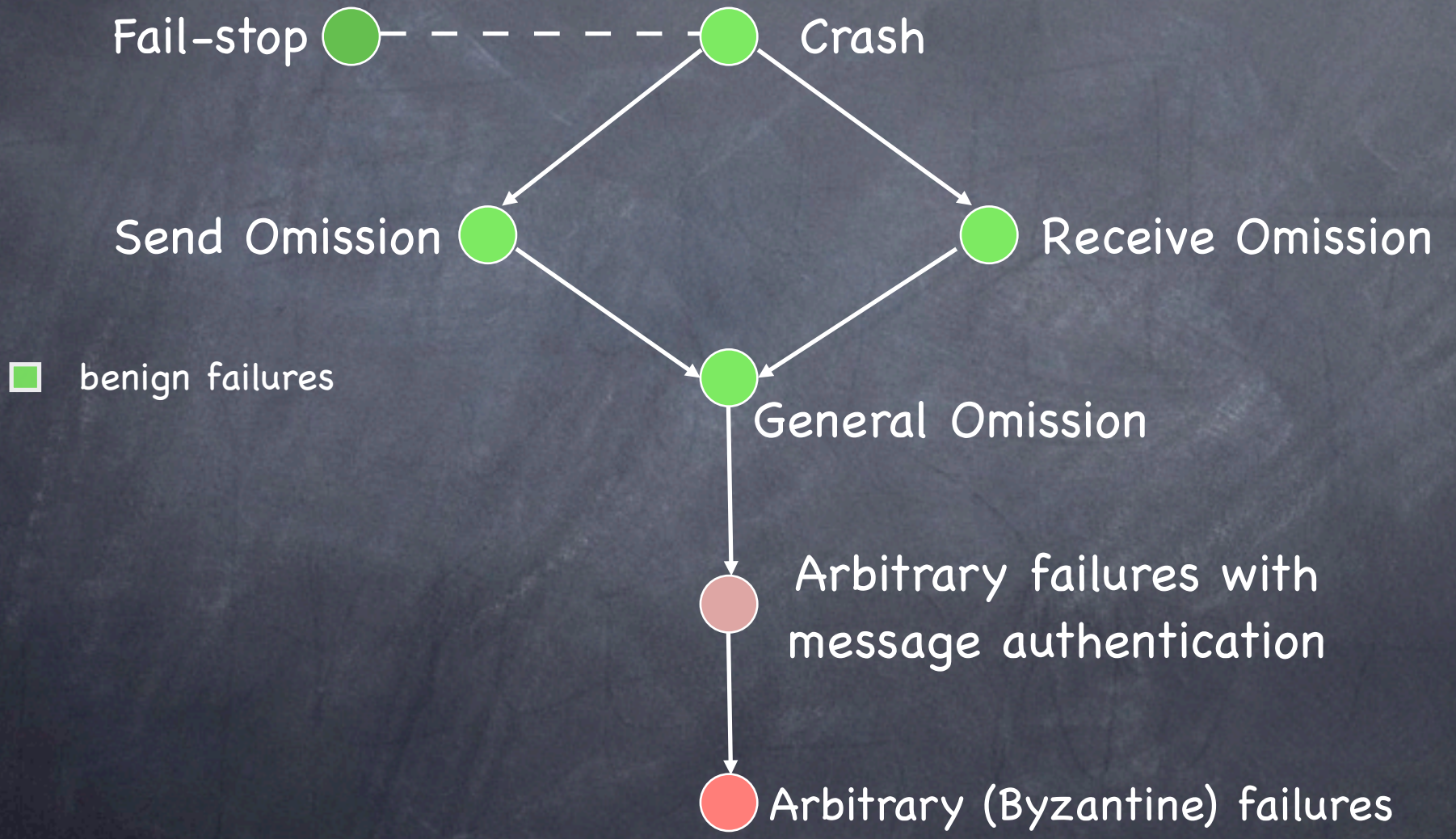
Broadcast

- If a process sends a message m, then every process eventually delivers m



- How can we adapt the spec for an environment where processes can fail? And what does "fail" mean?

A hierarchy of failure models



Reliable Broadcast

Validity	If the sender is correct and broadcasts a message m, then all correct processes eventually deliver m
Agreement	If a correct process delivers a message m, then all correct processes eventually deliver m
Integrity	Every correct process delivers at most one message, and if it delivers m, then some process must have broadcast m

Terminating Reliable Broadcast

Validity	If the sender is correct and broadcasts a message m, then all correct processes eventually deliver m
Agreement	If a correct process delivers a message m, then all correct processes eventually deliver m
Integrity	Every correct process delivers at most one message, and if it delivers $m \neq$ SF, then some process must have broadcast m
Termination	Every correct process eventually delivers some message

Consensus

Validity	If all processes that propose a value propose v, then all correct processes eventually decide v
Agreement	If a correct process decides v, then all correct processes eventually decide v
Integrity	Every correct process decides at most one value, and if it decides v, then some process must have proposed v
Termination	Every correct process eventually decides some value

Properties of send(m) and receive(m)

Benign failures:

Validity If p sends m to q, and p , q, and the link between them are correct, then q eventually receives m

Uniform* Integrity For any message m, q receives m at most once from p, and only if p sent m to q

* A property is uniform if it applies to both correct and faulty processes

Properties of send(m) and receive(m)

Arbitrary failures:

Integrity For any message m, if p and q are correct then q receives m at most once from p, and only if p sent m to q

Questions, Questions...

