Distributed Algorithms and Fault Tolerance
Fault tolerance in distributed systems

Thomas Ropars
thomas.ropars@imag.fr

ERODS research team – LIG/IM2AG/UGA

2015
Agenda

Introduction

Checkpointing

Message logging

Replication
Distributed memory vs shared memory

Two communication models for multi-processes applications:

- **Shared memory**: Processes communicate through read/write to a shared address space.
- **Distributed memory**: Processes communicate by sending/receiving messages over communication channels
  - Also called message-passing system.
Distributed memory vs shared memory

Two communication models for multi-processes applications:

- **Shared memory**: Processes communicate through read/write to a shared address space.
- **Distributed memory**: Processes communicate by sending/receiving messages over communication channels
  - Also called message-passing system.

This lecture is about fault tolerance in distributed memory systems
Execution model

- A fix set of processes $P = \{p_0, p_1 \ldots p_n\}$
- A set of one-way channels $C$ connecting processes. The channel $c_{ij}$ is the channel from process $p_i$ to process $p_j$.
Execution model

- A fix set of processes $P = \{p_0, p_1 \ldots p_n\}$
- A set of one-way channels $C$ connecting processes. The channel $c_{ij}$ is the channel from process $p_i$ to process $p_j$.

![Diagram showing processes and channels](image)

Designing a FT technique/protocol requires first to define a failure model.

- We consider a **fail-stop failure model** for processes
  - Processes fail by crashing
The execution of a process can be modeled as a sequence of events. Events can be related to communication or be internal events:

- \texttt{send(m)}
- \texttt{recv(m)}
- \texttt{internal(p)}
Lamport Happened-before relation

- noted →
- Events on one process are totally ordered
  - If e, e' ∈ H(p), then e → e' or e' → e
  - H(p) = history of process p
- send(m) → recv(m)
- Transitivity
  - if e → e' and e' → e'', then e → e''
Events of an execution are partially ordered by Lamport’s happened-before relation:
Events of an execution are partially ordered by Lamport’s happened-before relation:

- \( \text{recv}(m_1) \rightarrow \text{send}(m_6) \)
- \( \text{recv}(m_2) \parallel \text{send}(m_3) \)
Dealing with failures in distributed applications

Problems
▶ How to replay missing messages? ($m_0, m_2, \ldots$)
▶ How can we be sure that $p_1$ will follow the same execution path?
▶ Are messages sent by $p_1$ still valid?
Dealing with failures in distributed applications

Problems

▶ How to replay missing messages? ($m_0, m_2; \ldots$)
▶ How can we be sure that $p_1$ will follow the same execution path?
▶ Are messages sent by $p_1$ still valid?
Problems

▶ How to replay missing messages? \( m_0, m_2; \ldots \)
▶ How can we be sure that \( p_1 \) will follow the same execution path?
▶ Are messages sent by \( p_1 \) still valid?
Dealing with failures in distributed applications

Problems

- How to replay missing messages? (m_0, m_2; \ldots)
- How can we be sure that p_1 will follow the same execution path?
  - Are messages sent by p_1 still valid?
Backward error recovery (rollback-recovery)

**Basic solution**
Restart all processes from the beginning

- Efficiency?
Basic solution
Restart all processes from the beginning
  ▶ Efficiency?

Rollback recovery
  ▶ Checkpointing protocols
  ▶ Message logging protocols
Agenda

Introduction

Checkpointing

Message logging

Replication
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.
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 $e'$:

$$e' \in C \text{ and } e \rightarrow e' \Longrightarrow e \in C$$
Consistent global state

Cut $C_0$: Consistent?

Cut $C_1$: Consistent?

Cut $C_2$: Consistent?
Consistent global state

▶ Cut $C_0$: Consistent? Yes
▶ Cut $C_1$: Consistent?

