Termination protocol: Process states

At any time while running 3 PC, each participant can be in exactly one of these 4 states:

Aborted	Not voted, voted NO, received ABORT
Uncertain	Voted YES, not received PRECOMMIT
Committable	Received PRECOMMIT, not COMMIT
Committed	Received COMMIT

Not all states are compatible

	Aborted	Uncertain	Committable	Committed
Aborted	Y	Y	N	N
Uncertain	Y	Y	Y	N
Committable	N	Y	Y	Y
Committed	N	N	Y	Y

Termination protocol

- When p_i times out, it starts an election protocol to elect a new coordinator
- The new coordinator sends STATE-REQ to all processes that participated in the election
- The new coordinator collects the states and follows a termination rule

TR1. if some process decided ABORT, then
  decide ABORT
  send ABORT to all
  halt

TR2. if some process decided COMMIT, then
  decide COMMIT
  send COMMIT to all
  halt

TR3. if all processes that reported state are uncertain, then
  decide ABORT
  send ABORT to all
  halt

TR4. if some process is committable, but none committed, then
  send PRECOMMIT to uncertain processes
  wait for ACKs
  send COMMIT to all
  halt

Termination protocol and failures

Processes can fail while executing the termination protocol...

- if c times out on p, it can just ignore p
- if c fails, a new coordinator is elected and the protocol is restarted (election protocol to follow)
- total failures will need special care...

Recovering p

- if p fails before sending YES, decide ABORT
- if p fails after having decided, follow decision
- if p fails after voting YES but before receiving decision value
  - p asks other processes for help
  - 3PC is non blocking: p will receive a response with the decision
- if p has received PRECOMMIT
  - still needs to ask other processes (cannot just COMMIT)

Recovering p

- if p fails before sending YES, decide ABORT
- if p fails after having decided, follow decision
- if p fails after voting YES but before receiving decision value
  - p asks other processes for help
  - 3PC is non blocking: p will receive a response with the decision
- if p has received PRECOMMIT
  - still needs to ask other processes (cannot just COMMIT)

No need to log PRECOMMIT!

The election protocol

- Processes agree on linear ordering (e.g. by pid)
- Each p maintains set UP_p of all processes that p believes to be operational
- When p detects failure of c, it removes c from UP_p and chooses smallest q in UP_p to be new coordinator
- If $q = p$, then p is new coordinator
- Otherwise, p sends UR-ELECTED to q

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?
  - it concludes that c must be faulty
  - it removes from $UP_{p'}$ every $q' < q$

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?
  - it concludes that c must be faulty
  - it removes from $UP_{p'}$ every $q' < q$
- What if p' receives a STATE-REQ from c after it has changed the coordinator to q ?

A few observations

- What if p', which has not detected the failure of c, receives a STATE-REQ from q ?
  - it concludes that c must be faulty
  - it removes from $UP_{p'}$ every $q' < q$
- What if p' receives a STATE-REQ from c after it has changed the coordinator to q ?
  - p' ignores the request

Total failure

- Suppose p is the first process to recover, and that p is uncertain
- Can p decide ABORT?

Some processes could have decided COMMIT after p crashed!

Total failure

- Suppose p is the first process to recover, and that p is uncertain
- Can p decide ABORT?

Some processes could have decided COMMIT after p crashed!

- p is blocked until some q recovers s.t. either
  - q can recover independently
  - q is the last process to fail–then q can simply invoke the termination protocol

Determining the last process to fail

- Suppose a set R of processes has recovered
- Does R contain the last process to fail?