- Are these problems solvable at all?
- Can they be solved independent of the failure model?
- Does solvability depend on the ratio between faulty and correct processes?
- Does solvability depend on assumptions about the reliability of the network?
- Are the problems solvable in both synchronous and asynchronous systems?
- If a solution exists, how expensive is it?

Plan

- Benign Synchronous Systems
  - Consensus for synchronous systems with crash failures
  - Lower bound on the number of rounds
- Benign Asynchronous Systems
  - Impossibility of Consensus for crash failures
  - Failure detectors
  - PAXOS
- Byzantine (Synchronous and Asynchronous)
  - Reliable Broadcast for arbitrary failures
  - PBFT, Zyzzyva

Model

- Synchronous Message Passing
  - Execution is a sequence of rounds
  - In each round every process takes a step
    - sends messages to neighbors
    - receives messages sent in that round
    - changes its state
- Network is fully connected (an n-clique)
- No communication failures

A simple Consensus algorithm

Process p_i:

Initially $V = \{v_i\}$

To execute propose(v_i)

1: send $\{v_i\}$ to all

decide(x) occurs as follows:

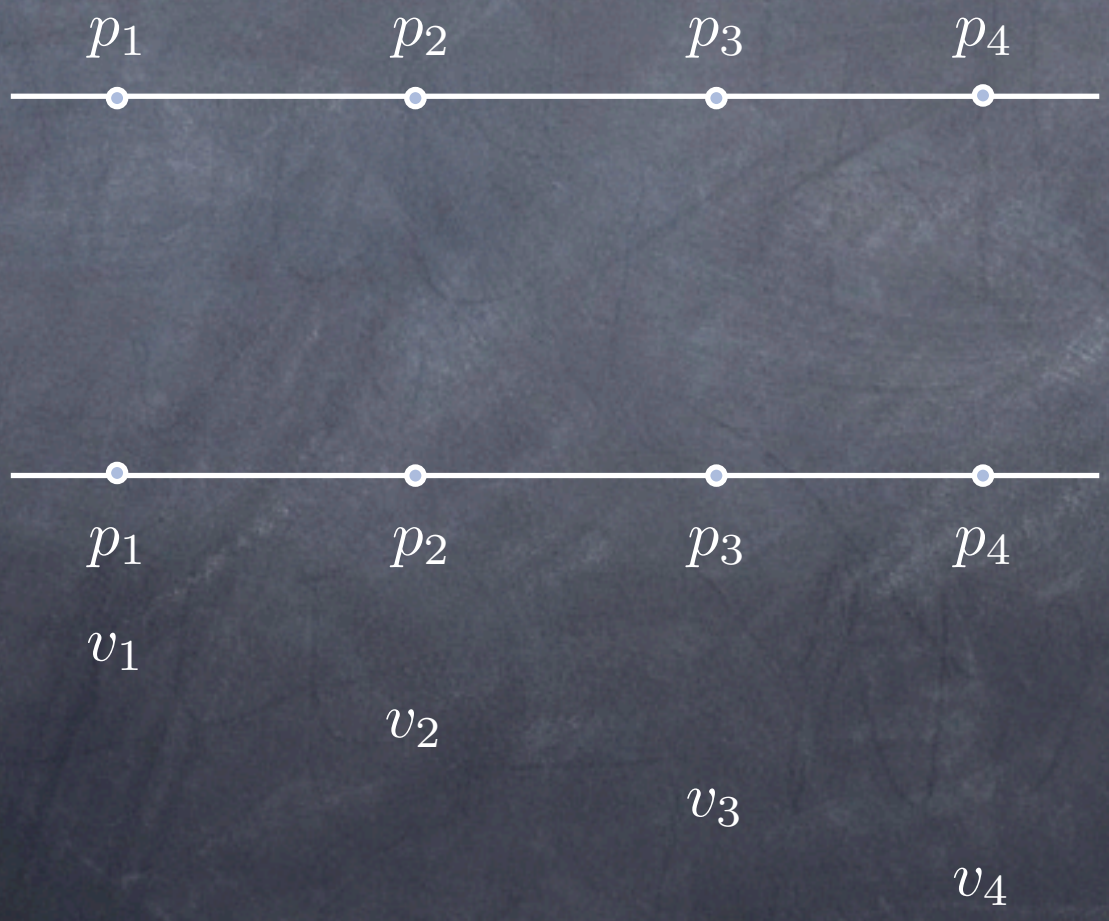
2: for all j, $0 \leq j \leq n-1$, $j \neq i$ do