▶ Cut $C_2$: Consistent?
Consistent global state

- Cut $C_0$: Consistent? Yes
- Cut $C_1$: Consistent?
- Cut $C_2$: Consistent?
Consistent global state

- Cut $C_0$: Consistent? Yes
- Cut $C_1$: Consistent? No
  - Message $m_3$ is an orphan message ($\text{recv}(m) \in C$, $\text{send}(m) \not\in C$)
  - A consistent state is one with no orphan messages
- Cut $C_2$: Consistent?
Consistent global state

- **Cut $C_0$:** Consistent? Yes
- **Cut $C_1$:** Consistent? No
  - Message $m_3$ is an orphan message ($\text{recv}(m) \in C$, $\text{send}(m) \not\in C$)
  - A consistent state is one with no orphan messages
- **Cut $C_2$:** Consistent?
consistent global state

▶ Cut $C_0$: Consistent? Yes
▶ Cut $C_1$: Consistent? No
  ▶ Message $m_3$ is an orphan message ($\text{recv}(m) \in C,$ $\text{send}(m) \notin C$)
  ▶ A consistent state is one with no orphan messages
▶ Cut $C_2$: Consistent? Yes
Uncoordinated checkpointing

Idea
Save checkpoints of each process independently.

Problem

▶ Is there a guaranty that we can find a consistent state after a failure?
▶ Domino effect
▶ Cascading (unbounded) rollbacks on all processes

If process $p_2$ fails, the only consistent state we can find is the initial state.

2015
Idea
Save checkpoints of each process independently.

Problem

▶ Is there a guaranty that we can find a consistent state after a failure?
▶ Domino effect
▶ Cascading (unbounded) rollbacks on all processes

If process $p_2$ fails, the only consistent state we can find is the initial state.
Uncoordinated checkpointing

Idea
Save checkpoints of each process independently.

Problem
- Is there a guaranty that we can find a consistent state after a failure?
- Domino effect
  - Cascading (unbounded) rollbacks on all processes
  - If process $p_2$ fails, the only consistent state we can find is the initial state
Uncoordinated checkpointing

**Comments**

- Algorithms to compute the recovery line have been proposed.
- Garbage collection is very inefficient
  - Hard to decide when a checkpoint is not useful anymore
  - Many checkpoints may have to be kept in storage
- Output commit problem
  - When can a message be sent to the outside world?
Coordinated checkpointing

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

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

Advantages

Drawbacks
Coordinated checkpointing

**Advantages**

- Limited extend of rollbacks in time
- Efficient garbage collection: Only the last checkpoint needs to be kept
- Output commit: All outputs before the last consistent checkpoint can be sent
- Simple to implement

**Drawbacks**
Coordinated checkpointing

**Advantages**

- Limited extend of rollbacks in time
- Efficient garbage collection: Only the last checkpoint needs to be kept
- Output commit: All outputs before the last consistent checkpoint can be sent
- Simple to implement

**Drawbacks**

- Unlimited extend of rollbacks in space
  - When a process crashes, all processes have to restart from the last checkpoint
- Performance cost of coordinating?
  - The coordination protocol
  - Saving the state of all processes at the same time
Blocking coordinated checkpointing

1. The initiator broadcasts a checkpoint request to all processes
Blocking coordinated checkpointing

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

```
checkpoint
request
p0

ack

ack

p1

p2
```

2015
Blocking coordinated checkpointing

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.
Blocking coordinated checkpointing

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

Diagram:

- checkpoint request
- ok
- ack
- p0
- p1
- p2
- 2015
Blocking coordinated checkpointing

**Correctness**

Does the global checkpoint corresponds to a consistent state, that is a state with no orphan messages?
Blocking coordinated checkpointing

Correctness
Does the global checkpoint corresponds to a consistent state, that is 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:

\[
\text{send}(m) \notin C \text{ and } \text{recv}(m) \in C
\]

