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

- 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 \([e_s, pc, Val_i] \) from \( p \)
- \( b \) returns \( Val_i \)
- If \( b \) receives \([E, pc, Val_i] \) from \( p \):
  - \( b \) sends ack to \( p \)
  - \( b \) buffers \( Val_i \) 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 \( Int \):
- \( e_b := e_b + 1 \)
- \( b \) starts \( e_b \) \( \quad \) failover epoch
- \( b \) 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 processor issuing the instruction delivers a completion interrupt, unless it fails. Either way, the I/O device is unaffected.

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.

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

Backup promotion and uncertain interrupts

P7: The backup's VM generates an uncertain interrupt for each I/O operation that is outstanding 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

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

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

How (not) to take a checkpoint

- Block execution, save entire process state to stable storage
  - very high overhead during failure-free execution
  - lots of unnecessary data saved on stable storage

How to take a checkpoint

- Take checkpoints incrementally
  - save only pages modified since last checkpoint
  - use "dirty" bit to determine which pages to save
- Save only "interesting" parts of address space
  - use application hints or compiler help to avoid saving useless data (e.g. dead variables)
- Do not block application execution during recovery
  - copy-on-write
The Domino Effect
The Domino Effect

The Domino Effect

The Domino Effect

The Domino Effect
The Domino Effect

Coordinated Checkpointing
- No independence
- Synchronization Overhead
- Easy Garbage Collection

Communication Induced Checkpointing: detect dangerous communication patterns and checkpoint appropriately
- Less synchronization
- Less independence
- Complex

How to Avoid the Domino Effect
The Output Commit Problem
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

+ Consistent states
+ Good performance
+ Garbage Collection
- Scalability
- Domino effect

Independent

Coordinated

Communication-induced

+ Consistent states
+ Autonomy
+ Scalability
- None is true

External Environment
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 Message Logging Works

To tolerate crash failures:
- periodically checkpoint application state;
- log on stable storage determinants of non-deterministic events executed after checkpointed state.
- for message delivery events:

\[ m = (m.\text{dest}, m.\text{rsn}, m.\text{source}, m.\text{ssn}) \]

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

Pessimistic Logging

p2 logs synchronously to stable storage the determinants of m1 and m2 before sending m3.

Sender Based Logging

(Johnson and Zwaenepoel, FTCS 87)
Message log is maintained in volatile storage at the sender.
A message m is logged in two steps:
1) before sending m, the sender logs its content: m is partially logged
2) the receiver tells the sender the receive sequence number of m, and the sender adds this information to its log: m is fully logged.

\[ (m.\text{data}, m.\text{ssn}) \]
\[ (m.\text{ssn}, m.\text{rsn}) \]
\[ (\text{ACK}, m.\text{rsn}) \]

p blocks?
q knows m is fully logged
Optimistic Logging

- \( p_2 \) sends \( m_3 \) without first logging determinants.
- If \( p_2 \) fails before logging the determinants of \( m_1 \) and \( m_2 \), \( p_3 \) becomes an orphan.

Eliminates orphans during recovery
- Non-blocking during failure-free executions
- Rollback of correct processes
- Complex recovery

Causal Logging

- No blocking in failure-free executions
- No orphans
- No additional messages
- Tolerates multiple concurrent failures
- Keeps determinant in volatile memory
- Localized output commit

Preliminary Definitions

Given a message \( m \) sent from \( m.source \) to \( m.dest \),

\[
\text{Depend}(m) = \left\{ p \in P \mid \forall (p = m.dest) \text{ and } p \text{ delivered } m \right. \\
\left. \lor \exists e_p : (\text{deliver}_{m.dest}(m) \rightarrow e_p) \right\}
\]

Log(\( m \)): set of processes with a copy of the determinant of \( m \) in their volatile memory

\( p \) orphan of a set \( C \) of crashed processes:

\( (p \notin C) \land \exists m : (\text{Log}(m) \subseteq C \land p \in \text{Depend}(m)) \)

The “No-Orphans” Consistency Condition

No orphans after crash \( C \) if:

\[
\forall m : (\text{Log}(m) \subseteq C) \Rightarrow (\text{Depend}(m) \subseteq C)
\]

No orphans after any \( C \) if:

\[
\forall m : (\text{Depend}(m) \subseteq \text{Log}(m))
\]

The Consistency Condition

\[
\forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m)))
\]
Optimistic and Pessimistic

