L I G

UNIVERSITE JOSEPH FOURIER
SCIENCES TECHNOLOGIE MEDECINE
GRENOBLE 1

CBSE'10

A Self-healing Component Sandbox for Untrustworthy Third Party Code Execution

Kiev Gama and Didier Donsez
Université Grenoble 1, France
(LIG Laboratory, ADELE Team)
kiev.gama@imag.fr
didier.donsez@imag.fr

Work partially funded by the EC FP7 Aspire Project

ASPIRE
Aspire Today, Inspire Tomorrow

Advanced Sensors and lightweight Programmable middleware
for Innovative Rfid Enterprise applications

Outline

- Motivations
- Techniques
- Approach
- Experiments
  - Domain based x OS-based isolation
  - Fault deployment
- Conclusions

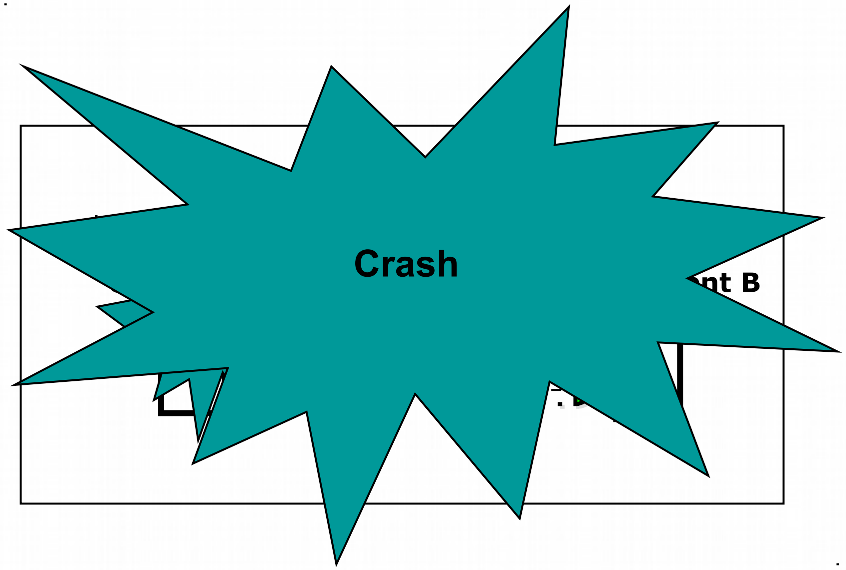
Motivations

- Component based applications **dependability**
- Third party code dynamically deployed
- Provide some sort of isolation for preventing fault propagation
- Not necessarily fault tolerance IN components
- Fault containment to protect underlying application
- Providing self-healing mechanisms to recover from a faulty state

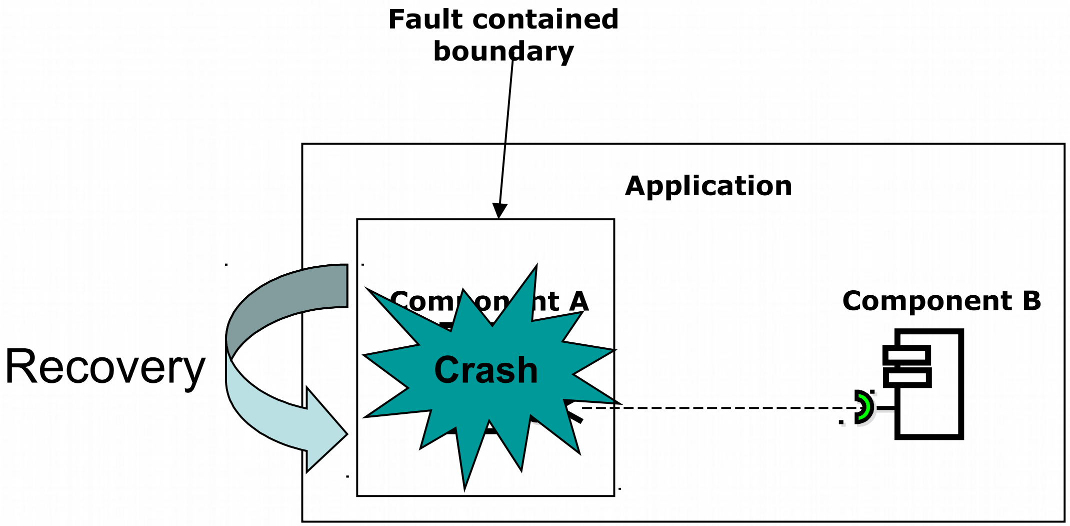
Why?

