Kent Academic Repository

Full text document (pdf)

Citation for published version

Quig, Bruce and Kölling, Michael and Rosenberg, John and Steele, Phillip (2005) Interactive Visualisation and Testing of Jini Services. In: Proceedings of the Fifth International Conference on Quality Software (QSIC 2005), Melbourne, Australia.

DOI

Link to record in KAR

http://kar.kent.ac.uk/14253/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version.
Users are advised to check **http://kar.kent.ac.uk** for the status of the paper. **Users should always cite the published version of record.**

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at **http://kar.kent.ac.uk/contact.html**

KAR Kent Academic Repository

University of Kent

Interactive Visualisation and Testing of Jini Services

Bruce Quig
Monash University
bruce.quig@infotech.monash.edu.au

John Rosenberg
Deakin University
johnr@deakin.edu.au

Michael Kölling
University of Kent
mik@kent.ac.uk

Phillip Steele
Monash University
phillip.steele@infotech.monash.edu.au

Abstract

Dynamic service-oriented architectures aim to provide more flexible and robust systems that are able to handle change over time. Their dynamic nature however, provides extra challenges when understanding, developing, testing and debugging them.

This paper identifies and discusses a number of issues and difficulties that are faced in their development. It discusses ways in which the building of such systems can be supported by development tools, focusing particularly on interactive testing and debugging mechanisms.

A prototype tool for monitoring and interactively testing Jini services that has been developed to address, and further investigate these issues, is then described.

1. Introduction

Distributed applications have numerous potential advantages, such as increased reliability, performance and maintainability. Building such systems however, is an inherently difficult task. It requires developers to deal with the unique characteristics of distributed systems such as increased latency, bandwidth restrictions and fluctuations, security implications and requirements, and the increased risk of partial and complete failure of other network nodes.

Networked systems are becoming more widespread, divergent in bandwidth, mobile, and are being populated by a widening variety of computing devices. There is a growing recognition of the benefits of more flexible and dynamic systems.

These systems need to:

- handle the arrival and departure of resources to the network,
- allow for the discovery of required resources,
- allow resources to describe their capabilities,
- handle partial failure,
- allow for system evolution.

Dynamic service-oriented architectures such as Jini, Salutation, UPnP and Web Services/WS-Discovery aim to provide more flexible and robust systems than more traditional distributed system architectures. In these systems change and evolution are considered the norm rather than the exception. Their dynamic nature however provides extra challenges when understanding, testing and debugging them.

In the following sections these issues are discussed at a conceptual level and also more specifically, using Jini as an example architecture. We discuss ways in which development tools can support the development of service-oriented systems, focusing particularly on visualization techniques and interactive testing and debugging mechanisms.

2. Issues

Many of the challenges faced in building dynamic distributed systems result from their general characteristics. In the following sections, we summarise the main problems.

2.1. Inherent complexity

The promise of dynamic service-discovery based systems is based upon their ability to handle what are recognised as a range of difficult issues in developing long running, robust, flexible and evolvable systems. Whilst the use of middleware frameworks, abstraction and re-usable libraries assist, the development process and end goals are undeniably complex.

2.2. State and availability assumptions

Traditional distributed systems are usually based upon the assumption that resources are known, stable and available. If system components have known identities and capabilities it is much easier to check their state and availability during testing and debugging. The transparency of the service discovery methods is an advantage in terms of maintaining a loosely coupled system, but can make debugging considerably more difficult.

2.3. Dynamic system state changes

In addition, the dynamic nature of services and their interactions means that over time there is no guarantee that the service used previously will either be available or be the most suitable service from the client's perspective.

2.4. Difficulty creating services in isolation

While services may be part of a larger overall application they are potentially difficult to test and debug in isolation.

A number of unit test frameworks are available, such as JUnit [1] for Java-based applications, to support lightweight development processes and regression testing for traditional applications.

To test services, it is currently necessary to develop client code to use the remote services, and to test its functionality. It is therefore necessary to either write extra client code in the form of unit tests, or simultaneously develop clients of these services to test their correctness. This needs to be repeated at every iterative step in development, discouraging developers from adopting a fine-grained iterative approach to development.

Additionally if these objects do not perform correctly, it is currently difficult to debug them without using some form of driver program that can then be run using a debugger. It would be beneficial if services could be tested and debugged at a service level via direct interaction mechanisms. The BlueJ programming environment [2], is an example (at a non-distributed level) of a tool that supports such interaction.

