Distributed components in a visualization environment
A Jini based distributed component architecture

Peter H.M. Jacobs
<p.h.m.jacobs@tbm.tudelft.nl>

February 11, 2002
Preface

The primary goal of Mark Hissink Muller, Niels Lang and myself during our internship at Sun Microsystems in California was to see how we could improve the visualization of 'Run-on-Sun' applications in Sun’s iForce Ready Center. This thesis will mostly describe a Jini based component architecture partly providing such.

Improving visualization means providing a more understandable view on questions like "How does this application work?" and "What may we expect under certain circumstances?" To reflect the importance of answering these questions in developing a visualization environment, this document is divided in the body focusing on these questions and appendixes describing implementations and a theoretical background. It must be said that the appendixes in this document are an integral part of this thesis. The more detailed, more technical and more delicate areas of interest are described there. Those readers not interested in bits and bytes better stay with the body!

Before starting with a description of the situational context, a few remarks can already be made about the title of this thesis. It speaks of a distributed architecture and therefore emphasizes on interoperability. The architecture is component based, which emphasizes on pluggability and flexibility. The necessity and implications of these terms will hopefully become obvious.

The first part of this thesis is written as a joint introduction with Mark Hissink Muller and Niels Lang. As of chapter 5 on page 29, this thesis reflects the individual implementation of what is described as the architectural framework.

Peter H.M. Jacobs
February 11, 2002, Delft, the Netherlands
Abstract

This thesis describes the result of the research conducted at Sun’s iForce lab in Menlo Park California on a component based anarchic peer to peer approach toward a distributed environment for visualization, simulation and control.

The research took place in iForce’s Trading exchanges group and was funded by Sun Microsystems. The project resulted from the cooperation between Delft University of Technology (TU Delft) and Sun Microsystems (Sun), and was carried out in the first half of 2001.

The basic underlying notion supporting the challenges faced during this research was to overcome the monolithic and often relational approach toward information systems in an age where globalization in business environments and globally distributed networks need the support of these systems more than ever.

The framework developed to support the distributed character of businesses, their control and manageability is designed in conjunction with Mark Hissink Muller and Niels Lang. This particular thesis describes the network topology, the interdependent, dynamic and often occasional relations between objects.

The first chapters introduce a global vision on the engineering fields involved, concluding with the requirements and perspectives on the distributed character in which we see businesses, organizations and software evolve. The trends toward communities both in transactions, development, and production of products combined with openness between organizations defined the design for our framework.

The individual part of this thesis focuses on the strengths of truly peer to peer development. A basic component is introduced providing the basic functionality all components in these networks share: look up and be looked-up, providing storage to intellectual behavior stored in subcomponents, provide a subscription model for a generic event model, serving the rules of being a good citizen in the network and administrating its behavior\(^1\).

Not only the basic component is introduced: The component is enriched with a library of subcomponents achieving a fairly complete reference implementation and defining the design patterns by which future development should occur. The last chapter describes the introduction of two business cases implemented with the framework.

One of these cases called Hercules\(^2\) became an open source project showing the strengths of computational computing based on the basic component sharing scenarios over several agents in the generation of loads to test Sun’s solutions. Some unique results of this approach were the 99% accuracy of the chained transactions and the flexibility by which large Sun servers and PDAs share scenarios not aware of the difference in their computational power.

\(^1\)Bootstrapping, destroying, movement, etc. etc
\(^2\)http://www.jini.org/projects/hercules
Contents

I Global introduction

1 Situational description ........................................ 8
   1.1 Sun Microsystems Inc .................................... 8
   1.2 iForce initiative ....................................... 9
   1.3 iForce Ready Center Menlo Park - Trade exchanges group .... 9
   1.4 Systems Engineering Group at Delft University of Technology .... 9

2 Global challenge description ................................ 11
   2.1 Objectives of the project ................................ 12
   2.2 Introduction report ..................................... 14

3 Theories involved ............................................. 16
   3.1 Decisionmaking theory ................................... 16
      3.1.1 Decisions as a mean to solve a problem .......... 16
      3.1.2 Requirements for an information architecture ... 18
   3.2 Systems engineering ..................................... 19
   3.3 Organizational sciences .................................. 20
   3.4 Business engineering .................................... 21
   3.5 Software engineering .................................... 21

4 The introduction of a framework for visualization, decision support and control .... 24
   4.1 Virtual representation of real-life objects .............. 24
   4.2 Anarchic topology based on a standard component ....... 26
   4.3 Visualization, simulation and control ................... 27

II Distributed components in a visualization environment

A Jini based distributed component architecture .......... 28

5 Introduction .................................................. 29
   5.1 A system engineering problem? ......................... 29
   5.2 Introduction of report ................................... 30
# CONTENTS

6  Component model ........................................ 31
    6.1  Theoretical concepts .................................. 31
      6.1.1  Divide et impera .................................. 31
      6.1.2  Event based approach ................................ 32
      6.1.3  The network is the computer ......................... 33
    6.2  A dynamic topology combined with flexibility in use ..... 34
    6.3  Requirements of the basic component .................... 35
    6.4  Underlying technologies ................................ 36
    6.5  Design of the basic component .......................... 37
      6.5.1  Functionality 1: Defining the component and providing storage to functional behavior ........ 37
      6.5.2  Functionality 2: How to address other components .......... 39
      6.5.3  Functionality 3: Accepting remote listeners ........... 42
    6.6  Extended overview and concluding remarks ................ 43

7  Designing functional behavior ................................. 45
    7.1  Modular design choices .................................. 46
    7.2  The component library .................................... 47
    7.3  Recommendations ......................................... 49

8  Proof of concept ........................................... 50
    8.1  CASE 1: Data Management Center .......................... 50
      8.1.1  Wrapping the Oracle instances ......................... 52
      8.1.2  Linking to the GUI .................................... 53
      8.1.3  Conclusions ........................................... 53
    8.2  CASE 2: Project Hercules ................................. 54
      8.2.1  Multi threading and fixed loads ....................... 54
      8.2.2  Flexible amount of threads and a steady load .......... 55
      8.2.3  Dynamic rescaling ..................................... 55
      8.2.4  Scenarios and configuration files .................... 55
      8.2.5  Recommendations ....................................... 56

III  Distributed components in a visualization environment

Appendices .................................................... 59

A  The Proof of Concept trajectory ............................... 60

B  From TCP/IP to RMI & CORBA ................................ 62
   B.1  Introduction .............................................. 62
   B.2  TCP/IP suite and Java networking ........................ 63
     B.2.1  TCP/IP Suite ........................................... 63
     B.2.2  Java networking ......................................... 63
   B.3  Remote Method Invocation ................................ 64
     B.3.1  Basic RMI ............................................... 64
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.3.2 Dynamic class loading</td>
</tr>
<tr>
<td>B.4 CORBA</td>
</tr>
<tr>
<td>B.5 Overlap between CORBA and RMI</td>
</tr>
<tr>
<td>C Jini Technology</td>
</tr>
<tr>
<td>C.1 Discovery and Join Protocols</td>
</tr>
<tr>
<td>C.1.1 Discovery protocol</td>
</tr>
<tr>
<td>C.1.2 Join protocol</td>
</tr>
<tr>
<td>C.2 Jini lookup service</td>
</tr>
<tr>
<td>C.3 Jini distributed events</td>
</tr>
<tr>
<td>C.4 Jini distributed leasing</td>
</tr>
<tr>
<td>C.5 Jini distributes transactions</td>
</tr>
<tr>
<td>C.6 JavaSpaces</td>
</tr>
<tr>
<td>C.7 What’s left</td>
</tr>
<tr>
<td>D OO queries based on Marshalled Objects</td>
</tr>
<tr>
<td>D.1 Marshalled objects</td>
</tr>
<tr>
<td>D.2 Constructing queries</td>
</tr>
<tr>
<td>D.3 Disadvantages</td>
</tr>
<tr>
<td>D.4 An extended approach</td>
</tr>
<tr>
<td>E Java XML Binding</td>
</tr>
<tr>
<td>E.1 Introduction</td>
</tr>
<tr>
<td>E.2 XMLSchemes</td>
</tr>
<tr>
<td>E.2.1 Namespaces</td>
</tr>
<tr>
<td>E.2.2 Primitive, simple and complex types</td>
</tr>
<tr>
<td>E.2.3 Attribute or value</td>
</tr>
<tr>
<td>E.3 Binding XML to Java</td>
</tr>
<tr>
<td>E.3.1 Binding flow</td>
</tr>
<tr>
<td>E.3.2 Mapping element types to classes</td>
</tr>
<tr>
<td>E.3.3 Interfaces</td>
</tr>
<tr>
<td>E.3.4 Setting the bean properties</td>
</tr>
<tr>
<td>E.4 Conclusions</td>
</tr>
<tr>
<td>F Jini HLA overview</td>
</tr>
<tr>
<td>F.1 Goals</td>
</tr>
<tr>
<td>F.1.1 HLA Goals</td>
</tr>
<tr>
<td>F.1.2 Jini Goals</td>
</tr>
<tr>
<td>F.2 Dictionary</td>
</tr>
<tr>
<td>F.3 Services</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Development of framework ........................................ 15
3.1 Relation between decision and system (after [Bots, 1997]) ............. 17
3.2 Information flows ................................................. 19
3.3 Waterfall model of software life cycle .................................. 22
3.4 Java community process ............................................ 23
4.1 Virtual representation of real time components .......................... 25
4.2 Virtual representation of real time components .......................... 25
4.3 Topology of components in the framework ............................... 27
6.1 Different network topologies ........................................... 34
6.2 Storage structures I .................................................. 37
6.3 Storage structures II ................................................ 38
6.4 Storage Structures III ............................................... 40
6.5 Passport .............................................................. 41
6.6 Object Oriented query ............................................... 42
6.7 Class diagram of JiniComponent ..................................... 44
7.1 Requesting a service .................................................. 46
7.2 Library of sub components ............................................ 48
8.1 Data Management Center ............................................. 51
8.2 Jini component use case .............................................. 52
8.3 Worker. run() ......................................................... 56
8.4 Hercules rescaling mechanism ......................................... 57
A.1 PoC-trajectory: overview ............................................... 60
B.1 Lowest network layers ................................................. 62
B.2 Sockets ............................................................... 64
B.3 RMI interface ......................................................... 65
B.4 RMI communication .................................................. 65
B.5 Dynamic downloading machine class .................................. 66
B.6 Dynamic downloading server class ................................... 67
LIST OF FIGURES

B.7  Dynamic class downloading ............................................ 68
B.8  CORBA communication .................................................. 69
C.1  All network layers ....................................................... 70
C.2  Unicast discovery example ............................................. 73
C.3  Remote event mechanism ............................................... 75
C.4  Third party event handler ............................................. 76
C.5  Distributed leasing ...................................................... 77
D.1  Serializability ........................................................... 82
D.2  Object Oriented query .................................................. 83
D.3  Operator object in the extended OO queries ......................... 83
E.1  XML instance without namespaces .................................. 85
E.2  XML instance with namespaces ....................................... 86
E.3  Restrictive primitive type .............................................. 86
E.4  XML instance with attribute ......................................... 87
E.5  Activity diagram of UnMarshall.java ................................. 88
E.6  XML Schema with Java references ................................... 89
E.7  Construct new Java instance based on XMLSchema .................. 90
E.8  Java interface ............................................................ 90
E.9  XML schema based on Java interface ................................. 90
E.10 XML instance based on Java interface ............................... 90
E.11 Java code parsing interface .......................................... 91
E.12 Setting the beanproperty ............................................. 91
F.1  Runtime infrastructure ................................................ 94
F.2  Jini Services ............................................................ 95
Part I

Global introduction
Chapter 1

Situational description

This report is a general introduction of a six month project carried out at Sun Microsystems' iForce Ready Center in Menlo Park, CA, USA. This project took place in iForce's Trading exchanges group and was funded by Sun Microsystems. The project resulted from the cooperation between Delft University of Technology (TU Delft) and Sun Microsystems (Sun), and was carried out in the first half of 2001 by three students of the TU Delft's faculty of Technology, Policy and Management. The initial objective for this project was to develop a visualization system to leverage iForce's ability to provide customers with more insight in the benefits that Sun's systems could realize for their (E-)business. This resulted in the development of a general framework for distributed visualization & control, that's introduced in this report.

The project was coached by prof. dr. H.G. Sol, dr. ir. A. Verbraeck and prof. dr. R. Wagenaar on behalf of the faculty of Technology, Policy and Management and by drs. L. Bonebakker on behalf of the iForce Ready Center at Menlo Park. The following paragraphs introduce Sun Microsystems, the iForce initiative, the Trading exchanges group and the Systems Engineering Group at Delft University of Technology.

1.1 Sun Microsystems Inc.

Sun Microsystems, a $20 billion company founded 18 years ago with offices in 170 countries, provides end-to-end solutions for doing business in the network age. Sun's products and services cover all the aspects of ICT, from reliable and scalable high-end enterprise servers, to the platform independent and object-oriented Java programming language.
1.2 iForce initiative

The iForce initiative within Sun as a part of Global Technical Operations aims to provide solution blueprints for mission critical applications that Run-On-Sun to deliver sustained business advantages for an enterprise.

