◆ **The problem:** given a distributed system, make it robust to failure of its nodes.

◆ **Motivations:**

   ➣ Distributed systems are becoming increasingly more used.  
     Because:
     ➣ Performance advantage. But,
     ➣ Higher failures probability.

   ➣ Failure are difficult to avoid. Therefore their effect should be transparently corrected.
◆ Transactions (data oriented applications)

◆ Group Communication (communication oriented applications)

◆ **Rollback-Recovery** (recovery at the system level)

  ➡ Checkpoint based
    ➤ Coordinated
    ➤ Uncoordinated
    ➤ Communication induced

  ➡ Log based
    ➤ Pessimist
    ➤ Optimist
    ➤ Causal
Components

◆ Processes connected by a network. 2 flavors:
  - Reliable communication channels (FIFO).
  - Unreliable communication channels.

If we assume unreliable communication channels some aspects of a failure can be masked as failures of the unreliable communication channels.

◆ Stable storage. Assumption: stable storage survives any failure the rollback-recovery system has to survive.

◆ Outside world process (OWP): represents the outside world. 2 special properties:
  - it never fails;
  - it cannot rollback or resend a message.
◆ **Checkpoint**: state of a process.

◆ **Deterministic event**: sent message.

◆ **Nondeterministic event**: received message.

◆ **State interval**: time interval from a nondeterministic event and another nondeterministic event received by a process.

◆ **Process execution**: sequence of state intervals.

◆ **Global state**: the union of the state of each process in the system.
◆ **Consistent state:** every message is in the state of sender and receiver.

![Diagram](image)

◆ **The goal:** restore a consistent state after a failure(s). This consistent state doesn’t have to be one in which the system was before the failure but one that is consistent with the interactions with:

- other processes;
- outside world.)
◆ **In transit messages:** messages sent but not yet received. In case of:

- **un**reliable communication channels: nothing to do, failure is the same as a normal communication failure.

- **reliable** communication channels: in-transit messages lost because of a process failure has to be handled by the recovery system.

◆ **Orphan message/process:** messages received but not transmitted (processes that have received one of such messages).

◆ **Piecewise deterministic assumption (PWD):** nondeterministic events are completely identifiable and re-playable.
◆ **Determinant:** everything needed to replay a nondeterministic message.

◆ **Maximum recoverable state:** is the latest consistent global state.

◆ **Recovery line:** is the set of most recent checkpoints that is a consistent global state.

◆ **Garbage collection:** here used to remove from stable storage useless information.

◆ **Domino effect:** cascade rollback till the beginning of the execution.
**Dependency graph:**

- The vertex are the checkpoints. The edges are the dependencies.

- $c_{j,n}$: $n$-th checkpoint made by the $j$-th process.

- $I(j, n)$: the interval of time between checkpoint $c_{j,n-1}$ and $c_{j,n}$.

The following figure shows an example of dependency graph in red (called rollback-dependency graph, variant in blue called checkpoint graph):
Rollback-Recovery Algorithms

Where we are

Rollback-Recovery

- Checkpoint based
  - Coordinated
  - Uncoordinated
  - Communication induced

- Log based
  - Pessimist
  - Optimist
  - Causal
Operation during normal work:

1. Periodically the state of a process is saved as a checkpoint on stable storage.

Operation during failure recovery:

1. The failed processes compute the recovery line;
2. Everyone rollbacks to the recovery line.
◆ Advantages:
   ➤ It doesn’t depend on the PWD assumption.
   ➤ Simple.

◆ Disadvantages:
   ➤ Depending on the flavor it can be sensible to the domino effect.
   ➤ Not a good choice for highly interactive applications (store the full state after every interaction with the OWP).

◆ 3 flavors:
   ➤ Uncoordinated
   ➤ Coordinated
   ➤ Communication-induced
◆ Operation during normal work

♫ Each process decides autonomously (i.e. independently from the other processes) when to take a checkpoint.