2.5. Understanding services

In an environment where service oriented architectures are deployed there is likely to be a need (and intent) to re-use existing services. Developers will use services written by others. It is therefore important to be able to browse repositories of existing objects and retrieve data about them. Even more useful would be the ability to directly interact with services. By this we mean the ability to interactively call the methods of these services and receive the results of these invocations.

2.6. Iterative development methodologies

There is a growing interest in highly iterative development methodologies such as extreme programming (XP), Crystal and Scrum. These are also referred to as lightweight or *agile* development processes. At the heart of these processes is the notion of developing complex systems by a finely grained iteration of developing prototypes that are constantly evolved, tested, refactored, and enhanced. In such systems the roles of unit testing and debugging are extremely important.

These tests are expected to be run often and need to be relatively quick to run. This is made somewhat more difficult when the configuration and management of a dynamic distributed infrastructure can be quite complex and resource intensive. Tests that are difficult or long running may discourage the use of a highly iterative approach.

Associated with these agile processes are endeavours looking at how they affect software testing. A prominent participant is the Context-Driven School [3], which place great importance on the context of a test as to determining its effectiveness. An exemplar of this is exploratory testing [4]. Exploratory testing is similar to ad hoc testing, where the tester chooses which tests to run and explores the behaviour of the software. Proponents of this form of testing assert that there is greater effectiveness in designing and running a test that is based upon the results of previous tests. In this way unexpected behaviour can be explored and understood at the time that is encountered. This type of testing is seen as a compliment to automated testing procedures, rather than as an alternative.

Adopting this type of testing methodology when testing a dynamic distributed system allows testers to be able to respond to changes in service availability, and other network conditions. To do this it would be necessary to firstly provide some mechanism for a tester to be able to monitor changes in the distributed network, and secondly to be then invoke tests that are designed in response to those changes.

2.7. Interactions between client and service are not visible

The dynamic distributed system architectures that have been mentioned focus on the provision and management of the service discovery process. They do not necessarily play any role beyond the hooking up of service consumers and providers. In Jini for instance, a chosen proxy is downloaded to the client in response to a search for a service. This proxy can then use any supported protocol to provide the service it represents.

This allows great flexibility in the types of services that can be accommodated in a Jini system. However this makes it harder to monitor and debug individual services and applications.

3. Tool requirements

In the previous section we described a number of the difficulties that are commonly encountered when developing services that will operate in a dynamic manner. In this section we will expand upon the

requirements of a tool to visualise and interact with Jini-based services.

In systems that are designed to respond to a dynamic network environment, it becomes important that such changes can be recognised and understood by developers. The use of a browsing tool that allows developers (and potentially application administrators after deployment) to investigate and interact with the current state of a dynamic service-based system provides would seem to provide a promising means of dealing with the issues previously identified. The use of two related techniques, *visualisation* and *direct manipulation*, appear to be well suited to the problem domain.

3.1. Visualisation

Software visualisation is a well-recognised technique to aid in the comprehension of software systems. Hyrskykari [5] notes that empirical assessment of program visualisation shows that it can be effective and that a key determinant is identifying the most appropriate types of graphical representations for a particular system.

3.1.1. Multiple, appropriate views

There is also evidence that there is likely to be more than one appropriate way for available data to be displayed. The provision of multiple views of the same system has also been found to improve program comprehension [6].

One of the commonalities between the different service architectures is that they all provide a level of implementation transparency, hiding details of the network and other environmental details. A client asks for services and receives responses about services that match certain criteria. A corresponding service level view would graphically show available services.

Whereas a service level view of an environment might show services and probably hide more implementation level details, there is also a need at some level to show the concrete implementation level specifics of services and other entities such as lookup services, devices and servers.

With the increased adoption of highly iterative development processes this is likely to be of even greater importance, as developers are more often changing the focus of their activities through the various stages development, testing, debugging and refactoring. The appropriate visual abstractions of the system, and also their granularity, could perceivably change during different tasks.

3.1.2. Extensible architecture for additional views

As an emerging field, where services may represent a wide variety of different services and devices it is also not clear what the most appropriate abstractions will be for all types of applications. These architectures may be used to implement high volume, server-based computations or to represent lightweight devices and services in a mobile, or pervasive computing infrastructure. There is therefore a requirement that a browsing tool should be extensible so that additional views can be added that may more appropriately support certain types of systems.

3.1.3. Dynamic updating of views

Additionally, the dynamic nature of the systems under discussion mandate that the tool should dynamically updated the representation of the system as the state of the system changes.

3.1.4. Handling potentially large data sets

A common problem for any graphical representations of data is that of handling potentially large amounts of data. Any system that attempts to visualise a dynamic network of services and their attributes will need to potentially deal with large numbers of objects.