The blueprints are not only designed, but also tested for the customer by means of the Proof of Concept trajectory. During the Proof of Concept trajectory the critical parts of the customer's solution will actually be built to be able to 'prove' the performance of the proposed blueprint.

The iForce Ready Centers consist of various groups with different focus. There are product-specific groups like the SAP-group, the Oracle/PeopleSoft and the iPlanet-group, as well as application-specific groups like the Supply Chain Management group and the Trade exchanges group, where this project was conducted.

1.3 iForce Ready Center Menlo Park - Trade exchanges group

The Trade exchanges group within iForce focus's on large e-Business problems that involve trade exchanges, also known as virtual marketplaces. Many fortune 100 enterprises coming to the lab get Sun's help on realizing maximum benefit from the opportunities today's networked digital economy offers. Because of the complexity in technology involved and the fact that trade exchanges are such a new subject, very specific expertise is necessary to get an E-business system that involves a trade exchange working. In the iForce lab proof of concept is established for these complex cases.

1.4 Systems Engineering Group at Delft University of Technology

Founded in 1864, Delft University of Technology is the oldest, largest, and most comprehensive technical university in the Netherlands. With over 13,000 students and 2,100 scientists (including 200 full-time professors), it is an establishment of both national importance and significant international standing. Renowned for its high standard of education and research, the University collaborates with other educational establishments and research institutes, both in the Netherlands and overseas. It also enjoys partnerships with governments, branch organizations, numerous consultancies, the industry, and companies from the small and medium business sectors.

Systems Engineering (SE) is a research group within the faculty of Technology, Policy and Management (TPM). As such its main strengths are in multidisciplinary projects. Central research issues for TPM are the process of problem analysis & solvation and complex design trajectories. One of the research directions for SE is ICT, focusing on E-commerce applications in different organizational settings.
1.4 Systems Engineering Group at Delft University of Technology

Group technology and simulation tools are being developed, tested and used within SE to conduct this research. A current research theme, in which these tools are used, is the modeling of complex systems, including their dynamics and control with component based architectures.

This report provides a general introduction on the framework that has been developed during the project. The details of the framework are discussed in three separate reports, each one produced by one of the authors.
Chapter 2

Global challenge description

Today’s business environment has changed rapidly over the past few decades. The basic trend driving this change is the process of globalization, which started with the disappearance of many traditional national economic boundaries during the last few decades. The process of globalization has been accompanied and accelerated by revolutionary developments in the Information and Communication technology, like the World Wide Web and the Internet in general. Information and Communication technology has become an indispensable instrument for companies to conduct business in large, differentiated, international markets.

The interaction between globalization in business environments and the shifting in technology toward globally distributed computer networks has caused many changes, among which:

- Upscaling of operations: Most multinational companies have merged during the last decade. Due to the global markets, managers often suggest that not being one of the top five eventually implies losing your business. The information and communication age is therefore focused on the integration of information from internationally diverse sources into management and production valued.

- Partner communities: Companies are becoming more aware of the increased number of potential partners. Most companies focus on their core technology and therefore seek a community of partners to achieve mutual gains in serving the customer.

  This community-approach toward customers is especially reflected in the current modularity of supply chains. Consequences of these peer-to-peer relations in business environments are the flexible way in which relations are maintained.

- Changes in information needs: The complex interaction of partners, customers and competitors, which was once enabled by advanced information systems, now in turn is driving the demand for even better information systems. Increased complexity and speed in the business environment has increased the
need for fast decision making, accurate information. Although current information systems are able to generate an impressive amount of data, managers keep complaining that they’re unable to get the right information to steer their companies.

- Service oriented solutions versus products: The openness of markets, the ability to compete on global markets and the complexity of changing interactions, knowledge and partners has lead to a more service oriented approach. Customers hardly aware of the details of their own problems seek tailored solutions instead of products to develop them. This business change has also caused a reaction in ICT. Sun’s Open Net Environment (ONE) vision, for example, focuses on the development and deployment of web services rather than applications.

Sun Microsystems and the iForce Ready centers have always focused on supporting the business environment in a way that reflects its distributed character. It was Scott McNealy, CEO of Sun, who restated in 1994 the vision formulated at the company’s establishment in 1982: the network is the computer. Sun has proudly preserved its image as being the dot in dot-com and in being the backbone of the internet focusing on integration, manageability and scalability.

Although Sun’s hardware is more and more focused on the complexity and flexibility of the information systems it supports\(^1\), the information systems (e.g. the software) themselves are still based on monolithic approaches.

The aforementioned changes of the business environment in combination with Sun’s vision of the Network as the Computer, has formed the basis for the iForce Initiative. With the iForce initiative, Sun has bundled its own knowledge with its global industry partners to be able to present its customers proven, best-of-breed solutions to support the customer’s business. However, it proved to be difficult to demonstrate to the customer the workings of Sun’s solutions, and the way they relate to the customer’s complex business reality. Therefore, based on the TU Delft’s experience with simulation and visualization, this project was set up to leverage iForce’s ability to demonstrate their solutions to their customers. In the following section the objectives of the project will be introduced in more detail.

### 2.1 Objectives of the project

The initial objective of the project was to leverage iForce’s ability to demonstrate the workings of Sun’s systems to the customer. Since Sun’s systems distinguish themselves by virtue of technical aspects like scalability, throughput and availability, Sun would like to demonstrate exactly those aspects of its system to the visitors of the iForce. It is iForce’s experience that customers’ approach presentations of

\(^1\)The federative domain approach in the online reconfigurable midframe line is just one the examples supporting this notion

\(^2\)iForce’s customers usually are CEO’s of large organizations
2.1 Objectives of the project

technical aspects skepticaly, because it is hard for the customer to judge the correctness of claims on technological performance. This skepticism is often amplified by the fact that technological specifications are usually obtained for some reference implementation, not the specific implementation needed by the customer.

It is therefore that the iForce uses a different approach toward demonstration of their systems. Their starting point is that the demonstrated system should be a physical implementation that could actually be deployed for the customer. In the Proof of Concept trajectory, the critical components of a customer's concept implementation will actually be built and tested in the iForce, by both Sun's and the customer's engineers. Still, it remains difficult to demonstrate the performance of a working implementation, since the customer may see only some flashing lights on some boxes that imply the performance of the system, but do not demonstrate it. Therefore, the technical performance of these implementations should not only be presented to the customer, but rather be derived directly from this implementation, preferably by the customer himself. The customer should be able to interact with the implementation and be able to monitor the workings of the implementation. Performance information could be visualized to enhance the communicatability of the information. This implies a visualization of the implementation performance that's fed by the implementation itself. It also implies a flexible and reusable visualization system, since iForce receives many customers and therefore needs to demonstrate many implementations.

A practical case in which iForce's challenge was indeed to add visualization and interaction to enhance the communicatability of the technical aspects of one of Sun's solution was the Data Management Center (DMC) case. The Data Management Center enables a customer's production system to perform a backup without interruption of the production system. This is a typical case where iForce wants to show the customer that the solution really works. The case has been served as a test case during the project and is presented in [Jacobs, 2001] and [Lang, 2001].

During the project, the first outlines of the framework, built to meet the first objective, also seemed applicable to an objective of Sun's customers. The typical customer administers a large network of ever more interrelated electronic devices (consisting of Point of Sales terminals, computers, production machines, etc.). Since these devices take part in the primary processes of a customer's organization, customers feel that the information produced by these devices could support them in the process of decisionmaking with regard to the organization. A new objective therefore became to:

- leverage an actor's ability to open up the information available in a network of electronic information producing devices in order to improve the ability to make decisions (regarding that network).

The main ingredients of a solution for this objective resemble those mentioned before: a link between visualization and real system, flexible visualization to handle changes in the underlying system and interaction with the system to provide remote control where possible. On first sight, also some differences may be noted: the visualization should focus on business rather than technical performance of the
underlying system. Also, this objective is a challenge for the customer rather than a direct challenge for iForce. Sun’s solutions, however, seemed to be a key enabler to meet this challenge. The relevance of this objective for iForce is further illustrated by the eServiceStation case: visualizing a business case consisting of more than 20000 petrol stations with the objective to control their manageability.

2.2 Introduction report

Based on both a deductive theoretical as an inductive case based approach, a conceptual framework is developed to meet the objectives formulated.

Using Sun’s technologies in combination with TU Delft’s experience in the field of simulation, a reference implementation of the conceptual framework has been built. This reference implementation functions as a demonstration tool to illustrate important concepts of the framework developed. However, the reference implementation could also form the basis for a research implementation to test the usability of the conceptual framework on a real-world case (for example a complex supply chain).

This report gives a general introduction on the development of the conceptual framework. Figure 2.1 on page 15 illustrates the way this report has been organized.

Chapter 3 on page 16 introduces the concepts on information used in the project. These concepts have been classified into several categories, that are introduced and illustrated by the theoretical and practical notions they’re based upon. This chapter also describes the set of requirements based on these theoretical engineering fields.

The framework introduced in chapter 4 on page 24 focuses on several aspects among which the representation of organizations, actors and devices in the architectural framework, the topology of information systems, the way services are provided and/or leased and the role of visualization, simulation and wrapping legacy devices.

Accordingly to these themes the chapter presents a short overview of the individual responsibilities in implementing the framework and therefore presents a short introduction in the three individual theses'.
2.2 Introduction report

Figure 2.1: Development of framework
Chapter 3

Theories involved

Based on the objectives stated in the previous chapter, this chapter focuses on a theoretical foundation of the engineering fields involved: theory on decision making, organizational theories, business engineering, system engineering and software engineering.

The main promise of this chapter is to present a short overview of theories and engineering fields involved to be able to introduce the resulting framework in the next chapter. The fields involved are therefore presented as equal in focus, hierarchy and without mutual relations. This chapter focuses on the changing aspects due to globalization, networked technology on the different fields described.

3.1 Decisionmaking theory

Sun’s customers regard their information systems ever more as a decisionmaking support tool in addition to the traditional information database. This section therefore discusses some theoretical notions on decisionmaking and their implications for an information infrastructure. A model of decisionmaking is introduced that focuses on decisionmaking as a (continuous) process driven by a certain problem perception. The section concludes with some distilled requirements for an information system providing decisionmaking support.

3.1.1 Decisions as a mean to solve a problem

In [Bots, 1997] decisions are thought to emerge out of an underlying problem perception of an actor, the problem owner. It therefore introduces an analytical model explaining the relationship between a single, rational actor, a problem that’s perceived by this actor and the relationship this actor has with regard to the observable system in which the problem occurs. A graphical view of this model is presented in figure 3.1. This model assumes that the world can be regarded as a system, a collection of elements and relations surrounded by an environment. The model illustrates that the problem owner should:
3.1 Decisionmaking theory

Figure 3.1: Relation between decision and system (after [Bots, 1997])

- Be aware of his problem perception. The problem perception may for example be formalized by means of an objective hierarchy or means-ends model.

- Observe a specific part of the world, which is bounded by and related to the problem perception. This part of the world will be identified as the relevant system. The whole system that can be observed by the actor will be identified as the observable system.

- Be aware of the way instruments and environment influence the relevant system. This implies that a model of the relevant system is available for the problem owner to enable him to predict in advance the consequences of a certain course of action for a given environment behavior.

- Have enough instruments to act on the system. The problem owner should at least have enough instruments at his disposal to counterbalance the environmental impacts on the relevant system.
3.1 Decisionmaking theory

The aforementioned consequences correspond with the five necessary conditions for effective control

- Clear objectives of the control process.
- Availability of a model of the controlled system.
- Available information on the state and environment of the controlled system.
- Availability of enough steering instruments.
- Availability of sufficient information and communication handling capacity.

The last condition follows from the need to process state changes of the relevant system fast enough in order to be able to apply suitable control mechanisms in time.

3.1.2 Requirements for an information architecture

The illustrated decisionmaking model implies the following consequences for an information architecture providing decision support:

- The actor should be able to put boundaries on the observable system in order to limit the amount of information received by the actor. The need for such a limitation of the received information is shown figure 3.2. The actor is only interested in the information in the relevant system and therefore needs to filter the superfluous information. The theory of systems suggests several dimensions for bounding systems, under which the following:

  - Definition of a subsystem, consisting of a subcollection of elements of the original system (including the attached relations). The elements of the subsystem are usually characterized by a certain condition. All elements may for example be located in a specific geographical area.
  
  - Definition of an aspect system, consisting of a subcollection of relations of the original system (including the attached elements). The relations of the aspect system are usually characterized by a certain condition. All relations having to do with information flows may for example be included.
  
  - Definition of a phase system, in which a system is observed during a certain time-interval\(^1\). The chosen time-interval is usually defined by a certain condition, for example maximum load on a system.

- The actor should be able to monitor the state of the system and its environment and be notified of changes, in order to be able to react accordingly, given the problem perception.

\(^1\) Once or periodic
3.2 Systems engineering

![Diagram of systems engineering](image)

Figure 3.2: Information flows

- The system information should be visualizable in such a way that it gives the actor the most insight in the relevance (and significance) of the information for the problem perception. In this way, the information and communication handling capacity of the actor may be leveraged. Since problem perceptions as well as information types differ, various ways of information visualization should be available for the actor.

