

# Modularization of Jini Services in Pervasive Systems: Conventional Bottle versus Contemporary Aspect

Robin Liu  
University of Victoria  
cliu@cs.uvic.ca

Yvonne Coady  
University of Victoria  
ycoady@cs.uvic.ca

## ABSTRACT

This paper reports on a preliminary experiment using aspect-oriented programming (AOP) [6] for pervasive computing. Specifically, aspects are used to modularize and structure crosscutting functionality into Jini services [3] in a simulated pervasive environment. We believe this proof-of-concept prototype demonstrates one way in which AOP can complement conventional OOP techniques in dynamic, pervasive environments.

## 1. INTRODUCTION

Management of pervasive systems must provide a comprehensive means of structuring code that is responsible for a wide range of network services. Jini [3] is a network architecture for structuring distributed systems. It provides a flexible infrastructure for delivering services and for creating spontaneous interactions between clients that use these services.

Jini's flexible infrastructure relies on the Java platform to support highly adaptive systems. Though Jini technology enhances the manageability of certain concerns associated with dynamic network services, inherent scattering and tangling of other concerns that cannot be effectively modularized with OOP still remains.

The goal of aspect-oriented programming (AOP) [6] is to provide structure for crosscutting concerns, such as resource profiling instrumentation. In our experiment, we leveraged the infrastructure offered by Jini and the structured approach to crosscutting concerns offered by AOP. A high-level overview of this architecture is highlighted in Figure 1.

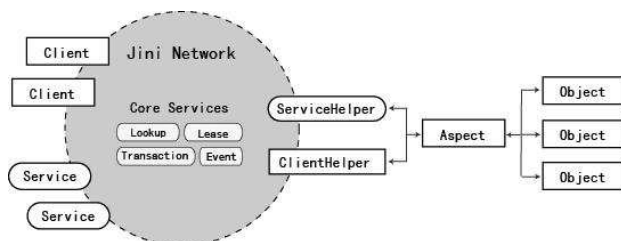


Figure 1: An aspect can be used to structure a crosscutting concern, such as resource profiling, as a one (or more) Jini service(s).

The shaded circle in Figure 1 represents a Jini network, with core Jini services (Lookup, Transaction, Lease and Event services) supplied within. The *clients* in the Figure can be any Jini-enabled application or device. The *aspect* structures functionality that crosscuts several objects, and exposes a one (or more) Jini service(s).

In a little more detail, the *ServiceHelper* is a class cooperating with the aspect, exposing the aspect's functionality in terms of Jini services. *ClientHelper* is a class that helps the aspect to use the other services on the Jini network. Through the *ClientHelper*, the aspect can retrieve information about other applications/devices on the network. The advantage of using such helper classes is the independence they provide the aspect from the target protocol/framework involved (for example, Jini vs JMX).

The rest of the paper proceeds as follows: Section 2 provides background for Jini, AOP, and profiling. The details of our proof-of-concept prototype are further described in Section 3, while Section 4 provides an evaluation of the implementation in a simulated pervasive system. Finally, Section 5 presents conclusions and future work.

## 2. BACKGROUND AND RELATED WORK

The motivation to combine Jini with AOP methodology stems from a common underlying theme: concerted efforts to improve the modularity of complex systems. The impact modularity has on productivity is becoming increasingly evident as development moves to open source projects.

Profiling is a classic example of a concern that does not typically adhere to traditional structural boundaries, but is a necessary ingredient for dynamically adaptable systems. Before launching into details of our experimental study with Jini, we briefly provide background showing how these three pieces of the puzzle, Jini, AOP and profiling, all fit together.

### 2.1 Jini Background

Jini technology enables developers to more easily construct and maintain systems made up of distributed objects. Jini supports spontaneous interaction between clients that use services over a network, agnostic of network, platform, operating system, and application. Hence, the set of problems Jini addresses are those associated with the situation where individual applications/devices dynamically

offer services to each other. These problems include finding and connecting to services, and evolving parts of the service set zero downtime.