- It means that \( m \) was sent after receiving \( ok \) by \( p_i \)
- It also means that \( m \) was received after receiving \textit{checkpoint} by \( p_j \)
- It implies that:

\[
\text{recv}(m) \rightarrow \text{recv}_j(\text{ckpt}) \rightarrow \text{recv}_i(ok) \rightarrow \text{send}(m)
\]
Assuming FIFO channels:

1. The initiator takes a checkpoint and broadcasts a checkpoint request to all processes.
Non-blocking coordinated checkpointing
Chandy, Lamport (1985): distributed snapshot algorithm

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).
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) **broadcasts 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**.
Non-blocking coordinated checkpointing
Chandy, Lamport (1985): distributed snapshot algorithm

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

Note that the full protocol also captures channels state
Agenda

Introduction

Checkpointing

Message logging

Replication
Message logging

Basic idea

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

Consistent state
Message logging

Basic idea

- Log all non-deterministic events during failure-free execution
  - Non-deterministic events are mainly message receptions
- 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 follow 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 delivery order of each message
▶ The content of the messages
  ▶ The content can be stored in the memory of the sender

Combination with checkpointing

▶ Checkpointing can be used to:
  ▶ Reduce the extent of rollbacks
  ▶ Reduce the size of the logs
▶ Uncoordinated checkpointing can be used (No risk of domino effect)
Message logging

Advantages

▶ Only the failed process has to restart
▶ No need to coordinate checkpoints
▶ Messages to the outside world (output commit) can be sent immediately

Drawbacks

▶ Storage space
▶ Performance
▶ Think about the cost to synchronize with a reliable storage for every message reception
Message logging

Advantages

- Only the failed process has to restart
- No need to coordinate checkpoints
- Messages to the outside world (output commit) can be sent immediately

Drawbacks
Message logging

Advantages

▶ Only the failed process has to restart
▶ No need to coordinate checkpoints
▶ Messages to the outside world (output commit) can be sent immediately

Drawbacks

▶ Storage space
▶ Performance
  ▶ Think about the cost to synchronize with a reliable storage for every message reception
Rollback-recovery techniques applicability

Rollback-recovery techniques are well suited for compute-intensive applications:

▶ Require very little additional resources
▶ Overhead during failure-free execution is low

What about applications with frequent interactions with the users (outside world)?
Rollback-recovery techniques applicability

Rollback-recovery techniques are well suited for compute-intensive applications:

- Require very little additional resources
- Overhead during failure-free execution is low

What about applications with frequent interactions with the users (outside world)?

- Coordinated checkpointing: would have to checkpoint too frequently
- Message logging: long recovery time after a failure
Rollback-recovery techniques applicability

Rollback-recovery techniques are well suited for compute-intensive applications:

▶ Require very little additional resources
▶ Overhead during failure-free execution is low

What about applications with frequent interactions with the users (outside world)?

▶ Coordinated checkpointing: would have to checkpoint too frequently
▶ Message logging: long recovery time after a failure

The alternative is replication
Agenda

Introduction

Checkpointing

Message logging

Replication
Multiple replicas of a server able to process requests
Replication principle

- Multiple replicas of a server able to process requests
- The failure of one replica is transparent for the client
  - Here the client is the *outside world*
Replication: First try

I want to replicate a register

- An object that can contain a single value
- Clients can read or write this value

First try

- Create 2 replicas ($R_0, R_1$) of the register
- Read/Write: Access one replica, and apply the operation there

Correct?
Replication: First try

I want to replicate a register

- An object that can contain a single value
- Clients can read or write this value

First try

- Create 2 replicas \((R_0, R_1)\) of the register
- Read/Write: Access one replica, and apply the operation there

Correct?

- One client write 1 to replica \(R_0\)
- Later, another client read from replica \(R_1\) and gets value \ldots 0 :-(
Case of the register:

- A read should return the most recent value written
Consistency criteria

Case of the register:

- A read should return the most recent value written
- How to define the "most recent value written"?