Determining the last process to fail

- Suppose a set R of processes has recovered
- Does R contain the last process to fail?
  - the last process to fail is in the UP set of every process
  - so the last process to fail must be in

$$\bigcap_{p \in R} UP_p$$

Determining the last process to fail

- Suppose a set R of processes has recovered
- Does R contain the last process to fail?
  - the last process to fail is in the UP set of every process
  - so the last process to fail must be in

$$\bigcap_{p \in R} UP_p$$

R contains the last process to fail if

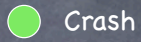
$$\bigcap_{p \in R} UP_p \subseteq R$$

State-Machine Replication

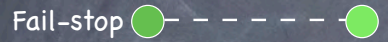
Modeling faults

- Mean Time To Failure/ Mean Time To Recover
  - close to hardware
- Threshold: f out of n
  - makes condition for correct operation explicit
  - measures fault-tolerance of architecture, not single components
- Set of explicit failure scenarios

A hierarchy of failure models

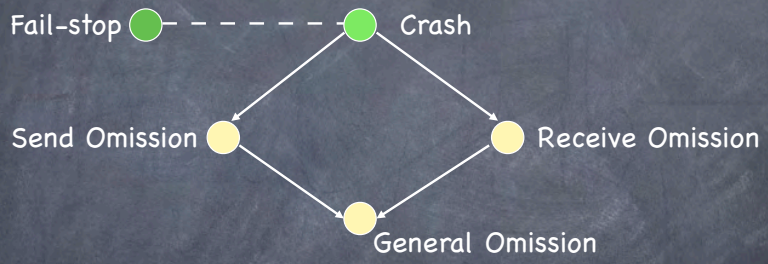


A hierarchy of failure models

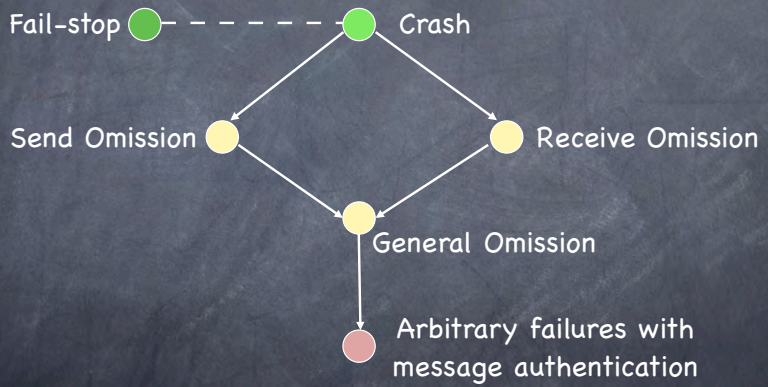