3.2. Interaction

Whilst static system visualisations may be beneficial, the ability to use direct manipulation techniques upon the graphical entities representing the system provides a number of advantages [7]. Direct interaction of the graphical entities provides a means to inspect their state and behaviour.

An example of this is the BlueJ programming environment [2]. BlueJ provides a number of means for users to interact with Java programs. It provides a graphical UML-style representation of Java classes showing classes and their relationships. Users directly interact with the graphical representations of the class to open editors, instantiate instances and inspect their state further. It allows users to experiment with code (compile, run, test) at a very fine-grained level without the usually associated overheads of writing driver programs or test classes.

Whilst it should be noted that BlueJ has a specific focus of providing visualisations and interactions aimed at novice programmers, it has proven to be successful at improving program comprehension, and has encouraged users to prototype, experiment and test their code early and often [8].

Interactively prototyping and testing services in a Jini network, in a manner similar to the way that BlueJ allows interaction at an object level, would provide a way for developers to informally learn about existing services, and enable them to test in an exploratory and context-driven manner.

4. Prototype tool

In this section we describe a prototype interactive browsing tool for Jini systems, *Juniper*, which was built to fulfil the requirements identified.

4.1. Multiple, appropriate views

The main window of the Juniper browser is divided into 4 main panes. The top left hand corner provides an *exploration* pane, the top right shows the main *visualisation* pane, the bottom left shows a *filter* pane pane and the bottom right hand area provides a *work* pane. Figure 1 shows Juniper's main window and the main four areas. The principle behind the browser is to provide a mechanism to display information about the key abstractions within a Jini system. The main abstractions that we have identified are those of services, lookup services and the local Jini system as a whole entity. There are also secondary abstractions that represent the internal state of these entities such as groups, attribute entries, leases and interfaces. The third element is the connections or relationships between these entities.

The exploration pane provides a tree-based mechanism to traverse through available elements for a particular view of the system. It is similar in concept to file system browsers found on most graphical operating systems. There are two main views of the system that represented by tabbed panes. These represent a logical view (*service view*) and a physical implementation view (*lookup service view*). This separation of views is not new, it is also used by the IncaX Service Browser [9]. Selecting entities within these views will load the graphical views of this entity into the visualisation pane. Each entity can possess more than one graphical view, which appear as tabbed panes within the visualisation pane.

Figure 1 shows the *lookup service* view with the root tree node selected which represents the local Jini network containing available Jini entities within multicast range. The rectangular, orange entities represent available lookup services, the round-edged, green entities represent services, and the connections between them represent registrations of the services with those lookup services.

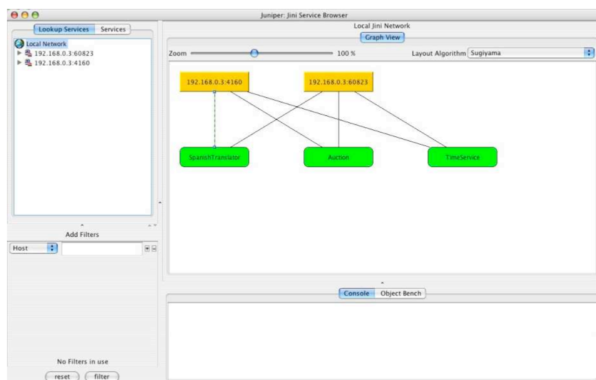

Figure 1 Juniper main screen

The *service view* (Figure 2) provides a logical view representing the services available in the network in a view that is conceptually consistent with the view of available Jini services as seen from a client's perspective.

In a Jini client, the system is queried for services based upon a query that normally uses interface type and attributes to determine suitability. Physical location of the service and lookup service from which it was found are transparent to the client.

Juniper provides an extensible framework for introducing additional views for the main Jini entities (Jini network, Services and Lookup services). Creating a new view of an entity simply involves subclassing an abstract View class. The system registers views to entities as part of its start up configuration and loads all applicable views when entities are selected in the exploration pane.

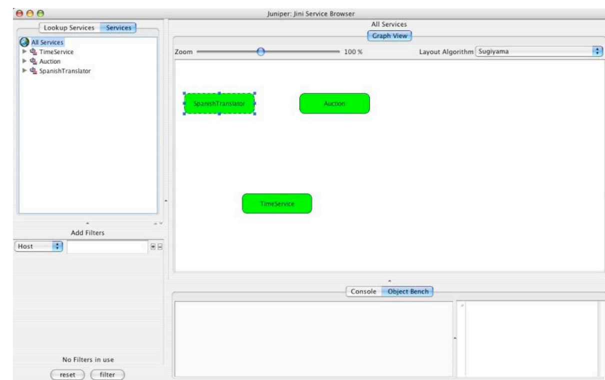

Figure 2 Service view

4.2. Extensible architecture for additional views

As an emerging field, where services may represent a wide variety of different services and devices it is also not clear what the most appropriate abstractions will be for all types of applications. These architectures may be used to implement high volume, server-based computations or to represent lightweight devices and services in a mobile, or pervasive computing infrastructure. There is therefore a requirement that a browsing tool should be extensible so that additional views can be added that may more appropriately support certain types of systems.

