JOpera: an Autonomic Platform for Service Composition

Cesare Pautasso
Department of Computer Science, ETH Zurich, Switzerland
pautasso@inf.ethz.ch – www.jopera.org
Service Composition is Recursive

The result of a service composition is a service

What are the implications?
Example: WS-BPEL and Service Oriented Architectures

BPEL Composition

WSDL

Web Service Interfaces
Recursive Service Composition Problems

- How to publish a composition?
- How many clients will invoke a composition?
- How to model a composition?
- How to execute a composition?
- What kind of services can be composed?
Modeling Service Compositions

• What are good abstractions for modeling a service composition?
  • Business Process Modeling Languages
    • Service invocation treated as *task*
    • Control flow (branches, loops, synchronization)
    • Data flow (and data transformations)
    • Exception Handling
    • Dynamic Late Binding
  • Syntax
    • Textual, Visual, XML, UML

[HCC2003, JVLC2005]
Executing Processes

Interpreted Execution:
The engine interprets the process model

- Requirements:
  - Efficiency, Scalability, Reliability
• For efficient execution, in JOpera process models are compiled to Java bytecode
What kind of services can be composed with WS-BPEL?

**Assumption:**
Web Services (SOAP/WSDL) are the only kind of services to be composed

**Problem:**
extensions to the BPEL standard are needed to support code snippets (BPELJ) and human tasks (BPEL4PEOPLE)
Dealing with heterogeneity in JOpera

- The JOpera composition language does not have to be changed when adding a new kind of service
Publishing a composition with JOpera

- JOpera processes are automatically published to clients using a variety of access protocols

Grid Clients
WS Clients
Eclipse RCP Clients

WSRF
WSDL
Java

JOpera Composition

WSDL
Java
Human
XML
SQL
SSH
Recursive Service Composition Problems

What kind of services can be composed?

How to publish a composition?

How many clients will invoke a composition?

How to model a composition?

How to execute a composition?
How many clients will invoke a process?

- Services built as **process-based compositions** of other services are published to be invoked by a large and unpredictable number of clients.

**Scalability on Clusters of Computers**

- Process Management Infrastructure needs to scale (many clients, many conversations).
- Web Service Composition Engines run on **cluster** of computers to handle large workloads [IJEC’04].
The Problem: How to Configure the Engine?

- The distributed engine needs to be configured:
  - Based on its current (unpredictable) workload
  - Based on the available resources of the cluster

- How many resources of the cluster should be assigned to the engine?

- Difficult to configure the engine *apriori*

- Difficult to manage the system *manually*
The Solution: Autonomic Computing

- The engine should configure itself
- Trade-off between two goals:
  - Best Performance (response time, throughput, ...)
  - Best Resource Allocation (size of the cluster)
- Requirements for the distributed engine design:
  - Support on-the-fly reconfiguration
  - Provide access to internal performance metrics
  - Expose an API for controlling the configuration
JOpera Distributed Engine Architecture

Clients

workload

API

JOpera Distributed Engine

Remote Service Provider
Adding Self-Management

1. Clients
2. API Process Control
3. Process State
4. Navigator
5. Compiled Process Structure
6. Task Queue
7. Event Queue
8. Dispatcher
9. Service Invocation Adapter
10. Remote Service Provider
11. State of the Configuration

API Configuration Monitoring and System Reconfiguration

Autonomic Controller

- performance indicators
- current configuration
- reconfiguration actions

Process Control

- workload

- API

- Process Queue

- Process State

- Compiled Process Structure

- Task Queue

- Event Queue

- Dispatcher

- Service Invocation Adapter

- Remote Service Provider

- State of the Configuration

- Clients

12 July 2005

Cesare Pautasso | www.jopera.org
Autonomic Controller Algorithm

1. Monitor
   - Current configuration
   - Performance indicators

2. Plan

3. Act
   - Reconfiguration actions

Information Policy

Optimization Policy

Selection Policy

Change
Autonomic Controller Policies

- Information Policy
  - Define which variables should be monitored
    Queue Length, Number of Navigator/Dispatcher Threads