- “Strength of a composition is defined by its weakest component” [Szyperski]
- We can’t easily predict and test all possible compositions
- Worse in dynamic platforms: we can not even predict what assembly will be deployed
- Need to execute untrustworthy (not necessarily malicious) components but still ensuring system reliability

What we don’t want to have



What we would like to have



Isolation Mechanisms in Java

- Namespace-based
  - Class loader hierarchy enforcing type isolation
  - Pseudo-isolation = No fault containment
- OS-based ✓
  - Uses processes as boundaries
  - Implies inter-process communication (IPC) costs
- Domain Isolation ✓
  - Java Isolates (sort of lightweight processes) defined in JSR-121
  - Implies IPC as well

Self-healing

- Automatic detection, diagnosis and repair of problems
- One of the key concepts in autonomic computing (self-manageable systems)
- Need of
  - Recovery mechanisms
  - Fault detection and forecast

Target Platform

- OSGi Service Platform
- Loose component decoupling through services
- Dependencies:
  - Defined at development time
  - Resolved at runtime
- Components may be installed and uninstalled during application execution

BUT…

- Weak isolation: memory leaks when components are uninstalled (precedent work)
- No fault containment in components

Our Approach

- A sandbox architecture for untrustworthy OSGi components
- A policy for sandboxing in two levels (service and component)
- Initial prototype based on Isolates (domain-based isolation)
  - Patched Apache Felix 1.4.0
  - SunLabs MVM (Multitasking Virtual Machine) with Isolate API
- Port of the previous solution to OS-based isolation

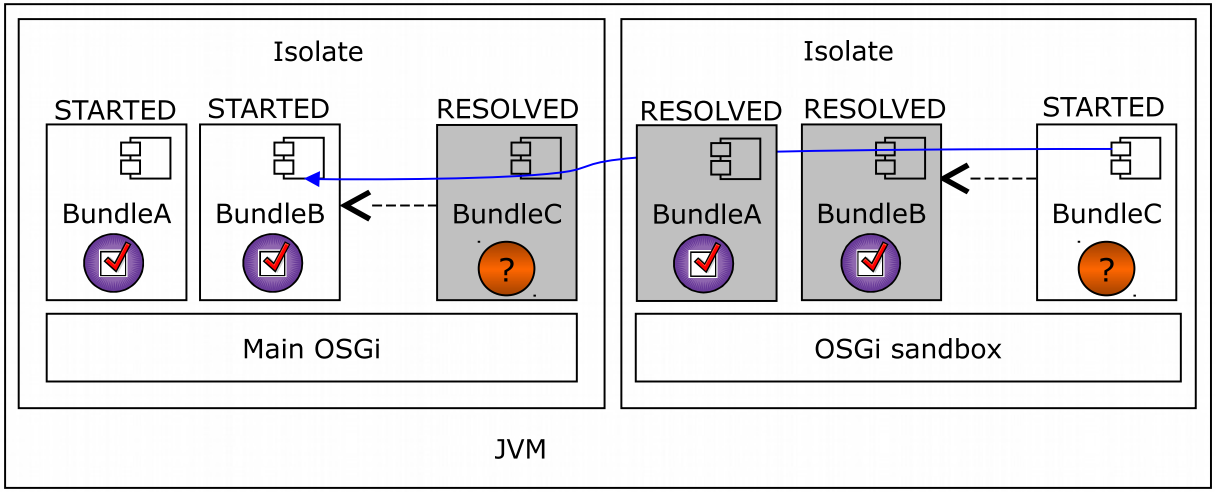
Prototype

- Two OSGi frameworks executing in fault contained boundaries
  - Main OSGi
  - Sandbox OSGi
- Initially implemented with Java Isolates
- Policy defines which components are (not) trustworthy
- Untrustworthy components execute in the sandbox
- Assumption for enabling transparent IPC between platforms
  - Services have methods with primitive types

