Large Scale Self-Stabilization

Pierre Lemarinier

Friday lunch, march 14th
Resume

- Postdoc in INRIA on the QosCos Grid European project

- Parallelism team, LRI, University of Paris XI

- Ph.D. defended on November 2006 (fault tolerance in MPI)

MPICH
MPI Implementation for Volatile resources
MPICH-V objectives

Goal: Fault tolerance for existing and new MPI applications

Problems:  1) volatile nodes (any number at any time)
           2) non named receptions

Objectives summary for each protocol:
  1) Automatic fault tolerance
  2) Transparency for the programmer & user
  3) Tolerate up to n simultaneous faults
      (n being the #MPI processes)
  4) Scalable Infrastructure/protocols

Objectives summary of this thesis:

Evaluating and comparing rollback recovery protocols for MPI application in a fair way based on:

✓ Fault frequency
✓ Scalability
✓ Impact on raw performance
✓ Impact on real applications
Fault tolerance

• **Different types of failure:** fail-stop or crash, ..., byzantine
  → We will only consider the simplest one: the **crash** of processes

• **Impossibility result** [FLP85]: no non-trivial agreement is possible in asynchronous systems if a crash failure may appear
  → We will consider asynchronous systems with **failure detector** [CT96, DGFG02]

• **Different fault tolerance techniques:** replication[Sch90, ISB93], self-stabilization [Dol00], rollback-recovery
  → We will consider only **rollback-recovery** protocols
Contribution

- Formalization of five protocols in a coherent theoretical model
- Implementation of these protocols in a common framework
- Evaluation according to multiple criteria
- First comparison of these protocols

<table>
<thead>
<tr>
<th>Coordinated Checkpointing</th>
<th>Message Logging</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pessimistic log</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Vcl</td>
<td>V1</td>
</tr>
<tr>
<td>Pcl</td>
<td></td>
</tr>
<tr>
<td>Non blocking</td>
<td>blocking</td>
</tr>
</tbody>
</table>

- Remote message logging
- Sender based message logging
- Sender based message logging
- Piggyback of causality information

- Channel memory in the communication path
- Event logger and wait for acknowledge
- Piggyback of causality information

Diagram showing the processes and interactions of the protocols.
What we have learned

- Pessimistic remote message logging protocol (MPICH-V1) → only for desktop Grids (direct communication between nodes is mandatory for high performances on the NAS).

- Remote synchronous pessimistic storage of causality information (MPICH-V2) adds a high latency overhead.

- Remote asynchronous pessimistic storage of causality information (MPICH-VCausal) solves the latency problem, but reduces bandwidth.

- In contrary to general belief, coordinated checkpoint provides very good performances in presence of faults (especially with local checkpoint copies).

- The synchronization time in CL is not the first limiting factor, which is the stress of the checkpoint server during checkpoints and restarts (This stress can be solved by using local copies of checkpoint images).

- Blocking coordinated checkpointing is quite simple to implement and efficient for high performance network systems, but less efficient than non blocking protocol for application running on Ethernet network systems.

- Tested 3 Causal message logging protocols provide similar performance.

- In Causal message logging Event Logger reduces the bandwidth penalty.
Large Scale Self-Stabilization

Thomas Hérault, Pierre Lemarinier, Olivier Pérès
Laurence Pilard, Joffroy Beauquier
Classical Self-Stabilization model

State/transition model:
- Each component has a local state
- Global system state is union of local states
- Behavior of the system is represented by the global state and the transition between the states
- Processes communicate by message passing

Self-stabilization peculiarities:
- Each process has a non-corruptible list of all neighbors
- Transient Byzantine failures: Transient malfunction (message corruption/memory error) can put the system to illegal state

A self-stabilizing system guarantees that:
- Regardless of the current state, the system is guaranteed to recover to a legal configuration in a finite number of steps
- Remain in the legal configuration thereafter until a malfunction occurs
Large scale and Self-Stabilization issues

In contemporary large scale systems
  • ...many processes
  • ...processes communicate via Internet Protocol: virtual complete communication graph

Practical issues
  • Can not anymore consider having a connection opened to each available neighbor.
  • Want to avoid having the list of all available processes in a node memory.

Note: The set of IP addresses can be totally ordered: consider the IP as the process ID.
A new model

Failures
- Stopping failures (crashes)
- All failures captured in initial configuration
- Only few IDs correspond to running processes

Neighbors
- Processes have no neighbor list
- Introduction of two specific services:
  - Oracle for resource discovery
  - Failure detector to deal with stopped processes
Oracle

Features:
• Gives one process ID each time queried
• The corresponding process may be stopped
• If a set \( S \) of processes query the oracle service an infinite number of time, then each process \( s \) of \( S \) obtains the highest ID in \( S \) at least once.

Implementation:
• Local device: one daemon per process
• Communications between daemons through multicast
• Bounded list of ID
• Regularly, every daemon randomly chooses an identifier in its list and multicasts it to all the other daemons
• Every time a daemon receives an identifier (from a process querying for another process, or from the multicast channel), it updates its list using an LRU sorting algorithm
Failure Detector

Features:
- When queried, returns whether a given process is stopped
- Unreliable, in class $\Diamond P$: eventually perfect
- As defined by Chandra and Toueg [CT95]

Implementation:
- Local device: one per process
- Only lookup by heartbeat messages the opened connection
- Initially not suspecting any process death
Case study: spanning tree

Build a spanning tree of degree $p$ (parameter)

Use only a small number of process ID on each process ($p + 1$)

Prove the algorithm and measure its performance
Spanning tree algorithm

Global invariant using order on identifiers: \( \forall p, \forall c \in children(p), c < p \)

Guarded rules
- Eliminate stopped processes
- Enforce global invariant
- Maintain links: \( \text{parent}(p) = q \iff p \in children(q) \)
- Roots search for new processes and contact them

Two heuristics for node insertion:
1) select the highest ID
2) random selection
Spanning tree algorithm

Global invariant using order on identifiers: $\forall p, \forall c \in \text{children}(p), c < p$

Guarded rules
- Eliminate stopped processes
- Enforce global invariant
- Maintain links: $\text{parent}(p) = q \iff p \in \text{children}(q)$
- Roots search for new processes and contact them

Two heuristics for node insertion:
1) select the highest ID
2) random selection
Convergence time

- Convergence times (first heuristics)
- Convergence times (second heuristics)

The graph shows the relationship between the number of processes and the time taken for convergence, with error bars indicating variability.
Node’s average depth

![Graph showing the relationship between the number of processes and the average depth. The graph includes two lines: one for average depth (first heuristics) and another for average depth (second heuristics).]
Depth of the spanning tree

- depth (second heuristics)
- depth (second heuristics) degree 3
- depth (second heuristics) degree 4
- depth (second heuristics) degree 5
- optimal depth

Number of processes vs. Depth
Questions?