4.3. Dynamic updating of views

Juniper registers itself as a listener for event notification for services and lookup services. Events such as the arrival and departure of services and lookup services can then be passed on to relevant views.

4.4. Handling potentially large data sets

Juniper addresses the scalability issues of representing graphical views in a number of ways. Firstly, like a number of other Jini service browsing tools it is designed to allow filtering operations. The filter pane provides the ability to search and limit the resultant set of entities to those that match the filter query. Queries can include multiple parameters including service groups, interface types, host, service ID and attributes.
The other strategy for managing scalability is the use in zooming within a number of the views shown in the visualisation pane. The graphical panes make use of the JGraph visualisation library, which provides support for the scaling of graphs, and also their automated layout. Juniper allows users to zoom in and out of the main views and also to select the automated layout strategy. Manual layout of graph objects is also supported.

4.5. Interaction

Beyond the standard user interface selection and interaction mechanisms found in the exploration and visualisation panes, Juniper allows other interactions with services. Services that are shown in the visualisation pane possess a popup menu that allows users to choose to *get* a service proxy. This process downloads a service proxy object for the selected service from a lookup service and places it upon the object bench located in the *work pane*.

Figure 3 shows a downloaded service proxy object with its popup menu activated. This menu provides a list of the available methods that this service object provides as well as the ability to inspect the internal state of the object. If a method is selected, it is invoked in a similar manner to the way invocations occur in BlueJ except that it can perform remote method calls as well as local method invocations.

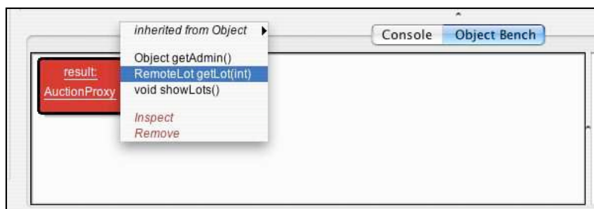

Figure 3 Object bench

From an implementation perspective this is achieved by running a second debug-enabled Java virtual machine (JVM) in which all objects on the object bench reside. When the initial service proxy request is made, client code to gain a remote service proxy is generated dynamically, compiled, loaded and run on the second VM. Upon successful completion of the call, a wrapper object is placed upon the workbench. A similar process is followed for any subsequent method invocation calls. This allows full debugging capabilities to inspect the internal state of each java object. Each internal field of an object can be recursively interrogated for its state as well.

All of this is done without the need for any prior exposure or availability of the Java class definitions for the service. Even with RMI's remote code downloading capabilities, it is usually necessary to have a class definition for the service interface required. Our architecture, through the use of reflection and background downloading of required classes, allows the interactive use of services with no prior knowledge of their type definition.

Invocation of methods that require parameters are supported by generating method dialogs that allow the setting of parameters in the same manner as BlueJ (**Error! Reference source not found.**).

Methods that have non-void return types are caught and represented in a result dialog (Figure 4). If the return type is non-primitive (an object) it is possible to either inspect it or to place it on the object bench as a target for further invocations if required.

Figure 4 Method dialog

There is also the additional capacity to create objects of any class type that is available on the environment's classpath. This allows us to create any objects that may be need as parameters for further invocations.

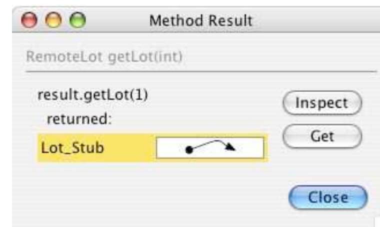

Figure 5 Method result inspection

Many Jini services also expose interfaces that allow them to be remotely administered. Within Juniper it is possible to use these interfaces by downloading a service proxy and invoking the service's getAdmin() method which is used to gain a reference to the administration proxy for the service. Figure 6 shows an object bench where firstly an Auction service proxy was placed upon the object bench. Its getAdmin() method was then called and the resultant administration object was also placed on the object bench. These methods can then be invoked to administer the service.

