A Dynamic Middleware-based Instrumentation Framework to Assist the Understanding of Distributed Applications

Denis Reilly

A thesis submitted in partial fulfillment of the requirements of

Liverpool John Moores University for the degree of

Doctor of Philosophy

July 06

Any maps, pages, tables, figures graphs,or photographs, missing from this digital copy, have been excluded at the request of the university.

Abstract

Distributed object and component-based middleware technologies dramatically simplify the development of distributed applications, but they offer little support to assist their runtime control and management. The control and management problem is exacerbated by the fact that distributed applications are notoriously complex at runtime, due to their inherent dynamics and the possibility of heterogeneous component technologies. A prerequisite to the management of any dynamic system is an understanding of the system itself, which calls for techniques capable of gathering information relating to structure and behaviour. In conventional engineering disciplines, such techniques are provided through *instrumentation*, which provides instruments, such as data loggers, gauges, probes, monitors, that further our understanding of a target system.

This thesis sets out on a journey, which aims to utilize the concepts of conventional engineering instrumentation to assist the understanding of distributed applications. The starting point for such a journey is that of traditional *software instrumentation*, which has been around for some time (circa 1970s), but has not reached the maturity of its conventional engineering counterpart. Initially, software instrumentation was used to assist the understanding and debugging of procedural language programs and later to assist the tuning and visualization of parallel programs. The basic technique of software instrumentation is the insertion of instrumentation code at points of interest throughout a program. However, where distributed applications are concerned this is impractical for several reasons, which include the distribution itself and the problem of runtime insertion, without having to take resources offline. If we are to use software instrumentation in distributed applications, these issues and others must be addressed.

The main aim of the thesis is to investigate the fundamental requirements of on-demand distributed software instrumentation, and the promotion of instrumentation as a new middleware service. The main contribution of the thesis is the conception of a *dynamic software instrumentation framework*. The framework consists of a series of related models including: a requirements model, a classification model, formal and semi-formal

analysis models and a programming model. An instrumentation architecture makes up the heart of the framework. The architecture regards instrumentation as *services*, which are intended to complement core middleware services. A proof of concept implementation of the architecture has been prototyped using Jini (a Java-based middleware technology) to provide an API for use in distributed Java applications. A series of case-studies are used to evaluate the architecture and assess the effectiveness and performance overhead of instrumentation services.

Overall, the thesis provides a reference framework, which can be used by system architects, application developers and middleware technology providers as a basis for the development of subsequent instrumentation efforts.

Dedicated to the Memory of Mum & Dad

Acknowledgements

I would like to thank my supervisory team, Prof. A. Taleb-Bendiab and Dr. Carl Bamford for their help and guidance throughout this research project. In particular, I would like to express a special thankyou to Prof. Taleb-Bendiab for his support, friendship and ability to motivate me and ask difficult questions at precisely the right time and for his help in writing the thesis.

I would like to express a special thankyou to Prof. M. Merabti for his support, friendship and guidance throughout my academic career, and in particular for support relating to the funding of this project and help in writing the thesis. I would also like to thank Dr. M Hanneghan for the useful advice.

My thanks also goes out to colleagues at Liverpool John Moores university for their support and friendship and over the past years. In particular, my thanks go out to the following (in no particular order): David Llewellyn-Jones, Mike Baskett, Rubem Periera, Mengji Yu, David Lamb, John Haggerty, Tom Berry, Andy Symons, Andy Laws, Ella Grishikashvili, Mark Allen, Bob Askwith, Chris Wren, Geof Staniford, Chris Bewick, Philip Miseldine, Martin Randles, Dhiya Al-Jumeily, Fausto Sainz Salces, Stu Wade, Nagwa Badr, Omar Abuelma'atti, Sud Sudirman, Paul Fergus, Hala Mokhtar and Gurleen Arora. I would also like to thank the CMS technicians for their friendship and outstanding technical support over the years. A special thankyou goes out to David Llewellyn-Jones for his time spent reading the thesis and technical help with EndNote.

I would like to thank my childhood friends spread throughout Liverpool who have stood by me through the good times and the not so good times: Dave Parry, Neil Larsen, Ian Parry, Michael Brereton (deceased), Chris Quinn, John May, Chris Ord (deceased), Michael McGinn, Margie Rice, Jeanie Woodbridge, John Taylor and Craig Johnston.

Finally, and by far most importantly I would like to thank the following: the love of my life for putting up with me during the writing of this thesis – the grumpiness, the erratic behaviour which she took in her stride; my late parents, without whom this project would not have happened (obviously); my brother and his wife, for the opportunity to watch Everton FC on Sky TV, which provided a welcome break from thesis writing.

List of Figures and Tables

Chapter 1

Introduction

The increasing complexity of distributed systems and their inherent dynamic behaviour suggests a need for management to ensure that they run smoothly and continue to provide secure reliable services. Such management is likely to be hindered by heterogeneity of hardware, networking and software technologies that may exist in a distributed system. Standards exist that hide the problems of heterogeneity of hardware, networking and programming languages. However, the problems associated with dynamic behaviour are not so easy to deal with. The components in a distributed system may undergo changes in state, such that their characteristics differ from one instant to the next - they may fail or behave unpredictably. Such problems have been the concern of a large body of research concerned with distributed systems management using conventional approaches [1-5]. Through this work the author looks into other traditional disciplines for further inspiration in developing an understanding of distributed systems.

1.1 An Engineering Solution

To assist the management of distributed applications the adopted approach is based on principles and techniques used in conventional engineering - more specifically real-time process control systems design. The conventional engineering disciplines of electrical/electronic and mechanical engineering are founded on scientific laws and principles that may be used to describe the behaviour of natural real-world systems. The conventional engineering disciplines have developed models of real-world systems that prove valid when subject to analysis and mathematical proof. Two particular engineering disciplines are those of instrumentation and control, which are often combined to measure, monitor and generally assess and manage the performance and behaviour of a target system.

Along with others active in the field of distributed systems understanding and analysis [6-9], the author argues the need for instrumentation and proposes a series of models and an

architecture that provides instrumentation to support distributed system management. Others, including [10-14] use the notion of instrumentation, based on gauges, monitors and probes to provide performance and behavioural information according to the architectural style of a distributed system.

Whilst the work described in this thesis shares similar objectives and roots with the above described efforts, it focuses on a novel proposition to promote instrumentation as a core middleware service. To strengthen this proposition, the thesis provides an examination of middleware programming and communication models and uses these models as the basis for the development of a series of instrumentation reference models. To this end, the work provides a rigorous consideration of instrumentation from basic requirements and conceptual representations through to the development of a dynamic instrumentation architecture.

1.2 Dynamic Software Instrumentation

Software instrumentation[1] has been used for some time in software engineering and parallel computing to debug and test software applications and also for monitoring performance and producing runtime metrics. Traditional, *static* instrumentation approaches involved the insertion of additional software constructs at design-time (via compiler directives), or when the system was off-line during maintenance, to observe specific events and/or monitor certain parameters. Where distributed systems are concerned, the limitations of static instrumentation have led to interests in *dynamic* instrumentation that can be applied (and removed) as required at runtime [6, 9, 11, 15]. Dynamic instrumentation can make use of instrumentation *services* such as gauge, monitor and probe services that can be dynamically attached to application components to measure specific runtime parameters and monitor their behaviour.

The *service-oriented* abstraction has fairly recently been adopted within middleware technologies and more generally distributed applications. This abstraction allows software components to join a dynamic federation and use its services and resources and offer services and resources of their own. This suggests that dynamic instrumentation may be developed using the same service-oriented abstraction. However, dynamic

[1] The term "instrumentation" is used henceforth to refer to software instrumentation.

instrumentation must provide additional functionality if it is to prove useful and flexible. In particular, the instrumentation must be *unobtrusive* in that it does not hinder the operation of application-level components and requires minimal extra programming effort to facilitate its integration. Dynamic instrumentation must also provide capabilities that allow it to be added and/or removed at runtime without having to disrupt the operation of a distributed system.

1.3 Statement of the Problem

The main research problem considered in thesis is:

> *"How can we develop an unobtrusive dynamic instrumentation architecture that can be used in conjunction with middleware technologies to further our understanding about the performance and behaviour of a distributed system?"*

To study this problem, the thesis sets out to address the following specific research questions:

- What are the types of parameters that need to be measured and monitored in a distributed application to assess performance and behaviour?
- What are the different types of instrumentation needed to measure and monitor these parameters?
- What are the programming and communication models required to facilitate the development and seamless integration of unobtrusive instrumentation with middleware and application-level components?
- How may we apply these models to develop a dynamic instrumentation architecture?
- How may we assess the performance of the instrumentation and quantify the overhead that it may introduce?

1.4 Aims and Objectives

The aims and objectives of the thesis fall into the two general categories of Analysis and Design/Implementation.

Analysis – which aims to provide a thorough analysis of instrumentation that delivers generic requirements, classification models and specifications relating to the use of instrumentation as a middleware service. In more detail, this will involve the following activities:

- A study of the fundamental and architectural models that are the foundations of distributed systems.
- A study of the programming and communication models provided by middleware technologies and used for the development of distributed systems.
- Analysis of the requirements of instrumentation services. These requirements will cover both functional requirements, concerned with the parameters that instrumentation services must measure and monitor and operational requirements, concerned with how instrumentation services coexist and interact with the distributed system under examination.
- A formal specification of the basic operations that facilitate the integration of instrumentation services within a distributed application.

Design/Implementation – which aims to develop an instrumentation architecture that implements the concepts and models emerging from the Analysis and demonstrates the use of dynamic instrumentation services. In more detail, this will involve the following activities:

- Design of an instrumentation architecture that provides a hierarchy of infrastructure and instrumentation services, which represent the different types of instrumentation service emerging from the analysis stage.
- Development of programming and communication models that represents the basic instrumentation service operations emerging from the analysis stage.
- Implementation of the programming and communication model to provide an architecture that supports dynamic instrumentation services, which can measure performance and monitor behaviour and provide seamless integration to external management agents.

- Testing/evaluation of the architecture through a series of realistic case studies that demonstrate its suitability for the runtime measurement and monitoring of distributed systems.
- Appraisal of the relative success of the work and suggestions for extension and/or future directions for others to consider.

1.5 Research Contributions

The main novel contributions to knowledge, emerging from the research stem from the various models and the instrumentation architecture developed throughout the thesis.

- Requirements model: The requirements model provides a unique analysis of instrumentation requirements from first principles, which is not addressed elsewhere in the literature. The requirements model examines the basic parameters to be measured/monitored and the different types of instrumentation and their functional and operational requirements. The requirements model is the culmination of previous research published by the author [15]. Primarily, the requirements model is intended to serve the remainder of the thesis, although it may prove useful to other practitioners in the field of distributed systems understanding.
- Classification model: The classification model provides an original classification of instrumentation services in terms of their roles and usage context. The classification model was developed from a previous instrumentation classification proposed by the author [15]. The classification model is general and may be used by other researchers to develop their own instrumentation system. The classification model also identifies a set of basic, or primitive, instrumentation services. In the absence of any instrumentation standard, the author chose a novel naming scheme for the instrumentation in line with their counterparts in conventional engineering or the physical sciences. It is anticipated that this may serve as a useful reference naming scheme for future researchers in the field of distributed software instrumentation.

- Formal analysis model: The formal model represents one of the few contributions that applies formal specification for the development of distributed instrumentation. Together with [16], (one of the few other contributions) it aims to emphasize the potential for formal modelling in the field of distributed instrumentation. The formal model considers the concept of an *abstract* instrument as the basis for functional instruments that provide instrumentation services. The formal model specifies the states and axioms governing an abstract instrument. The formal model was developed using *Object-Z*, which is an extension to the Z formal modelling language to accommodate object-orientation. Object-Z was chosen because of its support for object-orientation and ability to write specification which contain precise state models, strong typing and precise axioms.

- Programming and communication model: The programming model provides a novel contribution in that it allows instrumentation to be applied *unobtrusively*. In other words, applications can be instrumented with minimum disruption or additional coding to the application itself. The programming model facilitates the dynamic attachment and removal of instrumentation services at runtime with minimum disruption to application-level services. The application of the programming model is considered briefly in the author's previous research publications [15, 17, 18] and the thesis provides a more detailed coverage. The communication model describes the division of labour amongst instrumentation services and provides protocols to facilitate their interactions. The communication model allows basic, or primitive, instrumentation services to be grouped together to perform more complicated instrumentation tasks.

- Instrumentation architecture: The architecture combines the aspects of the requirements, classification, formal and programming models to provide a novel instrumentation API. The architecture consists of the infrastructure components and a small number of instrumentation services that can be used to measure/monitor distributed application components. The architecture is

described in the author's published work [15, 17, 18]. The architecture was developed using UML to provide an *extendable* instrumentation layer that sits in between core middleware services and application specific services. The architecture combines measurement and monitoring functionality together with the abstract operational specifications from the formal model. The focal point of the architecture is a small number of instrumentation services that can be instantiated to measure and monitor specific runtime parameters and behavioural information. These instrumentation services were chosen specifically to measure parameters of interest to the author, based on fifteen years previous experience working with distributed systems.

1.6 Scope of the Thesis

In general terms the thesis sets out to determine the instrumentation needs for distributed systems management and the manner in which this instrumentation may be applied. The thesis is not intended to serve as the definitive design of an instrumentation architecture, but to present a feasibility study in the development of an instrumentation architecture. The thesis investigates the notion of *on-demand* distributed software instrumentation, and the promotion of instrumentation as a new middleware service. This investigation is driven by the need to further our understanding of today's distributed systems, which are typically large and complex.

The culmination of the thesis is the development of a dynamic software instrumentation framework and the design of an instrumentation architecture. The architecture can be used to measure performance and monitor the behaviour of the software components that execute within a distributed system. The framework is intended to provide a *reference* framework that can be used by system architects, application developers and middleware technology providers as a basis for the development of subsequent instrumentation efforts.

The thesis is applicable to the class of distributed systems developed using a distributed object-based middleware. The work described is directly applicable to distributed systems developed using Jini middleware. The overall approach may be used in conjunction with other middleware technologies such as Java RMI, CORBA and even

Web Services although there are limitations to this applicability, which are discussed in the Conclusions (chapter 10). The architecture has been demonstrated for LAN-based distributed systems and as such issues of scale to cover wide-area systems have not been considered, although these are also mentioned in the Conclusions.

1.7 Thesis Structure

The thesis is structured as follows:

Chapter 2 describes the basic terminology, fundamental concepts and models relating to distributed systems.

Chapter 3 describes the programming models and technologies used for the development of distributed systems' application software. In particular, object-oriented middleware and distributed events are considered as the main programming technologies.

Chapter 4 presents a literature review of software instrumentation from its early foundations up to the "state of the art" practices of today.

Chapter 5 presents an informal requirements analysis to establish what instrumentation needs to measure/monitor and classification of different categories of instrumentation.

Chapter 6 presents a formal model of instrumentation services underpinned by the formal specification of the basic operations of an abstract instrument.

Chapter 7 presents an instrumentation architecture for measuring and monitoring applications. The architecture consists of the infrastructure components and a small number of instrumentation services that can be used to measure/monitor distributed application components.

Chapter 8 describes the implementation of the architecture using a combination of the Java programming language (J2SE v1.4) and Jini middleware technology

Chapter 9 describes how the architecture may be used through several instrumentation case-studies. Qualitative and quantitative analyses are also presented to assess the performance overhead of the instrumentation.

Chapter 10 draws overall conclusions on the novelty of the research and mentions directions for future related research.

Chapter 2

Distributed System Fundamentals

This chapter is intended to outline background information necessary to interpret the ideas presented in subsequent chapters. On occasion, the chapter draws on material presented in [19] to introduce the system models that help us understand and reason about the structure and behaviour of distributed systems. A brief overview of these models is necessary before moving on to consider the main topic of instrumentation in relation to distributed systems.

2.1 System Models

According to [20], a distributed system may be defined as:

> *"a collection of autonomous hosts that are connected through a computer network with each host executing service providing components and operating a distributed middleware to enable components to coordinate their activities giving the impression of a single, integrated computing facility".*

This definition essentially defines the elements that constitute a distributed system, but it does not explain the connectivity and placement of the constituent parts nor the relationships and interactions between them. In order to further our understanding of distributed systems and to reason about their performance and characteristics we must call on the *system models* that conceptualize and characterize distributed systems. In general, a model should contain only the essential elements that are required to understand and reason on some aspects of a given system's behaviour. With this in mind, a system model should address the following questions:

- What are the main elements in the system?
- How do these elements interact?

- What are the characteristics that affect their individual and collective behaviour?

The two categories of system models that we use to further our understanding are referred to as *architectural* models and *fundamental* models [19]. The architectural models are concerned with the placement of the constituent elements and the relationships that exist between them. Typical examples include the client-server model and the peer-to-peer model. The fundamental models are concerned with a more formal description of the properties that are common to all of the architectural models.

Four significant concerns that the fundamental models must address are: dependencies, timing, failure and security and these problems are addressed by the three fundamental models:

- The interaction model, which deals with dependency relationships, message passing interactions and the difficulties caused by timing.
- The failure model, which attempts to give a precise specification of the faults that may occur between components and/or communication channel. The failure model essentially defines correct components and reliable communications.
- The security model, which considers the possible threats to components and communication channels.

The remainder of this chapter briefly considers the architectural and fundamental models with a view to highlighting how instrumentation may be incorporated into the models to further our understanding and reasoning capabilities.

2.2 Architectural Models

The architecture of a system is essentially its structure in terms of the separate constituent elements – just like the architecture of a building. The overall goal of the architectural design is to ensure that this structure will meet both the present and the likely future demands placed on it. The major concerns at the architectural design stage are to make the system reliable, manageable, adaptable and cost-effective and, unsurprisingly, instrumentation can make valid contributions to each of these concerns. The architectural

design of a building shares similar aspects – it determines not only the appearance of the building, but also its general structure and architectural *style* (e.g. gothic, neo-classical, modern).

However, there is a significant difference between the architectural design of a distributed system and that of a building in that once a building is "cast in stone" the degree to which it can be altered or adapted is limited. In contrast, distributed systems may undergo significant alteration and adaptation achieved through architectural reconfigurations. As we shall see, the ability to achieve reconfiguration varies across different architectural models and some models are regarded as "fluid", whereas others are more "brittle". To determine the degree of reconfiguration, we need to know the current architectural model's state and dynamic instrumentation has the potential to provide this information.

2.2.1 Component Configurations

The main architectural models are also referred to as the *architectural styles* of distributed systems. The models or styles are based on the concept of *services* provided by communicating components engaged in message passing. An architectural model of a distributed system first simplifies and abstracts the functions of the individual components of a distributed system and then considers:

- The placement of these components across a network of computers – aiming to define useful patterns for the distribution of data and workload.
- The inter-relationships between the components – their functional roles and the patterns of communication between them.

An initial simplification of an architectural model is achieved by classifying components as *server* components, *client* components and *peer* components. The latter are components that provide services and communicate in a symmetrical manner to perform a task. This classification of components identifies the responsibilities of each and helps in assessing their workloads and determining the impact of failures in each type of component. The classification can also be used to specify the placement of components in a fashion that meets the performance and reliability goals of a system. The two most

widely used architectural models are client-server and peer-to-peer. These two models and variations on the basic client-server model are considered further in [19].

2.2.2 Software layers

The term software architecture referred originally to the structuring of software as layers or modules in a single computer and more recently in terms of the services offered and requested between components located in the same or different computers. The component/service-oriented view of a distributed system is often expressed in terms of software layers as shown in Figure 2.1

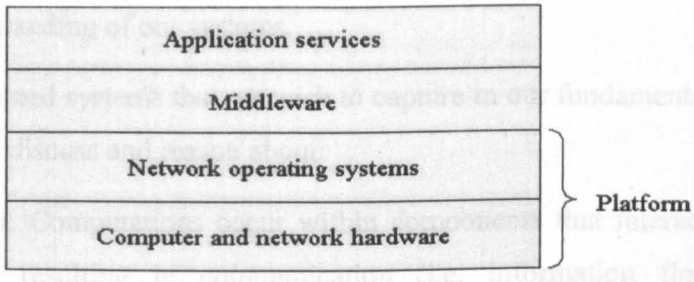


Figure 2.1: software layers

Figure 2.1, introduces the terms *platform* and *middleware*, which are defined as follows:

- Platform: the combination of hardware and network operating system layers are often referred to as the platform that supports the distributed system and its associated applications.
- Middleware: the layer of software that essentially bridges the gap between application components/services and the network operating system. Middleware, which is considered further in chapter 3, also masks heterogeneity and provides a convenient programming model for application developers.

2.3 Fundamental Models

The architectural models share the major design requirements, which are concerned primarily with the performance and reliability characteristics of components, services, networking and the distribution of resources in a system. In this section, we present

models based on the fundamental properties that allow us to be more specific about a system's characteristics and the failures and security risks they might exhibit.

There is much to be gained by knowing what our designs do, and do not, depend upon. Such *dependencies* will then allow us to decide whether a design will work if we try to implement it in a particular system, as we need only ask ourselves whether our assumptions hold in that system. Also, by making our assumptions clear and explicit, we can hope to prove system properties using mathematical techniques and these properties will then hold for any system that meets our assumptions. Finally, by abstracting only the essential system elements and characteristics away from the details such as hardware, we can clarify our understanding of our systems.

The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about:

- Interaction: Computations occur within components that interact by passing messages, resulting in communication (i.e. information flow) and co-ordination (synchronization and ordering of activities) between components. In the analysis and design of distributed systems we are concerned especially with these interactions. The interaction model must reflect the facts that components depend on each other and that communication delays may take place, thereby hindering co-ordination. The interaction model must also account for the difficulties of maintaining the same notion of time across all components in a distributed system.

- Failure: The correct operation of a distributed system is threatened whenever a fault occurs in any computer, component or the network that connects them. The failure model classifies such faults and provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

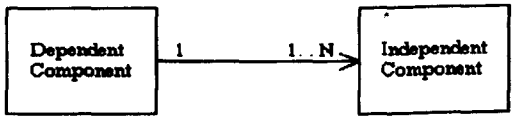
- Security: The modular nature of distributed system and their openness exposes them to attack by both external and internal agents. The security model defines and classifies the forms that such attacks may take, thereby providing

a basis for the analysis of threats to a system and for the design of systems that are able to resist them.

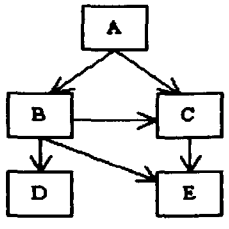
Detailed consideration of the fundamental models exceeds the scope of the thesis. We must emphasize at this stage that security aspects (relating to the security model) exceed the scope of the instrumentation architecture presented in the thesis. Security is only mentioned as one of the possible areas for future follow on research in the Conclusions.

However, before concluding this background chapter it is important to describe the concept of dependencies in distributed systems. This is necessary as dependencies are one of several significant aspects that the instrumentation intends to uncover. A dependency relationship is established when one component depends upon another component. To be more specific we may say that a dependency occurs when one component depends on the *services* provided by another component.

We consider service dependencies as opposed to component dependencies because a service can be used to represent a single *logical* concept such as a chat-room or printer service. Service dependencies may be modelled as a directed graph (digraph) in which a directed arc or edge implies that a certain node, or component, uses the service provided by another component(s). The directed edges are used to represent the service dependencies and nodes are used to identify the components that feature in the dependency relationships. Figure 2.2, shows a simple example that represents the dependencies between components *A*, *B*, *C*, *D* and *E*.



(a) Conceptual representation of services dependencies



(b) Service dependencies between components A, B, C, D, E

Figure 2.2: service dependencies

However, things are somewhat more complicated because service dependencies are *dynamic* since a component's state may change, giving rise to changes in its dependencies. In addition, a client component may only use, and depend upon, a service for a specific time period and after the expiry of this time period the dependency ceases to exist. This dynamic behaviour suggests that service dependencies, and hence the directed edges used to represent them, have a *lifetime* that must be must represented by dynamically maintaining the digraph to provide a faithful representation of the distributed application's behaviour.

The dynamics of service dependencies suggest a need for facilities that assist *dependency management.* As we shall see later in the thesis, instrumentation can be used to assist dependency management. This issue is explored further in chapters 7 and 8, which are concerned with the development of the instrumentation architecture and its implementation respectively.

2.4 Chapter Summary

This chapter has considered the architectural and fundamental models that further our understanding and reasoning capabilities in relation to the structure and behaviour of distributed systems. These models help in understanding and appreciating the requirements that must be met in the development of distributed systems. This appreciation will also help in the development of an instrumentation architecture that furthers an understanding of the structure and runtime behaviour of distributed systems. The next chapter considers the development of distributed systems and particularly the programming models and technologies that are used to develop the service providing components that constitute distributed systems.

Chapter 3

Distributed System Development

In this chapter we consider the network transport and software technologies that facilitate communication in distributed systems and provide *programming models* to assist the development of distributed application software. The chapter begins by introducing middleware as software technology used to bridge the gap between network operating system and application level software. The bulk of the chapter concentrates on utilities that middleware provides and in particular, outlines the principles of distributed object-based middleware and distributed events and notification. This is necessary because object-based middleware will feature in chapters 7 and 8, which are concerned with the design and implementation of the instrumentation architecture respectively. It is assumed that the reader has some familiarity with distributed systems development technologies, particularly object-based middleware.

3.1 Distributed Programming Models

The communication paradigm of distributed systems is that of *message passing*. Over the years different programming models have been developed to support the underlying message passing paradigm. These programming models range from the low-level socket abstraction, through to Remote Procedure Calls (RPC) right up to higher-level object-based Remote Method Invocation (RMI) and service-oriented abstractions. The socket model provides an API for the Internet protocols. RPC, RMI and service-oriented abstractions constitute *middleware*, which abstract sockets and, amongst other things, mask heterogeneity. The consideration of these models is important because not only will these models be used in the development of the instrumentation architecture, but they also reveal the protocols, constructs and mechanisms that allow communication between distributed resources.

Middleware was introduced in chapter 2 as a software layer that sits between the network operating system and application level components and their services. The term *Network Operating System (NOS)* is used to refer to an operating system that has in-built networking facilities that may be used to access remote resources. However, a NOS still retains the autonomy of its host such that, while remote resources can be accessed, a NOS cannot control or schedule remote processes in some other host. In the present climate, middleware plays a crucial role in the development and functioning of distributed systems.

The concept of middleware for distributed systems arose in response to increasing *heterogeneity* in computer systems. The growth of the Internet and the number of services relying on it forced developers to create standard APIs, which hide the underlying technologies. The term middleware was given to such APIs because they resided "in the middle" of the lower level platform layer and higher-level applications.

Generally middleware masks heterogeneity in network technology and hardware (host CPU). Heterogeneity in operating system is also usually masked. Depending on the type of middleware, heterogeneity in programming language and vendor implementation may sometimes be masked.

In addition to masking heterogeneity middleware also provides *transparency*, which is highly desirable in distributed systems. Transparency is the ability to conceal all the details of distribution so as to make things appear as a local setup. The ANSA Reference Manual 01.00 [21] identifies eight forms of transparency: access, location, concurrency, replication, failure, mobility, performance and scaling transparencies, which are described further in [19]. Location transparency and concurrency transparency are always provided by middleware. Depending on the type of middleware, some levels of replication, failure and mobility transparency may also be provided.

3.2 Object-Oriented Middleware - Distributed Objects

The term *Distributed Object Technology* is synonymous with object-oriented middleware. However, the term provides a greater sense of identity, suggesting that distributed object technology is a new paradigm rather than just middleware in an object-oriented flavour. Distributed object technology combines a distributed object model with

protocols and infrastructure services that allow objects to be spread over a network so that they may communicate with each other. The main operations required of the majority of distributed object technologies are:

- Creation of remote objects – a remote object is created in an address space and given some initial state value.
- Location of remote objects – the location of remote objects involves placing an object and its associated files somewhere on a network from where it may be accessed and used by clients.
- Method invocations on remote objects – in RPC systems the unit of communication is a procedure call. Remote objects encapsulate data and provide methods for accessing the data. Therefore in distributed object-based systems communication is achieved through method invocations made on remote objects, which return the results back to the caller.
- Deletion of remote objects – a remote object will consume resources during its lifetime. To reclaim these resources, remote objects need to be deleted when they are no longer needed.

As we shall see in subsequent chapters, the instrumentation services, amongst other things, are capable of providing either direct or indirect information relating to these operations.

An understanding of the principles underlying the majority of distributed object technologies is important to the remainder of this thesis. These principles are described below in relation to Java RMI which is a relatively simple Java-based middleware technology. Jini middleware technology is mentioned throughout the thesis and it is described further in chapter 8.

3.2.1 Basic principles of distributed objects

Distributed objects are objects that exist in an address space and offer methods that can be subjected to remote method invocations (RMI calls) from objects in separate address spaces. Typically, a separate address space may be a different virtual machine on the same computer or a virtual machine running in a different computer connected by a

network. By convention, the code issuing the call is referred to as the *client* and the target object (on which the method is invoked) is referred to as the *server* object (or *remote* object). A crucial aspect of distributed object technology is to make the remote nature of the call *transparent* so that, from the programmer's perspective, there is no (or extremely little) difference between remote and local calls.

A remote call is simplified by separating it into a *request* (asking for a service) and a *response* (sending results back to the client), which are considered further in subsequent sections. From the point of view of the client, the request and response can be completed as one atomic action, which is referred to as a *synchronous* call. Alternatively, they can be separated, such that the client issues a request and then issues a wait for a response, which is referred to as a *deferred-synchronous* call. In some cases the response part may be empty (i.e. no values are returned to the client), which is referred to as a call on a *one-way* method. Calls on one-way methods can be asynchronous since the client does not need to wait until the call is finished.

The main entities and concepts that make up remote method calls are:

- Remote objects – on which the client wants to call a remote method.
- Remote references - to identify the network location of the target remote object.
- Remote interfaces – to specify the methods of a remote object that are available for invocation.
- Interface Definition Language (IDL) - a language that can be used to specify a remote interface and deal with heterogeneity between different programming languages.
- Proxies - lightweight objects, used at both the client and the server, which "trick or fool" the real client/remote object into thinking that they are the real remote object/client respectively.
- Marshalling - serialization into byte stream and transmission across the network. Conversely, unmarshalling is the receipt of a serialized byte stream and the reconstitution into the original data structures and objects.

It is assumed that the reader has some prior knowledge of these entities as they will be used in subsequent sections to describe the basic principles and operation of Java RMI. The following sections describe communication in distributed object systems using material drawn from [22].

3.2.2 Distributed object communication

Remote method invocation is the main means of communication between distributed objects. Remote method invocation has its origin in the remote procedure call mechanism (RPC) and is regarded as the object-oriented version of RPC. Essentially, the remote method invocation process provides a protocol that specifies how distributed objects interact. Remote method invocation occurs in Java RMI, CORBA and Jini, although people tend to associate remote method invocation with Sun Microsystem's Java RMI implementation. There are slight differences between remote method invocation in Java RMI, CORBA and Jini however, we may consider an archetypal remote method invocation as shown below in Figure 3.1.

Figure 3.1: archetypal RMI software layers [22]

Figure 3.1 shows the archetypal software layers that sit below the client and server objects and facilitate remote method invocation. Figure 3.1 also shows an object registry, which the client object searches in order to obtain a remote reference to the remote server object. Of course for the client to find such a reference the remote server object must register a remote reference in the first place. The object registry is regarded as a core

service and its implementation varies for Java RMI, CORBA and Jini. The two main implementations are: a Naming Service, where the client looks up a reference "by name" (like a White Pages telephone directory), or a Directory Service, where the client looks up a reference "by type" (like a Yellow Pages telephone directory).

Figure 3.1 shows the logical path of the RMI call, which is the path that the client "thinks" is being followed. However, the call actually follows the physical path through the software layers that make up the RMI infrastructure software. On the client side, the call is passed to the client proxy, which represents the remote server object at the client. The runtime support layer is responsible for the interprocess communication required to transmit the call to the remote server's host, including the marshalling of the call parameters. The network support layer is responsible for establishing socket connections with the remote server's host and the implementation of the network transport layer. The software layers on the server side provide the same functionality only in the reverse sense, such that the RMI call is accepted, processed and any result is returned to the client.

We may pause at this stage to note two of the challenges that this process presents to the instrumentation architecture:

- For the instrumentation services to acknowledge and monitor RMI calls, we need a mechanism that intervenes in the call somewhere along the physical path
- When a client obtains a remote reference to a remote server object it becomes dependent on the remote server, until the remote reference becomes null. A different type of instrumentation service is needed to record such dependencies and therefore provide a picture of "what depends on what" within a particular application.

We shall see how these two challenges are addressed later in chapters 7 and 8. However, we may now proceed to describe Java RMI's implementation of the archetypal model.

3.2.3 Java RMI

Java RMI provides a Java-based implementation of the RMI protocol. Java RMI is the simplest of the distributed object technologies and provides a good starting point for those keen to learn how to develop distributed object applications. Figure 3.2 shows the Java RMI implementation of the archetypal model.

Figure 3.2: Java RMI layers [22]

The client and server proxies are represented as *stub* and *skeleton* files respectively, which, are generated using Java's `rmic` compiler. The remote reference layer understands how to interpret and manage references made from clients to the remote server objects. The transport layer establishes and maintains the socket connections between the client and server hosts. Java Remote Method Protocol (JRMP) is main transport protocol used to transfer data across the network and RMI-over-IIOP.

Java RMI provides a simple Naming registry, `rmiregistry`, which allows clients to lookup a remote reference by name. The `rmiregistry` maintains a table, which maps textual URL names to remote references hosted on a particular host. The `rmiregisty` is accessed using Java's Naming class, whose methods take a URL string as:

```
rmi://host:port/RemoteObject
```

where `rmi` is the protocol, `host` and `port` refer to the location of the `rmiregistry` process and `RemoteObject` is the name "bound" to the remote server object. The protocol may be omitted since RMI is implied. If the `host:port` are omitted then the local computer (localhost) is assumed on the default port (1099). The server registers the object using the `Naming.rebind` method as:

```
Naming.rebind("RemoteObject", remoteObject);
```

Where `remoteObject` is an instance of the remote server object that will process RMI calls. The client obtains a remote reference using the `Naming.lookup` method as:

```
remoteObject =
    (remoteObject)Naming.lookup("rmi://cmsdreil/RemoteObject");
```

When the client has a valid remote reference it is in a position to make RMI calls on the remote object.

If we assume that the client and remote server objects are located on separate hosts then the stub file needs to be transferred to the client's host. This could be done manually, but this is often inconvenient and defeats the objective of "self-serving" distributed systems. The preferred approach is to have the stub file downloaded automatically to the client from the remote server. This is actually one of Java RMI's most significant capabilities, namely the ability to dynamically download Java class files from any URL to a JVM running in a separate host [23].

3.3 Distributed Events and Notification

RMI calls provide a *synchronous* means of communication based on a request/reply pair. Essentially, the client (or caller) is required to wait to receive until a reply, or an exception is received from the server. The one exception to this rule is CORBA's one-way calls, which do provide an *asynchronous* mode of communication in that the client is not held up waiting for a reply. In the main we may regard RMI calls as providing a synchronous means of communication based on a request/reply pair.

While RMI calls provide an effective means of communication, their reliance on a request/reply makes them unsuitable for dealing with asynchronous communication. Event-based systems do support asynchronous communication through a process of *notification*. Objects *subscribe* to receive notifications of certain types of events. An

object may generate or publish an event, which is then sent as a notification to the subscriber.

3.3.1 Overview of Distributed Events

The idea behind the use of events is that one object can react to a change occurring in the state of another object. Notifications of events are asynchronous and determined by their receivers. Events provide a natural model for dealing with certain phenomena, which occur in conventional single-address space computer applications. For example, in interactive GUI applications, the actions that the user performs on objects, such as manipulating a button with a mouse, are seen as events that cause changes in the objects that maintain the state of the application. The objects that are responsible for displaying a view of the current state are notified whenever the state changes. Later in chapters 7 and 8 we shall see how events can be used to notify when an instrumentation service changes its state.

In general, a distributed event system has a different set of characteristics and requirements than that of a single-address-space event system. Notifications of events from remote objects may arrive in different orders on different clients, or may not arrive at all. The time it takes for a notification to arrive may be long (in comparison to the time for computation at either the object that generated the notification or the object interested in the notification). There may be occasions in which the object wishing the event notification does not wish to have that notification as soon as possible, but only on some schedule determined by the recipient. There may even be times when the object that registered interest in the event is not the object to which a notification of the event should be sent (third-party objects).

A significant feature of distributed notification is the ability to place a third-party object between the object that generates the notification and the party that ultimately wishes to receive the notification. Such third parties, which can be strung together in arbitrary ways, allow ways of off-loading notifications from objects, implementing various delivery guarantees, storing of notifications until needed or desired by a recipient, and the filtering and rerouting of notifications.

The remainder of this section briefly introduces distributed events in Jini. As with RMI calls, the significance of distributed events is twofold: first they are used to allow instrumentation services to communicate with one another and second, instrumentation services must be capable of recording application-level events. The description to follow makes use of material provided in the Jini Distributed Event Specification [24].

3.3.2 Jini Distributed Events

Jini middleware has already been mentioned previously as a Java based middleware that supports standard Java RMI proxies. Jini also supports non-Java RMI proxies (i.e. stubs) and smart proxies. It should be pointed out that Jini is not just "a more substantial RMI". Jini goes much further than Java RMI in providing a programming model based on *services* and *federations*, which are dynamic collections of services that other services may join and leave dynamically. The most recent version of Jini (Jini 2.0) provides Jini Extensible Remote Invocation (JERI) [25], which is essentially a new RMI programming model.

Jini is the middleware technology used to implement the instrumentation architecture and a thorough consideration of Jini is provided in the implementation chapter – chapter 8. This section is only concerned with Jini's support for asynchronous distributed events. Jini distributed events will also be considered in chapter 8 for notification of changes in state in instrumentation services.

The basic, concrete objects involved in a distributed event system are listed below and their relationships are illustrated in the basic event model of Figure 3.3.

- The object that registers interest in an event,
- The object in which an event occurs (referred to as the event generator),
- The recipient of event notifications (referred to as a remote event listener),
- The remote event itself (an object passed from generator to listener to indicate an event occurred).

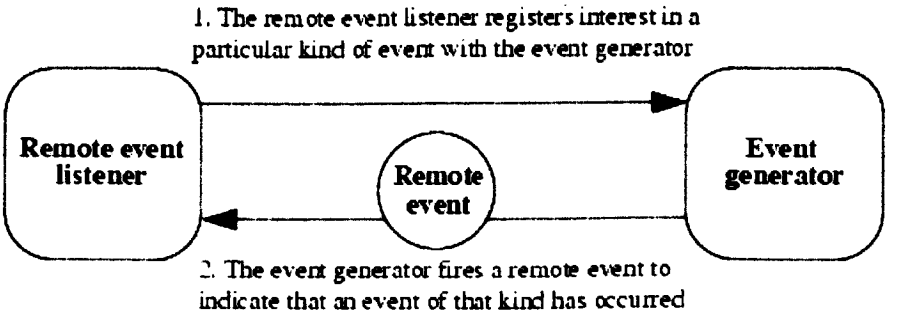


Figure 3.3: distributed event objects

An *event generator* is an object that may undergo some form of state change that might be of interest to other objects and allows other objects to register interest in those events. This is the object that will generate notifications when events of this kind occur, sending those notifications to the event listeners that were indicated as targets in the calls that registered interest in that kind of event.

A *remote event listener* is an object that is interested in the occurrence of some kinds of events in some other object. The major function of a remote event listener is to receive notifications of the occurrence of an event in some other object (or set of objects).

A *remote event* is an object that is passed from an event generator to a remote event listener to indicate that an event of a particular kind has occurred. At a minimum, a remote event contains information about the kind of event that has occurred, a reference to the object in which the event occurred, and a sequence number allowing identification of the particular instance of the event. A notification will also include an object that was supplied by the object that registered interest in the kind of event as part of the registration call.

Jini events will also feature later in chapter 8, which considers their use within the instrumentation architecture.

3.4 Distributed Component Technologies

Distributed objects provide the main building blocks for building distributed-object based applications. However, as these systems grew in terms of scale and complexity it was realized that more sophisticated software engineering practices were needed to ease both

the development process and the overall maintainability of distributed applications. This realization led to the concept of distributed component technologies that utilized the principles of software component engineering to allow applications to be assembled from *off-the-shelf* components.

3.4.1 Component Concepts

Components can be regarded as a collection of objects, which communicate to provide a set of services. Components are typically deployed as standalone entities in suitable formats such as Dynamic Link Library files (DLL) or Java Archive files (JAR). Components provide well defined interfaces through other components can use their services. This gives the impression that components are connected via interfaces and provides an attractive *component-connector* model for developers.

In many ways, software components are analogous to integrated circuits (ICs), which are used in electronics. ICs are "black boxes" that encapsulate functionality and provide services based on a specification. Developing applications with software components is analogous to wiring together ICs to build a complex circuit instead of using discrete components (resistors, capacitors, inductors etc.).

Local component technologies emerged from object-oriented programming to assist application development through a coarse-grained assemblage of components. Microsoft's Component Object Model (COM) was one of the earliest local component technologies. ActiveX controls are among the many types of components that use COM technologies to provide interoperability with other types of COM components and services. Sun Microsystems introduced its JavaBeans technology to bring components to the Java world.

In a similar fashion, distributed component technologies emerged as a natural progression from object-oriented middleware. Local components exist in the same host and communicate by sending events to each other in a publish/subscribe fashion. Distributed components combine the concepts of local component technologies with the communication principles of object-oriented middleware to allow components to communicate across hosts. In general, communication is achieved using the techniques and protocols of object-oriented middleware (i.e. RMI or RPC).

Distributed component technologies include Microsoft's DCOM (an extension to COM), CORBA's Component Object Model (CCM) and Sun Microsystem's Enterprise JavaBeans (EJB) an extension to JavaBeans. A thorough description of DCOM. CCM and EJB exceeds the scope of the thesis. However, for completeness, to conclude this section we describe how components can be regarded as entities that provide and use services provided by other components. This allows us to regard a distributed component-based application from a *service-oriented* view point.

3.4.2 Service-oriented abstraction

Sun Microsystem's Jini technology is not strictly marketed as a component technology. Jini was initially developed as a connection technology intended for connecting small-footprint computing devices to a network. However, Jini has matured into a Java-based distributed middleware technology that allows applications to be developed as a collection or *federation* of services. Jini as such does not provide comparable component facilities to EJB (i.e. beans, builder tools, containers etc.), but it can still be used to develop component-based applications.

In such applications, server components consist of several objects, which communicate to provide a set of services. Jini tends to pay more attention to the distribution of services that distributed components provide rather than the components themselves. Jini's service-oriented view regards an application as a federation of services that can be used dynamically by clients and other servers. During the federation's lifecycle existing service providing components may leave the federation or adapt the services they provide. Similarly, new service providing components may join the federation to increase the range of services provided.

More concerted efforts towards distributed service-oriented development comes from technologies such as Openwings [26]. Openwings is a service-oriented framework that provides a variety of core services including: component services, connector services, platform services and data services. Currently, Openwings builds on top of Java and Jini technologies to provide a more complete solution to distributed service-oriented development. However, Openwings is not specifically tied to Jini.

More recently, service-oriented development has followed along the lines of Web Services and Microsoft's .NET, which provides software technology for developing applications based on Web Services. However, Web Services are not based on any distributed object technology (unlike Java RMI, CORBA, DCOM and Jini). A common misconception is to compare Web Services alongside Java RMI, CORBA, DCOM, and Jini. Web Services do support distributed systems computing technologies, but that is where the common ground ends [27]. The only possible relation is that Web Services are now sometimes deployed in areas where distributed object and component applications have failed in the past. For this reason, the approach to instrumentation described in the thesis is not directly applicable to Web Services.

The thesis concentrates on monitoring the behaviour of service providing components and in turn the services they provide. The components that are monitored are based on some distributed object model and communication is based on remote method calls and event notifications. In order to monitor such components, the instrumentation services first need to acquire certain structural information about the components. Such information includes the collection of objects that make up the component and the other components that a component depends upon for service provision (i.e. service dependencies).

The instrumentation services must also be capable of acquiring dynamic or behavioural information exhibited by components. The "window" into this behavioural information is the passive monitoring of remote method calls and events generated and received by the component. In terms of the previous IC analogy, this is similar to attaching high impedance probe to the IC's pins and observing the signals on an oscilloscope. Essentially, the acquisition of a component's structural and behavioural information sets the challenge for the remainder of the thesis.

3.5 Chapter Summary

This chapter has considered the programming models that are used to develop distributed applications. The chapter has concentrated mainly on object-oriented middleware and inter-object communication using remote method invocation and distributed events. Attention has focused on Java RMI as it is a relatively simple middleware to consider.

The consideration of object-oriented middleware has been necessary in order to reveal the low-level dynamic interactions that the instrumentation services must be aware of. In particular, it is crucial that the instrumentation services are capable of intercepting RMI calls and events. The next chapter moves on to consider previous and current research concerned with monitoring and generally understanding distributed systems.

Chapter 4

Review of Software Instrumentation Research

This chapter reviews the current developments in software instrumentation. The chapter begins with a brief introduction to the historical background to software instrumentation. The chapter then goes on to describe of the recent "state of the art" developments, aimed at using software instrumentation in distributed applications. The chapter concludes by outlining the contribution of this thesis, which extends on previous instrumentation developments and refines several ideas considered in the more recent state of the art developments. The aim of the chapter is not only to review current instrumentation developments, but to highlight the problems facing software instrumentation development and the techniques used to address these problems.

4.1 Historical Considerations

The basic technique of software instrumentation is the insertion of instrumentation code at points of interest throughout a program so that we may further our understanding of the program when it executes. Originally, this technique was used to examine and monitor programs that run on single processor machines and for analyzing the performance of real-time systems. The parallel computing community later adopted instrumentation to debug, tune and visualize parallel applications [28-32]. More recently distributed system developers have recognized the potentials of instrumentation, used in a *dynamic* regime, to monitor and manage today's distributed applications.

Probably the earliest documented use of software instrumentation was that of Satterthwaite [33] who developed an integrated debugging system for use with ALGOL W[2] programs. Although [33] does not described the system as such, [34] regards this system as an example of a *dynamic analyzer* and goes on to describe the two fundamental parts:

[2] ALGOL W is an approximate extension to ALGOL 60 with additional list processing facilities and data types.

- An instrumentation part – which adds instrumentation statements to a program either while it is being compiled or before compilation. When the program is executed these statements gather and collate information on how many times each program statement is executed.
- A display part – which collects the information provided by the instrumentation statements and prints it out in a form which can be understood by the reader. Typically this produces a program listing where each line is annotated with the number of times the line has been executed.

[34] mentions how dynamic analyzers were used as part of a more comprehensive test environment for early programming languages (e.g. ALGOL and FORTRAN). They were also closely associated (even integrated) with the compiler, through which they could be switched on or off by a compiler directive. Two of the main problems of dynamic analyzers, as noted in [34], were:

- They relied on source code instrumentation, which was not always possible when the program relied on additional pre-compiled libraries.
- The instrumentation code often affected program performance, which presented problems in real-time applications.

The term 'dynamic' is used in the sense that code insertions revealed dynamic (runtime) information relating to a program. This is not to be confused with the dynamic concepts described in this thesis. The thesis uses the term dynamic in relation to components, which can provide instrumentation services on-demand and can be moved around a network so that they can monitor messages and events that pass between remotely located components.

Although this early approach may seem trivial in comparison to current software instrumentation approaches, it was regarded as advanced at the time. The approach also laid the foundation for modern profiling tools, such as `gprof` [35]. Satterthwaite's work helped programmers in understanding the behaviour of their programs and even went some way to helping them to write better programs. It also defined an important

landmark, which was the realization that program testing and traditional unobtrusive approaches to debugging were insufficient to understanding a program.

In some ways this may be thought of as the first documented point from which program instrumentation and monitoring systems were looked at with interest. However, as in many scientific disciplines, change fuels further advancement and this was certainly the case were software instrumentation was concerned. The change in question was the move from uni-processor computing towards *parallel and distributed computing.*

4.2 State of the Art Developments

The more recent state of the art developments take advantage of distributed object and component-based middleware technologies to provide instrumentation capabilities. Through these technologies instrumentation may be developed as components (e.g. Enterprise Java Beans) or as services, similar to the core services provided by middleware technologies themselves. Several researchers have also used *Reflective Middleware*, which uses instrumentation concepts as part of a more comprehensive system capable of reflecting on a system's structure and behaviour at runtime. *Aspect-Oriented Programming (AOP)* is another approach which is emerging as a popular means for the management and monitoring of enterprise-level applications, through its separation of concerns.

The following sections describe several significant recent state of the art developments in terms of their design, their capabilities and their scope. It must be highlighted that several of the developments below go further than simply providing instrumentation capabilities in that they provide more comprehensive distributed application management frameworks. In contrast, the thesis only considers instrumentation and by doing so, describes a framework on a different slant to these other recent developments. The different slant is that of a thorough academic treatment of dynamic instrumentation, from requirements and formal analysis through to design, implementation and evaluation.

4.2.1 Monitoring Distributed Object and Component Communication (MODOCC)

Monitoring Distributed Object and Component Communication (MODOCC) is a system designed to monitor behaviour in middleware-based applications. A complete description

of MODOCC, ranging from initial motivation and inception through to implementation and evaluation is provided in [6]. MODOCC regards the process of *software monitoring* as that of observing various aspects of the execution of some *monitored application*. An application is prepared for monitoring by *instrumenting* the application and this involves adding or changing something in the application or its execution environment. The party interested in monitoring is a *monitoring application*. A *monitoring system* supports the monitoring by performing *measurements* (*monitoring data*) and packaging the results for presentation to the monitoring application.

The notation bears some similarity to that used in this thesis. An application is monitored by the addition/removal of instrumentation. However, as will become apparent in subsequent chapters, this thesis also regards a monitor as one of several specific instrumentation units. This thesis uses the term *management agent* to refer to the party that uses the monitoring data to make sense of what is happening in the application and also to impart changes to the applications behaviour or its execution environment.

[6] uses a *separation of concerns* to decompose the monitoring system into three vertical layers or tiers, shown below.

Figure 4.1: MODOCC – decomposition of the monitoring system [6]

The monitor tier contains monitors, which are used to concentrate the knowledge that a monitoring system has about a monitored application. Monitors request and receive data

from a monitored application on behalf of the monitoring system. The instrumentation tier represents the monitored application within the monitoring system. This tier contains instrumentation units which include all the software components added to or modified in a monitored application and/or its execution environment. The Monitoring Support System tier (MSS) performs monitoring activities independent from the specific monitoring application domain and monitored application domain. An example is the dissemination of monitoring data by way of collecting monitoring data from instrumentation units and delivery to a monitor(s) in the monitor tier.

Four basic groups of monitoring activities are identified, each of which are carried out or shared by the three tiers, as shown in Figure 4.2.

Figure 4.2: MODOCC – monitoring activities [6]

- Generation: involves instrumentation units measuring and packaging monitoring data which is made available to the MSS.
- Dissemination: involves the MSS collecting monitoring data from an instrumentation unit and delivering it to interested monitors.
- Processing: involves the MSS analyzing the monitoring data from instrumentation and converts it into a format and level of detail suitable for monitors.
- Presentation: involves a monitor offering a view on monitoring data form MSS appropriate for the monitoring application

MODOCC describes and evaluates several earlier monitoring systems. The evaluation criteria are: architecture, middleware instrumentation for monitoring communication behaviour, support for analysis of concurrent activities and the overhead incurred by the monitoring system. The systems evaluated are: OLT [36], HiFi [37], MOTEL [16, 38, 39] and MIMO [40]. Of these systems, OLT is a commercial tool available from IBM and the others are academic research projects with some industrial participation. OLT and MOTEL are briefly mentioned below due to their sharing of some common ground with this thesis.

Object Level Trace (OLT) is part of IBM's Distributed Debugger, which is a client/server application for detecting and diagnosing errors in programs running across networked hosts. OLT models distributed applications at three levels: hosts, processes/threads and objects. A host is regarded as an execution environment, such as a Java Virtual Machine (JVM) running an instance of the WebSphere Application Server (WAS). A process/thread represents a unit of sequential execution and an object represents a programming language level object. Figure 4.3, taken from [6], illustrates the architecture of OLT.

Figure 4.3: OLT architecture [6]

The sequence of operations is relatively self-explanatory due to the annotated arrows shown in the figure and a detailed description of the mechanics of the architecture is provided in [6]. It should be highlighted that the OLT Runtime and Debugger Engine need to be installed on each host running WAS.

The main interest in OLT from the point of view of this thesis is its ability to monitor Java RMI remote objects. The main emphasis is on debugging Java RMI applications and little support is provided for dealing with CORBA applications. However, it is still possible to deal with CORBA indirectly through the support provided for CORBA in J2SE from v1.3 upwards. OLT automatically performs the necessary instrumentation so that developers can concentrate on debugging.

Of even greater interest is OLT's ability to record Java method calls made by clients on distributed objects, servlets, JSPs or EJBs residing on WAS. Unfortunately, little detail could be found regarding the internal workings of this feature of OLT (probably due to the commerciality). It is however known that OLT uses the debugging mode of the JVM to intercept the application execution to perform measurement. As will become apparent later the recording of remote method calls is also a feature of the instrumentation architecture described in this thesis. The approach used herein is to intervene on method calls and make them available as "first-class" objects in order to gain an insight into the dynamic behaviour of Java-based distributed applications.

The MOTEL system demonstrates how formal techniques can be applied to the analysis of middleware-based applications. This thesis also describes a formal model to formally represent instrumentation services, to be considered in chapter 6. However, the MOTEL approach is somewhat different. MOTEL uses formal methods to represent constraints/properties against which the behaviour of communication services may be monitored at runtime. The formal languages also differ in that this thesis uses Timed Communicating Object Z (TCOZ) [41] and MOTEL uses Linear-time Temporal Logic (LTL) [42].

[6] goes on to describe a design approach for *Generic Monitoring Systems (GMS)*. The design approach commences from the three tier model described above and considers a series of design questions which lead to a design process consisting of four stages:

- GMS design,
- GMS specialization,
- Instrumentation design,

- Monitor design.

These four stages are illustrated in Figure 4.4.

Figure 4.4: MODOCC – decomposition of design process [6]

The first stage is the design of a Generic Monitoring System (GMS). Here designers generalize the monitoring requirements to develop a GMS that works with monitoring data in an independent manner. The GMS represents a generalization of an MSS and provides an independent architecture that carries the benefits of future reuse and scalability.

In the second stage the GMS is specialized to provide an MSS suitable for monitoring a particular monitored application. At this stage designers also define a monitoring data structure from which concrete data structures can be created to represent instances of features to be monitored. The other main consideration at this stage is how the MSS will process the monitoring data coming from the instrumentation. The third stage deals with the design of the instrumentation, which is considered further below. The fourth stage deals with the design of monitors that can analyze monitoring data according to some monitoring model.

It is important to review the Instrumentation Design step of [6] because instrumentation is the main concern of this thesis. This thesis does not cover the design of a complete monitoring system in quite the same depth as [6]. This thesis does however provide a thorough coverage of instrumentation ranging from conceptual ideas, through to formal

modelling, design and implementation. With this in mind it is important to appreciate approaches to instrumentation design adopted by other researchers such as [6].

MODOCC considers Instrumentation Design explicitly in a separate step because the quality of the design determines the performance of the whole monitoring system. The instrumentation is modelled as a collection of one or more sensors. In the general case a sensor represents a device that responds to a physical stimulus (heat, light, sound etc.) and transmits a resulting impulse (as a measurement or the operation of a control). In MODOCC a software sensor is a small computer program that generates some data output when the environment in which it operates meets some condition defined in the sensor program. The sensor is said to 'trigger' the instrumentation to generate a monitoring report from its output.

Sensors feature in the overall design instrumentation design which consists of four main steps: sensor design, sensor placement, design of instrumentation tools and definition of the instrumentation architecture. During the sensor design stage designers will identify the need for individual sensors and exactly what these sensors are required to measure. For example, designers may want a sensor to detect specific events or measure a status. Designers will also choose data structures relating to the outputs and information content of sensors.

Sensor placement involves determining where and how to position sensors in relation to the monitored application. Several structured sensor placement techniques may be used:

- Using available APIs – some applications or execution environments may provide APIs suitable for monitoring. Examples include operating system-level notification mechanisms and middleware mechanisms such as CORBAs interceptors [43].
- Source code modification – direct modification of the application's source code to install sensors.
- Binary code modification – direct modification of the application's binaries, used typically when the source code is unavailable.

- Wrapping – a new component replaces an existing part of an application and when the new component is invoked it may trigger sensors that have already been embedded within.
- Hardware – the introduction of a hardware sensor to measure information about an application's execution in a non-obtrusive way. For example, "network sniffing" using an Ethernet card to read all traffic on a network.

Instrumentation tools are developed to automate the placement of sensors either at API source or binary level. After sensor placement the tools may also compile, build, package and deploy the modified application. The two main types of instrumentation tool are: design-time tools and runtime tools. In the final step of the Instrumentation Design process designers define the internal architecture of the instrumentation. The architecture defines the functional blocks that manage all the sensors and collect sensor data and present this data to the MSS – similar to "wiring-up" all the sensors and providing control and data capture functionality.

MODOCC represents a significant advance in the promotion of instrumentation to assist the understanding of distributed applications. It provides a thorough systematic study of instrumentation for use in distributed systems from the design stages of a generic Monitoring System through to the MODOCC system itself. As mentioned already there are similarities with the work described in the thesis. Furthermore, the timing of MODOCC and the thesis (circa. mid-2000) suggests that both are working towards the same aim – the promotion of instrumentation to assist distributed system developers.

4.2.2 Java Management extensions (JMX)

Java Management extensions (JMX) is a specification developed by Sun Microsystems that provides a complete application and network management specification. Although these management capabilities exceed the scope of the thesis, it is worthwhile considering JMX to see how instrumentation features in the specification. The JMX specification defines an architecture, design patterns, APIs and the services for application and network management in the Java programming language. The full JMX specification is provided in [24] and the following section draws on material in [24] to provide a brief overview of JMX.

Figure 4.5: JMX architecture [24]

The JMX architecture is divided into three levels:

1. Instrumentation level

The instrumentation level provides the specification for implementing *JMX manageable resources*. A JMX manageable resource may be an application, an implementation of a service, a device, a user and so on. Such a resource is developed in Java or at least offers a Java wrapper and has been instrumented so that it can be managed by JMX-compliant applications. The instrumentation of a given resource is provided by one or more *Managed Beans* (*MBeans*), which are either standard or dynamic. Standard MBeans are Java objects that conform to certain design patterns derived from the JavaBeans component model. Dynamic MBeans conform to a specific interface, which offers more flexibility at runtime. MBeans are considered further below. The instrumentation of a resource allows it to be manageable through the agent level, but MBeans do not require any knowledge of the JMX agent with which they cooperate. The instrumentation level also specifies a notification mechanism, which allows MBeans to generate and propagate notification events to components at the other levels.

2. Agent level

The agent level provides a specification for implementing agents. Management agents directly control the resources and make them available to remote management applications. Agents are usually located on the same machine as the resources that they control, although this is not a requirement. A JMX agent consists of an MBean server and a set of services for handling MBeans. Managers access an agent's MBeans and use the provided services through a protocol adaptor or connector, but agents do not require any knowledge of the remote management applications that use them.

3. Distributed services Level

The distributed service level provides the interfaces for implementing JMX managers. The level defines management interfaces and components that can operate on agents or hierarchies of agents. The components are able to:

- Provide an interface for management applications to interact transparently with an agent and its JMX manageable resources through a connector.
- Expose a management view of a JMX agent and its MBeans by mapping their semantic meaning into the constructs of a data rich protocol (for example HTTP or SNMP).
- Distribute management information from high-level management platforms to numerous JMX agents.
- Consolidate management information coming from numerous JMX agents into logical views that are relevant to the end user's business operations.
- Provide security. The combination of manager level with other agent and instrumentation levels provides a complete architecture for designing and developing complete management solutions. However, in this chapter we are concerned with the components that make up the instrumentation level. The key components of the instrumentation level are the MBean design patterns, the notification model and the MBean metadata classes, which are considered further below.

4. Instrumentation level components

An MBean is a Java object that implements a specific interface and conforms to certain design patterns. These requirements formalize the representation of the resource's *management interface* in the MBean. The management interface of an MBean is represented as:

- Valued attributes that may be accessed.
- Operations that may be invoked.
- Notifications that may be emitted.
- The constructors for the MBean's class.

The JMX architecture does not impose any restrictions on where compiled MBean classes are stored. They can be stored at any location specified in the classpath of an agent's JVM, or at a remote site if class loading is used.

JMX defines four types of MBeans: standard, dynamic, open and model MBeans, which each correspond to a different information need:

- Standard MBeans are the simplest to design and implement and their management interface is defined by their method names.
- Dynamic MBeans must implement a specific interface and they expose their management interface at runtime for greatest flexibility.
- Open MBeans are dynamic MBeans, which rely on basic data types for universal manageability.
- Model MBeans are also dynamic MBeans that are fully configurable and self-described at runtime. They provide a generic MBean class with default behaviour for dynamic instrumentation resources.

The standard MBean is the most common type of MBean. As an example, assume an application contains a `Logger` class that configures application debug messages by specifying a log filename and a verbosity level. The `Logger` class may be converted into a standard MBean by creating a management interface called `LoggerMBean`. Public 'set' and 'get' methods may be specified in the interface to expose the filename and verbosity attributes. Examples of these methods could be `setFileName()` and `getFileName()`.

The naming convention for 'set' and 'get' methods is based on the `setXXX` and `getXXX` method names used in JavaBeans.

The `Logger` class would then need to implement the `LoggerMBean` interface so that a JMX agent can use introspection to create metadata about the `Logger` MBean. The MBean is managed from the JMX agent by invoking attributes and other operational methods defined in the interface.