When a service is introduced to a network of Jini-enabled applications or devices, it first finds a place where it can advertise itself. This step involves publishing a Java object that implements the service API. Clients find services by looking for an object that supports the API. When a client receives the service's published object, it downloads code it needs in order to communicate with the service.

## 2.2 AOP Background

AOP is gaining momentum as a means of facilitating modularization of *crosscutting* concerns – concerns that are present in more than one module, and cannot be better modularized through traditional means.

An aspect is a module that structures crosscutting implementation. Looking at an aspect, a developer can see both the internal structure of a crosscutting concern, and its interaction with the rest of the program during execution. As brief example of the mechanisms used in the particular incarnation of aspect-oriented programming used in our experiment, *AspectWerkz* [2], the aspect below captures all calls to the constructor of *myObject* and simply prints some tracing information:

```
package myapp;
import org.codehaus.aspectwerkz.joinpoint.JoinPoint;

public class MyAspect {
    /**
     * @Around call(myapp.myObject.new(...))
     */
    public void addToCreate(JoinPoint joinPoint) {
        System.out.println("before creating...");
        joinPoint.proceed();
        System.out.println("after creating...");
    }
}
```

The *@Around* annotation associates the execution of the aspect's *addToCreate()* method with the execution of the constructor<sup>1</sup>.

## 2.3 Profiling Background

Log-based performance profiling has been used in distributed systems [5], operating systems [10], and adaptive applications [8], and continues to be used for performance analysis and fault detection. A common characteristic of implementations based on profiling technology is the fact that instrumentation requires invasive changes to multiple modules of the target system. In essence, profiling is a classic crosscutting concern.

Tools for profiling based on well-known published interfaces are gaining momentum. Magpie [7] profiles websites to measure resource consumption (CPU, disk, network usage) of HTTP requests, and builds probabilistic models for performance prediction, tuning and diagnosis. Pinpoint [4] uses a similar approach, relying on profiling, analysis and anomaly detection for fault detection. Commercial request tracing systems include PerformaSure[9] and AppAssure [1].

## 3. PROOF-OF-CONCEPT PROTOTYPE

Here we compare options for instrumenting pervasive systems with code for resource profiling without and with AOP, and overview our proof-of-concept implementation.

### 3.1 Instrumentation without AOP

Without AOP, profiling across resources can be done either by a client application layer, or a little lower, in what we call the framework/middleware layer. We overview the pros/cons of each approach below.

#### 3.1.1 Application Layer Instrumentation

The advantage of customizing instrumentation at the application layer is precision based on application-specific requirements, assuming enough information is exposed at a lower layer. That is, just the right amount of profiling can be done – potentially resulting in gains in efficiency and accuracy from an application perspective. The main drawback of this approach is that such instrumentation gets scattered and tangled throughout the application code. This makes the application code harder to write, maintain, and evolve.

#### 3.1.2 Framework/Middleware Layer Instrumentation

Moving instrumentation to a lower level in the system, such as the framework/middleware layer, relieves application code of this burden and provides system-wide instrumentation that can be globally configured. Like the application layer approach, this again depends upon the amount of information exposed to the framework/middleware. Instrumentation points are still limited to methods from specific, predefined framework/middleware interfaces or publicly accessible methods and member fields.

### 3.2 Instrumentation with AOP

In a large software system composed of distributed objects, use of certain kinds of resources is scattered cross the system. Furthermore, which resources are to be monitored and what components/modules are to be profiled change dynamically as the community evolves. The dynamic nature of pervasive computing requires applications to be more reflective, adaptive and easy to evolve. We believe AOP – especially dynamic AOP which enables