♫ Each process keeps track of the dependency with the checkpoints of other processes.
◆ Operation during failure recovery

▶ The recovering process broadcasts a dependency request message. Every process receiving this message will answer with the dependencies.

▶ Upon reception of the dependencies the recovering process computes the recovery line and broadcasts a rollback request message (dependency graph).
◆ Advantages:
  ➫ Very simple
  ➫ Maximum autonomy of each process in deciding when to take the checkpoint (efficiency)

◆ Disadvantages:
  ➫ Possibility of domino effect as shown by the following figure:
Uncoordinated Checkpoint (continued)

Legend:
- Red: Failure
- Black: Propagation
- White: Recovery line

Rollback-dependency graph:

Checkpoint graph:
Possibility of useless checkpoints (never part of a global consistent state)

Can be necessary to keep multiple checkpoints for each process (because there is no coordination). Garbage collection on the stable storage has to be used to remove the ones not anymore useful.

Absolutely inadequate for interactive applications.
Operation during normal work: 4 flavors

1. Blocking checkpoint coordination
2. Non-blocking checkpoint coordination
3. Checkpoint with synchronized clocks
4. Minimal checkpoint coordination
◆ Blocking checkpoint coordination

- Initiator process (coordinator) broadcasts a checkpoint request (when? periodically or before interactions with OWP).

- A process that receives a checkpoint request flushes the communication channels, takes the checkpoint without committing it and answers with an ACK to the coordinator.

- After the coordinator receives an ACK from everyone it broadcasts a commit checkpoint request.

- A process that receives a commit checkpoint request saves on stable storage the checkpoint that it did before sending the ACK.
Non-blocking checkpoint coordination

As shown in the following figure the problem is to avoid to take inconsistent checkpoints:

Legend:
- - - checkpoint request
- - message
There are two solutions, one for FIFO channels and the other for any type of communication channel:

Solution 1 (FIFO channels):
- Coordinator
- Piggybacked message
- Message

Solution 2:
- Coordinator
- Piggybacked message
- Message

Legend:
- - - - checkpoint request
- - message
- - - - piggybacked message
◆ Checkpoint with synchronized clocks
   ➡ All the processes have synchronized clocks.
   ➡ At well defined times all the processes take a checkpoint without the need of exchanging any message.
   ➡ After taking the checkpoint each process has to wait the sum of the maximum possible time difference between clocks and the maximum time necessary to identify a failure, before starting normal operations.
◆ Minimal checkpoint coordination

❖ Coordinated checkpoint can require a lot of synchronization messages.

❖ But not every process has to take a checkpoint: only those that have had communications with the process that request the checkpoint (initiator) should take a checkpoint.

❖ 2 phases protocol:

**First phase:** the checkpoint initiator send a checkpoint request to all the processes with whom it had communications after its last checkpoint. All the processes receiving a checkpoint request do the same.

**Second phase:** When no more messages have to be sent the processes that have received a message take a checkpoint.
Operation during failure recovery

Upon receiving a rollback message each process rollbacks to its respective checkpoint stored on stable storage.

Advantages:

- No domino effect.
- Easier garbage collection and small storage overhead.
- Easy recovery.

Disadvantages:

- Even if theoretically feasible to use coordinate checkpoint methods with interactive applications it is still too costly (not true anymore).
Based on the notion of Z-cycle and Z-path: two checkpoints \( c_i,x \) and \( c_j,y \) are connected by a Z-path if:

\[
x < y \text{ and } i = j \quad \text{(i.e. } y \text{ is a checkpoint taken after } x \text{ by the same process). Or,}
\]

\[
\exists \text{ a sequence of messages } m_1, \ldots, m_n \text{ such that}
\]

\[
\begin{align*}
\bullet &\quad m_1 \text{ is sent by } P_i \text{ after the checkpoint } c_i,x, \\
\bullet &\quad m_n \text{ is received by } P_j \text{ before the checkpoint } c_j,y, \\
\bullet &\quad \forall n > i > 1 \text{ either } m_i \text{ and } m_{i+1} \text{ happens in the same interval between checkpoints on the same process or } m_i \\
&\quad \text{is received before } m_{i+1} \text{ is sent by the same process.}
\end{align*}
\]
◆ A Z-cycle is a Z-path starting and ending with the same checkpoint.