The JMX specification defines a generic notification model based on the Java event model. Notifications can be emitted by MBean instances as well as by the MBean server. The JMX specification describes the notification objects and the broadcaster and listener interfaces that notification senders and receivers must implement. A JMX implementation may provide services that allow distribution of the notification model thus allowing a management application to listen to MBean and MBean server events remotely.

The instrumentation specification defines the classes that are used to describe the management interface of an MBean. These classes are used to build a standard information structure for publishing the management interface of an MBean. One of the functions of the MBean server at the agent level is to provide the metadata of its MBeans. The metadata classes contain the structures to describe all of the components of an MBean's management interface: its attributes, operations (methods), notifications and constructors. For each of these, the metadata includes a name, a description and its particular characteristics. For example, one characteristic of an attribute is whether it is readable, writeable or both. A characteristic of an operation is the signature of its parameter and return types.

The different types of MBeans extend the metadata classes in order to provide additional information. Through this inheritance, the standard information will always be available and management applications which know how to access the subclasses can obtain the extra information.

The JMX reference architecture has been implemented by several organizations and community groups. For example, [44] provides an implementation of the reference architecture.

4.2.3 Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA)

Dynamic Assembly for System Adaptability, Dependability and Assurance (DASADA) is a group of DARPA funded related projects, concerned with the assembly and management of distributed component-based systems. Several DASADA projects are actively investigating the use of software gauges and probes to dynamically deduce component configurations and examine distributed systems as an assemblage of components, [10, 12, 13, 45]. These four projects are described below.

1. Software Surveyor – Dynamically Deducing Component-ware Configurations

Software Surveyor [12] is a profiling toolkit to dynamically deduce and render the runtime configuration and behaviour of evolving component-based software. Information is synthesized from multiple sources and combined and rendered in a variety of formats and made easily accessible via the Web.

Software Surveyor addresses three distinct issues:

- What is the application doing?
- What is it supposed to be doing?
- Is it doing what it is supposed to?

Software Surveyor provides probes to collect a variety of information and an infrastructure to disseminate the information. Synthesis tools are provided for merging information streams and make sense of the information as a whole. Results of this analysis are aggregated to identify "behavioural norms" to augment incomplete performance specifications. The probe infrastructure and behavioural norms can then be used to signal users when the system is operating anomalously. Software Surveyor requires limited prior knowledge of application connectivity and has the ability to dynamically deploy probes. This allows Software Surveyor to be used with dynamically reorganizing applications and those lacking complete specifications.

AppliProbes provide information about events at the application interface and/or internal to the application. EnviroProbes uses operating system utilities to gather and emit information on system status and resource use.

A Java ByteCode Instrumentor (JBCI), shown in Figure 4.6, automates the insertion of probes and probe stubs into Java ByteCode. JBCI modifies Java class files by inserting calls to selected probes using selected customizable instrumentation techniques. JBCI can be extended with new probes and instrumentation techniques. Probes implemented in languages other than Java can be called via Java's Native Interface (JNI). The next version of JBCI will support on-the-fly probe insertion into running programs.

Figure 4.6: Java bytecode instrumentor (JBCI) [12]

Events are distributed by the Siena Event Distribution Infrastructure, which was developed at the University of Colorado [46]. XML2Java is used to translate XML to first class Java objects with application-specific behaviours. This allows XML-encoded events to be converted into a form that can be manipulated.

The main analysis tools provided in Software Surveyor are:

- Coalescer, which is used to merge streams of separately collected event information to create an event timeline. Coalescer can also perform limited aggregation of events by time interval.

- StackTracer, which converts streams of application events into a trace of program execution and emits an XML representation.
- EventMonitor, which categorizes events by type and produces summaries with expandable detail.
- Historian, which archives execution traces and computes statistics of behaviour.

The main visualization tools provided in Software Surveyor are:

- Mapper provides a timeline-oriented visualization of application behaviour.
- XML-capable Browser, which is used to view StackTracer and EventMonitor results.

2. Framework for Interoperable Reconfiguration Measures (FIRM) - Definition, Deployment, and Use of Gauges to Manage Reconfigurable Component-Based Systems

FIRM [10] is based on the definition of a set of novel gauges to assess a wide range of critical system properties, and a scalable infrastructure to manage both the deployment and use of gauges throughout an enterprise. FIRM primarily addresses the Continual Coordination thrust of the DASADA program by ensuring that reconfiguration-related interoperability problems are detected and mitigated at multiple points in the life cycle of a system. Amongst other aims, FIRM aims to address the often-heard cry of "DLL Hell!", which is raised when an installation of a new system modifies the existing installed base without sufficient verification of consistency of the new installation with the old.

The FIRM developers point out that configuration and reconfiguration can provide solutions to interoperability problems. Correct, enforced configuration can often avoid many interoperability failures. Further, the ability to reconfigure a system, even after deployment and during operation, allows the replacement of inappropriate or faulty components. Reconfiguration also provides the opportunity to insert gauges, wrappers,

mediators, and other interoperability repair and monitoring components into systems to alleviate interoperability mismatches or to detect incipient interoperability problems.

FIRM provides a set of novel gauges capable of evaluating system configurations with respect to important interoperability properties. These gauges include those to:

- Measure the consistency and inconsistency of configurations,
- Measure the actual configurations adopted by systems,
- Measure properties across all possible configurations of a system,
- Measure redundancy and reuse properties of systems, and,
- Predict the costs of moving from one configuration to another.

Gauge-based evaluations can be performed statically on the configuration specifications and on the deployed configurations. The evaluations can also be performed dynamically (or, in FIRM's terminology *activated*) on executing systems.

FIRM provides an infrastructure to effectively deploy and use gauges, whether of its own design or of those of others. In particular, FIRM provides the means to:

- Deploy, activate, and replace components.
- Apply gauges for coordination.
- Insert gauges into activated systems, and,
- Capture, fuse, and disseminate the outputs of gauges.

The existing research projects of Software Dock [47], Ménage [48] and Siena [46] form the technical underpinnings for FIRM. Software Dock is an agent-based, distributed infrastructure for describing, deploying, and activating components. Ménage is a representation of configurable architectures, extending traditional architecture description languages to address versioning, variability, and optionality in systems. Siena is a scalable event notification service used to capture, fuse, and disseminate information in a wide-area network. Experience gained from these projects has been used to leverage the more comprehensive FIRM project.

3. En-gauging Architectures

En-gauging is an architecture developed by Teknowledge [13], which assists the design and deployment of gauges on real distributed systems running on commercial platforms to monitor their architecture and measure their performance. Through *En-gauging*, dynamic system information may be collected in a repository, made available to a wide variety of subscribers both automated and human. The information may then be used to validate performance, resource requirements, and other selected service qualities and to augment the systems' robustness and responsiveness.

According to Teknowledge: "Modern systems benefit from two adaptive technologies: (1) the ability to compose systems from reusable modules developed and compiled separately, and (2) the ability to distribute computing processes onto autonomous computing nodes. Although these technologies enable the potential to adapt performance to widely varying contexts, much of the information important for such performance adaptation is still "compiled out" of modern systems.

Fortunately, determining when and how to adapt a running system to varying configurations and performance demands (QoS demands) can be separated from system functionality. The approach used in En-gauging to obtain such information involves modelling a system's nominal behaviour and comparing it to its actual behaviour for the system's current configuration. Whenever the system deviates from the model, either the system must be reconfigured to achieve its QoS demands or the resources reapportioned to balance those demands. Modelling the system's nominal behaviour enables these validations and adaptations to be separated from the system's functionality and to be supported by an external infrastructure.

The En-gauging architecture builds on Teknowledge's experience with the Acme architecture description language [49] and its Instrumented Connector technology (both developed under DARPA's EDCS Program). Acme and the Instrumented Connector technology are combined to monitor the actual run-time architecture of a system, to reify it into an architecture model repository, and to publish event notifications to "subscribers" interested in such changes to the architecture. Such subscribers comprise:

- Analyzers to determine whether dynamic system constraints are satisfied.
- Simulators to establish the system's nominal behaviour benchmark.

- Trackers to respond to differences between nominal and actual, and
- GUI animators, potentially evoking a human response to redirect system resources.

En-gauging integrates DARPA's Quorum QoS Condition Service (QCS) [50] with the Instrumented Connector technology. This provides the infrastructure that enables application designers to design and deploy the gauges needed to measure and validate the running system's performance. En-gauging also provides Composability Framework Services, which allow application engineers to decide how and when to use performance and configuration information for adaptation to meet the QoS demands.

4. Architecture-Based Languages and Environments (ABLE) – Using Gauges for Architecture-based Monitoring and Adaptation

The ABLE project [45] is concerned with the development of an "engineering" basis for software architecture. Part of this research has led to the developments of techniques for describing and exploiting architectural *styles*. The ABLE project has developed several architectural description languages: Acme, Wright and Armani [51]. From the point of view of instrumentation and the DASADA initiative, ABLE has also considered the use of gauges for architecture-based monitoring and adaptation.

Gauges and also probes are used as one of the main components of an *externalized* runtime adaptability mechanism. The gauges and probes are used to collect low-level performance information, which is then interpreted and used as the basis for automated adaptation. The main foundation for monitoring and subsequent adaptation is the architectural model. Such models are defined in terms of components and their communication paths or connectors. ABLE makes use of gauges and probes to provide a monitoring infrastructure, which in turn provides and abstraction from system level to observations in an architectural context.

The monitoring infrastructure consists of three levels: at the lowest level is a set of probes, which are deployed within a target system or physical environment. The probes may then report observations of the actual system via a *probe bus*. At the second level a set of gauges consume and interpret lower-level probe measurements in terms of higher-level model properties. Similar to probes, gauges also disseminate information via a

gauge reporting bus. The top-level entities are gauge consumers, which consume the information provided by gauges.

The information gathered by the monitoring infrastructure may be used to update and abstraction/model to make system repair decisions, to display warnings and alerts to system users, or to show the current status of the running system.

The designers of ABLE describe the key features of this infrastructure:

- Gauges are decoupled from the implemented system by virtue of the probe layer. This decoupling allows gauges to be run in a distributed fashion so that they do not affect the performance of the system being gauged.
- Gauges can be mixed and matched, supporting interoperability between gauges that evaluate quite different properties, where the gauges may be developed by different organizations.
- Gauges are insulated from lower-level transport mechanisms, enabling gauges to be deployed over both RPC-based channels and publish-subscribe mechanisms.
- Gauges can be incorporated into architectural descriptions, enabling automatic generation and execution of gauges.

4.2.4 Instrumenting Jini Applications

The instrumentation framework described in this thesis is implemented using Sun Microsystem's Jini middleware technology. With this in mind, this section reviews several projects, which has focused specifically on furthering an understanding of Jini-based applications.

1. Rio Project

Rio is a Jini community project [52], which has made a significant contribution through an architecture that simplifies the development of Jini federations. Rio does so by providing concepts and capabilities that extend Jini into the areas of QoS, dynamic deployment and fault detection and recovery. Rio makes use of Jini Service Beans (JSBs), Monitor Services and Operational Strings. Rio also provides a Watchable

framework, which provides a mechanism to collect and analyze programmer-defined metrics in distributed systems.

The abstraction adopted in Rio is that, fundamentally, there are two different types of service on a network:

- Infrastructure services, which provide the basic building blocks, which domain components can use to perform application specific duties. Some examples are the Jini Lookup Service, JavaSpaces and Jini's Transaction Manager.
- Application or domain-level components are the actual components that will provide the federation of services that the application has to offer.

JSBs are targeted at the application or domain-level components. A JSB provides simplicity in the development of Jini services. JSBs are Java objects that provide an easy to use programming model while maintaining access to low-level APIs and classes.

An Operational String is a construct that represents an aggregated collection of application and/or infrastructure software assets that when put together provide a specific service on the network. An Operational String provides the capability to view, monitor and determine the availability of an aggregated collection. In configuration an Operational String is articulated as an XML document. In execution the Operational String is viewed as a collection of service and infrastructure components.

Rio extends Jini's distributed event model to provide an easier to use model with increased semantics. The Rio model also allows events to be monitored and interpreted with greater ease. The extension is based on:

- Event descriptors, which provide a simple semantic for specifying and discovering event producers of a specific kind of event. An Event Descriptor is an attribute, which is part of the description of an event producer.
- Event Producers can be any Jini service. Formally, an Event Producer is a service that has a zero-to-many dependency between objects such that when its state changes all its dependents are notified. This semantic is also known as the observer-observable and/or publish subscribe pattern [53].

As we shall see later in the thesis (chapters 7 and 8) the instrumentation framework provides a higher-level abstraction of events. This abstraction is based on a new specific event object, DynamicEventObject, which repackages Jini's RemoteEvent class to provide supplementary information.

Rio provides a Watchable framework, which provides a mechanism to collect and analyze programmer-defined metrics, defined in local and distributed applications. The Rio Architecture Overview illustrates the Watchable framework with a class diagram, which is repeated below.

Figure 4.7: Watchable framework [52]

At the core of the Watchable framework is the WatchDataStore interface. A WatchDataSource stores a history of measured results and provides access to add, clear or fetch items from the historical record. The Watchable interface provides a means for locating remote WatchDataSources. Each historical record must implement the Calculable interface by providing methods to retrieve an identifier string and a double value. Nonetheless, this does not preclude forming more sophisticated Calculable records.

Again, as we shall see later in the thesis (chapters 7 and 8) the instrumentation framework provides an analyzer service, which bears some similarity to Rio's Watchable framework. The analyzer instrumentation service allows a particular attribute to be repeatedly

accessed within a Java thread. An analyzer is provided with a "compute" object, which is used to compute some application metric, based on the history of the attribute's value.

Rio also provides several further components and services:

- The Cybernode, which is an infrastructure component that provides fundamental life-cycle support for JSBs and makes it easy for JSB developers to deploy JSBs. The Cybernode utilizes polymorphism and aggregation techniques, securely loading JSBs over the network, instantiating them and providing core services needed to support JSBs and ensure their availability on the network. The Cybernode provides container-like functionality i.e. Cybernodes are where JSBs live. A Cybernode may contain more than one JSB. In execution, a Cybernode runs as a Java Virtual Machine. A compute-resource (hardware) may run on more than one or more Cybernodes. A Cybernode also provides a QoS attribute representing the capabilities of the compute resource on which it is executing.
- Monitor Service, which provides the capability to deploy and monitor Operational Strings. Monitoring an Operational String allows the Monitor to detect and recover from service failure on the network. Monitors provide an essential capability to detect the existence of running and/or available software assets and to enable recovery in the event of failure. Central to the monitor service is the concept of Quality of Service (QoS). The QoS concept is based on the notion that compute resources (hardware) have capabilities (CPU, disk, connectivity, bandwidth etc.) and software components have requirements (response time, throughput, hardware requirements etc. Monitor services are designed to check and monitor software requirements against the available compute resources.
- Watchsmith Service, which provides an implementation of a distributed Watch. The Watchsmith service allows multiple services to record Calculable records for a distributed logical unit of work such as an applications response time. Since applications developed using Jini are inherently distributed,

bounded units of work will typically span multiple Java Virtual Machines. The Watchsmith service provides a mechanism fro distributed recording.

- Logger Service, which provides distributed logging facilities through a generic interface based on the Java Logging API (JSR-047) [54]. The Logger is used by infrastructure services and JSBs. Log archives reflect the activities of the various reporting services. The content of a log message is up to the service developer. Services report log messages across multiple logging levels as defined in the `java.util.logging.Level` class.

2. Carp@

Carp@, pronounced CarpAt, [55] was developed to watch and visualize a network consisting of several Jini services with a view to managing the services at runtime. According to the developers of Carp@: *"In a dynamic ad-hoc networking environment, the concrete architecture evolves during runtime. Decisions like choosing an implementation for a component, deciding a communication structure are not done at design time but at runtime"*. Therefore, in the opinion of the Carp@ developers, there is a need to extract an architecture's description at runtime. This description can then be used as the basis to decide about the effect of changes.

Carp@ goes beyond showing simple Jini services like other browsers do and shows additional important information that is not available otherwise but is needed to understand the interaction in a Jini application, which include:

- Clients and Services – Carp@ shows both the services and clients that use services and how these components communicate through communication *channels*. This is necessary, because clients may misbehave, consuming excessive service resources.
- Messages – Carp@ enables developers to trace method calls, together with their arguments, as messages, that are sent between services and clients for each communication channel in an application.
- Provided and Required Interfaces – services can provide multiple interfaces and clients may require multiple services. Carp@ shows not only these

provided interfaces, but also the *required* interfaces as *ports* for each component.

- Locations – service/client location information is not made available in a general Jini application. If services/clients should misbehave, it is necessary to determine their locations so that performance bottlenecks can be detected and isolated.

All this information is gathered by Carp@ at runtime mainly through the use of *reflection*, which is described further in section 4.2.5 and chapter 8. The logical architecture of the Carp@ system is based on three layers:

- Mobility layer, which is the bottom-most layer contains services that can be used to start application services from remote locations or to move a running service from one location to another.
- Management layer, which sits in the middle of the Carp@ system is used to gather and store an application's runtime information. This layer provides several services including: a Report Service and a Meta-model Service. The Report Service gathers basic information by querying special meta-level objects referred to as Carp@ Beans. Carp@ Beans may be queried to obtain a variety of information ranging from the simple, such service names and attributes up to the more complex, such as exchanged messages, communication channels or interface ports. All this information is stored in the Meta-model Service that contains the Carp@ internal model of the application being observed. This model may be simply observed to provide a view of the application or used to actually manage the application through Carp@'s control console.
- Application level, which is the top-most level and consists of service providing components and clients that use services. The application level also contains the Carp@ console, which are used to view and even manage the applications structure and behaviour.

The basic technique of Carp@ is to find out as much as possible about an application by reflection and other application describing resources. It is the belief of Carp@'s developers that: an application must be fully understood before it can be changed at runtime. An administrator may then manipulate the application through an internal model retrieved by introspection. These changes, applied at the meta-level may then be reflected in the applications runtime behaviour.

Carp@'s abstraction or internal model is based on: locations, services, channels and ports. These properties are maintained within the meta-level to provide an applications internal model. The properties are acquired by Carp@ Beans, which communicate with each other and notify each other of changes in the applications structure/behaviour. Carp@ Beans are capable of obtaining structural information (component-connector configurations), behavioural information (message communications) and resource information (memory usage). Such information is represented at the meta-level, where it may be viewed through a Carp@ console and use to effect changes, which are reflected back on the application itself.

3. Dependency Management in Jini Federations

An important pre-requisite to furthering an understanding of a system is knowledge of how components rely on one another, or more particularly, how components depend on the services provided by other components. Such dependencies have already been introduced in chapter 2, which described a dependency as a directed relationship between a dependent component and one or more independent components. chapter 2 also described how dependencies may be represented as a digraph, which provides an instantaneous snapshot of how components depend on one another.

Dependencies feature in a number of different computing fields including databases, network management and software development and compilation. Dependencies have also been considered in the context of distributed systems by several authors, including Hasslemeyer and Voβ [9] and Keller [56, 57]. [9] describes a generic approach to instrumentation that effectively instruments a Jini lookup service using Java's dynamic proxy class to trace component interactions in a Jini federation. Later in [7] Hasselmeyer

extends on this earlier work by considering the management of service dependencies in *service-centric* applications.

Hasselmeyer provides a thorough treatment of dependencies and their management and several of the ideas presented are extended in this thesis to provide dependency probe instrumentation services. The design, implementation and application of deoendency probes are considered later in chapters 7, 8 and 9. Hasselmeyer describes dependency management as a special form of relationship management as dependencies are a specific type of relationship.

4.2.5 Reflective Middleware

Chapter 3 described what is meant by the terms middleware and went on to describe Java RMI as a relatively simple middleware technology. Reflective middleware is effectively middleware with reflective capabilities. [58] clarifies exactly what is meant by reflective middleware by defining both the terms 'middleware' and 'reflection'. The definition given for reflection is that of *"a system that provides a representation of its own behaviour, which is amendable to inspection and adaptation, and is causally connected to the behaviour it describes"*. The term causally connected means that *"changes made to the self-representation are immediately mirrored in the underlying system's actual state and behaviour and vice-versa"*.

A reflective middleware system could be developed from scratch although a popular approach is to take an existing middleware and provide it with the reflective capabilities. CORBA is often used as the base middleware from where to start, probably due to the mature suite of standards that CORBA now has to offer, with respect to high performance and real-time system requirements. Another benefit of CORBA is its provision of an *interceptor* facility, which offers a limited form of behavioural reflection [43]. Interceptors enable a third-party to extend a CORBA implementation with additional functionality in an ORB independent manner. Interceptors have been investigated as a means to providing QoS management in CORBA systems [43] and more generally as a means to providing transparent instrumentation for CORBA systems [8, 59, 60]. CORBA also provides a Portable Object Adaptor (POA) facility that allows CORBA server applications to be portable across heterogeneous ORBs.

The case for reflective middleware stems from the inability of existing 'standard' middleware (Java RMI, CORBA) to adapt to changes in the operational environment. While standard middleware can meet the need of traditional client-server applications it does not fair so well in applications with highly dynamic operational environments such as distributed multimedia, real-time systems and mobile and ubiquitous computing. An exhaustive review of reflective middleware exceeds the scope of the thesis, particularly in terms of causal connectivity and adaptation capabilities. However, it is relevant to provide an insight into reflective middleware approaches and more particularly to describe how instrumentation tasks such as monitoring and inspection feature in reflective middleware. The following paragraphs do just that by briefly reviewing the approaches to reflective middleware of several key researchers with a view to examining their approach to monitoring/inspection.

[8] considers the use reflective technology for monitoring distributed component interactions based on three specific technologies:

- CORBA interceptors,
- Reflection on the thread model through Java,
- CORBA POA.

CORBA interceptors are used to provide low-level access to CORBA request/reply at both the message level and process level.

[61] describes 'The Lancaster Experience', which surveys three generations of reflective middleware research carried out at Lancaster University. The first generation used the Python language to prototype reflective middleware platforms and for the definition of four orthogonal reference meta-models: interface, architecture, interception and resources. The second generation involved the design and implementation of an experimental reflective CORBA platform (OpenORB). The reflective CORBA platform provided all the characteristics of commercial ORB implementations, but supported openness and adaptation in terms of its internal structure. In particular, the platform allowed individual interaction types (streaming, messaging, transactions etc.) to be specialized for different classes of application. The third generation moves away from the

dependence on a commercial middleware (e.g. CORBA). In this approach reflection is used to discover the style of middleware required in a given context and then automatically configure the middleware framework.

[61] then goes on to describe the third generation Lancaster approach in more detail. This approach is based on the key concepts of components, component frameworks and reflection (OpenCOM). This component-based approach is attractive because (as mentioned in chapter 3) it allows components to be plugged-in to a component framework, which then in turn becomes a more comprehensive component itself. It also provides a means to overcome the monolithic characteristics associated with commercial middleware. The component-based approach is described further in [62] for use with grid computing and in [63] for use with mobile clients. From an instrumentation point of view OpenCOM defines several meta-models and one of these is an interception meta-model, which allows interceptors to interpose between components and their interfaces [64]. The interceptor code may then be used for the dynamic interception of method calls made on interfaces in order to monitor dynamic runtime behaviour.

[65] describes an adaptive and reflective middleware system (ARMS) for use with distributed real-time and embedded applications. Typically ARMS can be used to configure QoS in CORBA Component Model (CCM) applications. The ARMS research is based on TAO [66], which is an open-source, CORBA-compliant ORB designed to support distributed real-time and embedded applications with stringent QoS requirements. The sections below provide an overview of the ARMS research drawn from the extensive material provided in [65].

[65] distinguish between adaptive middleware and reflective middleware as follows: *"Adaptive middleware is software whose functional and/or QoS-related properties can be modified either:*

- *Statically – for example to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies; or*
- *Dynamically – for example in response to changes in environmental conditions or requirements, such as changing component interconnection*

topologies; component failure or degradation; changing power-levels; changing CPU demands; changing network bandwidth and latencies; and changing priority, security, and dependability needs.

Reflective middleware goes a step further to permit automated examination of the capabilities it offers, and then permits automated adjustment to optimize those capabilities" [67, 68].

[65] describes the key research challenges that CCM developers must address to support QoS-enabled distributed real-time and embedded applications. One challenge is that of dealing with an ORB's location transparency features so that they respect an application's broader QoS requirements and not apply group optimizations blindly. A second challenge is that of changing component behaviour and resource usage adaptively without having to modify or shut down an application obtrusively (i.e. a "live upgrade" capability).

ARMS combines adaptation and reflective concepts to tackle these problems. ARMS defines a series of Meta-Object Protocols (MOP) [69] at various middleware levels, ranging from the ORB core up to CCM services. Amongst other things MOPs are used to identify the behaviours that are to be isolated and examined, such as the transport mechanisms an ORB supports. Consideration of these MOPS exceeds the scope of this thesis. Furthermore, ARMS main concern is QoS management for distributed real-time and embedded applications and the thesis does not go so far as to consider real-time and embedded systems.

4.2.6 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [70] provides a design paradigm that achieves a high degree of the *separation of concerns* in software development. AOP extends the traditional object-oriented programming model to improve code reuse across different object hierarchies. AOP is based on the concept of an *aspect*, which represents common behaviour that is typically scattered throughout an object oriented application. Typically aspects may occur across methods, classes, object hierarchies, or even an entire object model.

As an example, consider the need to generate log messages for an application (i.e. logging), which involves sprinkling informative messages throughout the application's code. However, logging is irrelevant to the actual application in that it has nothing to do with the business logic. Logging is regarded as an *orthogonal* behaviour that requires duplicated code in traditional object-oriented systems. In AOP terms such behaviour represents a *crosscutting concern*, as it represents behaviour that "cuts" across multiple points in the object model, yet the behaviour is likely to be distinctly different. As a development methodology, AOP recommends that crosscutting concerns are abstracted and encapsulated into aspects.

In AOP terminology the *advice* is the additional code that is applied to an existing application. In the logging example, this is the logging code that is to be applied whenever the thread of execution enters or exits a method. *Pointcut* is the term given to the point of execution in the application at which a crosscutting concern needs to be applied. In the logging example a pointcut is reached when the thread enters a method, and another pointcut is reached when the thread exits the method. An aspect may also be regarded as the combination of the pointcut and the advice. In the logging example, a logging aspect is added to the application by defining a pointcut and giving the correct advice.

Weaving is the process of taking the aspects and the regular object-oriented application and "weaving" them into one single application. The alternative approaches to weaving are source-code weaving and byte-code weaving. Source-code weaving takes developed source code and outputs a modified source code that invokes the aspects. Bytecode weaving takes the compiled application classes bytecode and outputs a modified bytecode of the *woven* application. Source-code weaving was used in early AOP languages, but bytecode weaving has now been widely adopted, particularly in Java-based AOP languages, with the advent of the Java 5 JVM Tool Interface (JVMTI) [71]. Bytecode weaving is now used by several AOP-based, enterprise-grade products in the area of application management and monitoring, such as JBoss Application Server [72], which is mentioned below.

Crosscutting concerns are common in enterprise middleware and this has fuelled the interest in AOP in conjunction with middleware. To explain why this is so we may consider an application based on J2EE container managed services. After deployment, each J2EE component (e.g., a EJB or a servlet) automatically gets services, such as logging, security and transaction support from the container. These services are orthogonal to the core business logic. Application developers could reuse these services without writing any additional code. In short, the J2EE services have the basic characteristics of aspects as outlined above. However, compared with a true AOP solution, the J2EE services model has a number of limitations, which stem from the constraints that J2EE containers impose – in short J2EE dictates how an application should be developed.

Further evidence for the uptake of AOP for middleware instrumentation and monitoring is had from the various research frameworks and studies that apply AOP in order to modularize the instrumentation and monitoring process. Several such research efforts are considered below.

[73] considers the notion of "Aspectizing Middleware Platforms" by proposing the need for new architectural methodologies, such as AOP, that could be applied to current middleware to overcome their increasing complexity. [73] goes on to describe a case study that uses AspectJ [74] in conjunction with CORBA. The study illustrates how certain CORBA features such as fault tolerance and interceptor support can be modularized using AspectJ. Overall, the work demonstrates that, by applying AOP techniques, pervasive characteristics may be factored in and out of middleware systems thus making the architecture more modularized and customisable.

[75] describes findings from using AOP for the modularization of Jini services in pervasive systems. [75] compares a custom non-AOP instrumentation layer against AOP instrumentation. The non-AOP instrumentation is considered first at the application layer and second at the framework/middleware layer. The latter reflects the approach used in the thesis, although on a far simpler scale. The results show that the AOP approach improves the modularization of instrumentation, particularly in pervasive environments with many objects scattered across the system.

[76, 77] describes the SONAR framework based on the combination of *dynamic AOP*, JMX and XML. [76] argues that static approaches to instrumentation/monitoring are no longer suitable for today's complex dynamic distributed systems and goes on to describe how the SONAR framework provides capabilities to deal with complexity and dynamism. In SONAR dynamic aspects are used to provide runtime instrumentation that supports a crosscutting structure. XML is used as a language/framework *agnostic* language to fit with multiple AOP frameworks and JMX provides standard-compliant visualization/management. SONAR uses AspectWerkz [78] to provide dynamic AOP to structure system wide crosscutting concerns that may be added/removed at runtime.

The success of AOP is reflected through its incorporation in enterprise-grade products such as JBoss Application Server. JBoss AOP is a 100% pure Java AOP framework that is tightly integrated with JBoss Application Server. JBoss AOP helps solve the constraints that J2EE imposes through its support for "Plain Old Java Objects" (POJOs) as opposed to pre-defined "components". JBoss AOP allows EJB-style services to be applied to POJOs without the complex EJB infrastructure code and deployment descriptors. New aspects can be developed and deployed within the application server, where they become available for all applications.

4.3 Contribution of the Thesis

The contribution provided through this thesis makes use of several ideas described in the previous sections, particularly the use of dynamic proxies and monitor services. Bearing in mind the financial and commercial support behind several of the above developments, the thesis cannot deliver a system of the same compendium of functionality as JMX or the DASADA initiative. What the thesis does provide is a through treatment of distributed, dynamic instrumentation in isolation from any management framework. This treatment is provided from theoretical, practical and programmatic viewpoints.

Overall the main contribution of the thesis is the conception of a *dynamic software instrumentation framework*. This framework consists of a series of related models, which include: a requirements model, a classification model, formal and semi-formal analysis models and a programming model. The framework specifies an architecture, which regards instrumentation as *services*, which are intended to complement core middleware

services. A proof of concept implementation of the architecture has been prototyped using Jini (a Java-based middleware technology) to provide an API for use in distributed Java applications. A series of case-studies are used to evaluate the architecture and assess the effectiveness and performance overhead of instrumentation services.

In the chapters to follow, the thesis provides a reference framework, which can be used by system architects, application developers and even middleware technology providers as a basis for the development of subsequent instrumentation efforts.

4.4 Chapter Summary

The chapter has reviewed the practices in software instrumentation from its early inception through to the current commercial systems, such as JMX and the various state of the art research efforts. The review has deliberately considered software instrumentation in relation to the different computing platforms and programming technologies of the day.

The review started with uni-processor architectures and early languages of ALGOL and FORTRAN and then moved onto parallel architectures. The bulk of the chapter considered distributed systems in conjunction with object-oriented middleware and in particular the "state of the art" developments in instrumentation for distributed systems.

This chapter concludes the first part of the thesis, which has been concerned with "setting the scene" and providing the necessary background information. The next part of the thesis (chapters 5-8) provides the main contribution in terms of requirements analysis, formal modelling, design and implementation of the instrumentation services.

Chapter 5

Requirements of Instrumentation Services

This chapter considers the requirements that an instrumentation architecture must fulfil in order to measure and monitor the behaviour of a distributed system. The chapter begins by considering the different types of requirements that an instrumentation architecture must fulfil. The chapter then goes on to consider the parameters that must be measured and monitored and the different types of parameter. This is followed by an outline of the *functional* requirements in terms of what instrumentation services must measure and monitor. The *operational* requirements are then outlined, which govern how instrumentation services are incorporated, co-exist and interact with the application services. The chapter ends with a classification of the different types of instrumentation service in the form of an instrumentation hierarchy, which is intended to serve as the basis for the development of the instrumentation architecture.

5.1 Functional and Operational Requirements

In our consideration of requirements we distinguish between *functional* and *operational* requirements. The functional requirements deal with what the instrumentation services must measure and monitor and govern the different types of instrument that the architecture must provide. The operational requirements deal with the incorporation of these instruments within a distributed system and their application and attachment to the application services that they must measure and monitor. The reasons for this distinction are:

- To simplify the instrumentation architecture – by separating functional and operational requirements we are employing a *separation of concerns* that reduces the coupling between what instrumentation should measure and monitor and what facilities are needed to allow this measurement and monitoring to take place.

- To provide generic programming models – that developers may use as reference models that can be tailored to meet their own specific needs.
- To provide openness – in the sense that it would be possible to re-implement either the functional or operational aspects.
- To complement core middleware services – middleware services themselves adopt a similar separation of concerns. For example, CORBA and Jini both support the RMI protocol, based on the same semantics. However, the techniques used to incorporate CORBA, or Jini middleware utilities to a Java-based distributed system are likely to be different. Furthermore, as mentioned in chapter 1, one of the prime aims of the thesis is to promote the case for instrumentation as a core middleware service. To strengthen our case we feel that it is sensible to adopt a similar separation of concerns as the current middleware technologies.

To further justify our decision for this distinction, we consider a simple case from the field of instrumentation used in conventional engineering:

> *The altimeter is an instrument developed by the French physicist Lois Paul Cailletet [79] that measures vertical distance with respect to a reference level. Typically an altimeter is used to measure the altitude of the land surface or any air-bound object such as an airplane, hot-air balloon or satellite. The physics and engineering technologies used to develop the measurement functionality for an altimeter remain the same irrespective of the type of air-bound object for which the altimeter is intended. It is true that there may be certain design variations depending on the height above the reference level, but, adopting a simplistic view, we may regard these as variations based on scale (e.g. 1 mile vs. 100 miles). When we come to consider the incorporation of an altimeter into an air-bound vehicle then there are will be some differences depending on whether we are dealing with an airplane, hot-air balloon or satellite. However, there is also much common ground: the altimeter is likely to need a power supply; the altimeter will need to be fixed to the vehicle in some way etc. To summarize, we may*

treat the physics of an altimeter separately to its actual incorporation within an air-bound vehicle.

We consider the functional and operational requirements as the essential requirements, which must be met by an instrumentation architecture that provides a collection of instrumentation services. Our decision to consider instrumentation as services is once again influenced by the service-oriented abstraction to middleware technologies – which we intend to complement with instrumentation capabilities. Also, as will be considered further in Section 5.2, we choose to adopt a service-oriented abstraction of the applications within a distributed system. This allows us to abstract the physical components of a distributed system as a *federation* of *logical* services that can be measured and monitored by an instrumentation architecture that provides instrumentation services.

In subsequent sections we present an informal outline of the main functional and operational requirements, but first we must consider the different parameters and measurement types that our instrumentation services must accommodate.

5.2 Parameters and Measurement Types

Before considering the requirements we need to identify the elements that constitute a distributed system and focus on the parameters and information contents that characterize these elements. Primarily, we are seeking to identify an abstraction a distributed system that reveals the elements that can inform us about the state and behaviour of any distributed system.

5.2.1 Elements to measure

Recalling the definition of a distributed system (chapter 2, [20]) as:

> *"a collection of autonomous hosts that are connected through a computer network with each host executing service providing components and operating a distributed middleware to enable components to coordinate their activities giving the impression of a single, integrated computing facility".*

From this definition the main elements are identified as:

- The collection of hosts
- The computer network
- The collection of service providing components
- The distributed middleware

As it stands, this list is too general to consider the requirements for instrumentation services, so it must be refined and modified to fall within the intentions and scope of the thesis. The first two refinements concern the collection of hosts and the computer network. Each host could be running a collection of Virtual Machines (VMs), which may need to be measured and monitored during their operation. As the computer network is far too general, it is refined to limit consideration to the Network Operating System (NOS), or more specifically the services and resources provided by the NOS.

Recalling the software layer model of chapter 2 (Figure 2.1), we prefer to consider the uppermost levels as a federation of services, made up of application services and core middleware services. This implies that we intend to adopt the service-oriented programming model considered previously in chapter 3. The advantage of adopting this model is that it allows us to use a higher-level abstraction in that we are dealing with *logical* concepts, where we regard a service as a logical concept such as a chat-room or printer service.

The logical services are of course provided by physical components and in general, the relationship between a service and a component is not one-to-one in that several components may be required to provide a single logical service. A service may also be uniquely identified by its *service proxy* (chapter 3), which provides an interface to the application service and, from the point of view of instrumentation, a suitable point of attachment or application of instrumentation services. Finally, as we have chosen to develop our instrumentation as a collection of services it proves simpler to view the collection of application services to be instrumented as a federation of services. So we modify *"the collection of service providing components"* to *"the federation of application services"* and *"the distributed middleware"* to *"the core middleware services"*.

So the modified list of elements that instrumentation services are required to measure and monitor reads as:

- The collection of hosts
- The Virtual Machines (VMs) associated with each host
- The Network Operating System's (NOS) resources and services
- The federation of application services
- The core middleware services

Having identified the elements to be measured and monitored, we must next consider the different types of parameter that characterize these elements and also distinguish between the different measurements types for these parameters. Previously, chapter 2 described how a distributed system is characterized by its *structure* and *behaviour* and considered the architectural and fundamental models that we may use to characterize structure and behaviour.

The architectural or system models described the division of responsibilities between components and the placement of the components on computers. The architectural models essentially serve as the basis for the distribution of responsibilities in a distributed system. The fundamental models of interaction, failure and security are based on fundamental properties that allow us to be more specific about the interactions, failures and security risks that particular systems may exhibit. From these architectural and fundamental models we may identify the main types of parameter that feature in distributed systems.

5.2.2 Parameter types

There are two main categories of parameter that feature in a distributed system, namely the static and dynamic categories. Dynamic parameters may also be categorized as synchronous or asynchronous parameters. There is also a third *derived* type of parameter, namely dependency parameters, which may straddle both static and dynamic parameters:

Static Parameters

Static parameters, such as names, types, modes, Ids are represented using simple static data types, such as integers, strings and enumeration types, which are supported by programming languages. In general, static parameters are represented using static data types (Integer, String and enumeration types) and not as objects, which have capabilities to engage in message passing and thereby exhibit dynamic behaviour.

Dynamic parameters

Dynamic parameters may be classified as either asynchronous or synchronous dynamic parameters:

- Asynchronous dynamic parameters: are independent of time and as such their time of arrival at a destination cannot be guaranteed. Distributed events are the main type of asynchronous dynamic parameter that must be monitored by the instrumentation services. Asynchronous parameters may also occur as errors or exceptions that occur when a computation fails. Often, such errors and exceptions give rise to an "avalanche" effect through which further errors and exceptions may occur.
- Synchronous dynamic parameters: are dependent on time and feature in RMI calls, distributed transaction processing, concurrent computations and multimedia transmissions. RMI calls are the main type of synchronous dynamic parameter that must be monitored by the instrumentation services. RMI calls are synchronous because, as mentioned previously, the caller is forced to wait for the receipt of a reply or an exception, indicating that the call failed. Instrumentation services need to be able to monitor synchronous RMI calls, both when the call succeeds and when the call fails, resulting in an exception.

Dependency parameters

Chapter 2 considered the service dependencies that exist between the components of a distributed system. These dependencies may be modelled as a directed graph (digraph) in which a directed arc or edge implies that a certain node, or component, uses the service(s) provided by another component(s). We also learned from chapter 2 how service

dependencies are in general dynamic when a component's state changes, giving rise to changes in its dependencies.

Dynamic parameter representation

Typically, events are the main asynchronous dynamic parameters and RMI calls are the main synchronous dynamic parameters and both may be represented as object data types. Certain dynamic parameters may also be re-packaged into specific instrumentation objects according to their context. For example a remote event parameter is re-packaged as an instrumentation event object, which provides additional information that specifies the source and sequence number of the event, which allows comparisons with sequence numbers of other events of the same type.

This re-packaging of dynamic parameters is a major requirement of instrumentation services because, in isolation, certain dynamic events may prove meaningless to any external management agent. This requires that instrumentation services must provide additional information to assist the managerial agent to make sense of a dynamic parameter. By careful design of the instrumentation services hierarchy, the "front-line" or direct instrumentation services, responsible for dealing with dynamic parameters, can delegate the acquisition of the additional information to simpler services.

Dependency parameter representation

Dependency parameters are said to be derived since they are function of the components that constitute a distributed system and also a function of the state transitions, which may occur in a distributed system. At any particular instant in time, we may derive service dependencies using static and dynamic parameters from *bindings* between components. For example, we could build a static graph of nodes and directed edges to reflect the dependencies at that particular instant. However, over a period of time, the graph may change, dynamically as a consequence of changes in bindings and hence service dependencies. As we shall see later in chapters 7 and 8, a binding occurs when one component (the dependent) has downloaded a copy of the proxy of some other component (the independent) with a view to invoking the methods of the independent component. We may represent this relationship as directed edge in a dependency digraph.

Instrumentation must be capable of first determining dependency relationships, which may be accessed through binding relationships. After determining such relationships, instrumentation must represent them accordingly, but it must also be capable of detecting when dependencies have changed so that the graph can be revised. This challenge raises a further contribution of the thesis towards *communicating* instrumentation. By arranging instrumentation services into groups, parameters such as dependencies may be derived by group communication. In other words, the maintenance of a dynamic dependency graph may rely on several instrumentation services communicating changes in state of their associated application components amongst each other. The formalisms governing instrumentation communication are considered in chapter 6, which is concerned with a formal model of the operational aspects of instrumentation.

5.3 Functional Requirements

The functional requirements outline exactly *what* instrumentation services must measure and monitor. The details of how the measurement and monitoring is achieved in considered in chapter 7, which is concerned with the development of an instrumentation architecture. As mentioned previously, it is the author's belief that the functional requirements may be separated from the operational requirements. This approach not only simplifies our consideration of an instrumentation architecture, but also allows us to develop generic models of instrumentation that may be used, extended and even revised by developers to suit their own particular needs. Furthermore, it is anticipated that the separation strategy used to develop our instrumentation architecture will facilitate the integration of the architecture with current middleware technologies.

To specify the functional requirements, we consider the five elements highlighted previously. Typical parameters for each of the five elements are considered through the tables listed below. It should be pointed out that the parameters in the tables are only a general set of parameters and there are likely to be additional parameters associated with specific system requirements. It should also be pointed out that not all of the parameters are recorded or monitored by the instrumentation services described in the thesis. For example, all the parameters relating to the hosts and network operating system (i.e. the platform) were not recorded due to their scope exceeding that of the thesis. The

paremeters also reflect constructs and utilities used in Jini middleware, which may not be directly applicable to other middleware technologies – notably Java RMI, CORBA and Web Services.

Description	Parameter
Host Name	String
Host Id	IP-Address
Host Type	{Desktop-PC, Web-Server, File-Server, DNS-Server, Palm, PDA, Mobile-Phone}
Host Network Mode	{Wired, Wireless}
Host System Clock Time	Date
Host Networking Pattern	{Unicast, Multicast}

Table 5.1: Host Parameters

Instrumentation services are required to measure/monitor the parameters listed in Table 5.1 relating to each host computer in a distributed system.

Description	Parameter
VM Parent Host Id	Host
VM Id	String
VM Type	{J2SE, J2ME, Other}
VM Heap Size	Long
VM Memory	Long

Table 5.2: Virtual Machine Parameters

Instrumentation services are required to measure/monitor the parameters listed in Table 5.2 relating to each VM that is associated with a host computer in a distributed system.

Description	Parameter
NOS Type	String
NOS Version	Integer
Active Ports	SET OF Port
Open Sockets	SET OF Socket
Network Services	SET OF NetworkService

Table 5.3: Network Operating System Parameters.

Instrumentation services are required to measure/monitor the parameters listed in Table 5.3 relating to the network operating system. The parameters provide basic information about the network operating system, including its type and version, as well as the more detailed parameters:

- Active Ports – identifies the current set of ports that can be used for transmissions. Each port object contains an attribute that identifies the host from which the port may be accessed.
- Open Sockets – identifies the current set of socket connections including their associated protocol (TCP, UDP).
- Network Services – identifies the current set of network services (DHCP, DNS, etc.). Each NetworkService object contains an attribute that identifies the host on which the service is located and a state attribute, which is the enumeration type {Started, Stopped} to indicate the state of the service.

Description	Parameter
Service Name	IP-Address
Service Id	String
Registry Service	Registry
Lease Service	Lease
Transaction Service	Transaction
Service Attributes	SET Of Object
Service Method Signatures	SET Of Method
Service Serialized Code Size	Long
Service Type	{Non-Activatable, Activatable}
Service Group	Activation object
Class Files	SET OF Class
Clients Dependent on Service	SET Of Object
Service Dependencies of Service	SET Of Object
Service State	{Registered, Unregistered, Active, Dormant}
RMI Calls	SET Of RMI Call
Events Received	SET Of Event
Events Sent	SET Of Event
Exceptions Thrown	SET Of Exception

Table 5.4: Application Service Parameters

Instrumentation services are required to measure/monitor the parameters listed in Table 5.4 relating to the federation of application services that may be active or dormant in a distributed system. The parameters provide basic information about each service, including its name and Id, as well as the more detailed parameters:

- Registry – the registry service with which the application service is registered.

- Lease Service – the lease on the application service.
- Transaction Service – a transaction created by the application service.
- Service Attributes – the attribute values that define the application service's state.
- Service Method Signatures – the method signatures available for the application service (both local and remote method signatures).
- Service Serialized Code Size – the size of the code segment used to hold the application service objects.
- Service Type – the type of application service – active or passive.
- Service Group – the group of which the application service is part of (for active services).
- Class files – the class files from which the application service's objects have been created.
- Clients Dependent on Service – the clients that depend on the application service.
- Service Dependencies of Service – the other application services that the service depends on.
- Service State – the state of the service in terms of its registration and whether active or dormant – for activatable services.
- RMI Calls – the RMI calls made on the application service
- Events Received – the events received by the application service.
- Events Received – the events sent by the application service.
- Exceptions Thrown – the exceptions thrown by the application service (remote and local).

Description	Parameter
Middleware Type	{CORBA, RMI, Jini}
Middleware Version	Integer
Registry Services	SET OF Registry
Discovery Management Services	SET OF DiscoveryManagement
Transaction Services	SET OF Transaction
Lease Services	SET OF Lease

Table 5.5: Core Middleware Parameters

Instrumentation services are required to measure/monitor the parameters listed in Table 5.5 relating to the middleware services registered in a distributed system. The parameters provide basic information about the current middleware technology, including its type and version, as well as the more detailed parameters:

- Registry Services – identifies the current set of registery services in a distributed system (e.g. Jini lookup services). Each registry service contains an attribute that identifies the host with which the service is registered and a state attribute, which is the enumeration type {Started, Stopped} to indicate the state of the service.

- Discovery Management Services – identifies the current set of registered Discovery Management services in a distributed system. Each DiscoveryManagement service contains an attribute that identifies the host with which the service is registered and a state attribute, which is the enumeration type {Started, Stopped} to indicate the state of the service.

- Transaction Services – identifies the current set of registered Transaction services in a distributed system. Each Transaction service contains an attribute that identifies the host with which the service is registered and a state attribute, which is the enumeration type {Started, Stopped} to indicate the state of the service.

- Lease Services – identifies the current set of registered Lease services in a distributed system. Each Lease service contains an attribute that identifies the host with which the service is registered and a state attribute, which is the enumeration type {Started, Stopped} to indicate the state of the service.

These services are likely to vary across different middleware technologies. The list shown in Table 5.5 are typical services that feature in Jini middleware technology.

5.4 Operational Requirements

The operational requirements outline the additional functionality required of instrumentation services to facilitate their incorporation, co-existence and interaction with the application services that they are measuring and monitoring. The operational requirements also outline the functionality required to provide interfaces to other distributed agents (such as GUIs) and standard network management protocols (such as SNMP and WBEM). As mentioned previously, by separating the operational requirements from the functional requirements we may concentrate attention on the requirements of instrumentation services that facilitate their application and attachment to application services. It is anticipated that consideration of the operational functionality in isolation will assist and strengthen the case for instrumentation as a core middleware service.

A major operational requirement affecting the viability of instrumentation is the ease with which instrumentation services can be dynamically attached/removed to/from application services and joined together to provide instrumentation groups through which instrumentation services may communicate with each other. As mentioned previously, the realistic success of distributed instrumentation relies on its ability to be applied *unobtrusively*, so as to not hinder an application or introduce additional computational overheads. This design of unobtrusive instrumentation is one of the major contributions of the thesis. Its achievement is considered further in chapters 6, 7 and 8 and again, it relies on the design of the instrumentation hierarchy and the programming model used for instrumentation services.

The main operational requirements of the instrumentation services are considered below:

1. Registration/deregistration

Instrumentation services must be capable of registering with a registry (e.g. a Jini lookup service) that is found using some form of discovery protocol. The lookup service serves as a trading service and any instrumentation management software should be able to

examine a series of lookup services to determine the current state of available instrumentation (i.e. which instruments are registered and which are not).

2. Attachment/removal

Instrumentation services must provide functionality to facilitate unobtrusive attachment and removal to/from application services. This requirement, which is explored in detail in chapters 6 and 8, can be addressed through the design of the programming model, based on the use of interfaces and *dynamic proxy* classes[3] [80].

3. Inter-instrumentation communications

Instrumentation services must provide functionality to facilitate inter-instrumentation service communications. Such communications may be achieved through events or method invocations, depending on the situation and more importantly the information "richness" of the communication (events are simpler, but limited in terms of information content). These communications require that instrumentation services must implement read/write, invoke and notify methods. Instrumentation services must also provide facilities that allow them to be organized into groups or domains through which information may be shared and tasks delegated accordingly. This capability for organization requires that instrumentation services must implement join and unjoin methods, which allow them to join or leave instrumentation domains.

4. Interface to external management agents

The instrumentation architecture is intended to provide a set of services and utilities that facilitate the development and runtime management of distributed systems. As such they are both used in conjunction with other *external management agents*. The third party agents may typically include: federated management agents [81], management beans [52], application logging agents [82], control agents, GUIs and other visualization agents. The ease with which instrumentation may be integrated and used co-operatively with such agents is reliant on the interfaces that instrumentation services provide, or more generally the interfaces supported by the instrumentation architecture. This requires that instrumentation services provide a series of open, reliable, yet secure interfaces to facilitate *seamless* integration (although security aspects exceed the scope of the thesis).

[3] The dynamic proxy class will be considered further in chapter 8

Again, these requirements can be satisfied by adopting similar approaches to those used in current middleware technologies to provide open interfaces. For example, the Jini middleware technology has been extended by way of a ServiceUI facility that allows Jini applications to support any GUI developed using Java's Swing component model. Instrumentation services are also required to facilitate the integration of *legacy* systems and other systems that may not necessarily support a full programmers API. This may be achieved through the use of *surrogates* and *wrapper* classes that essentially allow instrumentation services to provide a basic API on behalf of a legacy system.

5. Interface to standard network protocols

Along similar lines to point 4 above, instrumentation services are also required to provide interfaces to standard network management protocols such as SNMP and WBEM. SNMP is based on a limited set of commands and responses and a message format that specifies an information content that can be used to provide information about large inter-networks. Instrumentation services are required to provide support for the command set and utilities to transmit and interpret SNMP messages, thereby allowing SNMP agents to utilize the facilities provided by the instrumentation architecture.

WBEM (Web Based Enterprise Management) is based on a Common Information Model (CIM) and defines CIM schemas and CIM operations that operate over HTTP and are used to write CIM XML documents. It would be asking too much to expect instrumentation services to provide full support for WBEM as the CIM is a comprehensive specification. However, instrumentation services are required to support the use of CIM operations, which are described in XML. Therefore, instrumentation services should provide capabilities for parsing CIM XML-based operations and delivering such operations over HTTP to CIM clients and servers as required (although support for WBEM exceeds the scope of the thesis).

Many of the above operational requirements can be addressed by combining standard object-oriented programming techniques (such as interfaces, wrapper classes) together with object-oriented design patterns (such as dynamic proxy, façade, surrogate and adapter). The fourth and fifth of the above operational requirements, namely interface to

standard network protocols and interface to external management agents will be considered further in chapters 7 and 8.

It should be pointed out that the above discussion represents a general view of the additional interface support that instrumentation services should provide. The thesis only considers a small subset of this support to provide proof of concept. The main concern of this thesis is that of meeting the basic functional requirements discussed previously and the operational requirements discussed above in points 1-3. From these operational requirements we may identify a set of basic instrumentation service operations as:

- Register – registers an instrumentation service with an active lookup service.
- Unregister – unregisters an instrumentation service that was previously registered with an active lookup service.
- Attach – attaches a registered instrumentation service, via a dynamic proxy, to an application component, so that the instrumentation service may monitor and measure the component.
- Detach – detaches an instrumentation service that was previously attached to an application component leaving the instrumentation service registered.
- Join – joins an instrumentation service, via its dynamic proxy, to a group of instrumentation services that are already joined so that the group may communicate and share each others' services.
- Unjoin – unjoins or removes an instrumentation service that was previously joined to a group of other instruments.
- Read – allows an instrument that is joined to a group of other instruments to receive information from any other instrument in the group.
- Write – allows an instrument that is joined to a group of other instruments to send information to any other instrument in the group.
- Invoke – allows an instrument to intervene in the method invocations that a client component makes on an application server component.

- Notify – allows an instrument that is joined to a group of other instruments to notify the other instruments of changes in its state via a multicast Write operation.

These basic operations essentially cover the operational requirements described above in points 1-3. In order to ensure that the states, transitions and axioms governing these operations are fully understood the operations are to be formally specified in chapter 6. Several of the techniques used to address the requirements discussed in points 4 and 5 are considered through the programming models that are considered in chapter 8.

5.5 Classification of Instrumentation Services

Instrumentation services may be classified according to the roles they play and the functionality they provide. The different roles are identified as below.

1. Direct vs. indirect instrumentation services

Direct instrumentation services are those that are directly attached to an application component to measure and monitor certain parameters. As such, direct services must implement the programming model and the necessary interfaces to facilitate dynamic attachment/removal. In contrast, indirect instrumentation services are not directly attached to an application component, but they are still capable of providing details about the performance and behaviour of an application component. Indirect instrumentation services may be used to communicate with other instrumentation services and, in particular, for delegating instrumentation activities and responsibilities.

2. Static vs. dynamic instrumentation services

Static instrumentation services are used primarily to record and log information and as such they have no capabilities for dealing with dynamic behavioural parameters. In contrast, dynamic instrumentation services are used to respond to and also acknowledge dynamic parameters. Typically, the dynamic services are capable of intercepting remote events and remote method invocations (RMI calls) and repacking these objects to provide meaningful behavioural information. Dynamic dependencies are also accommodated within the dynamic instrumentation service category as a consequence of the dynamics inherent in dependency relationships.

3. Synchronous vs. asynchronous instrumentation services

Asynchronous instrumentation services are a subclass of dynamic instrumentation services that measure and monitor dynamic behaviour such as events and dependencies. Synchronous instrumentation services are also a subclass of dynamic instrumentation services that measure and monitor synchronous RMI calls (recall, RMI calls are generally synchronous since the client is held up waiting for a reply or exception).

Based on this informal classification of roles, a basic instrumentation service hierarchy may be specified as shown overleaf in Figure 5.1.

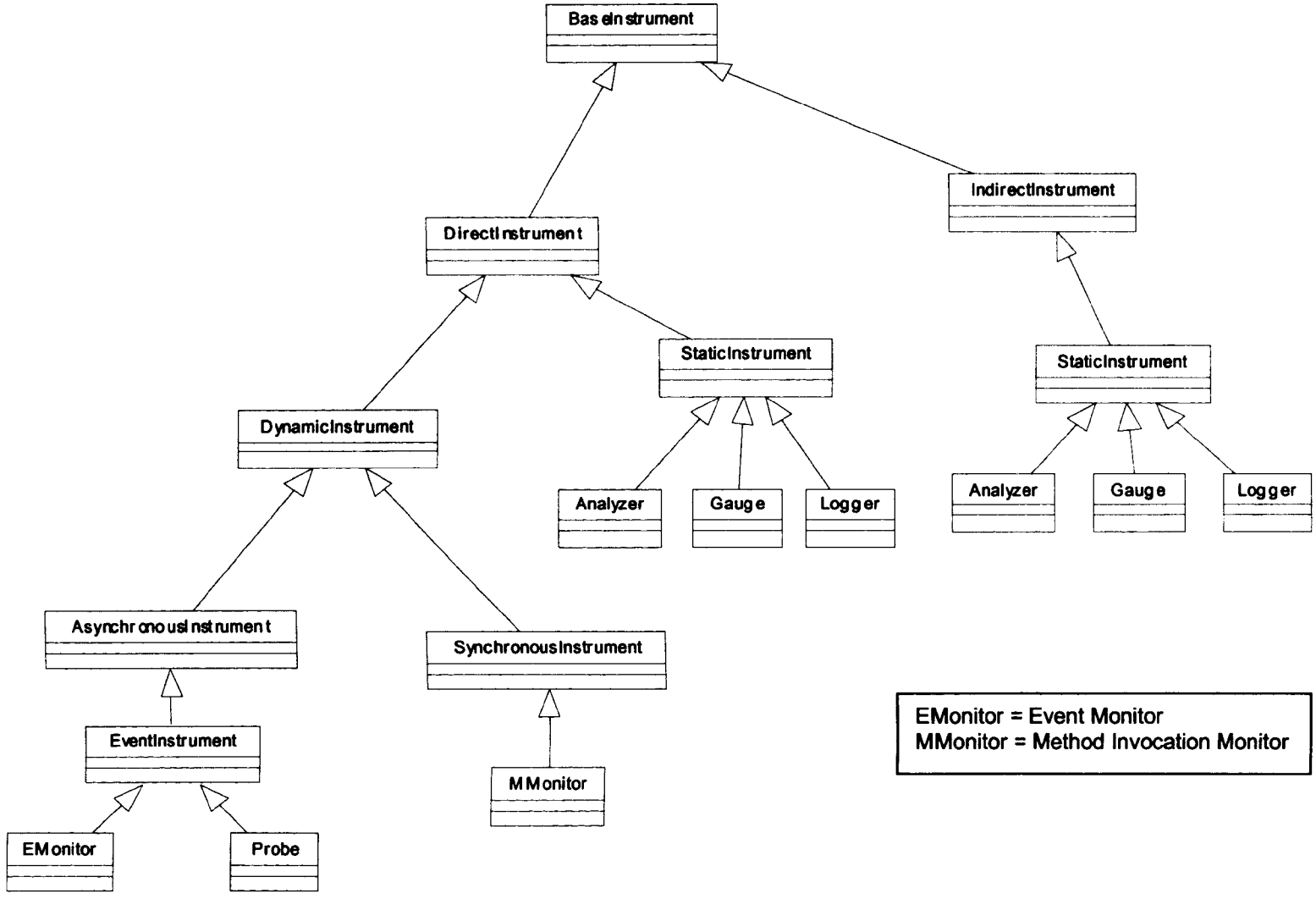


Figure 5.1: instrumentation hierarchy

At the root of the hierarchy is the base class BaseInstrument, which is inherited by all other instrumentation services. Then there is a distinction between direct and indirect instrumentation services. Either of these classes may be sub-classed to provide dynamic or static instrumentation services. Eventually, the hierarchy proceeds to the static leaf classes of *Logger*, *Analyzer*, and the dynamic leaf classes of *Gauge*, *Probe* and *Monitor*, which are the concrete instantiable instrumentation service classes. This basic hierarchy is to be developed further in chapters 7 and 8 to provide the infrastructure required for the instrumentation architecture's API. The activities of the concrete instantiable instrumentation services are also considered further in chapter 7, but we may conclude this chapter with a brief description of the roles of each of these instrumentation services.

1. Logger

A Logger is the simplest form of instrument, which simply records or logs some parameter of interest generally over a period of time. Loggers are the most general purpose type of instrumentation service in that they may be used to record parameters for application-level components or middleware services. Loggers record or log values to a data stream, which may be a simple file or even a stream to another object (e.g. using Java's `ObjectInputStream` or `PipedInputStream`). Loggers must be supplied with a reference to an object that specifies the parameter to be logged.

2. Analyzer

An Analyzer can be used to compute or derive certain metrics generally over a period of time relating to the application component to which it is attached. Analyzers are supplied with a reference to a separate user-defined object that performs the computation. So for example, an analyzer could be used to compute the mean-time between the component being accessed (i.e. the mean-time for which the component's service is not required). Essentially, the analyzer provides a facility through which users may perform their own specific analysis of an application. It does however, require that the user supplies an appropriate computational object that details the parameters to be analyzed as well performing the analysis computation.

3. Gauge

A Gauge is used to measure and display some value measured on an ordinal scale between lower and upper limits over a period of time. Typically, a gauge may be used to measure numeric parameters, such as the rate at which a component is accessed. Gauges must be supplied with a reference to an object that specifies the parameter to be gauged.

4. Probe

A Probe is responsible for deriving the service dependencies associated with a single particular application component. A probe will build a partial dependency graph for its associated component. The probe does not stop at direct or first-level dependencies, but will iterate beyond these dependencies using a *visitor* design pattern until there are no further dependencies. In other words, a probe will build a graph of all components that may affect the behaviour of the component with which the probe is associated. Probes are equipped with event handling capabilities through which they may receive notification of changes in dependencies. A single probe may only derive the dependencies for its associated application component. The dependency digraph for a complete application may therefore require the use of many probes communicating with each other.

5. Monitor

A Monitor may be a method invocation monitor (MMonitor) or an event-based monitor (EMonitor). A method invocation monitor monitors the method invocations that are made on a component. Method invocation monitors are capable of intervening the method invocations (RMI calls) made on a component, so that they may access the parameters used in the invocation and any return values or exceptions that result from the invocation. An event-based monitor monitors the events sent from a component, or received by a component. Event-based monitors are capable of re-packaging events as an instrumentation event objects to provide additional information that specifies the source and destination of the event and the scope of the event.