A hierarchy of failure models

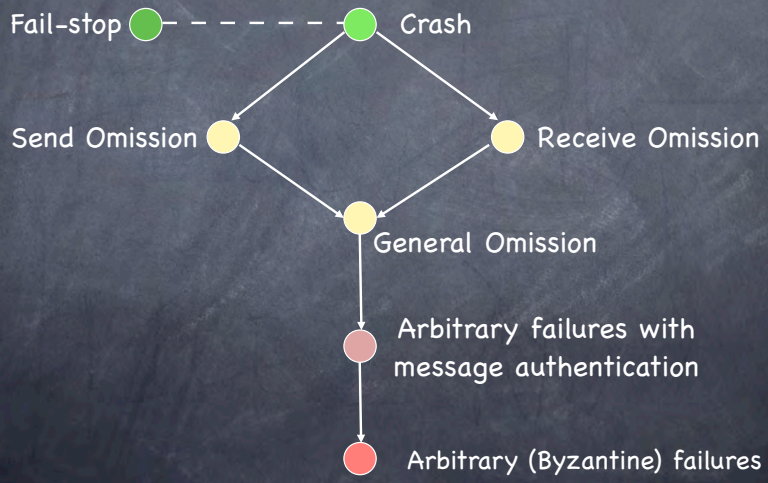
A hierarchy of failure models



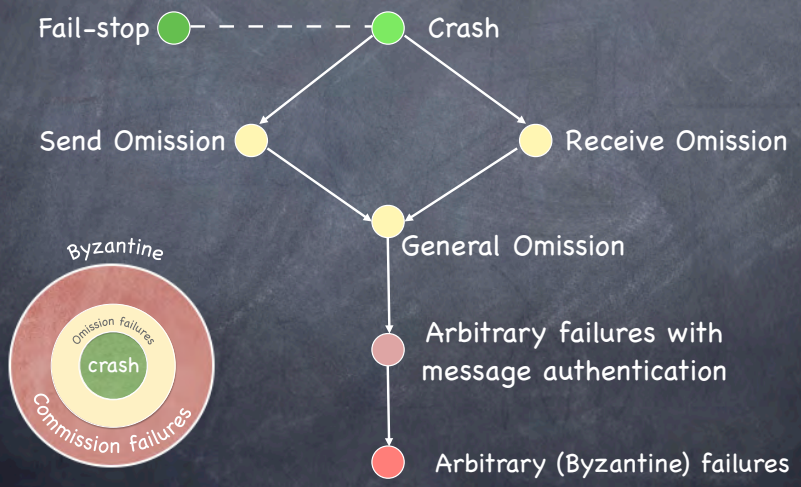
A hierarchy of failure models



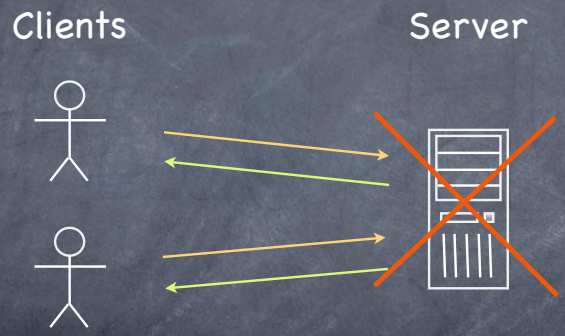
A hierarchy of failure models



A hierarchy of failure models



The Problem



Solution: replicate server!

Replication in space

- Run parallel copies of a unit
- Vote on replica output
- Failures are masked
- High availability, but at high cost

Replication in time

- When a replica fails, restart it (or replace it)
- Failures are detected, not masked
- Lower maintenance, lower availability
- Tolerates only benign failures

Non-determinism

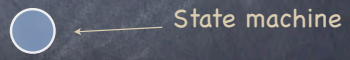
An event is non-deterministic if the state that it produces is not uniquely determined by the state in which it is executed

Handling non-deterministic events at different replicas is challenging

- Replication in time requires to reproduce during recovery the original outcome of all non-deterministic events
- Replication in space requires each replica to handle non-deterministic events identically