3: receive S_j from p_j

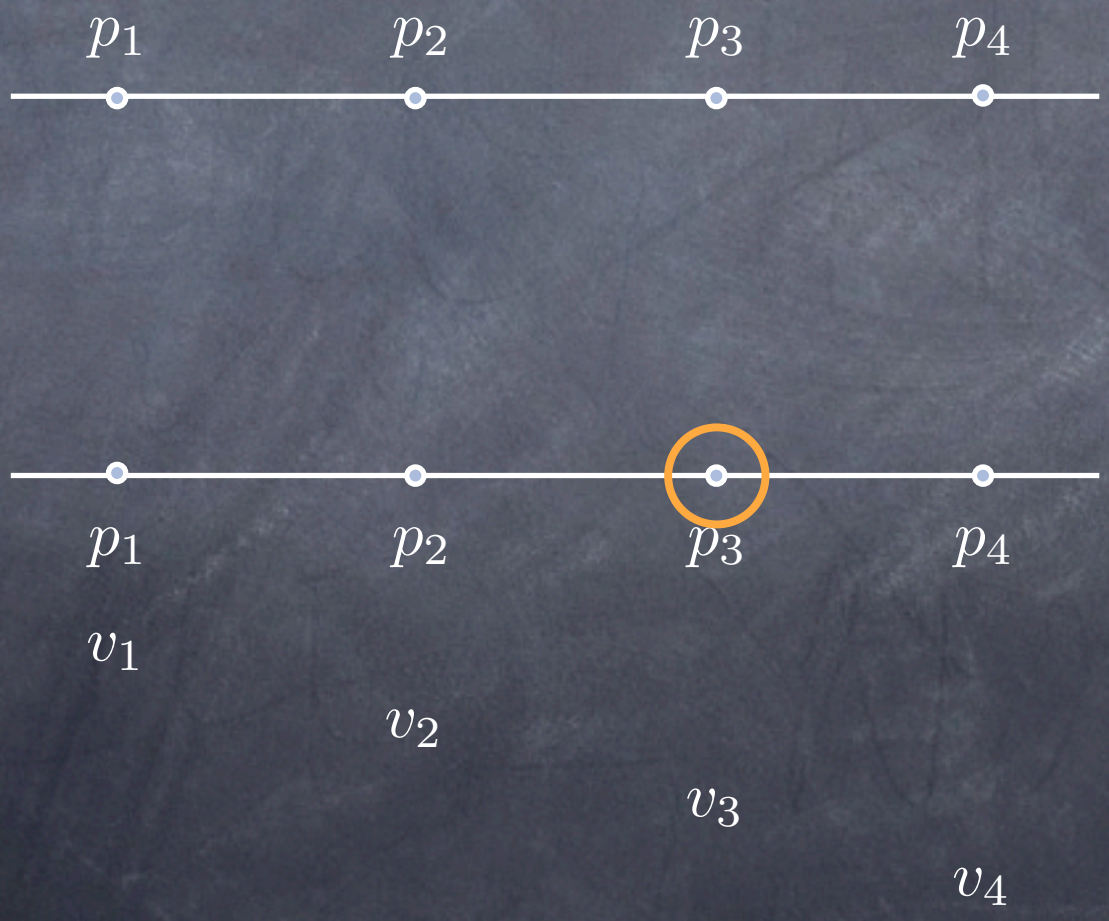
4: $V := V \cup S_j$

5: decide min(V)