The author is unaware of any software instrumentation standard that provides definitions relating of the above instrumentation services. To this end the author has chosen his own naming scheme. The names have been chosen to reflect similarities between instrumentation units used in conventional engineering or the physical sciences. Several research efforts, concerned with software instrumentation, identify instruments that bear

the same names to those provided above. However, the instruments identified by others do not necessarily provide the same functionality to their namesakes as described above.

For example, [45] consider the use of gauges to dynamically deduce component configurations. These gauges prescribe more complicated functionality than the gauges described in the thesis and in fact prescribe functionality more closer to the probe instruments described in the thesis. [8] describes CORBA-based monitors that are capable of "peeking" into the implementation of CORBA objects at runtime by using CORBA's *interceptor* mechanism. These monitors do bear similarities to the monitors described in the thesis, differing only in the underlying mechanism that is used – the monitors described in the thesis use a mechanism based on a dynamic proxy. [14, 83] provide a comprehensive studies of software probes, which considers amongst other things probe insertion, execution and invocation. The probe service described above does fall in line with several aspects of the classification, but not all such aspects.

However, such naming discrepancies or mismatches in classification should not hinder the development of distributed software instrumentation, which after all is a relatively new and emerging field. Instrument names simply provide a means to refer to instruments and provided the functionality of the instrument is well-defined then the name plays a secondary role. It is the author's belief that this will continue to be the case until a distributed software instrumentation standard or working committee is established.

To conclude this chapter, the philosophy behind the naming of the instrumentation services described above is described below. As mentioned above, the names were essentially based on similarities with instrumentation used in conventional engineering or the physical sciences:

- Logger – a logger is common instrument that records or logs data, such as a data logger that may be used by meteorologists to log climatic data.
- Analyzers – are often used by geologists to determine the percentage of certain minerals, occurring in rock samples.
- Gauges – occur frequently in many fields of science, engineering and everyday life. Generally, a gauge is regarded as an instrument that measures

and displays some value measured on an ordinal scale between lower and upper limits, such as a road-vehicle's fuel gauge.

- Probes – draw the analogy with space probes, which are dispatched to gather information about a planet or deep space. Through this analogy our own instrumentation probes are dispatched to gather information about the services that a particular component depends on.
- Monitors – are often used in medicine and health care to keep a check on some aspect of a patient's health, such as heart, respiratory or blood-pressure rates. Such monitors allow doctors to monitor behaviour without dramatic surgery – for example, a heart monitor allows a doctor to monitor a patient's heart without having to perform open-heart surgery. The monitor instrumentation service described in the thesis uses a similar analogy. It allows a components behaviour to be monitored at runtime without having to abruptly stop the system at a specific point in order to obtain information.

5.6 Chapter Summary

This chapter has outlined the main functional and operational requirements of instrumentation services and the different types of parameter that must be measured and monitored. The chapter has provided a classification of the different types of instrumentation service based on their roles and responsibilities. The set of ten basic instrumentation service operations may now be considered further and, in particular, formally specified, which is the subject of the next chapter. The chapter has also provided descriptions of the roles of each of the instrumentation services, which will be considered further in chapter 7. The instrumentation services were based on similarities with instrumentation used in conventional engineering or the physical sciences.

The analysis of requirements is intended to serve the remainder of the thesis, although it may prove useful to other practitioners in the field of distributed systems understanding. This chapter alone has provided a useful research contribution since literature pertaining to the basic requirements of instrumentation from first principles is not so abundant in the field of distributed systems understanding and management.

Chapter 6

Formal Model of Instrumentation Services

This chapter provides a formal model of the operational functionality required of instrumentation services as considered previously in chapter 5. The model is developed primarily using Object-Z, [84-86], which is an object-oriented extension to the formal specification language Z. The model also uses some Timed CSP [87] notation, which features in the combined logic-based modelling language of Timed Communicating Object-Z (TCOZ) [41, 88]. This combination was chosen, over other languages, because it combines the data modelling and algorithmic features and state representation of Z together with the process control capabilities of Timed CSP. Object-Z, as the name suggests, also represents the principles of object-orientation, which assists the development of succinct models that are intended for object-based implementations. Object-Z schemas are used to represent the behaviour and interactions that characterize the operational requirements of instrumentation services and, where necessary, Timed CSP operators are used to represent process sequencing, synchronization, concurrency and active objects.

The chapter begins with an overview of formal modelling approaches, a statement of the main aim of the formal instrumentation model and the justification for such a formal model. This is followed by the formal instrumentation model, which is presented as series of related models. In particular the formal models describe: the typing system; formal specifications of middleware lookup services and application-level components are the actual formal specification of instrumentation services. The final model is that of a middleware-based application that encapsulates the previous models.

6.1 Formal Modelling

Formal modelling is used to develop an *abstract* representation of a system of interest. In general, the abstraction will represent the *structural* and *behavioural* characteristics of the system. As opposed to ad-hoc and semi-formal methods of specifying these

characteristics, formal models typically have a sound mathematical framework around which they are developed. Formal models, which are also referred to as *specifications*, may be developed using a variety of formal specification languages. The word *formal* is used to indicate that such languages are based on some mathematical principle (e.g. set theory, predicate calculus or temporal logic) and specifications are developed using a mathematical-like notation as opposed to a programming language notation.

6.1.1 Formal Specification of Systems

Formal models are crucial to specifying systems that fulfil safety-critical roles, but they are being increasingly used to specify systems that fulfil non-critical roles, but raise the need for accurate and concise description. Formal specifications are *unambiguous* if they portray exactly one meaning. A specification is *consistent* if its specified set is non-empty. Specifications must be unambiguous and consistent, but they are allowed to be *incomplete*. Incompleteness of specifications is often unavoidable because anticipation of all the possible system scenarios is not possible.

Formal methods extend on formal specifications in that they refer to the combination of formal specifications and formal *reasoning* about the specifications[4]. This chapter does not go so far as to apply formal methods in the strict sense in that no formal reasoning is performed because the models are simply used to provide the basis of a precise software implementation. Such formal reasoning exceeds the scope of the research.

The choice of a particular formal specification language is significant to the abstraction of a system and hence the resulting specification. For example, the decision as to whether the system should be abstracted in terms of sets, or alternatively in terms of processes would have a strong bearing on the choice of specification language. Several existing formal specification languages are: VDM (a formal language that supports a *model-oriented* specification style and a set of built-in data types); Z (a formal modelling language based on set theory and predicate calculus); temporal logic (a *property-oriented* language for specifying properties of concurrent and distributed systems); CSP and Timed CSP (*process-oriented* languages for specifying concurrent and parallel processes).

[4] Formal Methods = Formal Specification + Formal Reasoning.

This section is not intended to provide a detailed description of formal specification, but merely to introduce sufficient material to support the description of the formal instrumentation model to be considered later in the chapter. However, more detailed treatments of formal specification and formal methods may be found in [89-91]. Before moving on to consider the development of the formal instrumentation model, it is important to clarify exactly what the formal model aims to achieve and to be able to justify the need for the model. These issues are considered below.

6.1.2 Main Aim of the Formal Instrumentation Model

The thesis is primarily concerned with the development of a dynamic instrumentation architecture. The architecture is to provide instrumentation services that can be used to instrument a middleware-based distributed application in order to learn more about its structure and behaviour. However, in this chapter, we are not so much concerned with the *instrumentation services*, but with the *instruments* themselves that are capable of providing the services. Much of the chapter refers to instruments as opposed to their services, which will be considered further in chapters 7 and 8. For this reason, the modelling abstraction is based on the physical concept of an instrument as opposed to the logical concept of an instrumentation service. A consequence of this abstraction is that we will be dealing with *sets* of instruments, which in turn influences our choice of modelling towards a Z-based specification language.

The main aim of this chapter is the development of a series of state models that encapsulates the behaviour and interactions of instruments within the broader context of an application. In order to achieve this aim, we cannot simply consider instrumentation in isolation as we are forced to consider instrumentation along with the other entities with which instruments interact. In particular, we must consider the behaviour and interactions between instruments and middleware services (primarily lookup services) and also instruments and application components. To complete the picture we must also consider the incorporation of instruments within an application and develop formalisms of the states that instruments may assume and formalisms of the basic operations that instruments should expose within an application.

To emphasize the main aim in another way, recall the conventional engineering example of chapter 5, namely an altimeter instrument. This example distinguished between the *physics* governing an altimeter and the *operational* aspects in terms of how an altimeter may be physically incorporated in an air-bound vehicle. If this chapter had been concerned with an altimeter, it would be expected to deliver the necessary formal abstractions governing the incorporation of *any* altimeter within *any* air-bound vehicle. This model would specify the axioms governing: the physical attachment of the altimeter; the connections to control to circuit boards and power supply and the interface commands or basic operations that the altimeter presents to the control system of any air-bound vehicle. However, the formal model need not consider the detailed physics governing the functionality of the altimeter. Of course, the physics would need to be considered, but this consideration could be provided in a separate series of models, thereby simplifying the overall treatment of an altimeter.

Finally, one may question *"why choose a formal modelling approach – couldn't the same concepts be described with UML?"*, which beckons some justification for using a formal approach. The answer to this question lies first in the abilities of formal modelling to produce precise unambiguous descriptions of complex systems and secondly in their ability to precisely describe states and state-based properties and transitions. The formal model is developed using a Z-based language that is considered in the next section and it is worth concluding the current section by emphasizing an important characteristic of Z-based specifications, relating to their use of types.

Every object in a mathematical language such as Z has a unique type, which is represented as a maximal set in a specification. As well as providing a useful stepping-stone to a software implementation, the notion of types means that algorithms can be developed to check the type of every object in a specification. Several type-checking tools exist to support the development of standard Z and Object-Z specifications, such as *CADiZ* [92], *ZTC* [93] and *Wizard* [94]. The models presented in the remainder of this chapter were checked using *ZTC* (for standard Z schemas) and *Wizard* (for Object-Z classes). Assuming these models specify the correct semantics, we may conclude that they provide a precise description of the structure and behaviour of instrumentation within a middleware-based application. The author also believes that the formal models

allows instruments to be considered succinctly in the broader context of a distributed application. A semi-formal approach would have led to a much larger unwieldy model in order to express instruments within the same context.

6.2 The Formal Instrumentation Model

The formal instrumentation model is presented as follows: first the typing system used throughout the model is specified. Secondly, formal models of the basic elements with which instruments interact (lookup services and client/server components) are specified. Thirdly, the formal model of instrumentation itself is specified and fourthly, a formal model of an application is specified to represent the interactions between client/server components, lookup services and instruments within an application.

In order to develop an abstract model of a system's state, we must represent:

- The main entities and their types.
- The relationships between the entities.
- Entity-attribute details, but *only* where the attribute information is essential.
- The constraints that must operate on and between entities.
- The states that the entities may assume.
- The actions or events causing entities to move from one state to another.

So, our first task is that of specifying the types used to declare the main entities as follows.

6.2.1 Typing System

The basic types below are used to represent the types of entities throughout the model.

- *CLASS* – represents a class in a programming language from which object instances can be created.
- *PROXY* – represents a standard proxy in a middleware application that is used for client-server communications. A proxy is regarded as an instance of a *CLASS*

- *STRING* – represents a string that is used to identify an *Item*'s id and the *Item*'s attributes. An *Item*, which is considered further below, is an object that contains a *PROXY*, an *id* and a set of attributes (*attrs*). The *id* and attributes are using during the lookup process were they are compared against the *id* and attributes of a Template (also considered further below).
- *INTERFACE* – represents an interface that is typically used in a programming language to specify the method signatures for remote methods.
- *METHOD* – represents a method (or member function) of a *CLASS* which implements a signature defined in an *INTERFACE*.

The model also uses several "so-called" *free* or *discrete* types to represent states, events, messages, roles and invocation results.

The *ROLE* free-type specifies the roles that may be adopted by application components. The *CLIENT* role represents a component that uses a service provided by some other component. The *SERVER* role represents a service-providing component.

ROLE ::= *CLIENT* | *SERVER*

The *RESULT* free-type specifies the two result conditions that follow the invocation of a method (of type *METHOD*). The *SUCCESS* value indicates that the method completed successfully, whereas the *EXCEPTION* value indicates that an *EXCEPTION* was encountered during the execution of the method.

RESULT ::= *SUCCESS* | *EXCEPTION*

As we shall see later, the state of an instrument is represented by the triple (*REG*, *ATTACH*, *JOIN*), which reflects whether or not an instrument is registered with a lookup service; whether or not it is attached to a component and whether or not it is joined to a group of other instruments. The free-types *REG*, *ATTACH* and *JOIN* are used to provide a compound value or Cartesian product for the state triple.

REG ::= *REGISTERED* | *UNREGISTERED*
ATTACH ::= *ATTACHED* | *DETACHED*
JOIN ::= *JOINED* | *UNJOINED*

The *EVENT* free-type is associated with the above states in that it contains the discrete values that signal events, which cause an instrument to enter the state associated with the value of a particular event.

$$EVENT ::= REGISTER \mid UNREGISTER \mid ATTACH \mid DETACH \mid JOIN \mid UNJOIN$$

The *MESSAGE* free-type describes the different contents of messages that may be received or sent by instruments. The first type is a simple *DATA* type; the second is a constructed event message type $CEVENT\langle\langle EVENT\rangle\rangle$, which takes a value of *EVENT* type to construct an event message; the third type of message specifies a serious *ERROR* condition that prevents message transmission from continuing. Such errors may range from a Java exception to a serious network failure and the details of such errors are not of significance, but what is significant is that receive/send transmissions are irrecoverably halted.

$$MESSAGE ::= DATA \mid CEVENT\langle\langle EVENT\rangle\rangle \mid ERROR$$

Finally, the free-type *TRANS* specifies that an instrument has just undergone a state transition from an initial state to a final state defined by one of the free types *REG, ATTACH, JOIN.*

$$TRANS ::= REGTRANS \mid UNREGTRANS \mid ATTACHTRANS \; DETACHTRANS \mid JOINTRANS \mid UNJOINTRANS$$

The free-type definitions above offer no extra descriptive power above and beyond what can be described by given sets and sufficiently rich axioms. However, the main advantage of free-type definitions is that they can be used to produce more elegant and compact data type constructions than corresponding given-set based descriptions [89].

The complete typing system is repeated below, which is now used to specify the formal models of lookup services, components, instruments and applications respectively.

$$
\begin{array}{lll}
ROLE & ::= & CLIENT \mid SERVER \\
RESULT & ::= & SUCCESS \mid EXCEPTION \\
REG & ::= & REGISTERED \mid UNREGISTERED \\
ATTACH & ::= & ATTACHED \mid DETACHED \\
JOIN & ::= & JOINED \mid UNJOINED \\
EVENT & ::= & REGEVT \mid UNREGEVT \mid ATTACHEVT \mid DETACHEVT \\
 & & \mid JOINEVT \mid UNJOINEVT \\
MESSAGE & ::= & DATA \mid CEVENT\langle\!\langle EVENT \rangle\!\rangle \mid ERROR \\
TRANS & ::= & REGTRANS \mid UNREGTRANS \mid ATTACHTRANS \mid DETACHTRANS \\
 & & \mid JOINTRANS \mid UNJOINTRANS
\end{array}
$$

Figure 6.1: basic typing system

6.2.2 Lookup Service and Application-level Component Models

The lookup service plays a crucial role in any middleware-based application. Existing middleware technologies provide lookup services and protocols that facilitate lookup by name (naming service) or by type (trading service). Whichever lookup approach is used a lookup service is essentially a repository, which stores a collection of application component proxies and provides protocols to allow other components to access these proxies by name or type.

The model uses a simplified abstract representation of a lookup service using the Object-Z class *LUS*. This is shown below in Figure 6.2 together with the associated schemas *Item* and *Template* and the axiom *instof*.

$$instof : CLASS \leftrightarrow PROXY$$

$$\exists c : CLASS \bullet (\exists_1 p : PROXY \bullet instof(c) = p \wedge p \neq NULL)$$

Item

$$
\begin{array}{l}
id : STRING \\
proxy : PROXY \\
attrs : \mathbb{P}\, STRING
\end{array}
$$

Template

$$
\begin{array}{l}
id : STRING \\
types : \mathbb{P}\, CLASS \\
attrs : \mathbb{P}\, STRING
\end{array}
$$

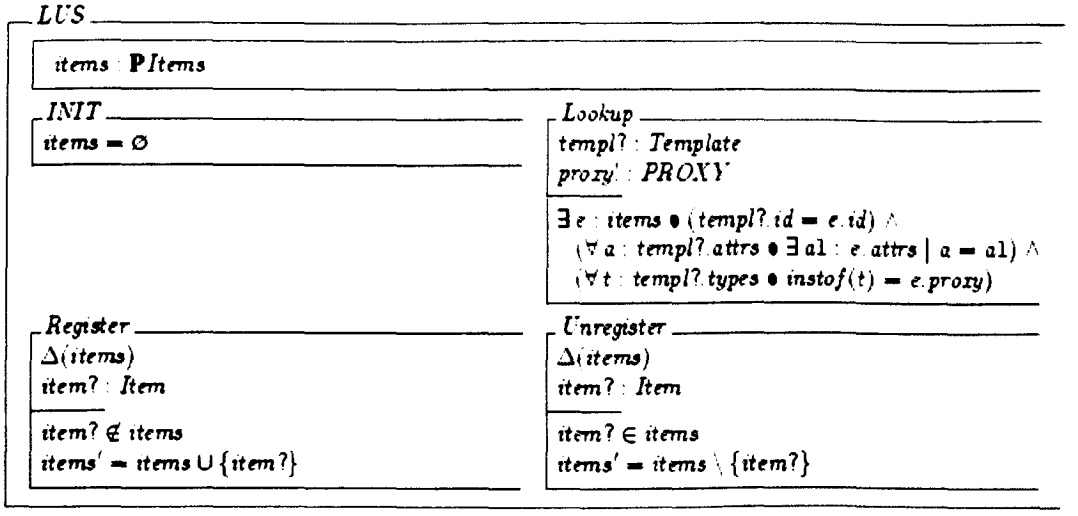


Figure 6.2: lookup service class

The state schema of *LUS* is represented as a set of *Items*. The *LUS* class contains an *INIT* operation schema, which simply initializes the set of *Items* to the empty set to represent the state of a lookup service daemon when it is started. The LUS class also contains Register, Unregister and Lookup operations. The Register and Unregister operations are used to register/unregister an Item respectively. The Lookup operation is a crucial LUS operation which allows clients to find a specific Item by matching the id and attributes of a Template against the items currently registered with the LUS. The criteria for comparing an Item against a Template are specified below.

Because the eventual implementation is to be Jini-based the criteria for lookup matching for a Jini Lookup Service have been used in the Lookup schema operation. The criteria have been rephrased from [95] in order to apply to the data types used in the model. Items in the lookup service are matched using an instance of Template. A service item (*item*) matches a service template (*tmpl*) iff:

- *item.id* equals *tmpl.id*
- *item.proxy* (the service proxy object) is an instance of every type in *tmpl.types*
- *item.attrs* contains at least one matching entry for each entry template in *tmpl.attrs*

The Object-Z class Component, shown below in Figure 6.3, is used to provide an abstract representation of a component that may feature in a middleware-based application.

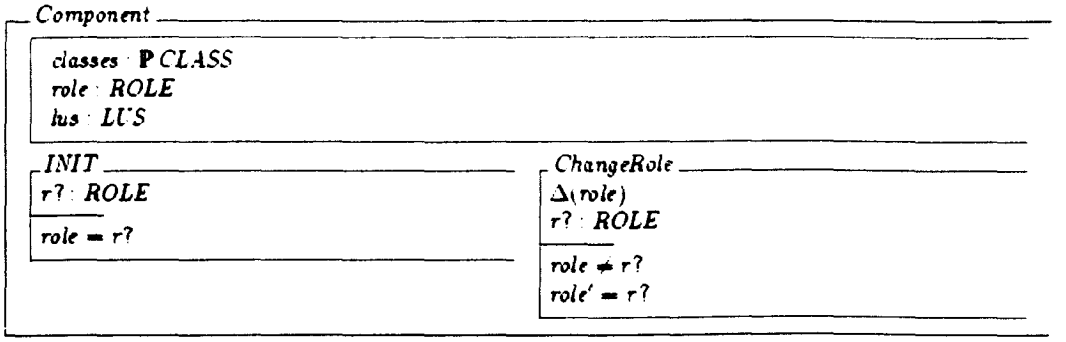
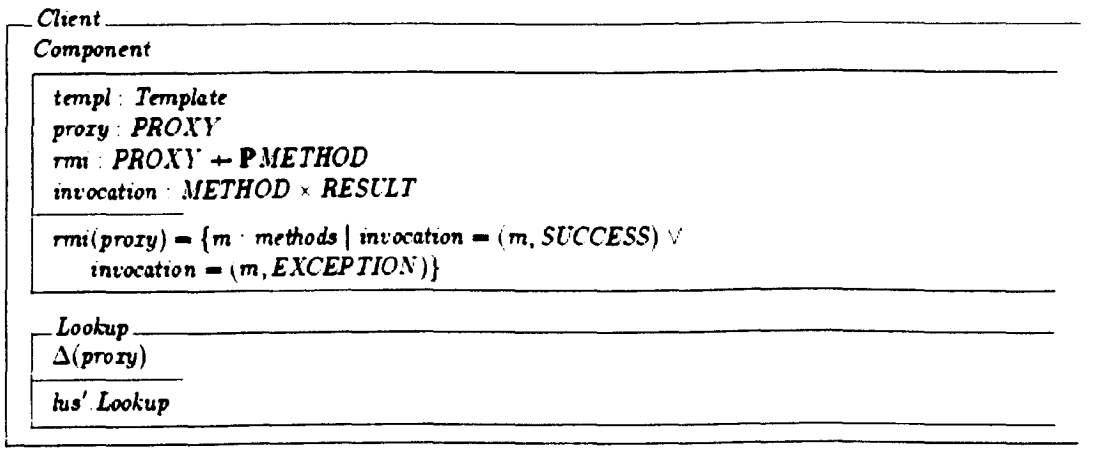


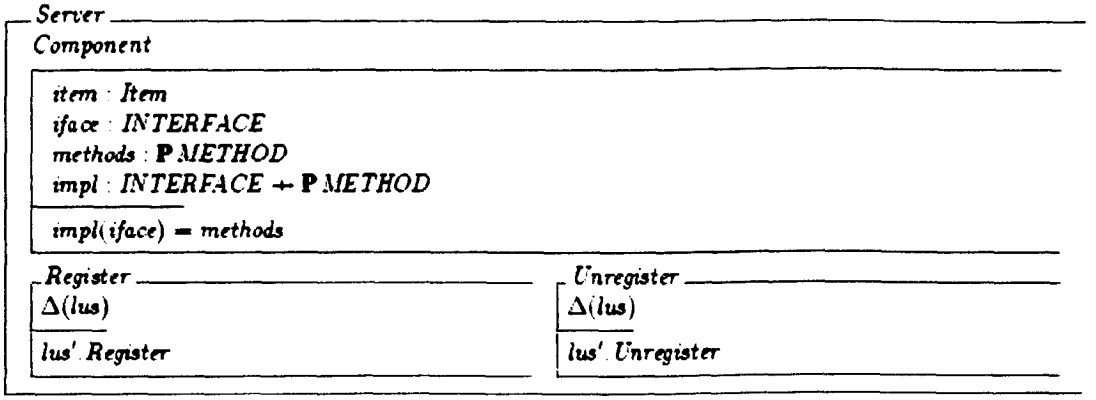
Figure 6.3: component class

The state schema of *Component* is represented by a set of classes (*classes*), a role (*role*) (which may be client or server), and a lookup service with which the component can be associated (*lus*). The *Component* class contains an *INIT* operation schema, which simply sets the role of the component when an instance of the *Component* class is created. The *ChangeRole* operation schema allows a client to take on the role of a server or vice-versa.



The *Client* class schema inherits the class of *Component* to represent a client component in an application. The state schema defines a template attribute *templ* that is used to find a match against an Item stored in the lookup service. The *proxy* attribute is a proxy that is returned by matching *templ* against an *Item* in the lookup service, thereby allowing the client to make RMI calls on the proxy. Note that the state of *proxy* is affected by the Lookup operation since *proxy* takes on a value when a match is found. The state schema

also defines a total function *rmi* that maps a proxy to a set of methods to represent the potential of a client to perform RMI calls on a proxy. A single remote invocation is represented by the Cartesian product (*invocation*). The *invocation* attribute represents the invocation of method *m*, which leads to a result r as the ordered pair (*m*, *r*), were the result is success or an exception. The *rmi* function maps the proxy onto the set of methods for which the results may have successful outcomes or lead to exceptions. The *Lookup* schema operation defines a single axiom which is that of the lookup service's own *Lookup* operation.



The *Server* class schema also inherits the class of *Component* to represent a server component in an application. The state schema defines an Item (*item*), an interface (*iface*), a proxy (proxy) and a set of *methods*. Note that *item* contains the server's proxy along with its id and attributes. The state schema defines a total function *impl* that maps an interface to a set of methods to represent the implementation of the server. The Server class also contains *Register* and *Unregister* operation schemas that allow the server to register/unregister its item with a lookup service. The axiom parts of the *Register/Unregister* schema operations are those of the lookup service's own *Register/Unregister* operations. Note that the state of *lus* is affected by the Register/Unregister operations as a new item is added to *lus*, or an existing item is removed respectively.

One aspect of the registration procedure that is not elaborated in the model is the *discovery protocol*. In middleware-based applications, before a server registers, it must first discover a lookup service with which it may then register. The simplest form of discovery protocol is that the component registers with the *nearest* lookup service. The

nearest lookup service is usually running on the same computer as the component and this lookup service is sometimes referred to as the lookup service running on *localhost*. More advanced discovery mechanisms may be used to allow components to discover and register with an alternate lookup service to that of the nearest. Throughout this model, the simple discovery protocol is assumed to avoid additional complexities.

One may argue that the previous specifications do not do justice to the functionality of a lookup service and client server application components. However, the specification does provide a sufficient abstraction for the development of the instrumentation and application models to follow, which is the main concern of this chapter.

6.2.3 Instrumentation Model

Having considered the formal models of the basic elements with which instruments interact (lookup services and service providing components) we may now proceed to consider the formal model of instruments themselves. However, before doing so, we must introduce some preliminary specifications that simplify the instrument model, which are shown in Figure 6.4.

$$
\begin{array}{|l}
\hline
DProxy \\
ifaces : \mathbb{P}\, INTERFACE \\
\hline
\end{array}
$$

$$
\begin{aligned}
INSTRUMENT ::= {} & | \; MONITOR \mid CMONITOR\langle\!\langle DProxy \rangle\!\rangle \mid GAUGE \mid CGAUGE\langle\!\langle DProxy \rangle\!\rangle \\
& | \; PROBE \mid CPROBE\langle\!\langle DProxy \rangle\!\rangle \mid LOGGER \mid CLOGGER\langle\!\langle DProxy \rangle\!\rangle \\
& | \; ANALYZER \mid CANALYZER\langle\!\langle DProxy \rangle\!\rangle
\end{aligned}
$$

$$
\begin{array}{|l}
\hline
[A, B, C] \\
firstElem : (A \times B \times C) \rightarrowtail A \\
secondElem : (A \times B \times C) \rightarrowtail B \\
thirdElem : (A \times B \times C) \rightarrowtail C \\
\hline
\forall a : A; b : B; c : C \bullet (firstElem(a, b, c) = a \wedge secondElem(a, b, c) = b \wedge thirdElem(a, b, c) = c) \\
\hline
\end{array}
$$

Figure 6.4: dynamic proxy and instrument types

The first schema *DProxy* represents a *dynamic proxy*, which is a special kind of proxy, which is crucial to instrumentation services. It should be noted that a dynamic proxy is entirely different to a proxy that a client uses to make RMI calls on a server. The dynamic proxy is included as part of J2SE's API and it is represented as a simple schema that

consists of a set of interfaces (*ifaces*). Any class that chooses to implement a dynamic proxy enters a contract, through which it must supply a generic *invocation handler* that is capable of invoking the methods specified by any interface within the set of interfaces. The implementation details of the dynamic proxy will be considered in far greater detail in chapter 8, but for the current chapter, we are concerned with the inclusion of the dynamic proxy within the Object-Z instrument class.

The second specification is that of an *INSTRUMENT* free-type that specifies the different types of instrument. The format of the *INSTRUMENT* type is a value depicting the type of a particular instrument and a constructer, which takes a *DProxy* entity to construct an instrument of the desired type (e.g. CMONITOR$\langle\langle$DProxy$\rangle\rangle$). The third specification is a generic schema that defines three total functions: *first*, *second* and *third*. These total functions specify mappings that return the first, second and third elements of an attribute whose type is the Cartesian product ($A \times B \times C$). As we shall se shortly, these total functions are used in the axioms that involve the state of an instrument that is represented by triple (*REG*, *ATTACH*, *JOIN*) outlined previously (Section 6.2.1).

Equipped with these specifications, we may move on to consider the abstract specification of an instrument. From chapter 5, we recall the set of basic instrument operations as:

- Register – registers an instrumentation service with an active lookup service.
- Unregister – unregisters an instrumentation service that was previously registered with an active lookup service.
- Attach – attaches a registered instrumentation service, via a dynamic proxy, to an application component, so that the instrumentation service may monitor and measure the component.
- Detach – detaches an instrumentation service that was previously attached to an application component still leaving the instrumentation service registered.
- Join – joins an instrumentation service, via its dynamic proxy, to a group of instrumentation services that are already joined so that the group may communicate and share each others' services.

- Unjoin – unjoins or removes an instrumentation service that was previously joined to a group of other instruments.
- Read – allows an instrument that is joined to a group of other instruments to receive information from the other instruments.
- Write – allows an instrument that is joined to a group of other instruments to send information to the other instruments.
- Invoke – allows an instrument to intervene in the method invocations that a client component makes on an application server component.
- Notify – allows an instrument that is joined to a group of other instruments to notify the other instruments of changes in its state via a multicast Write operation.

These are the very operations that must be represented in the formal instrumentation state model. The Object-Z class *Instrument*, shown below in Figure 6.5, provides an abstract representation of an instrumentation service. The class schema first inherits the classes of *LUS* and *Server* that were considered previously. The state schema and the various operation schemas that make up the *Instrument* class follow the inheritance specifications and these schemas are each described further below.

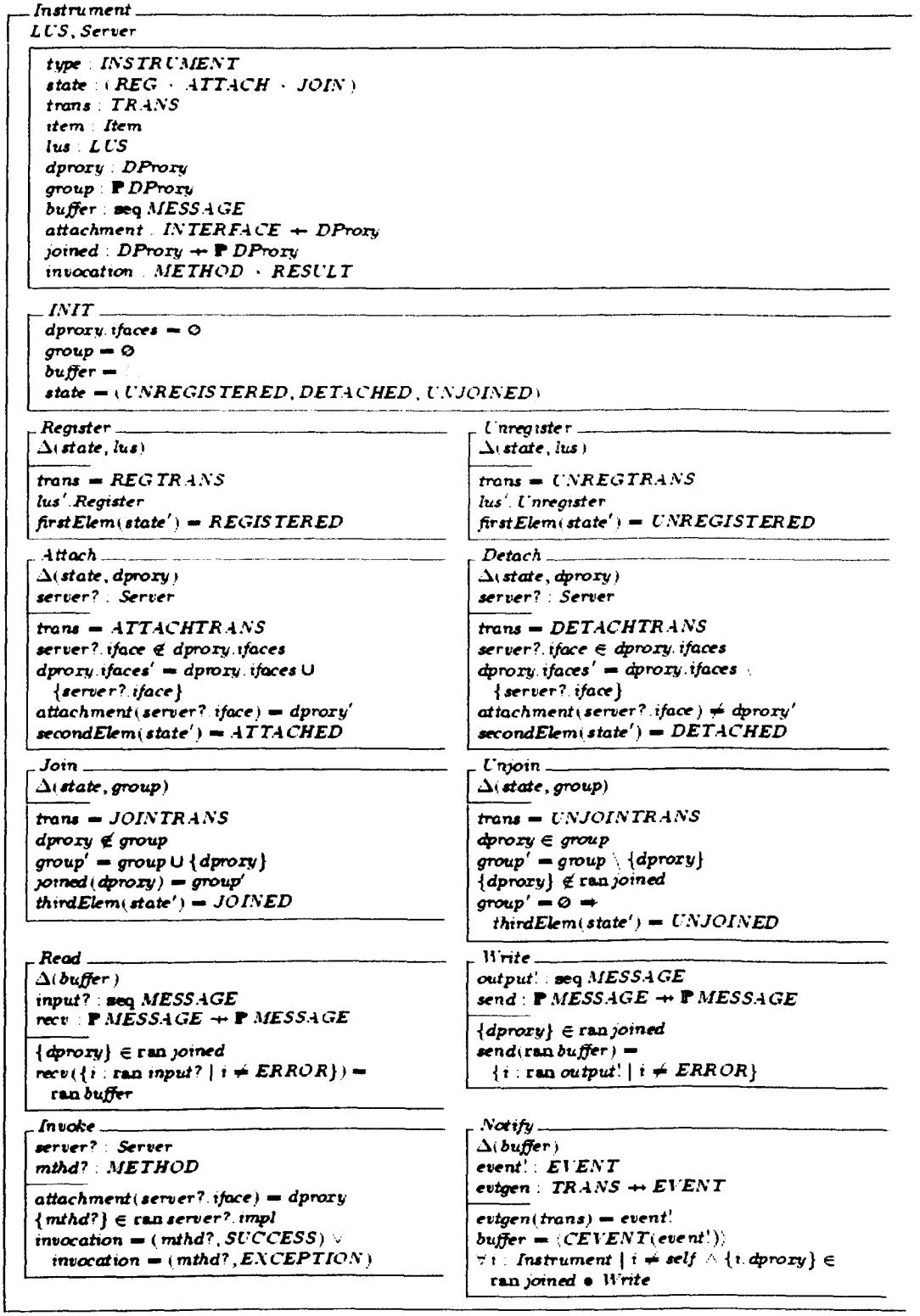


Figure 6.5: instrument class

The state schema of *Instrument* consists of the following state attributes:

- *type* – the type of instrument (monitor, gauge, probe, logger, analyzer).
- *state* – the state of an instrument that represented by triple (*REG*, *ATTACH*, *JOIN*).
- *trans* – the transition that takes an instrument from some initial state to a final state (*REGTRANS*, *ATTACHTRANS*, *JOINTRANS*).
- *item* – the instruments service item, which contains the instrument's standard proxy.
- *lus* – the lookup service with which the instrument registers/unregisters
- *dproxy* – the lookup service with which the instrument registers/unregisters.
- *group* – the group of instruments that have been joined via their dynamic proxies and have the potential to communicate with each other using *Read*/*Write* operations.
- *buffer* – a read/write buffer represented as a sequence of messages.
- *attachment* – a total function that maps a server's standard proxy to an instruments dynamic proxy, thereby implying that the instrument is attached to the server component.
- *joining* – a total function that maps the dynamic proxy of an instrument to a set of dynamic proxies of other instruments, thereby implying that the instrument is *joined* to a group of other instruments. As a member of the group of instruments, the joined instrument may receive/send messages from/to other instruments using *Read*/*Write* operations.
- *invocation* - an attribute that represents the invocation of method m, which leads to a result r as the ordered pair (*m*, *r*), similar to the invocation attribute in the *Client* state schema.

The Instrument class contains an *INIT* operation schema, which first initializes the set of interfaces (that an instruments dynamic proxy may implement) to the empty set, the

group of instruments to the empty set and the read/write buffer to the empty sequence. The *INIT* operation then initializes the initial state to the triple (*UNREGISTERED*, *DETACHED*, *UNJOINED*).

The remaining schema operations specify the basic instrument operations identified in chapter 5 as explained below. The operations *Register*, *Unregister*, *Attach*, *Detach*, *Join* and *Unjoin* all change the state of the instrument. These state transitions are represented by the Δ(*state*) specifications and axioms that use *firstElem*(*state'*), *secondElem*(*state'*) and *thirdElem*(*state'*), which specify the final state using the ' (apostrophe) decoration. All of the schemas operators are applied to the current instrument (*self*) and when this is obvious the keyword *self* may be omitted.

Register/*Unregister* – register/unregister an instrument with/from an input the lookup service (*lus*) in a similar fashion to the Register/*Unregister Server* schema operations. *Register*/*Unregister* also alter the state of the current instrument Δ(*state*). The axioms of *Register* specify that the registration/unregistration follow from the transition REGTRANS. The *firstElem*(*state'*) = *REGISTERED* axiom specifies that the *REG* component of *state* after the *Register* operation is *REGISTERED*. The axioms of *Unregister* essentially specify conditions that are the reverse of *Register*. As was the case for the *Component* class, the details of how instrument's discover a lookup services (i.e. the discovery protocol) with which they may register are not elaborated in the *Register* schema. Again, it is assumed that instrument's will discover the nearest lookup service and register with that lookup service.

Attach/*Detach* – attach/detach an instrument to/from an input application-level server *server*? and alter the state of the current instrument Δ(*state*). The axioms of *Attach* specify that the server's interface must not already exists in the set of interfaces implemented by the instrument's dynamic proxy; the set of interfaces implemented after the *Attach* operation *dproxy.ifaces'* is the union of the original set and the input server's interface *dproxy.ifaces'* = *dproxy.ifaces* ∪ {*server*?.*iface*}. The total function *attachment* maps the server's interface to the dynamic proxy of the instrument as *attachment*(*server*?.*iface*) = *dproxy'*. Finally, *secondElem*(*state'*) = *ATTACHED* specifies

that the *ATTACH* component of *state* after the *Attach* operation is *ATTACHED*. The axioms of *Detach* essentially specify conditions that are the reverse of *Attach*.

Join/Unjoin – join/unjoin the dynamic proxy of an instrument to/from a group of dynamic proxies, *group*, that already join a group of instruments. *Join/Unjoin* also alter the state of the current instrument $\Delta(state)$. The axioms of *Join* specify that the dynamic proxy of the current instrument must not already exist in the group of proxies; the group of dynamic proxies after the *Join* operation *group'* is the union of the initial group and the current instrument's dynamic proxy $group' = group \cup \{dproxy\}$. The total function *joined* maps the instrument's dynamic proxy to the group of dynamic proxies $joined(dproxy) = group'$. Finally, $thirdElem(state') = JOINED$ specifies that the *JOIN* component of *state* after the *Join* operation is *JOINED*. The axioms of *Unjoin* essentially specify conditions that are the reverse of *Join*.

Read/Write – allow the current instrument to receive/send messages from/to other instruments. Read alters the state of the current instrument buffer $\Delta(buffer)$, whereas *Write* does not, since the current contents of *buffer* are sent to a receiving instrument. These schema operations operate on the buffer attribute in the state schema – *buffer* : **seq** *MESSAGE*). In Z, a sequence is a special kind of function that is used to model an ordered collection of objects, whereas with sets there is no ordering. A sequence of *MESSAGE* items $s = \langle msg_1, msg_2, msg_3 \rangle$ is represented as the set of mappings $\{1 \mapsto msg_1, 2 \mapsto msg_2, 3 \mapsto msg_3\}$ and the contents of s is its range (**ran** s), which provides the set of objects with the indices 1, 2 and 3 removed as $\{msg_1, msg_2, mssg_3\}$.

The axioms of *Read* specify that the dynamic proxy of the current instrument must be joined to a group of dynamic proxies. The total function *recv* then maps the contents of the input sequence *input*? (i.e. **ran** *input*?) to the contents of the current instrument's buffer (i.e. **ran** *buffer*), assuming no *ERROR* element exists in **ran** *input?*.

$$recv(\{i : \textbf{ran}\ input? \mid i \neq ERROR\}) = \textbf{ran}\ buffer$$

The axioms of *Write* are similar to those of *Read*, except that for *Write* an output message *output?* is produced.

Invoke – is used by the current instrument to invoke the methods of a service providing input server component *server?*. The capability for instrumentation services to invoke methods on service providing components has already been mentioned in chapter 5 and will be considered further in chapters 7 and 8. Through this capability, instrumentation services may intervene client invocations and gather information about the parameters and results of the invocations. *Invoke* operates on an input method *mthd?* of the input server component *server?*. It does not affect the *state* attribute of the current instrument (since it has already changed). The axioms of *Invoke* specify that there must be an attachment mapping between the input server's interface and the current instrument's dynamic proxy as *attachment*(*server?.iface*) = *dproxy*; the input method must be in the range of the *impl* total function (**ran** *server?.impl*). The *invocation* attribute specifies that the invocation must be either of the ordered pairs (*mthd?*, *SUCCESS*) or (*mthd?*, *EXCEPTION*).

Notify – is used by the current instrument to notify other instruments of any change in its state Δ(*state*). *Notify* works in a multicast fashion in that the current instrument notifies all instruments in the group of which the current instrument is a member. *Notify* generates an output *event!*, which is sent to all other instruments in the group using the *Write* operation. In the axiom part of *Notify* the total function *evtgen* is used to generate the output event. The instrument's buffer must then equate to a message constructed from the output event as *buffer* = ⟨ *CEVENT*(*event!*) ⟩. The message is then sent to all instruments in the group (i.e. the instrument's whose dynamic proxies are in **ran** *joined*) except for the current instrument *self*. The final axiom also specifies that the message is sent by the *Write* schema operation.

$$\forall i : Instrument \mid i \neq self \land \{i.dproxy\} \in \mathbf{ran}\ joined \bullet Write$$

As already mentioned, the main aim of the chapter is the development of a formal state model that encapsulates the behaviour and interactions of instruments within an application and the *Instrument* class is crucial to achieving this aim. It is apparent that the *Instrument* class contains no information that specifies how instrumentation services measure or record the various parameters of an application. The details relating to the

measurement and recording functionality of instrumentation services will be the subject of chapters 7 and 8.

Before we move on to consider the formal model of an application, it is worthwhile re-emphasizing why a formal model is necessary. It is the author's belief that the separation of behaviour and interaction from the measurement and recording functionality leads to a simpler concise model of instrumentation. It is also the author's belief that the formal treatment of behaviour and interaction delivers the precise structure, states and axioms. These structure, states and axioms cannot be represented with such precision using less formal approaches. There is also a likelihood that aspects of the formal model may be overlooked with less formal approaches. These views are based on the clarity of the abstraction that can be achieved through a formal model and the fact that the formal model can be checked for correctness and precision using tools such as *ZTC* and *Wizard*.

In contrast, the functionality governing the measurement and recording aspects of instrumentation, does not justify a formal treatment, as we shall discover in chapters 7 and 8. This is so because measurement and recording functionality is not so tightly *coupled* to the elements that are being measured and recorded (i.e. the elements that make up a distributed application, such as middleware and the application's software components). However, where the behaviour and interaction model is concerned, the coupling between instrumentation services and application components is far tighter.

On this note, we may proceed with the formal model to further develop our understanding of instrumentation operating in the context of a middleware-based application

6.2.4 Application Model

The Object-Z class *Application*, shown below in Figures 6.6 and 6.7, combines the lookup service, client/server component and instrumentation models considered previously. The class provides an abstract representation of an *active* (i.e. running) middleware-based application that incorporates dynamic instrumentation. The *Application* class schema inherits the classes of *LUS*, *Client*, *Server* and *Instrument* considered previously. As *Application* is an active class (unlike the *LUS*, *Server*, *Client*

and *Instrument* classes) it contains a TCOZ *MAIN* process that will be considered later in this section.

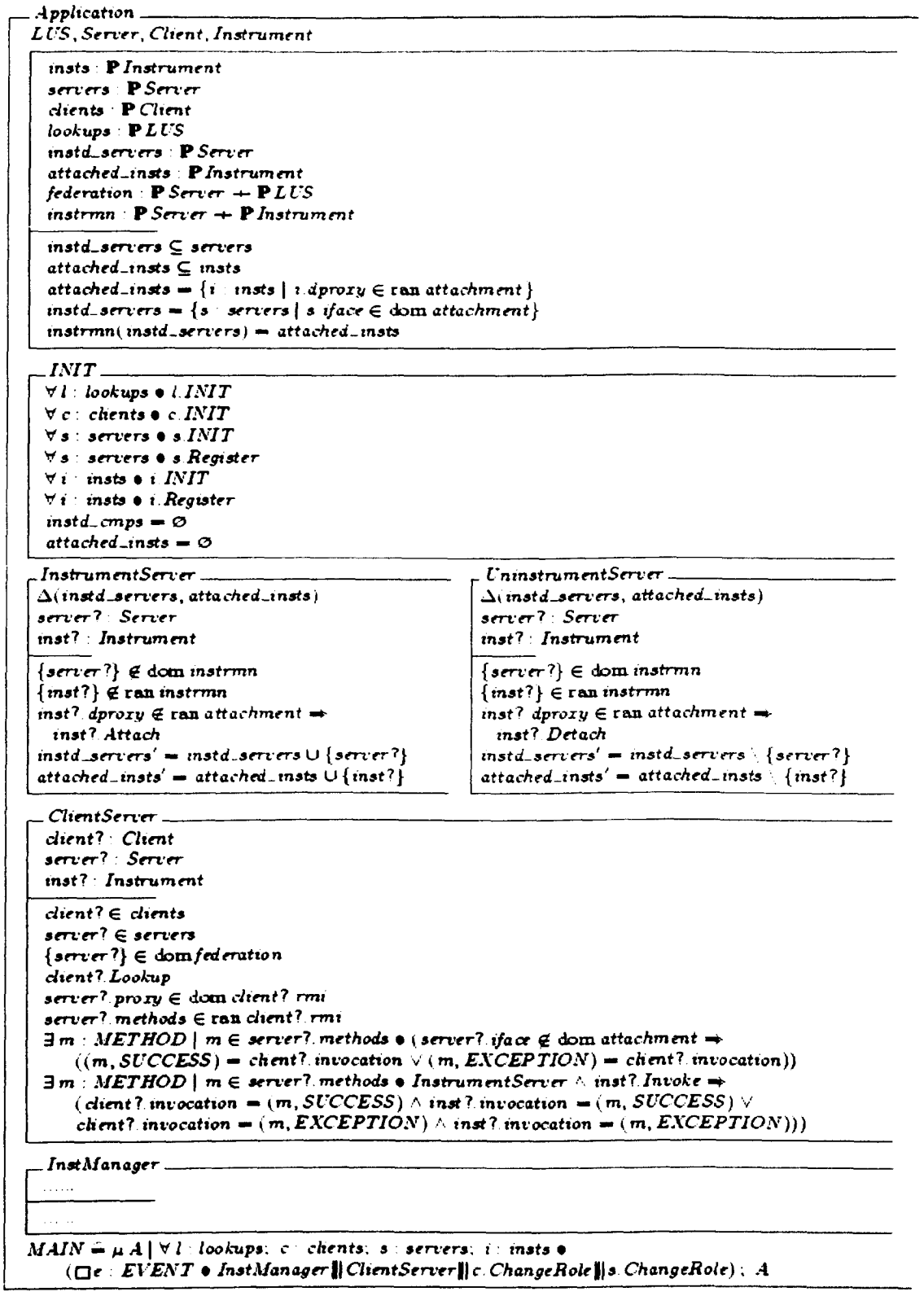


Figure 6.6: application class

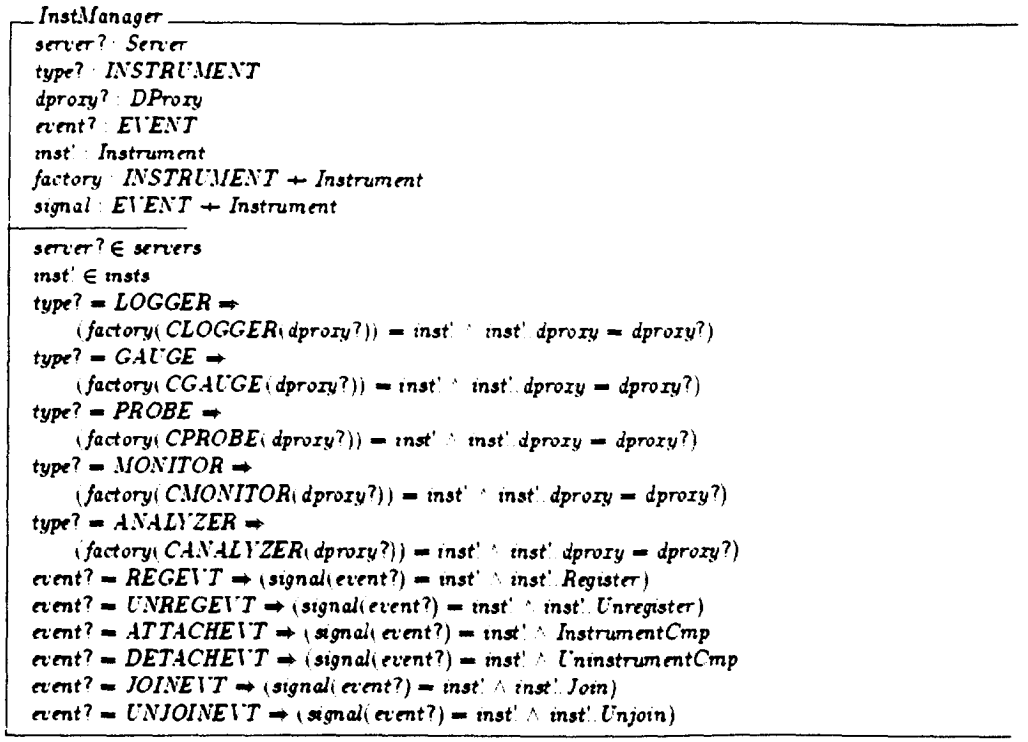


Figure 6.7: application class - instrument manager schema operation

The state schema and the various operation schemas that make up the *Application* class follow the inheritance specifications and these schemas are each described further below.

The state schema of Application consists of the following state attributes:

- *insts* – the set of instrument s (monitors, gauges, probes, loggers, analyzers).
- *servers* – the set application server components that are assumed to be registered with a lookup service when the application is active.
- *lookups* – the set of lookup services that are running.
- *instd_servers* – the set of currently instrumented server components (i.e. those that are attached to instruments).
- *attached_insts* – the set of instrument services that are currently attached to application server components.
- *federation* – a total function that maps a set of registered application server components to a set of lookup services with which the application server

components are registered. The *federation* function represents the federation of application services that the application's server components provide.

- *instrmn* – a total function that maps a set of application server components to a set of instruments, which effectively instrument the server components. *instrmn* essentially provides a snapshot of the applications instrumentation mappings.

The axiom part of the state schema first specifies that *instd_servers* is a subset of *servers* ($instd\_cmps \subseteq cmps$) and *attached_insts* is a subset of *insts* ($attached\_insts \subseteq insts$). Note that these axioms involve the subset $\subseteq$ operator and not the strict subset operator $\subset$, so it is possible for the sets to contain the same elements. The next axiom for *attached_insts* specifies it to be the set of instruments such that the dynamic proxy of each instrument *i* is in the range of the total function *attachment*.

$$attached\_insts = \{ i : insts \mid i.dproxy \in \mathbf{ran}\ attachment \}$$

The next axiom for *instd_servers* specifies it to be the set of components such that the interface of each server component *s* is in the domain of the total function *attachment*.

$$instd\_servers = \{ s : servers \mid s.iface \in \mathbf{dom}\ attachment \}$$

Equipped with these two axioms, the *instrmn* axiom simply maps *instd_servers* to *attached_insts*.

The *Application* class contains an *INIT* operation schema, which initializes the elements of the sets: *lookups*, *clients*, and *servers* using their own *INIT* operations and also registers the *servers* and *insts* using the *Register* operations defined in their associated classes. The final axioms of the *INIT* operation schema specify *instd_servers* and *attached_insts* as empty sets.

InstrumentServer/*UninstrumentServer* – provide "transaction-like" operation schemas in that they combine all the axioms that are required to take a component and an instrument from an uninstrumented state to an instrumented state (*InstrumentCmp*) and vice versa (*UninstrumentServer*). *InstrumentServer*/*UninstrumentServer* also alter the state of the state schema attributes *instd_servers* and *attached_insts* as Δ(*instd_servers*,

attached_insts). As we shall see shortly from the *InstManager* schema, these schema operations are triggered by *ATTACH* and *DETACH* events respectively. The first and second axioms specify that *server?* must not be in the domain of the total function *instrmn* (**dom** *instrmn*) and *inst*? must not be in the range of *instrmn* (**ran** *instrmn*). The third axiom states that if *inst*? is not attached (i.e. its dynamic proxy is not in the range of function *attachment*) then its state variables must be those that result from the *Attach* operation of the *Instrument* class.

$$inst?.dproxy \notin \mathbf{ran}\ attachment \Rightarrow inst?.Attach$$

The final two axioms specify the final states of *instd_servers* and *attached_insts* as *instd_servers'* and *attached_insts'* as set unions such that *instd_servers'* = *instd_servers* $\cup$ {*server?*} and *attached_insts'* = *attached_insts* $\cup$ {*inst?*}. The axioms of *UninstrumentCmp* essentially specify conditions that are the reverse of InstrumentCmp.

ClientServer – represents the client server communications that may take place in a middleware-based application that are achieved via clients invoking methods on servers. The input parameters for *ClientServer* are: the server component *server*?, the client component *client*? and an instrument *inst*? that may be used to intervene the method invocations made upon *server*? when it is instrumented.

The first two axioms of *ClientServer* specify that *client*? and *server*? must be members of the their associated sets clients and servers that constitute the application's components. The third axiom specifies that *server*? must be in the domain of the total function *federation*. The fourth axiom specifies the client lookup using the client's own *Lookup* schema operation. The fifth and sixth axioms specify that the server's proxy must be in the range of the client's *rmi* total function and the server's methods must be in the range of the *rmi* function respectively. These two axioms essentially specify that the client has the capability to perform RMI calls on any of the server's methods.

The seventh axiom specifies the client's invocation of method *m* on the server, when the server is not instrumented, which may lead to either of the ordered pairs (*m*, *SUCCESS*) or (*m*, *EXCEPTION*). In more detail, the axiom states that if a method *m* that is a member of *server?.methods* exists, then, for this method, it is true that, when the server is not instrumented ($\{server?\} \notin \mathbf{dom}\ instrmn$), it follows that any invocation of this method by

client? will result in either *SUCCESS* or an *EXCEPTION* (*client*?.*invocation* = (*m*, *SUCCESS*) ∨ *client*?.*invocation* = (*m*, *EXCEPTION*)).

The final axiom is similar, except it specifies invocation when the server is instrumented. This is indicated by the *InstrumentServer* operation and the inst?Invoke operation (*InstrumentServer* ∧ *inst?.Inkove*). If this final axiom is satisfied then, as for axiom seven, it follows that for any invocation of this method, on the instrumented *server*?, by *client*? the *invocation* attribute will be either of the ordered pairs (*m*, *SUCCESS*) or (*m*, *EXCEPTION*). However, because *server?* is now instrumented then the instrument's invocation attribute *inst?.invocation* will also be (*m*, *SUCCESS*) or (*m*, *EXCEPTION*). Effectively this specifies that *inst?* will participate in the invocation, just like *client?* and will receive the same invocation result.

client?.*invocation* = (*m*, *SUCCESS*) ∧ *inst*?.*invocation* = (*m*, *SUCCESS*) ∨
client?.*invocation* = (*m*, *EXCEPTION*) ∧ *inst*?.*invocation* = (*m*, *EXCEPTION*)

Essentially specifies that an invocation by *client*? on an instrumented *server*? that is instrumented by *inst*?, leads to either of the ordered pairs (*m*, *SUCCESS*) or (*m*, *EXCEPTION*).

It must be emphasized that the final axioms specify the states governing method invocations and do not actually elaborate an invocation. The semantics of *client?.invocation* and *inst?.invocation* are considered further in chapters 7 and 8, but for now, we may simply state that *inst*? actually performs the invocation on behalf of *client*?. *inst*? uses the method invocation parameters that are passed to it from *client*?; *inst*? then performs the invocation and returns any results back to *client*?. The approach to achieving this is based on a combination of the Remote Method Invocation (RMI) protocol considered in Section 3.2.4, and the dynamic proxy that will be considered in chapters 7 and 8.

InstManager – shown in Figure 6.7 is the final schema operation of the *Application* class. *InstManager* specifies the states governing an instrumentation manager process. In the programmatic sense, an instrumentation manager may be a remote unmanned application control program, or even a human controlling the applications instrumentation through some instrumentation GUI. The main roles of *InstManager* are the creation of

instruments to demand and the signalling of instruments to change state, via the *Register*, *Unregister*, *Join*, *Unjoin* schemas of the *Instrument* class and the *InstrumentCmp*/*UninstrumentCmp* schemas of the *Application* class.

The attributes for *InstManager* are:

- *server*? – an input component to be instrumented/uninstrumented.
- *type*? – an input type of instrument requested.
- *dproxy*? – an input dynamic proxy that will be used to create an instrument.
- *event*? – an input event that will be used to signal an instrument to change its state.
- *inst*! – an output instrument that the factory will be created by a factory, or signalled to change its state by an instrumentation event.
- *factory* – a total function that maps a type of instrument to the actual instrument created of that type.
- *signal* – a total function that maps an instrumentation event to an actual instrument, such that the instrument assumes the state prescribed by the event.

The first two axioms of *InstManager* specify that *server*? must be a member of the application server components *servers* and *inst*! must be a member of the applications instruments *insts*. Axioms three to seven specify the states governing the construction of instruments. These axioms use the *factory* total function in conjunction with the constructors, such as $CLOGGER\langle\langle DProxy\rangle\rangle$ that were specified in section 6.3.3. Axiom three states that if the input type *type*? is *LOGGER*, then it follows that *factory* maps the constructed *INSTRUMENT* free type *CLOGGER*(*dproxy*?) to the output instrument *inst*! and the dynamic proxy of *inst*! is that of the input dynamic proxy used as the parameter in *CLOGGER*. This is specified as $factory(CLOGGER(dproxy?)) = inst! \land inst!.dproxy = dproxy?$ for a logger instrument and axioms four to seven specify the same for the other types of instrument.

Axioms eight to thirteen specify the state transitions of *inst*! that are signalled by the input event *event*?. Axiom eight states that if the input event is *REGEVT*, then it follows

that *signal* maps *event?* to the output instrument *inst!* and the state variables of *inst!* must be those that result from the *Register* operation of the *Instrument* class. This is specified as *signal*(*event?*) = *inst!* $\wedge$ *inst!.Register* for a *REGEVT* event and the axioms for *JOINEVT*, *UNJOINEVT*, *ATTACHEVT* and *DETACHEVT* events specify similar conditions. Note that as mentioned previously, *InstrumentCmp/UninstrumentCmp* provide "transaction-like" operation schemas in that they combine the *Register* and *Attach* operations of the *Instrument* class. By doing so, they provide all the axioms that are required to take a component and an instrument from an uninstrumented state to an instrumented state (*InstrumentCmp*) and vice versa (*UninstrumentCmp*).

The final specification of the Application class of Figure 6.6 is that of the *MAIN* process that is used in TCOZ to indicate that the Application is being defined as an active class. A *MAIN* process is often defined as a μ expression [89], which is also referred to as a *definite description*. Such expressions are used when it proves difficult to construct a predicate to describe unique behaviour in a complex system. A μ expression may be used to provide a unique binding that satisfies any given constraints.

So, for example, if a *Stack* was declared as an active class, we may write the *MAIN* process for a stack, *S*, as:

$$MAIN \mathrel{\widehat=} \mu S \bullet (Push \,\Box\, Pop);\ S$$

This example *MAIN* process uses the external choice operator to allow choice between *Push* and *Pop* operations, according to what events are requested by the stack's environment.

The *Application MAIN* process is specified as:

$$MAIN \mathrel{\widehat=} \mu A \mid \forall l : lookups;\ c : clients;\ s : servers;\ i : insts \bullet (\Box e : EVENT \bullet InstManager \parallel ClientServer \parallel c.ChangeRole \parallel s.ChangeRole);\ A$$

The *MAIN* process uses the TCSP external choice operator in its intentional form to specify that the environment may choose any appropriate event *e* to determine the behaviour of *InstManager*. The TCSP asynchronous parallel indicates that *ClientServer*, *InstManager* and the *ChangeRole* operation execute concurrently without any synchronization. The final part of the *MAIN* process essentially closes the loop on application *A* using the TCSP notion of process sequencing (*P* ; *Q*), which acts as *P* until

P terminates by communicating $\surd$ and then proceeds as Q. So *MAIN* acts as the *InstManager*/*ClientServer* combination until they terminate and then proceeds as A, which returns back to the *InstManager*/*ClientServer* until A eventually terminates, thereby terminating the *MAIN* process.

6.4 Chapter Summary

This chapter has provided a formal model of the instruments that provide instrumentation services, based on a combination of Object-Z schemas together with some TCSP operators. The overall instrumentation model has been described through a series of related models that have described: the typing system; middleware lookup services and application-level components; the actual formal model of instrumentation service providing instruments. The final model considered the incorporation of instruments within an application and developed the formalisms of the states that instruments may assume and formalisms of the interface commands that instruments should expose within an application.

These state models and axioms will serve as the formal basis for chapter 8, which is concerned with the implementation of instrumentation services. However, for the next chapter, we move on to address the functional requirements of the instrumentation services (i.e. the details relating to the measurement and recording functionality of instrumentation services – similar to the physics of the altimeter analogy).

The chapter has provided a novel contribution by formally specifying the operations required of an abstract instrument in terms of a formal state-based model and associated axioms. In addition, it is hoped that the chapter may go some way towards promoting the use of formal methods for the specification of structural and behavioural characteristics associated with distributed systems.

Chapter 7

An Instrumentation Architecture for Measuring and Monitoring Applications

This chapter considers the development of the instrumentation architecture based on the classification of instrumentation services of chapter 5 (Figure 5.1). The main aim of the chapter is the development of an architecture that fulfils the *functional* requirements defined in chapter 5, namely the measurement and monitoring functionality. The architecture is developed through a series of UML models that represent the functional aspects of measurement and monitoring in a middleware-based application.