---

<sup>1</sup> For detailed treatment of syntax, see AspectWerkz [2]. We should note that, though we chose AspectWerkz for its ability to weave at runtime, we were not able to successfully use this feature in our prototype at this time.

```

1 public interface ResourceService extends Remote {
2     ResourceReport getResourceReport() throws RemoteException;
3
4     ResourcePolicy getResourcePolicy() throws RemoteException;
5
6     boolean setResourcePolicy(ResourcePolicy policy) throws RemoteException;
7 }
8
9 class ResourceServiceProxy implements Serializable, ResourceService {
10     final ResourceService serverProxy;
11
12     ResourceServiceProxy(ResourceService serverProxy) {
13         this.serverProxy = serverProxy;
14     }
15
16     public ResourceReport getResourceReport() throws RemoteException {
17         return serverProxy.getResourceReport();
18     }
19
20     public ResourcePolicy getResourcePolicy() throws RemoteException {
21         return serverProxy.getResourcePolicy();
22     }
23
24     public boolean setResourcePolicy(ResourcePolicy policy) throws RemoteException {
25         return serverProxy.setResourcePolicy(policy);
26     }
27 }
28
29 public class ResourceServiceHelper implements ResourceService, ServerProxyTrust, ProxyAccessor {
30     public ResourceAspect aspect;
31
32     protected void init() throws Exception {
33         DiscoveryManagement discoveryManager = (DiscoveryManagement) config.getEntry(...);
34         JoinManager joinManager = new JoinManager(proxy, ..., discoveryManager, ...);
35     }
36
37     public ResourceReport getResourceReport() {
38         return aspect.getResourceReport();
39     }
40
41     public ResourcePolicy getResourcePolicy() {
42         return aspect.getResourcePolicy();
43     }
44
45     public boolean setResourcePolicy(ResourcePolicy policy) {
46         return aspect.setResourcePolicy(policy);
47     }
48 }

```

Figure 2(a): The *ResourceService* interface and proxy enables Jini clients to find the service offered by the *ResourceServiceHelper*. The proxy is registered with Jini's Lookup Service, and the client must download the proxy to communicate with the service. Lines 5-6 in the *ResourceServiceHelper* show the use the discovery manager to find the Lookup Services. The proxy is then registered with the Lookup Service.

```

1 public class Client {
2     protected void init() throws Exception {
3         ServiceDiscoveryManager serviceDiscovery = (ServiceDiscoveryManager) config.getEntry(...);
4         ServiceItem serviceItem = serviceDiscovery.lookup(new ServiceTemplate(..., new Class[] {ResourceService.class}, ...), ...);
5         ResourceService server = (ResourceService) serviceItem.service;
6         ...
7     }
8 }

```

Figure 2(b): The *Client*, where lines 3-5 show the use the *ServiceDiscoveryManager* to look up the remote *ResourceService*.

```

1 /**
2  * @Aspect perJVM
3  */
4 public class ResourceAspect {
5     private Map map = new Hashtable();
6     private ResourcePolicy policy = new ResourcePolicy();
7     /**
8      * @Around call(* simulator.Resource+.process(..))
9      */
10    public Object adviceMethod(final JoinPoint joinPoint) throws Throwable {
11        Resource resource = (Resource) joinPoint.getTargetInstance();
12        Object[] params = ((MethodRtti) joinPoint.getRtti()).getParameterValues();
13        policy.check(resource, params);
14        ...
15        final Object result = joinPoint.proceed();
16        ...
17        return result;
18    }
19 }

```

Figure 2(c): A simple example of a *ResourceAspect*. In line 2, the *ResourceAspect* is deployed as one per JVM; in line 5 all the target objects along with their usage data are stored in a Map; line 8 specifies to weave around simulated Resource's resource consuming method "process()"; finally, lines 10-18 define the advice, where the request parameters (resource amount and ...) and the target request are passed to the policy object to check if the request can be granted. (Note that line 15 specifies that, if the request is granted, execution should proceed to the target actual resource consuming method "process()").

crosscutting concerns to be introduced to a system at runtime – can provide important language level support for such a need. With AOP, instrumentation can be much more precise, and its crosscutting nature can be structured and modularized. Since AOP is a language level feature, instrumentation aspects can be easily applied to an application layer and/or framework/middleware layer without any modification of existing code. Figure 2 outlines the code involved in our Jini/AOP example.