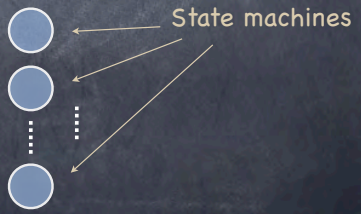
The Solution

1. Make server deterministic (state machine)



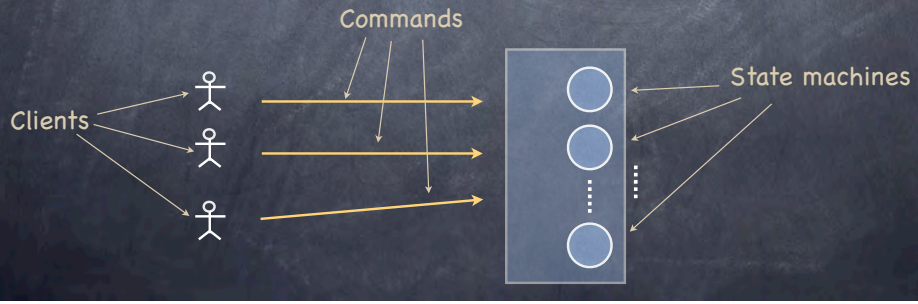
The Solution

1. Make server deterministic (state machine)
2. Replicate server



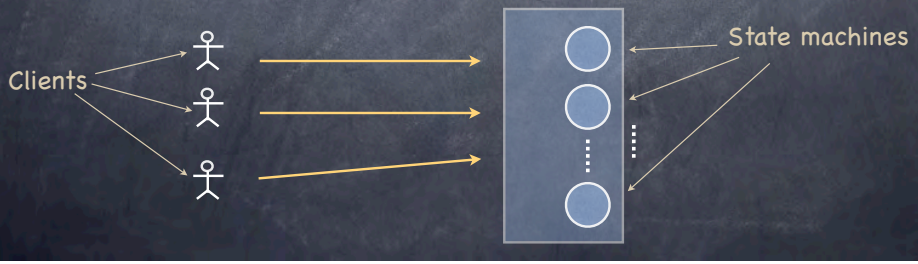
The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions



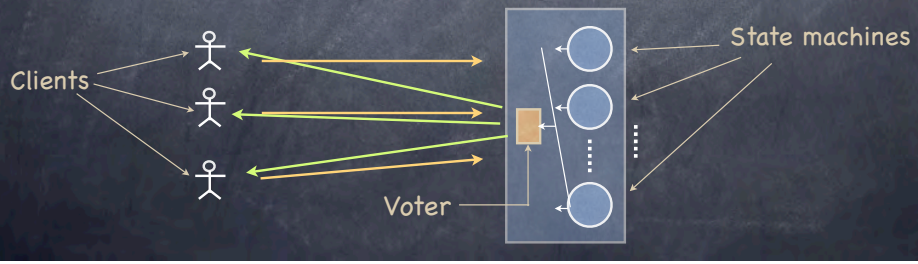
The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance

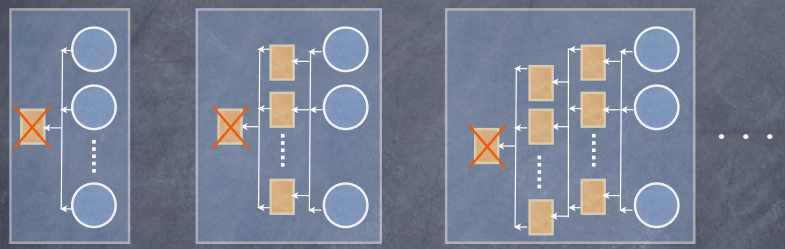


The Solution

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance

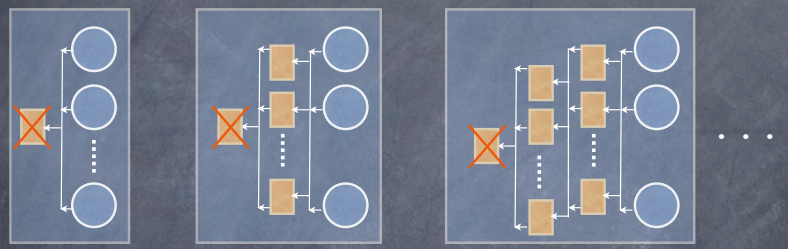


A conundrum

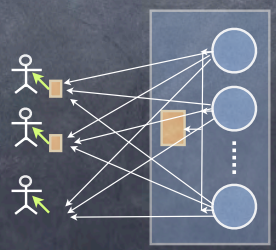


A: voter and client share fate!

A conundrum



A: voter and client share fate!



State Machines

- Set of state variables + Sequence of commands
- A command
  - Reads its read set values (opt. environment)
  - Writes to its write set values (opt. environment)
- A deterministic command
  - Produces deterministic wsvs and outputs on given rsv
- A deterministic state machine
  - Reads a fixed sequence of deterministic commands

Replica Coordination

All non-faulty state machines receive all commands in the same order

- Agreement: Every non-faulty state machine receives every command
- Order: Every non-faulty state machine processes the commands it receives in the same order

Primary-Backup

The Idea

- Clients communicate with a single replica (primary)
- Primary:
  - sequences clients' requests
  - updates as needed other replicas (backups) with sequence of client requests or state updates
  - waits for acks from all non-faulty clients
- Backups use timeouts to detect failure of primary
- On primary failure, a backup is elected as new primary

Primary-backup and non-determinism

- Non-deterministic commands executed only at the primary
- Backups receive either
  - state updates (non-determinism?)
  - command sequence (non-determinism?)