An execution



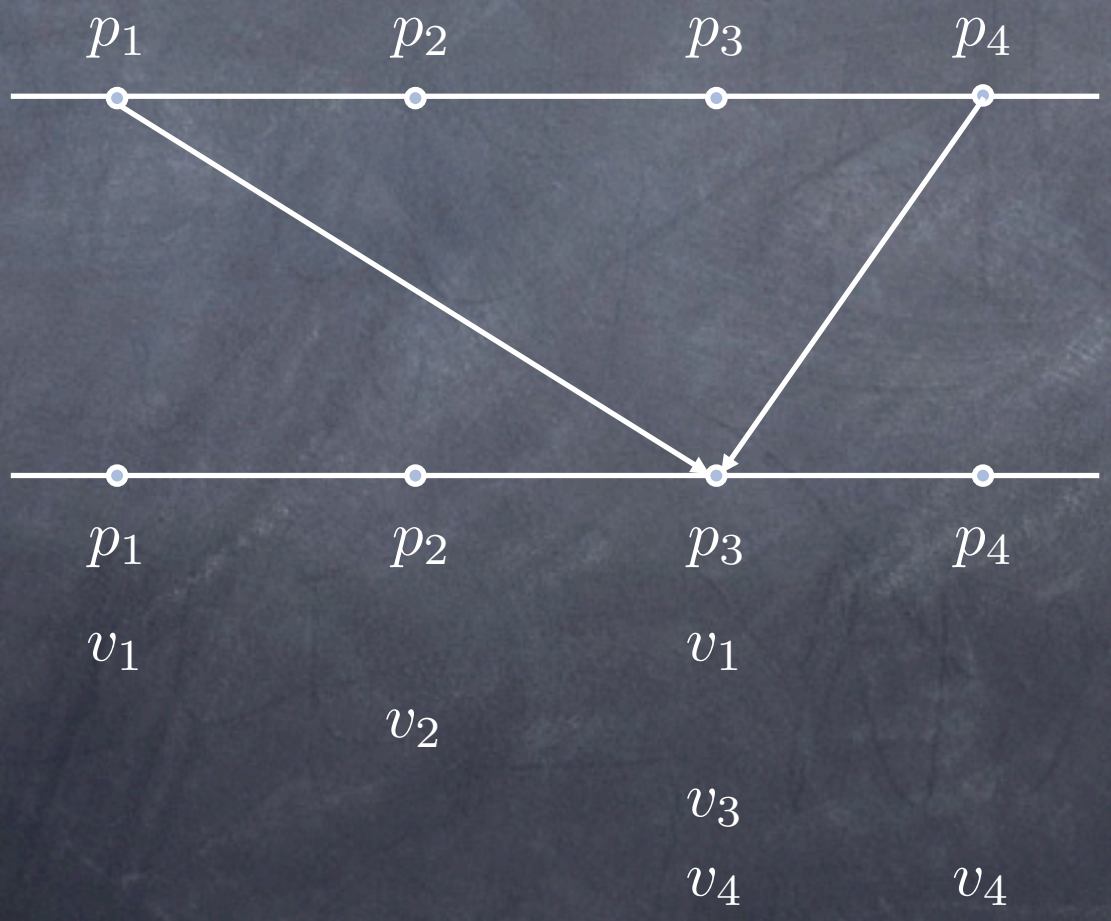
An execution



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

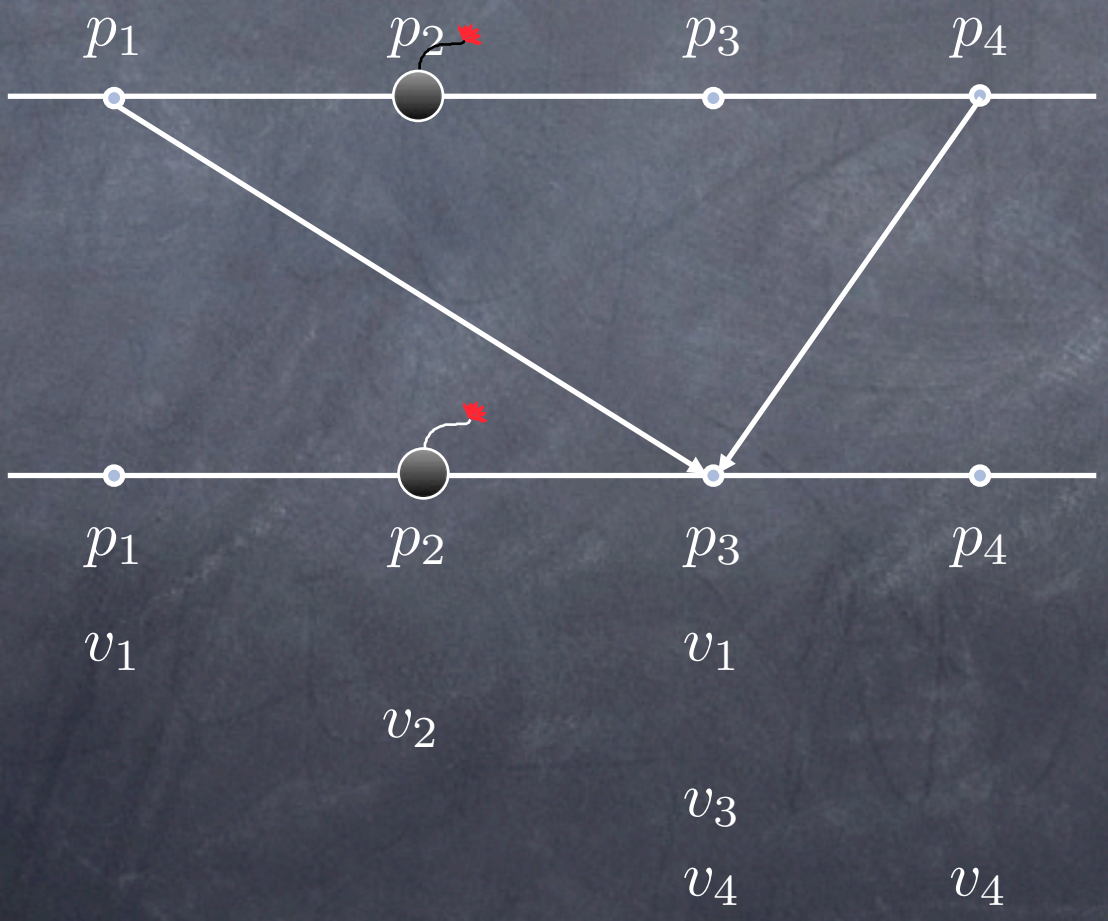
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

