Parallel Algorithms and Programming

Fault tolerance for Parallel Applications

Thomas Ropars

thomas.ropars@unvi-grenoble-alpes.fr

2018

Agenda

Murphy's law

Whatever can go wrong will go wrong at the worst possible time and in the worst possible way.

Agenda

About failures in large scale systems

The basic problem

Checkpoint-based protocols

Log-based protocols

Recent contributions

Alternatives to rollback-recovery

Mean Time Between Failures

Any component in a computing system may fail:

- This probability can be expressed as a function of the Mean Time Between Failure (MTBF)

Example: MTBF of a disk

Typical MTBF of a disk range between 30 and 120 years (source: seagate)

- Does this mean that my disk can run for 30 years without failures?

Mean Time Between Failures

Any component in a computing system may fail:

- This probability can be expressed as a function of the Mean Time Between Failure (MTBF)

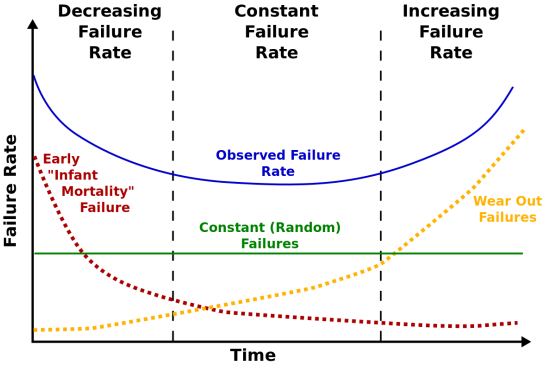
Example: MTBF of a disk

Typical MTBF of a disk range between 30 and 120 years (source: seagate)

- Does this mean that my disk can run for 30 years without failures?
    - No, this MTBF does not take into account aging
    - This number corresponds to the MTBF during normal life (e.g., 3 years)
        - Testing 1000 disks during 6 months, we observe that only 6 fail $\rightarrow$ MTBF of 83 years

More about MTBF

The bathtub



- Infant mortality is due to defective products
- During normal operation, failure rate is low and almost constant

MTBF of complex systems

In a system integrating many components, the failure of any of the components can result in the failure on the whole system.

We use 1000 disks to build a large storage server.

- Recall: 1000 disks run during 6 months and only 6 fail.
- Failure rate $= \frac{6}{1 \times 0.5} = 12$
- MTBF $= \frac{1}{12} = 1$ month
- Note that most data are still available when a disk fails

MTBF range of other complex systems

- A laptop/desktop
  - Typical MTBF in the order of 3 years
- A data-center
  - Built out of 1000 *low-cost* nodes
  - MTBF $= 3/1000 \simeq 26$ hours
  - Large scale datacenters are in the scale of tens of thousands of nodes
  - Note that in this context, the failure of a node usually does not prevent the system from functioning.
- A supercomputer
  - Typical MTBF of a node $= 5$ years
  - Largest supercomputers $= 100000$ nodes
  - System MTBF $= 26$ minutes
  - Bad news: applications are usually tightly coupled

Characterization of Faults

A failure occurs when an error/fault reaches the service interface and alters the service.

- Domain
  - Hardware faults
  - Software faults
- Intent
  - Non-malicious
  - Malicious

Characterization of Faults: Persistence

Transient (soft) faults/errors

- Occurs once and disappears
- Eg, bit-flip due to high-energy particles
- Tend to be due to transient physical phenomena

Intermittent faults/errors

- Occurs occasionally
- Eg, a router drops some packets

Permanent (hard) faults/errors

- Occurs and doesn't go away
- Eg, a dead power supply

What kind of failures for large supercomputers?

Example of Blue Waters (B. Kramer, C. Di Martino et al)

Crash failures

- Hardware faults
  - Node failure MTBF: 6.7 hours
- Detected (uncorrectable) soft errors
  - 261 days $\Rightarrow$ 1.5 Millions of memory errors
  - 99.997% of the errors were corrected (28 uncorrectable errors)

What kind of failures for large supercomputers?

Example of Blue Waters (B. Kramer, C. Di Martino et al)

Software failures

- Some facts:
  - Accounts for 75% of the system-wide outages (SWO)
  - 60% of the SWO are due to problems in the failover procedures.
  - Software is the main contributor to repair time (53% – even if only 20% of the errors)
  - Main contributors: 1) File system; 2) Interconnect; 3) Resource manager.
- Additional comments
  - No bathtub curve for software

What kind of failures for large supercomputers?

Silent data corruptions (SDCs)

- Is it really a problem?
  - Data are missing

Failure model

Correctness of a fault tolerance techniques has to be validated against a failure model.

The failure model

- Crash (fail/stop) failures of nodes
- No recovery

Failure model

Correctness of a fault tolerance techniques has to be validated against a failure model.

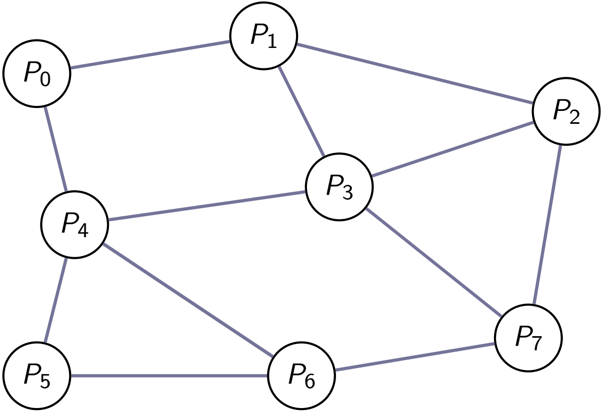
The failure model

- Crash (fail/stop) failures of nodes
- No recovery

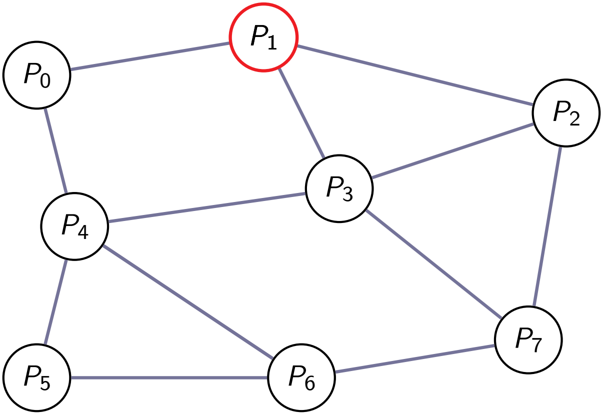
We seek for solutions that ensures the correct termination of parallel applications despite crash failures.