The chapter begins by describing how system-wide information (relating to the underlying computing platforms) can be acquired for Java-based application. The chapter then goes on to develop the architecture, based on the classification hierarchy of chapter 5, through a series of semi-formalized UML models. In particular, models are developed that provide the structure and measurement functionality for the actual instantiable instrumentation services of logger, analyzer, gauge, probe and monitor as well as the various support or infrastructure classes represented in Figure 5.1.

The direction in which we are heading is towards the implementation of instrumentation and like the ascent of Everest, this journey is complicated so two separate teams have taken different routes with a view to meeting at the summit. The first team took the formal specification route of chapter 6, whereas the second team is about to start the route of the current chapter. Both teams intend to meet at the summit of chapter 8, where they will combine their ideas to pitch the flag of the expedition, namely the implementation of instrumentation architecture.

7.1 Accessing Application Information

When we set about instrumenting a complete application, there are two main categories of entities that we are likely to want to learn more about. We may consider these

categories as *system-wide* resources and *application-specific* parameters themselves. The system-wide resources relate to the computing *platforms* on which our application components run and the network and associated networking devices which connect platforms. Chapter 2 has already defined the platform as the combination of operating system and computer hardware, including networking capabilities.

Although system-wide resources are essential to a complete understanding of a distributed system, they are not the main concern of this thesis. This thesis is more concerned with measuring and monitoring the application-level components and the distributed middleware infrastructure. However, the problem is that to measure and monitor the latter, we are likely to need some information regarding the computing platforms.

For this reason, the next section consider some 'pragmatic' techniques that may be used to access information relating to Java-based platforms. We do not explore the use of these techniques further in this chapter, but they will be revisited in chapter 8 (concerned with the implementation) and again in chapter 9, which considers instrumentation case studies.

7.1.1 Accessing System-wide Resources

Certain system information, relating to host computers, operating system and virtual machine environment may be obtained using language specific, or middleware specific utilities. For example, Java provides the `java.lang.System` class that provides a `getProperties` method that provides information about the Java Virtual Machine (JVM) environment and operating system. Java also provides a useful `java.lang.Runtime` class, which provides a `getRuntime` method that can be used to go beyond the JVM level and access the operating system itself. In particular, the `Runtime` class provides capabilities for determining the total memory and free memory of a host and the heap size of a JVM. Java and Java RMI are often used as the base development language for developing distributed systems in conjunction with Jini middleware technology. The Java RMI/Jini combination makes it possible to access system resources on a range of networked devices, including servers, workstations, printers and other small foot-print devices not usually found on a network. Java RMI's `RemoteServer` class provides a

`getClientHost` method that can be used to capture the address of the client executing a remote call on a server.

The Java Management Extension [24] (considered previously in chapter 4) may be used to access information through existing network management protocols. JMX provides an agent-based API that allows management services to be used within distributed Java-based applications. In particular, JMX provides integration with existing management technologies such as SNMP and future technologies such as the Web Based Enterprise Management (WBEM) standard, which is a relatively new management protocol for use in Storage Area Networks (SANS). Finally, whilst the above techniques provide techniques for accessing system-wide resources they do not necessarily provide direct capabilities to record or log information. Such logging may be carried out through general-purpose third-party logging software, such as the Jakarta *log4j* project [82], which provides general purpose logging utilities for Java based applications.

Having considered the techniques used to access system-wide resources, we may move on to consider our main concern, namely the access of application-specific properties, which manifest as the *structural* and *behavioural* properties of our application's components.

7.1.2 Accessing Component Structural and Behavioural Properties

When we instrument the components of an application, we are interested in determining the *structure* and *behaviour* of the components. Before we proceed, we must be clear about what structure and behaviour refer to: we may regard a component at runtime as a collection of objects, grouped together to provide some prescribed functionality. These objects are instances of the classes that constitute the structure of the component. The behaviour of a component is a runtime characteristic that is defined in terms of the collective states of its objects and the interactions between its objects and interactions with the outside world (i.e. interactions in the broader context of an application). The state of a single object at runtime is defined by its attributes and the runtime parameters of its methods during their invocation.

We may determine the structure using the *introspection* capabilities provided by modern object-oriented programming language. Introspection is the technique through which we

may "look into" a component to determine its structure, based on its attributes, method signatures, constructors, inheritance hierarchy etc. Java has a well-developed introspection library: for example, Java provides the `java.lang.Class` method, which can be used to access a variety of parameters associated with a Java class (class name, class attributes, class methods etc.). Similar structural parameters may be accessed in CORBA-based applications, through CORBA's IDL utilities and the CORBA *Interface Repository*, which is based on *Abstract Syntax Trees* (*ASTs*), as described in [96].

Java also provides a *reflection* API through which can be used to introspect classes and also acquire behaviour of the objects that implement classes. The reflection API, which was touched on in chapter 3, is based on the principle of computational reflection, which is described further in [97]. Essentially, reflection provides capabilities through which we may look into or *peek* at a components implementation and runtime behaviour. Java's reflection API allows us to reflect on runtime objects and access the attributes and any inner classes, which define the object's state. The reflection API also allows us to access the methods and their associated parameters and return types, which define state transitions and behaviour. CORBA provides *interceptors*, which can be used to provide reflection like capabilities. These capabilities are considered in [43] which describes experiences in using interceptors to implement reflection. As considered in chapter 4, the work of [8] describes how reflection and CORBA interceptors may be used to peek into CORBA objects at runtime.

Whilst we may use introspection and reflection to determine a components structure and behaviour, we often also need to learn more about a component's interaction with other components and more particularly the components on which it depends. Chapters 2 and 5 have previously considered to notion of dynamic dependencies between components. Chapter 5 described how dependencies are determined from the *bindings* associated with a component. Chapter 5 also described how a binding occurs when one component (the dependent) has downloaded a copy of the proxy of some other component (the independent) with a view to invoking the methods of the independent component. Reflection could be used to determine component bindings, but it would prove difficult to identify all dependencies, beyond the immediate dependencies. This is so because the immediate independent components may have dependencies themselves and to obtain a

complete representation of dependencies, we must visit these secondary dependencies. The alternative approach we use is based on the service dependency work conducted by [7] and makes use of *Administrable* and *Dependent* interfaces.

7.1.3 Administrable and Dependent Interfaces

Jini applications (considered further in chapter 8) consist of Java components, referred to as Jini services, which communicate with each other through proxies using the RMI protocol. Proxies are moved around in a Jini system and clients may download server proxies so that they may communicate with the server via RMI calls on the methods that its proxy specifies. Jini middleware technology provides an `Administrable` interface through which its own core services and programmer-defined application services may be administered. If a service implements the `Administrable` interface then it must implement a `getAdmin` method, which returns an administration object that implements whatever administration interfaces are appropriate for the particular service. Jini itself provides several administration interfaces for its own core services, but the main intention of the `Administrable` interface is to provide a *hook* onto which application programmers may attach their own administration interfaces.

The instrumentation architecture uses the `Administrable` interface in conjunction with a `Dependent` interface (Figure 7.1). The `Dependent` interface declares `getDeclaringClass` and `getBindings` methods and a `ServiceAdmin` class (Figure 7.1) implements the `Dependent` interface. Any application component, which is likely to depend on other components must implement the `Administrable` interface and therefore must implement the `getAdmin` method. The `getAdmin` method is required to return a `ServiceAdmin` object.

As we shall see shortly, instrumentation services that are designed to determine dependencies need only invoke the components `getAdmin` object to access its bindings from the `ServiceAdmin` administration object that is returned. By adopting this approach, we have introduced a compromise to instrumentation being unobtrusive. The need for application programmers to implement the `Administrable` interface and return a `ServiceAdmin` object does raise two drawbacks: first, programmers are required to expend extra effort and second, the extra code does intrude on their own application

classes. However, the extra effort, which is relatively small, does provide a direct and consistent means for dealing with the complexities of dependencies and consequently, the compromise is thought to be justified.

```
public interface Dependent {
    public Class getDeclaringClass();
    public Object[] getBindings();
}

public class ServiceAdmin implements Dependent {
    public Object obj = null;
    public Object[] bindings = null;

    public ServiceAdmin(Object obj) {
        this.obj = obj;
    }

    public Class getDeclaringClass() {
        return this.obj.getClass();
    }

    public Object[] getBindings() {
        return bindings;
    }
}
```

Figure 7.1: dependent interface and service admin object

The use of the `Administrable` and `Dependent` interfaces and the combination of the `ServiceAdmin` class and the `getAdmin` method will be considered further in chapter 8.

7.2 Architectural Models

This section develops the architectural models that elaborate the classification of instrumentation services of chapter 5 to represent measurement and monitoring functionality. Before we embark on the development of the UML models, we illustrate the architecture informally through Figure 7.2.

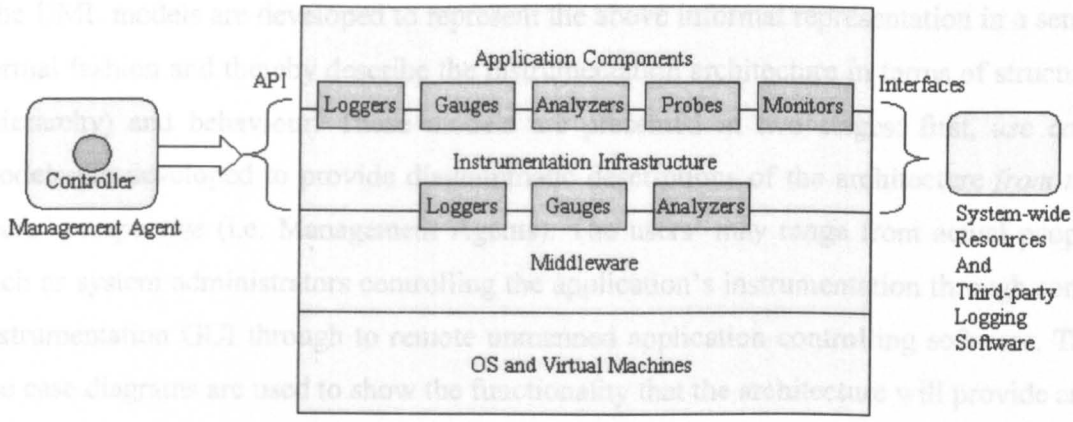


Figure 7.2: instrumentation layer

This figure shows the instrumentation infrastructure sandwiched between middleware and application component layers. The instruments, Loggers, Gauges, Analyzers, Probes and Monitors sit on the outer surface of this infrastructure, straddling the Middleware and Application Component layers. All the instruments sit between the infrastructure and the Application Components whereas only Loggers, Gauges and Analyzers sit between the infrastructure and Middleware, since these are the only three instruments designed for measuring and monitoring middleware. To the left there is a Management Agent, containing a Controller, which may access instrumentation services via an API. To the right there are utilities that provide access to system-wide resources and third-party logging software applications. Instrumentation services themselves may require access to these resources or may use the services of general purpose logging software to log or record system-wide resources.

In this chapter, we are concerned with the development of the instrumentation infrastructure, the five instantiable instruments and the instrumentation API. As we shall see shortly, the instrumentation infrastructure is made up of a hierarchy of protected support classes that are exclusive to the architecture and may not be instantiated by Management Agents. The five instruments are represented as public classes that may be instantiated and used via their APIs by Management Agents. The API is represented as the various constructors and methods that the five instruments provide. The interfaces, used to access system resources and third-party logging software, are not considered in this chapter, but they will be considered further in chapters 8 and 9.

The UML models are developed to represent the above informal representation in a semi-formal fashion and thereby describe the instrumentation architecture in terms of structure (hierarchy) and behaviour. These models are presented in two stages: first, *use case* models are developed to provide diagrammatic descriptions of the architecture *from the users' perspective* (i.e. Management Agents). The users' may range from actual people such as system administrators controlling the application's instrumentation through some instrumentation GUI through to remote unmanned application controlling software. The use case diagrams are used to show the functionality that the architecture will provide and to show which users will communicate with the system in some way when it provides that functionality.

The second stage of models is based on class diagrams and sequence diagrams, which capture the structure and interactions respectively required for measuring and monitoring application components. The link between the two stages of modelling is that classes and their attributes and methods should become apparent from the use case models. In other words, we conduct the use case modelling from the users' perspective to uncover or reveal the classes, attributes and methods that must be represented in the classes that form the architecture. It is true that we already have a pretty good idea regarding the classes, which emerged from the instrumentation classification of chapter 5, but the same cannot be said for the attributes and methods of the classes.

7.2.1 Use Case Models

As shown in Figure 7.3, the use case models emerge from the overall coarse-grained package diagram.

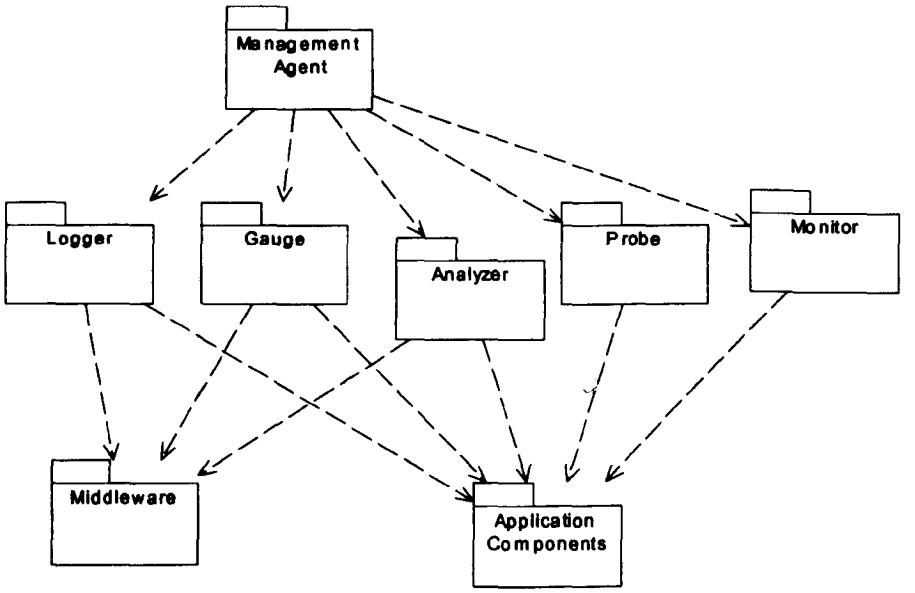


Figure 7.3: system package diagram

The package diagram shows the five instrument packages and their associations with Management Agent, Middleware and Application Component packages. By looking in to each of these packages we may develop the use case models that represent the main entities (stick-like actors) and the interactions between these entities through use cases (elliptical processes).

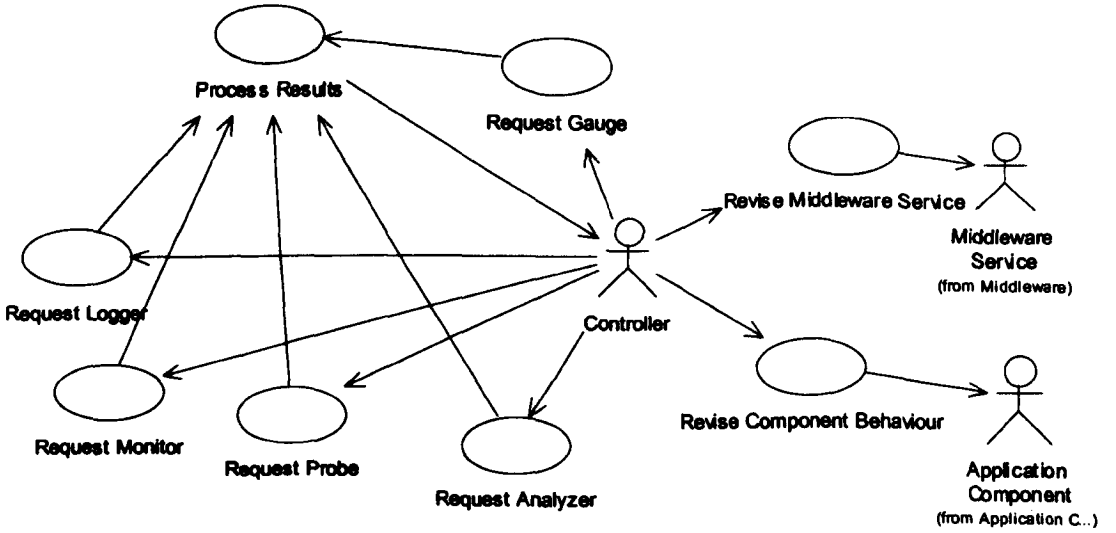


Figure 7.4: management agent use cases

The first use case model is that of the Management Agent, shown in Figure 7.4. This model shows the Controller playing a central role in coordinating the activities associated

with the management of an application. The Controller may request services from any of the five instruments. The Controller may then process the results from the instrumentation services and use this information to revise either the middleware or application components ("Process Results" use case). As they stand, use cases such as "Request Logger" tell us little about the activities involved in this process, so we may expand each of the instrument request processes as activity diagrams. The activity diagrams shown below in Figures 7.5 to 7.9 represent the activities that take place within the request processes, just as if we were to "lift the lid" and "look into" each of these processes.

Because the request processes occur within the Management Agent package, the activity diagrams of Figures 7.5 to 7.9 represent activities undertaken by the Management Agent and not the instrument in question. The diagrams do however represent the states of the instruments whilst the various activities are taking place. Within the diagrams, the boxes with rounded ends represent activities and states are represented as boxes with rounded corners.

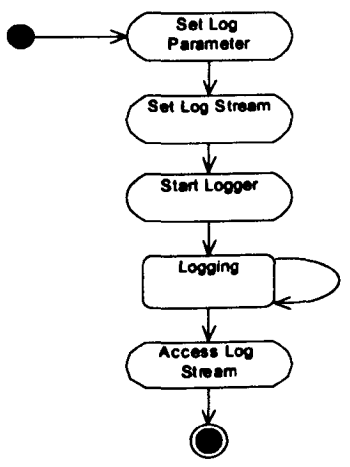


Figure 7.5: logging activity diagram

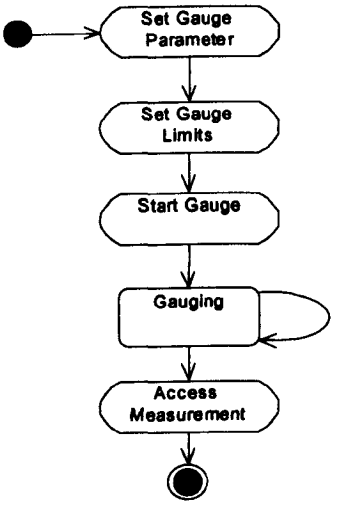


Figure 7.6: gauge activity diagram

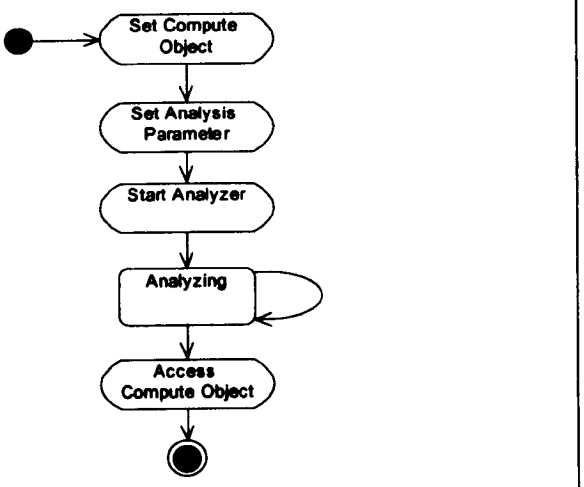


Figure 7.7: analyzer activity diagram

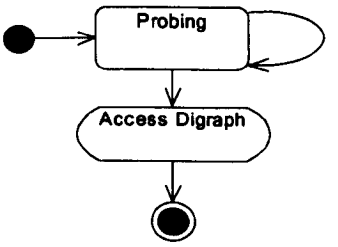


Figure 7.8: probe activity diagram

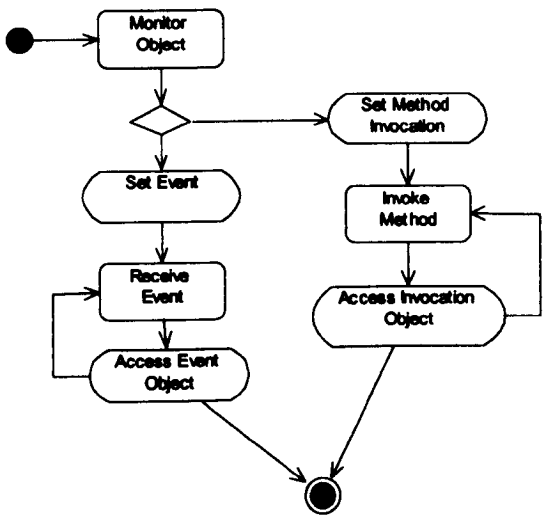


Figure 7.9: monitor activity diagram

The first three activity diagrams of Figures 7.5 to 7.7 are similar and represent the Management Agent setting appropriate parameters before Loggers, Gauges and Analyzers are started. The instruments then enter their appropriate states of Logging, Gauging and Analyzing and when these states end the Management Agent may access the results. The request probe activity diagram is the simplest of all the activity diagrams since the probe does most of the work and as we shall see shortly, a probe gets its parameters directly from application components as component bindings. The request monitor activity diagram differs in that it contains two optional routes – one for event monitoring and one for method invocation monitoring.

Notice that none of the diagrams include an attachment operation through which an instrument is attached to the component that it measures or monitors. Of course, this is a necessary operation, which must take place before an instrument can perform any form of measurement or monitoring. However, this is regarded as one of the basic instrument operations that were considered in chapter 6. The semantics governing these operations have already been covered in chapter 6 and, with the exception of invoke, their semantics are not considered further in this chapter. The basic operations will however be represented as methods in the class diagrams to follow later.

The activities relating to processes "Process Results", "Revise Component Behaviour" and "Revise Middleware Service" have not been elaborated further, through activity diagrams, as they are not relevant to the development of the instrumentation architecture.

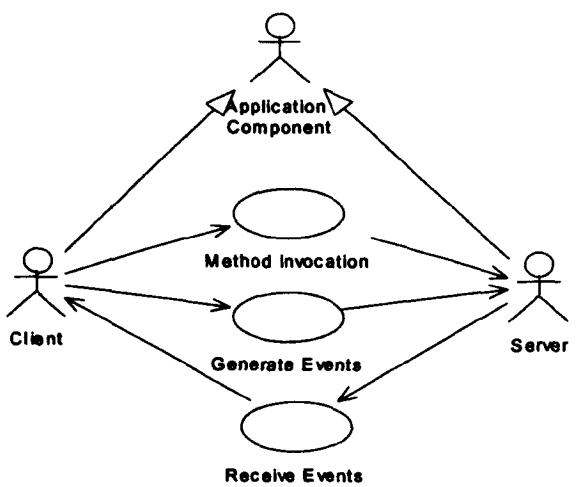


Figure 7.10: application components use cases

The Application Component package expands to the use case model shown in Figure 7.10. This model shows that an application component may be a client or a server and the two may engage in method invocations (client invokes methods on server) or they may transmit events amongst one another. Of course, application components do a lot more, but from the point of view of our model, method invocations and event transmissions are the main concern.

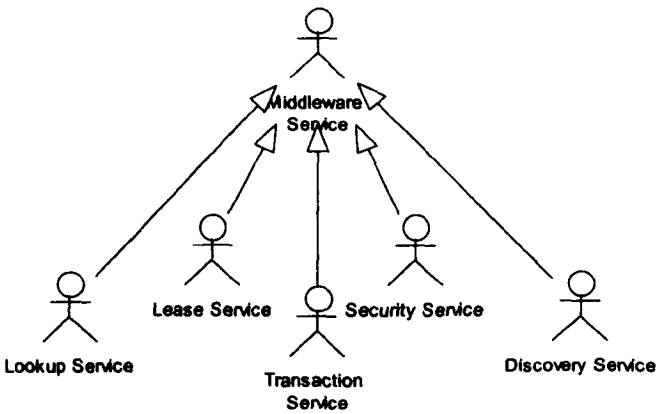


Figure 7.11: middleware services hierarchy

The Middleware package expands to the use case model shown in Figure 7.11. This model shows a simple representation of a middleware services as one of several core services found in most current middleware technologies. Again, although this model omits the vast amount of activity undertaken by middleware services, it is sufficient for the development of the overall instrumentation model (i.e. provides a suitable abstraction of middleware services).

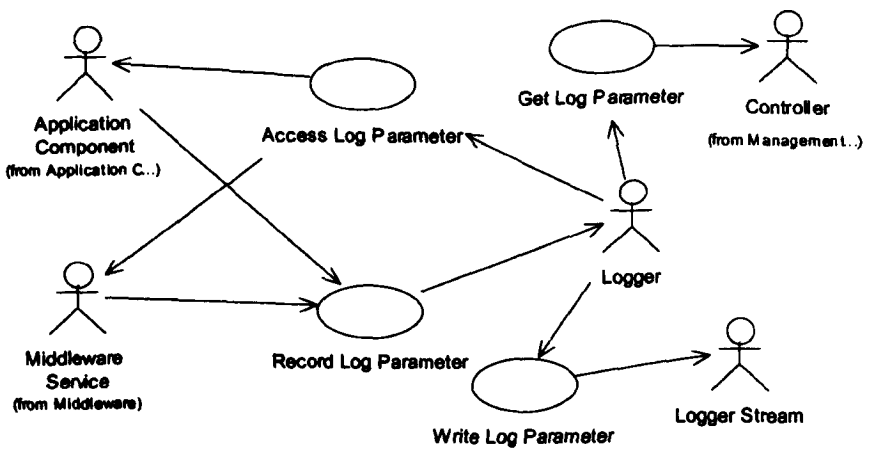


Figure 7.12: logger use cases

The Logger package expands to the use case model shown in Figure 7.12. This model shows the interactions between a Logger, an Application Component or Middleware Service, a Controller and a Logger Stream. The Logger must first get the parameter to be logged from the Controller and then access this parameter from either an Application Component or Middleware Service. The Logger then records the parameter and writes it to a stream.

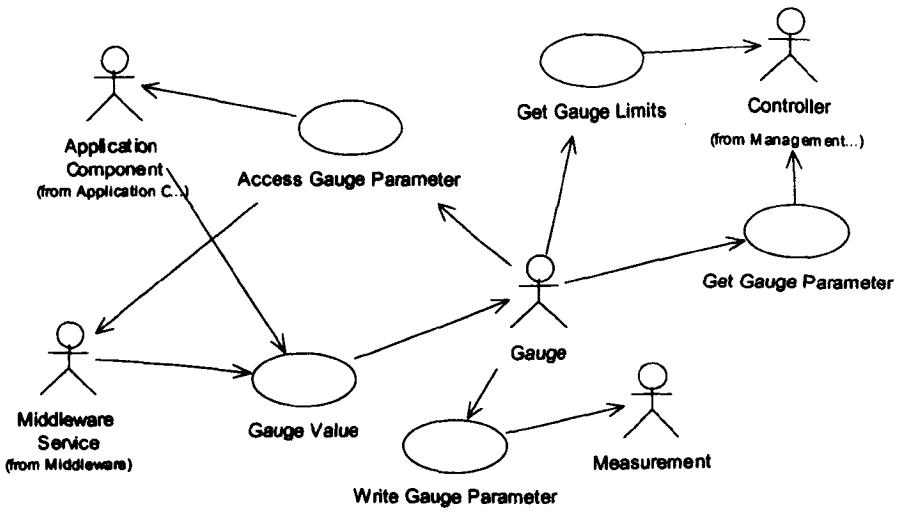


Figure 7.13: gauge use cases

The Gauge package expands to the use case model shown in Figure 7.13. This model is similar to the Logger model of Figure 7.12, except that the gauge must also get the gauging limits from the Controller. The Gauge does not write its data to a stream, but to a measurement object that may be accessed by the Controller when it is processing the Gauge's results.

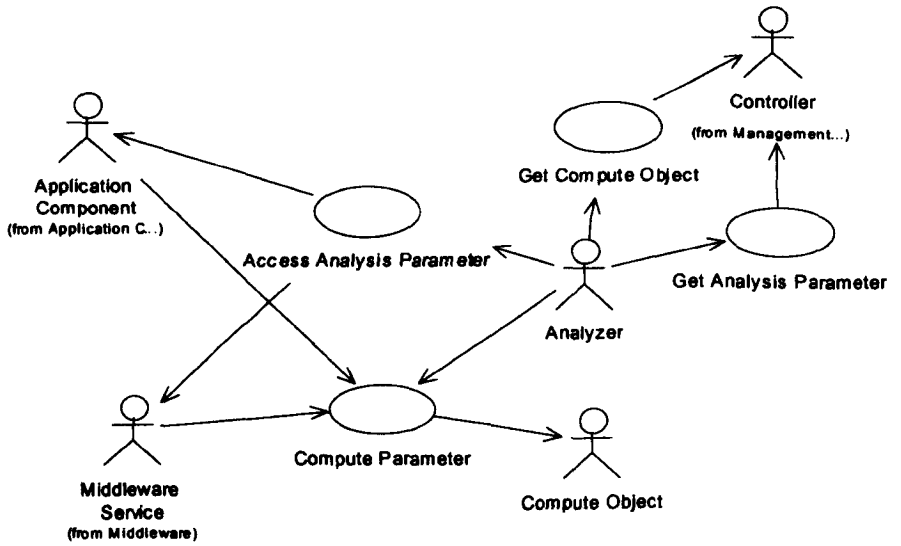


Figure 7.14: analyzer use cases

The Analyzer package expands to the use case model shown in Figure 7.14. This model is similar to the Logger and Gauge package, except that the analyzer must also get the Compute Object, which will be used to perform the analysis and store the result.

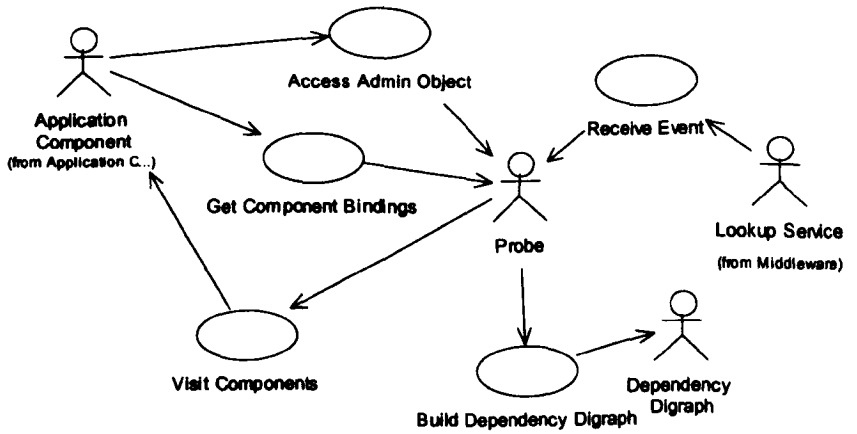


Figure 7.15: probe use cases

The Probe package expands to the use case model shown in Figure 7.15. The main difference in this model is that the Controller is not present, but a middleware Lookup Service is. This is so because the Probe will determine the dependencies of the component to which it is attached and the Probe needs to be made aware of any changes in its component's dependencies via the Lookup Service with which the component is registered. The Probe determines the dependencies of its component by determining the

component's bindings. After acquiring this set of direct dependencies, the probe must recursively visit each of the other independent components to see if they themselves have any dependencies of their own before the probe can build the complete dependency digraph.

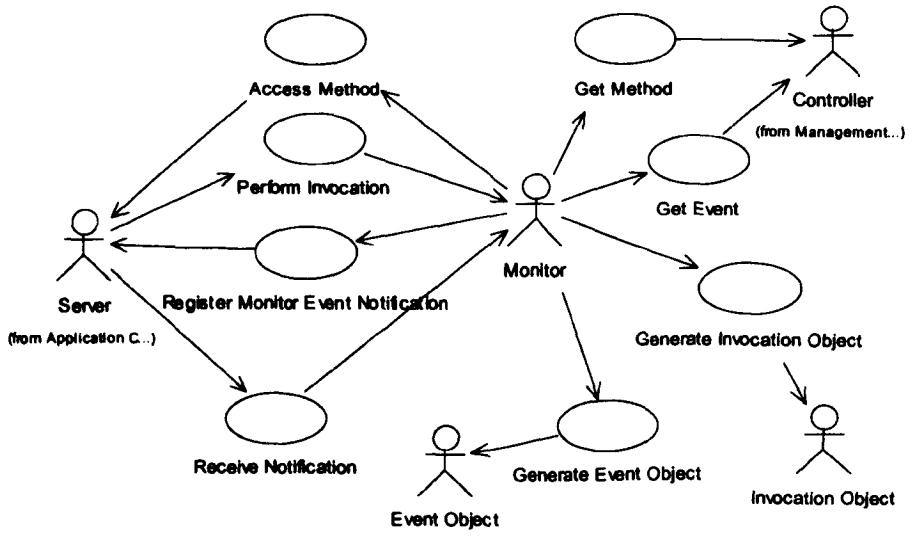


Figure 7.16: monitor use cases

The Monitor package expands to the use case model shown in Figure 7.16. The monitor model represents the interactions for event monitoring and method invocation monitoring, although the two different tasks will be conducted by different types of monitor. The monitor begins by getting either an event or a method from the Controller. If the monitor is an event type, it must register in order to receive notifications of any such events from the Application Component. After notifications are received the monitor then repackages the event in the form of an event object. In contrast, method invocation monitors must first access the appropriate method before they perform the invocation on behalf of a client component. The method, its parameters and any results from the invocation are then repackaged as an invocation object.

As mentioned previously, we already have a pretty good idea regarding the classes, from the instrumentation classification of chapter 5, but the use case models have revealed additional, less obvious classes. We have also revealed the activities that take place between Management Agent Controllers and instrumentation services. These activities will assist in developing the API that instrumentation services must provide to

Management Agents via their constructors and methods. For this chapter we are only concerned with developing the basic shape of the API so that we may craft it further and incorporate middleware utilities in chapter 8, which deals with the implementation of instrumentation services.

7.2.2 Class and Sequence Diagrams

The classes that constitute the architecture are based on the instrumentation classification developed in chapter 5. We may recall the classification hierarchy (Figure 5.1), which is repeated in Figure 7.17 to assist the explanation of the instrumentation architecture classes. The following class diagrams and associated sequence diagrams provide a semi-formal description of the architecture that may be used as the basis of an implementation. The class diagrams represent structure and the sequence diagrams represent the behaviour required of the architecture. The design starts with the root of the hierarchy tree and works through the various infrastructure classes towards the concrete instantiable instrumentation services.

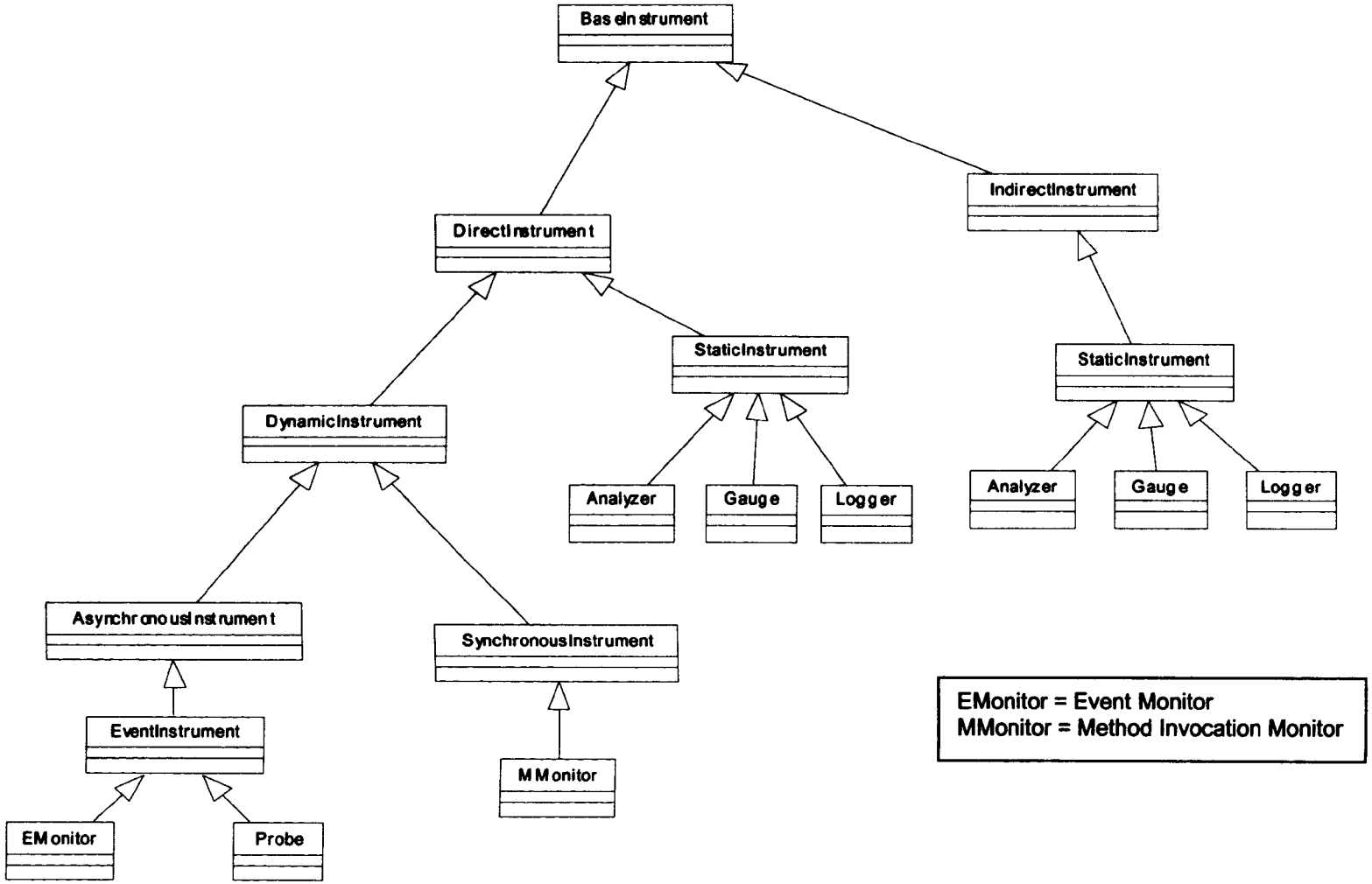


Figure 7.17: instrumentation hierarchy

7.2.3 BaseInstrument Class

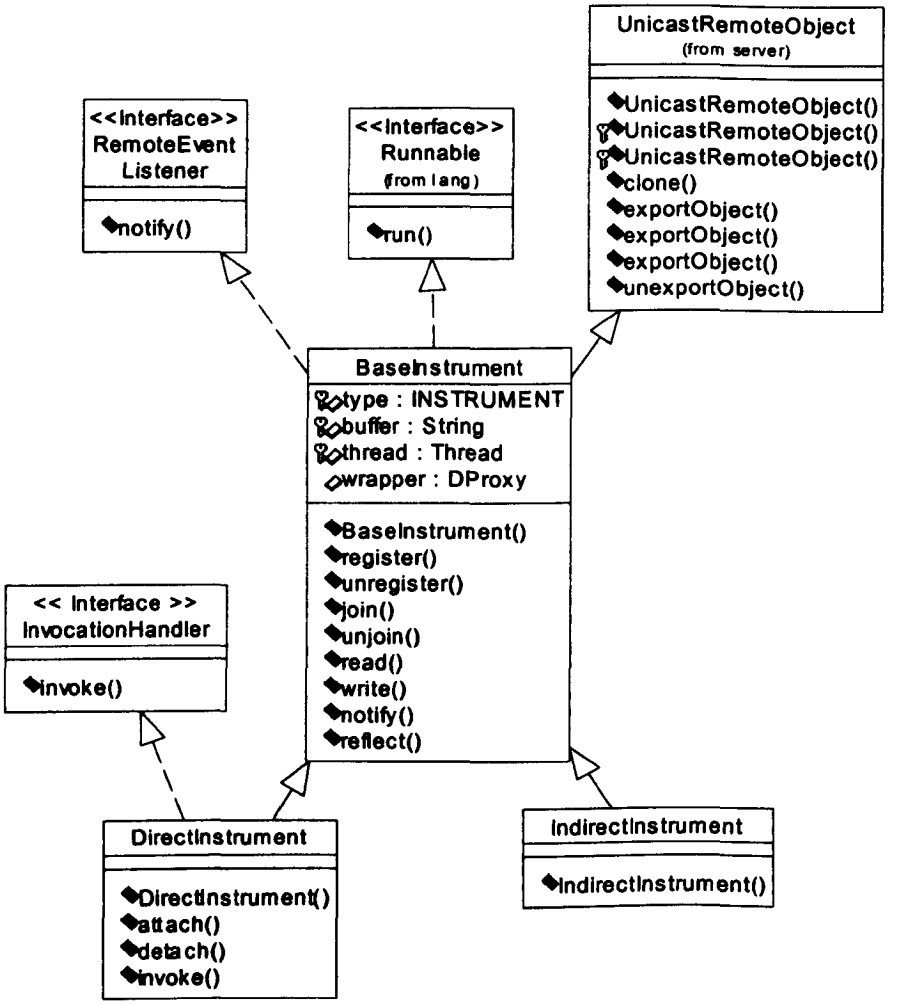


Figure 7.18: BaseInstrument class

chapter 8 considers the implementation of the architecture using Java and with this in mind, several of the class diagrams refer to classes supported by versions of J2SE from v1.3 upwards. The `BaseInstrument` class, shown in Figure 7.18, is at the root of the instrumentation class hierarchy. `BaseInstrument` subclasses the `UnicastRemoteObject` class from the `java.rmi.server` package to allow Java RMI communication. `BaseInstrument` implements the `Runnable` interface from the `java.lang` package so that instruments may run as Java threads. `BaseInstrument` also implements the `RemoteEventListener` interface provided by Jini middleware. This

requires that `BaseInstrument` implements a `notify` method, which is in fact an implementation of the basic instrument operation Notify that was considered in chapters 5 and 6. Through the `notify` method the current instrument may notify other instruments, to which it is joined, that its state has changed. `BaseInstrument` contains the following attributes:

- `type` – the type of instrument (logger, gauge, analyzer, probe or monitor).
- `buffer` – a message string that is used for communications with other instruments
- `thread` – a Java thread in which the instrument may run, when it is required to run for a period of time until it is terminated.
- `wrapper` – a dynamic proxy through which instruments may be attached to application components.

`BaseInstrument` declares a constructor and methods that cover seven of the basic instrument operations, considered previously in chapters 5 and 6, namely `register`, `unregister`, `join`, `unjoin`, `read`, `write` and `notify`. The `reflect` method is a generic method that uses Java's reflection API to access a specific parameter of an object at runtime, where the object is an instance of some specific class. The parameter may typically be an attribute, a method, a parameter or return value of a method invocation, an inner class, or an inherited class. The `reflect` method is a private method that may only be invoked by other classes within the instrumentation hierarchy.

`DirectInstrument` and `IndirectInstrument` are the two subclasses of `BaseInstrument`. These classes distinguish between instruments that are directly attached to application components and those that are indirectly attached, generally via a direct instrument. The `DirectInstrument` class provides the capability to attach instruments to application components, via the dynamic proxy `wrapper` attribute. To provide this dynamic attachment capability, `DirectInstrument` implements the `InvocationHandler` interface from Java's reflection API. The `DirectInstrument` class also implements the final three basic instrument operations of Attach, Detach and Invoke, via `attach`, `detach` and `invoke` methods respectively.

It is at this stage that we may see the benefit of the separating instrumentation functional and operational requirement. Between them, `BaseInstrument` and `DirectInstrument` declare methods relating to the ten basic instrument operations of: Register, Unregister, Attach, Detach, Join, Unjoin, Read, Write, Invoke and Notify. The states and axioms governing these operations have already been specified in chapter 6, so we need not concern ourselves with these methods for the remainder of the class hierarchy model.

However, the Invoke and Notify operations are given some further explanation as these two operations may be overridden by method and event monitor instruments respectively. The basic `invoke` method is used by `DirectInstrument` to perform method invocations on a server on behalf of a client, as specified in the *Invoke* and *ClientServer* schemas of chapter 6. The basic `notify` method is used by `BaseInstrument` to inform other instruments that its state has changed in some way, as specified in the *Notify* schema of chapter 6. Later in the section, we shall see how these methods are overridden to perform more specific monitoring activities.

7.2.4 Static Instrumentation Services: Logger, Gauge and Analyzer

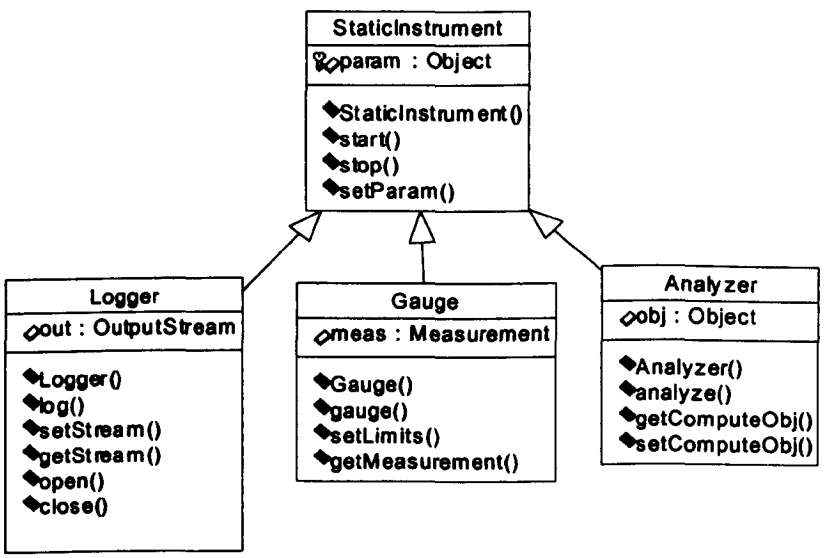


Figure 7.19: StaticInstrument class hierarchy – Logger, Gauge and Analyzer

The StaticInstrument class, shown in Figure 7.19, may subclass either DirectInstrument or IndirectInstrument. StaticInstrument contains the single attribute param, which is the parameter that is to be logged, analyzed or gauged. The three subclasses of StaticInstrument are the three static instruments Logger, Gauge and Analyzer, which are instantiable classes. StaticInstrument contains start and stop methods to start and stop logger, gauge and analyzer threads respectively. The setParam method is used by management agents to set the parameter of interest for the instrument.

The Logger class contains the out attribute, which is a Java OutputStream to which information is to be written. Logger contains a log method, which may run as a thread to log the parameter over a period of time. Logger also contains open and close methods to open and close the logger's stream respectively and getStream and setStream methods, used by management agents to set and access the logger's stream respectively.

The Gauge class contains the meas attribute, which is an object of type Measure. Gauge contains a gauge method, which may run as a thread to gauge the parameter over a period of time. Gauge also contains a setLimits method, used by management agents to set the gauge limits and a getMeasurement, used by management agents to access the gauged measurement.

The Analyzer class contains the obj attribute, which is a java.lang.Object that represents the compute object used in the computational aspect of the analysis. Analyzer contains an analyze method, which may run as a thread to analyze the parameter over a period of time. Analyzer also contains getComputeObj and setComputeObj methods, used by management agents to access and set the computed object respectively.

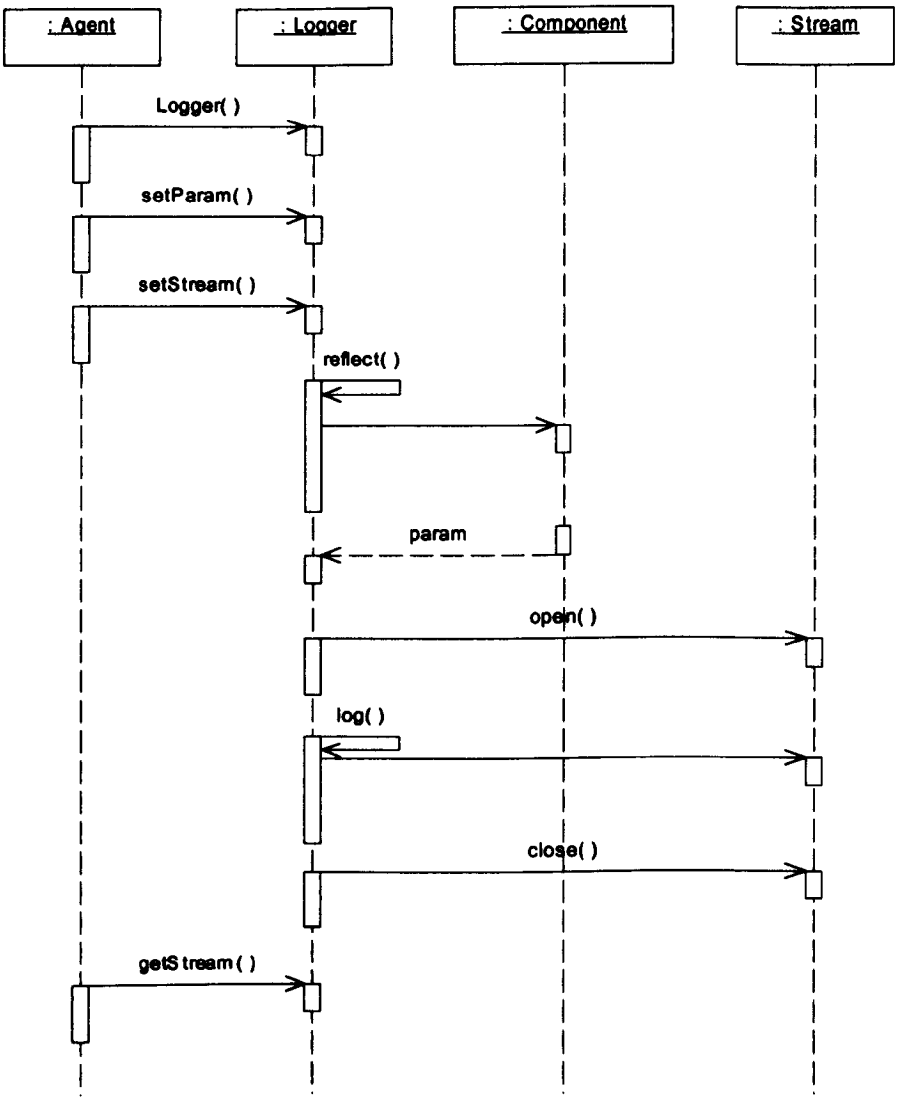


Figure 7.20: Logger sequence diagram

The sequence diagram of Figure 7.20 represents the time sequence of interactions between the objects involved in a logging activity, which are represented as boxes arranged horizontally in the diagram. The vertical lines or "swim-lanes" represent the *lifeline* of each object and time is measured vertically downwards. Horizontal lines are used to represent the activation of communication messages such as object creation, via constructors and method invocations and their results.

The logger sequence begins when a management agent creates a `Logger` object, via its constructor `Logger()`.The agent then invokes the logger's `setParam` and `setStream` methods to set the parameter to be logged and the stream to which results are to be

logged respectively. Typically, in a Java implementation, these invocations would pass Java references of objects to the logger. The logger then invokes its `reflect` method to access the specified component parameter to be logged. The logger then opens the stream and its `log` method is then invoked to record or write the parameter to a stream. The `log` method may either be invoked in a single pass to record one single value of the parameter. Alternatively, as shown in Figure 7.20, it may be invoked within a thread to repeatedly record the value based on the granularity that is set for the thread. When the log method returns the logger closes the stream and the management agent may access the stream using the `getStream` method.

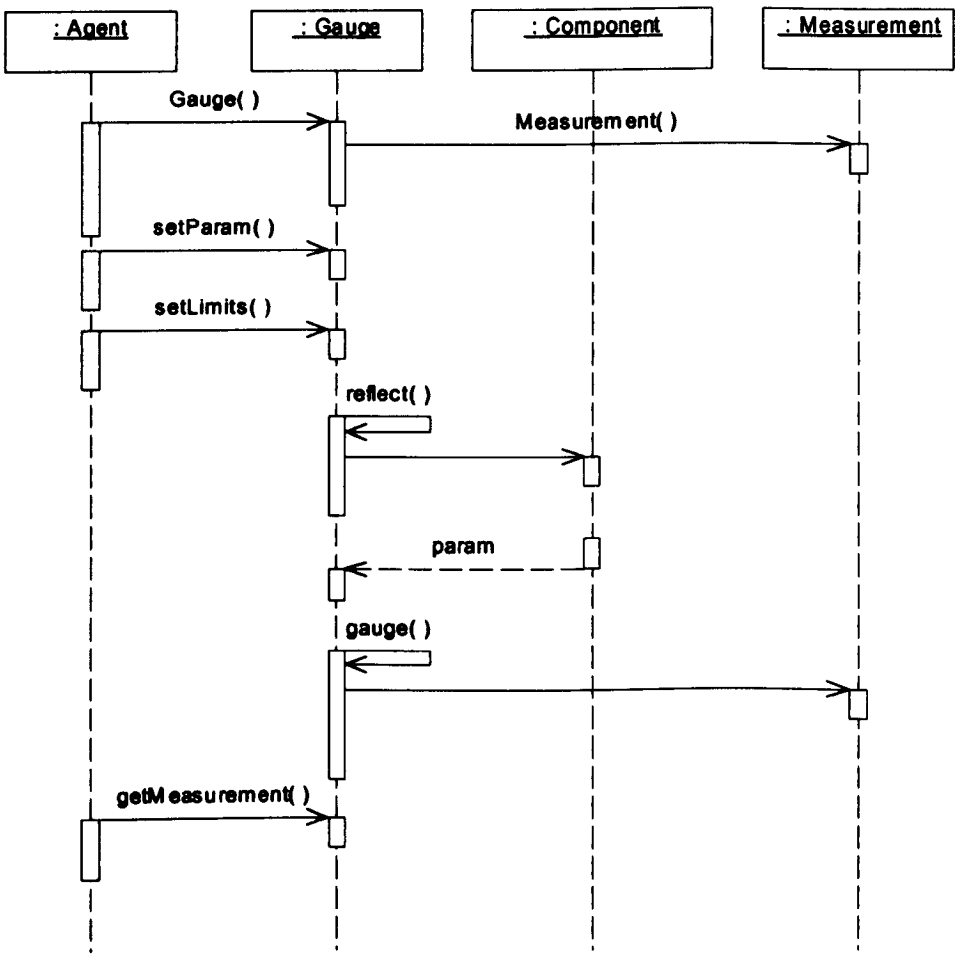


Figure 7.21: Gauge sequence diagram

The gauge sequence diagram is shown in Figure 7.21. The sequence for the gauge is similar to that of the logger with a few small exceptions: when the gauge constructor is

invoked, by a management agent, it goes on to call a measurement constructor to create a measurement object. The agent then invokes the `setParam` and `setLimits` methods to set the parameter to be gauged and the gauging limits respectively. As for the logger, the `gauge` method may be invoked in a single pass fashion, or as shown in the diagram, within a thread. When `gauge` returns, the management agent may access the measurement via the `getMeasurement` method.

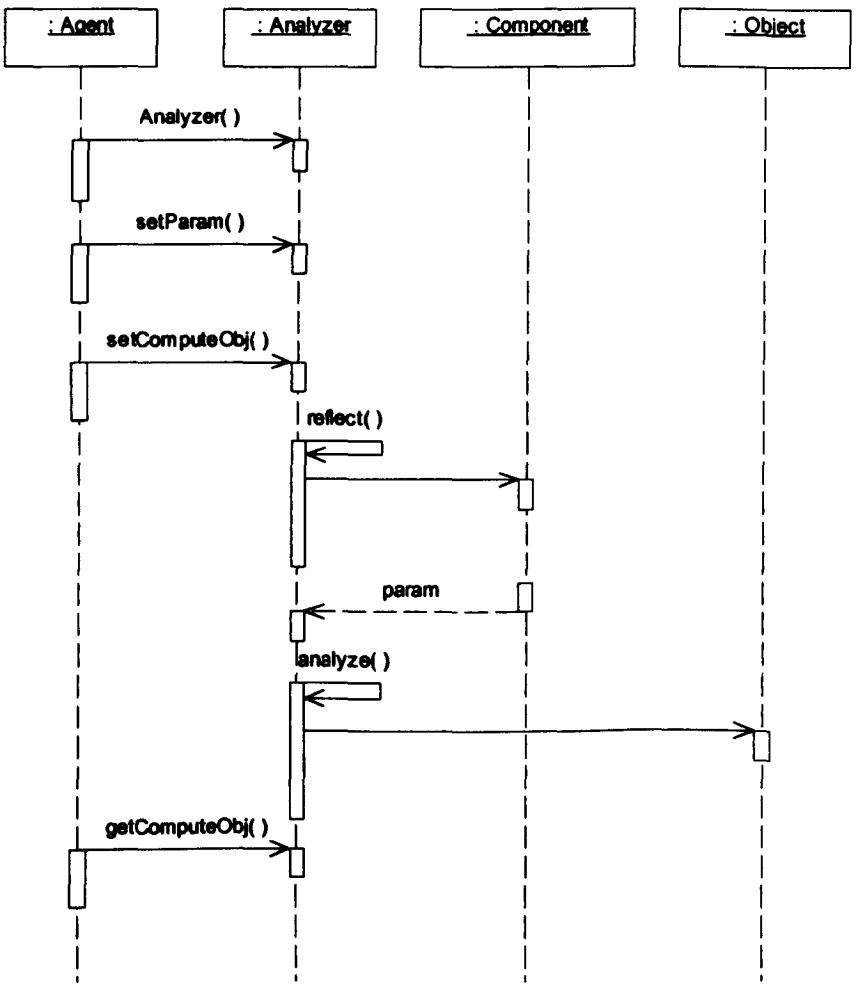


Figure 7.22: Analyzer sequence diagram

The analyzer sequence diagram is shown in Figure 7.22. The sequence for the analyzer is similar to those of logger and gauge except that a reference to a computational object that will perform the analysis is passed to the analyzer via the `setComputeObj` method.

When the analysis is complete, the management agent may access the computational object via the `getComputeObj` method.

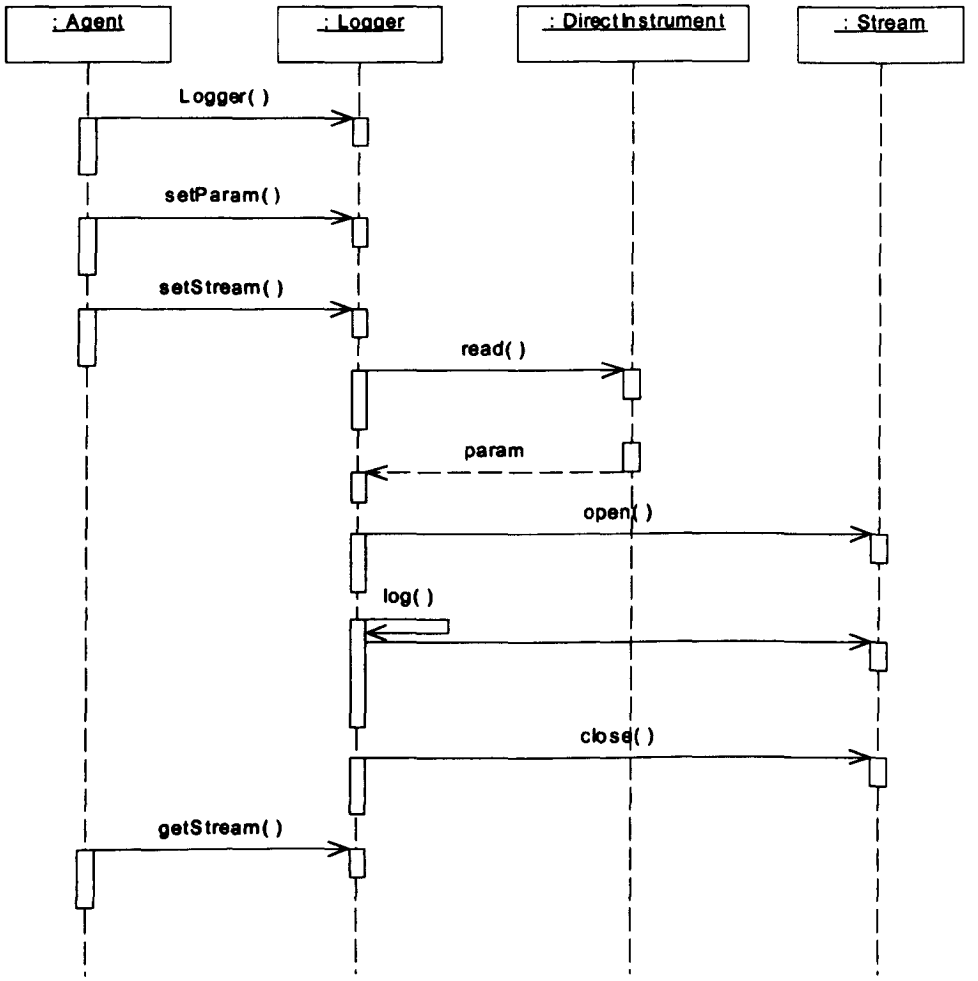


Figure 7.23: indirect Logger sequence diagram

The diagrams of Figures 7.20 to 7.22 represent the sequences for the *direct* static instruments of logger, gauge and analyzer. However, these static instruments may also be *indirect* if the `StaticInstrument` class subclasses `IndirectInstrument`. Such indirect static instruments are not directly attached to an application component and they must use a direct instrument in order to access a component's parameters indirectly. Figure 7.23 shows the sequence diagram for such an indirect logger instrument. In Figure 7.23, the component is replaced with another instrument, which must subclass `DirectInstrument` (i.e. it may be a logger, gauge, analyzer, probe or monitor, but it must subclass `DirectInstrument`). Through this arrangement, the

indirect logger may use the `read` method of the `BaseInstrument` class to indirectly access a component's parameters.