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

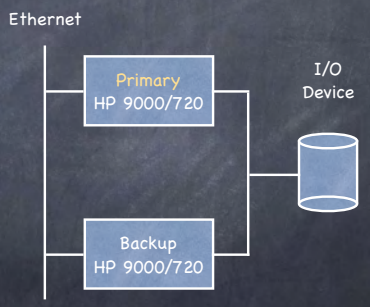
- Two types of commands
  - virtual-machine instructions
  - virtual-machine interrupts (with DMA input)
- State transition must be deterministic
  - ...but some VM instructions are not (e.g. time-of-day)
  - interrupts must be delivered at the same point in command sequence

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)

The Hypervisor prototype

- Supports only one VM to eliminate issues of address translation
- Exploits unused privileged levels in HP's PA-RISC architecture (HV runs at level 1)
- To prevent software to detect HV, hacks one assembly HP-UX boot instruction

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

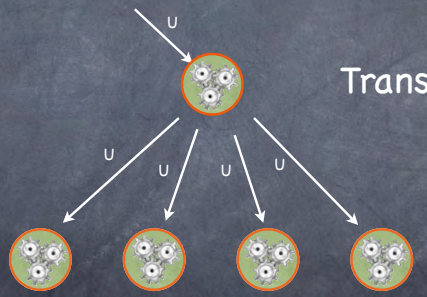
RC in the Hypervisor

- 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

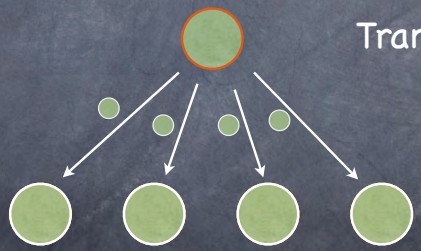
- Optimizations
  - p sends Int immediately
  - p blocks for acks only before output commit

Primary-backup: Updates



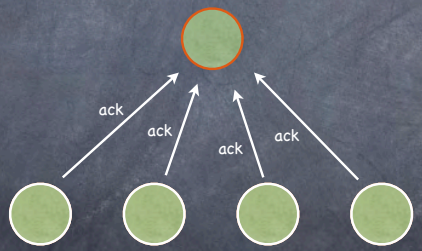
Transfer update

Primary-backup: Updates

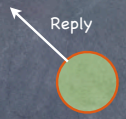


Transfer state

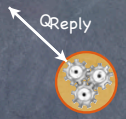
Primary-backup: Updates



Primary-backup: Updates

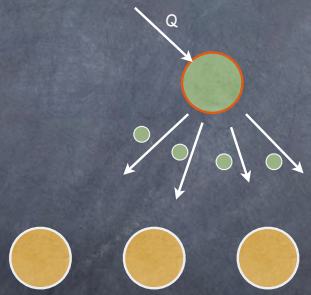


Primary-backup: Queries



However...

Primary-backup: Queries



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

Chain replication

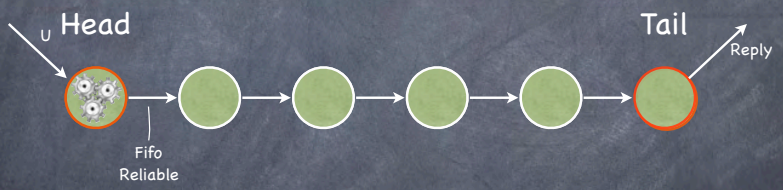
Van Renesse, Schneider, OSDI '04

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

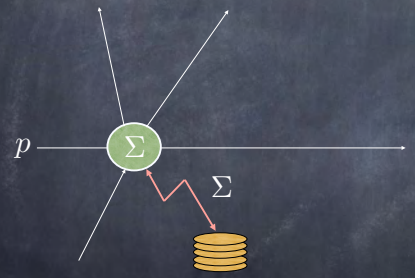
Tail can respond immediately,
without waiting for the new update