Agenda

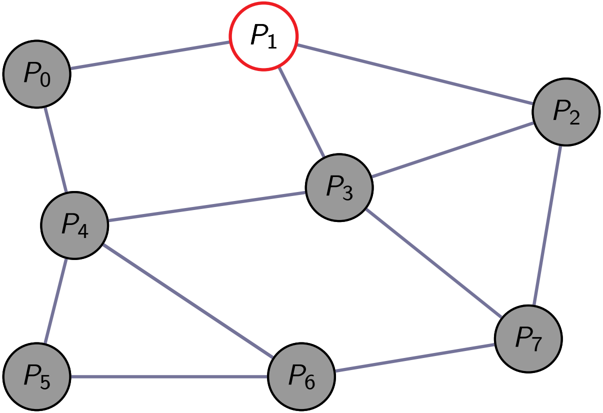
Failures in distributed applications



Failures in distributed applications



Failures in distributed applications



Tightly coupled applications

- One process failure prevents all processes from progressing

Problem definition

A message-passing application

- A fix set of N processes
- Communication by exchanging messages
  - MPI application
- Cooperate to execute a distributed algorithm

Problem definition

A message-passing application

- A fix set of N processes
- Communication by exchanging messages
  - MPI application
- Cooperate to execute a distributed algorithm

An asynchronous distributed system

- Finite set of communication channels connecting any ordered pair of processes
  - Reliable
  - FIFO
  - Ex: TCP, MPI
- Asynchronous
  - Unknown bound on message transmission delays
  - No order between messages on different channels

Problem definition

Crash failures

- When a process fails, it stops executing and communicating.
- All data stored locally is lost

Problem definition

Crash failures

- When a process fails, it stops executing and communicating.
- All data stored locally is lost

Fault tolerance

- How to ensure the correct execution of the application in the presence of faults?
  - The execution should terminate
  - It should provide the correct result

Backward error recovery

Rollback-recovery (other name)

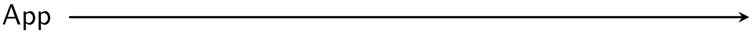
- Restores the application to a previous error-free state when a failure is detected
- Information about the state of the application saved during failure free execution
- Assumes the error will be gone when resuming execution
  - Transient (soft) error
  - Use spare resources to replace faulty ones in case of hard error

BER techniques

- Checkpointing: saving the system state
- Logging: saving changes made to the system

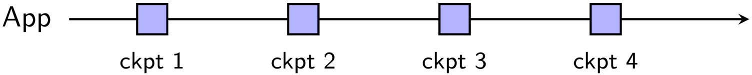
Checkpointing

- Periodically save the state of the application



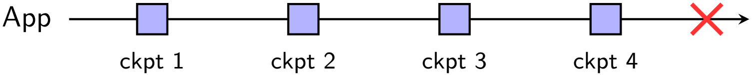
Checkpointing

- Periodically save the state of the application



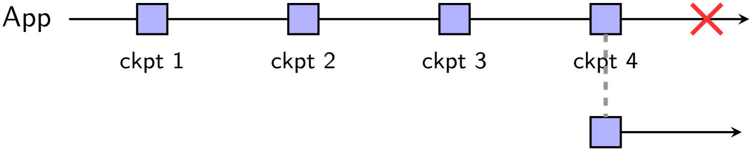
Checkpointing

- Periodically save the state of the application



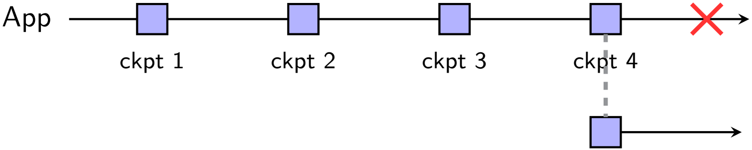
Checkpointing

- Periodically save the state of the application
- Restart from last checkpoint in the event of a failure



Checkpointing

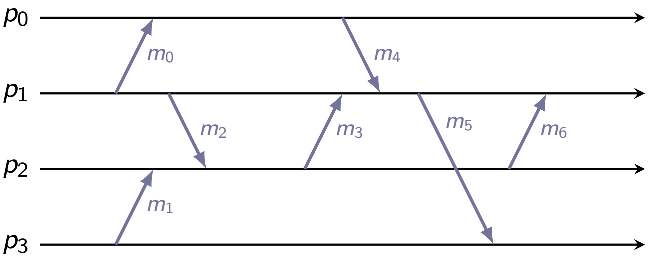
- Periodically save the state of the application
- Restart from last checkpoint in the event of a failure



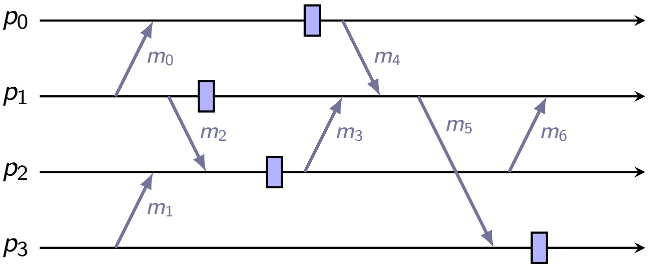
Checkpoint data is saved to reliable storage:

- Reliable storage survives expected failures
- For single node failure, the memory of a neighbor node is a reliable storage
- The parallel file system is a reliable storage

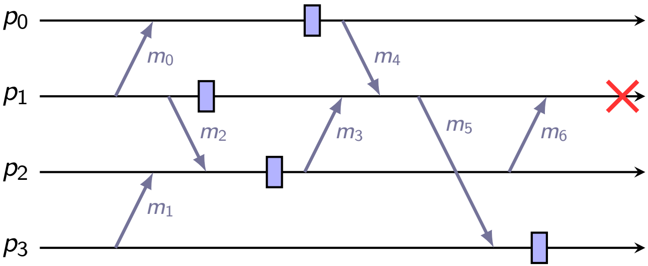
Checkpointing a message-passing application



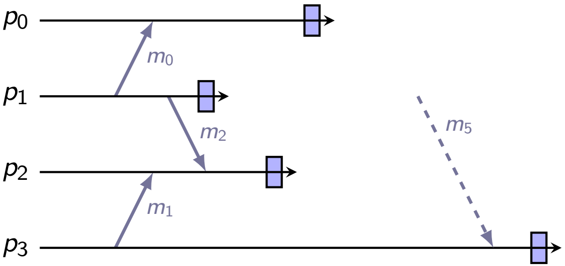
Checkpointing a message-passing application



Checkpointing a message-passing application



Checkpointing a message-passing application



- There is no guaranty that m_5 will still exists (with the same content)
- Processes p_0, p_1 and p_2 might follow a different execution path
- The state of the application would become inconsistent
    - Ensuring a consistent state after the failure is the role of the rollback-recovery protocol

Events and partial order

- The execution of a process can be modeled as a sequence of events.
- The history of process p, noted H(p), includes send(), recv() and internal events.