- The actor should be able to use the information available of the observable system as a basis for a descriptive or predictive model of the system. If this model is to be a computer model, this implies that information is available in a general computer-readable form.

- The actor should be able to act on the system by use of his instruments. The information system could support this for some instruments by sending control information to parts of the relevant system.

3.2 Systems engineering

System engineering is the engineering field focusing on hierarchical decomposition of systems. This engineering field is not described to result in requirements for the framework but justifies further models by introducing A way of thinking. Decomposability is achieved to reduce complexity with respect to insight in (parts) of the system.

The modular hierarchical decomposition can be achieved in the following ways:

- Phase systems regard the system only during specific time frames.

- Sub models and aspect systems only regard specific parts of a system.
3.3 Organizational sciences

- Further decomposition is achieved by separating processes from objects. Objects themselves are characterized by the object oriented approach of aggregation and specialization.

A further notion on system engineering is the difference in soft- and hard systems. Hard systems only regards the objects and relations that can be quantified. Soft systems focus as well on soft elements, values, relations and views. Due to the \textit{Way of support} chosen for this framework\(^2\) a hard approach is chosen.

3.3 Organizational sciences

While in the field of organization & management the authoritative view on the decisionmaking context used to be based on a hierarchical structure of the actors involved, nowadays the view that decisionmaking takes place in networks of actors is gradually receiving more attention.

In \cite{deBruijn1999} it is stated that the main characteristics of these networks are pluralism, mutual dependencies of the actors involved, closedness of actors and dynamics.

- \textit{Pluralism} refers to the variety of the actors involved in a typical decision-making network. Actors may differ in several dimensions like size, function, available resources, services & products offered or authority. Pluralism may manifest itself between different organizations (inter-organizational pluralism), but also between suborganizations within one organization (intra-organizational pluralism).

- \textit{Mutual dependencies}: In the classical view actors are subordinate to other actors, in an invariable, hierarchical structure. In this view the subordinate actors are completely dependent upon their superiors. In contrast, the starting point of the network view is that the relationship between actors is better described by a relationship of mutual dependencies. One of the factors that cause a mutual rather than one-sided dependency relationship is the fact that actors are involved in multiple processes of decisionmaking. Because different types of decisions are involved, the dependency relationship may also differ from one process to another. In the end this mechanism will dampen the formal power of a formally superior actor.

- \textit{Closedness}: The classical view on decisionmaking processes assumes the availability of all relevant information. In the network view, however, information and resources are thought to be distributed among the actors involved. Moreover, these actors are assumed to treat these resources strategically: they will not provide external access to resources if this harms their interests. This mechanism can be identified as exposure closedness. Actors can also display reception closedness, e.g. they do not respond to external signals.

\(^2\) Using software as a solution
3.4 Business engineering

- *Dynamics:* The classical view on decisionmaking processes assumes a static decisionmaking context: it is clear which actors are involved, what resources are available, what the current decision is about and what the dependency relationships between the actors are. In contrast, the network view assumes these aspects all to be dynamic in nature.

Several requirements for the framework based on these organizational theories and resulting from the field of System engineering are:

- The framework must support the richness of actors, their interdependencies and their often dynamic relations. Since transactions in real-life are often based on complex and coincidental networks of actors, the framework must support these concepts and the corresponding peer to peer relations.
- The actors in the framework must be self empowered in the sense that they must be able to collect information and (re)act on it on an independent instigation. Empowerment also suggests controlling the amount of service or information an actor provides to others.
- The architectural framework must represent a correct and relevant overview of the real-life global chains of networks and businesses.

3.4 Business engineering

Business engineering is the engineering field focusing on business process (re)design in conjunction with (technology) solutions based on the strategy of an organization. Global markets and the current standard of technology made businesses seek improvements in these processes by questioning their efficiency and manageability. Current trends focus on decentralizing these processes and therefore assume more locale and accurate information on different environments. An example of this trend is the *Produce to order* concept which nowadays gains a lot of interest by the European car industry.

The main requirement for the framework due to the need of supporting insight in business processes and potential improvements is to visualize the business processes in the framework. Since business process (re)design is clearly a goal of the framework, interdependencies with other systems like simulation and visualization packages should be easily established.

3.5 Software engineering

The basic promise of this section is to explicit the current ideas in software engineering and therefore to form an understandable basis for the approaches chosen for the implementation of what is been presented as the framework.

Software engineering is the field of computer science that deals with the building of software systems which are so large or so complex that they are built by a team or teams of engineers.[Ghezzi, 1991]
3.5 Software engineering

[Ghezzi, 1991] introduces a firm basement for software engineering by the waterfall model of software engineering illustrated in figure 3.3.

![Waterfall model of software life cycle](image)

Figure 3.3: Waterfall model of software life cycle

The waterfall model of software engineering defines the lifetime of software, focusing on the distinguishable phases.\(^3\)

Though this model has served most software development for the last decades, a current shifting is seen toward community development of software.

In line with the organizational networks, the globalization of the business environments and the strengths of global interaction and communication, companies like Sun Microsystems, IBM, Oracle, etc. have gained a far greater interest in a community process of software engineering and therefore to partner in development. The single actor perspective of the waterfall model has changed into an inter-organizational approach not focusing on phases but on rounds, not focusing on project but on process.

Figure 3.4 gives an example of this approach in the Java community process. This process describes how expert groups of several companies initially formalize a challenge or goal for the improvement of any Java library/solution. These expert groups are inter-organizational and present their ideas to the public, which can react on their proposals.

Key consequence of this community approach is the need for generic reusable solutions, which are not focused on the expectations of a single actor, but serve communities and hopefully ones to come.

\(^3\)Conceptual phases are often referred to as the high phases, where implementation phases are referred to as the low phases.
Figure 3.4: Java community process
Chapter 4

The introduction of a framework for visualization, decision support and control.

This chapter introduces the framework as it is designed according to the previously stated requirements. It is merely a short and mainly visual illustration of what may be expected in the three different thesis.

As detailed issues on implementation, design and theoretical notions are covered independently this chapter forms the end of the global part and has hopefully triggered the interest on all three further documents.

4.1 Virtual representation of real-life objects

The part dealing with the actual business case and its representation in the framework is covered by [Muller, 2001] in his paper titled: Component based framework for multi actor decision support and control.

Figure 4.1 presents the dynamic core components of the framework. Real devices or simulated devices are wrapped to a virtual representation called a virtual device.

Actors have a virtual representation in the framework called Virtual Actor, which allows to interact with the system. Actors can control1 one or more Virtual Devices by sending a Business Action to a Business Rule.

The Business Rule takes care of the invocations with specific Virtual Devices/real devices. When a certain condition is met that the actor is interested in, the Filter notifies the Virtual Actor by sending out a Business Event.

Figure 4.2 shows how filters are updated by virtual devices on state changes and how the messages are forwarded to virtual actors. All dynamic components

---

1 invoke an action which causes a state change
4.1 Virtual representation of real-life objects

![Diagram](image-url)

Figure 4.1: Virtual representation of real time components

![Diagram](image-url)

Figure 4.2: Virtual representation of real time components

in the framework extend functionality from a component called NaradComponent. The NaradComponent contains a few (in the current implementation standard) sub-components which offer basic functionality like a-synchronous communication
4.2 Anarchic topology based on a standard component

and security through the use of cryptography.

Message objects
Non-moving object in the framework are able to communicate by exchanging messages or moving objects. An actor, for instance, is able to interact with a Filter by sending it a signed 'Request For Rights'. An actor sends a signed 'Request For Rights' to an instantiated filter which places a listener on a Virtual Device. A Filter communicates with an Actor by sending him/her a Business Event.

4.2 Anarchic topology based on a standard component

The previous section has described how real-life objects are translated to their corresponding virtual representation in the framework. This description has only focused on the semantics and more logical translation. How these components relate to each other in ever changing networks, in the upmost efficient and flexible way based on the requirements stated in the previous section, formed the basis for research on distributed anarchic topologies. This research is presented as *Distributed components in a visualization environment* by [Jacobs, 2001].

Figure 4.3 on page 27 gives an overview of resulting topology of this research and therefore of the topology of the framework. The main characteristics are:

- The topology of the framework support multiple organizations or federations, each possibly consisting of multiple groups. Each group or federation has clear boundaries, but connections between them may be established by components.

- Since the topology is dynamic one must assume redundant components and complex but partly hidden relations in providing business logic to other components.

- Components or services within the framework are arranged in a chaotic order which implies the creation, movement and disruption of components and their mutual relations.

- The topology is based on a peer-to-peer approach focusing on the equal base on which components communicate. There are no conceptual differences between components in the perspective of the topology of the network.

- The lack of centralized management implies that every component must be able to bootstrap itself, provide intelligence to its surrounding and must be able to lookup its neighbors.

- The outline color of the components points to the basic interface each component in the framework extends. These basic components serve as a module providing most of the above abilities.
4.3 Visualization, simulation and control

Visualization, simulation and control formed the basis for the research titled *Simulation and distributed visualization* by [Lang, 2001]. Keynotes on this research will be added to this section by [Lang, 2001] shortly.
Part II

Distributed components in a visualization environment

A Jini based distributed component architecture
Chapter 5

Introduction

This part describes the individual research conducted on the topology of the distributed components based on the challenge description, theory and requirements stated in the global part I. Before starting with an overview of this report, a short summary is presented on the framework, its status and requirements.

Based on the representation of real-life objects into the virtual representation of the architectural framework illustrated in figure 4.1 on page 25, this thesis describes the results of the research conducted on a globally distributed environments and introduces a basic component providing the functionality all components share.

Key concepts are to focus on dynamics, anarchy and peer to peer relations by which the real-life objects relate to each other. Supporting this existence and these relations is seen as a key aspect of enriching the information system by not reducing complexity to centralized and often relational solutions.¹

The main research question is therefore how to achieve a topology as illustrated in figure 4.3 on page 27 and how to use this mindset of decentralized, distributed topology within the services provided by all components within the framework.

5.1 A system engineering problem?

The first section of this introduction describes the relation between the fields of system and software engineering and organizational sciences.

The first question is how to see current business systems, transactions and actors software systems are supposed to serve. Should we consider the problem description to which a software system is applied to as one system consisting of several subsystems each perhaps representing an organization? Another vision would be to focus on the diversity of organizations and therefore to introduce a web of systems and subsystems. In this vision a soft approach toward organizations is the founding concept.

¹Complexity is defined as a function of the number of elements (components) and the number of interdependent relations.[Bots, 1997]
5.2 Introduction of report

The different roles of system engineering and organizational sciences as leading foundation for software is roughly illustrated by the agent technology versus monolithic ERP systems. Though the second is firmly integrated in current businesses, neither have resulted in manageable software systems serving real-life problems.

The faculty of Technology, Policy and Management and especially the section of System engineering focus on decomposition in sub systems with the explicit goal of modelling, serving, understanding and managing complex real life problems. What will be described in the rest of this document is the result of research conducted on the nexus of organizational sciences and system engineering translated into a software solution. The goal of this research was clearly to focus on a topology in the middle between a one-system and the absolute agent approach.

5.2 Introduction of report

The first chapter of this report, chapter 6, starts by a reflection of the previous stated requirements into more concrete requirements used for the design of the network and the basic component providing the services all components share.

As will be outlined, all intelligence, behavior and functionality will be dynamically added or removed by the use of sub components. The design patterns of the functional behavior of services as well as an overview of the library developed so far is described in chapter 7 on page 45.

A proof of concept of the basic component in conjunction with the library of services is presented in chapter 8 on page 50. This chapter starts by implementing the network, basic components and appropriate functionality for the DMC case. A second case used to test the strengths and weaknesses of the component and its services is presented in section 8.2 on page 54.

Chapter 7? forms the final chapter of this thesis by describing the conclusions and recommendations of the research done at Sun’s iForce center. As stated in the preface, the appendixes are included as theoretical background for this research and present the more delicate software concepts.
Chapter 6

Component model

What is described so far emphasizes the distributed character of the bridge from organizations, information and technology. Actors, actions, transactions, storage and usage of information, etc. all seem to take place in globally connected networks.

The challenge of this thesis is whether a basic component can be provided as foundation of the architectural framework, that interacts with the closedness of the networks, supports the distributed interdependencies and enriches insight in the complexity of the information model by hiding implementation details.

The strength of this basic component would therefore be that in extending it, a virtual actor, device or wrapper will inherit a transparent and generic layer of functionality.

As this foundation or basic component will reflect the current standards of software and software engineering a short overview of concepts is introduced.

6.1 Theoretical concepts

This section briefly explains some concepts on software, information models and implementations to support the notion that the basic component and the architectural framework itself are new and based on the current insights on the engineering fields involved.

6.1.1 Divide et impera

The first principle of software engineering discussed here is the concept of modularity. First a few formal notions on modularity are explained.

The main benefit of modularity is that it allows the principle of separation of concerns to be applied in two phases; when dealing with details of each module in isolation and when dealing with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system. [Ghezzi, 1991]
6.1 Theoretical concepts

To achieve modular composability, decomposability, and understandability, modules must have high cohesion and low coupling. [Ghezzi, 1991]

Modules are often referred to as software components. The blueprints of a modular approach in designing software are therefore referred to as a component based architecture. Two consequences of a modular or component based approach are:

- **Interchangeably**: The notion that modules may be replaced by better or newer versions is of great importance. Software engineers referring to this concept, often only refer to interchangeability in development. What is introduced in the rest of this thesis will focus on interchangeable components in runtime. We will introduce the concept of dynamic interchangeable component behavior in a globally distributed network.

