Towards Implementing Multi-Layer Reflection for Fault-Tolerance

François Taïani, Jean-Charles Fabre, Marc-Olivier Killijian

Toulouse, France

DSN-2003, San Francisco, CA, USA June 22\textsuperscript{nd}-25\textsuperscript{th}
Modern systems are large and complex
- many software layers and components
- heterogeneous abstraction levels
- increased use of COTS

Dependability is orthogonal to all system layers

Adding fault-tolerance to those systems must be done:
- separately from functional development to address complexity
- encompassing all system layers for maximum coverage

Our proposal: Multi-Layer Reflection
Outline

- What is Reflection?
- Why Multi-Layer Reflection?
- Development Approach
- Case Study: Replication of a Multi-Threaded Server
- Conclusion
What is Reflection?

"the ability of a system to think and act about itself"

- meta-level
- observation
- meta-interfaces
- base-level
- original system
- fault-tolerance
- control
- meta-model
- "generic connector"

 seperating fault-tolerance from functional concerns
Why Multi-Layer Reflection?

- Ad-hoc fault-tolerance in a multi-layer system
Why Multi-Layer Reflection?

- *Ad-hoc* fault-tolerance in a multi-layer system

fault-tolerance
"patches"

*ad hoc* inter-level coordination

*ad hoc* connection
FT code↔original code
Multi-Layer Reflective Architecture

aggregation of meta-information

generic, self-contained meta-interface
Multi-Layer Reflective Architecture

Which information is needed for fault-tolerance?

How and where to obtain this information?
Development Approach

[Multi-Layer Reflective Architecture]

1. family of mechanisms
2. reflective footprint
3. instrumentation of the chosen architecture
Obtaining the Reflective Footprint

- Analysis of a family of replication strategies
  - primary backup replication
  - active and semi-active replication

- Example of reflective features that are needed to implement the mechanisms of this family:
  - state capture (observation)
  - state restoration (control)
  - request message interception (observation)
  - request message dispatching (control)
  - non-deterministic decision points
# Obtaining the Reflective Footprint

**Development Approach**

## Reflective Facets

<table>
<thead>
<tr>
<th></th>
<th>Communication</th>
<th>Execution</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reification</strong></td>
<td>RequestReception</td>
<td>ExecutionPointStart</td>
<td>NonDeterministicPlatformCall</td>
</tr>
<tr>
<td></td>
<td>RequestSending</td>
<td>ExecutionPointEnd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReplySending</td>
<td>ExecutionPointReach</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReplyReception</td>
<td>NonDeterministicFlowChange</td>
<td></td>
</tr>
<tr>
<td><strong>Introspection</strong></td>
<td>getRequestContent</td>
<td>getExecutionPoint</td>
<td>getServerState</td>
</tr>
<tr>
<td></td>
<td>getReplyContent</td>
<td></td>
<td>getPlatformState</td>
</tr>
<tr>
<td><strong>Behavioral Intercession</strong></td>
<td>doSend</td>
<td>createExecutionPoint</td>
<td>forceResultOfPlatformCall</td>
</tr>
<tr>
<td></td>
<td>doReceive</td>
<td>setExecutionPoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>forceResultOfFlowChange</td>
<td></td>
</tr>
<tr>
<td><strong>Structural Intercession</strong></td>
<td>piggyBackDataOnMsg</td>
<td></td>
<td>setServerState</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>setPlatformState</td>
</tr>
</tbody>
</table>
Instrumentation

- In a multi-component system: Information/control possible in different layers / abstraction levels
  - Higher layers (application, language):
    - abstract info / rich semantics
  - Lower layers (OS, middleware):
    - detailed info / poor semantics

- Goal of our approach:
  - to combine the best of both perspectives
  - requires understanding of inter-layer coupling

- We developed a reverse-engineering tool to help us construct model of inter-layer interaction
  - helps decide where to insert instrumentation points
Case Study: Replication & Multithreading

Goal: Transparent replication of a CORBA server
- multi-layer: POSIX (OS) + CORBA (middleware)
- multithreaded: concurrent processing of requests
- thread pool: upper limit on concurrency
Case Study: Replication & Multithreading

- Goal: Transparent replication of a CORBA server
  - multi-layer: POSIX (OS) + CORBA (middleware)
  - multithreaded: concurrent processing of requests
  - thread pool: upper limit on concurrency

- Problem 1: state capture / restoration
  - application state
  - middleware + OS state
Case Study: Replication & Multithreading

- Goal: Transparent replication of a CORBA server
  - multi-layer: POSIX (OS) + CORBA (middleware)
  - multithreaded: concurrent processing of requests
  - thread pool: upper limit on concurrency