[1] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM* (1978).

Events and partial order

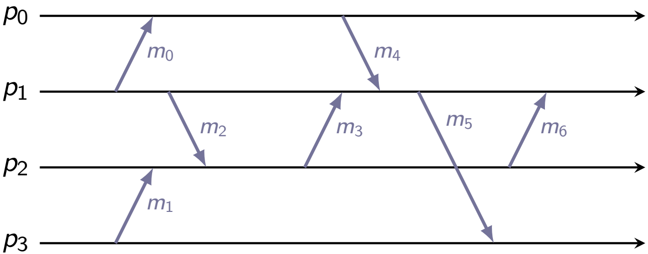
- The execution of a process can be modeled as a sequence of events.
- The history of process p, noted H(p), includes send(), recv() and internal events.

Lamport's Happened-before relation[1]

- noted $\rightarrow$
- Events on one process are totally ordered
  - If e, e' $\in H(p)$, then $e \rightarrow e'$ or $e' \rightarrow e$
- send(m) $\rightarrow$ recv(m)
- Transitivity
  - if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

[1] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM* (1978).

Happened-before relation



Happened-before relations:

- $\text{recv}(m_2) \to \text{send}(m_5)$
- $\text{send}(m_3) \parallel \text{send}(m_4)$

Consistent global state

A rollback-recovery protocol should restore the application in a consistent global state after a failure.

- A consistent state is one that could have been seen during failure-free execution
- A consistent state is a state defined by a consistent cut.

Consistent global state

A rollback-recovery protocol should restore the application in a consistent global state after a failure.

- A consistent state is one that could have been seen during failure-free execution
- A consistent state is a state defined by a consistent cut.

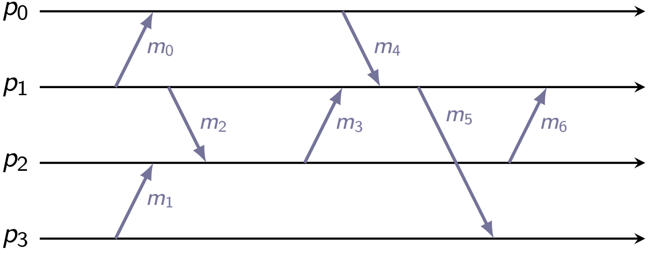
Definition

A cut C is consistent iff for all events e and e':

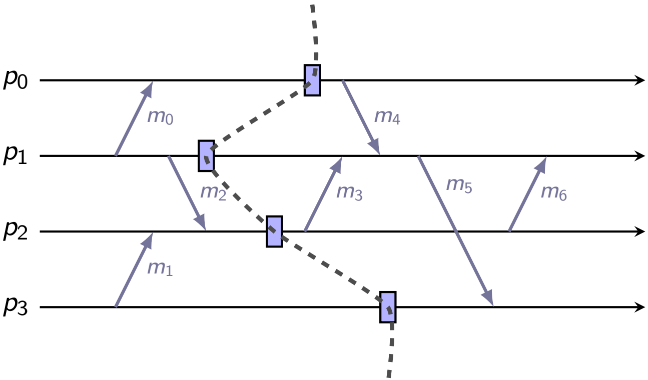
$$e' \in C \; and \; e \rightarrow e' \Longrightarrow e \in C$$

- If the state of a process reflects a message reception, then the state of the corresponding sender should reflect the sending of that message

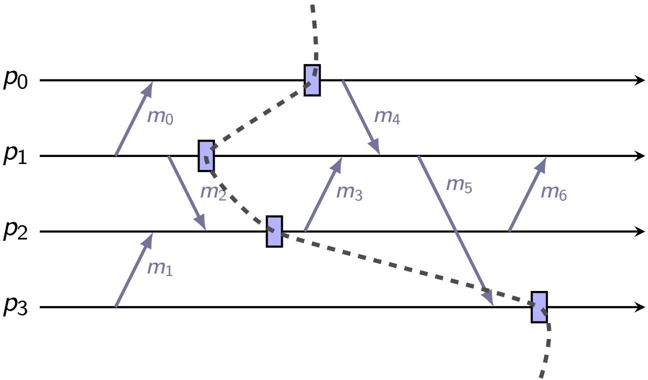
Consistent global state



Consistent global state



Consistent global state



Inconsistent recovery line

- Message m_5 is an orphan message
- P_3 is an orphan process

Before discussing protocols design

- What data to save?
- How to save the state of a process?
- Where to store the data? (reliable storage)
- How frequently to checkpoint?

What data to save?

- The non-temporary application data
- The application data that have been modified since the last checkpoint

What data to save?

- The non-temporary application data
- The application data that have been modified since the last checkpoint

Incremental checkpointing

- Monitor data modifications between checkpoints to save only the changes
  - Save storage space
  - Reduce checkpoint time
- Makes garbage collection more complex
  - Garbage collection = deleting checkpoints that are no longer useful

How to save the state of a process?

Application-level checkpointing

The programmer provides the code to save the process state

- ☺ Only useful data are stored
- ☺ Checkpoint saved when the state is small
- ☹ Difficult to control the checkpoint frequency
- ☹ The programmer has to do the work

System-level checkpointing

The process state is saved by an external tool (ex: BLCR)

- ☹ The whole process state is saved
- ☺ Full control on the checkpoint frequency
- ☺ Transparent for the programmer

How frequently to checkpoint?

- Checkpointing too often prevents the application from making progress
- Checkpointing too infrequently leads to large roll backs in the event of a failure

Optimal checkpoint frequency depends on:

- The time to checkpoint
- The time to restart/recover
- The failure distribution

Agenda

Checkpointing protocols

Three categories of techniques

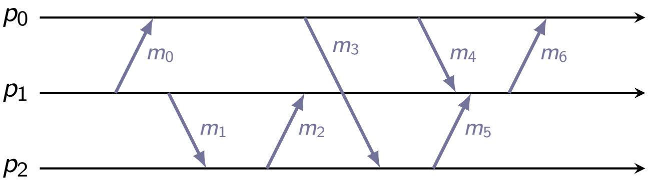
- Uncoordinated checkpointing
- Coordinated checkpointing
- Communication-induced checkpointing (not efficient with HPC workloads[1])

[1] L. Alvisi et al. "An analysis of communication-induced checkpointing". *FTCS*. 1999.

Uncoordinated checkpointing

Idea

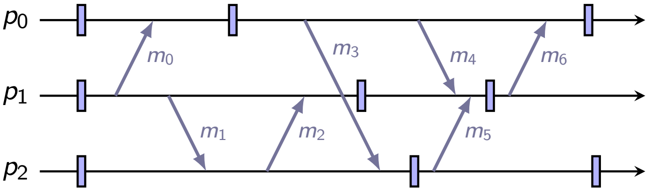
Save checkpoints of each process independently.



