Concurrency problems in autonomic systems

Studying synchronization and coordination protocols for multiple autonomic managers

Rémi SHARROCK
Autonomic workshop
Toulouse 16 oct14
Agenda

• Motivation

• Architectures for multiple autonomic managers.

• The concurrency problem

• Protocol for the competitive concurrency

• Protocol for the cooperative concurrency
Architectures for multiple autonomic managers
Autonomic system architectures

Autonomic system examples:
Autonomic system architectures

- One Element may interact with another Element:
  - directly
  - indirectly (stigmergy)

- Relation « INFLUENCES »
Autonomic system architectures

• An Element is Manageable = Touchable if it is Observable + Controllable

• Relation « MANAGES »

• An AM may be Manageable : ME = AM
Autonomic system architectures

- Autonomic system = relations graph
Concurrency problems in autonomic systems
Autonomic systems concurrency problems

• Many architectural patterns:
Autonomic systems concurrency problems

• 3 concurrency types:

Disjoint

- No protocol
- Inconsistencies

Competitive

- Synchronization protocol
- Starvations, deadlocks, Single Point of Failure

Cooperative

- Coordination protocols
- Non-reachable consensus
Protocol for the competitive concurrency - Synchronization protocol
Competitive concurrency: synchronization protocol

- A simple protocol: the TOKEN protocol:

![Token Request](image)

- Token Transfert
- Token Request
- Token Acquired
- Token Release

- AM Manages
- AM Manages
- ME « Shared » Knowledge

![Token Transfert](image)
Cloud computing scenario

- Component-Based Applications:

- Several Internal Configurations

- Several QoS levels

- Infrastructure:
  - Virtualized environments
  - Elasticity
  - Consolidation (VM migration)
Cloud computing scenario

- **Application AM:**
  - QoS self-optimization
  - Manages Application Configurations and component-VM mapping

- **Infrastructure AM:**
  - PM utilization rate self-optimization
  - Manages VM allocation and PM-VM mapping (VM consolidation)
Events and actions for the applications

(a) Workload Increased
- Stop Component
- Unbind Component
- Release VMs

(b) Workload Decreased
- Stop Component
- Unbind Component
- Release VMs

(c) Renting Fees Changed
- Stop Component
- Deploy Component
- Bind Component
- Start Component
- Release VMs

(d) Scale Down
- Stop Component
- Deploy Component
- Bind Component
- Start Component
- Release VMs

(e) Invoke Handler
- Deploy Component
- Bind Component
- Start Component

Endogenous Events
- Public Knowledge Changed Events
- Actions on the Managed System
- Invoke Handler
- Interloop Events
- Interloop Actions (notification)
Events and actions for the infrastructure

(a) Notify VMs Creation
(1) Power on PM
(2) Create VM

(b) Power off PM
(1) Scale Down
(2) Set Timeout
(3) Power-off PM

(c) Energy Shortage

(d) Update Renting Fees
Low PM Utilization

(e) Unused PM

(f) Power off PM
Timeout Expired

Endogenous Events
Change Public Knowledge
Actions on the Managed System
Invoke Handler

Interloop Events
Interloop Actions (notification)
• **TOKEN REQUEST**: every time the renting fees need to be read or modified

• **TOKEN RELEASE**: every time the actual AM no longer need to modify the renting fees

• **TOKEN TRANSFER**: every time an « application AM » performs an interloop action on the « Infrastructure IM » that leads to a Change Renting Fees

• e.g. Request VM interloop action
Protocol for the cooperative concurrency
- Coordination protocol
Cooperative concurrency: coordination protocol

- A coordination protocol difficult to understand:
  - Protocol goal: all AMs must have the same « vision » of the concurrency in order to take cooperative decisions.
  - « vision » = state machine in the Knowledge
  - same « vision » = same state in the state machine
Cooperative concurrency: coordination protocol

- A simple example to understand: having the SAME management
Cooperative concurrency: coordination protocol

- A simple example to understand: having the SAME management

```
• OUPS !
```
Cooperative concurrency: coordination protocol

- Leader to order, ACCEPT messages
Cooperative concurrency: coordination protocol

- Leader to order, ACCEPT messages

• Without ME
Cooperative concurrency: coordination protocol

- Leader to order, ACCEPT messages

- **OUPS!**
  - ACCEPT messages can be reordered
Cooperative concurrency: coordination protocol

- Sequence number, discard out of order

- What about S3 on AM2?
  - buffer?
  - retransmission mechanism?
Cooperative concurrency: coordination protocol

- Sequence number, discard out of order

- OUPS!
  - message lost: AM2 won’t do anything!
Cooperative concurrency: coordination protocol

- Learn stops retransmission
- don’t send new accept until learn
Cooperative concurrency: coordination protocol

- Learn stops retransmission

- with ME visualized
Cooperative concurrency: coordination protocol

- What about AM crash?

- Leader takeover

- PB: a2 then a4 on ME is inconsistent (S2 -> S4 does not exist)
Cooperative concurrency: coordination protocol

- What about AM crash?