- Problem 1: state capture / restoration
  - application state
  - middleware + OS state

- Problem 2: control of non-determinism
  - assumption: multi-threading only source of non-determinism
  - how to replicate non-deterministic scheduling decisions?
No guarantee on middleware behavior:
- arbitrary scheduling of requests by middleware

Replicating scheduling decisions observed in the application is not enough:
- because of thread pool (for example size 2)
- even with total order-multicast on the network

The decision taken by the middleware regarding dispatching can't be controlled from the application.
OS Level Only

- Low level thread synchronization can be controlled:
  - The same thread scheduling can be enforced on all replicas
  - Requests are dispatched and processed in the same order
  - All replicas reach the same state
    (assumption: MT = only source of non-determinism)

- But this over-constrains the replicas' execution:
  - impossible to relate OS level activities to request processing
  - impossible to distinguish scheduling decisions that influence determinism and those that do not.

Not equivalent, replication of every decision
Smart Replication of Scheduling

- With CORBA and application semantics:
  - Application and CORBA reflection give semantic to the actions taken by the application.
  - This semantic allows optimal use of OS level reflection.

- Example: with a thread pool:
  - Which thread executes which request does not matter
  - The following 2 executions are equivalent:

  **no need to replicate this scheduling decision**
[Case Study: Replication & Multithreading]

The Multi-Layer Meta-Model

- Meta-model centered on the lifecycle of a CORBA request
  - aggregates OS-level synchronization and request lifecycle

Meta-model centered on the lifecycle of a CORBA request

- request reception
- request post-processing
- request pre-processing
- OS level synchronization
- sending of reply
Middleware Instrumentation

- Behavioral middleware model:
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model

[Case Study: Replication & Multithreading]
Middleware Instrumentation

- Behavioral middleware model:
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model

RequestBeforeApplication
[Case Study: Replication & Multithreading]

Middleware Instrumentation

- Behavioral middleware model:
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model

RequestBeforeApplication
RequestAfterApplication
[Case Study: Replication & Multithreading]

Middleware Instrumentation

- Behavioral middleware model:
  - relates OS level actions to application level operations
  - identifies points of instrumentation of meta-model

RequestBeforeApplication
RequestAfterApplication
RequestContentionPoint
Replication: The Whole Picture

- Behavioral control
  - interception of request execution life cycle steps
  - non-deterministic contention points can be controlled

- State observation and control
  - Middleware state can be recovered by "fast-reexecution"
    - re-injection of ongoing requests
    - dispatching of active requests to the pool
    - "shunting" execution for requests already processed
  - Application level state: reuse of other approaches
    - language based reflective approach to restore state variable
    - platform based approaches to restore OS dependent application state (e.g. thread stacks)
The Meta-Interface

class Request;
class Thread;
class StackChunk;
class ReifiedEvent;
class RequestLifeCycleEvent extends ReifiedEvent {
    public Request reifiedRequest;
    public Thread reifyingThread;
}
class BeginOfRequestReception extends RequestLifeCycleEvent;
class EndOfRequestReception extends RequestLifeCycleEvent;
class RequestBeforeApplication extends RequestLifeCycleEvent;
class RequestAfterApplication extends RequestLifeCycleEvent;
class BeginOfRequestResultSend extends RequestLifeCycleEvent;
class EndOfRequestResultSend extends RequestLifeCycleEvent;
class RequestContentionPoints extends RequestLifeCycleEvent;

class IntercessionCommand;
class ContinueExecution extends IntercessionCommand;
class SkipCallToApplication extends IntercessionCommand;

interface MetaLevel {
    IntercessionCommand reifyEventToMetaSynchronous(ReifiedEvent e);
}
interface BaseLevel {
    State captureApplicationState();
    void restoreApplicationState(State s);
    StackChunk captureApplicationStack(Thread t);
    void restoreApplicationStack(Thread t, StackChunk stack);
    void InjectRequestAtCommunicationLevel(Request r);
}
Conclusion

- Complex fault tolerant systems:
  - Separation of concerns for reusability, adaptability, evolvability
  - Observability and controllability over multiple layers required

- Multi-layer reflection
  - Consistent and disciplined way to address this problem
  - Applicable to complex systems as it enables to master complexity
  - This is possible: our case study is a first step… more work to come!

- Recent and on-going work
  - DAISY: an adaptive fault tolerant system based on some limited off-the-shelf reflective mechanisms (CORBA PI, Java Serialization)
  - Some observability and controllability problems solved thanks to the multi-layer reflection concepts (e.g. additional reflective features introduced into Orbacus and Linux)
Prospective

- Components and OSS is not enough!

« Reflective component model »