Techniques used for Self-healing

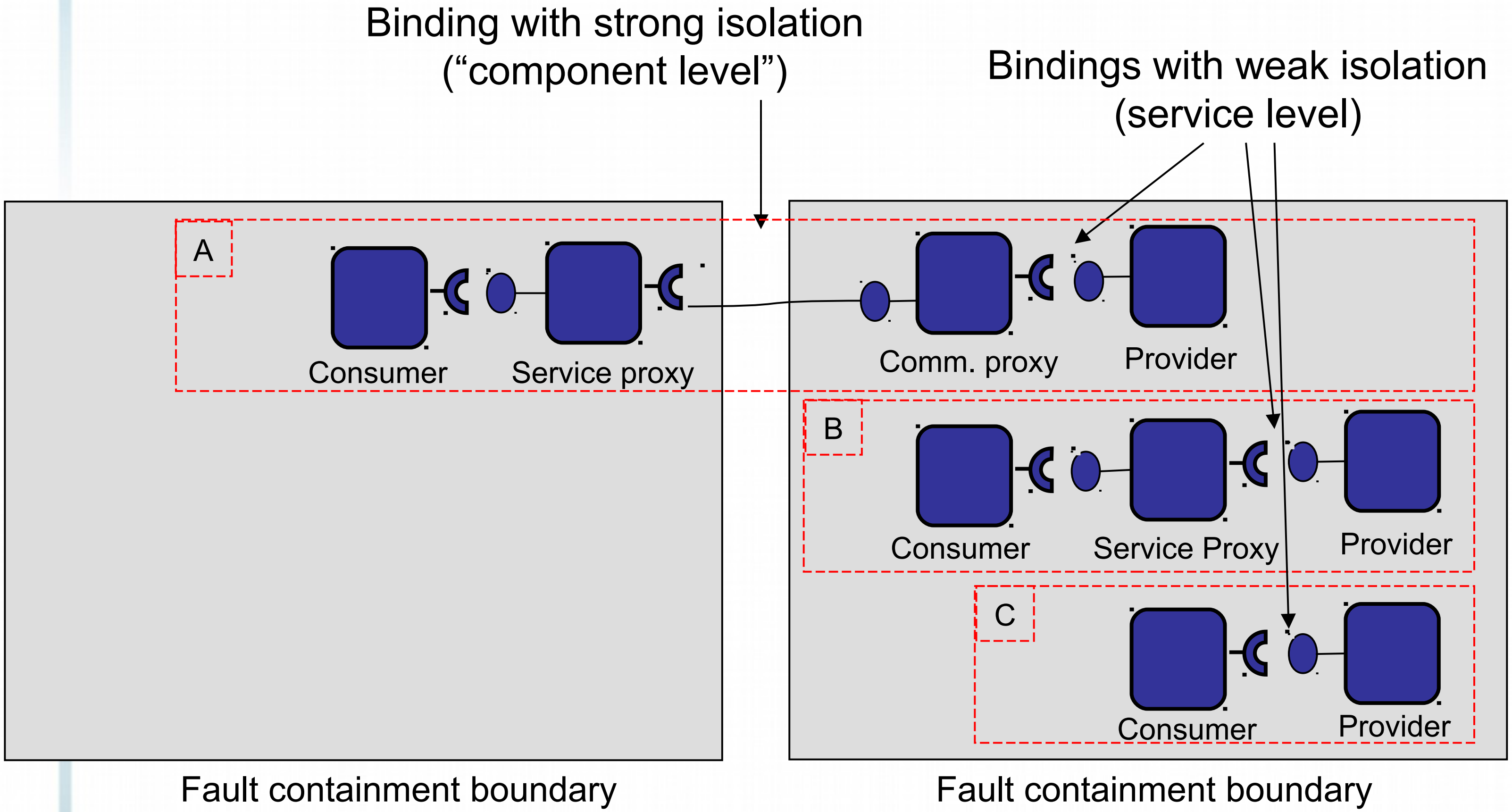
- Automatic detection, diagnosis and repair of problems
- Introducing an autonomic manager for the sandbox
  - Control loop using a sense, analyze and react principle
- Recovery oriented approach
  - Microreboots
  - Software rejuvenation
- Fault detection and forecast
  - Pragmatic approach trying to detect and avoid typical faults