Uncoordinated checkpointing

Idea

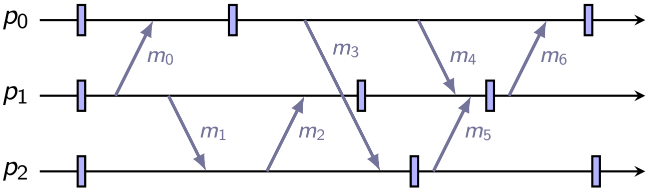
Save checkpoints of each process independently.



Uncoordinated checkpointing

Idea

Save checkpoints of each process independently.



Problem

- Is there any guaranty that we can find a consistent state after a failure?
- Domino effect
  - Cascading rollbacks on all processes (unbounded)
  - If process p_1 fails, the only consistent state we can find is the initial state

Uncoordinated checkpointing

Implementation

- Direct dependencies between the checkpoint intervals are recorded
  - Data piggybacked on messages and saved in the checkpoints
- Used after a failure to construct a dependency graph and compute the recovery line
  - [Bhargava and Lian, 1988]
  - [Wang, 1993]

Other comments

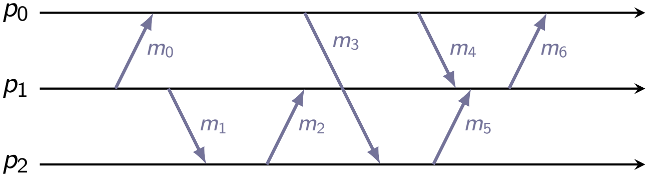
- Garbage collection is very inefficient
  - Hard to decide when a checkpoint is not useful anymore
  - Many checkpoints may have to be stored

Coordinated checkpointing

Idea

Coordinate the processes at checkpoint time to ensure that the global state that is saved is consistent.

- No domino effect

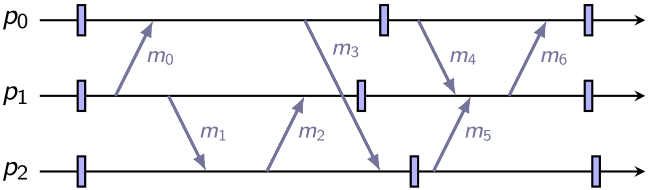


Coordinated checkpointing

Idea

Coordinate the processes at checkpoint time to ensure that the global state that is saved is consistent.

- No domino effect



Coordinated checkpointing

Recovery after a failure

- All processes restart from the last coordinated checkpoint
  - Even the non-failed processes have to rollback
- Idea: Restart only the processes that depend on the failed process[1]
  - In HPC apps: transitive dependencies between all processes

[1] R. Koo et al. "Checkpointing and Rollback-Recovery for Distributed Systems". *ACM Fall joint computer conference*. 1986.

Coordinated checkpointing

Other comments

- Simple and efficient garbage collection
  - Only the last checkpoint should be kept
- Performance issues?
  - What happens when one wants to save the state of all processes at the same time?

Coordinated checkpointing

Other comments

- Simple and efficient garbage collection
  - Only the last checkpoint should be kept
- Performance issues?
  - What happens when one wants to save the state of all processes at the same time?

How to coordinate?

At the application level

Idea: Take advantage of the structure of the code

- The application code might already include global synchronization
  - MPI collective operations
- In iterative codes, checkpoint every N iterations

Time-based checkpointing[1]

Idea

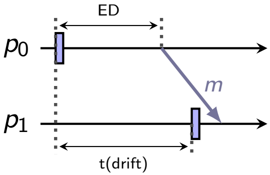
- Each process takes a checkpoint at the same time
- A solution is needed to synchronize clocks

[1] N. Neves et al. "Coordinated checkpointing without direct coordination". *IPDS'98*.

Time-based checkpointing

To ensure consistency

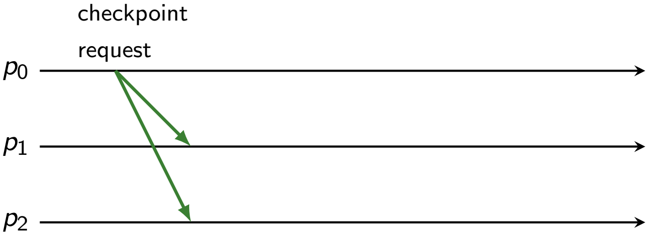
- After checkpointing, a process should not send a message that could be received before the destination saved its checkpoint
  - The process waits for a delay corresponding to the effective deviation
  - The effective deviation is computed based on the clock drift and the message transmission delay



$$ED = t(clock\ drift) - minimum\ transmission\ delay$$

Blocking coordinated checkpointing[1]

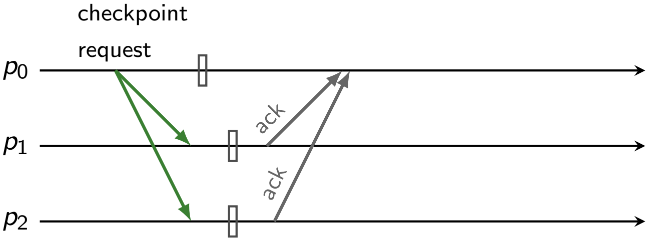
1. The initiator broadcasts a checkpoint request to all processes



[1] Y. Tamir et al. "Error Recovery in Multicomputers Using Global Checkpoints". *ICPP*. 1984.

Blocking coordinated checkpointing[1]

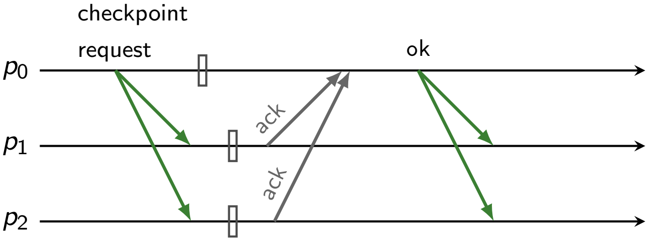
1. The initiator broadcasts a checkpoint request to all processes
2. Upon reception of the request, each process stops executing the application and saves a checkpoint, and sends `ack` to the initiator



[1] Y. Tamir et al. "Error Recovery in Multicomputers Using Global Checkpoints". *ICPP*. 1984.