- **Independently**: The main question of the formal notions on components is how to achieve this modularity and therefore how to achieve low coupling. First of all low coupling is achieved by the distinction of a component and its interface. Rigid boundaries and interfaces are not enough though. Especially in a distributed environment a component must be able to cope with disconnected networks, unavailable partners, failing hardware, conflicting events, etc to remain reasonably independent. What makes it even more difficult is that implementations of these requirements must follow the same modular and independent approach.¹

6.1.2 Event based approach

Before introducing perspectives on events, this subsection begins with a number of formal definitions of an event.


As will be made clear in the following chapters, events are far more important than suggested in the definitions stated here. A field which has proven a great interest and adoption of all several event mechanisms is the field of discrete event simulation in which objects place activities on future event lists. This familiarity with events will hopefully prove the easiness of use of these models in the framework explained. More on discrete event simulation and its usability in conjunction with this framework can be found at [Lang, 2001].

The reason why this concept is not formalized in clear definitions is because an event must be studied in conjunction with its source and its destination.

¹Concepts must be easily translated among programming languages and operating systems.
6.1 Theoretical concepts

- **The pushing source**: In the case of the pushing source, an event becomes a message. Whenever an e-mail message is posted, the message is sent to invoke (re)action on the destination. Pushing an event is a synchronous action, since it is performed at once.

- **The pulling destination**: In the case of the pulling destination, an event becomes a value of interest. In the case of the pulling destination attributes and characteristics of the source are questioned. Visualization panels of a financial institutes often pull the results of the stock exchange on a regular base (once every 30 seconds). Pulling events is just as synchronous as pushing events.

- **The subscription model**: In the case of the subscription model, an event becomes a notification. This asynchronous model consists of a source placing one or more listeners on a destination, asking the destination to notify whenever certain subscriptions are met. Most European governments place listeners at financial institutes to be notified whenever certain doubtful transactions occur. This model is asynchronous since the listener cannot be aware of the moment a notification occurs.

6.1.3 The network is the computer

Software components are all designed with a specific goal of providing certain functionality to users, other components, etc. Their existence also suggest that there is an interest in this functionality. Providing and invoking this functionality is therefore often translated into a client requesting the functionality and the server providing such.

The client-server distinction suggests that these distinct components can be physically localized\(^2\) and that a component is either a server or a client. The client-server model does not reflect the concept that the functionality provided by a server may well be implemented by all kind of client behavior on other servers.

What will be presented in this thesis is the concept of *The network is the computer*. This truly distributed and anarchic concept consists of an ever changing network consisting of computational agents\(^3\) invoking services to get some work done. There are no servers, no managers and no central repositories managing the network topology. Figure 6.1 illustrates both the monolithic client-server model as the anarchic agent based network structure.

In the rest of this document the functionality provided and requested by components is referred to as a service.

\(^2\)The biggest computer in our office is the server

\(^3\)Several computational agents may resde on one physical computer
6.2 A dynamic topology combined with flexibility in use

The previous chapter in conjunction with the section on Jini technology described in appendix C on page 70 have hopefully introduced the advantages of serving flexible, dynamic and global transactions by software that is based on just these concepts. The flexible topology resulting from these ideas and illustrated in figure 4.3 on page 27 are merely a beginning though.

The rest of this chapter will focus on the components themselves. What will be described here is that anarchy in topology can only be valuable as long as the network remains manageable. Hierarchical manageability and the possibility of using, reusing and upgrading the functionality of components by newer or more intelligent behavior became the founding notion behind a standard basic component.

The standard component which serves as the foundation for all components

---

Figure 6.1: Different network topologies
6.3 Requirements of the basic component

in the framework (mainly described by [Muller, 2001]) is referred to as the basic component.

6.3 Requirements of the basic component

The previous section discussed three cornerstones of what is presented in the rest of this thesis as the basic component, its modules, extensions and tests. The awareness of these cornerstones in conjunction with the concepts on information and organizations stated in the previous chapter, formed the basis for the requirements of the basic component. These are:

- **Generic in use:** The basic component should form the foundation of most components used in the framework. The collection of components is diverse; from visualization to wrappers and from actors to virtual carwashes, gaspumps, and other devices. The first requirement is that only bare functionality is provided.

- **Lean and mean:** Since the basic component must be independent of its underlying hardware, the basic component must be as lean and mean as possible. It should be functioning on large mainframes just as efficient as on PDAs.

- **Provide storage to subcomponents:** The third requirement is that the basic component should provide storage for subcomponents defining functional behavior. This requirement also implies that the component also provides mechanisms to requests and invoke the services provided by these subcomponents.

- **Lookup and be looked-up:** This requirement states that a mechanism must be provided to components by which they are capable of looking up other components and be looked up by others.

- **Distributed citizenship:** As stated both in the introduction of this chapter as in the information model stated in the previous chapter, this architectural framework is new in the sense that it does not hide or simplify the distributed and complex characteristics of information and organizations. The basic component therefore must support the concept of anarchic distribution as illustrated in figure 6.1 and provide those services that are obliged for all citizens in this network.

- **Recognizability and manageability:** Anarchic networks of ever changing topology, coming and going services, users and actors have a tendency of unrecognizability and unmanageability. Structures have to be defined by which lookup mechanisms are defined and by which functional behavior can be spread over the network.

- **Subscription:** The basic component is extended to provide functionality to other components in the anarchy of the network. Resulting from this perspective, the basic component will encompass the storage of object oriented subscriptions and notify the appropriate listeners.
6.4 Underlying technologies

- **Portability**: The concepts and their implementations should not be restricted to programming language or operating system. To support this requirement as much as possible and to tighten the relationship with Sun’s latest products and solutions, the entire framework is developed in the Java programming language.

### 6.4 Underlying technologies

Since this thesis most certainly is not the first on distributed computing and distributed software, standard technology is studied to define the basic component on. A theoretical background on the concepts of distributed software and technologies such as CORBA, RMI and socket connections are described in appendix B on page 62.

An important conclusion that must be drawn from these technologies is that they do not overcome the concepts of centralized managers and hard coded dedicated connections. Though especially the first two have greatly improved distributed computing in providing object semantics over a network, this centralized approach led to a currently rising interest in peer-to-peer computing.

The best examples of peer-to-peer solutions are software packages like Napstar\(^5\) and Gnutella\(^6\); components finding, serving and using each other on an equal basis. Java has currently two libraries for decentralized distributed computing:

- **JXTA**: this Java peer-to-peer library is meant to develop Napstar-alike software packages. From what I have seen so far, JXTA’s main strength is file-sharing, which resulted in stream based connectivity. JXTA is accompanied by a bash-alike terminal preventing users to have acquainted any knowledge of the Java programming language.

- **Jini**: The older Jini libraries are much more focused on extending CORBA and RMI with services overcoming any hard coded centralized use. Jini does not redefine its underlying CORBA or RMI approach, and therefore has resulted in rich object semantics over a decentralized network. The services Jini provides on top of CORBA and RMI are described in appendix C on page 70.

Because of the above, Jini was chosen as the technology standard to design the basic component with. The basic component will therefore occur in the code and in the class diagrams as the JiniComponent.

---

\(^5\)http://java.sun.com

\(^6\)http://www.napstar.com

\(^6\)http://www.gnutella.org

Peter H.M. Jacobs

Sun Microsystems & DeR University of Technology
6.5 Design of the basic component

Though Jini has opened many doors toward truly distributed computer environments, Jini is not about the small collection of administrating services it provides. It is the mindset that is considered its core, and developers encompassing the technology do so by encompassing this mindset. Since this section describes the design of our basic component, it must reflect the decentralized concepts and ideals and forms therefore undoubtedly the core of this thesis.

It is hopefully this mindset and the implementation of the basic component designed during the research done at the iForce center that will lead to further use and improvements.

In this section the requirements are conceptualized step by step and it is shown what the consequences of the design choices are for the requirements stated. Several design choices, iterations and implementations eventually led to the JiniComponent or basic component used as the foundation object of the architectural framework.

6.5.1 Functionality 1: Defining the component and providing storage to functional behavior

The first step was to define the component and to provide a storage and retrieval mechanism for subcomponents eventually contributing to the functional behavior of the device, actor or wrapper extending the component.

![Diagram](image)

(a) Direct invocation  (b) One query mechanism

Figure 6.2: Storage structures I

The first concept, figure 6.2(A), illustrates a client component invoking a communicating subcomponent which invokes a logging service. The logging service
6.5 Design of the basic component

invokes a time synchronization service. This chain is hidden for the client and all modules can be replaced individually. Colors define the interface of the component that is looked for. It must be clear that logging, time synchronizing and communicating services are not obliged for the basic component and therefore are not initially stored in the basic component. They are merely used as examples of subcomponents as described in chapter 7 on page 45.

The main problem of this design is that all the subcomponents must implement lookup mechanisms to find other subcomponents in the storage room of the basic component. This most certainly would contradict the requirement of modularity and would not make the component as lean and mean as required. Updating client lookup mechanisms would affect all subcomponents stored.

Figure 6.2(B) provides an initial solution to this problem. Subcomponents invoke a getComponents(...) method on the basic component to request other subcomponents on which they request one or more services.

![Diagram](image)

(a) Returning a list  (b) Explicit functionality

Figure 6.3: Storage structures II

Though the second subfigure is a step forward, 6.3(C) illustrates a further complication. The first question is whether it is plausible and acceptable that more than one logging subcomponent is stored in the basic component. Though it makes its usage more difficult the answer is yes. It must be possible to use redundant subcomponents and for example log both to file as to send an emergency email. The contrary is explained in the same figure since time synchronization subcomponents are not present.

Extending the JiniComponent therefore implies that searching subcomponents results in a list of all known subcomponents. It is important to oversee the consequences of not knowing the number of resulting subcomponent, since not using them
6.5 Design of the basic component

all or presuming one is available highly diminishes the stability of the component. Figure 6.3(D) illustrates that the lean and mean requirement, modularity and independability are met by placing all the intelligence of the basic component itself in subcomponents as well.

Figure 6.4 on page 40 presents the final illustration of the component storage design in the basic component. First of all this figure illustrates that there is no conceptual difference between an extended basic component and a subcomponent. Requesting time synchronization subcomponents may well include requesting a network client service, a lookup service and eventually returning one or more remote components.

Secondly this figure illustrates that the basic component is kept both lean and mean as generic. Without the network client module network searches cannot be called upon by the getComponents(...) method. The network client built in the first release of the implementation is restricted to Jini network searches. Other implementations may well implement Socket, JXTA, Corba, JDBC or LDAP searches.

6.5.2 Functionality 2: How to address other components

After defining how subcomponents are stored in the basic component, this subsection describes how to address them. In the previous figures this topic is illustrated by using different colors for different (sub)components. First a short overview on current query technologies is presented.

- Jini's federative approach: Jini networks consist of federations and groups all having at least one lookup service. Lookup services store attributes of all registered services and queries are submitted based on these queries. As appendix D on page 80 describes this model is badly implemented by Jini.

- Directory alike approach: There are several methods of using a directory-alike approach like LDAP, URI's, URL's or packages. All these methods are based on a hierarchical tree concept.

The main characteristic all these approaches share is that they store an array of object oriented attributes. These attributes are for example a name, location, building, floor, security keys, a reference to their administrator, etc. In the architectural framework, this led to the concept of a passport. Every basic component is instantiated together with its unique passport.

Providing bi-directional formats

Another requirement was the independability of the basic component with respect to its environment. Instead of looking for the most appropriate query technology, this led to design that all should be supported. One of the services defined in the passport had to be a change of format. Figure 6.5 illustrates the choice in
implementing the passport object. In both illustrations it is clear that the core consists of a set of unique attributes describing the component.

The main difference is whether to include all format services in the passport object or whenever needed to extend the passport object with the appropriate format implementation. Though the second option looks far more elegant and generic, the first implementation is chosen due to security constraints. If extending the passport object would be an option, one could easily overwrite its implementation.

The strength of the format services included as that they are bi-directional and therefore prevent synchronization problems. An example is that whenever the LDAP representation of a passport is used and one of its organizational attributes is altered, its Jini federation is altered immediately and the component will subscribe itself to its new appropriate lookup service.

**Querying the passport object**

The basic component itself is equipped with the revision of Jini’s Marshalled Object Oriented query mechanism. The manner in which this model is revised in conjunction with [Lang, 2001] is described in appendix D on page 80. The basic idea is though that queries are performed by creating a template object as illustrated in figure 6.6 on page 42.

The appendix describes how this revision provides region definitions on top of this template based matching and therefore provides the possibility to express the need for all passports in building 3 or higher.
6.5 Design of the basic component

![Diagram](image)

(a) Stored object  
(b) Query object

Figure 6.6: Object Oriented query

6.5.3 Functionality 3: Accepting remote listeners

Services and components are provided within the architecture because their functionality or behavior is of particular interest to its surrounding. This implies that there is a general need for updates in a components attribute values and therefore in its status. For a component this results in providing at least one of the three event approaches described in section 6.1.2 on page 32.