Example

- Initial value $x = -1$
- Process $p$ writes 0 to $x$. Later it reads $x$ and gets value 0.
- Process $q$ writes 1 to $x$. Later it reads $x$ and gets value 1.

How to decide whether it is correct?
Definition (home made)

An execution is linearizable if it can correspond to a sequential execution where each operation appears to occur at a time $t$ which is between the actual start and end of the operation.

Linearizability is the strongest consistency criteria.
Linearizability

$p_0$  

draw 0

$p_1$

draw 1

sequential

Linearizable?
Linearizability

$p_0$  write 0  read 0

$p_1$  write 1  read 1

sequential

Linearizable? YES
Linearizability

\[ p_0 \quad \text{write 0} \quad \text{read 0} \quad \text{sequential} \quad \text{read 1} \quad \text{write 1} \quad p_1 \]

Linearizable?
Linearizability

Linearizable? NO
Ensuring linearizability

What is the solution to ensure linearizability of a replicated object?
Ensuring linearizability

What is the solution to ensure linearizability of a replicated object?

- Having all replicas processing all requests in the same order

For replicated registers (object with a read/write interface), a weaker (less expensive) solution can be used:

- Solution based on overlapping quorums
- Not discussed today
Two techniques:
- Passive replication
- Active replication
Active replication

A set of *active* replicas that are all ready to process requests from clients.

The client can use the first answer (no byzantine error).
Active replication

How to ensure linearizability when multiple clients send requests at the same time?

▶ Use atomic broadcast (Total-order broadcast)
▶ Ensures that all replicas process requests in the same order
▶ Technique also called state-machine replication

Bad news
▶ Atomic broadcast and consensus are equivalent problems
▶ Solving one = solving the other
▶ There exists no deterministic algorithm that solves consensus in an asynchronous system with reliable channels if one single process may crash (FLP impossibility result)
▶ Other assumptions need to be made (failure detector, partial synchrony).
Active replication

How to ensure linearizability when multiple clients send requests at the same time?

▶ Use atomic broadcast (Total-order broadcast)
▶ Ensures that all replicas process requests in the same order
▶ Technique also called state-machine replication
Active replication

How to ensure linearizability when multiple clients send requests at the same time?

- Use atomic broadcast (Total-order broadcast)
- Ensures that all replicas process requests in the same order
- Technique also called state-machine replication

Bad news

- Atomic broadcast and consensus are equivalent problems
  - Solving one = solving the other
- There exists no deterministic algorithm that solves consensus in an asynchronous system with reliable channels if one single process may crash (FLP impossibility result)
  - Other assumptions need to be made (failure detector, partial synchrony).
Passive replication

A primary replica receives the requests, sends updates to the backups and answers the client.

The primary has to wait for acknowledgments before answering client’s requests.
Passive replication: Dealing with failures

The failure occurs before the primary has sent updates to the backups:

The failure occurs after the primary has sent updates:
Passive replication: Dealing with failures

The failure occurs before the primary has sent updates to the backups:

▶ The client will time-out and resend its request

The failure occurs after the primary has sent updates:
Passive replication: Dealing with failures

The failure occurs before the primary has sent updates to the backups:
- The client will time-out and resend its request

The failure occurs after the primary has sent updates:
- The client will time-out and resend its request
- Some replicas might have applied the update but not all
  - Problem: The request should be executed only once!
- Problem boils down to solving consensus
First comments

- Active replication might be more attractive since failures can be made transparent to the clients.
- Passive replication might be less energy consuming since backups are mostly idle.

About determinism
Active vs passive replication

First comments

- Active replication might be more attractive since failures can be made transparent to the clients
- Passive replication might be less energy consuming since backups are mostly idle.

About determinism

- Active replication requires the application to be deterministic
  - Problem? Multi-threaded applications are non deterministic
- Passive replication can deal with non-determinism
References


Other sources:
- Lecture notes of D. Sorin
- Lecture notes of A. Schiper