Simplified View

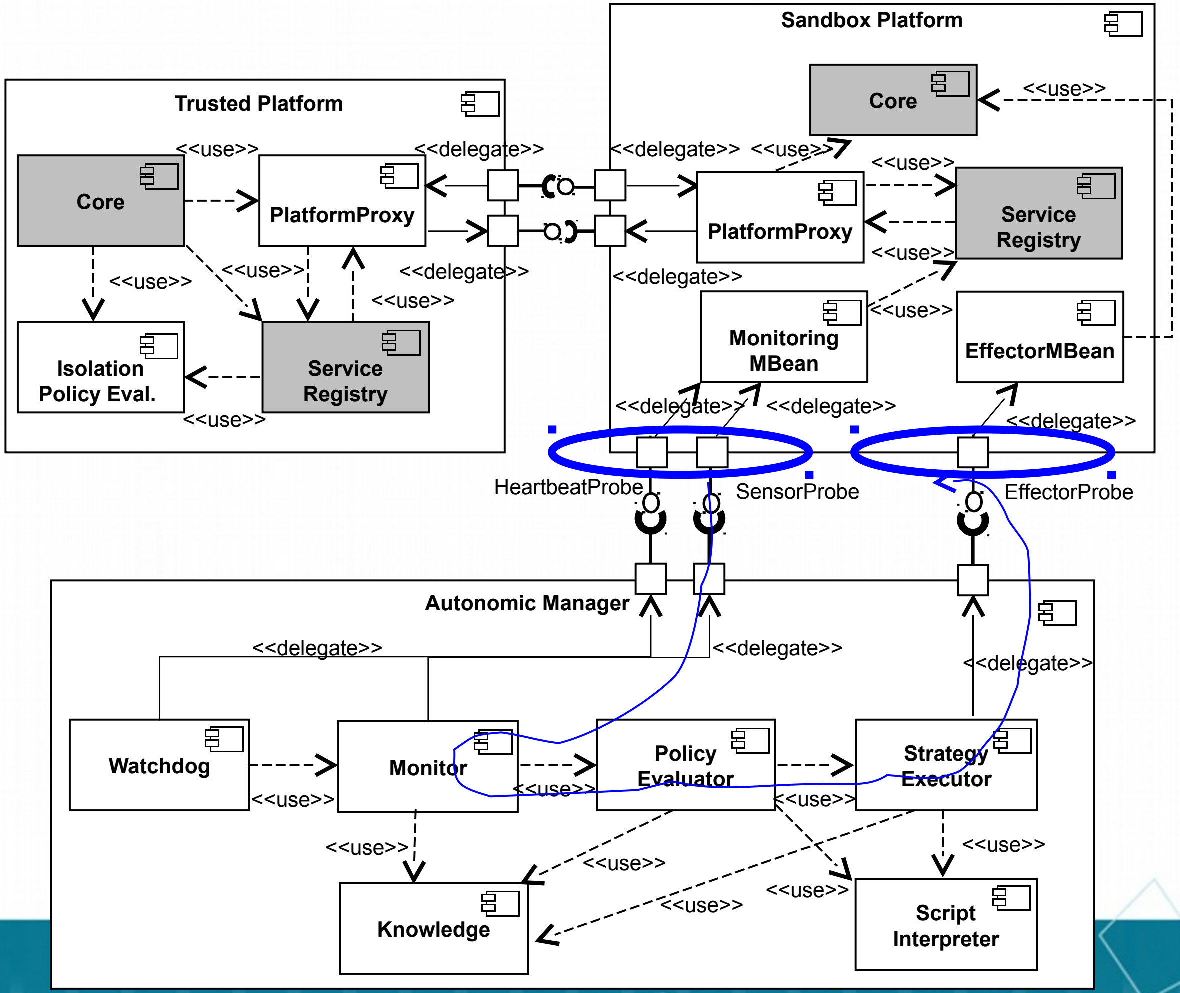


*In OSGi jargon, a component is called bundle (deployment point of view)

Isolation Levels



Architecture



Experiment I

- Does domain isolation performs better than OS-based isolation?
- Evaluation of
  - memory footprint
  - startup and reboot time
- Comparing different combinations of Application + Sandbox:
  - 2 Isolates on the MVM (Java 1.5)
  - 2 Sun Hotspot JVMs 1.5
  - 2 Sun Hotspot JVMs 1.6
- No autonomic manager, only watchdog working
- Communication layer of custom protocol on top of
  - Link API for the MVM
  - Sockets for the regular JVMs

Memory

App. Startup x Sandbox Reboot

Combination	Application Startup time (ms)	Sandbox Crash detection time (ms)	Sandbox Reboot time (ms)
MVM (2 Isolates)	3186	32	303
2 x MVM 1.5	3449	697	3064
2 x JVM 1.5	3945	660	3047
2 x JVM 1.6	3859	658	2537

Reboot side effects

- State corruption in services
  - Services need to be stateless OR
  - State must be maintained outside the application (e.g. persistence)
- Sudden disruption ends ongoing operations
- ”Event storm”
- During sandbox reboot, application is on degraded mode

Experiment II

- Watchdog individually tested was OK
- Validation of the autonomic manager effectiveness
- Detection of “known” faults
  - Memory consumption
  - CPU consumption
  - Thread instantiation
  - Service invocation
  - Application crash (e.g. illegal operation performed by a loaded library
- Prediction of faults
  - Stale service retainers
- Fault “deployment” instead of fault injection
- Major limitation: no fine grained information at the component level
  - E.g. Bundle A is consuming X MB

Conclusions and Perspectives

- Communication protocol not so performing (side finding)
- Domain-based isolation has significant
- No big differences in memory consumption between domain-based and OS-based approaches
- OS-based isolation is also feasible
- Mechanisms for fault detection are still too trivial
- Automatic promotion of well-behaving components
  - Fine grained monitoring is necessary for taking such decision

[Obrigado|Thanks|Merci]