Though synchronous approaches may well be optimal under certain circumstances, they usually increase network traffic exponentially since they do not only send and receive events on change. These mechanism should therefore only be used in situations were regular updates are regular\(^7\). The basic component in its basic form does not support them though. It is the asynchronous subscription model that is implemented.\(^8\)

For the basic component this implied storage of subscriptions on its behavior and to notify listeners on requested changes. The design of this mechanism combines Jini’s remote event model with our own attribute based, object-oriented query mechanism described in appendix D.

In the perspective of the basic component described in this chapter nesting these subscriptions and therefore introducing *third party event handlers*\(^9\) is an outstanding way to achieve a completely distributed and scalable architecture. Because of the security involved one may notice that nested subscriptions are not introduced by Mark Hissink Muller[Mueller, 2001].

As may be noticed the basic component only provides a generic implementation of accepting subscriptions and notifying results. Client behavior which includes the

\(^7\)Visualization panels are good examples

\(^8\)Extended basic components may well include pulling event mechanisms in their extension.

\(^9\)see C.4 on page 76
6.6 Extended overview and concluding remarks

Figure 6.7 on page 44 gives an extended overview of the basic component and therefore of the result of this chapter. This section will reflect the requirements mentioned earlier and state further recommendations regarding development in of a basic foundation of truly distributed and decentralized computing.

The main results of the basic component with respect to the foundation object it was meant to be are:

- Modularity is achieved by a strict separation of interfaces and implementations. The boundaries are rigid and all modules are as loosely coupled as possible.

- Flexibility is achieved by an extension model and network lookup model transparent to components and subcomponents. The administrator component of 6.7 could just as easily extended the basic component itself and therefore administer several components remotely. It is up to the developer to click the building blocks to a framework.

- Manageability is achieved by removing, adding and updating subcomponents in runtime to a component. Because subcomponents are found by a request to the getComponent(...) method of the basic component, new subcomponents are immediately returned and therefore used.

- Efficiency is achieved by modularity in the sense that there is only one log mechanism, one authorization mechanism, etc. For reasons of efficiency of network traffic, a boolean attribute in the getComponents invokes a network search.

- Recognizability is achieved by a full body passport object and its bi-directional format services. Figure 6.5 on page 41 has showed that security constraints have prevented a passport interface and made the passport a non-extendible, final, object. A recommendation to further development would be how to overcome these constraints without these implications.

- The basic component in its pure sense is as lean and mean as possible. The object does not rely on other objects than illustrated in figure 6.7 and is therefore usable on both PDAs as on mainframes. The implementations of the subcomponents might change though.
Chapter 7

Designing functional behavior

This chapter describes in more detail how to deal with the functional behavior of components. Before doing so, the following definition is stated as a crucial mindset of dealing with subcomponents.

As an object is defined by its methods and operations, a set of operations and attributes defines an object.¹

What this definition states is the bi-directional relation between attributes, operations and their corresponding objects. Though many developers will disagree on this concept, the topology introduced in this thesis and the way functional behavior is stored in its upmost flexible way is based on this definition.

The consequences of designing subcomponents in the same mindset as this thesis and this bi-directional relation are:

- There is no conceptual difference between the services provided by a component and those provided by a subcomponent. Components may be extending the basic component or may be stored in other components. Redundancy may be implemented by instantiating both. A truly anarchic topology will not be aware of the difference.

- Since services are requested based on their interface, a clear understanding of the functionality of interfaces is crucial. Designing a JDBC wrapper interface is for example useless since there is probably no conceptual difference with a JavaSpace interface. Extending objects, defining interfaces and creating packages is not a subjective process based on beauty or recognizability to others, but must be based on the clear concepts underlying.

¹This concept and its strictness is introduced by [Bekke, 1993]
7.1 Modular design choices

As described in the previous chapter there is no conceptual difference in the way services are presented by either a component or a subcomponent. Modularity implies composability and decomposability in multiple hierarchical levels. The previous chapter however introduced the full body JiniComponent and subcomponents stored in this JiniComponent. The main question therefore becomes when to extend the JiniComponent.

Before introducing the design patterns involved, figure 7.1 illustrates how services can be implemented and therefore be provided to the network.

![Diagram](image)

**Figure 7.1: Requesting a service**

The figure illustrates a communicator service requesting a logger service. An array of three loggers is returned, all implementing the appropriate interfaces. When to implement the functionality as JiniComponent, stand alone component or as subcomponent depends on the following design patterns.

- **Modularity in runtime**: The main reason for using a subcomponent infrastructure instead of implementing these services as methods of an extension of the basic component is the modularity in runtime. It is this aspect that provides true flexibility not only in topology but also in use.
7.2 The component library

It is this modularity in runtime that in conjunction with the concepts on dynamic class downloading that has provided a much more flexible approach towards object oriented programming.

- **Unique versus generic**: Functionality that is only needed by a small part of the framework and in its uniqueness is highly dependable of hard- and software is more likely to be added as subcomponent to the framework. The JDBC-1.1.8 wrapper for Oracle version 7.0.2 on Solaris 2.5 is more likely to be added as subcomponent of a more general data extraction component. The goal is clearly to hide implementation details to actors, devices and wrappers of the framework.

- **Scalability**: Though an administrating component is as explained in the previous chapter a rather generic component, it will often be added as subcomponent to the JiniComponent. One central administrator would most certainly diminish the scalability of the framework. In choosing whether to extend the JiniComponent one should consider the possibility of bottlenecks and the unavailability due to network failures.

- **Local versus Remote**: As is shown in the object model of the JiniComponent the way modules and components find each other is by invoking the `getComponents(...)` method of the JiniComponent. Invoking this method makes the network transparent since the result will both return locally stored modules as `stubs` of remote modules.

It is essential to understand that clients requesting submodules must be able to handle zero to n instances of the required (sub)component.

- **Level of intelligence**: A more difficult choice is where to divide a module into sub-modules. It is a delicate choice whether a transaction manager for example keeps track of its own time synchronization or uses external modules. The advantages of an intelligent module is the easiness of programming\(^2\) and therefore the performance of a module.

A serious problem of too intelligent modules is the hard coded functionality that does not easily permit the behavioral transfer between components. As is shown in appendix B.7 on page 68 uploading a components business intelligence has in my personal perspective opened many doors to real distributed frameworks.

Considering these axes, the next section will give an overview of the sub components designed within the period at Sun’s iForce lab. Hopefully others will encompass the idea and enrich the library with functional behavior not yet met.

\(^2\)One does not to consider network problems, the fact that modules might not be available, etc. etc.
in the previous section and hopefully serve as a startpoint for further development. Based on the definition stated in the beginning of this chapter it must be clear that the components built in this library only implement a small set of interfaces\(^3\). The first two design choices are used in the illustration of figure 7.2 on page 48 as axes of the component library. All modules can be found in the \texttt{com.sun.narad} package.

\(^3\)log, administrate, …
7.3 Recommendations

As stated the library gains in value by better and newer implementations of its content. One recommendation would be to extend the library in a way the Java programming language is extended\(^4\). Some deeper insight in requirements and areas of interest would most certainly be required to prevent uncontrolled explosion of half implemented components and behavior.

A second research recommendation would be to study some particular fields of interest among which:

- **Security and Real time solutions** In my personal perspective both real time systems as secure transactions cannot be achieved by simply adding corresponding components to the framework. They both enforce both users as developers to redesign all parts of their framework according to the requirements of either the real time or secure solution. Especially the matter of security in conjunction with decentralized distributed networks\(^5\) is an area not yet covered.

- **Amplifying effects** It is important to understand that there not one answer to the question if a particular behavior should be implemented as component or as subcomponent. More research on distributed decentralized networks could quantify amplifying effects leading to bottlenecks and therefore give a more accurate answer based on the current status of the network. The current status of the topology of the network is until now not included in these considerations.

---

\(^4\)The Java development process

\(^5\)JXTA, JINI, etc.
Chapter 8

Proof of concept

In line with Sun’s approach to complex distributed problems and their solutions, this chapter encompasses a proof of concept of the component framework laid out so far. This chapter describes hands-on experience and insight in the usability of what is designed in the previous two chapters.

As stated in chapter 6 on page 31 the JiniComponent and its subcomponents were primarily designed as foundation for actors, devices and wrappers. Its usage in these situations is described by [Muller, 2001], who has developed demonstration applications using the component in the eServiceStation case: a business case concerning more than 20000 petrol stations in the United States.

Two more technical cases were used in this chapter to demonstrate the usability of the component framework. The second case, project Hercules is in the phase of being presented to the open source community. More information can be found at http://www.jini.org/hercules

8.1 CASE 1: Data Management Center

One of Sun’s products in their iForce ready center is the Data Management Center. Though its value to many customers, the Data Management Center (DMC) is hard to visualize and demonstrate. The requirement of this case was to visualize this technical solution focusing on user interaction.

Figure 8.1 on page 31 gives an overview of the Data Management Center. The areas of interest of this case are:

- Production cluster: The two computers connected to the three storage systems called production cluster are the main area of interest in any company using the DMC solution. The production cluster encompass the database with customer information, production details or business transactions. What must be assumed is the requirement of every organization to keep their production cluster online whenever possible.
Figure 8.1: Data Management Center
8.1 CASE 1: Data Management Center

- **Offline reporting computer**: Generating statistics from the data stored in the production environment is prevented since it would affect its performance too much. Data are first copied to an offline reporting server before managers can collect statistics.

- **Backup server**: The backup server illustrated in the middle of the figure backs up the data of the production cluster and makes the data available to both storage (tape) devices as offline reporting servers.

The Data Management Center is a Sun solution providing a complete backup of a production cluster without having to suspend production activity due to file locks of the underlying online database(s). This is done by a snapshot technology moving the third storage mirror of the production cluster logically to the storage cluster of the backup system.

The idea of the visualization of this case was to create a steady load on the production database, to visualize this load and to update its content in such a way that only an instant backup could ensure synchronization with the offline reporting server. The two steps of this case were to wrap the oracle instances and to connect GUIs to these wrapper. The GUIs were designed by [Lang, 2001].

8.1.1 Wrapping the Oracle instances

As described using interaction with both the production cluster as the reporting and backup servers was one of the requirements of this project. This was achieved by wrapping the Oracle instances and having them controlled remotely by independently attached user interfaces. Figure 8.2 shows the class diagram of the Oracle wrapper. As illustrated both single queries as workload consisting of several queries a second can be triggered.

![Figure 8.2: Jini component use case](image)

Peter H.M. Jacobsa  Sun Microsystems & DoRe University of Technology
8.1 CASE 1: Data Management Center

8.1.2 Linking to the GUI

The only step to be taken by the GUIs was to include a network client subcomponent and have it look for the two oracle wrappers based on their passport. Since every JiniComponent (oracle wrapper) is equipped with the functionality of accepting and handling listeners, implementing the appropriate notify methods on the client ensured optimal functionality.

8.1.3 Conclusions

Since this chapter is called Proof of concept, the first conclusions are regarding the successful use of our JiniComponent as foundation for both the OracleWrapper as parts of the GUI.

The DMC code, demo and concepts have more than proven the success of the framework and its concepts. First of all the Oracle wrapper took due to its underlying basic component less than 30 lines of code. After instantiating several wrappers, the following tests were conducted:

- Different loads were placed on the wrapper the underlying Java and JDBC libraries were stable enough to translate these correctly to the Oracle instance. On a Sun SPARC II upto 35 transactions per second were fired, which is more than enough for the demonstration goal of this case.

- Different subcomponents were dynamically uploaded to test the dynamic download mechanism and to inspect whether these subcomponents were used instantaneously. The basic component was tested with the more complicated transaction services to see whether the instantiation of several redundant transaction managers was translated in appropriate commit-rollback mechanisms. All results were just as foreseen.

- Subscriptions based on the object oriented query mechanism designed in conjunction with [Lang, 2001] were placed on the oracle wrapper by the GUI objects. Both the remote asynchronous subscription model as the query mechanism worked just as expected.

- Third party event handlers were used in conjunction with a JavaSpace to test the scalability of multiple GUIs in combination with the nested subscription model. Both resulted in the same behavior as seen with direct subscriptions.

Though the requirements of the DMC case were user interaction and visualization, conclusions regarding them are not stated here, since the visualization part was built by [Lang, 2001].

Since the results of the basic component were just as hoped for and a more detailed insight on the strengths of the component was finally proven, the basic component could now serve as basis for all extensions designed by [Muller, 2001] and a more severe set of tests called: Project Hercules.
8.2 CASE 2: Project Hercules

The second case described in this chapter is project Hercules. This case describes how a distributed load generator consisting of multiple engines each consisting of multiple threads was built to ensure a steady simulation of customer interaction with any computer system. The goal of this case was to get hands on experience with amplifying effects in decentralized networks and to prove the scalability of the component framework.

Most customers willing to spend their time at Sun's lab are there to achieve a certain performance goal. These goals are often expressed as *If we can serve 1250 simultaneous users with an average response time of 4 to 5 seconds...*

These goals can only be measured when the load placed on the system is known and controllable. As is described, project Hercules is one of the first load generators that can uphold a steady load defined as a number of requests per second.