Solution: leader waits for learn before changing state
Cooperative concurrency: coordination protocol

• What about AM crash?

• Leader takeover is now safe
Cooperative concurrency: coordination protocol

- What about AM crash?

  - Leader remains in control if the other crashes: no need to wait for Learn messages anymore. AM1 changes to S3 state without waiting after crash detection.

  - Other case: if Learn messages are lost, the accept will be retransmitted: no PB.
Cooperative concurrency: coordination protocol

• What about partial network crash (partition) ?

  - Difficult (interesting !) situation:
  - network link AM1 - ME is OK
  - network link AM2 - ME is OK

• but network link AM1 - AM2 crashes !
Cooperative concurrency: coordination protocol

- What about partial network crash (partition)?
Cooperative concurrency: coordination protocol

- Solution: add another AM, make progress in majority partition
Cooperative concurrency: coordination protocol

- New leader in majority partition

```
<table>
<thead>
<tr>
<th>AM1</th>
<th>K1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LEADER</td>
</tr>
<tr>
<td>AM3</td>
<td>K3</td>
</tr>
<tr>
<td>AM2</td>
<td>K2</td>
</tr>
<tr>
<td>ME</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<ACC, S2, 1> <ACC, S2, 1> <ACC, S2, 1>
```

```
retransmit
```

```
partition
detection
```

```
<ACC, S2, 1> S2
```

```
<LRN, S2>
```

```
e1 e2
```

```
a2 a2
```
Cooperative concurrency: coordination protocol

- What about link crash and AM crash at the same time?
Cooperative concurrency: coordination protocol

- What about link crash and AM crash at the same time?
Cooperative concurrency: coordination protocol

- What about link crash and AM crash at the same time?

```
LEADER
| AM1
| K1
| LEADER
| AM3
| K3
| AM2
| K2
| ME
e1 e2

<ACC, S2, 1>

<ACC, S3, 1>

<LRN, S3>

S3

a3
```
Cooperative concurrency: coordination protocol

• What about link crash and AM crash at the same time?

• AM2 has crashed but it could have executed a2 or a3 and we don’t know anything about it

• We could ask ME ? NO ! We don’t want to rely on ME !
Cooperative concurrency: coordination protocol

- Solution too long to explain graphically but:
  - Round numbers are assigned, 1 leader per round:
    - AM1: 1, 4, 7 ...
    - AM2: 2, 5, 8 ...
    - AM3: 3, 6, 9 ...
  - Identify the leader:
    - <ACC, rnd, STATE, seqno> sent by leader of round rnd
    - <LRN, rnd, STATE> sent to leader of round rnd
  - Explicit leader takeover protocol (distinguish messages from different leaders)
    - <PREP, rnd> request to become leader for round rnd
    - <PROM, rnd, (oldrnd, STATE)> promise not to accept messages from a lower round (from an older leader), STATE was accepted in round oldrnd
  - Leader resends accept for messages identified in the PROM message
    - after receiving the PROM
    - after a partition merge (a partition is recovered)
Cooperative concurrency: coordination protocol

• What about more than 1 crash?

• increase the number of AMs

• to limit progress to a majority partition:
  • we can only tolerate fewer than half of the AM crash
  • to tolerate $f$ crashes we need at least $2f + 1$ AM
Cooperative concurrency: coordination protocol

• What about a combination of messages loss and AM crashes?

• Prevent non-leader AM from executing after receiving an accept

• There are 2 solutions:
  • Wait for all-to-all learn
  • Wait for commit from leader
Concurrency problems in autonomic systems

Studying synchronization and coordination protocols for multiple autonomic managers

Rémi SHARROCK

Autonomic workshop

Toulouse 16 oct14
Cooperative concurrency: coordination protocol

- Wait for all-to-all learn
Cooperative concurrency: coordination protocol

- Wait for commit from leader
endogenous event 1

endogenous event 2

endogenous event 3

waiting queue

exogenous event 4

INTRA Invoke Handler

Event 2 treated

Treq = Token request
Trel = Token release
TA = Token Acquired
TT = Token Transfer

INTRA = Intraloop action
INTER = Interloop action/event
Setup

• V1: Simulator implemented in Java 6
  • Intel Core 2 Duo processor, 4GB RAM, Mac OS X Lion

• V2: Deployed on Grid‘5000
  • Different geographic sites in France
## Setup

<table>
<thead>
<tr>
<th>Class</th>
<th>Workload Increased</th>
<th>Workload Decreased</th>
<th>Low PM Utilization</th>
<th>PM Unused</th>
<th>Energy Shortage</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.05</td>
<td>0.01</td>
</tr>
<tr>
<td>Low</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.02</td>
<td>0.01</td>
</tr>
</tbody>
</table>

- Dataset: 10, 20, 30, 50 and 70 AMs for 1 IM
- Average Token Acquire Time
- Per-event Average Processing Time
Results

stability

scalability

Low

High