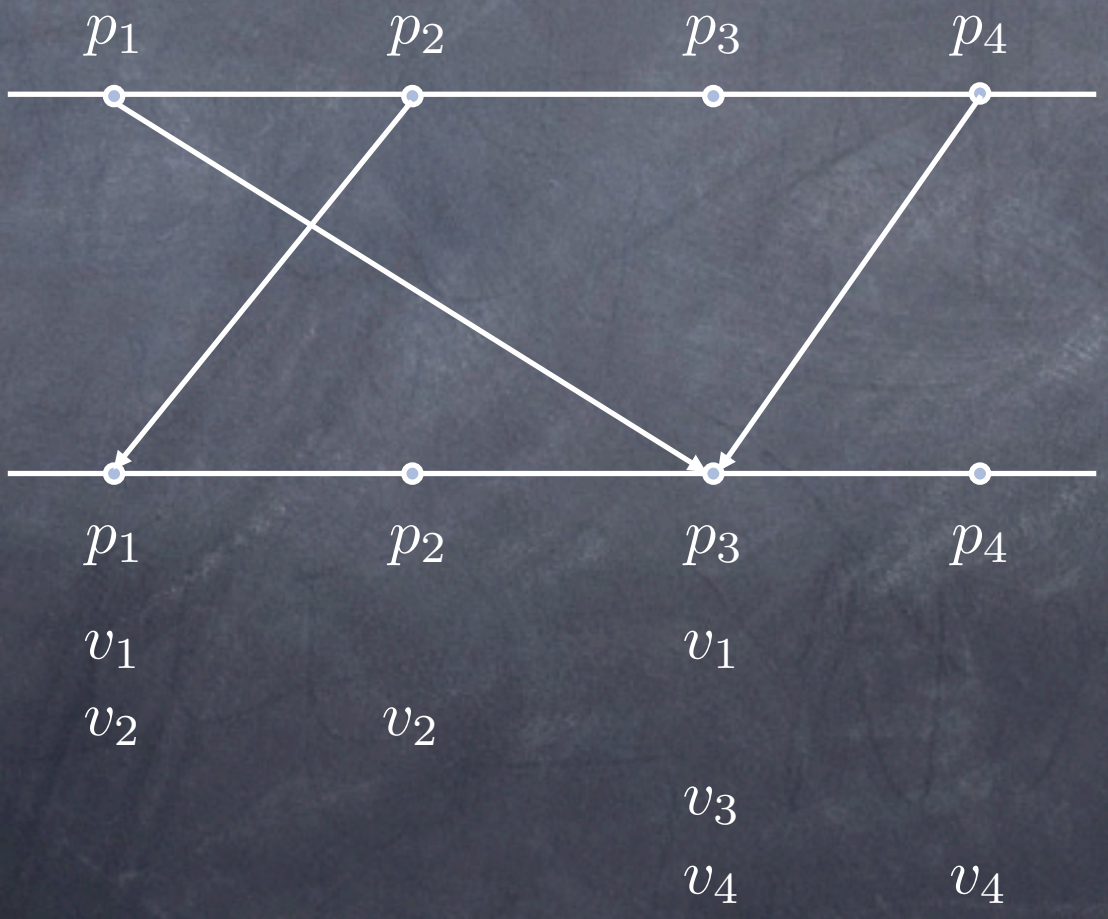
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

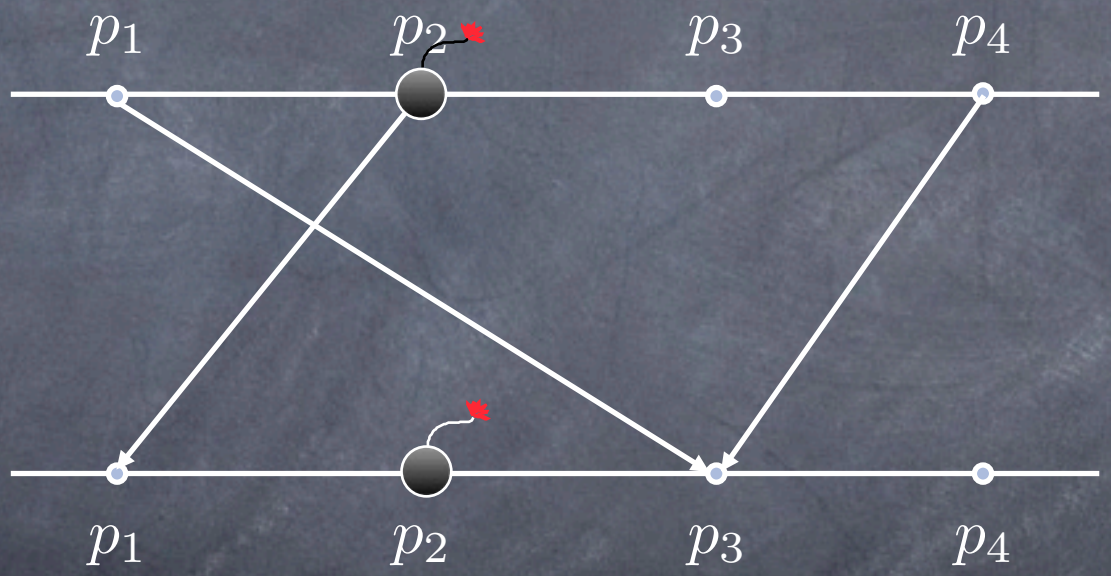
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