8.2.1 Multi threading and fixed loads

The idea of a load generator is to simulate simultaneous customer interaction with a system. As stated it is essential to control and measure this load to be able to explain the resulting system behavior. Before continuing with the somewhat unique approach of project Hercules the following concepts are defined.

*Thread:* A thread is an execution context that can be independently scheduled.; it may be associated with a user program or it may run only some kernel functions[Bovet, 2001]. A thread is often referred to as a worker within the system.

*Multi threaded applications:* Applications well designed in terms of many relatively independent execution flows sharing a large portion of the application data structures[Bovet, 2001].

Most current load generators\(^1\) are multi threaded applications setting up a load by defining a scenario, creating a fixed amount of threads and have these threads perform as much work as possible for a restricted amount of time.

To show why this approach is not very strong, consider the standard example of a database that is being tested for its performance. The load generator is started by creating two threads each setting up a certain load. As the number of threads is increased the databases response time will fall back until the databases breaks and response times fly sky high.

For a software engineer, the break point is the most interesting point. Since small system re-configurations might easily lead to a shiftment of this point, this is where the system needs to be studied very carefully. The main question is whether we can keep the load stable at this point. Using the multi threaded approach described in the previous paragraph this is not the case. As the database breaks

\(^1\) Apache's AB, PureLoad, LoadRunner, etc.
8.2 CASE 2: Project Hercules

and response times increase exponential, a thread is locked and therefore must wait longer before it is free to set up another request.

This implies that as soon as the system's response times increase, the same amount of threads can activate less requests per second and will therefore decrease the load on the system, which will help the database to recover.

8.2.2 Flexible amount of threads and a steady load

The most important requirement of project Hercules was to achieve a fixed load in time. No matter whether the response times increase, the amount of requests per second had to be steady.

One way how to achieve this was to use endless JiniComponents and to forget multithreading altogether. A truly distributed globally connected network consisting of millions of JiniComponents can easily keep up the load as required. The reason why multithreading was not neglected was because of the overhead introduced by using too many JiniComponents. Threads are small efficient and controllable. In project Hercules every JiniComponent depending on underlying hardware and operating system owns between 100 and 500 threads.

First a thread pool is created for every JiniComponent or its extension : the Hercules engine. The primary task of all the threads in the pool is simply to sleep. The pool is managed by a selector thread that will schedule action according to the amount of requests per second. On action the selector thread looks for a sleeping, non-interrupted, thread. This thread then is awakened from its sleep and does the job. Figure 8.3 illustrates the run method of the Hercules Worker method.

8.2.3 Dynamic rescaling

The technique described so far will achieve the requirement as long as there are non-interrupted sleeping threads. As soon as all threads are interrupted and therefore busy performing there task, Hercules will fail as well.

To resolve this problem, new threads are to be constructed during the run whenever needed. Creating threads during the run is directly measurable in the performance of the application though. To overcome this problem and to set up a highly scalable and performing load generator, the entire generator is distributed over several JiniComponents.

They share a scenario, synchronize there time, etc. etc. Whenever the selector thread of an engine notices that a threshold of interrupted threads is reached, it publishes a request on a JavaSpace to the other engines to take over for the time needed to construct new threads. This mechanism is partly illustrated in figure 8.4

8.2.4 Scenarios and configuration files

A distributed load generator can handle a scenario consisting of several tasks. The tasks built until now are HTTP and DNS tasks. It is possible to add new tasks
public synchronized void run()
{
    long[] processTime = new long[2];
    while(true) // Always
    {
        try
        {
            this.wait(); //sleep
        }
        catch(InterruptedException interruptedException)
        {
            this.interrupt();
            //set the interrupt attribute to true
            try
            {
                task.run(); //perform your task
            }
            catch(NullPointerException ioException)
            {
                exceptions++;
            }
            this.interrupted();
            //set the interrupted attribute to false
        }
    }
}

Figure 8.3: Worker.run()

without recompilation of the engine. A scenario is configured in an XML configuration file. The XML file resulted in appendix E on page 84. Though binding XML-instances to Java object seemed as a core technology of the Java framework, this library is still in progress. The result was that such a library is developed independently and will be published to the open source community shortly.

8.2.5 Recommendations

This load generator was developed to show how the idea of a distributed, scenario sharing generator could be implemented in less then 200 lines of code by simply extending the basic Jini component and adding the appropriate functional communication, logging and synchronizing modules. There are several recommendations be-
fore using this code commercially. All code can be found in the `com.sun.hercules` package.

- Scalability is proven: Since Hercules is based on a decentralized network, scalability occurred as expected. In contrary with other load generators the results were accurate up to milliseconds. The synchronization and distribution activities of the scenario did not affect the accuracy of the system and configurations up to 10 engines with a scenario of more than 1000 HTTP requests per second on different hardware were linked to achieve a breakdown of Sun’s internal proxy servers. Since the amount of threads can be predicted with this load generator, accuracy was measured above 99%.

- Sharing a scenario between multiple engines and defining thresholds depending on the capacity of a system is a sensible tasks since amplifying effects occur. More research should point out to a wise algorithm for configuring
the engines and setting up scenarios. A mathematical approach to this subject can be found at [Iverson, 1999]

- **Create new tasks.** As explained a scenario is a sequence of different tasks. Tasks implemented so far are DNS and HTTP tasks. One of the strengths of commercially available load generators is the availability of all kinds of secure and insecure TCP/IP tasks. The next step would to implement these tasks.

- **Java is mentioned to be platform independent.** Especially in this case, Java is not operating system independent. Most bare network operations are directly translated to requests to the operating system. DNS caching, HTTP caching etc. must be controlled carefully. A good knowledge of the operating system is therefore essential.

- **Again the usage of the basic component is more than proven.** Synchronization and scheduling of a scenario is made possible due to the easiness by which asynchronous lookup and subscription is handled. Modularity and flexibility have resulted in the possibility of distributing scenario’s concurrently over PDAs and mainframes. This is unique!
Part III

Distributed components in a visualization environment

*Appendices*
Appendix A

The Proof of Concept trajectory

The Proof of Concept (PoC) trajectory aims to give customers a real-world demonstration of a Sun-based IT solution for their specific business system by building the the actual situation in the iForce Lab before it will be rolled out. Except for Sun and the customer, other actors may be involved as well, for example independent software vendors (ISV), system integrators or consultants. During the PoC engagement, Sun places its experts and facilities at the disposal of the customer free of charge.

In figure A.1 an overview is presented of a typical Proof of Concept trajectory.

Figure A.1: PoC-trajectory: overview

The PoC-trajectory consists of the following phases:

- **Customer reception**
  During the customer reception at the Vision & Values labs, Sun demonstrates demos of state-of-the-art technology solutions to the customer. In combination with overview presentations, the customer will get a feel for Sun’s vision of the networked business system. The customer will usually be represented by several executive managers. This phase normally takes up
to half a day. When the customer decides to extend the engagement to a full PoC-trajectory, preparations will be made for the planning session. One of the main preparations on the side of the customer will be the business architecture, serving as the basis for the planning session. It usually takes a couple of weeks before continuing the PoC-trajectory with a planning session.

- **Planning session**
  During the planning session, the initial business architecture will be conceptualized in more detail (if necessary) in order to develop a supporting technology architecture. Sun’s experts will work closely together with the customer, usually represented by a team of IT-staff. The deliverable of this phase is a (Sun based) technology architecture, supporting the customer's needs, that may be tested in the iForce lab. The session will typically take up to 2 days. It may take up to several months before the next phase, the lab engagement, is started.

- **Lab engagement**
  This phase will typically take up to 2 weeks, during which the critical sub-systems of the architecture will be implemented & tested in the iForce lab. These systems will be tested with regard to the customer’s main interests. Typical interests include feasibility, performance and flexibility.

After a successful PoC, the customer may decide to engage in a 1-2 month scalability and load testing trajectory, in which the complete system will be tested for durability and scalability.
Appendix B

From TCP/IP to RMI & CORBA

B.1 Introduction

This chapter covers the lower layers of the Java network model. Figure B.1 gives an overview of these layers. The lowest layer, just on top of the bare network protocols, is the Java networking layer. The layer above is called the Remote Method Invocation (RMI) layer and is unique to Java because it supports Java's object semantics over the network protocols. Parallel to the RMI layer is the CORBA layer. CORBA encompasses a suite of specifications from the Object Management Group\(^1\) and allows an application to reach non-Java components. There is quite some overlap between CORBA and RMI which is explained in the last section of this chapter.

![Diagram of network layers]

Figure B.1: Lowest network layers

\(^1\)http://www.omg.org
B.2 TCP/IP suite and Java networking

B.2.1 TCP/IP Suite

Although this paper does not cover bare networking essentials, a short description of the internet communication protocol is essential to understand many of the Jini protocols covered in appendix C starting on page 70.

TCP/IP is the is the standard for internet communication and consist out of the Internet Protocol (IP) and the Transport Control Protocol (TCP). Olaf Kirch [Kirch, 1998] describes IP as:

> The scheme of directing data to a remote host is called routing, and packets are often referred to as datagrams in this context. To facilitate things, datagram exchange is governed by a single protocol that is independent of the hardware used: IP, or Internet Protocol.

TCP he describes as

> A very important thing to know about IP is that, by intent, it is not reliable. .... IP solves this problem by simply discarding it. The packet is irrevocably lost. It is therefore the responsibility of the communicating hosts to check the integrity and completeness of the data and retransmit it in case of error. This process is performed by yet another protocol, Transmission Control Protocol (TCP), which builds a reliable service on top of IP.

UDP is a sibling protocol of TCP and is described as

> Of course, TCP isn’t the only user protocol in TCP/IP networking. Although suitable for applications like rlogin, the overhead involved is prohibitive for applications like NFS, which instead uses a sibling protocol of TCP called UDP, or User Datagram Protocol. Just like TCP, UDP allows an application to contact a service on a certain port of the remote machine, but it doesn’t establish a connection for this. Instead, you use it to send single packets to the destination service—hence its name..... While this is both faster and more efficient than TCP for simple transactions, UDP was not designed to deal with datagram loss.

Concluding, one can say that TCP/IP is the suite used for communication on the internet, where IP directs data between hosts and TCP controls the integrity. UDP can be used instead of TCP, but is not able to control packet loss.

B.2.2 Java networking

Because it provides the lowest level of application access to the network substrate (TCP/IP), the simplest and most direct way to program applications using Java is
through sockets. A socket is a conceptual communication endpoint of a network. Figure B.2 shows these imaginary endpoints.

![Diagram of client-server connection](image)

**Figure B.2: Sockets**

One can open a socket connection in Java and transfer nearly anything through the socket. Several examples of Java socket programming are included among which the com.sun.narad.proxy.TCPProxy.java.

For distributed applications one is more interested in invoking remote operations instead of just sending and receiving streams. These are several complications implementing these operations on socket connections.

- First of all programming both the client side as the server side of a method interface based on sockets connection is error prone.
- The coupling mechanism is very strong. If the implementation of the method on the server changes, the client communication mechanism needs to be changed as well.

Java created a solution for the above by developing a layer on top of the Java Networking called RMI. The following section explains the main concepts of RMI.

### B.3 Remote Method Invocation

#### B.3.1 Basic RMI

As explained RMI provides the programmer with the ability to call on remote methods as if they were locally present. This section explains briefly how to implement RMI.

First one needs to construct an interface that incorporates all the methods that can be invoked by remote clients. In figure B.3 the interface CustWork is shown with its method getServerName. Because the main and the CustWork are not copied to the interface, they will not be provided to the client for remote invocation. The CustWork interface extends the Remote interface: a generic interface without a body used for the creation of stubs. A stub is Java code that deals with all the crucial information a client needs to connect back to the server. The server object usually extends the UnicastRemote object to present itself as an RMI-serving object.
Figure B.3: RMI interface

The second step is to generate this stub. This can be done by external software tools like rmic that is supplied with the Java Standard Development Kit. As shown in figure B.4, combining the client with the interface and the stub is the next step in the process.

The last step is to add the stub and the interface to the client and to start the rmiregistry that is used as a naming server for the stub to locate the server application in the Java Virtual Machine of the hosts on which both the server as the rmiregistry reside. Because the stub is available to the client by the corresponding interface, the client will invoke methods on an object that seems local but is in fact remote.

Figure B.4: RMI communication
B.3 Remote Method Invocation

B.3.2 Dynamic class loading

Though there are many features about RMI that are not explained in detail, there is one feature that is crucial to understanding the next chapters. This feature is called *dynamic class loading*.

Dynamic class loading enables a client to send an extended class of the remotely invoked class as an argument of the remotely invoked method.