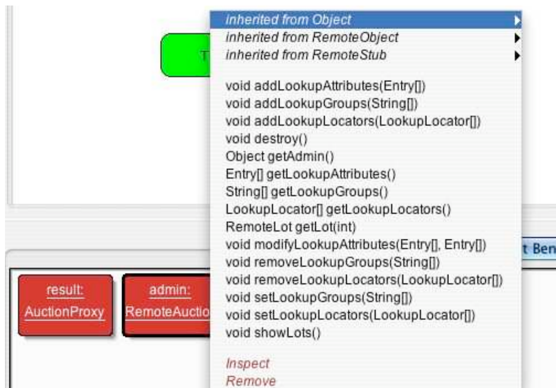

Figure 6 Service admin methods

5. Related Work

The Jini Starter Kit provides example code including a rudimentary graphical browser that allows filtering of services, administration of services and inspection of their published attribute entries. The interface is implemented as a Java Swing application. It does not attempt to provide any visualisation of the entities or their relationships.

Inca X [9] is an IDE for developing Jini applications. It also provides a service browser plug-in for their environment. There are a number of similarities to our environment, and it provided an exemplar for a subset of the features we have identified. It provides some graphical representation of Lookup services, and the services that are registered with it. It does not however provide a network level view where lookup services, services and the registrations between them are easily identified. The browser plug-in's views are not extensible.

It provides some interaction with services by allowing users to access the administration interfaces supported by a service. It also supports any attached user interfaces that support Jini's ServiceUI standard.

6. Conclusions and Future Work

We have described many of the common challenges faced in developing dynamic service discovery-based systems. We have argued that software visualisation and interactive runtime execution tools can provide additional support in their development and identified requirements for such. We have presented an integrated tool for viewing and interacting with dynamic Jini services that aims to then meet that criterion.

The resultant tool, Juniper, provides the following features:

- It provides a number of complete system visualisations not available in other tools.
- It provides a simple, extensible architecture for the addition of additional views.
- It allows a form of exploratory and context-driven testing which would not be otherwise possible without writing additional driver code or access to class definitions.
- It enables developers of service clients to interact with existing services as a means of understanding and reasoning about them, allowing developers to experiment with their own and other services. This includes services for which there is no prior knowledge or availability of the service interface classes.

6.1. Future work

One of the goals in building this prototype tool was to provide a vehicle for more thorough investigation of the problem domain, by providing the ability to apply techniques that were previously unavailable for these types of systems. Our experience with these techniques in other fields, chiefly in the education and training of programmers, has been extremely promising.

Part of the power of this tool is gained by the way in which it integrates two traditionally separate tools, one that provides visualisations of the system, and the other that provides fine-grained interactive execution and debugging capabilities. We found ourselves using it in combination with a separate IDE that could more easily configure the launch and development of services. Integrating the tool with an IDE such as Inca X or Eclipse would make it even easier to incorporate the use of the tool within the chosen development methodology.

Juniper makes it easy to perform exploratory testing on any available Jini service. This is a complementary activity to automated testing. Exploratory testing does require discipline on the part of the tester when repeating tests. Further integration with a unit test framework could be useful to provide a means of selective recording an interactive testing session as an automated test.

7. References

[1] JUnit, "JUnit, Testing Resources for Extreme Programming," http://www.junit.org, accessed 20 September, 2003, 2002

[2] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ system and its pedagogy," Computer Science Education, Special issue on Learning and Teaching Object Technology, vol. 13, 2003.

[3] Context-Driven School, "The Seven Basic Principles of the Context-Driven School," http://www.context-driven-testing.com/, accessed April 1, 2005

[4] J. Bach, "Exploratory Testing Explained," http://www.satisfice.com/articles/et-article.pdf, accessed April 1, 2005

[5] A. Hyrskykari, "Development of Program Visualization Systems," presented at 2nd Czech British Symposium onVisual Aspects of Man-Machine Systems, Prague, Czechoslovakia, 1995.

[6] S. Meyers and S. P. Reiss, "An Empirical Study of Multiple-View Software Development," presented at SIGSOFT symposium on Software development environments, Virginia, USA, 1992.

[7] B. Schneiderman, "Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces," presented at International Workshop on Intelligent User Interfaces, New York, 1997.

[8] K. V. Haaster and D. Hagan, "Teaching and Learning with BlueJ:an Evaluation of a Pedagogical Tool," presented at Information Science + Information Technology Education Joint Conference, Rockhampton, QLD, Australia, 2004.

[9] Inca X, "Inca X - IDE and Runtime environment for Jini 2.0," http://www.incax.com/, accessed April 1, 2005