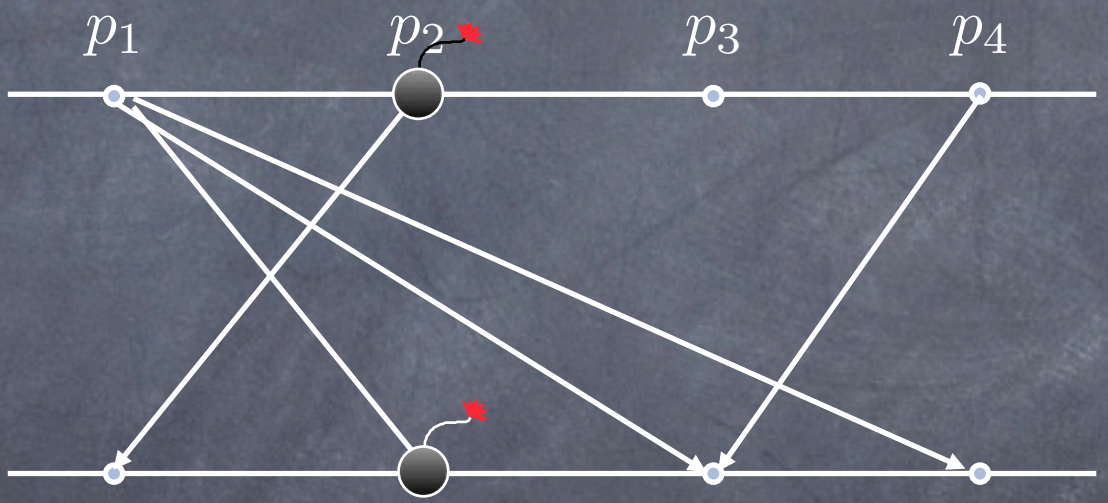
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

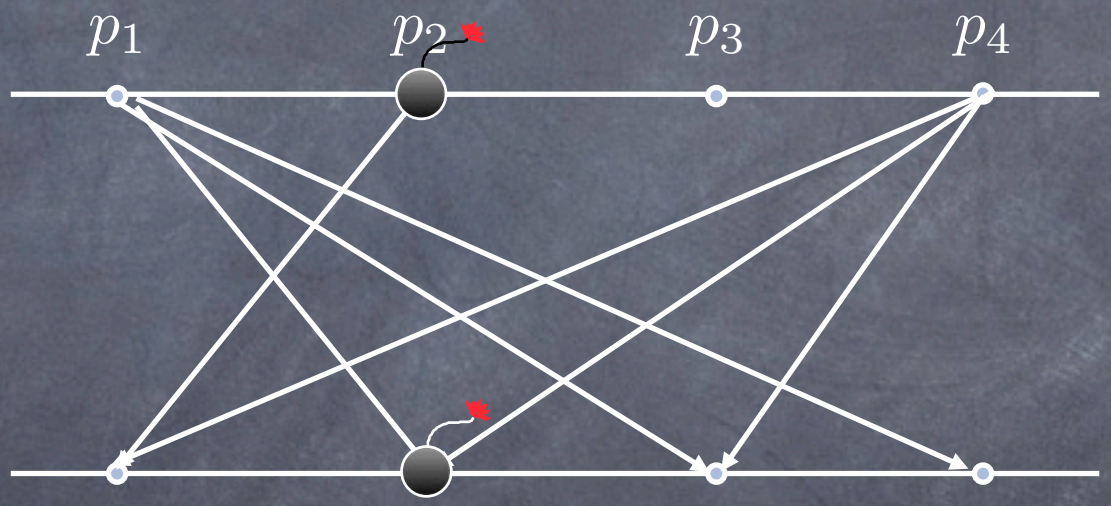
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

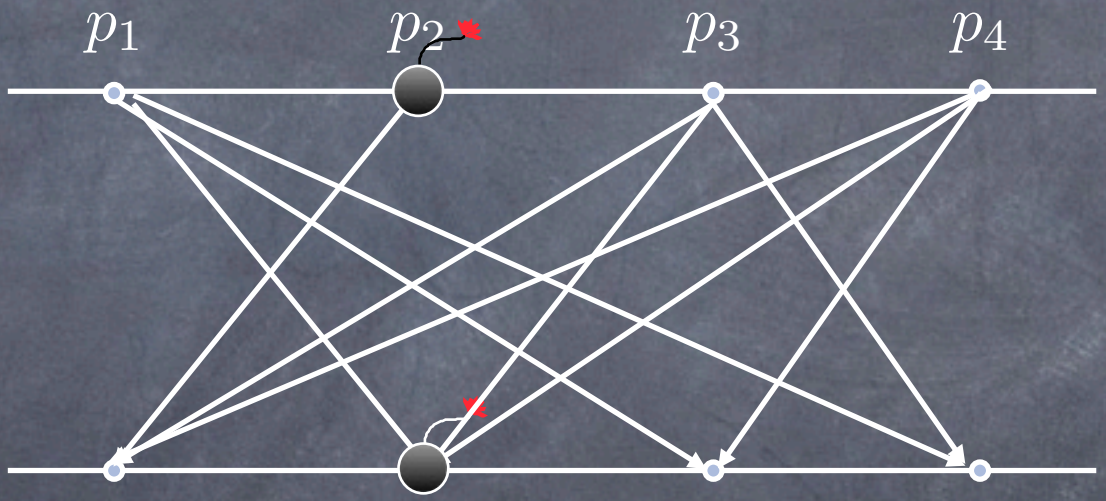
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