First, Figure 2(a) overviews some of the service code involved. The *ResourceService* interface and proxy allow clients to find and communicate with the *ResourceServiceHelper* (corresponding to *ServiceHelper* in Figure 1). During initialization, the *ResourceServiceHelper* starts by creating its proxy, then uses Jini’s built-in *DiscoveryManagement* to discover Jini’s *Lookup Service* and uses the *JoinManager* to finally register the proxy with the *Lookup Service*. Figure 2(b) shows a simple *Client* initialization method, using the *ServiceDiscoveryManager* to look up the *ResourceService*. Finally, Figure 2(c) shows the AspectWerkz version of a simple example of a *ResourceAspect*. This is almost a poster child for AOP, profiling all methods that consume resources.

#### 4. EXPERIMENT

The experiment we performed uses an aspect to modularize resource profiling as a single Jini service within a simulation environment for a pervasive system. An overview of the simulation environment is shown in Figure 3, and further described and evaluated in the following subsections.

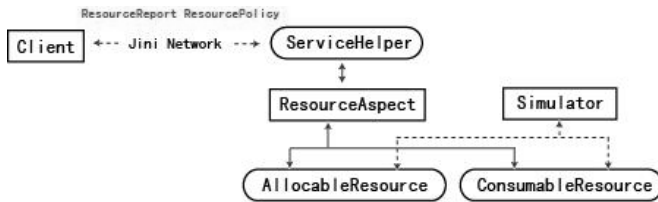


Figure 3: The *ResourceAspect* instruments resources with code to monitor and report low level details of resource usage. The aspect communicates with the *ServiceHelper*, which exposes the aspect’s functionality to the Jini network as one (or more) Jini service(s). The aspect exposes services for *ResourceReport* and *ResourcePolicy* respectively.

##### 4.1 Simulation Setup

The simulation scenario proceeds as follows: the *Simulator* creates resources and calls the resource consuming methods. The *ResourceAspect* instruments resources, and thus has access to all the resource information. The aspect controls resource usage based on an associated policy/strategy, which is separate from both the aspect and

the resource for flexibility. The *Client* pulls: (1) the resource report, and (2) resource control policy from the *ResourceAspect* through the *ServiceHelper*.

In Figure 3, the *Client* is a Jini client application; the *ResourceAspect* provides customized monitoring advice to instrument the methods that consume certain resources; a *ResourcePolicy* object represents the policy that the *ResourceAspect* uses to control consumption of each resource, which is consequently transparent to the target methods. In this simulation, we have explored the most fine grained configuration possible, where there is a different *ResourcePolicy* object for each method that the *ResourceAspect* advises; finally, the *ResourceReport* contains the information, such as type and usage, about the resource.

For simplicity, two types of resources are defined in the simulation, representing *allocatable* and *consumable* resources respectively:

- The *AllocatableResource* simulates a resource that can be allocated upon request and recovered/de-allocated after the request has been serviced. We envision this resource as corresponding to resources such as network bandwidth, memory and CPU time.
- The *ConsumableResource* simulates a resource that cannot be recovered/deallocated after a request has been serviced. We envision this resource as corresponding to resources such as battery power or write only secondary storage.

In the experiment, a multi-threaded driver is used dispatch requests. This driver simulates requests (for a given amount of a resource, and for a given duration) from clients in a pervasive system. A resource can be allocated to multiple requests simultaneously. If however, the amount of the resource a request requires is more than what is currently available, the request is queued at the resource until at least that quantity of the resource becomes available.