All the previous sequence diagrams show static instruments recording and measuring parameters for application components, but the static instruments may equally be applied to record or measure middleware service parameters. Figure 7.24 shows the sequence diagram of a logger recording a parameter associated with a middleware service. The sequence diagram is similar to those of Figures 7.20 to 7.22, except that instead of using the `reflect` method, the `getAdmin` method of the middleware service is invoked. This invocation returns an administration object from which the logging parameter may be accessed. Recall how Section 7.1.3 remarked that Jini provides several administration interfaces for its own core services. Jini's core middleware services implement these interfaces and by doing so, they implement `getAdmin` methods through which the associated administration objects may be accessed. Chapter 8 will consider the programmatic details relating to the invocation of `getAdmin` methods and the subsequent access of administration objects.

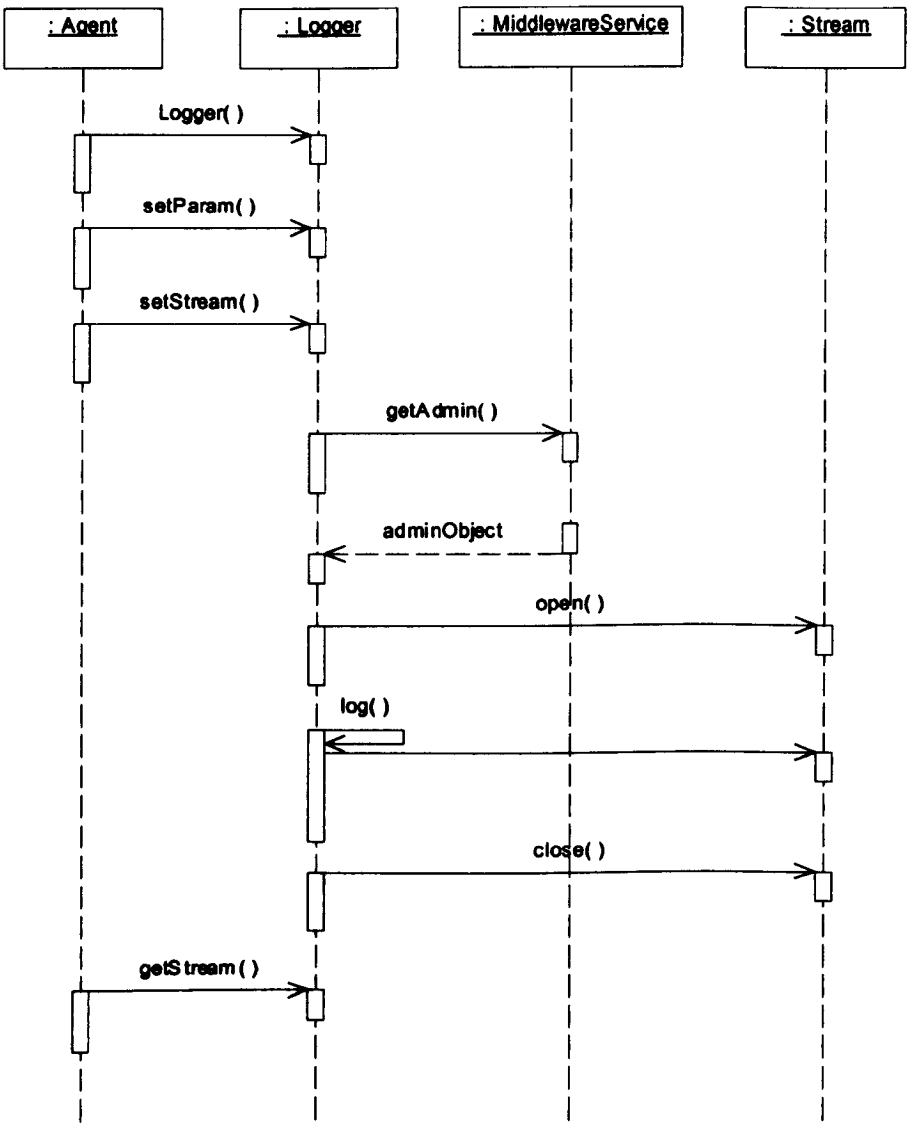


Figure 7.24: Middleware Logger sequence diagram

7.2.5 Dynamic Infrastructure Classes

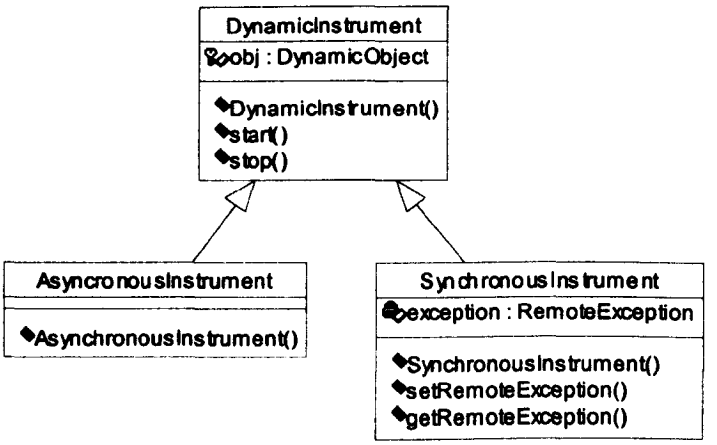


Figure 7.25: DynamicInstrument class hierarchy

The `DynamicInstrument` class, shown in Figure 7.25, is a subclass of `DirectInstrument`. `DynamicInstrument` contains the single attribute `obj`, which represents an object of class `DynamicObject`. The two subclasses of `DynamicInstrument` are `AsynchronousInstrument` and `SynchronousInstrument`. Asynchronous instruments are used for monitoring asynchronous parameters such as distributed events and dynamic dependencies via probe instruments. Synchronous instruments are used to monitor synchronous parameter such as RMI calls where the client must be prepared to wait to receive exceptions back from the server.

Like the `StaticInstrument` class, `DynamicInstrument` contains `start` and `stop` methods to start and stop the different types of dynamic instruments respectively. The `AsynchronousInstrument` subclass is a simple class, which consists of only a constructor. The `SynchronousInstrument` subclass contains a `RemoteException` attribute that may result if an RMI call should fail.

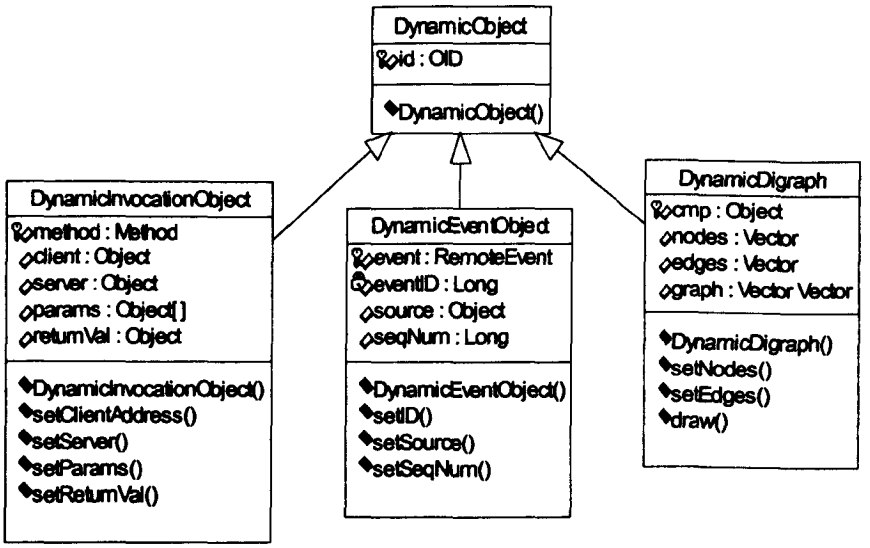


Figure 7.26: DynamicObject class hierarchy

The `DynamicObject` class, shown in Figure 7.26, is used in conjunction with the `DynamicInstrument` class. `DynamicObject` contains a single `id` attribute, which is an object id. (`OID`) (a typical instrumentation scenario, may contain many such distributed dynamic objects that are uniquely identified by their OID). The `DynamicObject` class has three subclasses, namely `DynamicInvocationObject`, `DynamicEventObject` and `DynamicDigraph` to represent the three different types of dynamic behaviour and characteristics associated with application components. `DynamicInvocationObject` and `DynamicEventObject` essentially *repackage* the dynamic behaviours of method invocations and events respectively to provide supplementary information relating to the behaviour. `DynamicDigraph` produces a new artefact, which is a graph of the dependencies associated with a particular application component.

The `DynamicInvocationObject` class contains an attribute that represents the method on which invocation is to take place and further attributes that represent: the server, the client's address, the invocation parameters and the invocation result (if any). All these attributes are of `java.lang.Object` type, except `params`, which is an array of `java.lang.Object (Object [ ] )`. `DynamicInvocationObject` also contains associated "set" methods that an invocation monitor instrument may use to set the appropriate parameters relating to the invocation.

The DynamicEventObject class contains attributes that represent the event's identifier, and the source object associated with the event. The seqNum attribute represents the value of the sequence number on the event kind that was current when the registration was granted, allowing comparison with the sequence number in any subsequent events. The event represented by DynamicEventObejct is of type RemoteEvent, which is a Jini middleware class that extends java.util.EventObject and the source of the event is of java.lang.Object type. DynamicEventObject also contains associated "set" methods that an event monitor instrument may use to set the appropriate parameters relating to the event.

The DynamicDigraph class has a cmp attribute, which represents the component for which the digraph is to be derived. The nodes and edges attributes represent the other components in the digraph and the relationships or connectivity between all the components respectively. The graph attribute is the graphical combination of the nodes and edges attributes The nodes attribute is a java.util.Vector in which each element is a reference of java.lang.Object type. The edges attribute is a java.util.Vector of Edge type, where Edge is represented as a pair of references of java.lang.Object types. The graph attribute is a Vector of Vectors, which is derived by combining the nodes and edges vectors. DynamicDigraph contains associated "set" methods that a probe instrument may use to set the appropriate parameters relating to the digraph. DynamicDigraph also contains a graph drawing method, draw, which does not physically draw a graph in the graphics sense, but assembles an in-memory representation of a graph based on its node and edge connectivity.

7.2.6 Asynchronous Instrumentation Services: Probe and Event Monitor

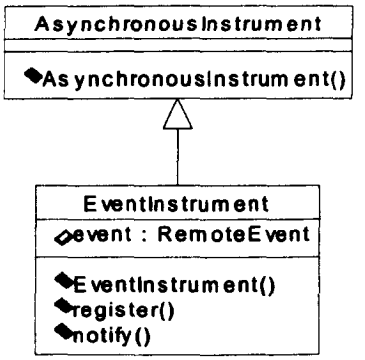


Figure 7.27: AsynchronousInstrument class hierarchy

As shown in Figure 7.27 the `EventInstrument` class subclasses `AsynchronousInstrument` in order to factor out the common event handling functionality required by the `Probe` and `EMonitor` instruments. The `EventInstrument` class contains an event attribute of type `RemoteEvent` to represent the event of interest. The `EventInstrument` class also contains a `register` method to register its interest in occurrences of the event of interest.

Note that `EventInstrument` overrides the `notify` method of `BaseInstrument`. The `notify` method in `BaseInstrument` is used by the current instrument to notify other instrument's, to which it is joined, that its state has changed. In `EventInstrument` the `notify` method still maintains this capability, but adds additional code so that the `EventInstrument` is made aware of occurrences of the remote event of interest (i.e. the overridden notify is invoked whenever the event occurs, but this is not necessarily communicated to other instruments to which the event instrument is joined).

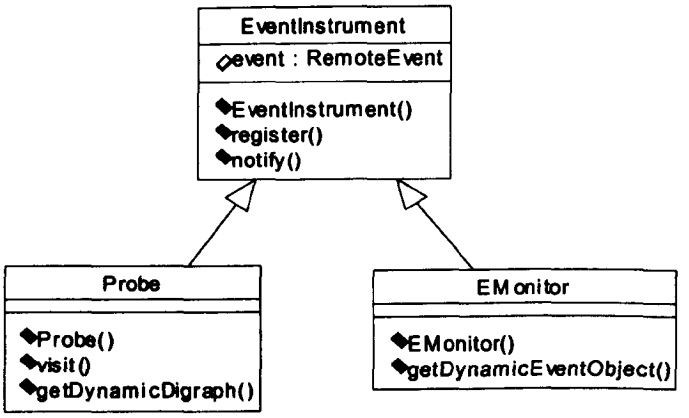


Figure 7.28: EventInstrument class hierarchy – Probe and EMonitor

The `EventInstrument` class has two subclasses, `Probe` and `EMonitor`, which are instantiable instrument classes in that management agents may directly create objects of these classes. The `Probe` class is used for deriving the dependency digraph for a particular component. It contains a `visit` method, which implements a *visitor* design pattern, [53], which allows the probe to recursively visit other application components and their lookup services to build up a complete picture of the particular application component's dependencies. This recursive descent and the visitor design pattern will be considered further in chapter 8. The `getDynamicDigraph` method may be invoked by a management agent to access the resulting dependency digraph after it has been drawn/redrawn (bearing in mind this digraph may change over a period of time). The `EMonitor` class is used to repackage an event of interest. `EMonitor` contains a `getDynamicEventObject`, which may be invoked by a management agent to access the repackaged event object.

7.2.7 Synchronous Instrumentation Services: Method Invocation Monitor

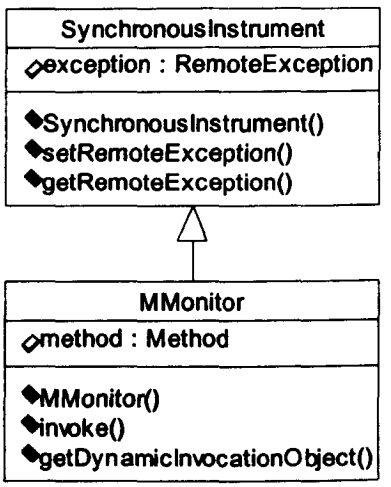


Figure 7.29: Synchronous class hierarchy - MMonitor

The `MMonitor` subclasses `SynchronousInstrument` to provide a facility for dealing with synchronous RMI calls, which have the potential to throw a `RemoteExcpetion`. The `MMonitor` class contains a `method` attribute of type `java.lang.reflect.Method` to represent the method of interest. The `MethodInstrument` class also contains an `invoke` method, which overrides the invoke method of `BaseInstrument` to add additional code, which captures the characteristics of the invocation.

Chapter 8 will describe the implementation of the `BaseInstrument` `invoke` method, which is based on Java's dynamic proxy facility (`java.lang.reflect.Proxy`). Chapter 8 will also describe how `MethodInstrument` overrides the invoke method so that the method invocation may be repackaged as a method invocation object. The `MethodInstrument` class has a single subclass, `MMonitor`, which is also an instantiable instrument class. `MMonitor` contains a `getDynamicInvocationObject`, which may be invoked by a management agent to access the repackaged method invocation object.

As for the instantiable static instrument classes, sequence diagrams may be used to represent the time sequence of interactions between the objects involved in the dynamic

instrument's activities. These sequences, which are similar to the sequence diagrams for static instruments (Figs. 7.20 to 7.22) are represented below in Figs. 7.30 to 7.32.

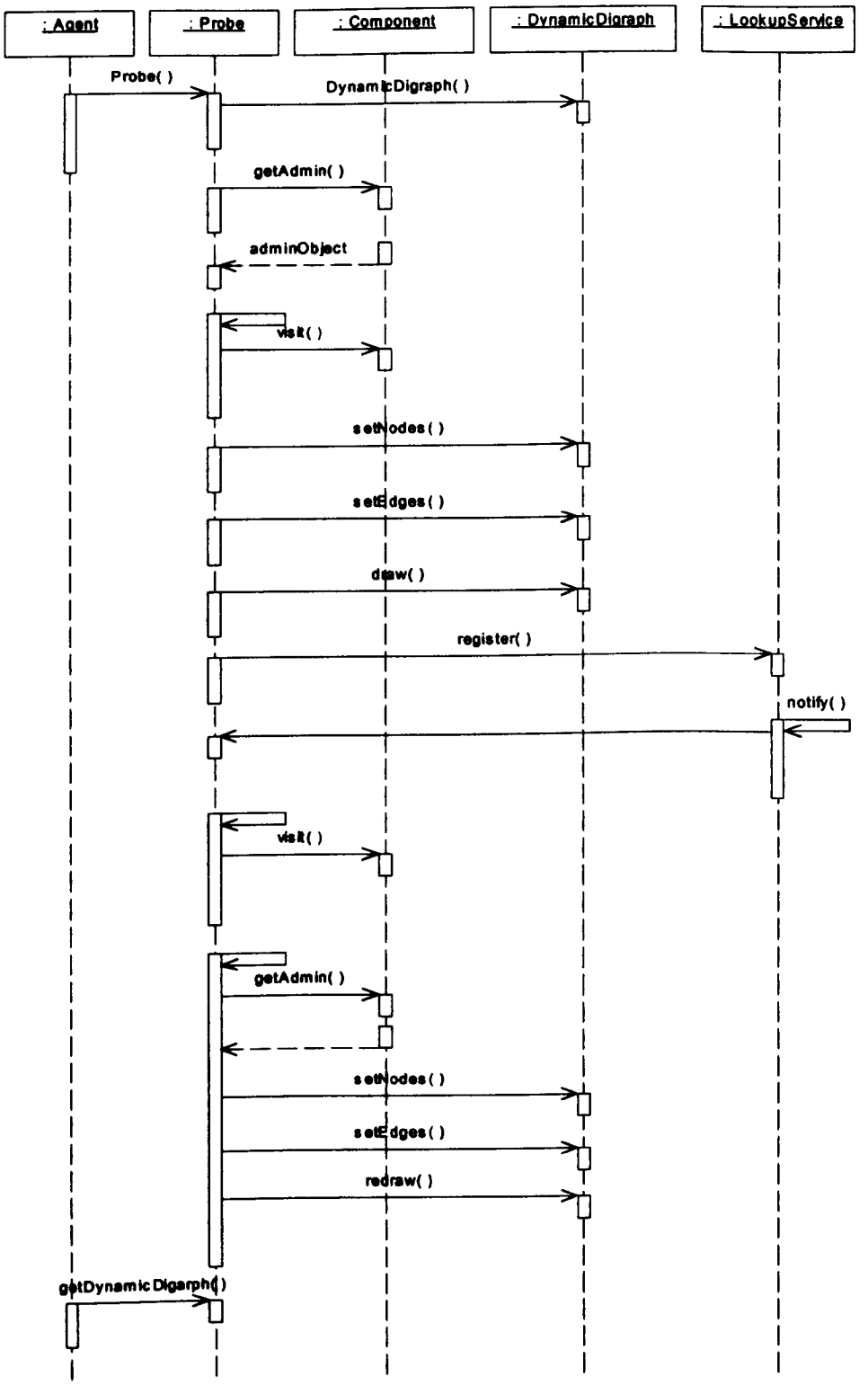


Figure 7.30: Probe sequence diagram

The probe sequence diagram is shown in Figure 7.30. The probe sequence begins when a management agent creates a probe object, via its constructor `Probe()`. The probe's own constructor then goes on to create a digraph, via the `DynamicGraph` constructor. Assuming the component implements the `Dependent` interface, the probe may then invoke the `getAdmin` method of the application component to access its administration object. This object will in turn allow the probe to access the component's bindings, via the `getBindings` method as described in Section 7.1.3.

Equipped with this immediate set, the probe must traverse each of these other components and access their own `getAdmin` methods to determine their secondary dependencies using the `visit` method. As the probe recurses through the various dependent and independent components it may derive the nodes and also the edges between nodes. When the recursion is finished (i.e. the `visit` method returns) the `setNodes` and `setEdges` methods of the `DynamicDigraph` class are used to set its `nodes` and `edges` attributes accordingly. The probe's `draw` method is then used to build an in-memory representation of the initial digraph.

If the probe is invoked in single-pass mode then the initial digraph represents the instantaneous dependencies associated with the component. However, if the probe is run as a thread it may at some stage be notified of changes in its components bindings via the notify method. This notification does not come from the component, but from the lookup service with which the component is registered. Such notifications occur when the component accesses the lookup service in order to download a new proxy of another application component from the lookup service. This action results in a change in the lookup service's mappings, which leads to an event being generated of which the probe is notified.

The lower half of the sequence diagram represents this action when a probe is running in thread mode. After the `draw` method of the initial graph drawing (as described above) returns, the probe registers with the components lookup service and receives any subsequent notifications of changes in the components bindings, via its `notify` method. The cycle above is then repeated and the probe gets a new administration object and redraws the dependency digraph. In single pass mode, a management agent may access

the digraph when the probe's draw method returns, via the `getDynamicDigraph`. In thread mode, a management agent may repeatedly access the digraph object when the probe's `draw` method returns and the probe thread is still running.

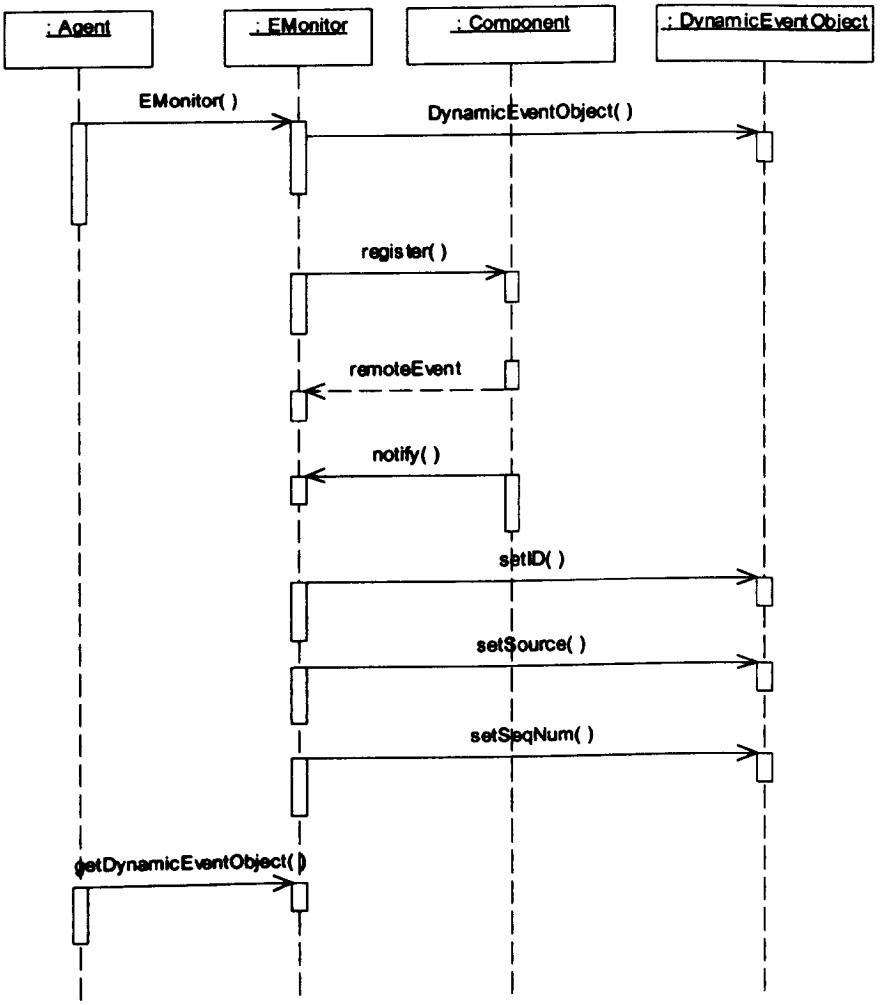


Figure 7.31: EMonitor sequence diagram

The event monitor sequence diagram is shown in Figure 7.31. The event monitor sequence begins when a management agent creates an event monitor object, via its constructor `EMonitor()`.The monitor's own constructor then goes on to create a dynamic event object, via the `DynamicEventObject` constructor. The monitor then uses its `register` method to register with the application component so that it may receive notifications of the remote event. When such events do occur the monitor's `notify` method is invoked and the monitor may then use the `setSource` and `setDest` methods to repackage the event as a dynamic event object. The management agent may then use

the `getDynamicEventObject` to access the repackaged event object. Like the probe, a monitor may either be invoked in a single pass to acknowledge a single event, or alternatively, it may be invoked within a thread to acknowledge events over the period of time for which the thread runs.

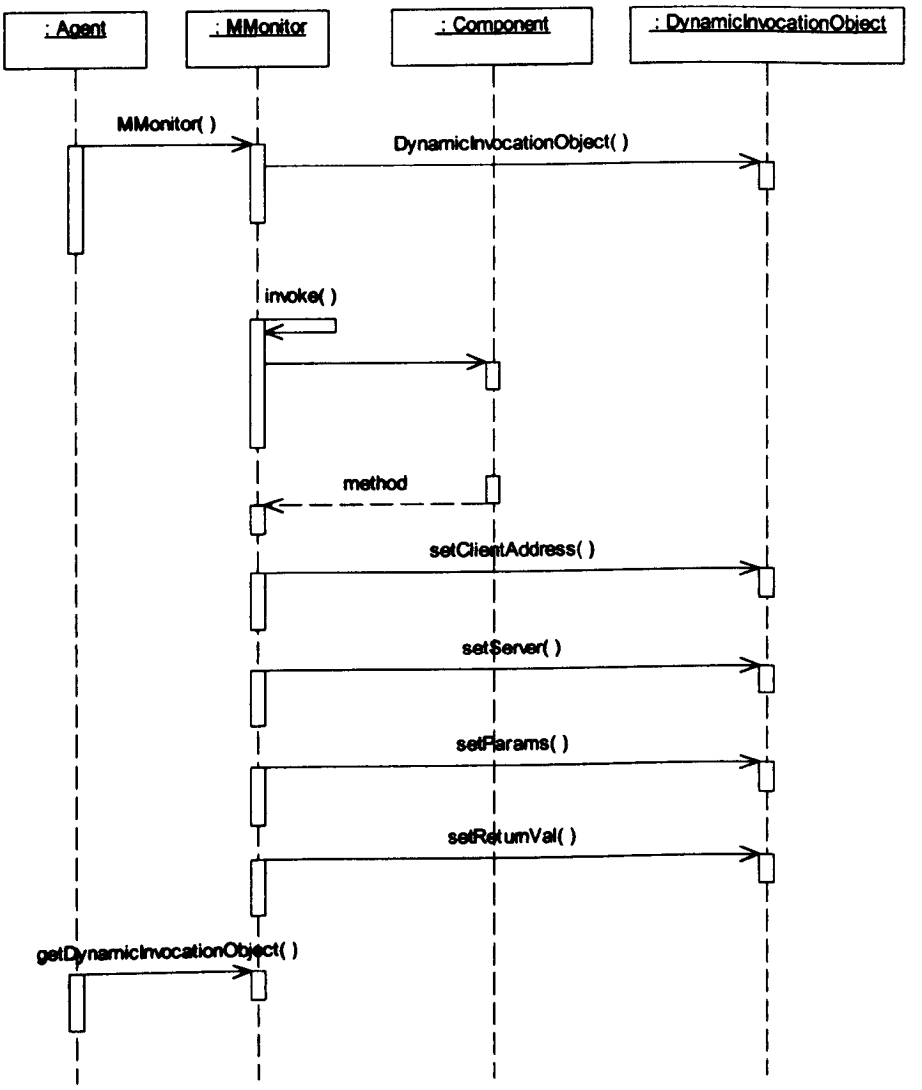


Figure 7.32: MMonitor sequence diagram

The method invocation monitor sequence diagram is shown in Figure 7.32. The sequence diagram is similar to that of the event monitor. The main difference is that the method invocation monitor uses its `invoke` method to perform the invocation of the method on a server component, as prescribed by the management agent. As mentioned previously, `BaseInstrument` implements an `invoke` method, which is called each time

a client invokes a method on a server and this action will be considered further in chapter 8. The overridden `invoke` method extends on this action by capturing the parameters and any return values that feature in the invocation and this extension will also be considered in chapter 8.

By capturing the invocation parameters and return value, the monitor instrument has full access to the characteristics of the invocation and it uses the `setClientAddress`, `setServer`, `setParams` and `setReturnVal` to repackage the invocation as an invocation object. The management agent may then use the `getDynamicInvocationObject` to access this invocation object. Like probe and event monitors, the invocation monitor may be invoked in single pass or thread mode.

7.3 Chapter Summary

This chapter has developed the instrumentation architecture for measuring and monitoring distributed applications. The architecture is based on the classification of instrumentation services of chapter 5 (Figure 5.1) and it has been developed through a series of UML models. The development process has taken advantage of the separation of operational and functional aspects relating to instrumentation services.

The main strength of the architecture is that it provides an extendable instrumentation layer capable of supporting additional specific instrumentation services to suit specific application requirements. The architecture comprises the infrastructure classes and a small number of general purpose instrumentation services that can be instantiated to measure/monitor distributed application components. The general purpose instrumentation services may be combined to conduct more complex measurement/monitoring tasks.

Whilst chapter 6 considered the operational aspects, this chapter has concentrated on the measurement and monitoring (i.e. functional) aspects. Now that these two parallel analysis and development stages of the journey have been completed, we may move on to the summit of our journey. This summit is that of the implementation of the instrumentation architecture to be considered next in chapter 8.

Chapter 8

Implementing the Instrumentation Architecture

This chapter considers the implementation of the instrumentation architecture using Java and Jini middleware technology. The implementation essentially brings together the formal analysis model of chapter 6 and the semi-formalized models of chapter 7. In doing so, the implementation fulfils the operational and functional requirements considered in chapter 5. The chapter begins with an overview of Jini middleware technology and describes the basic operations considered in chapters 5 and 6. The chapter goes on to describe several programming constructs, which are central to the implementation. In particular, the use of Java's dynamic proxy, Java's reflection API, Jini's `Administrable` interface and the visitor design pattern are all considered in relation to the architecture's infrastructure classes. The chapter then describes the instantiable instrumentation services, which may be used directly by management agents. The chapter ends by considering how third-party software applications may be used in conjunction with the architecture.

8.1 Jini Middleware Technology

Jini Middleware Technology (also referred to as Jini Network Technology) is a Java-based middleware developed by Sun Microsystems. Originally, Jini was developed as a technology to provide *plug and play* capabilities in networks that may contain a diverse range of physical devices including devices not normally found in a conventional network. However, Jini has rapidly pitched its stall amongst existing middleware technologies, such as Java RMI, CORBA and DCOM. Jini is in some ways related to Java RMI and Java RMI provides the main communication protocol for Jini. Aspects of Java RMI and its use in distributed application programming have already been mentioned in chapter 3. Through this section, we shall see how Jini raises the level of abstraction of Java RMI to facilitate service-oriented programming.

8.1.1 Jini Service-oriented Architecture

Jini's abstraction of a network is that of a *federation* of services, where a service represents a logical concept such as a printer or a chat-room. The notion of services is taken even further in the Openwings framework [26], as was mentioned in chapter 4. Jini provides a general-purpose middleware based on dynamic discovery and lookup protocols and communication is based on Java's RMI protocol. One of Jini's greatest strengths is its support for *mobile code*, which allows Java objects to be moved around a network in a "freeze-dried" format from which they may be reconstituted when they arrive at their destination. This so-called freeze-dried format is that of Java serialization and code is said to be *marshalled* over the network.

It is not the purpose of this chapter to extensively describe Jini's architecture, but only to provide a general background and to describe the major concepts relating to its architecture. This background is intended to provide sufficient information to support the description of the instrumentation architecture's implementation. However, more detailed and thorough treatments of Jini may be found in [25, 98, 99]. Jini's architecture is organized, into three categories: infrastructure, programming model and services, as shown in Figure 8.1.

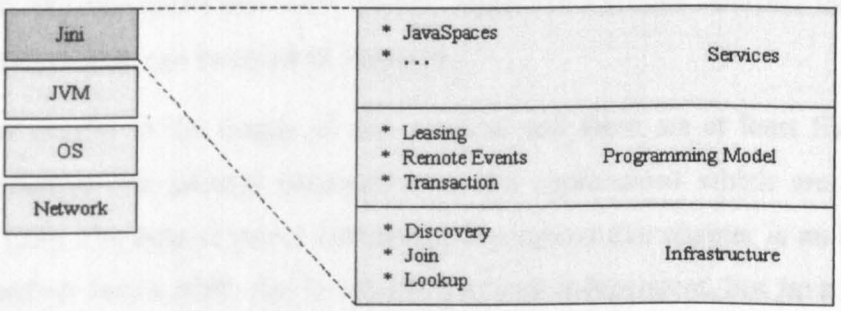


Figure 8.1: Jini architecture

The infrastructure layer (sometimes called *core* services) provides services for discovering Jini *communities* (groups of services), joining services on these communities and searching for services (lookup). The infrastructure is also responsible for providing minimal conditions for services to get into Jini networks. The programming model layer provides a set of APIs that enables the construction of

reliable application-level services. Whilst the first layer is concerned with infrastructure issues such as service availability and location, the second layer covers application-specific problems in a distributed context such as fault-tolerance (Leasing Service), asynchronous communication (Distributed, or Remote Events) and distributed consistency (Transaction Service). The last layer contains services that make use of both the programming model and infrastructure. These services will typically be programmer-defined services although Jini does provide its own JavaSpaces service, which implements a distributed shared memory facility.

8.1.2 Jini Services

Jini's fundamental abstraction is that of a service and Jini services may consist of hardware, software or a combination of the two. A service implements an interface (a Java interface), which describes the behaviour of the service. This interface is required by the platform for every service, since it is the interaction point between the service and its clients. The implementation of the service, however, is known only to the service itself. The interface is actually a *remote interface*, of the form described in chapter 3. In chapter 3, we learnt that remote interfaces are one of the two main concepts at the heart of a distributed object model (the other being remote object references). We also learnt that every remote object has a remote interface that specifies which of its methods can be invoked remotely.

Proxies are crucial to the design of Jini services and there are at least five different ways of creating Jini proxies (depending on the application) which are considered further in [25]. The type of proxy considered throughout this chapter is an RMI-based proxy, based on Java's RMI. Jini is actually protocol independent, but Java RMI is the most popular choice of underlying communication protocol. RMI-based proxies are generated by Java's RMI compiler (`rmic`), from information provided by a Java RMI interface and the remote objects that implement the interface. As mentioned in chapter 3, a Java RMI interface extends Java RMI's `Remote` interface and specifies the signatures of methods that are to be implemented by one or more remote objects. The `SimpleServiceInterface` shown below is an example of an RMI interface.

```
public interface SimpleServiceInterface extends Remote {
```

```
    private void doSomething();
    ....
}
```

A remote interface is often the starting point for developing a Jini service, since it allows the developer to specify the signatures of the methods that will subsequently be implemented by the service itself. As we shall see shortly, the interface also fulfils the important role of providing the remote reference through which a client may use RMI calls to communicate with a service. Essentially, the interface, or more specifically remote interface, plays a crucial role in allowing a client to communicate with a remote server object. Later in section 8.2.2, we shall see how remote interfaces serve as the main means through which we can attach instruments to Jini services, via the dynamic proxy class.

Within a Jini community a service is described by three elements: an identifier, a proxy and an array of attributes. The proxy is mandatory in that it must not be NULL, whereas the identifier and attributes must be present but may be NULL valued. The identifier is assigned to the service when it starts, the proxy is a mobile code entity, which represents the service at any of its clients. The proxy also implements the service's interface and isolates the communication protocol with the backend server from the client. The proxy essentially fulfils the roles of the stub and skeleton files used by Java RMI, as was described in chapter 3. Attributes provide additional information relating to the service, such as its location, its status, any GUI associated with the service. The three elements are referred to as the *service item* (also the *service object*) and are represented by Jini's `ServiceItem` class. The service item was introduced previously in the formal modelling chapter (chapter 6).

In Jini applications, services may associate themselves with one another to form communities, which are also known as *groups*. We may recall from chapter 5 and 6 the basic instrument operations of Join and Unjoin, which are intended to allow instrumentation services to organize into instrumentation groups. A community is regarded as a logical entity represented by a `java.lang.String`, which reflects either the physical or organizational structure of its services. As we shall see in chapter 9, this facility allows us to construct complex instruments as a community of primitive instruments, which communicate amongst each other. Within a community, services

interact with one another either as clients or servers. To support this interaction, they must get references to themselves and each other and this is accomplished through a special Jini core service, referred to as a *Lookup Service*.

8.1.3 Discovery Protocol

The Lookup Service acts as trader/broker between a client and the RMI registry to match a template specified by the client based on type and associated attributes. A lookup service describes the services that are available in a Jini community and provides operations for registration and service searching. However, before these operations can be used, a new service must get a reference to a lookup service that is active (running) within an existing community. This process is defined by the *Discovery* protocol. In this protocol, a service does not need to know the location of the lookup service, which means that clients need no prior configuration to find a service that they want to use. An asynchronous protocol version, implemented with UDP multicast, searches for lookup service references within the local network radius. When a reference is found, a remote notification is sent back to the new service and the new service is then able to retrieve the lookup service's own proxy. The code fragment below illustrates the use of this protocol.

```
public class SimpleService {
    ....
    class Listener implements DiscoveryListener {
        public void discovered(DiscoveryEvent evt) {
            ServiceRegistrar[] lookupServices;
            LookupServices = evt.getRegistrars();
            ....
        }
    }
    ....
    LookupDiscovery ld =
            new LookupDiscovery(new String[] {"instrument_1"});
    ld.addDiscoveryListener(new Listener());
    ....
}
```

The protocol may be implemented by the class `LookupDiscovery`. When the `ld` object is created, an asynchronous search of the community named "`instrument_1`" begins in the local network. Then a listener object is registered with `ld` so that when a lookup service is found, the `discovered` method is invoked to get a reference to the lookup

service. Note that the Jini lookup service itself implements the `ServiceRegistrar` interface.

Once a new service has a reference to the lookup service, it can register itself by invoking the lookup service's `register` method, as shown in the code fragment below.

```
ServiceItem item = new ServiceItem(id, this, attributes);
....
ServiceRegistration sr = lookupService.register(item,
                                                Lease.FOREVER);
```

The `item` argument represents the service item. It provides an identifier to the service (of `ServiceID` type) as the first argument, and a list of attributes (an array of `Entry` objects), to represent the service's properties, as the third argument. The second argument is a reference to an instance of `SimpleService` (specified as `this`). This argument is actually the proxy part of the service item, which will relay any remote invocations back to the object represented by `SimpleService` item.

In some cases, a reference to a *backend* server object may be more appropriate. For example, we may use `new SimpleServiceImpl()` as the second argument, where `SimpleServiceImpl` is a backend server implementation object to which remote method invocations are to be forwarded. The actual registration is performed by the `register` method, which registers `item` with a `lookupService` object. The second parameter in the register method is a constant declared by the `Lease` class defining for how long the service registration is valid. The `sr` object is a record of the registration and, through the lease object that is enclosed on it a `SimpleService` is able to renew its interest in maintaining registration.

8.1.4 Lookup Protocol

A service in a community may also want to look for another service, so as to act as a client of the other service, thereby using the functionality that the other service provides. This search is performed using the *Lookup* protocol and the code fragment shown below may be used to implement the lookup protocol.

```
Class[] classes = new Class[] {SimpleServiceInterface.class};
ServiceTemplate template = new ServiceTemplate(id, classes,
```

```
                                                    attributes);
....
Object proxy = lookupService.lookup(template);
```

The `template` object specifies the service's identifier, the interface that the service implements and the service's attributes. The result of the lookup method is a service proxy matching all the data specified by the template. The criteria used for matching by the lookup service was considered previously in chapter 6 (section 6.2.2).

Figure 8.2 illustrates the three steps that take place in Jini client-server communications. Figure 8.2 (a) shows the initial situation where a client wants to use the functionality provided by a service. Figure 8.2 (b) shows the steps that take place resulting in the client being in a position to make RMI calls on the service.

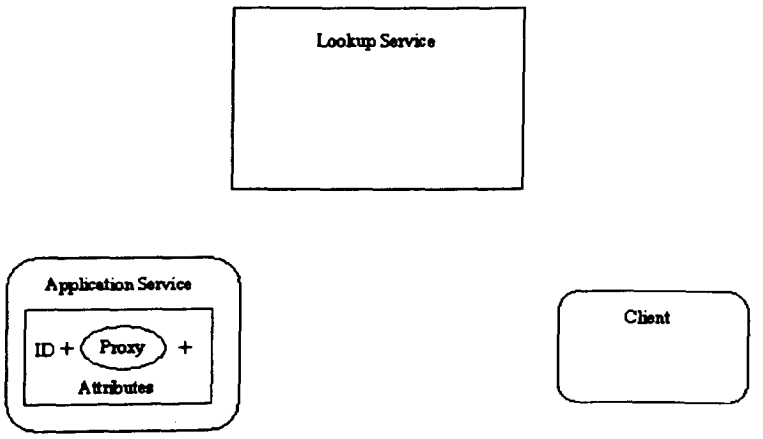


Figure 8.2 (a): Jini client-server communication – initial state

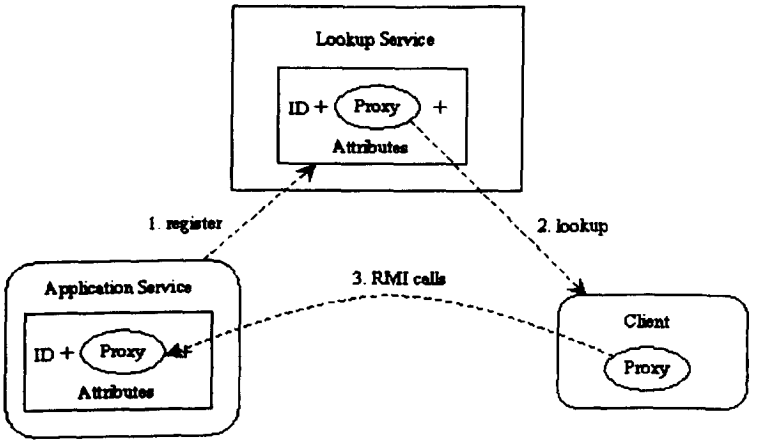


Figure 8.2 (b): Jini client-server communication – RMI calls

The first step is service registration, where the service registers its service item with the lookup service. The second step is client lookup, where the client constructs a template and the lookup service sends the client a proxy, which matches the client's template. At this stage, there are three copies of the service's proxy in existence: one stored in the service itself, one stored in the lookup service and now one stored locally in the client. The client may use its local copy to communicate with the service's proxy by invoking its methods using Java RMI calls. Later in Section 8.2.2, we use this pattern of communication to our advantage by *wrapping* the service's proxy within an instrumentation service's proxy to acknowledge all method invocations made by clients.

Having briefly considered the relevant aspects of Jini, we may now proceed to see how these ideas may be used to implement the dynamic instrumentation services that constitute the instrumentation architecture.

8.2 Implementing Dynamic Instrumentation Services

Dynamic instrumentation services are implemented as Jini services themselves. Chapters 6 and 7 have considered the basic instrument operations and the class hierarchy underlying the architecture. This section first describes the implementation of the basic instrument operations. This is followed by descriptions of the infrastructure classes and the instantiable instrument classes of Logger, Gauge, Analyzer, Probe and Monitor. The section also considers the programming constructs of dynamic proxy, visitor design pattern and `Administrable` interface, which are used to implement certain instrumentation services.

8.2.1 Discovery and Registration

The structure of the instrumentation infrastructure and instantiable classes was outlined in chapter 7. Throughout this section, the implementation of these classes are described through a series of incremental stages. Each stage will consider the implementation of several related methods, which either implement the basic instrument operations or provide measurement/monitoring functionality.

We begin by considering the Register and Unregister basic instrument operations, which are implemented within the `BaseInstrument` class. The `register` and `unregister` methods are implemented using the Jini discovery protocol and the service registration techniques described previously, as shown in the code below.

```
public interface BaseInstrumentInterface extends Remote,
                                    RemoteEventListener {
    public void register() throws RemoteException;
    public void unregister()throws RemoteException;;
    ....
    ....
}

public class BaseInstrument extends UnicastRemoteObject
                implements DiscoveryListener, LeaseListener,
                           BaseInstrumentInterface {
    ServiceItem item = null;
    ServiceID id = null;
    Entry[] attributes = null;
    ServiceRegistrar registrar = null;
    protected boolean[] state = null;
    protected LeaseRenewalManager leaseManager =
                new LeaseRenewalManager();
    protected Object wrapper = null;
    ....
    ....

    public BaseInstrument() throws Exception {
        super();
        state = new boolean[] {false, false, false};
        ....
    }

    public void register() throws RemoteException {
        LookupDiscovery ld = null;
        try {
            ld = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
        }
        discover.addDiscoveryListener(this);
        fireRemoteEvent(0);
    }

    public void unregister() throws RemoteException {
        try {
            leaseManager.cancel(sr.getLease());
        } catch(UnknownLeaseException e) {
            System.err.println(e.toString());
        }
        fireRemoteEvent(1);
    }
```

```
    public void discovered(DiscoveryEvent evt)
                    throws RemoteException {
        ServiceRegistrar registrar = evt.getRegistrars()[0];
        ServiceItem item = new ServiceItem(id, this, attributes);
        ServiceRegistration sr = null;
        try {
            sr = registrar.register(item, Lease.FOREVER);
        } catch(java.rmi.RemoteException e) {
            System.err.print("Register exception: ");
            e.printStackTrace();
        }
        try {
            System.out.println("service registered at " +
                    registrar.getLocator().getHost());
        } catch(Exception e) {
            System.err.println(e.toString());
        }
        leaseManager.renewUntil(sr.getLease(), Lease.FOREVER,
                                this);
    }
    ....
    ....
}
```

`BaseInstrument` implements the remote interface `BaseInstrumentInterface`, which specifies the method signatures for the basic instrument operations and the `reflect` method, which is considered in section 8.2.4. `BaseInstrument` also implements Jini's `DiscoveryListener` and `LeaseListener` interfaces. By implementing `DiscoveryListener` there is no need to use the inner-class `Listener`, as shown in the previous discovery protocol code.

The constructor of `BaseInstrument` first calls the superclass method `super` to create an instance of `UnicastRemoteobject`. The constructor then initializes the instruments state variable, which is a boolean array of three elements. The three elements represent an instrument's registration, attachment and joining states respectively (element 1: `register = true/unregister = false`, element 2: `attached = true/detached = false`, element 3: `joined = true / unjoined = false`). Later in section 8.2.3 we shall see how the state variable is updated on receipt of specific events.

After `BaseInstrument` has been created, its `register` method may be invoked. The `register` method simply sets up the discovery protocol by adding the discovery listener. When the first lookup service is found (`evt.getRegistrars()[0]`), by the invocation of `discovered`, the service item of `BaseInstrument` is registered via the

lookup service's own `register` method. The `discovered` method also uses a lease renewal utility, `leaseManager.renewUntil`, which will essentially keep the `BaseInstrument` registered until it is eventually unregistered, when the `unregister` basic instrument operation is invoked. Given that an instance of `BaseInstrument` will remain registered until its lease is cancelled, we may deduce a simple implementation of the unregister operation, which simply cancels the service's lease.

Both the register and unregister methods use the `fireRemoteEvent` method to transmit an event, indicating the registration state, to the current instrument and any instruments to which it is joined. This will be considered further in section 8.2.3.

8.2.2 Dynamic Instrumentation Proxies

Figure 8.2 illustrated the sequence of actions and subsequent RMI communication for a simple Jini-based client-server arrangement. In order to instrument such an arrangement, we need to develop a technique through which we may place instrumentation services in between the client and server. By doing so, we are then able to measure and monitor aspects of the communication. The key to placing instrumentation services between clients and servers lies in the *Dynamic Proxy* design pattern, which serves as the basis for the implementation of the Attach, Detach and Invoke basic instrument operations.

The dynamic proxy is a design pattern, which allows new interfaces to be implemented at runtime by forwarding all calls to an invocation handler. Java provides a dynamic proxy facility through the class `java.lang.reflect.Proxy` and the `InvocationHandler` interface. The following description of Java's dynamic proxy facility makes use of the example provided in [100], but with a modified description.

Listed below is a program, which represents the movement of an "explorer" around a Cartesian grid. The program includes an interface named `Explorer` and an implementation of the interface, `ExplorerImpl`. The explorer can travel in any compass direction, and can report its current location. The class `ExplorerImpl` uses two integer values to track the explorer's progress around the grid. The `TestExplorer` class sends the explorer on 100 random steps, and then logs the explorer's position.

```
import java.lang.reflect.Method;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

interface Explorer {
    public int getX();
    public int getY();
    public void goNorth();
    public void goSouth();
    public void goEast();
    public void goWest();
}

class ExplorerImpl implements Explorer {
    private int x;
    private int y;
    public int getX() {return x;}
    public int getY() {return y;}
    public void goNorth() {y++;}
    public void goSouth() {y--;}
    public void goEast() {x++;}
    public void goWest() {x--;}
}

public class TestExplorer {

    public static void test(Explorer e) {
        for (int i = 0; i < 100; i++) {
            switch ((int)(Math.random() * 4)) {
            case 0:
                e.goNorth();
                break;
            case 1:
                e.goSouth();
                break;
            case 2:
                e.goEast();
                break;
            case 3:
                e.goWest();
                break;
            }
        }
        System.out.println("Explorer ended at "
                           + e.getX() + "," +
                                    e.getY());
    }

    public static void main(String[] args) {
        Explorer e = new ExplorerImpl();
        test(e);
    }
}
```

The result of running `TestExplorer` would produce a single line of output, such as:

```
Explorer ended at -2,8
```

Now, imagine that the requirements for the application change, and it becomes necessary to log the explorer's movement at each step. Because the client test program was coded against an interface, this is straightforward: a `LoggedExplorer` wrapper class could be created, which logs each method call before delegating to the original Explorer implementation. This is an acceptable solution because it does not require any changes to `ExplorerImpl`. Using this approach, the new `LoggingExplorer` wrapper class could be written as below.

```
class LoggingExplorer implements Explorer {
    Explorer realExplorer;
    public LoggingExplorer(Explorer realExplorer) {
        this.realExplorer = realExplorer;
    }
    public int getX() {
        return realExplorer.getX();
    }
    public int getY() {
        return realExplorer.getY();
    }
    public void goNorth() {
        System.out.println("goNorth");
        realExplorer.goNorth();
    }
    public void goSouth() {
        System.out.println("goSouth");
        realExplorer.goSouth();
    }
    public void goEast() {
        System.out.println("goEast");
        realExplorer.goEast();
    }
    public void goWest() {
        System.out.println("goWest");
        realExplorer.goWest();
    }
}
```

The `LoggingExplorer` class delegates to an underlying `realExplorer` interface, which allows logging to be added to any existing `Explorer` implementation. The only change client test programs of the `Explorer` interface need to make is to construct the `LoggingExplorer` so that it wraps the `Explorer` interface. To do this, the `TestExplorer`'s main method may be modified as follows:

```
public static void main(String[] args) {
   Explorer real = new ExplorerImpl();
   Explorer wrapper = new LoggingExplorer(real);
      test(wrapper);
}
```

When the above program is run, the output would be similar to that shown below.

```
goWest
goNorth
....
goWest
goNorth
Explorer ended at 2,2
```

By delegating to an underlying interface, a new layer of functionality has been added without changing the `ExplorerImpl` code and this was achieved with only a trivial change to the client test program.

However, the `LoggingExplorer` wrapper class approach has two major drawbacks: first, it is tedious because each individual method of the `Explorer` interface must be re-implemented in the `LoggingExplorer` implementation. The second drawback is that the underlying problem (i.e. logging) is generic, but the solution is not. If it becomes necessary to log some other interface, then a separate wrapper class must be written.

The Dynamic Proxy class API can solve both of these problems. A dynamic proxy is a special class created at runtime by the Java Virtual Machine (JVM). A proxy class that implements any interface, or even a group of interfaces, may be requested by calling the proxy's `newProxyInstance` method, as shown below.

```
Proxy.newProxyInstance(ClassLoader
                       classLoaderToUse,
          Class[] interfacesToImplement,
        InvocationHandler objToDelegateTo)
```

The JVM manufactures a new class that implements the interfaces that are specified, forwarding all calls to `InvocationHandler`'s single method:

```
public Object invoke(Object proxy, Method meth, Object[] args)
          throws Throwable;
```

All that is required is an implementation of the `invoke` method in a class that implements the `InvocationHandler` interface. The proxy class then forwards all calls to this `invoke` method.

Such a proxy may be used to implement the `Explorer` interface by replacing the `LoggingExplorer` wrapper class with the `Logger` class shown below.

```
public class Logger implements InvocationHandler {
    private Object delegate;

    public Logger(Object o) {
        delegate = o;
    }

    public Object invoke(Object proxy, Method meth, Object[] args)
                    throws Throwable {
        System.out.println(meth.getName());
        try {
            return meth.invoke(delegate, args);
        } catch (InvocationTargetException e) {
          throw e.getTargetException();
        }
    }
}
```

This implementation of the `invoke` method can log any method call on any interface. It uses reflective invocation on the `Method` object to delegate to the real object.

The `TestExplorer` main method may then be modified, as shown below, to create a dynamic proxy class.

```
public static void main(String[] args) {
    Explorer real = new ExplorerImpl();
    Explorer wrapper = (Explorer)
                 Proxy.newProxyInstance(
            Thread.currentThread(
              ).getContextClassLoader(),
            new Class[] {Explorer.class},
            new Logger(real));
    test(wrapper);
}
```

The static method `Proxy.newProxyInstance` creates a new proxy that implements the array of interfaces passed as its second parameter. In this example, the proxy only needs to implement the `Explorer` interface. All invocations of `Explorer` methods are

then dispatched to the `InvocationHandler` that is passed as the third parameter. On running the updated code each step of the `Explorer` is logged to `System.out`.

The dynamic proxy class solves both of the problems of the wrapper approach. First, there is no tedious re-implementation of methods because `invoke` can handle all methods. Second, and most important, the dynamic proxy logger can be used to log calls to any interface in the Java language. The dynamic proxy logger is indeed *dynamic*, since it can adapt to implement any specified interface. The logging operation, of the dynamic proxy logger, is method-generic, that is, logging does not require any decision making based on the specifics of the method being called. This is exactly what we want for instrumentation services and dynamic proxies excel when adding method-generic services.

However, there is one drawback incurred from using dynamic proxies: like all reflective code, they are somewhat slower than "normal" code. If there is doubt in the performance of dynamic proxies, benchmarks associated with Java reflection should be consulted, such as those described in [101]. Chapter 9, which considers real instrumentation case studies considers the performance of applications with and without instrumentation services.

The previous logger example was simply used to demonstrate the concept of the dynamic proxy. It is not the actual implementation of our own logger instrumentation service, although the logger instrumentation service does take full advantage of the above ideas. The dynamic proxy is incorporated into the `DirectInstrument` class, which extends the `BaseInstrument` class. This gives all direct instantiable instruments (logger, gauge, analyzer, probe and monitors) the capability to implement any interface that is implemented by a Jini application service. Indirect instruments do not implement a dynamic proxy, since they receive information indirectly from other direct instruments. As we shall see later, certain instruments (method invocation monitors) will override the basic `invoke` method of `DirectInstrument` to provide more specialized facilities for dealing with method invocations.

The incorporation of the dynamic proxy within `DirectInstrument` involves the implementation of the instrument operations, Attach, Detach and Invoke as shown below.

```
public class DProxy implements InvocationHandler {
    private Object obj;

    public DProxy(Object obj) {
        this.obj = obj;
    }

    public Object invoke(Object proxy, Method meth, Object[] args)
            throws Throwable {
        System.out.println(meth.getName());
        try {
          return meth.invoke(obj, args);
      } catch (InvocationTargetException e) {
          throw e.getTargetException();
      }
    }
}

public class DirectInstrument extends BaseInstrument {
    ....
    ....

    public DirectInstrument() throws Exception {
        super();
    }

    public void attach(Object impl, Class[] ifaces) {
        if (state[0]) {
            if (wrapper != null)
                wrapper = null;
            wrapper = (Object) Proxy.newProxyInstance(
                    impl.getClass().getContextClassLoader(),
                    ifaces,
                    new DProxy(impl));
            fireRemoteEvent(2);
    }

    public void detach(Object impl, Class ifaœ) {
        Class[] ifaces = wrapper.getClass().getInterfaces();
        Class[] newIfaces = new Class[ifaces.length-1];
        for (int i = 0; i < ifaces.length; i++)
            if (ifaces[i] != iface) newIfaces[i] = ifaces[i];
        wrapper = null;
        attach(impl, newIfaces);
        fireRemoteEvent(3);
    }
    ....
    ....
}
```

The DProxy class implements an instrumentation service's dynamic proxy. As DProxy implements InvocationHandler, it must implement an invoke method, so that the JVM can forward all calls made on the specified object, obj, to the invoke method. The wrapper object (inherited from BaseInstrument) represents an instance of a dynamic proxy that may be accessed throughout the DirectInstrument class and any of its subclasses. The parameters of the attach method are an object impl, which implements the interfaces specified in the array of interfaces, ifaces.

The attach method first requires that the instrumentation service is registered (i.e. state[0] is true). The attach method then uses the Proxy.newProxyInstance to create a new proxy that implements the array of interfaces. All invocations made on the methods specified by the interfaces, ifaces, are then dispatched to the invocation handler (new DProxy(impl)) that is passed as the third parameter.

The detach method simply removes and interface, iface, from the list of interfaces implemented by the proxy instance. As there is no direct means to change the list of interfaces of a proxy instance, the detach method first removes the original proxy instance (wrapper = null) and invokes the attach method to create a new instance, containing the revised interface array, newIfaces. Note that each time Proxy.newProxyInstance is called, a new proxy instance is created, so it is important to ensure that there is only ever one instance within DirectInstrument and any of its subclasses. Similar to the register and unregister methods, the attach and detach methods use the fireRemoteEvent method to transmit an event, indicating the attachment state, to the current instrument and any instruments to which it is joined.

The attach and detach methods may be invoked to attach and detach an instrumentation service to a Jini service as shown below, where inst is the instrumentation service in question.

```
public class SimpleService implements SimpleServiceInterface1,
                                      SimpleServiceInterface2 {
    ....
    ....
}
....
....
```

```
SimpleService ss = new SimpleService()
....
inst.attach(ss, ss.getClass().getInterfaces())
....
inst.detach(ss, SimpleServiceInterface2)
```

In many cases, the array of interfaces implemented by a dynamic proxy will only consist of a single interface, as was the case for the previous logger example. However, instances do arise when a class implements several interfaces, which each prescribe different behaviours. When this is so, the multiple interfaces often range from generic to more specific interfaces. For example, a graphics object used in a GUI may implement graphics-based interfaces and also mouse event listener interfaces.

The previous example of the dynamic proxy based logger was simply intended to demonstrate the concept of the dynamic proxy and as such, it did not consider aspects of distribution. However, where distributed instrumentation services are concerned, this issue must be explained further: the significant point lies in the differences between the interface array used in the logger example and `DirectInstrument` dynamic proxies. In the logger example, standard Java interfaces are used, whereas in `DirectInstrument`, the interfaces are Java RMI-based interfaces, or remote interfaces. If a dynamic proxy provides a wrapper, which implements a remote interface then method invocations made on a remote object, via its remote interface, are forwarded to the dynamic proxy's `invoke` method.

Essentially, the operation of instrumentation attachment may be regarded as *wrapping* a Jini service proxy within the dynamic proxy of an instrumentation service to provide a *compound* proxy. The client may still use the service's proxy as before to communicate with remote server objects, via Java RMI calls. However, it is unaware that these calls are forwarded to the dynamic proxy's `invoke` method on the server-side. This instrumented communication pattern may be illustrated by modifying Figure 8.2 to represent the action of wrapping the service's proxy to create a compound proxy, as shown in Figure 8.3.