Blocking coordinated checkpointing[1]

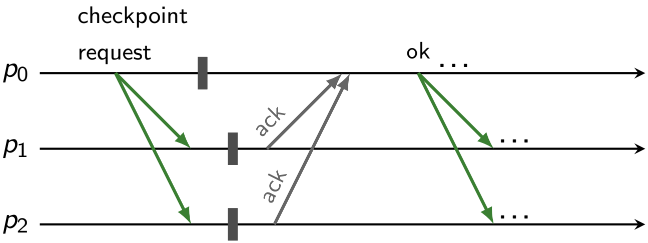
1. The initiator broadcasts a checkpoint request to all processes
2. Upon reception of the request, each process stops executing the application and saves a checkpoint, and sends `ack` to the initiator
3. When the initiator has received all acks, it broadcasts `ok`



[1] Y. Tamir et al. "Error Recovery in Multicomputers Using Global Checkpoints". *ICPP*. 1984.

Blocking coordinated checkpointing[1]

1. The initiator broadcasts a checkpoint request to all processes
2. Upon reception of the request, each process stops executing the application and saves a checkpoint, and sends `ack` to the initiator
3. When the initiator has received all acks, it broadcasts `ok`
4. Upon reception of the `ok` message, each process deletes its old checkpoint and resumes execution of the application



[1] Y. Tamir et al. "Error Recovery in Multicomputers Using Global Checkpoints". *ICPP*. 1984.

Blocking coordinated checkpointing

Correctness

Does the global checkpoint corresponds to a consistent state, i.e., a state with no orphan messages?

Blocking coordinated checkpointing

Correctness

Does the global checkpoint corresponds to a consistent state, i.e., a state with no orphan messages?

Proof sketch (by contradiction)

- We assume the state is not consistent, and there is an orphan message m such that:

$$send(m) \notin C \ and \ recv(m) \in C$$

- It means that m was sent after receiving *ok* by p_i
- It also means that m was received before receiving *checkpoint* by p_j
- It implies that:

$$recv(m) \rightarrow recv_j(ckpt) \rightarrow recv_i(ok) \rightarrow send(m)$$

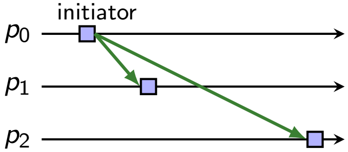
Non-blocking coordinated checkpointing[1]

- Goal: Avoid the cost of synchronization
- How to ensure consistency?

[1] K. Chandy et al. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems* (1985).

Non-blocking coordinated checkpointing[1]

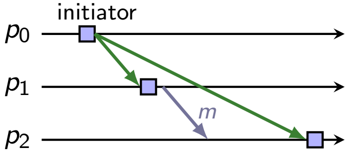
- Goal: Avoid the cost of synchronization
- How to ensure consistency?



[1] K. Chandy et al. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems* (1985).

Non-blocking coordinated checkpointing[1]

- Goal: Avoid the cost of synchronization
- How to ensure consistency?

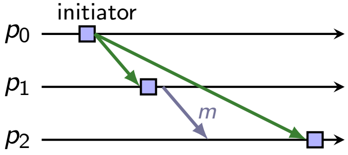


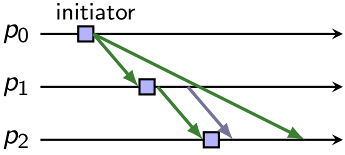
- Inconsistent global state
- Message m is orphan

[1] K. Chandy et al. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems* (1985).

Non-blocking coordinated checkpointing[1]

- Goal: Avoid the cost of synchronization
- How to ensure consistency?





- Inconsistent global state
- Message m is orphan

- Consistent global state
  - Send a marker to force p_2 to save a checkpoint before delivering m

[1] K. Chandy et al. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems* (1985).

Non-blocking coordinated checkpointing

Assuming FIFO channels:

1. The initiator takes a checkpoint and broadcasts a checkpoint request to all processes

Non-blocking coordinated checkpointing

Assuming FIFO channels:

1. The initiator takes a checkpoint and broadcasts a checkpoint request to all processes
2. Upon reception of the request, each process (i) takes a checkpoint, and (ii) broadcast `checkpoint-request` to all. No event can occur between (i) and (ii).

Non-blocking coordinated checkpointing

Assuming FIFO channels:

1. The initiator takes a checkpoint and broadcasts a checkpoint request to all processes
2. Upon reception of the request, each process (i) takes a checkpoint, and (ii) broadcast `checkpoint-request` to all. No event can occur between (i) and (ii).
3. Upon reception of `checkpoint-request` message from all, a process deletes its old checkpoint

Agenda

Message-logging protocols

Idea: Logging the messages exchanged during failure free execution to be able to replay them in the same order after a failure

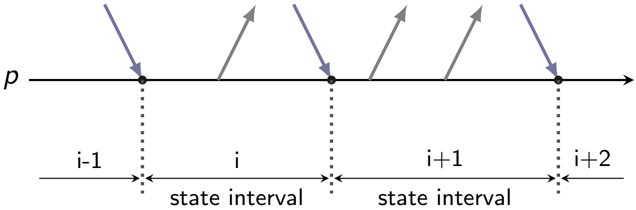
3 families of protocols

- Pessimistic
- Optimistic
- Causal

Piecewise determinism

The execution of a process is a set of deterministic state intervals, each started by a non-deterministic event.

- Most of the time, the only non-deterministic events are message receptions



From a given initial state, playing the same sequence of messages will always lead to the same final state.

Message logging

Basic idea

- Log all non-deterministic events during failure-free execution
- After a failure, the process re-executes based on the events in the log

Consistent state

- If all non-deterministic events have been logged, the process follows the same execution path after the failure
  - Other processes do not roll back. They wait for the failed process to catch up

Message logging

What is logged?

- The content of the messages (payload)
- The delivery order of each message (determinant)
  - Sender id
  - Sender sequence number
  - Receiver id
  - Receiver sequence number

Where to store the data?

Sender-based message logging[1]

- The payload can be saved in the memory of the sender
- If the sender fails, it will generate the messages again during recovery

Event logging

- Determinants have to be saved on a reliable storage
- They should be available to the recovering processes

[1] D. B. Johnson et al. "Sender-Based Message Logging". *The 17th Annual International Symposium on Fault-Tolerant Computing*. 1987.

Event logging

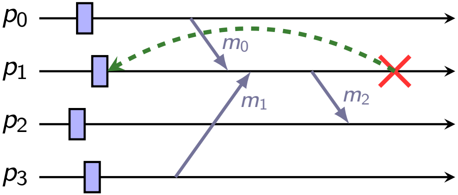
Important

- Determinants are saved by message receivers
- Event logging has an impact on performance as it involves a remote synchronization

The 3 protocol families correspond to different ways of managing determinants.

The always no-orphan condition[1]

An orphan message is a message that is seen has received, but whose sending state interval cannot be recovered.