##### 4.2 Simulation Results & Analysis

The resource consumption results of a simple simulation execution are shown below. This simulation uses only 3 different *allocatable* resources, each with its own unique *resource id*. As a direct result of the aspect, each resource reports: the amount of the resource in use and not in use; the number of requests waiting/processing; and finally, the amount and duration of the most recent request serviced.

```

-----
Resource id: simulator.Resource@2ab653
Not in use: 55
In use: 481
Number of requests waiting: 2
Number of requests in process: 2
Last request amount: 259
Last request interval: 2135
-----

```

```
-----  
Resource id: simulator.Resource@acdd02  
Not in use: 2  
In use: 525  
Number of requests waiting: 4  
Number of requests in process: 2  
Last request amount: 242  
Last request interval: 2275  
-----
```

```
-----  
Resource id: simulator.Resource@f0b4a3  
Not in use: 48  
In use: 498  
Number of requests waiting: 1  
Number of requests in process: 2  
Last request amount: 249  
Last request interval: 2460  
-----
```

In the snapshot above, each service is simultaneously processing 2 requests, while other requests are queued. The important take-away point when looking at this output is that the resource profiling code is configured completely within the aspect.

#### 4.2.1 Analysis

This simulation shows how AOP can provide a powerful way modularize and structure Jini services that crosscut resources in a pervasive system. Specifically, resource profiling code introduced by an aspect can be customized to collect low level data, and yet still be kept separate with the system. It is important to note here that, since the current version of AspectWerkz does not have built-in support for distributed crosscutting concerns, the proof-of-concept implementation is limited to resources within a single JVM.

### 4.3 Jini: Bottle versus Aspect?

The prototype implementation demonstrates how AOP can be merged with Jini technology to improve modularization for crosscutting Jini services. Though this stands as simply a proof-of-concept experiment, the fact that it worked now begs the question: what kind of services could benefit from modularization as contemporary aspects versus conventional (bottled) OOP.

The advantage of this implementation over a non-AOP approach is essentially modularity, resulting in increased flexibility, extensibility and reusability.

Flexibility is increased because can be specified and customized in one centralized place, separate from the application and framework/middleware involved. Through runtime-weaving technologies, instrumentation can ultimately be extended dynamically at runtime while the pervasive community evolves. Dynamically adjusting between aggressive profiling for diagnostic purposes versus

lightweight approaches to eek out performance is particularly important in dynamic pervasive environments. Finally, we anticipate that general purpose aspects could form a common starting point for reuse, perhaps in the form of available libraries.

## 5. CONCLUSIONS AND FUTURE WORK

The prototype implementation shows how AOP can improve the modularity of code in pervasive environments. As opposed to having this code scattered and tangled across classes, the example shows how resource profiling techniques used by clients of Jini services can be localized, structured, and managed as aspects.

The current prototype has shown several encouraging proof-of-concept results. In terms of future environments, we plan to investigate crosscutting concerns within an open-source J2EE server built on a JMX microkernel, and experiment with fine-grained load balancing aspects within cluster environments. In terms of language features, we plan to investigate available metadata (JSR-175) for additional management capabilities, utilize control flow information structured within aspects for path-specific resource provisioning, and experiment with dynamic aspects (aspects loaded/unloaded at runtime) for adaptation.

## References

- [1] AppAssure, [www.alignmentsoftware.com](http://www.alignmentsoftware.com).
- [2] AspectWerkz, <http://aspectwerkz.codehaus.org/index.html>.
- [3] Jini., <http://www.jini.org/>
- [4] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. *Pinpoint: Problem determination in large, dynamic, Internet services*. Proc. International Conference on Dependable Systems and Networks (IPDS Track), pages 595-604, June 2002.
- [5] G. C. Hunt and M. L. Scott. *The Coign automatic distributed partitioning system*. Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI'99), pages 187-200, Feb. 1999.
- [6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, *Aspect-Oriented Programming*, European Conference on Object-Oriented Programming (ECOOP), 1997.
- [7] Magpie, <http://research.microsoft.com/projects/magpie/>.
- [8] D. Narayanan, J. Flinn, and M. Satyanarayanan. *Using history to improve mobile application adaptation*. Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applications, pages 31-40, Dec. 2000.
- [9] PerformaSure, [www.sitraka.com/software/performasure](http://www.sitraka.com/software/performasure).
- [10] M. Seltzer and C. Small. *Self-monitoring and self-adapting operating systems*. Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), pages 124-129, May 1997.