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"

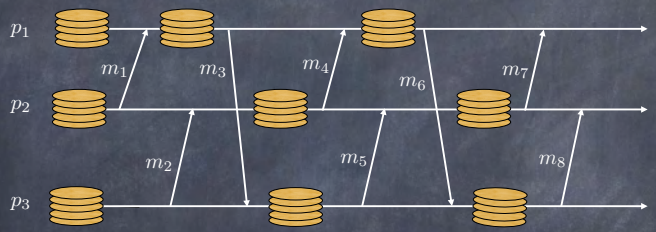
Rollback-Recovery

Uncoordinated Checkpointing

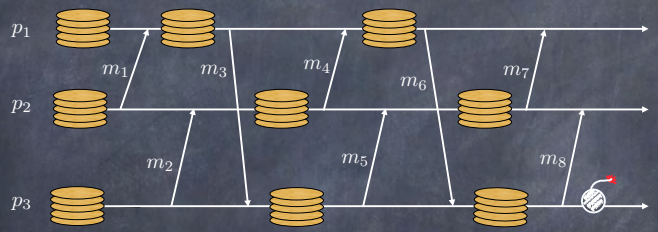


- Easy to understand
- No synchronization overhead
- Flexible
  - can choose when to checkpoint
- To recover from a crash:
  - go back to last checkpoint
  - restart