If the determinants of messages $m0$ and $m1$ have not been saved, then message $m2$ is orphan.

[1] L. Alvisi et al. "Message Logging: Pessimistic, Optimistic, Causal, and Optimal". *IEEE Transactions on Software Engineering* (1998).

The always no-orphan condition

- e: a non-deterministic event
- Depend(e): the set of processes whose state causally depends on e
- Log(e): the set of processes that have a copy of the determinant of e in their memory
- Stable(e): a predicate that is true if the determinant of e is logged on a reliable storage

To avoid orphans:

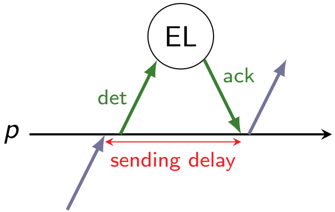
$$\forall e : \neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$$

Pessimistic message logging

Failure-free protocol

- Determinants are logged synchronously on reliable storage

$\forall e : \neg Stable(e) \Rightarrow |Depend(e)| = 1$



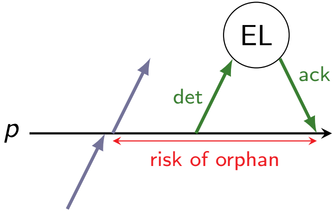
Recovery

- Only the failed process has to restart

Optimistic message logging

Failure-free protocol

- Determinants are logged asynchronously (periodically) on reliable storage



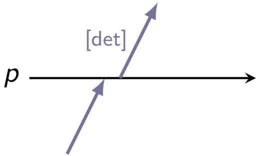
Recovery

- All processes whose state depends on a lost event have to rollback
- Causal dependency tracking has to be implemented during failure-free execution

Causal message logging

Failure-free protocol

- Implements the "always-no-orphan" condition
- Determinants are piggybacked on application messages until they are saved on reliable storage



Recovery

- Only the failed process has to rollback

Comparison of the 3 families

Failure-free performance

- Optimistic ML is the most efficient
- Synchronizing with a remote storage is costly
- Piggybacking potentially large amount of data on messages is costly

Recovery performance

- Pessimistic ML is the most efficient
- Recovery protocols of optimistic and causal ML can be complex

Message logging + checkpointing

Message logging is combined with checkpointing

- To reduce the extends of rollbacks in time
- To reduce the size of the logs

Which checkpointing protocol?

- Uncoordinated checkpointing can be used
  - No risk of domino effect
- Nothing prevents from using coordinated checkpointing

Agenda

Limits of legacy solutions at scale

Coordinated checkpointing

- Contention on the parallel file system if all processes checkpoint/restart *at the same time*
  - More than 50% of wasted time?[1]
  - Solution: see multi-level checkpointing
- Restarting millions of processes because of a single process failure is a big waste of resources

[1] R. A. Oldfield et al. "Modeling the Impact of Checkpoints on Next-Generation Systems". *MSST 2007*.

Limits of legacy solutions at scale

Message logging

- Logging all messages payload consumes a lot of memory
  - Running a climate simulation (CM1) on 512 processes generates $> 1\text{GB/s}$ of logs[1]
- Managing determinants is costly in terms of performance
  - Frequent synchronization with a reliable storage has a high overhead
  - Piggybacking information on messages penalizes communication performance

[1] T. Ropars et al. "SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing". *SuperComputing 2013*.

Coordinated checkpointing + Optimistic ML[1]

Optimistic ML and coordinated checkpointing are combined

- Dedicated *event-logger* nodes are used for efficiency

Optimistic message logging

- Negligible performance overhead in failure-free execution
- If no determinant is lost in a failure, only the failed processes restart

Coordinated checkpointing

- If determinants are lost in a failure, simply restart from the last checkpoint
  - Case of the failure of an event logger
  - No complex recovery protocol
- It simplifies garbage collection of messages

[1] R. Riesen et al. "Alleviating scalability issues of checkpointing protocols". *SuperComputing 2012*.

Revisiting communication events[1]

Idea

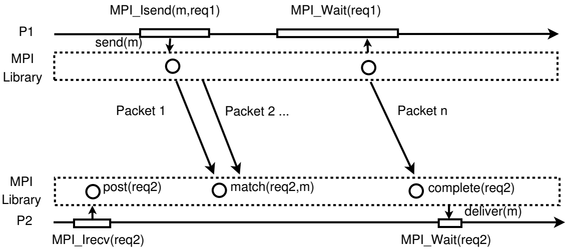
- Piecewise determinism assumes all message receptions are non-deterministic events
- In MPI most reception events are deterministic
  - Discriminating deterministic communication events will improve event logging efficiency

Impact

- The cost of (pessimistic) event logging becomes negligible

[1]A. Bouteiller et al. "Redesigning the Message Logging Model for High Performance". *Concurrency and Computation : Practice and Experience* (2010).

Revisiting communication events



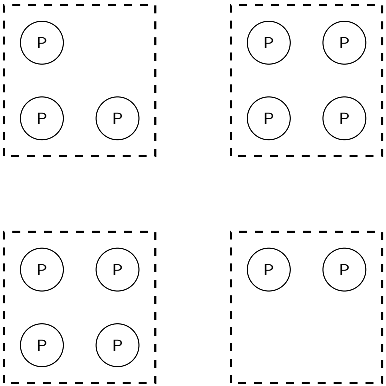
New execution model

2 events associated with each message reception:

- Matching between message and reception request
  - Not deterministic only if `ANY_SOURCE` is used
- Completion when the whole message content has been placed in the user buffer
  - Not deterministic only for `wait_any/some` and `test` functions

Hierarchical protocols[1]

The application processes are grouped in logical clusters



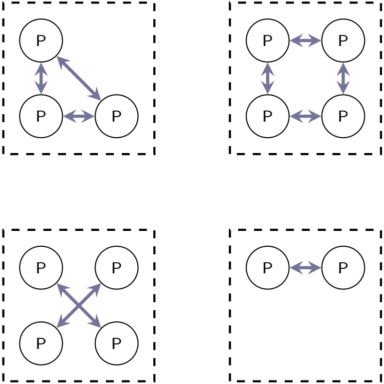
[1] A. Bouteiller et al. "Correlated Set Coordination in Fault Tolerant Message Logging Protocols". Euro-Par'11.

Hierarchical protocols[1]

The application processes are grouped in logical clusters

Failure-free execution

- Take coordinated checkpoints inside clusters periodically



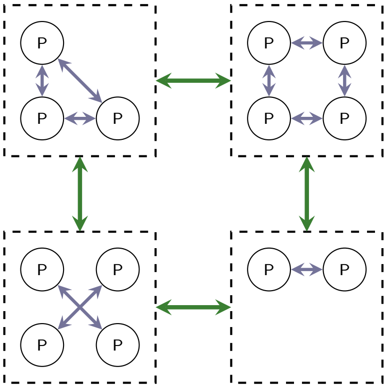
[1]A. Bouteiller et al. "Correlated Set Coordination in Fault Tolerant Message Logging Protocols". Euro-Par'11.