◆ A checkpoint is useless if and only if it is part of a Z-cycle.

◆ These protocols have two types of checkpoints:
  ➪ **local**: the process decides autonomously to take this checkpoint.
  ➪ **forced**: the process has to take this checkpoint to avoid Z-cycles.
Operation during normal work: 2 flavors

- Model-based communication induced checkpoint
- Index-based communication induced checkpoint
Model-based communication induced checkpoint

- Using heuristics (called model) it avoids the creation of Z-cycles. Usually pessimistic: takes more checkpoints than necessary (example of heuristic: 1) checkpoint before sending a message (not right) 2) checkpoint before receiving a message not separated by the previous message sending event by a checkpoint).
Communication Induced Checkpoint (cont.)

Index-based communication induced checkpoint

- each checkpoint has associated a timestamp such that if a checkpoint $c_{i,x}$ happens before a checkpoint $c_{j,y}$ then the first has a timestamp that is smaller than the second.

- Piggyback to each message the current timestamp.

- A process takes a checkpoint after receiving a message with a timestamp greater than the current process timestamp.
Operation during failure recovery

- Upon receiving a rollback message each process rollback to their respective checkpoint stored on stable storage.

Advantages:

- No domino effect
- No coordination required
◆ Rollback-Recovery
  ➡ Checkpoint based
    ➪ Coordinated
    ➪ Uncoordinated
    ➪ Communication induced

 ➡ Log based
  ➪ Pessimist
  ➪ Optimist
  ➪ Causal
◆ Each process takes periodic checkpoints. Between two checkpoints each process stores the determinant of each nondeterministic event received.

◆ After a failure the processes have to:
  - Recover the last checkpoint.
  - Replay the determinant of each nondeterministic event received after the checkpoint to which the process has rollback.
always-no-orphan property: A process becomes orphan after a failure (of some process) if it doesn’t fail but its state depends on a nondeterministic event that is not stored on stable storage and that is not in the volatile memory of another surviving process. In other words:

\[ \neg Stable(e) \rightarrow (Depend(e) \subseteq Log(e)) \]

where \( Stable(e) \) is true if the nondeterministic event \( e \) is stored on stable storage. \( Depend(e) \) is the set of processes that depend on \( e \) (i.e. that can become orphans) and \( Log(e) \) is the set of processes that have logged on their volatile memory \( e \).
Log-based Rollback Recovery

 Advantages:

- Good for interactive applications
- Only one checkpoint for process required on stable storage
- Depending on the flavor it can be able to reconstruct the exact state before failure
- No domino effect

 Disadvantages:

- Needs the PWD assumption.

 Flavors:

- Pessimistic
- Optimistic
- Causal-based
◆ Pessimistic because: log the determinant of every nondeterministic message (pessimistic assumption: failures are frequent).

◆ Pessimistic version of the always-no-orphan property:
  \( \neg Stable(e) \rightarrow (|Depend(e)| = 0) \).

◆ Operation during normal work
  ➔ Log determinant of every nondeterministic event.
  ➔ Periodically each process takes a checkpoint to limit the amount of messages that have to be replayed in case of a failure.

◆ Operation during failure recovery
  ➔ Rollback to the most recent checkpoint and replay the stored log after that checkpoint.
Advantages:

- Simple management of nondeterministic messages (good approach for interactive applications).
- Simple garbage collection.
- Failures don’t affect any other process except those that failed (others don’t have to rollback).
◆ Disadvantages:

- Higher overhead in case of failure free operations.
- Special hardware for the stable storage (solid state non-volatile memories, special bus for logging).
- Log messages only when the process communicates with others (problems in case the protocol assumes reliable channels).

\[ \neg Stable(e) \rightarrow (|Depend(e)| \leq 1) \]
◆ Major difference with pessimistic logging: doesn’t force a synchronous commit of the log to stable storage but it locally buffers it for some time (asynchronous logging).