No orphans after crash \( C \) if:
\[ \forall m : (\text{Log}(m) \subseteq C) \Rightarrow (\text{Depend}(m) \subseteq C) \]
Optimistic weakens it to:
\[ \forall m : (\text{Log}(m) \subseteq C) \Rightarrow \Diamond (\text{Depend}(m) \subseteq C) \]

No orphans after any crash if:
\[ \forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m))) \]
Pessimistic strengthens it to:
\[ \forall m : (\neg \text{stable}(m) \Rightarrow |\text{Depend}(m)| \leq 1) \]

Causal Message Logging

No orphans after any crash of size at most \( f \) if:
\[ \forall m : (-\text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m))) \]
Causal strengthens it to:
\[ \forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m) \land \Diamond (\text{Depend}(m) = \text{Log}(m)))) \]

An Example

Causal Logging:
\[ \forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m))) \]
If \( f = 1 \), \( \text{stable}(m) = |\text{Log}(m)| \geq 2 \)

Recovery for \( f = 1 \)

Parents of \( p \) Messages previously sent to \( p \) by its parents
What is the next message from each parent?
Who is my next parent?

Determinants of messages delivered by parents
SSN order
RSN order

Determinants of messages delivered by \( p \)
Children of \( p \)
Family-Based Logging

Each process $p$ maintains $D_p = \{ \#m : p \in \text{Depend}(m) \}$ in volatile memory.

**On sending a message $m'$**
- adds $m'$ to volatile send log
- piggybacks on messages to $q$ all determinants $\#m \in D_p$ s.t. $|\text{Log}(m)| = f \wedge (q \notin \text{Log}(m))$

**On receiving a message $m'$**
- adds $m'$ to $D_p$ any new determinant piggybacked on $m'$
- adds $\#m'$ to $D_p$
- updates its estimate of $|\text{Log}(m)|$ for all determinants $\#m \in D_p$

|\text{Log}(m)|_p = f \wedge (q \notin \text{Log}(m))

The Idea

Because $\forall m : (\neg \text{stable}(m) \Rightarrow (\text{Depend}(m) \subseteq \text{Log}(m)))$
we can approximate $\text{Log}(m)$ from below with:
and then use vector clocks to track $\text{Depend}(m)$!

$\text{Log}(m) = \begin{cases} \text{Depend}(m) & \text{if } |\text{Depend}(m)| \leq f \\
\text{Any set } S : |S| > f & \text{otherwise} \end{cases}$

Dependency Vectors

**Dependency Vector** (DV): vector clock that tracks causal dependencies between message delivery events.

$\text{deliver}_p(m) \rightarrow \text{deliver}_q(m') \equiv \text{DV}_p(\text{deliver}_p(m))[p] \leq \text{DV}_q(\text{deliver}_q(m'))[p]$
Weak Dependency Vectors

**Weak Dependency Vector (WDV):**
track causal dependencies on deliver(m) as long as

\(|\text{Depend}(m)| \leq f\)

\((\text{deliver}_p(m) \rightarrow \text{deliver}_q(m')) \land (|\text{Depend}(m)| \leq f) \Rightarrow WDV_p(\text{deliver}_p(m))[p] \leq WDV_q(\text{deliver}_q(m'))[p]\)

\(WDV_p(\text{deliver}_p(m))[p] \leq WDV_q(\text{deliver}_q(m'))[p] \Rightarrow \text{deliver}_p(m) \rightarrow \text{deliver}_q(m')\)

Message Logging at a Glance

**Pessimistic**
+ No orphans
+ Easy recovery
- Blocks

**Optimistic**
+ Non-blocking
- Orphans
- Complex recovery

**Causal**
+ Non-blocking
+ No orphans
- Complex recovery

Dependency Matrix

Use WDV s to determine if \(p \in \text{Log}(m)\):

\(p \in \text{Depend}(m) \land |\text{Depend}(m)| \leq f \Rightarrow WDV_p[m.\text{dest}] \geq m.\text{rsn}\)

Each \(p\) keeps a **Dependency Matrix** (\(\text{DM}_p\))

Given \(\#m = \{u, s, 14, 15\}\):

\(WDV_p[m.\text{dest}] \geq m.\text{rsn} \Rightarrow p \in \text{Depend}(m)\)

\(\text{Log}(m)_p = \{p, q, s\}\)