Can p_3 decide?



p_1	p_2	p_3	p_4
v_1	v_1	v_1	v_1
v_2	v_2		
v_3		v_3	v_3
v_4		v_4	v_4

Echoing values

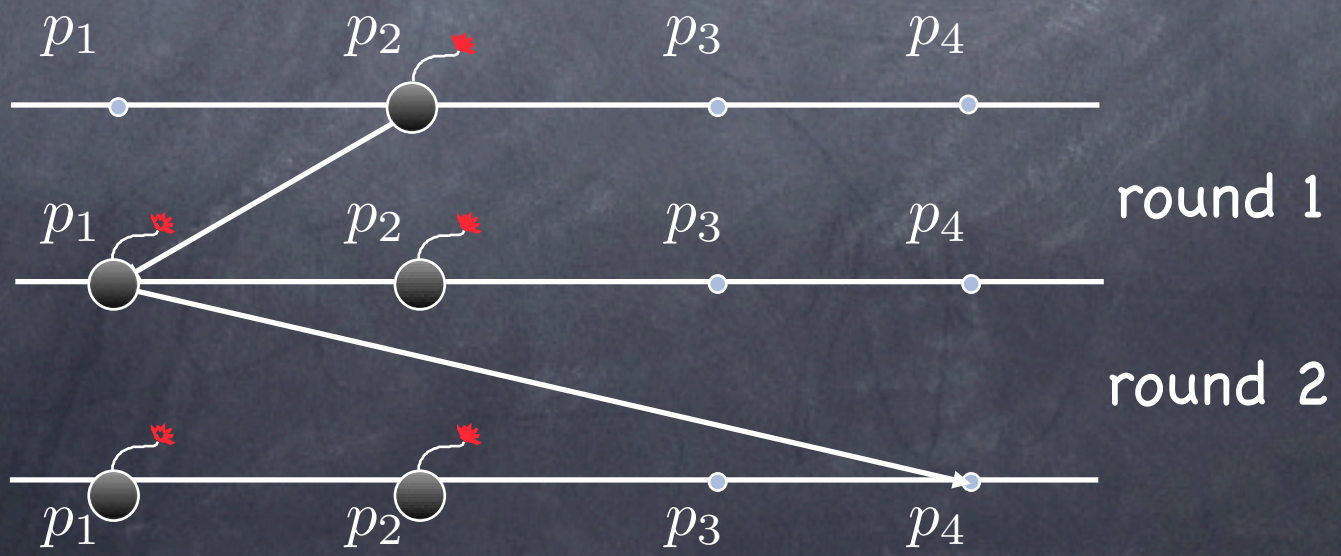
- A process that receives a proposal in round 1, relays it to others during round 2.

Echoing values

- A process that receives a proposal in round 1, relays it to others during round 2.
- Suppose p_3 hasn't heard from p_2 at the end of round 2. Can p_3 decide?

Echoing values

- A process that receives a proposal in round 1, relays it to others during round 2.
- Suppose p_3 hasn't heard from p_2 at the end of round 2. Can p_3 decide?