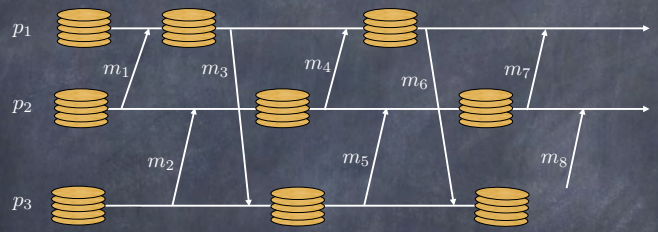
The Domino Effect



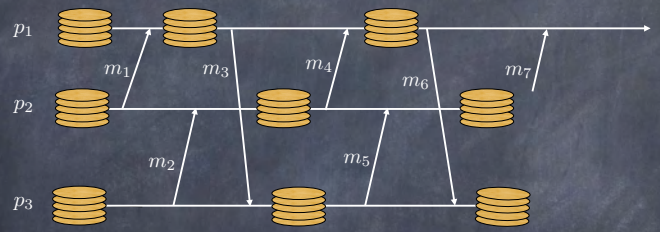
The Domino Effect



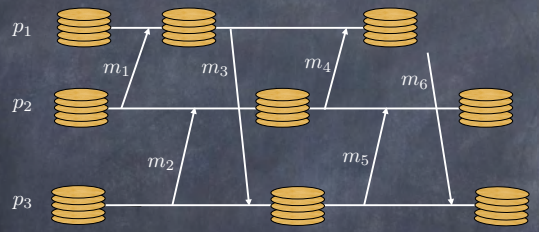
The Domino Effect



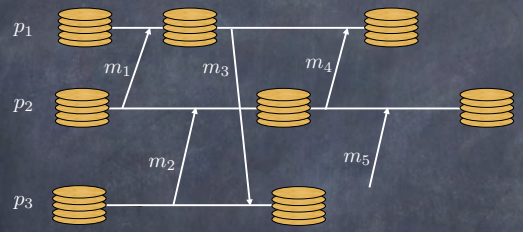
The Domino Effect



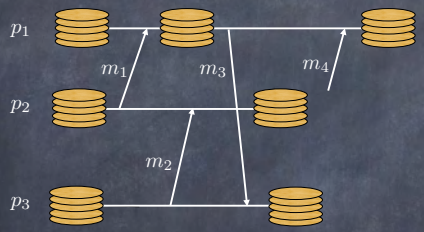
The Domino Effect



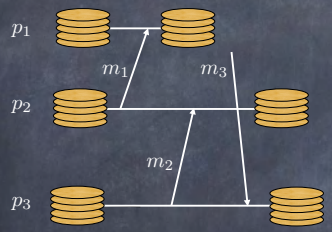
The Domino Effect



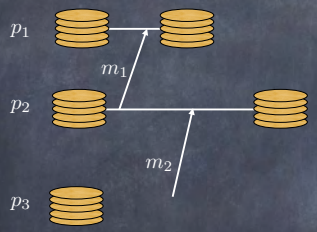
The Domino Effect



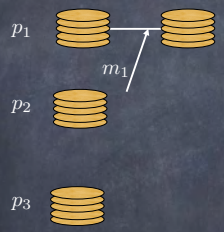
The Domino Effect



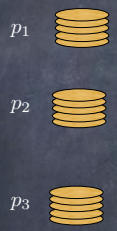
The Domino Effect



The Domino Effect



The Domino Effect



How to Avoid the Domino Effect

Coordinated Checkpointing

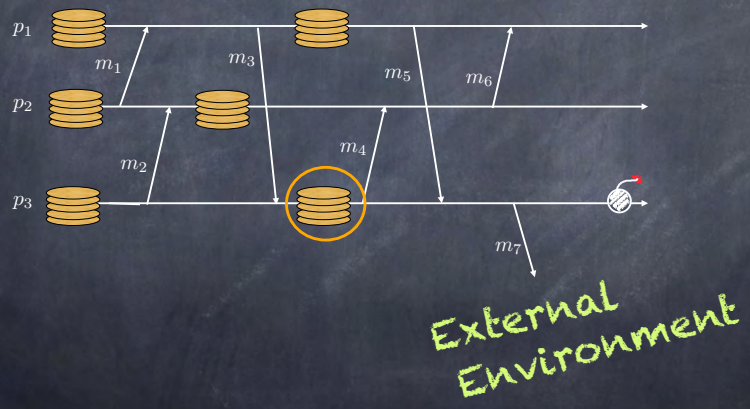
- Easy Garbage Collection
- No independence
- Synchronization Overhead

Communication Induced Checkpointing

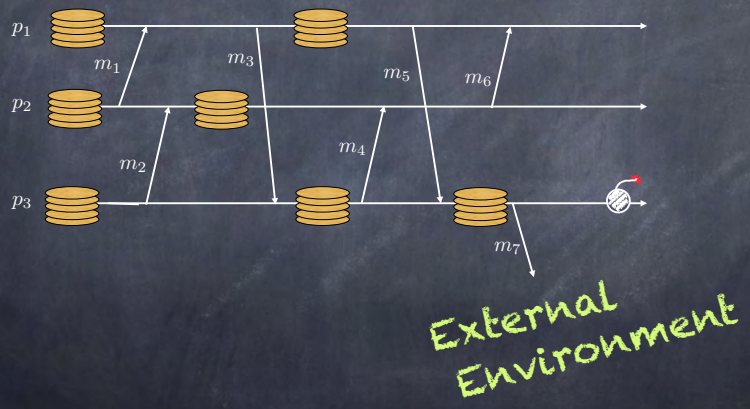
detect dangerous communication patterns and checkpoint appropriately