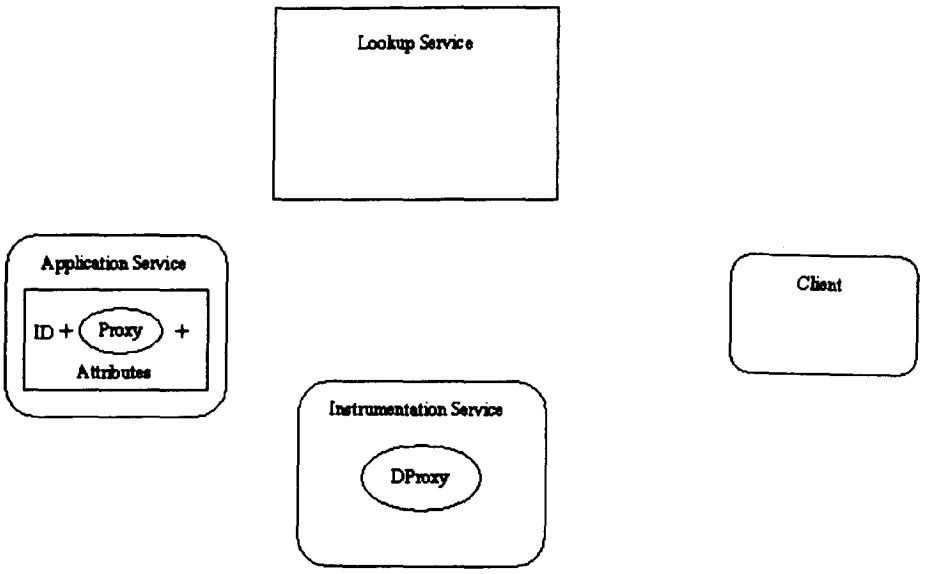


Figure 8.3 (a): instrumenting application service – initial state

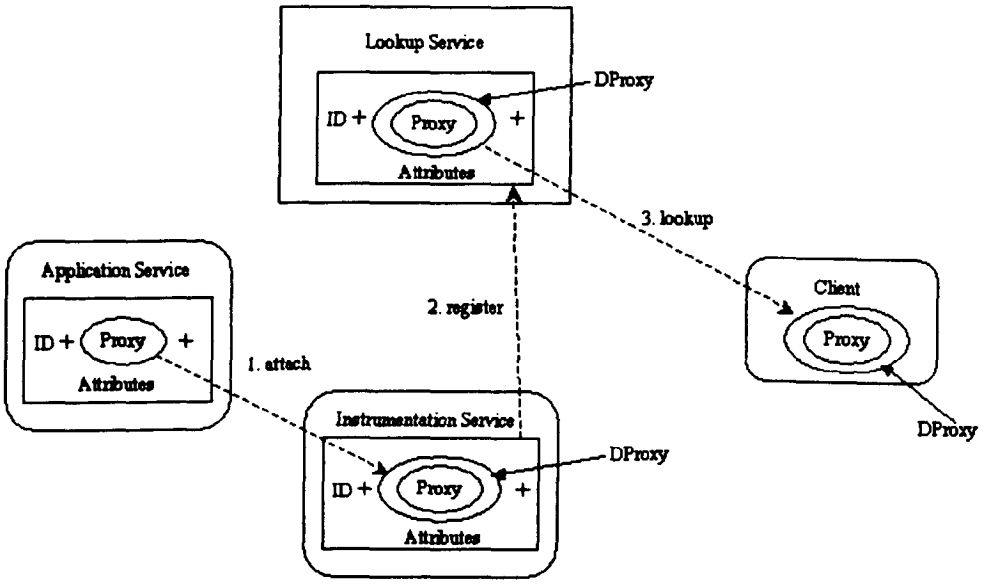


Figure 8.3 (b): instrumenting application service – attach operation

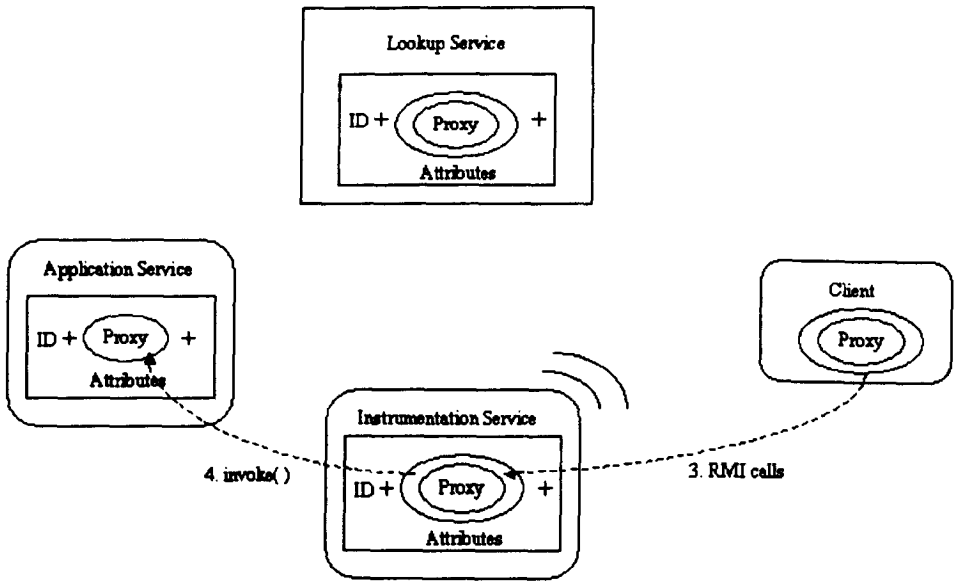


Figure 8.3 (c): instrumenting application service – intervening RMI calls

Figures 8.3 (a) to (c) represent the stages that take place when an application service is instrumented using the attach operation: and the client makes RMI calls on the instrumented application service:

- Figure 8.3 (a) shows the initial state before the application service has registered with the lookup service.
- Figure 8.3 (b) shows the attach operation, where the application service's proxy is wrapped within the instrumentation services dynamic proxy (DProxy) to create a compound proxy.
- The application service's item is then registered but the actual proxy object bound in the lookup service registry is the application service's proxy wrapped up in the dynamic proxy (i.e. the compound proxy).
- When the client performs a lookup a match is found against the application service's proxy but the actual object sent to the client is the compound proxy (actual proxy wrapped in DProxy).
- Figure 8.3 (c) shows the client making RMI calls on the now instrumemted application service. DProxy maintains a reference to the application service's proxy so RMI calls are propagated from DProxy first to the

service's proxy and then onto the remote object that implements the application service.

- The instrumenting of the application service and intervention on RMI calls is completely transparent to the client.
- If the detach operation was applied to Figure 8.3, the communication pattern would revert back to that of the uninstrumented pattern of Figure 8.2.

Note that typically the instrumentation service would also have registered with the lookup service This allows the instrumentation service to be made aware of any event notifications that the application service receives/sends. It also allows the instrumentation service to receive notifications from other instrumentation services. However, if the instrumentation service registration was shown the diagram would be cluttered and obscure the main point of portraying the instrumenting of an application service.

8.2.3 Instrumentation Service Communications

Instrumentation services may communicate with other instrumentation services using the basic operations of Read, Write and Notify. However, before instrumentation services can communicate they must be *joined* to one or more other instrumentation services. Joining takes place through the instrumentation service's service item and its dynamic proxy. If an instrumentation service finds the service item of another instrumentation service and accesses the dynamic proxy instance of the other service, it may then communicate with the other service via `read`, `write` and `notify` methods. The `join`, `unjoin`, `read`, `write` and `notify` methods are implemented within the `BaseInstrument` class as shown in the code below.

```
public interface BaseInstrumentInterface extends Remote,
                                                  RemoteEventListener {
    ....
    ....
    public void notify(RemoteEvent event)
                throws UnknownEventException, RemoteException;
    public EventRegistration addRemoteEventListener(
                RemoteEventListener listener,
                MarshalledObject handback)
                throws RemoteException;
```

```
    public Object getProxy();
    public ServerSocket getServerSocket();
    ....
    ....
}

public class BaseInstrument extends UnicastRemoteObject
                implements DiscoveryListener, LeaseListener,
                BaseInstrumentInterface {
    static final int MAX_INSTRUMENTS = 100;
    static final int PORT = 9000;
    ServiceItem item = null;
    ServiceID id = null;
    Entry[] attributes = null;
    ServiceRegistrar registrar = null;
    public boolean[] state = null;
    protected LeaseRenewalManager leaseManager =
                        new LeaseRenewalManager();
    protected Object wrapper = null;
    public Vector group = new Vector();
    public Vector proxies = new Vector();
    String buffer;
    ServerSocket serverSocket = null;
    Socket socket = null;
    BufferedReader in = null;
    PrintWriter out = null;
    public long count = 0L;
    public Dictionary listeners = new Hashtable();
    ....
    ....

    public BaseInstrument() throws Exception {
        super();
        state = new boolean[] {false, false, false};
        try {
            serverSocket = new ServerSocket(PORT);
        } catch (IOException e) {
                System.err.println(e.toString());
        }
        ....
    }

    public void join() throws RemoteException {
        if (state[0]) {
            Class[] classes =
                   new Class[] {BaseInstrumentInterface.class};
            ServiceTemplate template =
                     new ServiceTemplate(id, classes, attributes);
            try {
                ServiceMatches matches =
                     registrar.lookup(template, MAX_INSTRUMENTS);
                for (int i = 0; i < matches.items.length; i++) {
                    BaseInstrumentInterface bi =
              (BaseInstrumentInterface)matches.items[i].service;
                    if (bi != null) {
                        bi.addRemoteEventListener(bi, new
                          MarshalledObject("BaseInstrument"));
```

```
                    if (bi != item) {
                        group.addElement(bi);
                        proxies.addElement(bi.getProxy());
                    }
                }
            }
            fireRemoteEvent(4);
        } catch(Exception e) {
            System.err.println(e.toString());
        }
    }
}

public void unjoin() {
    group = new Vector();
    proxies = new Vector();
    fireRemoteEvent(5);
}

public Object getProxy() {
    return wrapper;
}

public String read() {
    if (state[0] && state[2]) {
       try {
           socket = serverSocket.accept();
           in = new BufferedReader(new
                 InputStreamReader(socket.getInputStream()));
           String s = in.readLine();
           in.close();
           socket.close();
           return s;
       } catch (IOException e) {
           System.err.println(e.toString());
       }
    }
}

public boolean write(BaseInstrumentInterface receiver) {
    if (state[0] && state[2]) {
       try {
           for (int i = 0; i < group.size(); i++) {
               BaseInstrumentInterface bi =
                (BaseInstrumentInterface)group.elementAt(i);
               if (bi == receiver)
                   socket = bi.getServerSocket().accept();
           }
           if (socket != null) {
               out =
                 new PrintWriter(socket.getOutputStream());
               out.print(buffer);
               out.close();
               socket.close();
               return true;
           }
           else {
```

```
                return false;
            }
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }
}

public ServerSocket getServerSocket() {
    return serverSocket;
}

public EventRegistration addRemoteEventListener(
        RemoteEventListener listener,
        MarshalledObject handback)
        throws RemoteException {
    try {
        listeners.put(listener, handback);
        return new EventRegistration(0, this, null, count);
    } catch(Exception e) {
        System.err.println(e.toString());
    }
}

public void notify(RemoteEvent event)
            throws UnknownEventException, RemoteException {
   try {
        BaseInstrumentInterface bi =
             (BaseInstrumentInterface)event.
             getRegistrationObject().getSource();
        if (bi != this) {    // events from other instruments
            switch ((int)event.getID()) {
            case 0:
                //register
                break;
            case 1:
                //unregister
                group.remove(bi);
                proxies.remove(bi.getProxy());
                break;
            case 2:
                //attach
                break;
            case 3:
                //detach
                proxies.remove(bi.getProxy());
                break;
            case 4:
                //join
                break;
            case 5:
                //unjoin
                group.remove(bi);
                proxies.remove(bi.getProxy());
                break;
           default:
                System.err.println("Unknown Event");
```

```
                break;
            }
        }
        else {   // events from ourself
            switch ((int)event.getID()) {
            case 0:
                //register
                state[0] = true;
                break;
            case 1:
                //unregister
                state[0] = false;
                break;
            case 2:
                //attach
                state[1] = true;
                break;
            case 3:
                //detach
                state[1] = false;
                break;
            case 4:
                //join
                state[2] = true;
                break;
            case 5:
                //unjoin
                state[2] = false;
                break;
            default:
                System.err.println("Unknown Event");
                break;
            }
        }
    } catch(IOException e) {
        throw new UnknownEventException("IOException: " +
                                        e.getMessage());
    } catch(ClassNotFoundException e1) {
        throw new
             UnknownEventException("ClassNotFoundException: " +
                                        e1.getMessage());
    }
}

protected void fireRemoteEvent(long id) {
    Enumeration enum = listeners.keys();

    while (enum.hasMoreElements()) {
        RemoteEventListener listener =
                    (RemoteEventListener)enum.nextElement();
        RemoteEvent event = new RemoteEvent(this,id,count,
                    (MarshalledObject)listeners.get(listener));
        try {
            listener.notify(event);
        } catch(UnknownEventException e) {
            e.printStackTrace();
        } catch(RemoteException e1) {
```

```
                    e1.printStackTrace();
                }
            }
            count++;
        }
        ....
        ....
    }
```

The `join` method uses Jini's `ServiceMatches` class, which is part of the core `lookup` package.

```
package net.jini.core.lookup;

public Class ServiceMatches {
    public ServiceItem[] items;
    public int totalMatches ;
}
```

If a service wants to search for more than one match to a service template from a particular lookup service, then it specifies the maximum number of matches it would like returned as the second parameter in the `lookup` method (`lookup(template, MAX_INSTRUMENTS )`. The requesting service then receives a `ServiceMatches` object, which contains an array of items of `ServiceItem` type that match the specified template.

The number of elements returned in `items` need not be the same as `totalMatches`. For example, if there are five matching services stored in a lookup service then `totalMatches` will be set to five after the `lookup` method is invoked. However, if the second parameter in the `lookup` method is specified as two matches, then `items` will be set to be an array with only two elements. Not all elements of this array need be non-null, since one element may represent a "stale" service, whereas the other elements represent valid active services. Null elements allow the requesting service to distinguish between those services that may be used and those that may not if, for example, their leases have expired.

The join operation is performed primarily on an instrument's service item and, more significantly its dynamic proxy. The `join` method uses a template to find services that implement the `BaseInstrumentInterface`. When a group of instrumentation service

items have been found, they are stored in the `group` vector. The service item of `BaseInstrument` will also feature in the group of service items returned, so it and any null items are not stored in `group`. Remote event listeners are added to each service item returned in the group, including the service item associated with the current instance of `BaseInstrument`. As we shall see shortly, these remote event listeners allow the instrumentation services within the group to notify each other of any changes in their respective states.

The dynamic proxy wrappers of each instrumentation service are accessed using the `getProxy` method and stored in the `proxies` vector. Through these proxies, any instrumentation service, within the group, becomes privy to any method invocations made on the application-level services attached to the group. The `unjoin` method simply cancels the effects of any joining by resetting the `group` and `proxies` vectors. The `unjoin` method effectively severs the link between the current instrument and any other instruments, with which it was previously joined.

The main aim of the Join instrument operation is to allow compound instrumentation units to be dynamically constructed from the primitive instrumentation service classes of logger, gauge, analyzer, probe and monitor. So for example, an event monitor, a method invocation monitor and an analyzer may be joined together to form a compound instrument. This compound instrument may then be used to analyze the access patterns that clients make on a particular application-level service over a period of time. The monitors will be responsible for detecting and repackaging events and method invocations. The analyzer may then examine the repackaged objects and use them to compute access/usage patterns over a period of time by determining: which clients access the service, the specific events/invocations that each client receives/makes, and the frequencies at which communications occur.

When several instrumentation services are joined they may communicate with one another using `read`, `write` and `notify` methods. The `read` and `write` methods allow instrumentation services to directly read and write message streams between one another. The `read` and `write` methods use direct TCP socket connections so that reading and writing can proceed, whilst an instrumentation service is engaged in other

activities (e.g. a separate thread is running monitoring method invocations). The `read` method will read the stream transmitted via a socket into the instrumentation service's buffer. The `write` method will transmit the current contents of buffer via a socket to a specified receiving instrument whose service item is currently stored in `group`. Both `read` and `write` require that the state of the current instrument is registered and joined (`state[0] && state[2]`), but not necessarily attached.

The basic `notify` method is used to communicate state changes between a group of instrumentation services. Later, we shall see how the basic method is overridden by event monitor instrumentation services. The basic `notify` method is invoked whenever the current instrumentation service or any other instrumentation service, within a group, undergoes a change in state (e.g. it has been detached from an application service). Because each instrumentation service has a remote event listener (from `addRemoteEventListener`) it is registered to receive specific event within the group. Whenever such events occur the group members receive notification via their `notify` methods.

If `notify` is invoked and the event source is an instrumentation service other that the current instrumentation service, then the first switch statement is executed. If the event received indicates that an instrumentation service's state has changed to that of unregistered, or unjoined, then the instrumentation service is removed from `group`. If the event indicates that the instrumentation service is now detached, then its dynamic proxy is removed from `proxies`.

If `notify` is invoked and the event source is the current instrumentation service, then the second switch statement is executed. The event's ID is then used to update the `state` variable of the current instrument. One may regard this as an overly complex approach for maintaining the state of the current instrument. For example, one may argue: *"why not simply update the state variable within each of the basic instrument operations?"*. The answer to this, and the reason for the approach is that events that affect the current instrumentation service's state may not always be sent by the current instrumentation service itself. Certain events may come from elsewhere within an application, especially lookup services.

If a lookup service should fail, an event will be sent to notify of service registration failures and this event must be interpreted by an instrumentation service so that its registration state may be updated. On a less dramatic scale, a lookup service may refuse to renew the lease of an instrumentation service and this must also be interpreted to maintain a consistent registration state for the lookup service. In summary, the event-based approach faithfully maintains state of instrumentations services by accommodating events that may occur elsewhere in a distributed application.

The `fireRemoteEvent` method is used by each of the basic instrument operations to signal a change in state to all instrumentation services who have registered to receive notification of such events (i.e. all instrumentation services in `group`). The event is sent to all instruments by enumerating all event listeners, currently registered to receive events.

Having considered the implementation of the ten basic instrument operations, we may proceed to consider several important programming constructs used to access runtime information from application-level services. The first is the use of reflection to introspect and access structural and runtime information from application-level services. The second is the use of Jini's `Administrable` interface and the third is the *visitor* design pattern to determine the dependencies associated with application-level services.

8.2.4 Using Reflection to Access Runtime Information

Reflection is a feature of the Java programming language, which allows an executing Java program to examine or *introspect* upon itself and even manipulate internal properties of the program. Through the reflection API it is possible for a Java class to obtain: the names, types and runtime values of its attributes (fields), the signatures of all its methods and constructors and even the superclasses and interfaces. Reflection was introduced previously in chapter 7 as a technique that is used in the instrumentation architecture to access structural class information and runtime (behavioural) information relating to an object. In this section, we consider the use of Java's reflection API to provide reflective capabilities within the `BaseInstrument` class.

The main reflection entry point for the `BaseInstrument` class is the `reflect` method shown below.

```
public class BaseInstrument extends UnicastRemoteObject
                implements DiscoveryListener, LeaseListener,
                BaseInstrumentInterface {

    public static final int NCLASSES = 20;
    public static final int SUPERCLASSES = 0;
    public static final int INTERFACES = 1;
    public static final int CONSTRUCTORS = 2;
    public static final int FIELD = 3;
    public static final int METHOD = 4;
    public static final int ARRAY = 5;
    ....
    ....

    public BaseInstrument() throws Exception {
        super();
        ....
    }

    public Object reflect(Object object, String str, int type,
                                int index) {
        Object param = null;
        switch(type) {
        case SUPERCLASSES:
            param = reflectSuperclasses(object);
            break;
        case INTERFACES:
            param = reflectInterfaces(object);
            break;
        case CONSTRUCTORS:
            param = reflectConstructors(object);
            break;
        case FIELD:
            param = reflectField(object, str);
            break;
        case METHOD:
            param = reflectMethod(object, str);
            break;
        case ARRAY:
            param = reflectArray(object, str, index);
            break;
        default:
            System.err.println("Invalid Parameter");
            break;
        }
        return obj;
    }

    Class[] reflectSuperclasses(Object object) {
        Class[] classes = new Class[NCLASSES];
        Class subclass = object.getClass();
```

```
        Class superclass = subclass.getSuperclass();
        classes[0] = superclass;
        int i = 0;
        while (superclass != null) {
            subclass = superclass;
            superclass = subclass.getSuperclass();
            classes[i++] = superclass;
        }
        return classes;
    }

    Class[] reflectInterfaces(Object object) {
        Class clazz = object.getClass();
        Class[] interfaces = clazz.getInterfaces();
        return interfaces;
    }

    Object reflectConstructors(Object object) {
        Class clazz = object.getClass();
        Constructor[] constructors = clazz.getConstructors();
        return constructors;
    }

    public Object reflectConstructorParams(Constructor c) {
        return c.getParameterTypes();
    }

    public Object reflectConstructorExceptions(Constructor c) {
        return c.getExceptionTypes();
    }

    Object reflectField(Object object, String f) {
        Field fld = null;
        Object value = null;
        try{
            Class clazz = object.getClass();
            fld = clazz.getField(f);
            value = fld.get(object);
        } catch (Throwable e) {
            System.err.println(e);
        }
        return value;
    }

    public Object reflectFieldType(Field f) {
        return f.getType();
    }

    Object reflectMethod(Object object, String m) {
        Object method = null;
        try {
            Class clazz = object.getClass();
            Method[] methods = clazz.getMethods();
            for (int i = 0; i < methods.length; i++) {
                if (methods[i].getName().equals(m))
                method = methods[i];
            }
```

```
        } catch (Throwable e) {
            System.err.println(e);
        }
        return method;
    }

    public Class[] reflectMethodParams(Method m) {
        return m.getParameterTypes();
    }

    public Class reflectMethodReturn(Method m) {
        return m.getReturnType();
    }

    public Class[] reflectMethodExceptions(Method m) {
        return m.getExceptionTypes();
    }

    Object reflectArray(Object object, String f, int index) {
        Field arr = null;
        Object value = null;
        try {
            Class clazz = object.getClass();
            Field[] fields = clazz.getFields();
            for (int i = 0; i < fields.length; i++) {
                String fieldName = fields[i].getName();
                Class typeClass = fields[i].getType();
                if (fieldName.equals("arr") &&
                            typeClass.isArray()) {
                    Object array = fields[i].get(object);
                    value = Array.get(array, 2);
                }
            }
        } catch (Throwable e) {
            System.err.println(e);
        }
        return value;
    }
}
```

The `reflect` method is used to access a parameter that is to be measured or monitored for a target class and its associated runtime instance. The process through which the `reflect` method operates has already been considered in the sequence diagrams of chapter 7. Essentially, this process requires that a management agent specifies a runtime instance of a class (`Object object`) along with a type (e.g. `FIELD`) associated with the parameter to be reflected on. Additional parameters may also be used to identify the name of a particular attribute or method or the index of an array. Based on the type of parameter, the `reflect` method then calls an appropriate method to access the parameter of interest. The `reflect` method may be used to access: superclasses,

interfaces, constructors, attributes (fields), methods and arrays and also parameter types and exceptions associated with any of the previous.

The following code shows the reflect method in action, when incorporated into a simple test program.

```
public class TestReflection {

    public Object reflect(Object object, String str, int type,
                int index) {
        // coded as above
        ....
        ....
    }

    public static void main(String args[]) {
        TestReflection tr = new TestReflection();
        MyClass mc = new MyClass();
        System.out.println(tr.reflect(mc,"", SUPERCLASSES, 0));
        System.out.println(tr.reflect(mc, "idNumber", FIELD, 0));
        System.out.println(tr.reflect(mc, "lastName", FIELD, 0));
        System.out.println(tr.reflect(mc, "arr", ARRAY, 2));
        System.out.println(tr.reflect(mc, "larger", METHOD, 0));
        Object m = tr.reflect(mc, "doSomething", METHOD, 0);
        Class[] classes = tr.reflectMethodParams((Method)m);
        for (int i = 0; i < classes.length; i++) {
            Object object = classes[i];
            System.out.println(object.toString());
        }
    }
}

public class MyClass {

    public String firstName = null;
    public String lastName  = null;
    public int    idNumber  = 0;
    public int[] array = new int[]{5,6,7};

    public MyClass(String fname, String lname, int id) {
        this.firstName = fname;
        this.lastName  = lname;
        this.idNumber  = id;
    }

    public MyClass() {
        this("Denis", "Reilly", 12345);
    }

    public boolean larger(int a, int b) {
        if (a > b)
            return true;
        else
            return false;
```

```
    }

    public String toString() {
        return (lastName + ", " + firstName + " [" + idNumber +
                        "]");
    }
}
```

The example shows how we may access information for an instance `mc` of the class `MyClass`. When this example is run, the output is as shown below.

```
[Ljava.lang.Object;@273d3c]
12345
Reilly
7
public boolean larger(int,int)
int
int
```

The first print statement prints out the array of superclasses for `MyClass`, which in this case is the single class `java.lang.Object`. The second print statement prints the value of the attribute (field) `idNumber` and the third print statement prints the value of the attribute `lastName`. Notice that the values of `idNumber` and `lastName` are the runtime values, when an instance of `MyClass` has been created, and not the initial values. The fourth print statement prints the value of the third element of attribute `array`. The fifth print statement prints the method signature of the `larger` method. The final print statements print the types of parameters of the `larger` method. In this simple example they are primitive `int` types, but if they were classes other than primitives we could access their respective classes to determine their runtime values. We could also access the return value of the `larger` method using the dynamic proxy's `invoke` method that was described previously in section 8.2.2.

Taken together, the `reflect` and `invoke` methods allow an instrumentation service to access structural and runtime (behavioural) information from a target class. However, they may also be used to extend beyond a single target class. For example, the reflect method provides access to the superclass and other class fields of any given target class. We may then visit each of these classes and any of their superclasses and class fields and access the parameters of these classes in a recursive fashion. Through this

capability, we may build up a comprehensive picture of a classes' structure and runtime behaviour. Furthermore, by using the join operation, we may delegate the instrumentation of the target class and its ancestors and descendants amongst a group of instrumentation services.

8.2.5 Using Administrable and Dependent Interfaces to Represent Dependencies

Previously, chapters 5 and 7 considered the issues associated with determining the dynamic dependencies of an application-level service. Chapter 5 also described how a binding occurs when one service (the dependent) has downloaded a copy of the proxy of some other service (the independent) with a view to invoking the methods of the independent service. Chapter 7 went on to consider the problems of accessing the bindings of a service and proposed a compromise based on the use of the `Administrable` and `Dependent` interfaces. In this section, we consider the use of these interfaces to provide capabilities through which we may derive the bindings and hence dependencies associated with an application-level service.

The approach to represent dependencies is based on the service dependency work conducted by Hasselmeyer [7, 9]. The approach makes use of Jini's own `Administrable` interface. The `Administrable` interface allows application programmers to attach service-specific information to their services so that the information may be accessed and even changed by any client or any other service within a Jini federation. Such information may be retrieved, as an *admin. object*, by invoking the method `getAdmin`. Of course, any service that implements the `Administrable` interface is required to implement the `getAdmin` method, which returns an admin. object.

Chapter 7 described the compromise introduced into the architecture, which places a requirement on application programmers. The requirement, for any dependent services is that programmers must include an admin. object and must implement Jini's `Administrable` interface in order to return the admin. object. Through this compromise, probe instrumentation services may then gain access to the bindings associated with an application service.

The `Dependent` interface is specified as below.

```
public interface Dependent {
    public Class getDeclaringClass();
    public Object[] getBindings();
}
```

A service admin. object implements this interface as below.

```
public class ServiceAdmin implements Dependent {
    public Object obj = null;
    public Object[] bindings = null;

    public ServiceAdmin(Object obj) {
        this.obj = obj;
    }

    public Class getDeclaringClass() {
        return obj.getClass();
    }

    public Object[] getBindings() {
        return bindings;
    }
}
```

Then any application-level service, which is likely to be dependent on other services must include a `ServiceAdmin` attribute and must implement the `Administrable` interface by implementing the `getAdmin` method, which returns the `ServiceAdmin` object. The elaboration of these requirements is shown for the `SimpleService` example below, where the comments highlight the extra code that the programmer must add.

```
public class SimpleService implements SimpleServiceInterface {
    // include admin. object attribute
    public ServiceAdmin sa = null;
    public AnotherServiceInterface asi = null;
    ServiceTemplate template = null;
    ServiceRegistrar registrar = null;
    int count = 0;

    public SimpleService() {
        super();
        // construct admin. object
        sa = new ServiceAdmin(this);
    }

    public void findAnotherService() throws RemoteException {
        Class[] classes = new Class[]
                          {AnotherServiceInterface.class};
        template = new ServiceTemplate(null, classes, null);

        try {
```

```
                asi = (AnotherServiceInterface)
                                registrar.lookup(template);
                // add each remote interface reference
                // to admin. object's 'bindings'
                sa.bindings[count++] = asi;
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
            }
        }

        // implement getAdmin
        public Object getAdmin() {
            return sa;
        }
    }
```

Probe Service

Probe instrumentation services are responsible for determining the dependencies of a target application service and the code below shows how a probe uses the `getAdmin` method to access the immediate dependencies for a target application service. When a probe has access to these 'immediate' dependencies, it may visit each binding to see if the remote object, associated with the binding, has its own dependencies. This allows a complete graph to be built up, which represents all the dependencies (immediate, secondary, tertiary, etc.) associated with a target service.

```
public class ProbeInstrument extends EventInstrument {
    static final int NNODES = 100;
    static final int NEDGES = 100;
    ServiceAdmin sa = null;
    Object target = null;
    Object root = null;
    Object[] objs = null;

    public ProbeInstrument(SimpleServiceInterface ssi) {
        super();
        this.target = ssi;
    }

    interface Visitor {
        void visit(Object obj) throws RemoteException;
    }

    public class DynamicDependencyDigraph extends DynamicObject {
        ServiceAdmin sa = null;
        Object[] objs = null;

        class Node {
            Object obj;
            String label;
            void accept(Visitor visitor) throws RemoteException {
                visitor.visit(obj);
```

```
    }
}

class Edge {
    Node from;
    Node to;
    double len;
}

Node nodes[] = new Node[NNODES];
Edge edges[] = new Edge[NEDGES];
int nnodes, nedges, len = 0;

public DynamicDependencyDigraph() {
    super();
}


Node findNode(Node n) {
    for (int i = 0 ; i < nnodes ; i++) {
        if (nodes[i].equals(n)) {
            return nodes[i];
        }
    }
    return addNode(n);
}

Node addNode(Node n) {
    Node node = new Node();
    node.obj = n.obj;
    node.label = n.label;
    nodes[nnodes] = node;
    nnodes++;
    return node;
}

Node fromNode, toNode = null;

void addEdge(Node from, Node to, int len) {
    Edge e = new Edge();
    e.from = findNode(from);
    e.to = findNode(to);
    e.len = len;
    edges[nedges++] = e;
}

class DigraphVisitor implements Visitor {

    public void visit(Object obj) throws RemoteException {
        try {
            fromNode = new Node();
            fromNode.obj = obj;
            fromNode.label = obj.getClass().getName();
            sa = (ServiceAdmin)obj.getAdmin();
            if (sa.getBindings().length > 0)
                objs = sa.getBindings();
        } catch(java.rmi.RemoteException e) {
```

```
                    System.err.println(e.toString());
                }
            }
        }

        public void draw() {
            try {
                fromNode = new Node();
                fromNode.obj = root;
                fromNode.label = root.getClass().getName();
                Visitor visitor = new DigraphVisitor();
                int i = 0;
                while (objs.length > 0) {
                    toNode = new Node();
                    toNode.obj = objs[i];
                    toNode.label = objs[i].getClass().getName();
                    addEdge(fromNode, toNode, len);
                    visitor.visit(toNode.obj);
                    toNode.accept(visitor);
                    i++;
                }
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
            }
        }
    }

    public void getAdmin() throws RemoteException {
        try {
            sa =
        (ServiceAdmin)((SimpleServiceInterface)target).getAdmin();
            root = sa.obj;
            objs = sa.getBindings();
            DynamicDependencyDigraph graph =
                            new DynamicDependencyDigraph();
            graph.draw();
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
    }
}
```

The `ProbeInstrument` class extends the `EventInstrument` class (to be considered shortly) so that that it may receive notification of any changes in bindings from the application service's lookup service. The `ProbeInstrument` class also contains a `DynamicDigraph` inner-class, which extends the `DynamicObject` class. The structure of these classes was considered previously in chapter 7 and the code above shows the actual implementation of the `DynamicDigraph` class. The `DynamicDigraph` class contains its own inner-classes of `Node`, `Edge` and the important `DigraphVisitor` class. The `DigraphVisitor` class implements the visitor design pattern so that it may visit the

remote objects associated with each binding to recursively check out their own bindings.

The *visitor* design pattern, described further in [53], is often used to separate the structure of an object collection from the operations performed on that collection. For example, it can separate the parsing logic in a compiler from the code generation logic. By keeping the two separate, different code generators may then be used with relative ease. In this instance, the visitor is used to separate the traversal of the digraph from the logic used to actually build the graph. The visitor pattern also specifies how iteration occurs over the object structure.

To implement a visitor design pattern a visitor interface is first specified, which provides the method signatures of the `visit` methods that will visit the objects in a collection. Each object in the collection has an `accept` method that takes a `Visitor` object as an argument. The `accept` method of an object's class calls back the `visit` method for its class. A concrete `Visitor` class can then be written, which combines the `visit` and `accept` methods to visit all the objects in the collection and performs some particular operation on each object.

The `DigraphVisitor` class is the concrete visitor class, which is used by the `draw` method of the `DynamicDigraph` class. The `draw` method is responsible for building an in-memory representation of the dependency digraph as a graph of nodes interconnected by edges. The `draw` method contains a while loop, which joins nodes and edges together and visits the remote object associated with each node via calls to `visitor.visit(toNode.obj)` and `toNode.accept(visitor)`.

The main entry point for the `ProbeInstrument` is its own `getAdmin` method, which provides access to the `ServiceAdmin` object associated with a target application service. The target dependent service itself is added to the root of the dependency digraph. The immediate bindings are then obtained and stored in the `objs` array. A `DynamicDigraph` object is then created and its `draw` method is invoked, which initiates the recursive descent into the `objs` array, using the visitor pattern, to determine any secondary, tertiary etc. dependencies.

This approach for determining dependencies does impose additional effort on the applications programmer in that they are required to implement an additional interface and include a `ServiceAdmin` object. However, the approach does provide a consistent means for dealing with dependencies, which are inherently complex to determine with a wholly unobtrusive approach.

The above code shows how dependencies may be determined to provide an instantaneous snapshot of the remote objects on which a component depends. However, as mentioned in chapters 5 and 7, these dependencies may well change over a period of time. In order to monitor such dynamic dependencies, probes are equipped with event handling capabilities, which will be mentioned further in the next section.

Having considered the main programming constructs used to access runtime information from application-level services, we may proceed to look at the implementation of the remaining instantiable instrumentation services (logger, gauge, analyzer and monitor). These services are all descendants of the `BaseInstrument` class and have access to its state variables and methods. As considered below, the instantiable instrumentation services and their infrastructure classes also introduce additional functionality of their own.

8.2.6 Instantiable Instrumentation Services

The above sections have considered the implementation of the `BaseInstrument` class through a series of incremental stages. The above sections have also considered the implementation of the `DirectInstrument` and `ProbeInstrument` classes. This section considers the various infrastructure classes within the class hierarchy (Figures. 5.1 and 7.17) and goes on to describe the implementation of the remaining instantiable instrumentation services.

We begin with the `IndirectInstrument` class, shown below. `IndirectInstrument` is a simple class used to represent instrumentation services that are not directly attached to application services, but may receive information from a direct instrumentation service.

```
public class IndirectInstrument extends BaseInstrument {
    public IndirectInstrument() {
        super();
    }
}
```

Next, we have the `StaticInstrument` class, shown below, which is the parent class for the static instruments of logger, gauge and analyzer. There are two "flavours" of `StaticInstrument`, one that extends `DirectInstrument` and one that extends `IndirectInstrument` (`IStaticInstrument`). The code below represents the `DirectInstrument` flavour, but the code is identical for both classes.

```
public class StaticInstrument extends DirectInstrument {

    public Object param = null;
    public String name = null;
    public int type = 0;
    public int index = 0;
    public Thread thread = null;

    public StaticInstrument() {
        super();
        thread = new Thread(task, "Instrument Task");
    }

    public void setParam(Object object, String name, int type,
                                    int index) {
        param = object;
        this.name = name;
        this.type = type;
        this.index = index;
    }

    Runnable task = new Runnable() {
        public void run() {
            runInstrument();
        }
    };

    public void runInstrument() {
    }

    public void start() {
        thread.start();
    }

    public void stop() {
        thread = null;
    }
}
```

The `StaticInstrument` class implements a `setParam` method, which "primes" the class with the parameters that may later be used by any of its subclasses to invoke the `reflect` method considered previously. The `StaticInstrument` class also contains a Java thread and `start` and `stop` methods to start and stop the thread respectively. The method that the thread runs is `runInstrument`, which is left empty so that it may be overridden by a subclass to provide appropriate measurement/monitoring code.

It may seem wasteful to create a thread object for each instance of the `StaticInstrument` class. However, if a subclass instrumentation service is required to run in single-pass mode (i.e. measure or monitor a single value), then the thread object need not be started and it will be destroyed by the garbage collector, when the subclass instance itself is destroyed by the garbage collector. The incorporation of the thread within the `StaticInstrument` class, alleviates the need to replicate threads and their associated code within the subclasses of `StaticInstrument` (i.e. logger, gauge and analyzer instrumentation services).

Logger Service

The `LoggerInstrument` class, shown below, implements a direct logger that may be run in single-pass mode or thread mode according to the value of `mode`. The `LoggerInstrument` class contains methods that provide access to an `OutputStream` to which information is to be logged. The `log` method actually performs the logging, by acquiring the parameter of interest using the `reflect` method and writing its value to the `OutputStream`. If the logger is started in thread mode, the thread is started and the overridden `runInstrument` method is invoked to repeatedly log the parameter.

```
public class LoggerInstrument extends StaticInstrument {
    public PrintWriter out = null;
    public OutputStream stream = null;
    boolean mode = true;

    public LoggerInstrument(boolean mode) {
        super();
        this.mode = mode;
    }

    public void open() {
        out = new PrintWriter(stream);
    }
```

```
    public void close() {
        out.close();
    }

    public void setStream(OutputStream stream) {
        this.stream = stream;
    }

    public OutputStream getStream() {
        return stream;
    }

    public void log() {
        if (mode) {
            param = reflect(param, name, type, index);
            out.write(param.toString());
        }
        else {
            start();
        }
    }

    public void runInstrument() {
        while (true) {
            param = reflect(param, name, type, index);
            out.write(param.toString());
        }
    }
}
```

If the logger was an indirect logger it would not call the `reflect` method. Instead, it would call the `read` method of another direct instrumentation service to access `param`.

Gauge Service

The `GaugeInstrument` class, shown below, implements a direct gauge, which is similar the logger class above. The main difference is that a gauge provides a comparative measurement against low/high values. The low/high values are set using the `setLimits` method and each gauged value is stored in a `Measurement` object. As for an indirect logger, an indirect gauge would call the `read` method of a direct instrumentation service rather than the `reflect` method.

```
public class GaugeInstrument extends StaticInstrument {
    boolean mode = true;
    Object high, low = null;

    public GaugeInstrument(boolean mode) {
        super();
        this.mode = mode;
    }
```

```
    public class Measurement {
        public Object low;
        public Object reading;
        public Object high;
    }

    public Measurement measurement = null;

    public void setLimits(Object low, Object high) {
        this.low = low;
        this.high = high;
    }

    public Measurement getMeasurement() {
        return measurement;
    }

    public void gauge() {
        if (mode) {
            param = reflect(param, name, type, index);
            measurement = new Measurement();
            measurement.low = low;
            measurement.reading = param;
            measurement.high = high;
        }
        else {
            start();
        }
    }

    public void runInstrument() {
        while (true) {
            param = reflect(param, name, type, index);
            measurement = new Measurement();
            measurement.low = low;
            measurement.reading = param;
            measurement.high = high;
        }
    }
}
```

Analyzer Service

The `AnalyzerInstrument` class, shown below, implements a direct analyzer, which is similar the previous logger and gauge classes. The main difference is that an analyzer uses a computation object, `computeObject` to perform some form of computational analysis on the parameter of interest. The computation object is provided by the management agent which is using the analyzer and this object must implement a `compute` method that is used to perform the analysis. As for indirect loggers and

gauges, an indirect analyzer would call the `read` method of a direct instrumentation service rather than the `reflect` method.

```
public class AnalyzerInstrument extends StaticInstrument {
    boolean mode = true;
    public Object computeObject;

    public AnalyzerInstrument(boolean mode) {
        super();
        this.mode = mode;
    }

    public void setComputeObject(Object compute) {
        this.computeObject = computeObject;
    }

    public Object getComputeObject() {
        return computeObject;
    }

    public void analyze() {
        if (mode) {
            param = reflect(param, name, type, index);
            computeObject.compute(param);
        }
        else {
            start();
        }
    }

    public void runInstrument() {
        while (true) {
            param = reflect(param, name, type, index);
            computeObject.compute(param);
        }
    }
}
```

The `DynamicInstrument` class, shown below, is similar to the `StaticInstrument` class except that there is no need for a `setParam` method, because the dynamic instrumentation services of probe and monitor work directly on the remote object to which they are attached.

```
public class DynamicInstrument extends DirectInstrument {

    public Thread thread = null;

    public DynamicInstrument() {
        super();
        thread = new Thread(task, "Instrument Task");
    }
```

```
        Runnable task = new Runnable() {
            public void run() {
                runInstrument();
            }
        };

        public void runInstrument() {
        }

        public void start() {
            thread.start();
        }

        public void stop() {
            thread = null;
        }
    }
```

The `AsynchronousInstrument` class, shown below, is a simple class, which is used to simply distinguish between asynchronous and synchronous instruments.

```
    public class AsynchronousInstrument extends DynamicInstrument {

        public AsynchronousInstrument() {
            super();
        }
    }
```

The `SynchronousInstrument` class contains a `RemoteException` attribute, which is set by the `MMonitor` subclass if an RMI call should fail.

```
    public class SynchronousInstrument extends DynamicInstrument {
        RemoteException exception = null;
        Timer timer = null;

        public SynchronousInstrument() {
            super();
            init();
        }

        void setRemoteException(RemoteExcpetion e) {
            exception = e;
        }

        RemoteException getRemoteException() {
            return exception;
        }

    }
```

The `EventInstrument` class, shown below, is used by those instrumentation services that are required to receive event notifications from application services or lookup services (probes and event monitors). The `notify` method was declared previously to allow instruments to receive notification of events from other instrumentation services. The `EventInstrument` class overrides this `notify` so that, as well as receiving instrumentation event notifications, it may receive notifications from application services and lookup services.

```
public class EventInstrument extends AsynchronousInstrument {
    public RemoteEvent event = null;

    public EventInstrument() {
        super();
    }

    public EventRegistration register(Object object) {
        return this.addRemoteEventListener(object, new
                              MarshalledObject("Event Object"));
    }

    public void notify(RemoteEvent event)
                 throws UnknownEventException, RemoteException {
        try {
            Object object =
               (Object)event.getRegistrationObject().getSource();
            // deal with instrumentation event
            if (object instanceof BaseInstrumentInterface) {
                if (object != this) {  // events from
                                       // other instruments
                    switch ((int)event.getID()) {
                    case 0:
                        //register
                        break;
                    case 1:
                        //unregister
                        group.remove(object);
                        proxies.remove(object.getProxy());
                        break;
                    case 2:
                        //attach
                        break;
                    case 3:
                        //detach
                        proxies.remove(object.getProxy());
                        break;
                    case 4:
                        //join
                        break;
                    case 5:
                        //unjoin
```

```
                        group.remove(object);
                        proxies.remove(object.getProxy());
                        break;
                    default:
                        System.err.println("Unknown Event");
                        break;
                    }
                }
                else { // events from ourself
                    switch ((int)event.getID()) {
                    case 0:
                        //register
                        state[0] = true;
                        break;
                    case 1:
                        //unregister
                        state[0] = false;
                        break;
                    case 2:
                        //attach
                        state[1] = true;
                        break;
                    case 3:
                        //detach
                        state[1] = false;
                        break;
                    case 4:
                        //join
                        state[2] = true;
                        break;
                    case 5:
                        //unjoin
                        state[2] = false;
                        break;
                    default:
                        System.err.println("Unknown Event");
                        break;
                    }
                }
            }
            else { // events from application services
                   // and lookup services
                this.event = event;
            }
        } catch(IOException e) {
            throw new UnknownEventException("IOException: " +
                                            e.getMessage());
        } catch(ClassNotFoundException e1) {
            throw new
                     UnknownEventException(
                             "ClassNotFoundException: " +
                              e1.getMessage());
        }
    }
}
```

The `EventInstrument` class implements a `register` method, which uses the `addRemoteEventListener` method (of `BaseInstrument`) to add a new event listener for the object of interest. When a new listener has been added, notifications of any events are sent to the `notify` method defined in `EventInstrument`. This `notify` method retains the same code as for `BaseInstrument`, but appends the extra lines to receive non-instrumentation events. These events may be used by probes to rebuild dependency digraphs or repackaged by event monitors to create dynamic event objects, as will be considered shortly.

The implementation of a probe was considered previously through the `ProbeInstrument` class. As mentioned previously, the `ProbeInstrument` class can be used to provide an instantaneous snapshot of the remote objects on which a component depends. However these dependencies may well change over a period of time and probes may be required to rebuild the dependency digraph. This functionality is provided by the above `notify` method of the `EventInstrument` class. The probe may register to receive event notifications from the lookup service with which the probe's application service is registered. Such events may be checked to see if they affect the probe's application service and if so, be used to force the probe to rebuild the dependency digraph.

Event Monitor Service

An event monitor serves a different purpose, when `EventInstrument`'s `notify` method receives event notifications. An event monitor will repackage an event into a dynamic event object, which may then be inspected by a management agent. An event monitor is implemented through the `EMonitorInstrument` class shown below. The `EMonitorInstrument` class simply sets the fields of a `DynamicEventObject` using the `event` object inherited from the `EventInstrument` class. Similar to the previous static instruments, `EMonitorInstrument` may be run in single pass mode, or in thread mode. In thread mode the `runInstrument` method is used to repackage a series of events while the thread remains active.

```
public class EMonitorInstrument extends EventInstrument {
    boolean mode = true;

    public EMonitorInstrument(boolean mode) {
```

```
        super();
        this.mode = mode;
    }

    public class DynamicEventObject extends DynamicObject {
        public long id;
        public Object source;
        public long seqNum;
    }

    public DynamicEventObject eventObject = null;

    public Object getDynamicEventObject() {
        return eventObject;
    }

    public void monitor() {
        if (mode) {
            eventObject = new DynamicEventObject();
            eventObject.id = event.getID();
            eventObject.source =
                    event.getRegistrationObject().getSource();
            eventObject.seqNum = event.getSequenceNumber();
        }
        else {
            start();
        }
    }

    public void runInstrument() {
        while (true) {
            eventObject = new DynamicEventObject();
            eventObject.id = event.getID();
            eventObject.source =
                     event.getRegistrationObject().getSource();
            eventObject.seqNum = event.getSequenceNumber();
        }
    }
}
```

Method Invocation Monitor Service

The `MMonitorInstrument` class, shown below, is used to intervene on method invocations made upon application services. The `MethodInstrument` class overrides the `invoke` method so that method invocations, made on application services, may be repackaged as dynamic method invocation objects.

```
public class MMonitorInstrument extends SynchronousInstrument {
    public Method method = null;
    public Object returnVal = null;
    public Object[] params = null;

    boolean mode = true;
```

```
public MMonitorInstrument(boolean mode) {
    super();
    this.mode = mode;
}

public class DynamicMethodInvocationObject extends
                                                DynamicObject {
    public String clientAddress;
    public Object server;
    public Object client;
    public Object[] params;
    public Object returnVal;
}

public DynamicMethodInvocationObject invocationObject = null;

public Object getDynamicMethodInvocationObject() {
    return invocationObject;
}

public void monitor() {
    if (mode) {
        invocationObject =
                  new DynamicMethodInvocationObject();
        invocationObject.clientAddress = obj.getClientHost();
        invocationObject.server = obj;
        invocationObject.params = params;
        invocationObject.returnVal = returnVal;
    }
    else {
        start();
    }
}

public void runInstrument() {
    while (true) {
        invocationObject =
                   new DynamicMethodInvocationObject();
        invocationObject.clientAddress = obj.getClientHost();
        invocationObject.server = obj;
        invocationObject.params = params;
        invocationObject.returnVal = returnVal;
    }
}

public Object invoke(Object proxy, Method meth, Object[] args)
       throws Throwable {
    System.out.println(meth.getName());
    try {
      method = meth;
      params = args;
      returnVal = meth.invoke(obj, args);
      return returnVal;
  } catch (InvocationTargetException e) {
      setRemoteException(e.getTargetException());
  }
}
```

```
}
```

The `MMonitorInstrument` class, shown below, implements a method invocation monitor. This class simply sets the fields of a `DynamicMethodInvocationObject` using the `method` object (inherited from the `MethodInstrument` class) and the object, `obj`, on which the method was invoked (inherited from the `DProxy` class). Similar to the previous static instruments, `MMonitorInstrument` may be run in single pass mode, or in thread mode. In thread mode the `runInstrument` method is used to repackage a series of method invocations while the thread remains active.

The previous instantiable instrumentation services may be used by management agents to measure and monitor the runtime behaviour of application services. However, a need may arise to perform more specific logging or monitoring activities. One example of such an activity occurs when a programmer has already primed the classes of an application's code using the *log4j* package. Another example occurs when more specific information is required of the devices attached to a network. The architecture does provide some limited support for such activities through its support for the use of third-party applications.

8.3 Third-party Software Support

The final part of the implementation description concerns the provision of interfaces through which the functionality provided by third-party software may be utilized. It would prove extremely difficult, if not impossible to provide support for all third-party logging or monitoring software. With this in mind, we focus attention on two important software applications, namely the Jakarta *log4j* logging package [82]and the *AdventNet* SNMP package [102]. The intention is not to directly introduce new functionality into the architecture, but to demonstrate the relative ease of providing interfaces to external third-party software, which can be used to gather information relating to more specific aspects of an application.

The main entry point for third-party software is the `ThirdPartyAppInterface` interface shown below. The use of this interface essentially separates any third-party applications from the instrumentation architecture, but allows any management agent,

which is using the architecture's API, to invoke the third-party applications. Of course as each new third-party application is added, the interface needs to be changed and any implementation classes associated with each new entry must also be coded.

```
public interface ThirdPartyAppInterface {

    public interface SnmpAppInterface {
        public void runSnmpApp(String app, String[] params);
    }

    public interface Log4JAppInterface {
        public void log(int mode);
    }
}
```

The interface defines two inner interfaces, which each specify methods that facilitate the execution of SNMP and *log4j* applications. The classes that implement these interfaces are considered in the sections below.

8.3.1 SNMP Support

The Simple Network Management Protocol (SNMP) is considered more thoroughly in [103, 104]. This section provides a brief overview of SNMP and considers how the *AdventNet* SNMP package [102] may be incorporated within the instrumentation architecture.

SNMP is a network management protocol based on the manager/agent model. SNMP facilitates communication between an SNMP agent (a managed device, e.g. a computer or a router), and an SNMP manager (a human) or management application (a network management software application) as shown in Figure 8.4. Communication is via SNMP Protocol Data Units (PDUs). The manager and agent use a Management Information Base (MIB) and a relatively small set of commands to exchange information. The MIB is organized in a tree structure with individual variables, such as point status or description, being represented as leaves on the MIB branches. A long numeric tag or object identifier (OID) is used to distinguish each variable uniquely in the MIB and in SNMP messages.

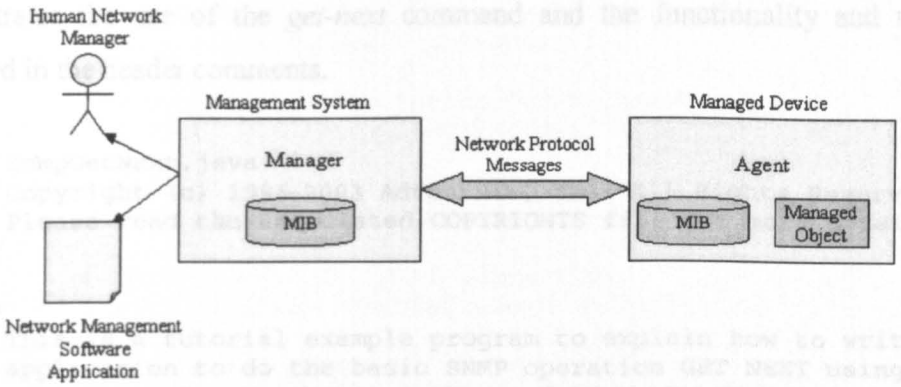


Figure 8.4: overview of SNMP

SNMP uses five basic command messages: *get*, *get-next*, *get-response*, *set*, and *trap* to communicate between the manager and the agent. The small number of commands used is one of the reasons why SNMP is regarded as "simple". There are essentially four kinds of operations, which are permitted between managers and agents (managed device).

- The manager can perform a *get* (or read) to obtain information from the agent about an attribute of a managed object.
- The manager can perform a *get-next* to do the same for the next object in the tree of objects on the managed node.
- The manager can perform a *set* (or write) to set the value of an attribute of a managed object.
- The agent can send a *trap*, or asynchronous notification, to the manager telling it about some event on the managed device.

The *AdventNet* Java SNMP package [102] provides an API for the development of SNMP managers as Java management applets or Java management applications. The package provides a series of SNMP beans, which implement the five basic commands and provide additional utilities, such as polling an agent. These beans and utilities may be incorporated into Java applets or applications to provide Java-based network management capabilities. Several examples are provided in the *AdventNet* SNMP API tutorials and the code for two of these examples is repeated below. The first example

demonstrates the use of the *get-next* command and the functionality and usage is described in the header comments.

```
/*
 * SnmpGetNext.java
 * Copyright (c) 1996-2003 AdventNet, Inc. All Rights Reserved.
 * Please read the associated COPYRIGHTS file for more details.
 */

/**
 * This is a tutorial example program to explain how to write an
 * application to do the basic SNMP operation GET NEXT using
 * com.adventnet.snmp.beans package of AdventNetSNMP api.
 *
 * The user could run this application by giving the following
 * usage.
 *
 * java SnmpGetNext hostname OID [OID]
 *
 * where
 *    hostname is the RemoteHost (agent).The Format is string
 *    without doubleqoutes/IpAddress.
 *    OID is the Object Identifier.
 *    Multiple OIDs can also be given. The entire OID can be
 *    given or it can be given in the form of 1.1.0. If the oid
 *    is not starting with a dot (.) it will be prefixed by
 *    .1.3.6.1.2.1 . So the entire OID of 1.1.0 will become
 *    .1.3.6.1.2.1.1.1.0 .
 *
 * Example usage:
 *
 * java SnmpGetNext adventnet 1.1.0 1.2.0 1.3.0 1.4.0
 *
 */

import com.adventnet.snmp.beans.*;

public class SnmpGetNext {

    public static void main(String args[]) {

        if( args.length < 2)
        {
            System.out.println(
                 "Usage : java SnmpGetNext hostname OID ");
            System.exit(0);
         }

        // Take care of getting the hostname and the OID

        String remoteHost = args[0];
        String OID = args[1];

        // Instantiate the SnmpTarget bean
        SnmpTarget target = new SnmpTarget();
```

```
        //set host and other parameters
        target.setTargetHost(remoteHost);

        String oids[] = new String [args.length - 1];
        for (int i=1; i<args.length; i++) oids[i-1] = args[i];

         // multiple OID's can be processed
        target.setObjectIDList(oids);

        // do the SNMP GET NEXT  operation
        String result[] = target.snmpGetNextList();

        // print the results
        for (int i=0;i<oids.length;i++) {
            System.out.println("OBJECT ID:" +
                      target.getObjectID(i));
            System.out.println("Response: " + result[i]);
        }
        System.exit(0);
    }
}
```

The second example demonstrates the polling of a remote host to any remote events traps that it may generate. Again, the functionality and usage is described in the header comments.

```
/* SnmpPolling.java
 * Copyright (c) 1996-2003 AdventNet, Inc. All Rights Reserved.
 * Please read the associated COPYRIGHTS file for more details.
 */

/**
 * This is a tutorial example program to explain how to write an
 * application to do the polling operations using
 * com.adventnet.snmp.beans package of AdventNetSNMP api.
 *
 * The user could run this application by giving the following
 * usage.
 *
 * java SnmpPolling hostname OID
 *
 * where
 *
 *    hostname is the RemoteHost (agent).The Format is string
 *    without double qoutes/IpAddress.
 *    OID is the Object Identifier.
 *    The entire OID can be given or it can be given in the form
 *    of 1.1.0. If the oid is not starting with a dot (.) it will
 *    be prefixed by .1.3.6.1.2.1 . So the entire OID of 1.1.0
 *    will become .1.3.6.1.2.1.1.1.0 .
 *
 * Example usage:
 *
```

```
 * java SnmpPolling adventnet 1.1.0
 *
 */

import com.adventnet.snmp.beans.*;


public class SnmpPolling implements ResultListener  {

    SnmpPoller poller = new SnmpPoller();

    public static void main(String args[]) {

        if( args.length < 2)
        {
             System.out.println(
                    "Usage : java SnmpPolling hostname OID ");
             System.exit(0);
        }

        // Take care of getting the hostname and the OID
        String remoteHost = args[0];
        String OID = args[1];

        SnmpPolling polling = new SnmpPolling();

        //set host and other parameters
        polling.poller.setTargetHost(remoteHost);
        polling.poller.setObjectID(OID);
        polling.poller.setPollInterval(1);
        polling.poller.addResultListener(polling);
    }


    public void setNumericResult(long l){
    }

    public void setResult(ResultEvent result){
        try {
            System.out.println(result.getStringValue());
        } catch (DataException de) {
            System.out.println("Error in getting agent data: " +
                   d + result.getErrorString());
        }
    }

    public void setStringResult(String s){
    }
}
```

Either of these SNMP applications may be run in its own JVM through the class `SnmpApp`, which implements the `runSnmpApp` method specified in the `SNMPAppInterface`. The `runSnmpApp` method starts a new JVM using the

`Runtime.getRuntime().exec(cmd)` command, which creates a new JVM environment for the Java command line specified in `cmd`.

```
public class SnmpApp implements
            ThirdPartyAppInterface.SnmpAppInterface {

    public SnmpApp() {
    }

    public void runSnmpApp(String app, String[] params) {
        try {
            String cmd = "java -cp " + app + params;
            Process p = Runtime.getRuntime().exec(cmd);
            try {
               int exitCode = p.waitFor();
               if (exitCode != 0)
               System.out.println("Exec: failed to launch SmtpApp"
                                  + app);
            }
            catch(InterruptedException e) {
               System.err.println("Error launching SnmpApp"
                                  + app);
               e.printStackTrace();
            }
        } catch(Exception e1) {
            System.err.println("Failed to launch SnmpApp" + app);
            e1.printStackTrace();
        }
    }
}
```

For example, a management agent may create an instance of `SnmpApp` and run the command to poll a remote host, as below.

```
ThirdPartyAppInterface.SnmpAppInterface sai = new SnmpApp();
sai.runSnmpApp("SnmpPolling", "150.204.48.41", "1.0.1");
```

8.3.2 *log4j* Support

log4j is an open source logging tool developed under the Jakarta Apache project [82]. *log4j* provides a set of APIs that allows programmers to write log statements in their code and configure them externally, using property files. There are three aspects of *log4j*: *logger*, *appender*, and *layout*. A logger logs to an appender in accordance with a particular layout (or style). Each class in an application may have an individual logger or may use a common logger. *log4j* provides a root logger that all loggers inherit from and if a class does not have access to a logger, it may use the root logger by calling `Logger.getRootLogger()`, although this is not recommended.

The preferred way to create a logger and use it for logging a group classes, is to use the static method of the `Logger` class. This may be called as `Logger.getLogger(clazz)`, which retrieves a logger by using the name of the class `clazz`, within the group. If the particular logger has not already been created it will be created afresh, and there will always be one instance of this logger in the JVM associated with the class. The loggers for a group of classes are arranged hierarchically in accordance with the class hierarchy associated with the group.

Loggers need to know where to send requests for logging and this is where the appenders feature. *log4j* supports writing to files (`FileAppender`), to the console (`ConsoleAppender`), to databases (`JDBCAppender`), to NT event logs (`NTEventLogAppender`), to SMTP servers (`SMTPAppender`), to remote servers (`SocketAppender`), and others. An appender defines the properties of the logging target to *log4j*.

log4j provides five levels of logging: `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL`. Each logger in *log4j* is assigned a level. If a level is not initially assigned to a logger, *log4j* automatically assigns the level of the logger to that of the parent logger, which may be another logger or the root logger. The root logger always has a default level assigned, which is `DEBUG` so that all loggers are guaranteed to have this level. A log request made from within an application, using a particular logger, will be sent to a corresponding appender only if the level of the log request is greater than or equal to the level of the logger itself. This is a very important rule, which lies at the core of *log4j*'s capabilities.

Logging code could be added to the `MyClass` example considered previously, as shown below. The revised `MyClass` includes the code to create a logger and two log statements. The first statement is at the `DEBUG` level to log the parameters used in the `larger` method. The second log statement is at the `ERROR` level to check for null valued strings in the `toString` method.

```
// import log4j Logger package
import org.apache.log4j.Logger;

public class MyClass {

    public String firstName = null;
    public String lastName  = null;
```

```
    public int    idNumber  = 0;
    public int[] array = new int[]{5,6,7};
    public Logger log = null;

    public MyClass(String fname, String lname, int id) {
        this.firstName = fname;
        this.lastName  = lname;
        this.idNumber  = id;
        log = Logger.getLogger(MyClass.class);
    }

    public MyClass() {
        this("Denis", "Reilly", 12345);
        log = Logger.getLogger(MyClass.class);
    }

    public boolean larger(int a, int b) {
        log.debug("a: " + a + " b: " + b);
        if (a > b)
            return true;
        else
            return false;
    }

    public String toString() {
        if (lastName == null || firstName == null)
            log.error("Null value");
        return (lastName + ", " + firstName + " [" +
                    idNumber + "]");
    }
}
```

To incorporate *log4j* functionality within the instrumentation architecture we use a similar approach to that of the SNMP application. *log4j* functionality is provided through the `Log4JApp` class, which implements the `log` method specified in the `Log4JAppInterface`. The `log` method will get a logger for the specified class, `MyClass` and set the logging level according to the mode parameter.