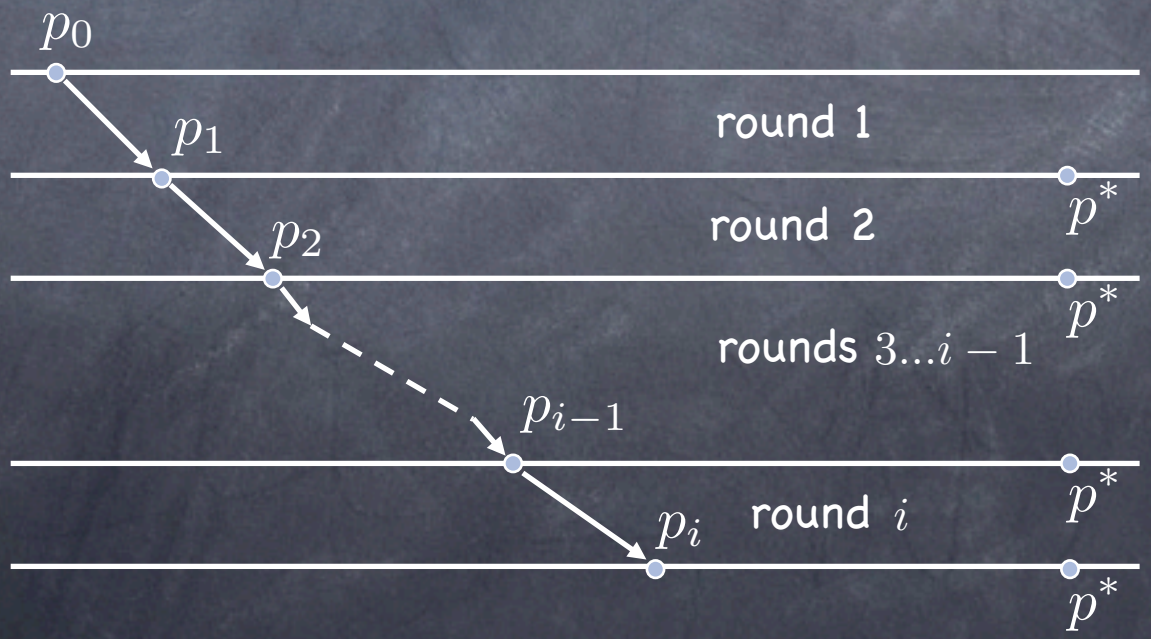
What is going on

- A correct process p^* has not received all proposals by the end of round i. Can p^* decide?
- Another process may have received the missing proposal at the end of round i and be ready to relay it in round $i+1$!

Dangerous Chains

Dangerous chain

The last process in the chain is correct, all others are faulty



Living dangerously

How many rounds can a dangerous chain span?

- f faulty processes
- at most $f+1$ nodes in the chain
- spans at most f rounds

It is safe to decide by the end of round $f+1$!

The Algorithm

Code for process p_i :

Initially $V = \{v_i\}$

To execute propose(v_i)

round k, $1 \leq k \leq f+1$

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all

2: for all j, $0 \leq j \leq n-1$, $j \neq i$ do

3: receive S_j from p_j

4: $V := V \cup S_j$

decide(x) occurs as follows:

5: if $k = f+1$ then

6: decide min(V)

Termination and Integrity

Initially $V = \{v_i\}$

To execute propose(v_i)

round k, $1 \leq k \leq f+1$

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all
2: for all j, $0 \leq j \leq n-1$, $j \neq i$ do
3: receive S_j from p_j
4: $V := V \cup S_j$

decide(x) occurs as follows:

5: if $k = f+1$ then
6: decide min(V)

Termination

Termination and Integrity

Initially $V = \{v_i\}$

To execute propose(v_i)

round k, $1 \leq k \leq f+1$

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all
2: for all j, $0 \leq j \leq n-1$, $j \neq i$ do
3: receive S_j from p_j
4: $V := V \cup S_j$

decide(x) occurs as follows:

5: if $k = f+1$ then
6: decide min(V)

Termination

Every correct process

- reaches round $f + 1$
- decides on min(V) --- which is well defined

Termination and Integrity

Initially $V = \{v_i\}$

To execute propose(v_i)

round k, $1 \leq k \leq f+1$

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all
2: for all j, $0 \leq j \leq n-1$, $j \neq i$ do
3: receive S_j from p_j
4: $V := V \cup S_j$

decide(x) occurs as follows:

5: if $k = f+1$ then
6: decide min(V)

Integrity

At most one value:

Only if it was proposed:

Termination

Every correct process

- reaches round $f+1$
- decides on min(V) --- which is well defined

Termination and Integrity

Initially $V = \{v_i\}$

To execute propose(v_i)

round k, $1 \leq k \leq f+1$

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all
2: for all j, $0 \leq j \leq n-1$, $j \neq i$ do
3: receive S_j from p_j
4: $V := V \cup S_j$

decide(x) occurs as follows:

5: if $k = f+1$ then
6: decide min(V)

Integrity

At most one value:

- one decide, and min(V) is unique

Only if it was proposed:

Termination

Every correct process

- reaches round $f+1$
- decides on min(V) --- which is well defined

Termination and Integrity

Initially $V = \{v_i\}$

To execute propose(v_i)
round k, $1 \le k \le f+1$

1: send $\{v \in V : p_i$ has not already sent $v\}$ to all
2: for all j, $0 \le j \le n-1$, $j \neq i$ do
3: receive S_j from p_j
4: $V := V \cup S_j$

decide(x) occurs as follows:
5: if $k = f+1$ then
6: decide min(V)

Termination

Every correct process

- reaches round $f+1$
- decides on min(V) --- which is well defined

Integrity

At most one value:

- one decide, and min(V) is unique

Only if it was proposed:

- To be decided upon, must be in V at round $f+1$
- if value = v_i, then it is proposed in round 1
- else, suppose received in round k. By induction:
- $k = 1$:
  - by Uniform Integrity of underlying send and receive, it must have been sent in round 1
  - by the protocol and because only crash failures, it must have been proposed
- Induction Hypothesis: all values received up to round $k = j$ have been proposed
- $k = j+1$:
  - sent in round $j+1$ (Uniform Integrity of send and synchronous model)
  - must have been part of V of sender at end of round j
  - by protocol, must have been received by sender by end of round j
  - by induction hypothesis, must have been proposed