- Less synchronization
- Less independence

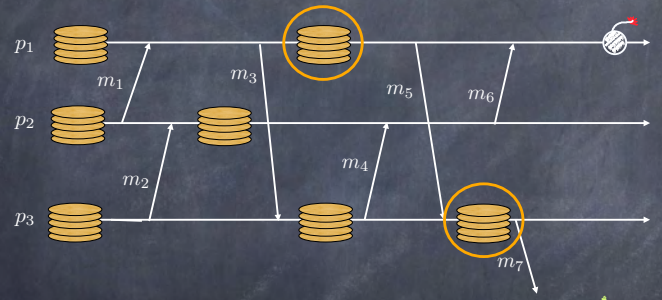
The Output Commit Problem



The Output Commit Problem

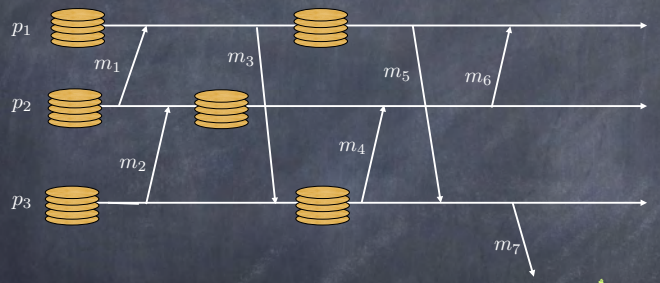


The Output Commit Problem



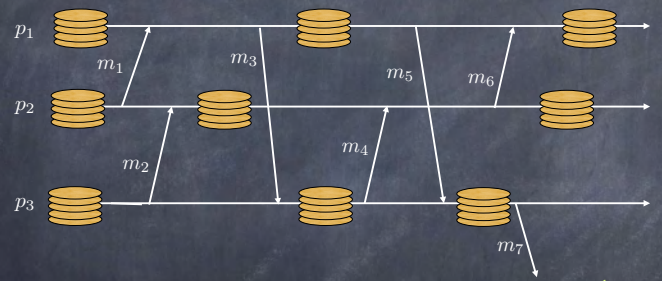
External Environment

The Output Commit Problem



External Environment

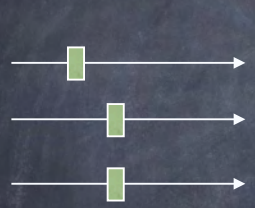
The Output Commit Problem



- Coordinated checkpoint for every output commit
- High overhead if frequent I/O with external environment

External Environment

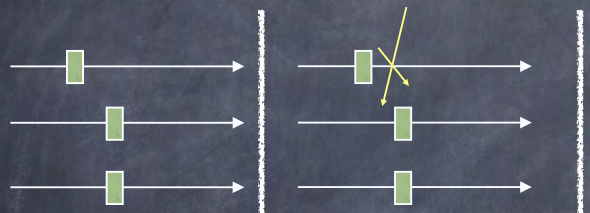
Distributed Checkpointing at a Glance



Independent

\+ Simplicity
\+ Autonomy
\+ Scalability
\- Domino effect

Distributed Checkpointing at a Glance



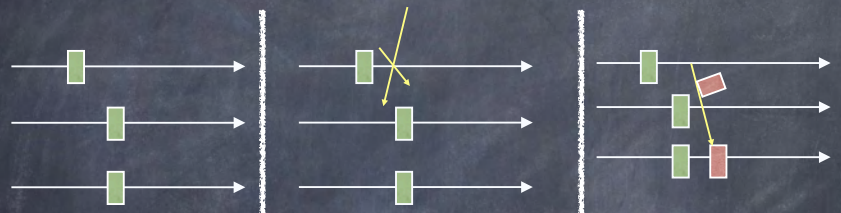
Independent

+ Simplicity
+ Autonomy
+ Scalability
- Domino effect

Coordinated

+ Consistent states
+ Good performance
+ Garbage Collection
- Scalability

Distributed Checkpointing at a Glance



Independent

+ Simplicity
+ Autonomy
+ Scalability
- Domino effect

Coordinated

+ Consistent states
+ Good performance
+ Garbage Collection
- Scalability

Communication-induced

+ Consistent states
+ Autonomy
+ Scalability
- None is true

Message Logging

- Can avoid domino effect
- Works with coordinated checkpoint
- Works with uncoordinated checkpoint
- Can reduce cost of output commit
- More difficult to implement

How It Works

To tolerate crash failures:

- periodically checkpoint application state;
- log on stable storage determinants of non-deterministic events executed after checkpointed state.

Recovery:

- restore latest checkpointed state;
- replay non-deterministic events according to determinants