Though this may seem like an endless recursion this is not the case. To explain the mechanism let's consider the following example. A machine is defined as an object with a start and a stop method. Both methods have a parameter of the Machine type. This would result in the Java class described in figure B.5

```java
import java.rmi.server.UnicastRemoteObject;

public class Machine extends UnicastRemoteObject {
    public void start(Machine machine) throws RemoteException {
        machine.start(machine);
    }

    public void stop(Machine machine) throws RemoteException {
        machine.stop(machine);
    }
}
```

Figure B.5: Dynamic downloading machine class

Now two new machines are defined: a server (computer) and a television. They extend the basic machine and therefore will both have a start and stop method. In their extension they override these methods though. A television start method is different from a server start method. The server is described in figure B.6 A client now performs the following commands

```java
Server server = new Server();
MachineInterface machine = [Stub_To_Remote_Machine];
machine.start(server);
```

By remotely invoking the `machine.start(server)` method the client assumes that the remote machine object is aware of its input : the server. This might not be the case though. If the machine has never heard of a server object, it cannot
public class Server extends Machine
{
    public void start(Machine machine) throws RemoteException
    {
        startOperatingSystem();
    }

    public void stop(Machine machine) throws RemoteException
    {
        reboot -halt now;
    }
}

Figure B.6: Dynamic downloading server class

call upon its start method which will eventually start the operating system. What
the client tries to achieve here is therefore not only to invoke a remote method but
to add business logic to the remote virtual machine. Without the server class in its
classpath the machine will fail.

Dynamic downloading prevents this failure because it will have the server point
out to a class server to download the class object dynamically when necessary.
The class server is like a webservice listening to a certain TCP-port for requests on
classes. The server is therefore started with a pointer to this server by the following
command:

java -classpath $JAVA_PATH -Djava.rmi.server.codebase=http://host:port/ Server

Instead of either failing the request or adding the class objects to all remote
classpaths a client now uploads the server class to the class server only ones. A
sequence diagram of this mechanism is described in figure B.7 on page 68. As
can be concluded, dynamic loading of classes highly increases the flexibility of the
distributed application.

B.4 CORBA

If all networking were programmed in Java, the RMI technology combined with
the socket technology would be enough to set up distributed components in the
network architecture. However, there are many systems that do not run on Java.

CORBA is the Common Object Request Broker Architecture that allows a sys-
tems of objects implemented on heterogeneous systems to operate with each other.
CORBA is being developed by the Object Management Group to accomplish this

\footnote{The codebase of the class server is included in the Marshalled object of the server. More on marshalled objects is explained in appendix D on page 80}
heterogeneous network interface. The OMG has defined its independent interface
language (IDL) that makes the basis of each CORBA component. A developer
who wants to create a CORBA component on a specific platform like C++ or Java
first defines its interface in this IDL language. After defining this one needs to link
this interface to the platform specific ORB. This ORB consists platform specific
CORBA operations. Linking the IDL file to the ORB results in the stub and the
skeloton.

The communication scheme resembles RMI’s scheme and is shown in figure B.8
on page 69. CORBA clients lookup their service counterparts by either a lookup
service similar to rmiregistry or by a server side generated ior file that consists of
the server hostname an port. More on the actual implementation of a CORBA
bridge between Java and C++ can be found at [Lang, 2001]

B.5 Overlap between CORBA and RMI

Java 1.3 contains Sun’s implementation of an ORB. This indicates that a Java client
and server can choose between CORBA or RMI for their communication. Commu-
nication with non-Java components is being established by CORBA. CORBA uses
the IIOP protocol on top of TCP/IP where as the default for RMI is JRMP. Sun Microsystems and IBM have recently developed IIOP support for RMI\(^3\). This enables the direct communication between RMI and CORBA components over IIOP.

\(^3\)http://java.sun.com/products/rmi-iiop/index.html
Appendix C

Jini Technology

This chapter covers the next layer in the Java network model. As is shown in figure C.1 this is Jini. The beneath quotation explains the cornerstones of Jini.

![Diagram showing Jini, RMI, CORBA, Java Networking, and network protocol]

Figure C.1: All network layers

Jini provides loosely coupled lookup services, group partitioning, dynamic federations, remote bridging, distributed leasing and remote events.[Li, 2000]

As will be shown in this chapter Jini is neither just about devices, nor is it a substitution for CORBA or RMI. Though Jini provides services to the network, it is, by itself, not capable of communicating. To explain what services Jini provides, an ideal network needs to be defined.

An ideal network is a network that is never down, does not rely on changes in physical composition and contains continuously coming and going services that broadcast their services to the network as they join or leave.[Li, 2000]
C.1 Discovery and Join Protocols

In implementing this ideal network there are several problems and questions. The most important are:

- How is a client going to locate the computer(s) capable of performing the required service?
- How is the client going to interact with this ever changing service provider?
- How will the network deal with, or indeed know about, a new computer offering a service and wanting to join the community?
- How will the network cope when the computer that fails or disconnects while performing a service?
- How will the network cope when a client fails or disconnects while a service is being performed on its behalf?
- How do services handle feature enhancements and upgrades?

The following sections will explain how Jini offers services to the network that cope with these questions.

C.1 Discovery and Join Protocols

The first action of every component is to bootstrap itself. Both clients and services share common rules on bootstrapping: the discovery and join protocols. Subsection C.1.1 describes how services discover Jini groups by finding lookup services while subsection C.1.2 on page 73 describes the join protocols.

C.1.1 Discovery protocol

The discovery protocol allows a Jini network to locate a lookup service. Jini services are always registered within a group. As will be described in greater detail in section C.2 on page 74 a lookup service administers one or more groups. The next paragraphs describe the different approaches to discovering the network.

Multicast discovery

The multicast discovery mechanism is used to uncover any and all nearby services by a multicast broadcast over the network. There are two approaches to the multicast discovery protocol:

- The multicast request protocol is used by components when they first start up and want to find a lookup service. It is an IP multicast message over the network containing the group the component wants to join, a group of lookup services it already knows about, its IP number and the TCP port it is listening to for a response. A lookup service responsible for this particular group will answer the request by sending a unicast TCP message containing
its RMI stub and giving the component a unique ID and registering it in its database. Since the response includes an RMI stub further communication between the component and the lookup service is by the RMI protocol.

- The multicast announcement protocol is used as a backup protocol by the lookup service. Lookup services in Jini are obliged to announce their existence regularly. Should a client fail in its multicast request, it will finally find the lookup by this protocol. More importantly, though, the multicast announcement protocol is used to announce new lookup services. This keeps the existing services up-to-date.

The lookup service sends regularly\(^1\) a datagram to multicast group 224.0.1.84, port 4160\(^2\). This datagram package consists of the lookup service ID, its host and port information of its unicast request and a list of the groups it manages.

Though this multicasting seems very flexible, there is one problem. Where does the multicast message stop? Will a Jini component trying to discover a lookup service on a local LAN transmit multicast datagrams around the world? To prevent possible overload, multicast packages sent by a service are limited by what is called the IP multicast radius. The radius controls how far the multicast packages will travel outside of the local sub-network.\(^3\)

The geographical constraint on IP multicast messages by their radius would not allow worldwide Jini networks though. For these situations, where a lookup service is needed beyond the reach of the multicast discovery, unicast discovery provides the solution.

**Unicast discovery**

The unicast discovery protocol is used when connecting to a specific already known service. The unicast discovery is done by a dedicated TCP broadcast. As stated the main reason for using unicast discovery is to federate two remote federations and therefore connecting isolated IP multicast groups. This concept is referred to as *peer lookup*. As the next subsection shows unicast discovery is more hard-coded because the user has to know how to reach the remote lookup service.

**Basic discovery examples**

This subsection gives some basic examples of how to use lookup discovery. Figure C.2 on page 73 gives a short example of the unicast discovery way of finding a dedicated lookup service.

Since a multicast request is not dedicated, one cannot expect an immediate answer. It is unknown if and who is going to reply to the request. The asynchronous communication between the initiator of the request and lookup services

---

\(^1\)Jini specifications recommend every 120 seconds

\(^2\)Strangely enough 4160 is the hexadecimal code of CAFEBABE

\(^3\)Routers often use a *Time To Life* value as radius.
C.1 Discovery and Join Protocols

is handled by implementing a discovery listener. An example can be found at

```java
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;

public class UnicastDiscover
{
    public static void main(String[] args)
    {
        try
        {
            LookupLocator lookupLocator = \n            new LookupLocator("jini://host:port/");

            ServiceRegistrar lookupService = \n            lookupLocator.getRegistrar();

            if (lookupService !=null)
            {
                System.out.println(lookupService.getGroups());
            }
        }
        catch(Exception exception)
        {
            System.out.println(exception.toString());
        }
    }
}
```

Figure C.2: Unicast discovery example

Though all implementation of these discovery classes may well be developed
depends, Sun has provided several basic, mid-level and high-level implemen-
tations of both the unicast and the multicast discovery interfaces. The collection of
these classes are often called the basic helper classes.

C.1.2 Join protocol

The join protocol is not a communication protocol, but merely a set of design
requirements that should be followed by a service to properly join and leave a
C.2 Jini lookup service

Jini federation. The join protocol describes what additional behavior must be implemented on a service to maintain a consistent and stable state after or during a crash. The information that each service is expected to keep includes:

- A service ID
- Attributes that describe the service, used during lookup
- A set of groups that the service wants to join.
- A set of specific lookup services that the service should contact by Unicast discovery.

Though one is free to implement this behavior themselves, Sun provides a helper class called `net.jini.lookup.JoinManager`. The narad implementation of administering services can be found in the `com.sun.narad.administrator` package.

C.2 Jini lookup service

Jini’s lookup service is in many ways unique. First of all the lookup service is the only service that exists when bootstrapping a Jini network. The lookup service is absolutely essential for a stable, loosely coupled network. The responsibilities of a registrar in the Jini network are

- Registering new services. As the social security officer of a network, the lookup service will provide each service with an unique service ID. Whether a service crashes or moves (logically or physically) its ID never changes. This provides a client the ability to send a unicast message to the service. The lookup service will furthermore provide each service with its proxy. The lookup service will finally accept the service’s proxy and store it as a Marshalled object.

- Basic lookup service: As already stated a client can find a service by sending a profile to the lookup service and ask the lookup service to match this profile on its stored services. A profile consists of a template containing entries. These entries may be user defined, though a few basic entries are always defined. Though it seems (and is often presented) that basic lookup service does not differ too much from a basic database server in providing results to queries, there are fundamental differences and implications of the way Jini provides this service. These differences are discussed in appendix D on page 80

- Other responsibilities: A lookup service is responsible for bootstrapping a federation, managing a persistent database storage of the services and supporting inter federation communication.

---

4A Jini network contains federations containing groups. Each group contains one or more services
C.3 Jini distributed events

The distributed events functionality of a Jini network extends the Java event mechanism over a network. It enables clients to subscribe to state changes on remote objects.

A logical sequence diagram of Jini's remote event model is presented in figure C.3. Because both the client as the service call upon each other over a network, they both need the appropriate RMI stubs of their counterpart. Dynamic downloading provides the stub of the listening client to the service. The listening client implements a remote event interface and therefore has to define an appropriate notify method. Figure C.3 shows a sequence diagram of the mechanism.

There are several design scenarios when a third-party event handler is introduced. A third-party event handler is an object between client(s) and services(s) responsible for the event communication. A third-party handler is often used when:

1. A client or a service is not willing or capable of receiving or sending events. The third-party handler functions as proxy for such an object. A good example is a PDA like the Palm or a mobile phone.

2. A third-party handler provides value added services to the distributed event mechanism. For example it can broadcast one message to several interested components. This is shown in figure ??

![Diagram of Jini distributed events](image-url)
C.4 Jini distributed leasing

Although the lease concepts is one of the most fundamental concepts behind Jini, most Jini networks use Sun’s reference lease manager and won’t cover the idea behind. Before the lease mechanism is actually explained a quick review of the Jini promises is being made.

As described in the above sections it is now clear how Jini provides a loose coupling mechanism to services and clients. It is also clear how both clients and services bootstrap themselves and how further improvements in the implementation of a service will not affect the implementation of other components. The questions on coping with network troubles are not answered yet. These questions will partly be answered by the leasing concept.

Figure C.5 gives an overview of Jini’s leasing concept. In a Jini network every resource is either being leased or will be removed from the appropriate storage (lookup service, JavaSpace, etc.)

A registered service must contain a lease contract with a lookup service to maintain registered. A client who writes a variable into a JavaSpace must lease this connection to keep the variable in the space. This concept prevents lookup services to address crashed services and applications.

The Jini lease specifications are mostly defined as interfaces. This keeps the implementation very flexible and suitable for different networking protocols like RMI or CORBA. Jini’s reference implementation also introduces leasemaps. A leasemap is a bundle of several leases, used to minimize network traffic. A leasemap further provides methods to cancel and grant all bundled leased in one step.
C.5 Jini distributes transactions

A Jini infrastructure itself does not have a recovery mechanism to cope with services that fail while they are providing their services. This is because the interaction between client and service is application specific. Jini provides a mechanism for this interaction by its distributes transaction service.

Jini provides a transaction service by a two phase commit protocol using a transaction manager. Jini’s reference implementation of this transaction manager is called Mahalo. Jini’s transaction manager can be used as a reliable synchronization mechanism and supports the classical ACID semantics:

- **Atomicity**: if one of the operations within a transaction fails, then all operations already done need to be rolled back.
- **Consistency**: the state of the system is known both before as after the transaction. A system can therefore never be left in an unpredictable state.
C.6 JavaSpaces

- Isolation: isolations is the effect that operations in a transaction won't affect other operations in simultaneous transactions.
- Durability: concerns the state of the system following a completed transaction.