Hierarchical protocols[1]

The application processes are grouped in logical clusters

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages



[1] A. Bouteiller et al. "Correlated Set Coordination in Fault Tolerant Message Logging Protocols". Euro-Par'11.

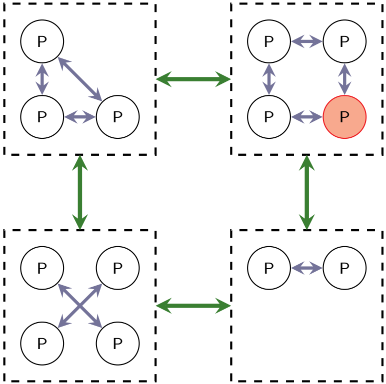
Hierarchical protocols[1]

The application processes are grouped in logical clusters

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages

Recovery



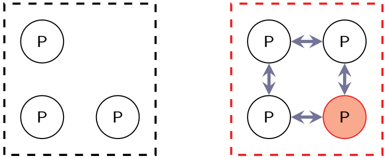
[1] A. Bouteiller et al. "Correlated Set Coordination in Fault Tolerant Message Logging Protocols". Euro-Par'11.

Hierarchical protocols[1]

The application processes are grouped in logical clusters

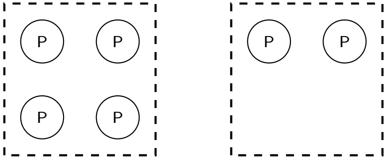
Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages



Recovery

- Restart the failed cluster from the last checkpoint



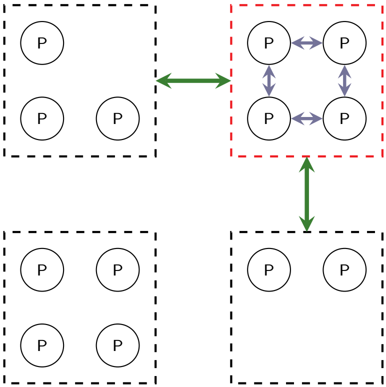
[1] A. Bouteiller et al. "Correlated Set Coordination in Fault Tolerant Message Logging Protocols". Euro-Par'11.

Hierarchical protocols[1]

The application processes are grouped in logical clusters

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages



Recovery

- Restart the failed cluster from the last checkpoint
- Replay missing inter-cluster messages from the logs

[1] A. Bouteiller et al. "Correlated Set Coordination in Fault Tolerant Message Logging Protocols". Euro-Par'11.

Hierarchical protocols

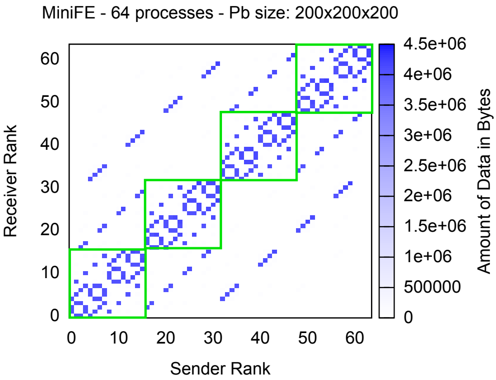
Advantages

- Reduced number of logged messages
  - **But** the determinant of all messages should be logged[1]
- Only a subset of the processes restart after a failure
  - **Failure containment**[2]

[1] A. Bouteiller et al. "Correlated Set Coordination in Fault Tolerant Message Logging Protocols". Euro-Par'11.

[2] J. Chung et al. "Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems". *SuperComputing 2012*.

Hierarchical protocols



Good applicability to most HPC workloads[1]

- $<$ 15% of logged messages
- $<$ 15% of processes to restart after a failure

[1] T. Ropars et al. "On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications". Euro-Par'11.

Revisiting execution models[1]

Non-deterministic algorithm

- An algorithm A is non-deterministic is its execution path is influenced by non-deterministic events
- Assumption we have considered until now

[1] F. Cappello et al. "On Communication Determinism in Parallel HPC Applications". *ICCCN 2010*.

Revisiting execution models[1]

Non-deterministic algorithm

- An algorithm A is non-deterministic is its execution path is influenced by non-deterministic events
- Assumption we have considered until now

Send-deterministic algorithm

- An algorithm `A` is **send-deterministic**, if for an initial state Σ, and for any process `p`, the sequence of send events on `p` is the same in any valid execution of `A`.

[1] F. Cappello et al. "On Communication Determinism in Parallel HPC Applications". *ICCCN 2010*.

Revisiting execution models[1]

Non-deterministic algorithm

- An algorithm A is non-deterministic is its execution path is influenced by non-deterministic events
- Assumption we have considered until now

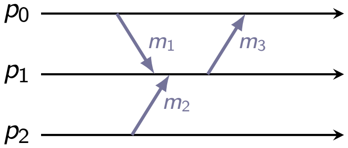
Send-deterministic algorithm

- An algorithm `A` is **send-deterministic**, if for an initial state Σ, and for any process `p`, the sequence of send events on `p` is the same in any valid execution of `A`.
- **Most HPC applications are send-deterministic**

[1] F. Cappello et al. "On Communication Determinism in Parallel HPC Applications". *ICCCN 2010*.

Impact of send-determinism

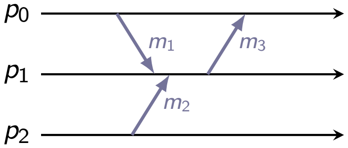
The relative order of the messages received by a process has no impact on its execution.



[1] A. Guermouche et al. "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications". *IPDPS2011*.

Impact of send-determinism

The relative order of the messages received by a process has no impact on its execution.



It is possible to design an uncoordinated checkpointing protocol that has **no risk of domino effect**[1].

[1] A. Guermouche et al. "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications". *IPDPS2011*.

Revisiting message logging protocols[1]

For send-deterministic MPI applications that do not include ANY_SOURCE receptions:

- **Message logging does not need event logging**
- Only logging the payload is required
- This result applies also to hierarchical protocols

[1] T. Ropars et al. "SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing". *SuperComputing 2013*.

Revisiting message logging protocols[1]

For send-deterministic MPI applications that do not include ANY_SOURCE receptions:

- **Message logging does not need event logging**
- Only logging the payload is required
- This result applies also to hierarchical protocols

For applications including ANY_SOURCE receptions:

- Minor modifications of the code are required

[1] T. Ropars et al. "SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing". *SuperComputing 2013*.