Of course, this approach assumes that `MyClass` is associated with an individual logger. Assuming it is, then any inner-classes or subclasses of `MyClass` may also be logged as they will inherit the logger of `MyClass`. One drawback to this approach of using *log4j* is that the `Log4JApp` object must run in the same JVM as the `MyClass` object (typically it must be on the same computer) that is being logged.

```
public class Log4JApp implements
            ThirdPartyAppInterface.Log4JAppInterface {
    static final int DEBUG = 0;
    static final int INFO = 1;
    static final int WARN = 2;
```

```
    static final int ERROR = 3;
    static final int FATAL = 4;
    Class clazz = null;

    public Log4JApp(Class clazz) {
        this.clazz = clazz;
    }

    public void log(int mode) {
        Logger log = Logger.getLogger(clazz);
        switch (mode) {
        case DEBUG:
            log.debug();
            break;
        case INFO:
            log.info();
            break;
        case WARN:
            log.warn();
            break;
        case ERROR:
            log.error();
            break;
        case FATAL:
            log.fatal();
            break;
        default:
            System.err.println("Invalid logging level");
            break;
        }
    }
}
```

A management agent may create an instance of `Log4JApp` and set the logging level of `MyClass` to `INFO` as shown below.

```
ThirdPartyAppInterface.Log4JAppInterface l4jai =
            new Log4JApp(MyClass.class);
l4jai.log(INFO);
```

log4j is mostly configured using an external configuration file although the API provides classes for configuring the *log4j* system through code as well. For this relatively simple *log4j* interface configuration files are used, which requires that an appropriate configuration file is in place. A simple configuration file that may be used to log information for `MyClass` to the `System.out` stream is shown below.

```
# Set root category priority to DEBUG and its only appender to A1.
log4j.rootCategory=DEBUG, A1
# A1 is set to be a FileAppender which outputs to System.out.
log4j.appender.A1=org.log4j.FileAppender
log4j.appender.A1.File=System.out
# A1 uses PatternLayout.
log4j.appender.A1.layout=org.log4j.PatternLayout
```

```
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x -
   %m%n
```

The `log4j.appender.A1.File` line specifies that all logging information is sent to `System.out`. The final line of the configuration file specifies the format used for the logger output. The resulting output according to this format is shown below. The output shows: the date and time, the line number in the class file, the method, the logging level, the class being logged and the result of any specific logging code added by the programmer.

```
2003-09-02 14:07:41, 24 [larger]   DEBUG MyClass - a: 2 b: 1.
2003-09-02 14:07:41, 33 [toString] ERROR MyClass - Null value.
```

Another configuration file that may also be used to log information for `MyClass` to the `System.out` stream is shown below. This file uses multiple appenders: the first appender, which logs information to `System.out` and a second appender, which directs output to the `example.log` file. The final appender statements specify that `example.log` will be rolled over when it reaches 100 KB. When rollover occurs, the old version of `example.log` is automatically moved to `example.log.1`.

```
log4j.rootCategory=debug, stdout, R
log4j.appender.stdout=org.log4j.FileAppender
log4j.appender.stdout.File=System.out
log4j.appender.stdout.layout=org.log4j.PatternLayout
# Pattern to output the caller's file name and line number.
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) -
   %m%n
log4j.appender.R=org.log4j.RollingFileAppender
log4j.appender.R.File=example.log
log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

Note that to obtain those different logging behaviours, it is not necessary to recompile the code that is being logged. Other logging options include logging information to a Unix/Linux `syslog` daemon or redirection of all logging output to an NT Event logger. Logging messages may even be forwarded to a remote *log4j* server, which would log according to local server policy.

8.4 Chapter Summary

This chapter has described the implementation of the instrumentation architecture using a combination of Java and Jini middleware technology. The chapter has considered: the implementation of the basic instrument operations; the main programming constructs used within the implementation; the implementation of instantiable instrumentation services. Management agents may instantiate any of these instrumentation services and use their API to measure/monitor application services.

The architecture has applied several programming constructs in a novel fashion to develop instrumentation services that can measure and monitor application components unobtrusively. In particular, Java's reflection API was used to acquire structural class information and runtime (behavioural) information relating to an object; Java's dynamic proxy class was used to facilitate the attachment of instrumentation services; Jini's Administrable interface was used to help expose component bindings from which an applications dependencies may be determined.

The chapter has also considered how third-party software applications may be used from within the architecture to perform more specific logging (*log4j*) and monitoring (SNMP) tasks. The chapter concludes the second part and main contribution of the thesis. The final part of the thesis (chapters 9 and 10) consider the application of the instrumentation architecture, for measuring and monitoring applications (chapter 9) and drawing of overall conclusions (chapter 10).

Chapter 9

Instrumenting Distributed Applications

Having considered the analysis (chapter 6), modelling (chapter 7) and implementation (chapter 8) of dynamic instrumentation services we are now in a position to put theory into practice. This chapter intends to do so by demonstrating how the instrumentation services may be used to measure and monitor several distributed applications. The chapter begins by introducing the graphics-based test harnesses that are used to demonstrate the instrumentation architecture and its services. The chapter then describes four instrumentation case-studies, which are intended to demonstrate how the architecture may be used.

In particular, the following case studies are considered:

- Basic logging and method invocation monitoring for a simple distributed application.
- The determination of dynamic dependencies for a multi-service application.
- Analysis of a non-trivial distributed application to determine client-server access patterns
- The use of instrumentation to assist the design of a distributed application.

The chapter concludes with a qualitative assessment of the performance overhead introduced by the instrumentation services.

9.1 Instrumentation Test Harnesses

The instrumentation architecture is intended to be used by a management agent, which is responsible for the management and control of a distributed application. This thesis is not so much concerned with the management and control aspects of, but more so with how they may use the instrumentation architecture to gather information that may then serve as the basis for management and control strategies. The use of the architecture is

demonstrated through two Graphical User Interface (GUIs), which were developed to both demonstrate the architecture and assess its effectiveness. The GUIs were developed using Java's AWT and Swing APIs and allow the creation of instrumentation services to demand and the recording of distributed application parameters. Note that the GUIs are not part of the instrumentation architecture, they serve only as demonstration aids.

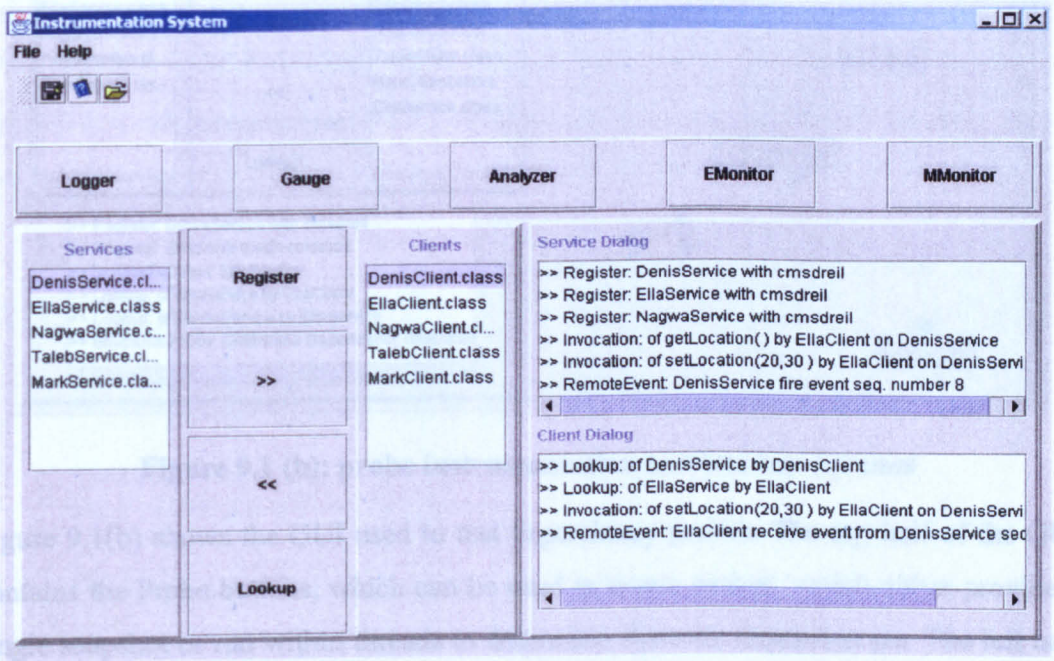


Figure 9.1 (a): basic instrumentation services test harness

Figure 9.1 (a) shows the main GUI used to test the logger, gauge, analyzer and monitor instruments. The top half of the GUI contains buttons, which can be used to create instrumentation services as required. The left half of the GUI contains groups of clients and application services. The Register button is used to register any application service with a lookup service and the Lookup button is used to allow any client to find a registered service. The ">>" and "<<" buttons may be used to add new roles to services or clients, by transferring either from one list to the other. So for example, if `DenisService` is currently selected and the ">>" button is clicked then `DenisService` may also play the role of a client. The right half of the GUI provides a record of: the operations performed on clients and servers, interactions between clients and services and instrumentation applied to clients and/or services.

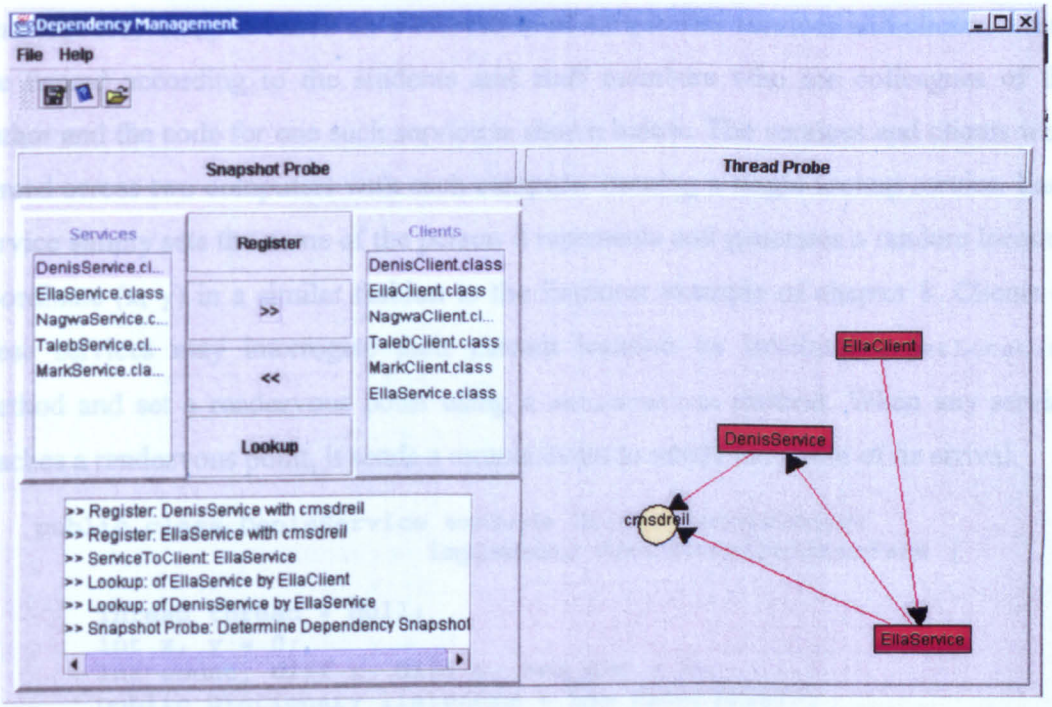


Figure 9.1 (b): probe instrumentation services test harness

Figure 9.1(b) shows the GUI used to test dependency probes. The top half of the GUI contains the Probe buttons, which can be used to create probes, which either provide a single snapshot or run within threads to determine dynamic dependencies. The left half of the GUI is similar to that of Figure 9.1 (a), except that there is a single dialog box to record client/service operations and interactions. The right half of the GUI contains a 2D graphics pane, which is used for drawing the dependency digraph.

9.2 *Instrumentation Case Studies*

Four instrumentation case-studies were conducted to demonstrate and assess different functional aspects of the instrumentation architecture. The first two case-studies were conducted on simple distributed applications, which serve no useful purpose other than to generate information, which can be monitored. The third case study is more realistic in that it uses a simple distributed mobile phone billing application to assess client-server access patterns. The fourth case-study considers a simple distributed application with respect to how instrumentation may be used for the realistic appraisal of implemnation alternatives.

9.2.1 Simple Logging and Monitoring

The first case-study is based on a federation of simple Jini services and clients, which are named according to the students and staff members who are colleagues of the author and the code for one such service is shown below. The services and clients were spread across two computers with each computer running a single lookup service. Each service simply sets the name of the person it represents and generates a random location coordinate (x, y) in a similar fashion to the Explorer example of chapter 8. Clients of these services may interrogate their current location by invoking a `getLocation` method and set a rendezvous point using a `setLocation` method. When any service reaches a rendezvous point, it sends a remote event to notify the client of its arrival.

```
public class DenisService extends UnicastRemoteObject
                          implements DenisServiceInterface {

    Thread thread = null;
    int x, y = 0;
    int count, diff_x, diff_y, seq_num = 0;
    public Dictionary listeners = new Hashtable();

    class Location {
        int x, y;
    }

    Location location, rendezvous = null;

    public DenisService() {
        register();
        init();
    }

    void register() {
        // register with a lookup service
        ....
        ....
    }

    public void init() {
        Runnable task = new Runnable() {
            public void run() {
                roam();
            }
        };

        thread = new Thread(task, "DenisService");
        thread.start();

    }

    void roam() {
```

```
        location = new Location();
        while (rendezvous == null) {
            location.x = (int)(Math.random() * 50);
            location.y = (int)(Math.random() * 50);
            System.out.println("Location " + location.x  + " "
                               + location.y);
            if (count > 90) {
                setLocation(20, 30);
                count = 0;
            }
            count++;
        }
        diff_x = rendezvous.x - location.x;
        diff_y = rendezvous.y - location.y;
        while (Math.abs(diff_x) > 0 && Math.abs(diff_y) > 0) {
            if (diff_x > 0) {
                location.x++;
                diff_x--;
            }
            else {
                location.x--;
                diff_x++;
            }
            if (diff_y > 0) {
                location.y++;
                diff_y--;
            }
            else {
                location.y--;
                diff_y++;
            }
            System.out.println("Location " + location.x  + " " +
                                        location.y);
        }
        System.out.println("Redezvous at " + location.x  + " " +
                                        location.y);
        fireRemoteEvent(0);
        rendezvous = null;
    }

    public Location getLocation() {
        return location;
    }

    public void setLocation(int x, int y) {
        rendezvous = new Location();
        rendezvous.x = x;
        rendezvous.y = y;
    }

    void fireRemoteEvent(long id) {
       Enumeration enum = listeners.keys();

       while (enum.hasMoreElements()) {
           RemoteEventListener listener =
                       (RemoteEventListener)enum.nextElement();
           RemoteEvent event = new RemoteEvent(this,id,seq_num,
```

```
                    (MarshalledObject)listeners.get(listener));
            try {
                listener.notify(event);
            } catch(UnknownEventException e) {
                e.printStackTrace();
            } catch(RemoteException e1) {
                e1.printStackTrace();
            }
        }
        seq_num++;
    }

    public static void main(String[] args) {
        DenisService ds = new DenisService();
    }

}
```

The instrumentation case study uses logger instrumentation services to track the coordinates of each application service. Event and method invocation monitor services were also attached to each application service to acknowledge events generated by, and method invocations made on the application services respectively. The overall case study was conducted through two separate studies: first the instrumentation services were attached during startup (i.e. when each application service was registered) and second, the instrumentation services were attached/detached at the discretion of the controller using the Logging and Monitoring GUI (i.e. the author). The test conditions were as follows:

- Each application service was registered with one of two lookup services running on two separate computers (`cmsdreil` and `cmsegris`).
- Both computers were of the same specification, namely a Pentium III computer with 256 MB RAM Running Windows 2000.
- Each application service was run in its own JVM (i.e. each application service contained a `main` method).
- Each instrumentation service was run on the same JVM as the application service to which it was attached.
- Virtual memory gauges were used to measure the virtual memory available in each JVM in a single computer and these values were averaged over the

JVMs to give a single measure of virtual memory for each computer in each of the studies.

- No other applications were run on the computers and several non-essential Windows services were halted.

The transcript below shows the output for a typical study, which involves a lookup service running on the computer `cmsdreil`.

```
DenisService: register with cmsdreil
Logger:  register with cmsdreil
Logger: attach to DenisService
Logger: start logging
Logger: coordinates 28, 57
Logger: coordinates 11, 32
Logger: coordinates 28, 30
EMonitor: register with cmsdreil
MMonitor: register with cmsdreil
EMonitor: attach to DenisService
MMonitor: attach to DenisService
EllaClient: found DenisService
MMonitor: getLocation invoked on DenisService - returns 12, 23
MMonitor: setLocation invoked on DenisService - params 20, 30
Logger: coordinates 13, 24
Logger: coordinates 14, 25
Logger: coordinates 15, 26
Logger: coordinates 16, 27
Logger: coordinates 17, 28
Logger: coordinates 18, 29
Logger: coordinates 19, 30
Logger: coordinates 20, 30
EMonitor: DenisService fire event - seq. number 8
```

The transcript shows how logger and event and method invocation monitors are attached to log data and monitor the behaviour of `DenisService`. The invocation monitor acknowledges the method invocations made by `EllaClient` and the event monitor acknowledges the event fired by `DenisService` when it reaches the rendezvous destination.

The virtual memory gauge simply checks the initial total amount of memory available in each JVM. The gauge then starts `a java.swing.Timer(int delay, ActionListener listener)` with `delay` set to 1ms. Each time that the delay expires, the listener callback calls the `java.lang.Runtime.freeMemory` method to check the amount of free memory available within the applications JVM.

Figures 9.2, represent the amount of memory used for the two logging and monitoring studies. Figure 9.2 (a) shows the memory usage when no instrumentation services are used (the *placebo*). Figure 9.2 (b) shows the memory usage when instrumentation services are attached at startup. Figure 9.2 (c) shows the memory usage when instrumentation services were attached/detached at the discretion of the controller (the author in this case).

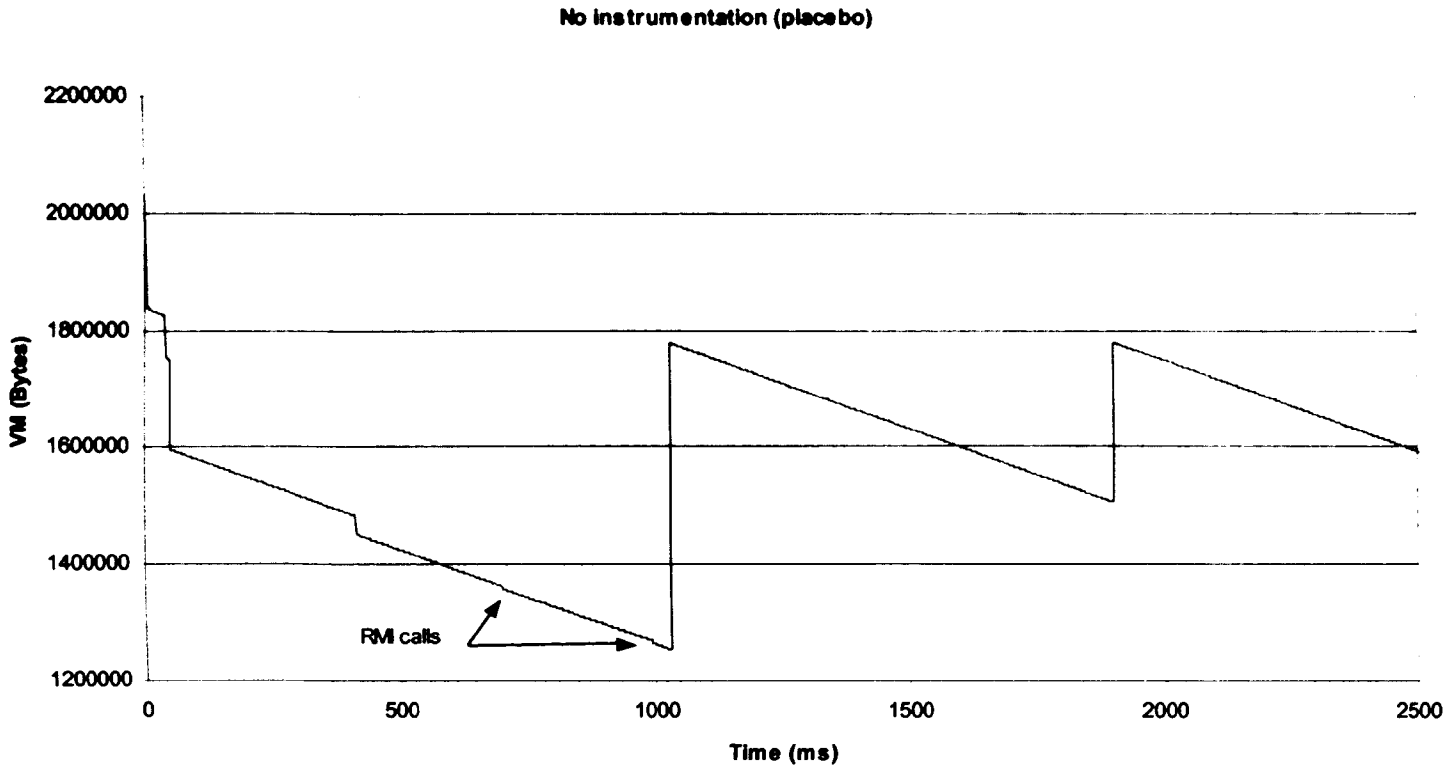


Figure 9.2 (a): VM usage for no instrumentation (placebo)

The vertical scale of each graph represents the amount of virtual memory available in the JVM and the horizontal scale represents time in milliseconds. Each graph has a saw-tooth profile, which represents how the virtual memory falls as the thread runs and rises when the garbage collector is invoked.

If we examine Figure 9.2 (a), we see two noticeable falls in VM, occurring at 40 and 415 milliseconds. The first fall occurs when `DenisService` registers with lookup service `cmsdreil`. The second fall occurs when a client uses Jini's lookup protocol to locate and download a copy of the `DenisService` proxy. There are also two further very small falls, which are barely noticeable for the placebo case. These very small falls, at 717, 1000 and milliseconds, occur when a client invokes the `getLocation` and `setLocation` methods of `DenisService` respectively.

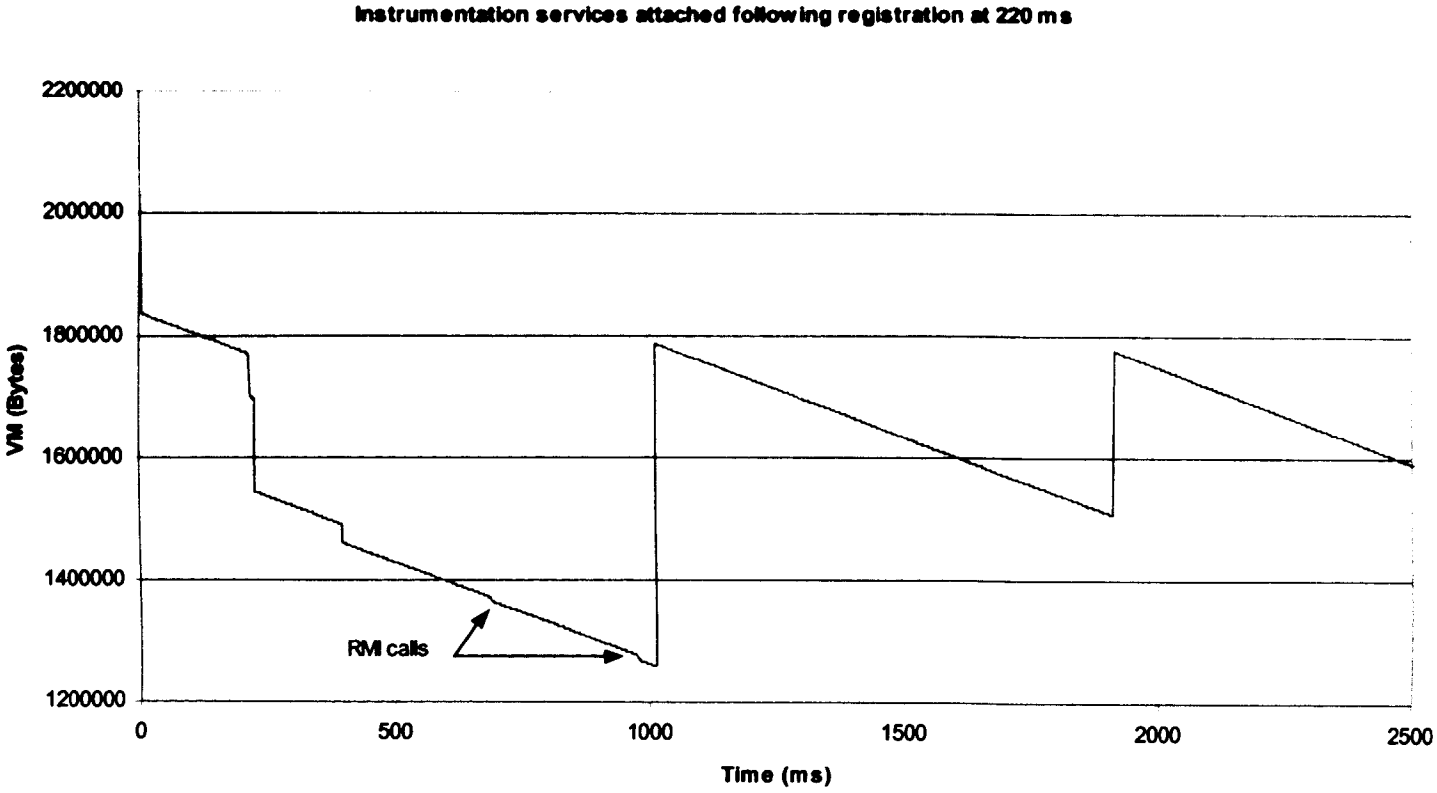


Figure 9.2 (b): VM usage for instrumentation service with simultaneous registration/attachment

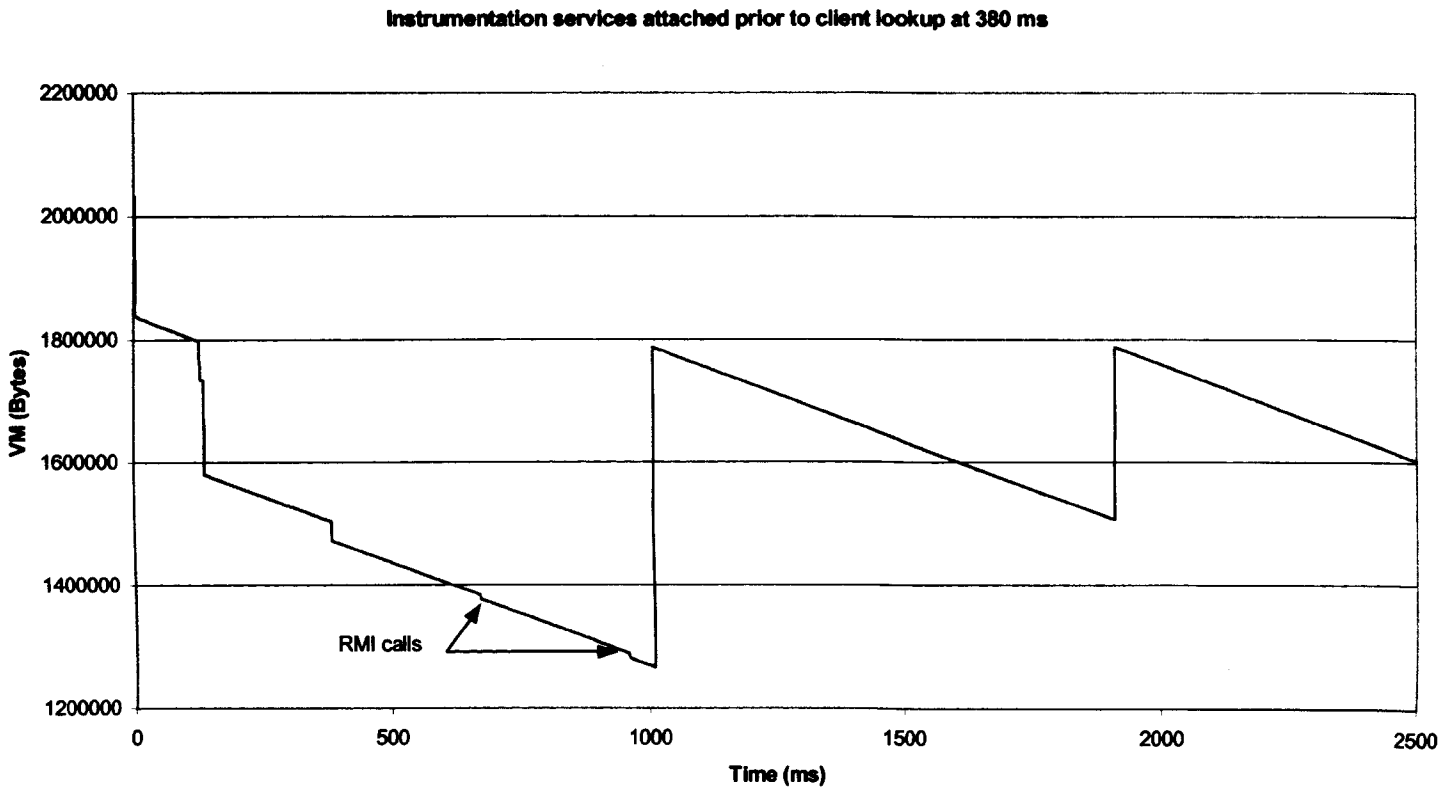


Figure 9.2 (c): VM usage for instrumentation service with delayed attachment

If we examine Figure 9.2 (b), we see the same noticeable VM falls as for Figure 9.2 (a). However, the first fall occurs later at 216 milliseconds and is now slightly larger. The increase in this fall and its delay (216 as opposed to 40 milliseconds) are a consequence of the registration and attachment of the monitor services. The invocation falls occurring at 717 and 1000 milliseconds are also slightly larger to those of the placebo case. This is so due to the extra overhead incurred when the monitor service intervene the `getLocation` and `setLocation` methods invocations made on `DenisService`.

If we examine Figure 9.2 (c) we see that the first fall is not as large of that of Figure 9.2 (b) and the delay somewhat reduced – 128 as opposed to 216 milliseconds. This is so because the monitor services are only registered and not attached when the application starts. The monitor service is actually attached just before the client lookup, occurring at 415 milliseconds, and this results in only a marginal increase in the fall at 415 milliseconds in comparison to that of Figure 9.2 (b). Figure 9.2 (c) also contains the same two small falls, which represent the monitor service's intervention on method invocations.

Overall, there is no great difference between the VM falls of the two instrumentation studies and the uninstrumented placebo case. So, we may conclude that the VM overhead of instrumentation service registration/attachment is relatively small. However, although the VM overhead is small, the register and attach (and join) operations can introduce significant time delays. These time delays and a more thorough assessment of instrumentation service performance are considered in section 9.3.4. The time delay is even more significant when instrumentation services register with lookup services running on different computers to themselves. The time delay is also significant when instrumentation services lookup other instrumentation services, on remote computers, with whom they wish to join.

9.2.2 Determining Dynamic Dependencies

The second case-study considers the determination of the dynamic dependencies amongst a small federation of application services and their clients running on a single computer. The services serve no useful purpose other than to print messages to

acknowledge the invocation of their methods. The case-study makes use of the Dependencies GUI considered previously. The federation consists of five services (`AService`, `BService`, `CService`, `DService`, `EService`) and a single lookup service, `cmsdreil`. The case-study also used five clients (`AClient`, `BClient`, `CClient`, `DClient`, `EClient`), which may each use any of the five services. Each service was coded in accordance with the compromise of chapter 8 in that each service and client implement the Jini `Administrable` interface and contains a `ServiceAdmin` object.

Again, the overall case study was conducted through two separate studies: first the instantaneous (snapshot) dependencies were determined for a group of application services and second, the initial dependency snapshot was altered by changing the group's configuration using the Dependencies GUI. For both studies, the target application client component (for which dependencies were determined) was `BClient` and a single probe service was attached to this client. For the first study, the probe did not register to receive event notifications of changes in bindings and was immediately detached after the dependencies were determined. In the second study the probe did register to receive such notifications from the lookup service `cmsdreil` and remained attached to `BClient` throughout the duration of the study. The test conditions were similar to those of Section 9.2.1, except that only a single computer was used.

Figures 9.3, represent the dependencies for each study. Figure 9.3 (a) the dependency snapshot when the service bindings have been established. Figure 9.3 (b) shows how the initial digraph is redrawn after the initial dependencies have altered.

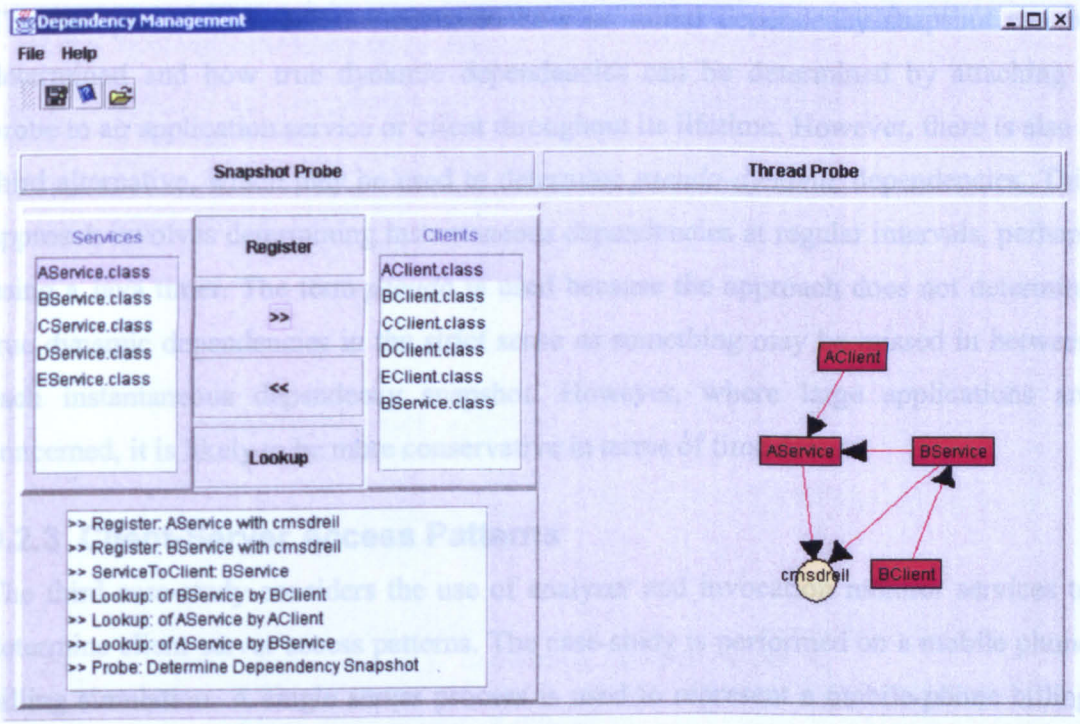


Figure 9.3 (a): dependency case study – initial dependencies

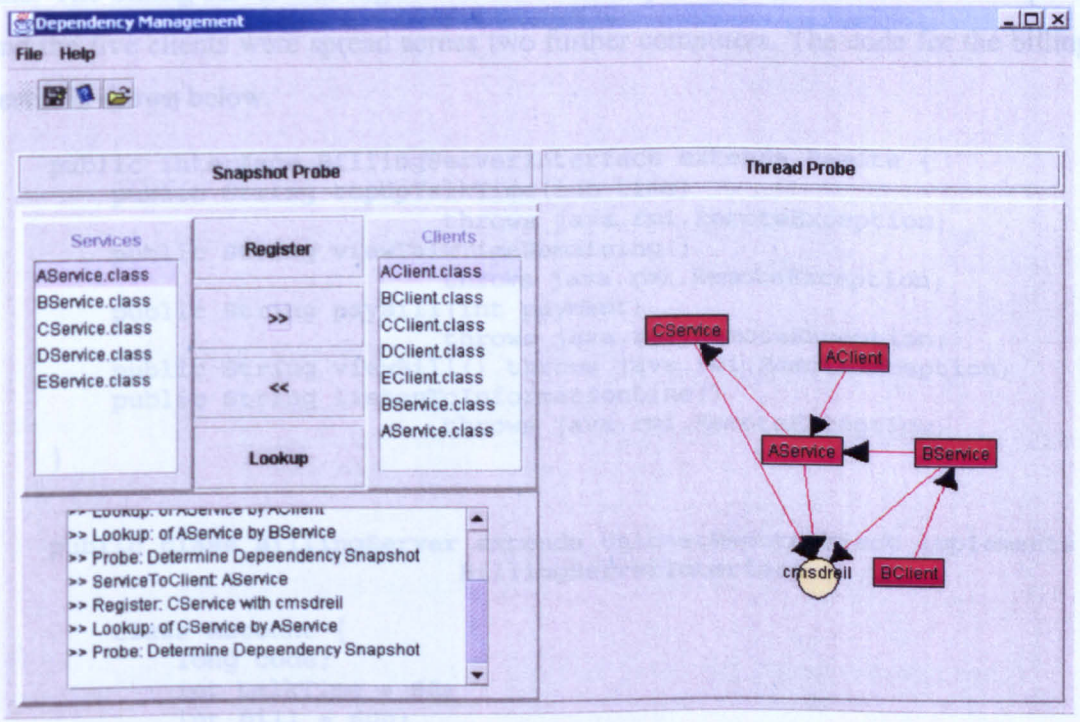


Figure 9.3 (b): dependency case study – final dependencies

The two previous studies demonstrated how an initial dependency snapshot can be determined and how true dynamic dependencies can be determined by attaching a probe to an application service or client throughout its lifetime. However, there is also a third alternative, which may be used to determine *pseudo-dynamic* dependencies. This approach involves determining instantaneous dependencies at regular intervals, perhaps using a Java timer. The term *pseudo* is used because the approach does not determine true dynamic dependencies in the strict sense as something may be missed in between each instantaneous dependency snapshot. However, where large applications are concerned, it is likely to be more conservative in terms of time delays.

9.2.3 Client-Server Access Patterns

The third case-study considers the use of analyzer and invocation monitor services to determine client-server access patterns. The case-study is performed on a mobile phone billing simulation. A single server process is used to represent a mobile-phone billing server and clients may access this server to enquire remaining talk-time available, top-up talk-time, enquire the outstanding bill, make payments or listen to an information line. The billing server was registered with a lookup service located on one computer and the five clients were spread across two further computers. The code for the billing server is shown below.

```
public interface BillingServerInterface extends Remote {
    public String topUpTalkTime(int time)
                        throws java.rmi.RemoteException;
    public String viewTalkTimeRemaining()
                        throws java.rmi.RemoteException;
    public String payBill(int payment)
                        throws java.rmi.RemoteException;
    public String viewBill() throws java.rmi.RemoteException;
    public String listenToInformationLine()
                        throws java.rmi.RemoteException;
}


public class BillingServer extends UnicastRemoteObject implements
                         BillingServerInterface {

    class Account {
        long code;
        int talkTime = 60;
        int bill = 600;
    }
```

```
Account[] accounts = new Account[5];
Account theAccount = null;

public BillingService() throws RemoteException {
    super ();
}

void register() {
    // register with a lookup service
    ....
    ....
}

void init() {
    accounts[0].code = 12345;
    accounts[1].code = 23451;
    accounts[2].code = 34521;
    accounts[3].code = 45321;
    accounts[4].code = 54321;
}

void validate(long code) throws java.rmi.RemoteException {
    for (int i=0; i < accounts.length; i++)
        if (accounts[i].code = code) theAccount = accounts[i];
        else theAccount = null;
}

public String topUpTalkTime(int time)
                        throws java.rmi.RemoteException {
    theAccount.talkTime += time;
    theAccount.bill += time*10;
    return "Talk time available " + theAccount.talkTime +
                                        " mins.";
}

public String viewTalkTimeRemaining()
                        throws java.rmi.RemoteException {
    return "Talk time available " + theAccount.talkTime +
                                        " mins.";
}

public String payBill(int payment)
                        throws java.rmi.RemoteException {
    if (payment > theAccount.bill) {
        theAccount.talkTime += (payment - theAccount.bill)/10;
        theAccount.bill = 0;
    }
    else
        theAccount.bill -= payment;
    return "Outstanding bill " + theAccount.bill;
}

public String viewBill() throws java.rmi.RemoteException {
    return "Outstanding bill " + theAccount.bill;
}

public String listenToInformationLine()
```

```
                    throws java.rmi.RemoteException {
        theAccount.bill += 50;
        return "Welcome to the WAP billing server....";
    }

    public static void main ( String args[] ) {
        BillingServer bs = new BillingServer();
        bs.init();
        bs.register();
    }
}
```

A method invocation monitor service was attached to the billing server and an indirect analyzer service was attached to the invocation monitor. The invocation monitor was responsible for acknowledging method invocations made on the billing server by clients and writing the resultant invocation objects to the indirect analyzer. The analyzer was used to determine the frequency of access of each of the billing server's methods, via its `compute` method (section 8.2.6).

The test conditions were similar to those of sections 9.2.1 and 9.2.2 and five clients were used to access five accounts on the billing server over a period of 2.8 hours to simulate a one-week period. A one-week period involves 7 days of 24 hours, so a 2.8 hour simulation means that 1 day is represented as 24 minutes. Each client used random number generators to randomly select a period of time to the next invocation and randomly select a method to invoke. The frequency of the random invocations was increased for the last 48 minutes to simulate increased weekend activity.

The analyzer was coded to produce intermediate access frequency reports every 24 minutes (i.e. every day) and an overall access frequency report at the end of the simulation. Excerpts of the final report for the five clients accessing their respective accounts is shown below.

```
00:04:23 - Client2 (23451): Talk time available 120 mins.
00:07:45 - Client1 (12345): Talk time available 180 mins.
00:09:06 - Client3 (34521): Talk time available 90 mins.
00:12:21 - Client2 (23451): Welcome to the WAP billing server....
00:15:03 - Client5 (54321): Talk time available 120 mins.
00:18:23 - Client1 (12345): Talk for 10 mins.
00:20:54 - Client4 (45321): Talk time available 90 mins.
00:23:41 - Client3 (34521): Outstanding bill 800.
00:27:23 - Client5 (54321): Talk for 10 mins.
00:29:53 - Client4 (45321): Talk for 20 mins.
```

```
....
....
....
02:43:52 - Client3 (34521): Talk for 15 mins.
02:44:49 - Client1 (12345): Welcome to the WAP billing server....
02:45:41 - Client4 (54321): Talk for 10 mins.
02:46:23 - Client5 (54321): Outstanding bill 300.
02:47:14 - Client1 (12345): Talk time available 30 mins.
02:48:00 - Client3 (34521): Talk time available 60 mins.
```

This case-study demonstrates how basic or primitive instrumentation services may be combined to perform more complex instrumentation tasks. The study also demonstrates how indirect instrumentation services may be used in conjunction with direct instrumentation services. As a general rule, indirect instruments are more conservative in terms of their use of resources since they do not use the more costly Java reflection to acquire information.

9.2.4 Use of Regular or Activatable Jini Services?

The final case-study uses the virtual memory gauges (considered previously) in conjunction with indirect loggers and method invocation monitors to demonstrate how the combination may be used to provide a qualitative assessment of the implementation of a distributed application. In the previous case-studies virtual memory gauges were used to print memory values to the screen (i.e. to Java's `System.out` stream). For this case-study, indirect loggers were used to read the memory values from the gauges and store them to a file. The gauges are used to gauge virtual memory usage for each individual JVM used within the application. The virtual memory readings together with the number of active JVMs are recorded by indirect loggers and stored in logfiles from which performance comparisons can be made.

The simple example application services in the previous case-studies used RMI proxies as the main mechanism for actually providing the service to a client. These services subclass `UnicastRemoteObject`, and live within a server whose principal task is to keep the service alive and registered with a lookup service. If the server fails to renew a lease then the lookup service will eventually discard the service and if the server fails to keep itself and its service alive then the service will not be available when a client wants to use it.

The `UnicastRemoteObject` approach results in a server and a service which most of the time will be idle, probably swapped out to disk but still using virtual memory. From JDK 1.2 and upwards, there is an extension to RMI called *activation* [105], which allows an idle object to sleep, and be recalled to life when needed. In this way, it does not occupy virtual memory while idle. Of course, a process needs to be alive to restore such objects, and RMI supplies a daemon `rmid` to manage this. In effect, `rmid` acts as a virtual memory manager as it stores information about dormant Java objects in its own files and restores them from there as needed.

In this case-study, we are concerned with assessing the performance of a simple application implemented using the two Jini service implementation alternatives. The alternatives are: `UnicastRemoteObject` services and activatable services, where `UnicastRemoteObject` services subclass RMI's `UnicasteRemoteObject` class and activatable services subclass RMI's `Activatable` class.

1. Active Objects and Activation

According to the Activation Tutorial in the Java RMI specification [105] "*an 'active' object is a remote object that is instantiated and exported in a JVM on some system.... A 'passive' object is one that is not yet instantiated (or exported) in a JVM, but which can be brought into an active state.*" The transformation of a passive object into an active object is a process referred to as *activation*.

`UnicastRemoteObject` services exist within a server, where they are kept alive, consuming memory, even when idle, until they are eventually discarded. A simple `UnicastRemoteObject` service may combine the server and the service within a single Java class file. Alternatively, more complex `UnicastRemoteObject` services may consist of a server within one Java class file and the service (or backend server) within a second Java class file. In the case of the latter, the server will call the service's constructor to create an instance of the service, which it wishes to register and maintain a remote reference to the remote object that implements the service. By using the `UnicastRemoteObject` approach, both the server and its associated service are hosted by the same JVM. The service is essentially kept alive within the JVM so that clients may invoke its methods as and when desired.

An *activatable* service, considered further in [98], is registered by a server, but then the server is allowed to "die" thereby freeing up its JVM. The activatable object, which implements the activatable service, may then sleep until it is accessed by a client when it is resurrected, within a separate JVM, by the *Activation System*. The Activation System is essentially Java's RMI daemon, `rmid`, which maintains references to the dormant service so that they can be resurrected when needed by clients. The Activation System essentially guarantees that RMI calls on this service will not fail, even when the service is sleeping. This approach allows resources (primarily virtual memory) to be conserved, but the price to pay is that a new JVM "may" need to be started whenever a client needs to access an activatable service.

However, the need to introduce a new JVM for each activatable service may be avoided by arranging a group of activatable service within an *Activation Group*, so that all the services within the *Activation Group* may share the same JVM whenever their methods are invoked by clients. On first impression, activatable services seem like an attractive option. However, their efficiency is a function of the number of JVMs that need to be started whenever an activatable service is accessed. If the number of JVMs is high, then activatable services may in fact introduce greater performance overheads above that of `UnicastRemoteObject` services. Conversely, if the number can be kept moderately low, then performance savings may be had from using activatable services.

2. The Case-Study

The case-study uses the same five simple services (`AService` to `EService`) and clients (`AClient` to `EClient`) considered in Section 9.2.2 to assess the performance of `UnicastRemoteObject` service against activatable service implementations. As for Section 9.2.2, the five services and clients were hosted on the same computer. Three separate studies were performed according to the following arrangements:

- The `UnicastRemoteObject` services were all run within the same JVM.
- The activatable services were split into two groups and each group was run in a separate JVM.
- The `UnicastRemoteObject` services were run within five separate JVMs

For each of the three studies a separate JVM was used to run the lookup service `cmsdreil`.

The first study measures the performance of a `UnicastRemoteObject` service implementation with the five services run in a single shared JVM. Virtual memory gauges were attached to each `UnicastRemoteObject` service and indirect loggers were used to record the gauge readings. Figure 9.4 shows a typical VM usage graph for `AService`. The noticeable features on this graph are the falls in virtual memory occurring at 402 milliseconds (initial lookup service discovery and registration) and 1242 and 2098 when a method of `AService` is invoked by a client. We can also see a slow gradual fall in virtual memory, which is incurred in keeping `AService` alive. For the first study nine method invocations were made and these invocations were spread across the five services (`AService` to `EService`).

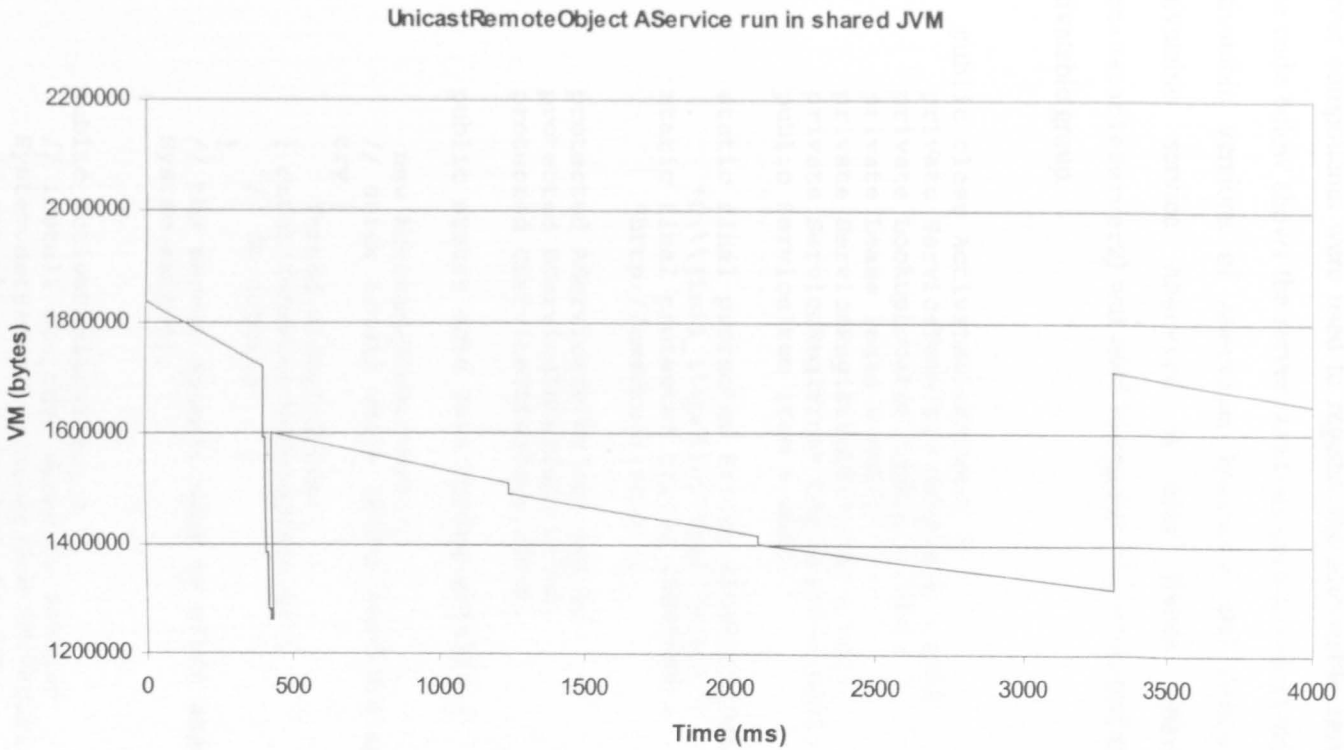


Figure 9.4: VM usage for UnicastRemoteObject service run in a shared JVM

The second study measures the performance of an activatable service implementation for which the activatable services were organized into two activation groups. Two server components were used to register the activatable services for the two groups. The code below shows the server (`ActivatableServer1`) that was used to register the activatable versions of `AService`, `BService` and `CService`. The code for the activatable service `AService` is also shown below. A similar server (`ActivatableServer2`) was used to register `DService` and `EService` within a second activatable group.

```
public class ActivatableServer1 {
    private ServiceTemplate template = null;
    private LookupLocator lookup = null;
    private Lease lease = null;
    private ServiceRegistration reg = null;
    private ServiceRegistrar registrar = null;
    public ServiceItem item = null;

    static final protected String SECURITY_POLICY_FILE =
          "C:\\jini1_1\\policy\\policy.all";
    static final protected String CODEBASE =
           "http://cmsdreil:8081/";

    protected AServiceInterface aStub;
    protected BServiceInterface bStub;
    protected CServiceInterface cStub;

    public static void main(String argv[]) {

        new ActivatableServer1();
       // stick around while lookup services are found
       try {
           Thread.sleep(10000L);
       } catch(InterruptedException e) {
          // do nothing
       }
       // the server doesn't need to exist anymore
       System.exit(0);
    }

    public ActivatableServer1() {
       // install suitable security manager
       System.setSecurityManager(new RMISecurityManager());
        // set the properties for the JVM
        System.setProperty("java.rmi.server.codebase",
                    CODEBASE);
        System.setProperty("java.security.policy",
                 SECURITY_POLICY_FILE);
```

```
// Install an activation group
Properties props = new Properties();
props.put("java.security.policy",
 SECURITY_POLICY_FILE);
ActivationGroupDesc.CommandEnvironment ace = null;
 ActivationGroupDesc group = new ActivationGroupDesc(props,
                                                     ace);
ActivationGroupID groupID = null;
try {
    groupID =
        ActivationGroup.getSystem().registerGroup(group);
} catch(RemoteException e) {
    e.printStackTrace();
    System.exit(1);
} catch(ActivationException e) {
    e.printStackTrace();
    System.exit(1);
}

try {
    ActivationGroup.createGroup(groupID, group, 0);
} catch(ActivationException e) {
    e.printStackTrace();
    System.exit(1);
}
MarshalledObject data = null;
ActivationDesc desc = null;
System.out.println("Group ID " +
     ActivationGroup.currentGroupID().toString());
try {
    aStub = (AServiceInterface) Activatable.register(desc);
     System.out.println("Activatable aStub " +
                    aStub.toString());
    bStub = (BServiceInterface) Activatable.register(desc);
     System.out.println("Activatable bStub " +
                    bStub.toString());
    cStub = (CServiceInterface) Activatable.register(desc);
     System.out.println("Activatable cStub " +
                    cStub.toString());
} catch(UnknownGroupException e) {
    e.printStackTrace();
    System.exit(1);
} catch(ActivationException e) {
    e.printStackTrace();
    System.exit(1);
} catch(RemoteException e) {
    e.printStackTrace();
    System.exit(1);
}

 try {
     lookup = new LookupLocator("jini://cmsdreil");
 } catch(java.net.MalformedURIException e) {
     System.err.println("Lookup failed: " + e.toString());
     System.exit(1);
 }
 try {
```

```
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " +
                               e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " +
                               e.toString());
            System.exit(1);
        }