- Optimization Policy
  - Map Monitored Variable to Reconfiguration Actions
    1. Simple Threshold Policy
    2. Differential Policy
    3. Proportional Policy

- Selection Policy
  - Choose how to implement a reconfiguration plan
Evaluation of the Control Policies

- **Workload:** Peak Response Benchmark
  - 800 concurrent processes initiated at the same time

- **Performance Indicators:**
  - Total Execution Time
  - Average Resource Allocation

- **32 node cluster environment (one thread/node)**

- **Baseline:** Static Manual Configuration
  - Fast: 10 Navigators, 22 Dispatchers
  - Slow: 22 Navigators, 10 Dispatchers
Baseline: Slow/Fast Static Configuration

Avg Resource Allocation

Total Execution Time

Number of Dispatchers+Navigators

0 5 10 15 20 25 30 35

400 800 1600

Workload size

400 800 1600

Workload Size

0 50 100 150 200 250

Time (seconds)

400 800 1600

Workload Size

20 0 50 100 150 200 250

Time (seconds)

400 800 1600

Workload Size

□ static 10/22  ■ static 22/10
1. Simple Threshold Policy

Action

Start one

Stop one

Queue Length

- Start one thread if Queue Length > T
- Stop one thread if Queue Length = 0
Tracing the Simple Threshold Policy

- Task queue
- Process queue

Graphs showing the queue size over time and the number of threads for navigators and dispatchers.
2. Differential Policy

- Start one thread if Queue Length Variation > $T_{start}$
- Stop one thread if Queue Length Variation < $T_{stop}$
Tracing the Differential Policy

![Graphs showing Queue Size and Threads over Time]

- Task queue
- Process queue
- Navigators
- Dispatchers
3. Proportional Policy

- Start/Stop $N$ threads, proportional to the Queue Length Variation
Tracing the Proportional Policy

![Graph showing changes in queue size and threads over time. The graph compares Navigators and Dispatchers.]
State Space Comparison of the Policies

- **Fast Performance**:
  - Simple Proportional Differential

- **Slow Performance**:
  - Static

Control Policy:
- Simple
- Proportional
- Differential
- Static

Number of Dispatchers vs. Number of Navigators graph.
Performance Comparison of the Policies

- static 10/22
- simple
- differential
- proportional
- static 22/10

### Avg Resource Allocation

- Number of Dispatchers + Navigators
- Workload size (400, 800, 1600)

### Total Execution Time

- Time (seconds)
- Workload Size (400, 800, 1600)
Autonomic Execution Summary

- **Manual** configuration & management of a distributed process-based Web service composition engine is difficult and expensive.
- To address this problem, we have shown how to apply **autonomic computing** techniques.
- Our evaluation indicates that different control policies can be used to explore the trade-off between performance vs. resource utilization.

[ICAC2005, ICWS2005]
Conclusion

- **Modeling** service composition behavior
  - Process-centric *composition language* (Visual & XML)
  - Development and Debugging tools for Eclipse
  - Composition not limited to Web services

- **Execution** of the composition models
  - Efficiency (compiled to Java bytecode)
  - Distributed engine (on a cluster of computers)
  - Autonomic platform (self-healing, self-tuning)
  - Extensibility (Eclipse plug-ins to provide custom service publishing and invocation adapters)
References on the language


References on the system


Demo: Bottom-up and Top-down Composition

1. Select component services from a **library**
2. Build a process using a drag, drop and connect **visual** environment
3. Run, Test, and Debug the process execution **within the same visual environment**
4. Define what services are missing and add the necessary code snippets
5. Publish the process as Web Service
Example Scenario

- Stock Quote Currency Conversion

Stock Quote Price Service

Currency Exchange Rate Service
Drag, Drop and Connect
Run, Monitor, Steer and Debug
Publish as a Web/Grid service

With one mouse click!

Processes Published as Web Services

- Test_HalloWorld.wsdl
JOpera: an Autonomic Platform for Service Composition

Cesare Pautasso
Department of Computer Science, ETH Zurich, Switzerland
pautasso@inf.ethz.ch – www.jopera.org