Agenda

About failures in large scale systems

The basic problem

Checkpoint-based protocols

Log-based protocols

Recent contributions

Alternatives to rollback-recovery

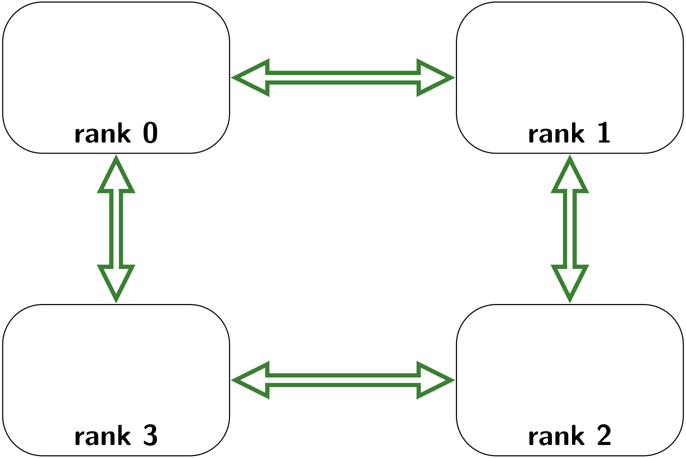
Failure prediction[1]

Idea

- Online analysis of supercomputers system logs to predict failures
  - Coverage of 50%
  - Precision of 90%
- Take advantage of this information to take **preventive actions**
  - Save a checkpoint before the failure occurs

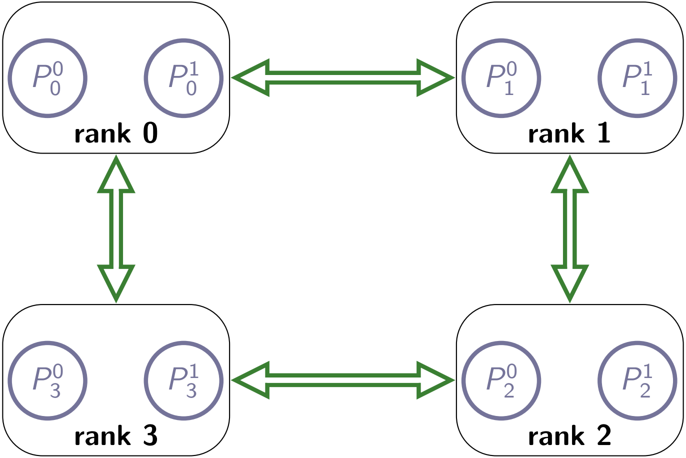
[1] M. S. Bouguerra et al. "Improving the Computing Efficiency of HPC Systems Using a Combination of Proactive and Preventive Checkpointing". *IPDPS'13*.

Active replication[1]



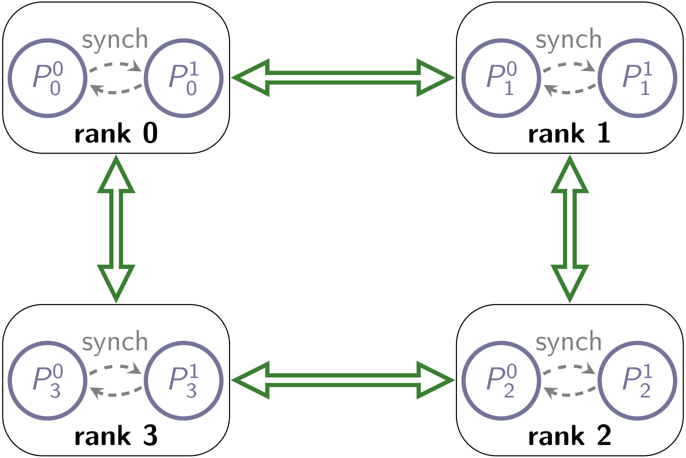
[1]K. Ferreira et al. "Evaluating the Viability of Process Replication Reliability for Exascale Systems". *SuperComputing 2011*.

Active replication[1]



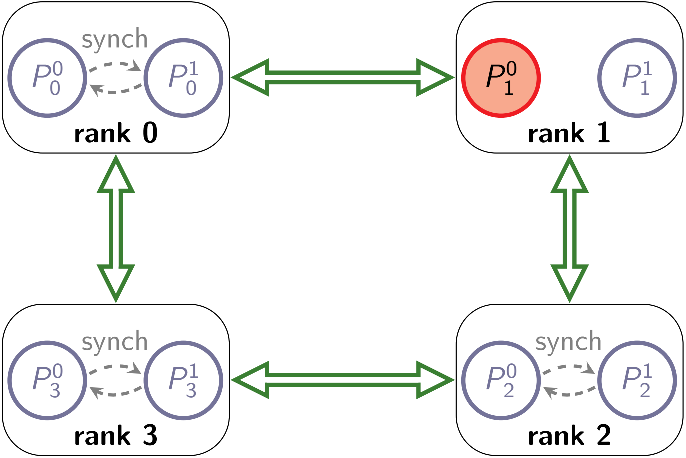
[1] K. Ferreira et al. "Evaluating the Viability of Process Replication Reliability for Exascale Systems". *SuperComputing 2011*.

Active replication[1]



[1]K. Ferreira et al. "Evaluating the Viability of Process Replication Reliability for Exascale Systems". *SuperComputing 2011*.

Active replication[1]



[1]K. Ferreira et al. "Evaluating the Viability of Process Replication Reliability for Exascale Systems". *SuperComputing 2011*.

Active replication

In the crash failure model

- Minimum overhead: 50% (2 replicas of each process)
  - It is actually possible to do better!
- Failure management is transparent
- Synchronization: less than 5% for send-deterministic applications

It could be of interest to deal with **silent errors**

Algorithmic-based fault tolerance (ABFT)

Idea

- Introduce **information redundancy** in the data
  - Maintain the redundancy during the computation
- In the event of a failure, reconstruct the lost data thanks to the redundant information
- Complex but very efficient solution
  - Minimal amount of replicated data
  - No rollback

User-Level Failure Mitigation (ULFM)

Context: Evolution of the MPI standard (fault tolerance working group)

Idea

- Make the middleware fault tolerant
  - The application continues to run after a crash
- Expose a set of functions to allow taking actions at the user level after a failure:
  - Failure notifications
  - Checking the status of components
  - Reconfiguring the application

Conclusion

- Many solutions with different trade-offs
  - Reference: Survey by Elnozahy *et al*[1]
- A still active research topic
- Specific solutions are required
  - Adapted to extreme scale supercomputers and applications

[1] E. N. Elnozahy et al. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems". *ACM Computing Surveys* 34.3 (2002), pp. 375–408.