        // register ourselves
        item = new ServiceItem(null, stub, null);
        try {
            reg = registrar.register(item, Lease.FOREVER);
            lease = reg.getLease();
            new LeaseRenewalManager(lease, Lease.FOREVER, null);
            System.out.println("ActivatableServer1 registered..."
                               + reg);
        } catch(java.rmi.RemoteException e) {
            System.err.println("Register exception: " +
                              e.toString());
        }
    }
} // ActivatableServer1

public class AService extends Activatable
                          implements AServiceInterface {

    public AService(ActivationID id, MarshalledObject data)
           throws java.rmi.RemoteException {
        super(id, 0);
        System.out.println("AService registration object " +
                                                    this);
    }

    public String sayHello() throws RemoteException {
        System.out.println("Hello from AService...");
        return "Hello from AService...";
    }
} // AService
```

In a similar fashion to the first case-study, virtual memory gauges and indirect loggers were used to record virtual memory usage for the servers. Figure 9.5 shows the VM usage graph for ActivatableServer1, for which we see a larger fall in virtual memory when the activation group is registered with the Activation System. The fall and also the time delay are larger than those incurred by the UnicastRemoteObject service implementation during the initial lookup service discovery and registration process (Figure 9.5). However, once the activation group has been registered, the server may then die, as it has done its job, thereby freeing up a JVM.

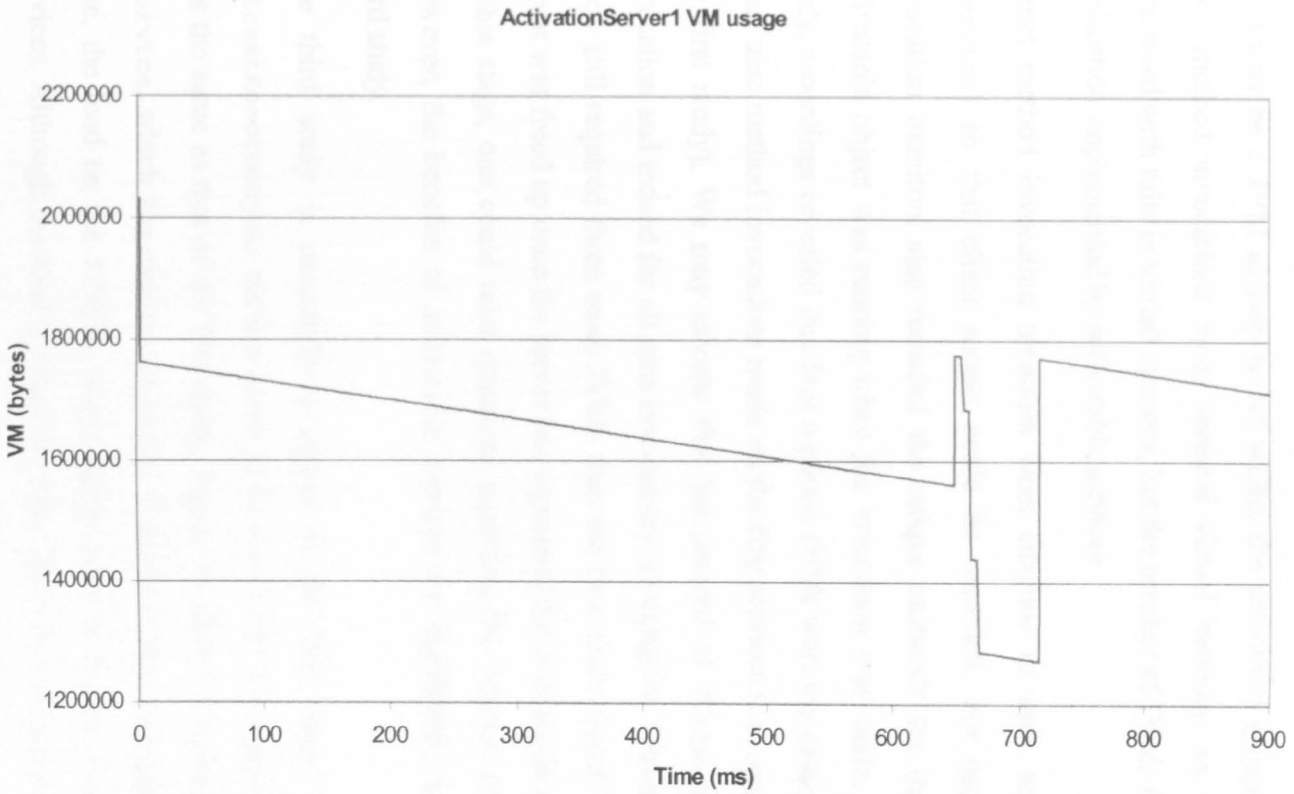


Figure 9.5: VM usage for ActivationsServer1 for activatable services AService, BService and CService

The Activation System allows activatable objects, such as `AService`, to begin execution on an as needed basis. When an activatable remote object is accessed (via method invocation), if that remote object is not currently executing, the Activation System initiates the object's execution inside an "appropriate" JVM. An appropriate JVM may be a JVM already active within the activation group, or else a new JVM. Such method invocations incur minimal virtual memory, so we are not so much concerned with falls in virtual memory, but the number of JVMs that are initiated for an application implemented by activatable services.

Direct method invocation monitors were attached to each activatable object (e.g. `AService`) so that client access could be recorded. For each client access, the invocation monitors also recorded the unique hashcode for the JVM in which the activatable object was running when the invocation was made. For the second case-study, recordings revealed that four separate JVMs were necessary to accommodate the same nine method invocations made on the five services (i.e. same nine invocations as the first study). We may assume that the amount of virtual memory used in each invocation and indeed for all nine invocations is negligible. However, this second case-study still required three more JVMs than the first study (recall that the JVM for the server was freed up once the server has registered the activatable services and died). So at this stage, one could raise questions regarding the benefits of activatable services. However, the benefits of activatable services are highlighted when we consider the third study.

The third study is essentially a repeat of the first study, but this time, each `UnicastRemoteObject` service is run in its own JVM. The approach to recording data was the same as that of the first study. Figure 9.6 shows a typical VM usage graph for `AService`, which has similar noticeable features to that of Figure 9.4. However, this time, the load on the JVM is much lighter since it is only running one of the five services. Although the load is relatively light, this and four other JVMs are completely tied up and cannot be used to run any other `UnicastRemoteObject` services. It is at this stage, on comparing studies two and three, that we see the advantage of using

activatable services. Essentially study two uses four JVMs with hardly any load, whereas study three uses five JVMs with relatively light loads.

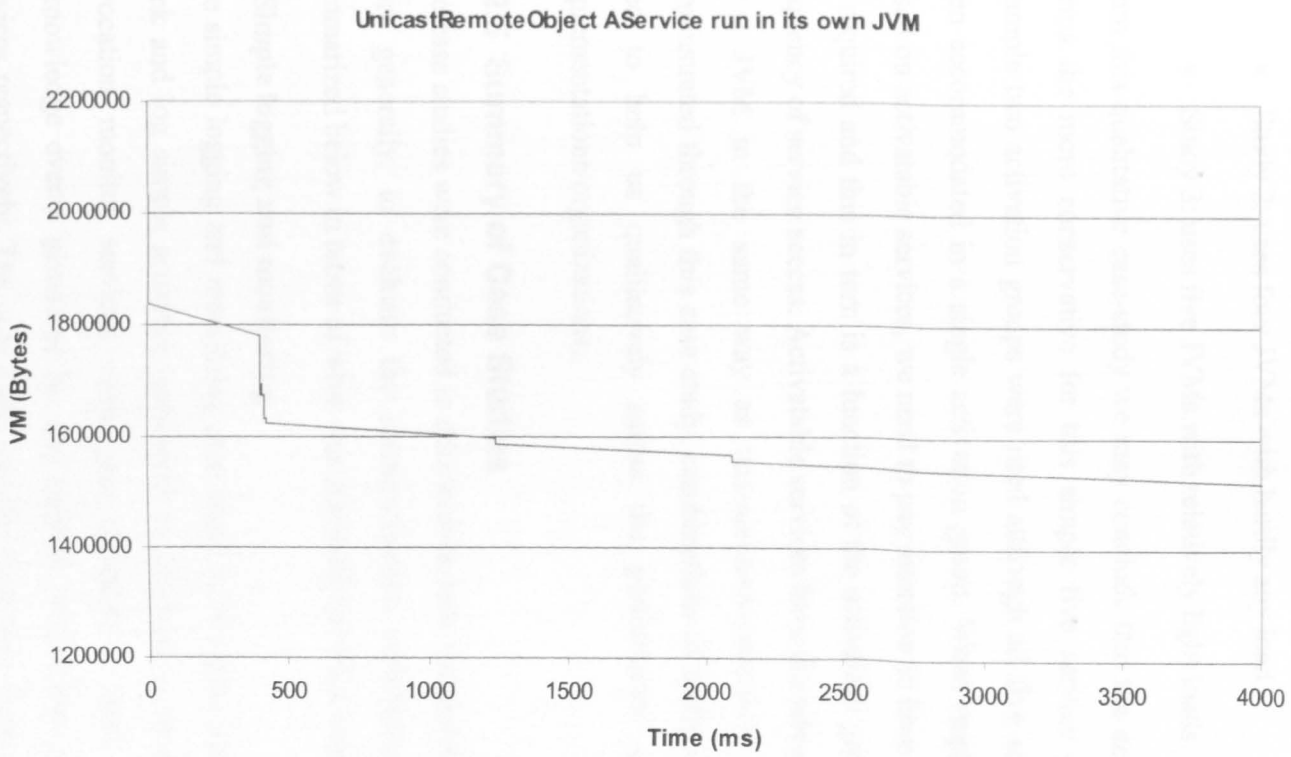


Figure 9.6: VM usage for UnicastRemoteObject service run in its own JVM

We may summarize the three studies as:

- Study 1: uses one heavily loaded JVM
- Study 2: uses four JVMs with hardly any load.
- Study 3: uses five JVMs with relatively light loads.

From this qualitative case-study we may conclude that the activatable implementation seems the more conservative for this simple five service example. In this simple example two activation groups were used although all five services could easily have been accommodated in a single activation group. When implementing an application based on activatable services, we need to pay attention to how many JVMs are likely to be required and this in turn is a function of the activation group organization and the frequency of service access. Activatable services have the advantage that they do not tie up a JVM in the same way as `UnicastRemoteObject` services. As we have demonstrated through this case study, combinations of instrumentation services can be used to help us qualitatively assess the performance of different application implementations/organizations.

9.2.5 Summary of Case Studies

The case studies were conducted to demonstrate how the architecture may be used and, more generally, to evaluate the instrumentation architecture. Each case study is summarized below in terms of what was achieved and what was learned.

1. Simple logging and monitoring

The simple logging and monitoring case study used logger instrumentation services to track and log simple activities performed by application services. Event and method invocation monitor services were also attached to each application service to acknowledge events generated by and method invocations made on the application services respectively. The case study demonstrated how these instrumentation services could be used without any additional application-level service code – i.e. *unobtrusively*.

The case study considered two different strategies for applying the instrumentation: attachment when each application service was registered, and delayed attachment at the

discretion of a controller (i.e. the author). It was concluded that overall the instrumentation had little affect on the amount of VM used by the application as a whole. However, the instrumentation registration and attachment operations can introduce significant time delays. Furthermore, the second application strategy may prove more favourable as it does not introduce the cumulative delay of simultaneous instrumentation registration/attachment.

2. Determining dynamic dependencies

The dynamic dependencies case-study considers the determination of the dynamic dependencies amongst a small federation of application services and their clients running on a single computer. The case-study first considered the instantaneous (snapshot) dependencies and then went on to demonstrate how dependencies may be re-determined after changes in the application's configuration. The case study considered how the probe needs some way of knowing when component bindings have altered. The approach that was demonstrated was to have the probe register to receive notifications from the lookup service in order to be aware of changes in bindings.

Unlike the other case studies, some additional code was needed in the application components in order to expose the bindings that the probe could then use. The attachment of a probe to an application service is required to determine the true dependencies emanating from a specific application component. Several such probes would be needed to build up a complete dependency snapshot. An alternative approach was mentioned, although not demonstrated, to determine *pseudo-dynamic* dependencies. This alternative approach involves determining instantaneous dependencies at regular intervals, perhaps using a Java timer. This approach alleviates the need for probes having to register to receive notification of changes in bindings although it does not provide a true dependency picture.

3. Client-Server access patterns

The client-server access patterns case-study made use of analyzer and invocation monitor services to determine client-server access patterns. The case-study was performed on a mobile phone billing simulation. The invocation monitors were used to acknowledge method invocations make on a billing server by clients. Results from

monitors were forwarded to an indirect analyzer, which was used to determine the frequency of access of each of the billing server's methods.

The general aim of this case-study was to use the instrumentation architecture for a realistic simulation run over a period of time with pseudo-random behaviour. More specifically, the case study aimed to show how basic or primitive instrumentation services may be combined to perform more complex instrumentation tasks. The study also demonstrated the use of indirect instrumentation services in conjunction with direct instrumentation services.

4. Regular and activatable Jini services

The final case-study demonstrated how instrumentation can be used to provide a qualitative assessment of the implementation alternatives for a distributed application. Again, a combination of virtual memory gauges, indirect loggers and method invocation monitors were used to demonstrate how instrumentation services may be combined to carry out complex measurement/monitoring tasks.

The study considered three separate realistic Jini service implementation arrangements:

- Several regular (`UnicastRemoteObject`) services run within the same JVM.
- Several activatable services split into two groups with each group run in a separate JVM.
- Several regular services were run within five separate JVMs.

From the instrumentation results it was possible to conclude that the activatable service implementation was more conservative for a study based on application five services. Furthermore, the load results from the instrumentation revealed that a single activation group would suffice.

9.3 Discussion and Qualitative Performance Assessment

This section discusses several novel approaches to the use of the instrumentation architecture. Guidelines are provided that may appeal to application developers or other researchers to assist the measurement, monitor and overall management of distributed applications. The section goes on to provide a qualitative assessment of the

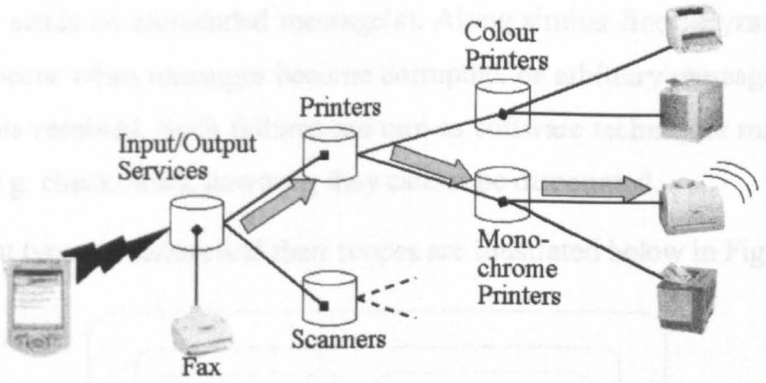
performance overhead incurred by the instrumentation architecture. The assessment highlights where performance may be affected and why and to what extent.

9.3.1 Centralized vs. Decentralized Instrumentation Control

The previous case studies considered the use of instrumentation services for the measurement and monitoring of several relatively simple distributed applications. For these case-studies the number of application services and instrumentation services were relatively small so it was possible to use a *centralized* regime of control. A centralized control regime means that all the instrumentation services were created, coordinated and controlled by a single program. Through such a form of control the instrumentation services may still be distributed over several computers, but they are controlled by a single program running on a single computer.

When distributed applications consist of a large number of application and instrumentation services, a *decentralized* regime of control may prove more favourable. With decentralized control the instrumentation services are created, coordinated and controlled by several programs. The control programs may communicate with each other, or communication may remain autonomous in which case communication is achieved via the instrumentation services.

The decentralized approach may organize instrumentation services according to *function*. For example, a distributed application may contain application services responsible for: backend database access, user interface presentation, resource sharing and access and control of hardware (printers, cameras etc.). Figure 9.7 shows a series of Input/Output devices that may be organized hierarchically using an arrangement of lookup services, which may be spread over several computers. Alternatively, instrumentation services may be organized according to *location*.



A hierarchical arrangement of Lookup services (running on separate computers) which provide access to a variety of Input/Output devices

Figure 9.7: lookup service chaining

There are no hard and fast rules governing the choice between centralized and decentralized approaches. However, as a general qualitative guideline, larger applications of fifty or more application services with five or more lookup services may benefit a decentralized instrumentation approach.

9.3.2 Using Instrumentation Services to Detect Failures

In distributed systems there are several types of failure, which may affect the application components or the communication channels through which they communicate. The main types of failure, as classified by [106], are: omission failures, timing failures and Byzantine (arbitrary) failures. Omission failures refer to cases when a component or communication channel fails to perform actions that it is supposed to do. Typically, a component omission failure occurs when a component crashes or fails to respond (fail-stop) and a channel omission failure may occur when a message is sent, but never received.

Timing failures may occur in synchronous communications when components fail to execute steps within time-limits or when messages arrive too early or too late due to problems in components or their communication channels. Byzantine failures represent the worst-case failure semantics in which any type of error may occur. Typically, a Byzantine component failure is one in which the component arbitrarily omits to send a

message(s) or sends an unintended message(s). Along similar lines, Byzantine channel failures may occur when messages become corrupted, or arbitrary messages are sent or arbitrary results received. Such failures are rare as software techniques may be used to detect them (e.g. checksums), however, they cannot be discounted.

These different types of failure and their scopes are illustrated below in Figure 9.8.

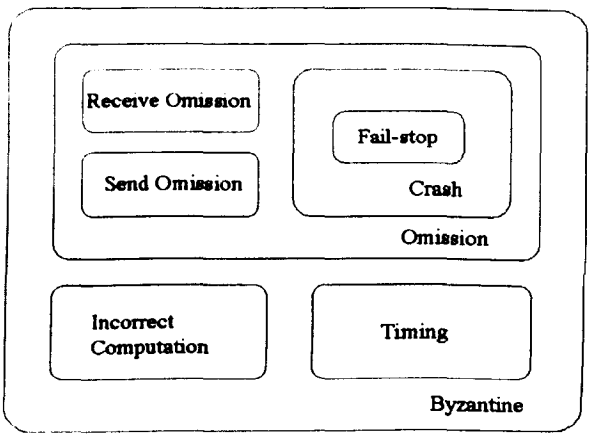


Figure 9.8: failure types and scopes

A large body of research has been conducted on understanding the problems facing the reliable detection of failure, typified by [107-110]. In this section we briefly describe how instrumentation monitor services may be used to detect the simplest of failures, namely component crash type failures.

Figure 9.9 illustrates an arrangement in which two instrumentation services have been located in the circuit that RMI calls traverse. One instrumentation service is located on the server's host and the other on the client's host. Through this arrangement we may detect component crash type failures by coordinating the client/service instrumentation services to exchange send/acknowledge events between each other. We cannot guarantee the detection of such failures, particularly when they are masked by other failures, such as send/receive omission type failures, which may occur when a communication link fails. If the communication link does fail, it is highly unlikely that the client/service instrumentation services will be able to communicate send/acknowledge events unless they follow a different path within the network. A

further shortcoming of this approach occurs when either host fails or experiences a problem , in which case either instrumentation service would also fail.

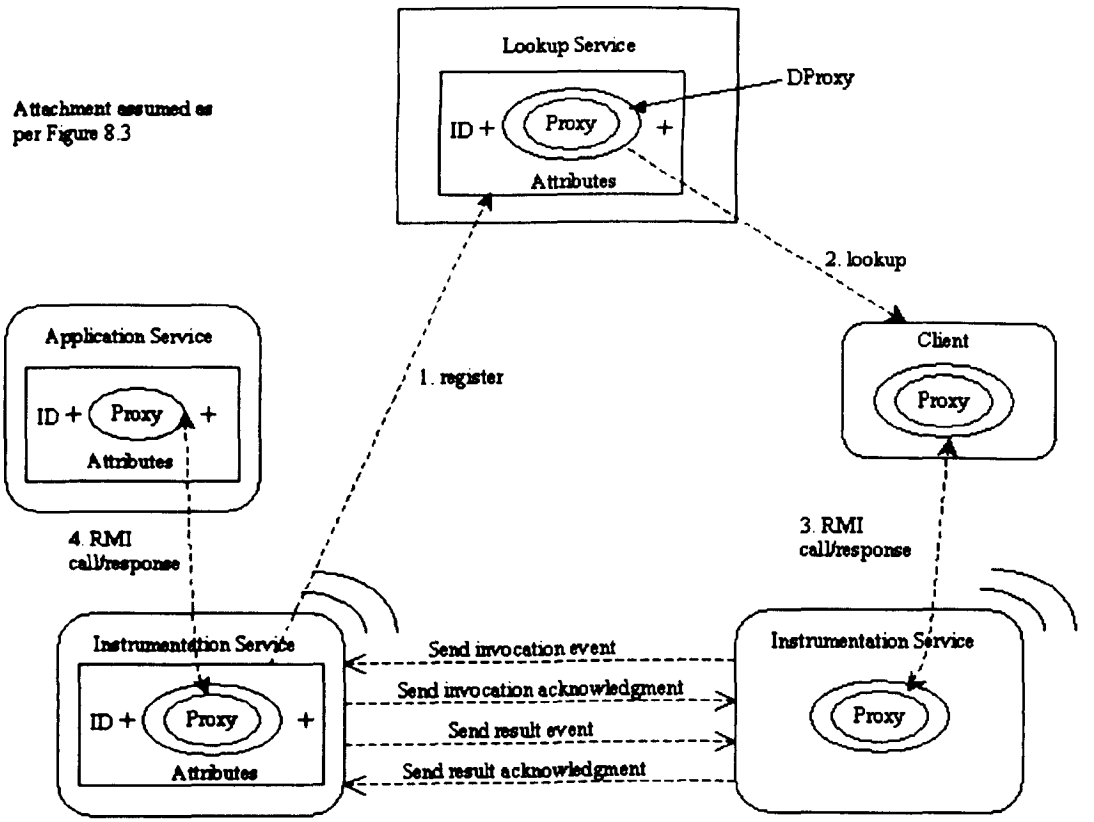


Figure 9.9: failure detection through instrumentation

Crash type failures are detected by checking the event sequences on both the client-side and the service-side, as shown in Figure 9.9. When a client invokes a method on its copy of the proxy, the client-side instrumentation service sends an event to the service-side instrumentation service. The service side instrumentation service then sends an acknowledgement event. At this stage, the service will execute the method and as was considered in chapter 8, the service-side instrumentation service intervenes on this invocation using the dynamic proxy construct.

If the method execution succeeds, the service-side instrumentation service sends a result event to the client-side instrumentation service. However, if the client-side instrumentation service does not receive such an event, it assumes that the service has suffered a fail-stop type failure. Assuming that the method was executed successfully,

the client-side instrumentation service concludes the dialogue by sending a result acknowledgement event. If the service-side instrumentation service does not receive this final event, it assumes that the client has suffered a fail-stop type failure.

The above approach describes a simple approach to using instrumentation services to detect the simplest of component failures. The approach is by no means fool-proof (crash failures may well be masked by other failures) and it has not been tested during the current research. However, the simple approach could be extended to develop more specialized failure detection instrumentation services. For example, such instrumentation services may maintain a heartbeat with their respective client and service components, or they may attempt several retries to check if a component really has crashed, or its is just temporarily busy or slow in responding. The development of such failure detection instrumentation services exceeds the scope of the current research and for this reason, it is mentioned in the "Future Work" section of chapter 10.

9.3.3 Extending the Architecture – Customized Instrumentation Services

The instrumentation architecture provides a set of instrumentation services, which developers may use to measure/monitor certain parameters of an application. The five main instrumentation services provided (logger, gauge, analyzer, probe and monitor) are basic or primitive services that may be used in the construction of more complex compound instrumentation services. The way in which the architecture has been designed and implemented was such to allow programmers to use its functionality in a way that meets their own specific needs. As was the case with the previous case-studies, specific instrumentation tasks may require a certain group of primitive instrumentation services. When this is so, programmers may use the API to define their own instrumentation classes, which combine several primitive instrumentation services.

For example, indirect loggers are often used in conjunction with other instrumentation services to log or record the information provided by the other instrumentation services. When programmers choose to design their own personal instrumentation services they do not have to use the Join operation. The Join operation is provided to allow instrumentation services to dynamically organize into groups at runtime (i.e. Join is a

dynamic operation). The Join operation also allows instrumentation services on different computers to group together. As considered further below, the Join operation does carry a performance overhead, which results in a time-delay, so it should only be used when it is absolutely necessary. If a programmer has identified a particular group of instrumentation services that serves some useful purpose and the group is to reside on the same single computer, then the API may be used to create a new instrumentation service class. This class may contain within it the necessary primitive instrumentation services that together perform the desired task, without having to use the Join operation (i.e. the group is statically joined).

Taking things a stage further, programmers may decide to develop their own instrumentation *factories*, which produce their own instrumentation service using software factories. A factory, in this context, is a piece of software that implements one of the factory design patterns considered in [53]. In general, a factory implementation is used when it is necessary to use one object to control the creation of and/or access to other objects. Software factories use the same principles as factory patterns that feature in everyday life.

In a similar fashion, an instrumentation factory may receive instructions from an application control program to create a particular instrumentation service. If this instrumentation service is a compound service, constructed of several primitive instrumentation services then the factory will create the necessary objects that make up the compound service and hand-back a reference to the control program through which it may access the compound service. The main advantage of using factories is that a consistent product is produced each time that the factory is called into use.

9.3.4 Instrumentation Performance Overhead

As far as possible, the approach has aimed to deliver instrumentation that is simple to use and is *transparent* from the point of view of the applications programmer. The approach has also aimed to provide instrumentation services, which are dynamic in the sense that they may be added and removed as required. To a large extent, these aims have been achieved, with the small exception regarding the compromise used to determine dynamic dependencies. However, the approach has given little consideration

to optimizing or tuning the performance of the instrumentation services in order to limit their overhead.

Essentially, the performance overhead of the instrumentation services is spread over three main areas:

1. The ten basic instrumentation service operations.
2. Jini's discovery and lookup protocols.
3. The use of Java refection.

The case studies (described previously in this chapter) revealed that the basic operations introduce minimal VM overhead, but several operations introduce significant timing delays. Jini's discovery and lookup protocols and RMI calls introduce timing delays due to the marshalling of remote objects over a network. Reflection introduces additional timing delays, which may affect the overall speed of a computation when used excessively. Each of these areas is assessed qualitatively below.

1. Performance Overhead of Instrumentation Services Operations

The most costly operations are Register and Invoke. The Register operation relies on Jini's discovery and lookup protocols and the Invoke operation relies on the RMI call mechanism and Java reflection. As a consequence, both the Register and Invoke operations rely heavily on objects being marshalled/unmarshalled across the network. For the Register operation objects are marsahlled/unmarshalled from a Jini lookup service to/from an instrumentation service. For the Invoke operation objects are marshalled between three entities: the application service, the instrumentation service and the client.

The marshalling of objects and parameters introduces significant time delay to their transportation across the network irrespective of middleware technology (Java RMI, CORBA, Web Services, or Jini). The speed at which objects may be marshalled depends on the response time of the sending-receiving object pair and the speed of the network link. For the instrumentation framework the marshalling/unmarshalling delay is incurred for the application services that make up the application itself and for the instrumentation services. Consequently the delay will be more significant as an

application scales and the instrumentation used to measure and monitor the application also scales.

From the author's own experience, discovery may take between 0.1 – 0.2 second and lookup between 0.2 - 0.3 seconds on a small office-based LAN. Jini's discovery and lookup protocols are dependent on the computers operating system and the networks DNS server and some limited benchmarks and analysis are available to assess lookup service performance and marshalling/unmarshalling delays [111, 112]. The discovery delay occurs typically for each instrumentation service, deployed to measure/monitor an application, although the delay is only incurred once, during the registration of the instrument. In addition to the delay caused by marshalling/unmarshalling the Invoke operation suffers an additional delay due to its use of Java's reflection API, which is considered further below.

2. Performance Overhead of Java Reflection

Reflection is used by certain instrumentation services to introspect classes to access application service parameters that are to be measured/monitored. Reflection is used extensively by the Invoke method for intervening on method invocations. To appreciate the extra time taken for reflective code to execute, we may consider the simple benchmarks provided in [113]. Table 9.1. The table compares the different method invocation strategies of: direct method invocation, invocation via an interface and invocation using the `invoke` method of Java's reflection API. The latter invocation mechanism is used in the Invoke instrumentation service operation for intervening application service method invocations.

JDK	Direct Test	Interface Test	Reflection Test
Sun 1.4	52 ms	54 ms	543 ms
Sun 1.4 -server	26 ms	56 ms	279 ms
Sun 1.3	124 ms	128 ms	2168 ms
Sun 1.3 -server	41 ms	58 ms	2012 ms
IBM 1.3	75 ms	78 ms	2134 ms

Table 9.1: reflection benchmark results

9.4 Chapter Summary

This chapter has described several relatively simple yet realistic case-studies, which demonstrate how the instrumentation architecture may be used for the measurement/monitoring of distributed applications. The chapter began with a description of the test-harness that was used for the studies. The execution of each case-study was then described and an assessment of the instrumentation overhead was also provided.

The chapter has also discussed some ideas regarding the different ways in which the architecture may be used and the ways in which the architecture may be extended to allow programmers to develop their own customized instrumentation services. It is important at this stage to point out two significant strengths of the architecture. The first is the ability to unobtrusively measure/monitor application components by way of dynamic attachment. The second is the ability to combine basic or primitive instrumentation services into *instrumentation units* that can carry out complex measurement/monitoring tasks.

The final part of the chapter provided a brief qualitative assessment of the performance overheads from using the instrumentation services in terms of time-delays and resource usage. Although no attempt was made to reduce these overheads, several guidelines were provided that may help to limit their effect. By reaching this stage, we have travelled quite some distance on our journey of dynamic instrumentation.

The thesis has described a long journey of which the milestones have been: requirements analysis, formal specification, functional modelling, architectural design, implementation and the use of dynamic instrumentation architecture. All that remains is to conclude the journey by appraising the unique contribution of the research and highlighting areas where future work could take place.

Chapter 10

Conclusions and Future Work

This thesis has described an instrumentation framework that can be used to assist the understanding of distributed applications and the framework has been applied to applications developed using Jini middleware technology. All that remains is to bring the thesis to a close by concluding the work done to date and the results achieved to date. This chapter does so by first summarizing the research and the main contributions of the thesis. The chapter then goes on to consider the novel contributions of the research. The chapter concludes by highlighting possible future research directions for follow up work to that described in the thesis.

10.1 Summary

The thesis has described a dynamic software instrumentation framework. The framework consisted of a series of related models and an architecture. The main framework models were: a requirements model, a classification model, formal and semi-formal analysis models and instrumentation programming and communication models. The requirements analysis established what instrumentation needs to measure/monitor and the classification model classified different categories of instrumentation. The formal model specified the basic operations of an abstract instrument. The semi-formal model used UML to describe the functional aspects relating to measurement and monitoring. The programming and communication models considered how instrumentation services interact with one another and the application components that they are required to measure/monitor.

The architecture consisted of the infrastructure components and a small number of instrumentation services that can be used to measure/monitor distributed application components. The architecture was implemented using a combination of the Java programming language (J2SE v1.4) and Jini middleware technology. Several instrumentation case-studies have been described, which demonstrate the use of the

architecture for the measuring and monitoring of distributed applications. A qualitative assessment is also presented to assess the performance overhead of the instrumentation.

As mentioned at the outset, the framework is applicable to the class of distributed systems developed using a distributed object-based middleware. Other classes of middleware do exist, namely Event-based middleware and Message-oriented middleware for which the framework is not directly applicable. These middlewares mainly use one-way communications rather than the request-reply communication found in object-based middleware. Event based middleware has potentially better scaling properties than object based middleware. Message-oriented middleware is favoured for applications in which messages need to be persistently stored and queued. Many of the ideas of the not directly applicable although some generic ideas such as dynamic attachment and message interception could be applied to assist the understanding of event-based or message-oriented middleware applications.

The framework may be used directly for distributed systems developed using Jini middleware and it has been demonstrated by way of several Jini applications. The overall approach may be used with other object-based middleware technologies such as Java RMI, CORBA although there are limitations. The approach may also be used with Web Services, which are not object-based, although again there are limitations to this applicability. For example the instrumentation Join operation relies upon features in Jini's own Join protocol. Alternative means would be required in order to implement the Join operation for Java RMI, CORBA and Web Services.

The architecture has only been demonstrated for LAN-based distributed systems involving a relatively modest number of application services. Issues of wide-area communication and scale have not been considered. Such issues are likely to prove significant for applications based on Web Services. An obvious issue relating to scale is that as an application scales it is likely that instrumentation will also scale and this in turn will impact on performance. To alleviate this it is important to provide facilities for instrumentation reuse and the instrumentation Join operation, although Jini specific, goes some way to providing reuse. The Join operation allows general purpose instruments to join other compound instrumentation units. The Unjoin operation frees

up the general purpose instruments so that they may be reused for other instrumentation tasks thereby removing the need to create additional general purpose instruments.

10.2 Research Contributions

This section concludes what has been achieved from the research and highlights the novel contributions that the thesis makes towards the field of distributed systems understanding and management. The section also compares the research against other recent related efforts studied in the literature.

10.2.1 Requirements Analysis

The consideration of requirements distinguished between *functional* and *operational* requirements. The functional requirements were concerned with what the instrumentation services must measure and monitor and the different types of instrumentation. The operational requirements were concerned with the incorporation of instrumentation within a distributed system and more specifically, their attachment.

This *separation of concerns* simplified the instrumentation architecture – by separating functional and operational requirements. The separation also reduced the *coupling* between what instrumentation should measure and monitor and what facilities are needed to allow this measurement and monitoring to take place. This in turn leads to *openness* in the sense that it would be possible to re-implement either the functional or operational aspects. The separation also led to different modelling approaches that were chosen to satisfy the specific characteristics of the functional and operational aspects in order to develop appropriate design models. A separation of concerns is also applied in [6] to decompose the monitoring system into separate tiers. However, the separation used in this thesis is applied at a much finer-grain, namely instrumentation services themselves.

The requirements analysis also provided a classification of instrumentation services according to the roles they play and the functionality they provide. In particular, the classification differentiated between:

- Direct vs. indirect instrumentation services.
- Static vs. dynamic instrumentation services.

- Synchronous vs. asynchronous instrumentation services.
- Event-handling vs. method-handling instrumentation services.

Undoubtedly this is not the only classification of instrumentation services, but it is the one proposed by the author, which served the rest of the thesis. The classification also provided an informal starting point for the instrumentation hierarchy underlying architecture.

The requirements analysis also attached names to the actual instruments used in the hierarchy. The instruments were named as: logger, analyzer, gauge, probe and monitor. The names were chosen to reflect similarities with instrumentation used in conventional engineering or the physical sciences. Overall, the requirements analysis provides a useful contribution by considering basic requirements of instrumentation from first principles. Such consideration is not so abundant in research related to distributed instrumentation systems.

10.2.2 Formal Modelling

There is little in the way of use of formal methods for specifying middleware systems and for considering interactions in distributed object based systems - [39, 114] are two of the few substantial formal specifications relating to middleware technology. It is hoped that the use of formal methods in this thesis may go some way towards promoting their use in the future. Typical applications may include protocol specification, storage schemes, specification of core services and concurrent object interactions.

MOTEL [16, 39] applies linear-time temporal logic to model an object-oriented distributed system. The emphasis of MOTEL is that of using formal models to assist the development of distributed systems. The contributions of MOTEL are a formal model and a property language. The efforts of MOTEL are to be applauded as they raise the awareness of the use of formal models in conjunction with distributed systems. However, the formal modelling used in this thesis differs in that it is concerned with an *abstract* representation of instrumentation.

The main aim of the formal modelling was the development of a series of state models that encapsulate the behaviour and interactions of instruments within the broader context of an application. It is the author's belief that the formal model allows instrumentation to be specified succinctly in this broader context. It is felt that a semi-formal approach would have led to a much larger unwieldy model in order to express the same concepts.

The formal model was developed using Object-Z, which is an extension to the Z formal modelling language to accommodate object-orientation. Object-Z was chosen because of its support for object-orientation and ability to write specification which contain precise state models, strong typing and precise axioms. In addition to Object-Z some Timed CSP (TCSP) was used to specify asynchronous events and concurrent behaviour.

Taken together Object-Z and TCSP constitute Timed Communicating Object-Z (TCOZ), which integrates the two separate modelling languages. TCSP primitive operations may be used in Object-Z classes to produce complete specifications. It is the author's belief that the combination could be used in the future to formally specify the structure and behaviour of middleware systems and their associated services.

Overall the formal modelling stage delivered a formal specification of an abstract instrument. This provided a useful research contribution and also strengthened the case for the use of formal methods for specifying interactions in distributed object based systems. The real strength stemmed from the fact that the formal model was actually used directly within the implementation to specify operational behaviour. All too often, formal models seem only to be used for expression and often do not directly feature within the actual target system's design/implementation.

10.2.3 Instrumentation Architecture

The architecture was based on the classification of instrumentation services that resulted from the requirements analysis. The architecture was developed to satisfy the functional requirements, namely: the measurement and monitoring functionality and incorporate the operational requirements. The activities of measuring and monitoring

were considered both for the computing platforms and the application components that execute on the platforms.

The architecture consisted of a series of classes organized to provide an infrastructure that sits between standard middleware services and application components. The focal point of the architecture was a small number of instrumentation services that can be instantiated to measure and monitor specific runtime parameters and behaviour information. These instrumentation services were chosen specifically to measure parameters of interest to the author, based on some fifteen years previous experience working with distributed systems. However, this is not to say that the instrumentation services constitute a definitive set that may be used with all applications.

It is anticipated that other researchers/developers may well have different views regarding appropriate instrumentation services. For example, no attention has been given to instrumentation services capable of measuring quality of service (QoS) in distributed multimedia applications. However, the architecture is, to a certain extent, extendable so that other instrumentation services may be incorporated. Furthermore, the small number of instrumentation services were intended to be general purpose and may be further specialized to suit specific needs.

As an example, an early version of the instrumentation architecture contained a `SynchronousTimedInstrument` class. This was originally intended to allow Timed Monitor instruments to be instantiated. However, this early version was evaluated and disregarded to avoid additional complexity and to limit the number of general purpose instrumentation services to a manageable size for the purpose of the research.

Overall the architecture provides a useful research contribution by proving the point that operational aspects such as instrumentation attachment and joining can be treated separately to functional aspects such as logging, gauging and probing. This also improves the architecture's openness by allowing either the operational or measurement functionality to be re-implemented.

10.2.4 Dependency Analysis

Dependencies have been mentioned on several occasions in the thesis as an important pre-requisite to furthering an understanding of a distributed system. Dependencies tell us how components rely on one another, or more particularly, how components depend on the services provided by other components.

The thesis has described how service dependencies may be derived from bindings between components. The bindings may then be used to build a graph of nodes and directed edges to reflect the dependencies at that particular instant. It is relatively straightforward to produce a static dependency snapshot, but extra effort is required to deduce dependencies dynamically as a consequence of changes in bindings.

The approach to determine dependencies relied upon an `Administrable` interface and an admin. object, which was used to represent the bindings for an application component. A visitor design pattern was also used so that a complete dependency picture could be built by iterating over the dependencies of successeive dependent components.

A Probe was chosen as the instrumentation service to determine dependencies. The analogy used was that of space probes, which are dispatched to gather information about a planet or deep space. Through this analogy the instrumentation probes may be dispatched to gather information about the services that a particular component depends on.

In order to deal with dynamic dependencies probes need to register to receive event notifications of changes in component bindings. The notifications from a lookup service are interpreted by a probe and the probe may then proceed to re-determine the application's dependencies. It was also mentioned how dependency analysis introduced a small compromise to instrumentation being unobtrusive. This compromise was the need for application programmers to implement the `Administrable` interface and return an admin. object in order to expose component bindings.

Overall, the dependency probes provide a useful research contribution by demonstrating how dynamic dependencies may be determined through a single instrumentation service. The approach to determine dependencies was based on the

work of [7]. However, the approach described in the thesis extends on this work by its ability to respond dynamically to changes in dependency, via notification of changes in bindings.

10.2.5 Comparison with Related Research

In order to "frame" the research contributions it is necessary to compare and contrast what has been achieved against other related research efforts. To this end, the research is appraised in relation to JMX, the DASADA projects and MODOCC.

1. JMX

The JMX specification [24] describes an architecture split into the three layers of: instrumentation, agents and distributed services. The specification or supporting literature provides little in the way of justification for this layering. The instrumentation layer is of greatest interest to this thesis and the main instrumentation component at this layer is the Managed Bean (MBean). JMX provides four types of MBean: Standard, Dynamic, Open and Model MBeans. The JMX specification described a notification model used in conjunction with MBeans. However, JMX provides little in the way of specific detail relating to what MBeans are intended to measure/monitor.

This thesis considered the basic requirements of instrumentation from first principles (operational and functional) and used these requirements as the basis for the eventual development of the architecture. One shortcoming of current instrumentation literature, such as JMX, (with the exception of MODOCC) is the lack of fundamental requirements issues relating to *what* is to be measured/monitored. This is an essential step before one proceeds to consider *how* measurement/monitoring is to be performed. It is felt that the thesis goes some way to address this shortcoming.

2. DASADA

DASADA is a group of projects concerned with the assembly and management of distributed component-based systems. Several DASADA projects have investigated, which use software gauges and probes to dynamically deduce component configurations. The DASADA projects reviewed in the thesis were: Software Surveyor, FIRM, En-Gauging and ABLE. These projects used gauges and probes in a variety of

different styles and forms to assist the understanding of component-based distributed systems.

A significant issue is the lack of clarity relating to what a constitutes a gauge and what constitutes a sensor or a probe. This thesis has made this distinction and assigned specific functionality to each of the different types of instrumentation considered (logger, gauge, analyzer, probe and monitor). The different types of instrumentation were based on similarities with instrumentation used in conventional engineering or the physical sciences.

The DASADA projects set out to determine an application's architecture and use this as the basis for subsequent adaptation. However, they do not consider the issue of dynamic dependencies, as considered in this thesis. The determination of such dependencies is crucial to understanding an application's structure. The thesis has considered the concept of dynamic dependencies and developed a specific instrumentation service (i.e. dependency probe) that can be used to determine and monitor dynamic dependencies.

3. MODOCC

Of all the literature reviewed the MODOCC system [6] bears the closest resemblance to the approach described in the thesis, although MODOCC follows a different approach to achieve a similar end. MODOCC describes a design approach for building monitoring systems. MODOCC starts out by considering the design of Generic Monitoring Systems (GMS) and considers fundamental questions relating to the design of a monitoring system and usage requirements.

The requirements are later refined to build the MODOCC system itself. In particular, MODOCC considers the design of sensors and the placement of sensors. MODOCC is to be applauded for the design approach for a Generic Monitoring System, and it does clarify exactly what the instrumentation is intended to monitor. However, it does not consider the operational aspects of instrumentation in quite the same light as the thesis. In particular, MODOCC does not consider issues such as the attachment of instrumentation to the components to be measured/monitored, as considerd in this thesis

10.3 Future Work

This section indicates several areas in which the work could be expanded. The areas sprang to mind as the research work progressed and also during discussions with the author's supervisor and researchers in the author's own academic school.

10.3.1 Security

Throughout the thesis the issue of security has been overlooked – simply because the extra effort to develop secure instrumentation would far exceed the scope of the thesis. However, secure instrumentation is a major issue particularly when instrumentation may be provided from a third-party. Many developers will be reluctant to adopt the use of instrumentation if it is not deemed secure.

The nature of instrumentation is to measure and monitor a distributed application and generally further an understanding of a distributed application. These tasks suggest that in the wrong hands instrumentation could be used maliciously. Furthermore, specific activities like intervening method invocations and wrapping application component proxies in instrumentation proxies would not be acceptable in secure environments without additional security measures.

Existing technologies do exist that could be applied directly to the instrumentation services described in this thesis. For example the use of a secure socket layer (SSL) and data encryption would go some ways to making the current instrumentation more secure. However, the author would advise a more thorough examination that dealt with issues such as *trust* and exactly what is meant by *trusted instrumentation*. Such work may even straddle the ideas of policy-based instrumentation described in the next section.

Once the issues of secure instrumentation have been dealt with then instrumentation itself may be applied to assist in security issues. For example instrumentation could be used to assist intrusion detection and assist in determining vulnerabilities that may arise out of interoperability. The latter issue is already under consideration at the author's own academic school. Work is currently ongoing that uses probe instrumentation as part of a prototype framework to improve security in interoperable environments [115].

10.3.2 Policy-based instrumentation

Policy-based management has received recent interest as a means to manage distributed systems [116-119]. Policies are rules governing the choices of behaviour of a system. Policy rules may be triggered dynamically to reserve resources, balance load or reconfigure the system in some way. Policies may be specified in a markup language and XML is a popular choice at present. Policies may be used in conjunction with *Role-Based Access Control (RBAC)* to control access to resources in a distributed system and provide a pragmatic security model.

It is the authors belief that policies could be used to better organize and deploy instrumentation services. In particular policy-based instrumentation is likely to be beneficial for large-scale distributed systems that may require significant instrumentation organized into several instrumentation *domains*. An instrumentation policy language could be developed to specify the rules governing the behaviour of the instrumentation.

Policies could be used to provide the necessary control functionality via triggers that call the instrumentation services' basic operations (register/unregister, attach/detach, join/unjoin, read/write, notify/invoke). Policies may be centralized or may themselves be distributed with responsibilities for managing specific instrumentation domains. The issue of *trust* was mentioned previously in section 10.2.1 and policies may also be used to provide a trust model to specify access rules for instrumentation services in relation to the distributed resources that they may/may not access.

10.3.3 Autonomic computing

Instrumentation is only intended to gather information and monitor the behaviour of a target application – it does not go so far as to providing any management or adaptation capabilities. However instrumentation can be combined with additional management or adaptation services to provide a management tier that is capable of reasoning about the behaviour of a target system and adapting the behaviour accordingly. Indeed this is the philosophy underlying reflective middleware which reflects on a system's behaviour and adapts this behaviour accordingly.

One approach to management/adaptation which is currently gaining momentum is that of *autonomic computing*. The term autonomic computing was "coined" by IBM to draw an analogy with the autonomic nervous system [120]. As pointed out in [120]: *"the autonomic nervous system frees our conscious brain from having to deal with vital, but lower-level functions"*. The concept of an autonomic computing system is one that "knows itself" to such an extent that it is capable of self-diagnosis and self-healing whenever internal problems and/or external disturbances are encountered.

A crucial aspect of a system that "knows itself" is the ability to gather information relating to its own execution environment. Indeed this is what the human autonomic nervous system through the bodies own sensors. Instrumentation may be used to gather information and present it to an autonomic manager in order to adapt/reconfigure a target system in the presence of disturbance/perturbation.

Projects are already underway in the author's own academic school to investigate the combination of autonomy and governance [121, 122]. The ability of self-adaptation and governance at runtime is attractive and would go some way towards managing today's large-scale complex system. It is the author's belief that one factor in their relative success will be the successful integration of instrumentation (although there are also many other factors). Autonomic, self-governing systems need unobtrusive, highly dynamic instrumentation capabilities in much the same light as those that nature provided for the human body!

References

[1] Kramer, J. and Magee, J.N., *Analysing Dynamic Change in Distributed Software Architectures.* IEEE Proceedings - Software, 1998. 145: p. 146-154.

[2] Kon, F., et al. *Dynamic Resource Management and Automatic Configuration of Distributed Computer Systems.* in *6th USENIX Conference on Object-Oriented Technologies and Systems 2001.* 2001.

[3] Dowling, J. and Cahill, V. *The K-component Architecture Meta-model for Self-adaptive Software.* in *Proceedings of the 3rd International Conference on meta-level Architectures and Separation of Cross-cutting Concerns (Reflection 2001) Lecture Notes in Computer Science Springer Verlag.* 2001.

[4] Sloman, M. *Management Issues for Distributed Systems.* in *IEEE 2nd International Workshop on Services in Distributed and Networked Environments (SDNE'95).* 1995. Whistler British Columbia Canada: IEEE Computer Society Press.

[5] Kramer, J. and Magee, J.N., *Dynamic Configuration for Distributed Systems.* IEEE Trans. on Software Engineering, 1985. SE-11: p. 424-436.

[6] Diakov, N.K., *MOnitoring Distributed Object and Component Communication,* in *Telematica Instituut.* 2004, Twente University: Enschede, The Netherlands.

[7] Hasselmeyer, P. *Managing Dynamic Service Dependencies.* in *12th International Workshop on Distributed Systems: Operations & Management (DSOM 2001).* 2001. Nancy, France.

[8] Diakov, N.K., et al. *Monitoring of Distributed Component Interactions.* in *Workshop on Reflective middleware RM200 in conjunction with IFIP/ACM International Conference on Distributed Systems Platforms and open Distributed Processing.* 2000. IBM Palisades Executive Conference Center New York USA.

[9] Hasselmeyer, P. and VoB, M. *Monitoring Component Interactions in Jini Federations.* in *The Convergence of Information Technologies and Communications (ITCom 2001).* 2001. Denver, CO: SPIE Proceedings.

[10] Wolf, A.L. and Kean, E., *FIRM - Framework for Interoperable Reconfiguration Measures (Definition, Deployment, and Use of Gauges to Manage Reconfigurable Component-Based Systems),* http://serl.cs.colorado.edu/~serl/dasada/, (Accessed: 25 Jabuary 2006).

[11] Garlan, D., Schmerl, B., and Chang, J. *Using Gauges for Architecture-based Monitoring and Adaptation*. in *Working Conference on Dynamic Systems Architecture*. 2001. Brisbane Australia.

[12] Wells, D. and Nagy, J., *Software Surveyor - Dynamically Deducing Componentware Configurations*, http://www.objs.com/DASADA/, (Accessed: 25 January 2006).

[13] Balzer, R. and Liuzzi, R., *En-gauging Architectures*, http://mr.teknowledge.com/DASADA.htm, (Accessed: 03 March 2006).

[14] Pazandak, P. and Wells, D. *ProbeMeister - Distributed Runtime Software Instrumentation*. in *First International Workshop on Unanticipated Software Evolution (USE2002) held in conjunction with ECOOP2002*. 2002. Malaga Spain: Springer Verlag LNCS 2548.

[15] Reilly, D. and Taleb-Bendiab, A. *Dynamic Instrumentation for Jini Applications*. in *3rd International Workshop on Software Engineering Middleware SEM002 (ICSE2002)*. 2002. Florida USA: Springer Verlag.

[16] Logean, X., *Run-time Monitoring and On-line Testing of Middleware Based Communication Services*. 2000, Swiss Federal Institute of Technology: Lausanne.

[17] Reilly, D., Taleb-Bendiab, A., and Laws, A. *An Instrumentation and Control-based Approach for Distributed Application Management and Adaptation*. in *ACM SIGSOFT Workshop on Self-healing Systems (WOSS'02)*. 2002. Charleston USA.

[18] Reilly, D. and Taleb-Bendiab, A. *A Service-Based Architecture for In-Vehicle Telematics Systems*. in *IEEE 22nd International Conference on Distributed Computing Systems (ICDCS 2002) WORKSHOPS - International Workshop of Smart Appliance and Wearable Computing (IWSAWC 2002)*. 2002. Vienna Austria.

[19] Coulouris, G., Dollimore, J., and Kindberg, T., *Distributed Systems - Concepts and Design*. 3rd ed. 2001: Addison Wesley, 0201619180.

[20] Emmerich, W., *Engineering Distributed Objects*. 2000: John Wiley & Sons Ltd., 0-471-98657-7.

[21] ANSA, *ANSA Reference Manual*. 1989, Architecture Projects Management Ltd.: Cambridge, UK.

[22] Liu, M.L., *Distributed Computing - Principles and Applications*. 2004: Pearson Addison-Wesley, 0321218175.

[23] Sun-Microsystems, *Dynamic code downloading using RMI (using the java.rmi.server.codebase property)*, http://java.sun.com/j2se/1.3/docs/guide/rmi/codebase.html, (Accessed: 12 March 2005).

[24] Sun-Microsystems, *Java Management Extensions (JMX) - Documentation*, http://java.sun.com/products/JavaManagement/reference/docs/index.html, (Accessed: 02 February 2006).

[25] Li, S., *Professional Jini*. 2000: Wrox Press Ltd., 1861003552.

[26] Openwings, *Openwings v1.1 Reference Implementation Specifications*, http://www.openwings.org/, (Accessed: 20 May 2005).

[27] Vogels, W., *Web Services Are Not Distributed Objects*. IEEE Internet Computing, 2003. 7(6): p. 59-66.

[28] Rover, D.T. *Performance Evaluation: Integrating techniques and Tools into Environments and Frameworks*. in *Roundtable, Supercomputing'94*. 1994. Washington DC, US.

[29] Simmons, M. and Koskela, R. *Performance Instrumentation and Visualization*. 1990: ACM and Addison Wesley.

[30] Waheed, A. and Rover, D.T. *A Structured Approach to Instrumentation System Development and Evaluation*. in *Proceedings of ACM/IEEE Supercomputing Conference SC'95*. 1995. San Diego, California, US.

[31] Heath, M.T. and Etheridge, J.A., *Visualizing the Performance of Parallel Programs*. IEE Software, 1991. 8(5): p. 29-39.

[32] Hao, M.C., et al. *VIZIR: An Integrated Environment for Distributed Program Visualization*. in *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS '95)*. 1995. Durham, North Carolina.

[33] Satterthwaite, E., *Debugging Tools for High Level Languages*. Software Practice and Experience, 1972. 2: p. 197-217.

[34] Sommerville, I., *Software Engineering*. 1990: Addison Wesley, 0201175681.

[35] Fenlason, J. and Stallman, R., *GNU gprof: the GNU Profiler*, http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html, (Accessed: 23 June 2004).

[36] IBM, *IBM Distributed Debugger and Object Level Trace version 9.1*, http://www-306.ibm.com/software/webservers/appserv/doc/v40/aes/infocenter/, (Accessed: 12 January 2006).

[37] Shaer, E.S.A., *A Hierarchical Filtering Based Monitoring Architecture for Large Scale Distributed Systems*. 1998, Old Dominion University: Norfolk.

[38] Logean, X., et al. *Runtime Monitoring of Distributed Applications*. in *Middleware'98 - IFIP International Conference on Distributed Systems Platforms and Open Distributed processing*. 1998. England.

[39] Dietrich, F., Logean, X., and Hubaux, J.-P., *Modeling and Testing Object-oriented Distributed Systems with Linear-time Temporal Logic*. Concurrency and Computation: Practice and Experience, 2001. 13(5): p. 385 - 420.

[40] Rackl, G., et al. *MIMO - An infrastructure for Monitoring and Managing Distributed Middleware Environments*. in *Middleware 2000 - IFIP/ACM International Conference on Distributed Systems Platforms*. 2000: Springer Verlag.

[41] Mahoney, B. and Dong, J.S. *Overview of the Semantics of TCOZ*. in *Integrated Formal Methods (IFM'99)*. 1999. York UK: Springer Verlag.

[42] Dietrich, F., *Modelling Object-oriented Communication Services with Temporal Logic*. 2000, Swiss Federal Institute of technology: Lausanne.

[43] Wedgam, M. and Halteren, A.v. *Experiences with CORBA interceptors*. in *Workshop on Reflective middleware RM200 in conjunction with IFIP/ACM International Conference on Distributed Systems Platforms and open Distributed Processing*. 2000. IBM Palisades Executive Conference Center New York USA.

[44] AdventNet, *XMOJO Project*, http://www.xmojo.org/products/xmojo/index.html, (Accessed: 02 February 2006).

[45] Garlan, D. and Stratton, R., *ABLE - Architecture based Languages and Environments*, http://www.cs.cmu.edu/~able/, (Accessed: 25 January 2006).

[46] Carzaniga, A., et al., *Siena - Scalable Internet Event Notification Architectures*, http://serl.cs.colorado.edu/~serl/siena/, (Accessed: 25 January 2006).

[47] Hall, R., Heimbigner, D., and Wolf, A.L., *Software Dock*, http://serl.cs.colorado.edu/serl/cm/dock.html, (Accessed: 26 January 2006).

[48] Hoek, A.v.d., Hiembigner, D., and Wolf, A.L., *Software Architecture, Configuration and Management and Configurable Distributed Systems: A Menage a Trois*. 1998, Department of Computer Science University of Colorado: Boulder US.

[49] Garlan, D., et al., *The Acme Architectural Description Language*, http://www.cs.cmu.edu/~acme/, (Accessed: 26 January 2006).

[50] Robinson, A. and Lounsbury, D. *Measuring and Managing End-to-end Quality of Service (QoS) Provided by Linked Chains of Application and Communications Services*. in *First Workshop on Evaluating and Architecting System dependabilitY (EASY)*. 2001. Göteborg Sweden.

[51] Garlan, D., et al., *ADLs and Related Languages*, http://www.cs.cmu.edu/~acme/adltk/adls.html, (Accessed: 12 December 2005).

[52] Jini.org, *Rio Project (version 3.0)*, http://rio.jini.org/, (Accessed: 22 January 2006).

[53] Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. 1995: Addison-Wesley, 0201633612.

[54] Java-Community-Process, *Java Logging API - JSR 47: Logging API Specification*, http://www.jcp.org/en/jsr/detail?id=047, (Accessed: 22 January 2006).

[55] Fahrimair, M., Salzmann, C., and Schoenmakers, M. *A Reflection Based Tool for Observing Jini Services*. in *Reflection and Software Engineering*. 2000: Springer Verlag.

[56] Keller, A. and Kar, G. *Dynamic Dependencies in Application Service Management*. in *2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*. 2000. Las Vegas NV US.

[57] Keller, A., Blumenthal, U., and Kar, G. *Classification and Computation of Dependencies for Distributed Management*. in *5th IEEE Symposium on Computers and Communication (ISCC 2000)*. 2000. Antibes Juan-les-Pins France.

[58] Coulson, G., *What is Reflective Middleware?*, http://computer.org/dsonline/middleware/RMarticle1.htm, (Accessed: 20 August 2005).

[59] Villagrá, V.A., et al. *An Approach to the Transparent Management Instrumentation of Distributed Applications*. in *8th IEEE/IFIP Network Operations and Management Symposium (NOMS'2002)*. 2002. Florence Italy.

[60] Asensio, J.I., et al. *Experiences with SNMP-based Integrated Management of a CORBA-based Electronic Commerce Application*. in *6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*. 1999. Boston Park Plaza Hotel Boston Massachusetts US.

[61] Blair, G., Coulson, G., and Grace, P. *Research Directions in Reflective Middleware: the Lancaster Experience*. in *3rd Workshop on Reflective and*

Adaptive Middleware (RM2004) co-located with Middleware 2004. 2004. Toronto Ontario Canada.

[62] Coulson, G., et al. *Towards a Component-based Middleware Framework for Configurable and Reconfigurable Grid Computing*. in *Workshop on Emerging Technologies for Next Generation Grid (ETNGRID-2004), associated with 13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004)*. 2004. Modena, Italy.

[63] Grace, P., Blair, G.S., and Samuel, S. *ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability*. in *International Symposium on Distributed Objects and Applications (DOA)*. 2003. Catania Sicily Italy: Springer Berlin / Heidelberg.

[64] Bencomo, N., et al. *Towards a Meta-Modelling Approach to Configurable Middleware*. in *2nd ECOOP'2005 Workshop on Reflection, AOP and Meta-Data for Software Evolution*. 2005. Glasgow Scotland.

[65] Wang, N., et al., *Towards an Adaptive and Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications*, http://www.computer.org/dsonline/middleware/RM.htm, (Accessed: 03 March 2006).

[66] Schmidt, D.C., Levine, D.L., and Mungee, S., *The Design and Performance of Real-time Object Request Brokers*. Computer Communications, 1998. 21: p. 294-324.

[67] Zinky, J.A., Bakken, D.E., and Schantz, R., *Architectural Support for Quality of Service for CORBA Objects*. Theory and Practice of Object Systems, 1997. 3(1).

[68] Kon, F. and Campbell, R.H. *Supporting Automatic Configuration of Component-Based Distributed Systems*. in *5th Conference on Object-Oriented Technologies and Systems*. 1999. San Diego CA USA: USENIX.

[69] Zimmermann, C., *Metalevels, MOPs and What the Fuzz is All About*. Advances in Object-Oriented Metalevel Architectures and Reflection (C. Zimmermann Ed.), ed. C. Zimmermann. 1996, 084932663X.

[70] O'Regan, G., *Introduction to Aspect-Oriented Programming*, http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html, (Accessed: 03 March 2006).

[71] Sun-Microsystems, *JVM Tool Interface (JVM TI)*, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html, (Accessed: 03 March 2006).

[72] JBoss.org, *JBoss Application Server*, http://labs.jboss.com/portal/jbossas, (Accessed: 03 March 2006).

[73] Zhang, C. and Jacobsen, H.A., *Aspectizing Middleware Platforms*. 2003, University of Tortonto, Computer Systems Research Group: Toronto.

[74] Eclipse.org, *AspectJ Project*, http://www.eclipse.org/aspectj/, (Accessed: 03 March 2006).

[75] Liu, R. and Coady, Y. *Modularization of Jini Services in Pervasive Systems: Conventional Bottle versus Contemporary Aspect*. in *Building Software for Pervasive Computing: OOPSLA '04*. 2004.

[76] Liu, C.R., Gibbs, C., and Coady, Y. *SONAR: System Optimization and Navigation with Aspects at Runtime*. in *AOSD'05 International Conference on Aspect-oriented Software Development Dynamic Aspects Workshop*. 2005. Chicago USA.

[77] Liu, C.R., Gibbs, C., and Coady, Y. *SONAR: Customizable, Lightweight Tool Support to Prevent Drowning in Diagnostics*. in *RAM-SE'05, 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*. 2005. Glasgow Scotland.

[78] Codehaus.org, *AspectWerkz 2 - Plain Java AOP*, http://aspectwerkz.codehaus.org/, (Accessed: 01 March 2006).

[79] Wikipedia, http://www.centipedia.com/articles/Louis_Paul_Cailletet, (Accessed: 23 January 2006).

[80] Sun-Microsystems, *Dynamic Proxy Classes*, http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html, (Accessed: 03 January 2006).

[81] Sun-Microsystems, W.C., *JSR 9: Federated Management Architecture Specification*, http://jcp.org/en/jsr/detail?id=9, (Accessed: 28 January 2006).

[82] Apache-Software-Foundation, *Logging Services: log4j Project*, http://logging.apache.org, (Accessed: 10 January 2006).

[83] Gill, P., *Probing for a Continual Validation Prototype*. 2001, Worcester Polytechnic Institute: Worcester MA USA.

[84] Rose, G.A., et al., *Object-Z (in Object Orientation in Z - pages 59-77)*. Workshops in Computing, ed. C.J.v. Rijsbergen. 1992: Springer Verlag, 3540197788.

[85] Smith, G., *The Object-Z Specification Language*. Advances in Formal Methods Series. 2000: Kluwer Academic Publishers, 0792386841.

[86] Derrick, J. and Boiten, E., *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology (FACIT). 2001: Springer, 185233245X.

[87] Derrick, J. *Timed CSP and Object Z.* in *ZB 2003: Formal Specification and Development in Z and B*. 2003: Springer.

[88] Mahony, B. and Dong, J.S. *Blending Object-Z and Timed CSP: An introduction to TCOZ.* in *20th International Conference on Software Engineering (ICSE'98)*. 1998. Kyoto Japan: IEEE Computer Society Press.

[89] Ratcliff, B., *Introducing Specification Using Z: A Practical Case Study Approach*. 1994: McGraw-Hill, 0077079655.

[90] Monin, F. and Hinchey, M., *Understanding Formal Methods*. 2003: Springer-Verlag, 1852332476.

[91] Sheppard, D., *An Introduction to Formal Specification with Z and VDM*. The Mcgraw-Hill International Series in Software Engineering, ed. D. Sheppard. 1995: McGraw-Hill, 1852332476.

[92] Jordan, D.T., McDermid, J.A., and Toyn, I. *CADiZ - Computer Aided Design in Z.* in *Workshops in Computing: Z User Workshop, Oxford 1990*. 1990. Oxford UK: Springer-Verlag.

[93] Jia, X., et al., *Z Type Checker (ZTC)*, http://venus.cs.depaul.edu/fm/ztc.html, (Accessed: 15 February 2005).

[94] Smith, G., *Object-Z Frequently Asked Questions*, http://www.itee.uq.edu.au/~smith/faq.html, (Accessed: 16 February 2005).

[95] Venners, B., *Jiniology: Locate services with the Jini lookup service - Discover the power and limitations of the ServiceRegistrar interface*, http://www.javaworld.com/javaworld/jw-02-2000/jw-02-jiniology.html, (Accessed: 20 April).

[96] Emmerich, W., *Encyclopedia of Software Engineering: OMG/CORBA - An Object-Oriented Middleware*, ed. J.J. Marciniak. 2002: John Wiley & Sons, 0471377376.

[97] Maes, P., *Computational Reflection*. 1987, Vrije Universiteit Brussel: Brussels Belgium.

[98] Newmarch, J., *Jan Newmarch's Guide to Jini Technologies*, http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml, (Accessed: 03 March 2006).

[99] Sun-Microsystems, *Jini Architecture Specification (Version 1.2)*, http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html, (Accessed: 12 January 2006).

[100] Sun-Developer-Network, S., *Technical Articles and Tips - JDC Tech Tips: May 30, 2000*, http://java.sun.com/developer/TechTips/2000/tt0530.html, (Accessed: 03 March 2006).

[101] Forman, I.R. and Forman, N., *Java Reflection in Action*. In Action. 2004: Manning Publications, 1932394184.

[102] AdventNet, *AdventNet SNMP API 3 - SNMP API Overview*, http://snmp.adventnet.com, (Accessed: 05 January 2006).

[103] Cohen, Y., *SNMP - Simple Network Managment Protocol*, http://www2.rad.com/networks/1995/snmp/snmp.htm, (Accessed: 10 January 2006).

[104] DPS-Telecom, *SNMP Tutorial Series: 5 Quick Steps to Understanding SNMP and its Role in Network Alarm Monitoring*, http://www.dpstele.com/layers/l2/snmp_tutorials.html, (Accessed: 08 January 2006).

[105] Sun-Microsystems, *Java Remote Method Invocation (RMI) - Activation Tutorials*, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/, (Accessed: 05 February 2006).

[106] Hadzilacos, V. and Toueg, S., *A Modular Approach to Fault-tolerant Broadcasts and Related Problems*. 1994, Dept. of Computer Science, University of Toronto: Toronto.

[107] Chandra, T.D. and Toueg, S., *Unreliable Failure Detectors for Reliable Distributed Systems*. Journal of the ACM, 1996. 43(2): p. 225-267.

[108] R Guerraoui, A.S. *Consensus Service: a Modular Approach for Building Fault-tolerant Agreement Protocols in Distributed Systems*. in *26th International Symposium on Fault-Tolerant Computing (FTCS-26)*. 1996. Sendai, Japan.

[109] Doudou, A., Garbinato, B., and Guerraoui, R. *Encapsulating Failure Detection: from Crash to Byzantine Failure Detection*. in *Reliable Software Technologies - Ada-Europe 2002, 7th Ada-Europe International Conference on Reliable Software Technologies*. 2002. Vienna, Austria: Lecture Notes in Computer Science, Springer.

[110] Doudou, A., et al. *Muteness Failure Detectors: Specification and Implementation*. in *European Dependable Computing Conference*. 1999: LNCS, Springer Verlag.

[111] Lee, C. and Helal, A. *Context Attributes: An Approach to Enable Context-awareness for Service Discovery.* in *3rd IEEE/IPSJ Symposium on Applications and the Internet.* 2003. Orlando Florida.

[112] Kahn, M.L., *The DARPA CoABS Grid Jini Performance Experiments (5th Jini Community Meeting)*, http://www-unix.mcs.anl.gov/gridforum/jini/JCM.coabs.experiments.12.11.00.pdf, (Accessed: 10 February 2006).

[113] Javangelist, *Reflection Performance*, http://javangelist.snipsnap.org/space/Reflection+Performance, (Accessed: 02 March 2006).

[114] Real, J.C., *Object-Z Specification of the CORBA Repository Service.* 1997, Université Libre de Bruxelles: Brussels.

[115] Llewellyn-Jones, D., et al. *Improving Interoperation Security through Instrumentation and Analysis.* in *First International Workshop on Interoperability Solutions to Trust, Security, Policies and QoS for Enhanced Enterprise Systems (IS-TSPQ 06).* 2006.

[116] Lupu, E., et al. *A Policy Based Role Framework for Access Control.* in *1st ACM/NIST Workshop on Role-Based Access Control.* 1995. Maryland USA: ACM.

[117] Lupu, E. and Sloman, M., *Conflicts in Policy-based Distributed Systems Management.* IEEE Trans. on Software Engineering, 1999. 25(Special Issue on Inconsistency Management): p. 852-869.

[118] Moffett, J. and Sloman, M., *Policy Hierarchies for Distributed Systems Management.* IEEE Journal on Selected Areas in Communications, 1993. 11: p. 1404-1414.

[119] Yoshihara, K., Isomura, M., and Horiuchi, H. *Distributed Policy-based Management Enabling Policy Adaptation on Monitoring using Active Network Technology.* in *12th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management.* 2001. Nancy France.

[120] Ganek, A.G. and Corbi, T.A., *The Dawning of the Autonomic Computing Era.* IBM Systems Journal, 2003. 42(1).

[121] Miseldine, P. and Taleb-Bendiab, A. *A Programmatic Approach to Applying Sympathetic and Parasympathetic Autonomic Systems to Software Design.* in *International Conference on Self-Organization and Adaptation of Multi-agent and Grid Systems (SOAS'2005).* 2005. Paisley Scotland UK.

[122] Miseldine, P. and Taleb-Bendiab, A. *CA-SPA: Balancing the Crosscutting Concerns of Governance Autonomy in Trusted Software.* in *International*

Workshop on Trusted and Autonomic Computing Systems (TACS-06), The IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006). 2006. Vienna University of Technology Vienna Austria.