◆ In case of failure the information stored in the local log is lost so it is possible to create (temporarily) orphan processes as shown in the following figure:
Operation during normal work

- Each process takes local checkpoints periodically plus it stores the determinat of nondeterministic events in a local log.
- Periodically flushes the local log to stable storage
- Track dependency with other processes to be able to detect orphan processes during recovery (dependency among state intervals).
Operation during failure recovery: 2 flavors

- Synchronous
- Asynchronous
ёт Synchronous: first: all collect dependency information to
discovery where to rollback. Second: all rollback. 2
dependency schemes:
1. Direct dependency tracking: each process piggyback
outgoing messages with the current process’ state
interval.
2. Transitive dependency tracking: each process $i$
piggybacks outgoing messages with a vector of indexes
$TD_i$ ($TD_i[i] = \text{state interval of } i, TD_i[j] = \text{most}
recent state interval on which } TD_i[i] \text{ depends}$.
Asynchronous:

- Recovering process: 1) broadcasts a rollback announcement. 2) rollback (new incarnation of itself).
- Process receiving a rollback announcement: 1) decides if the rollback announcement makes it orphan. 2) If yes, it rollbacks (starts itself in a new incarnation) and broadcast another rollback announcement message.

Messages from more incarnations of the same process can circulate at a given time.

**Exponential rollback:** indicates the pathological case of a process that produces an exponential number of rollback announcement messages.
Advantages:

- Lower overhead than pessimistic logging during failure-free operation.

Disadvantages:

- More complicated garbage collection and recovery algorithms than pessimistic logging.
The concept is to log only those messages on which the processes that \textbf{will} not fail or the OWP depend on.

These dependency information are stored in the volatile memory of a process (if not related to OWP) or on stable storage (if related to OWP).
◆ Operation during normal work
  ➫ Each process build a dependency graph in which it represents the causal relations on which its state depends (and the determinants).
  ➫ This information are piggybacked on the messages sent so that the receiver can update its dependency information.
Operation during failure recovery

The processes that fail use the information contained in the volatile memories of the processes that haven’t failed or on stable storage to replay their execution as shown by the following figure:

![Dependency graph of P₀](image)

Dependency graph of P₀:
Advantages:

- It combines the advantages of pessimistic and optimistic logging without their disadvantages.

Disadvantages:

- Naive implementation: the size of the graph piggybacked can be high. But it can be optimized.
<table>
<thead>
<tr>
<th></th>
<th>Checkpoint-Based</th>
<th>Log-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Uncoordinated</td>
<td>Coordinated</td>
</tr>
<tr>
<td>Needs PWD</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Chkpts per Process</td>
<td>1+</td>
<td>1</td>
</tr>
<tr>
<td>Domino effect possible</td>
<td>p</td>
<td>np</td>
</tr>
<tr>
<td>Orphans possible</td>
<td>p</td>
<td>np</td>
</tr>
<tr>
<td>Rollback extension</td>
<td>u</td>
<td>1</td>
</tr>
<tr>
<td>Recovery Data</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>Recovery Protocol</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>Output Commit</td>
<td>no</td>
<td>g</td>
</tr>
</tbody>
</table>
◆ No real difference between coordinated and uncoordinated checkpoint.

◆ Checkpoint can be accelerated using copy-on-write model (concurrent checkpointing).

◆ Store only the blocks that change (incremental checkpointing). Expensive, so probabilistic difference approach (signature based - proven to be inaccurate in practice).

◆ User (not everything available) / Kernel (everything available but not portable) / Compiler (difficult but potentially more efficient) level implementations.
◆ Logging was usually seen as a way to remove domino effect to uncoordinated checkpointing.

◆ Problems with nondeterministic events: interrupts (store instruction counts). System calls (pid, time, malloc).

◆ Unambiguous naming (for migration transparency).