On a logical level, this is what happens during a two phase commit transaction:

1. The transaction participants Jini services join up with the transaction manager. This is comparable with registering with a lookup service.
2. Whenever a participant creates a transaction, the transaction manager Mahalo tells all involved participants to prepare for the commit.
3. All involved participants prepare the transaction by performing in such a manner that the task can either be committed or rolled back.
4. Any of the participants that cannot prepare for the commit will signal for an abort.
5. Should one or more of the participants abort, the entire distributed transaction is aborted and all participants will be asked to roll back their state.
6. If all participants report successful preparation, the transaction manager asks them to commit.

Because participants need to take a lease on joining the transaction manager, the transaction manager will be immediately signaled whenever a participant crashes during the two phase commit transaction. Subsequently the manager will instruct all participants to roll back.

C.6 JavaSpaces

A JavaSpace is a Jini service that allows distributes services to share data and behavior by storing Java objects. Linda Systems\(^3\), founded by Dr. David Gelernter of Yale University, discovered that complex distributed computing problems could be better and more simply expressed through what amounts to a set of well defined read/write operations on a shared store.[Li, 2000]

These concepts lead to the JavaSpace technology. A JavaSpace is a Jini service providing the following object storage mechanisms:

- **Lease write(Entry entry, Transaction txm, long lease)** This method stores an entry in the javaspace, using a **two phase commit transaction** for a predefined lease time.
- **Entry read(Entry template, Transaction txm, long timeout)** This method tries to read an entry matching a template. The entry returned is not removed from the JavaSpace. The read method is blocking and will wait until a corresponding entry is returned or the timeout time is reached.

\(^3\)http://www.cs.yale.edu/Linda/linda.html
C.7 What’s left

- **Entry** `take(Entry template, Transaction txn, long timeout)` This method differs from the read method in removing a returned entry from the JavaSpace.
- **Entry** `readIfExists(Entry template, Transaction txn, long timeout)`. This method is non-blocking variant of the read method and will return immediately when no corresponding entry is present in the JavaSpace.
- **Entry** `takeIfExists(Entry template, Transaction txn, long timeout)`. This method is non-blocking variant of the take method.
- **EventRegistration** `notify(Entry template, Transaction txn, RemoteEventListener listener, long lease, MarshalledObject handback)` This method is associated with the remote event mechanism and places a remote listener on the JavaSpace listening for entries matching the template. The listener will be notified by its own notify method. The notification by the JavaSpace to the listener will not contain the actual entry. The listener must call the read or take method to accomplish this. Because the remote event notification prevents polling the space once every several seconds, this mechanism is used in project narad.
- **Entry** `snapshot(Entry entry)` This method can be called upon to prevent serializing the entry in the space. This is only wise and useful on often used entries since it forces the space to keep the entry as Java object in its memory. More on marshaling is explained in appendix D.

C.7 What’s left

This appendix has only described some of the key concepts of the Jini technology. It is a very flexible protocol used for distributed Java applications. Its use lies above all in the strength of dealing with changes in the network. There are many aspects of Jini not discussed in this appendix though. Other chapters, appendices and code samples will either point them out or make references to literature.
Appendix D

OO queries based on Marshalled Objects

This appendix describes how Object Oriented queries can be constructed to search for services, entries in a JavaSpace,. . . . Besides describing how this technique may be used, this article explains how this mechanism affects the underlying object model. Therefore, I personally consider it far from mature.1

Together with Niels Lang[Lang, 2001] the advantages of a good object oriented query model became obvious, which led to an extended approach described in section D.4 on page 82. The following subsection will start by describing some of the concepts underlying this mechanism.

D.1 Marshalled objects

Based on an object’s type, Java model has defined three different ways of transmitting objects over a network.

1. All simple or primitive data types2 are sent by value.

2. Any object variable that supports a remote interface should be passed by reference. This means that a stub is sent instead of the object itself.

3. All non RMI or primitive objects are marshaled before they are passed.

Since most objects are neither primitive nor CORBA/RMI objects, they are marshaled before passed over the network. As will be explained marshaling occurs nearly transparent to the developer and the implementation is somewhat hidden

1Especially JavaSpaces ruin the object model. Strangely enough, no remarks in this direction are described in the Jini literature published so far.

2These include int, long, char, . . . . They do not include java.lang.Integer, java.lang.String, . . . .
in the Java programming language. For this reason marshaling is one of the least understood concepts of Java. First a definition of a marshaled object is presented.

A marshaled object is an object containing a byte stream with the serialized representation of an object given to its constructor and a codebase URL from where the class can be loaded\(^3\). Any remote object in the MarshalledObject is represented by a serialized instance of its stub.

Serializing an object means to transform the object into a byte stream which can be send over the network. The receiving side must then deserializze this stream into the appropriate instance. Serializability of a class is enabled by implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable and the serializable interface itself has no methods or fields and serves only to identify the semantics of being serializable. All serializable objects are marshaled and unmarshaled transparently when transmitted over the network.

As figure D.1 shows, a class implementing the java.io.Serializable interface must take full responsibility for the public, protected and package fields of subtypes not implementing the the Serializable interface. As the object itself has to take responsibility it must implement both the standard serializing writeObject(...) as the readObject(...) methods to accomplish this. Deserializing such an object implies constructing an empty instance of the subclass and have the readObject(...) call upon its set methods. Whenever the subclass does not have a constructor without arguments, deserializing will result in a ClassCastException exception.

D.2 Constructing queries

Since two instances from the same class with the same values will result in the same Marshalled objects, one can construct marshaled template objects and have them bitwise compared to stored objects to see whether or not they represent the same data. The main advantage is that no knowledge of the objects is propreately.

Figure D.2 shows both a stored object as one of many template objects that would result in a succesful query.

D.3 Disadvantages

There are several disadvantages of using this technique.

- **Regions** are not supported. Since there is no knowledge about the contents of the marshaled object and queries are resolved just by comparing the byte streams, it is not possible to define regions.\(^4\)

\(^3\)see: dynamic class loading in section B.3.2 on page 66
\(^4\)A region might be the collection of all objects with an attribute >20 <60 ! = 45
D.4 An extended approach

Figure D.1: Serializability

- Public attributes: Since the above template and entry have different mar-
  shalled object, a bitwise comparison is not done on the objects themselves
  but on all separate underlying attributes. Before an entry is stored or a
  template is submitted all of its attributes must be serialized separately. To
  accomplish this, a storage service, like a JavaSpace, must have access to these
  attributes, which is only possible when these attributes are public. As men-
  tioned in the introduction, this affects the object model of the object in a
  rude way. It is in my personal opinion rather strange that the developers of
  this framework did not copy the solution provided by the Java beans frame-
  work which at least would have protected the attributes from external write
  permission.

D.4 An extended approach

The basic requirement of designing an extended object oriented query model was to
be able to place listeners based on complex object oriented queries, where the ob-
jects in question were not affected by their usage as template. A second requirement
was to facilitate regions and therefore make the model more intelligent.

The first step in achieving this model was by constructing a template object.
This is done in the following way.

1. Construct value object: First an object is constructed and filled with all
   necessary attribute values used as query values. This is identical to the
   previous explained way.
2. Encompass this value object with an hierarchical tree containing operator objects for the attributes set in the previously filled object. These operator objects define the regions of the query. The template object containing these two objects are illustrated in figure D.3

The query itself consists of recursively inspecting attributes. Depending on the operator a result is given. This mechanism is illustrated in figure ??

As may be concluded both requirements of the extension are met. The object oriented query model has therefore become a powerful and elegant way of subscription templates.
Appendix E

Java XML Binding

E.1 Introduction

The combination of Java and XML technology is often considered as the core of tomorrow’s World Wide Web. One of the main reasons for this consideration is their portability. It is all about portable code speaking a portable language. The transition from one into the other is still not provided efficiently though.

This article describes an approach to this transition by presenting an unmarshall class capable of directly constructing Java instances based on their XML representation. The official Java Architecture for XML Binding [Reinholt, 2001] is under early access development and will eventually replace this code. Until then, this article may well provide a ready to use generic solution and give some deeper insight in the difficulties of binding XML instance into Java.

E.2 XMLSchemas

In most programming languages there is a very strict separation between objects describing an instance (a class object) and the instance itself. Binding XML instances to Java instances therefore implies the distinction of an XML instance from it’s XML class. The XML class object should represent the main characteristics of the Java class whereas the XML instance should represent instance values.

Until last May there was only one official way to constrain and describe XML instances. This was done by a Document Type Definition. A DTD is in the process of representing a Java class not very useful. One of more reasons is that there is no sufficient way to constrain an element’s type. An element’s type is either empty, text or a subelement.

Since 1999 several institutes and companies have been working on a more sophisticated way to constrain an XML instance. This effort resulted in the official w3 recommendation of the XMLSchema as the successor of the DTD[Forgue, 2001]1.

1A good overview of the XMLSchema can be found at w3.org[Fallside, 2001]
Before the binding process can be explained several key facts of this technique must be clear.

### E.2.1 Namespaces

On first sight namespaces resemble Java's package structure. A namespace is a mapping between a prefix and a URI used for defining data structures that allow parsers to handle element collisions. Figure E.1 will illustrate its use.

```xml
<reservation>
  <restaurant>
    <name>Pasta & Basta</name>
    <address>231 First street, SF</address>
  </restaurant>
  <date>June 1th, 2001</date>
  <time>8pm</time>
  <party>
    <name>
      <first>John</first>
      <family>Smith</family>
    </name>
    <size>12</size>
  </party>
</reservation>
```

**Figure E.1: XML instance without namespaces**

This XML instance defines a reservation for dinner at the "Pasta & Basta" restaurant for a party of 12 people registered under the name of John Smith for June 1th at 8pm. If assumed that these are the data needed to make this a valid reservation, one can now look at ways to constrain this instance by defining its XMLSchema. Constraining the message means defining a unique meaning for the elements used in the instance. The name element in this example will therefore result in a naming collision.

The name element belonging to the definition of the restaurant consists of one string defining it while the name element defining the person who made the reservation consists of both a first name and a last name. How would a parser know which name to use? Using namespaces provides the solution as is illustrated in figure E.2.

In the above example three namespaces are defined: person, company and the default one. Restaurant is defined in the company namespace while date, time and party are defined in the default namespace. Party has one subelement name defined in the person namespace. The different namespaces have now provided a clear understanding of all elements including the name element.
E.2 XML Schemes

```xml
<reservation>
  xmlns="http://www.myexample.com"
  xmlns:company="http://www.myexample.com/company"
  xmlns:person="http://www.myexample.com/person">
  <company:restaurant>
    <company:name>Pasta & Basta</company:name>
    <company:address>231 First street, SF</company:address>
  </company:restaurant>
  <date>June 1th, 2001</date>
  <time>6pm</time>
  <party>
    <person:name>
      <first>John</first>
      <family>Smith</family>
    </person:name>
    <size>12</size>
  </party>
</reservation>
```

Figure E.2: XML instance with namespaces

E.2.2 Primitive, simple and complex types

As stated XML Schemes provide developers ways to constrain element types on a much more sophisticated way than DTD’s did so far. There are three different kind of types defined in an XML Schema.

- **Primitive types**: This is the collection of well known generic types like integer, long, double, date, etc. These types can be mapped from their XML Schema definition to Java without any problems.

- **Simple types**: These types are not primitive but do not allow attributes or subelements. These types are mostly used to extend and restrict primitive types as figure E.3 shows.

```xml
<simpleType name="myInteger">
  <restriction base="integer">
    <minInclusive="1000">
      <maxInclusive="2000">
    </minInclusive>
  </restriction>
</simpleType>
```

Figure E.3: Restrictive primitive type
• Complex types: This is the collection of all non-primitive, non-simple types.

E.2.3 Attribute or value

In XML there is a distinction between elements and attributes. Figure E.4 demonstrates the difference by defining a currency attribute.

```xml
<Order currency="EUR">
  <camera>
    <brand>nikon</brand>
    <type>F-100</type>
    <price>199.95</price>
  </camera>
</Order>
```

Figure E.4: XML instance with attribute

When to use an attribute or an element is rather vague in the XML definition. Most developers use attributes to express "meta data" like the version of the XML instance. These data do not describe the root element.\(^2\) Though nested attributes are not allowed in XML, XML Schemas can both constrain attributes and elements.

E.3 Binding XML to Java

This section explains how to bind XML instances to Java objects based on XML Schema validation and Java’s Bean model. It will start with an overview of process after which the most important parts are described separately.

E.3.1 Binding flow

The first step in binding the XML instance is to parse the actual XML message. In this approach JDOM\(^3\) is used. One could use a SAX or DOM approach as well. More information on different ways to parse an XML document are described in *Java and XML* [McLaughlin, 2000]

The next step is to search for the schema location in the XML instance and parse it as well. The application now has two element trees one describing the instance and one describing the schema. To be able to proceed the root element is mapped to its Java class representation and a new instance is constructed. This instance will eventually be returned as the result of the unmarshall method.

The next steps are done for all the children of the root element. These include mapping a child to its Java representation class, decide whether or not this is an interface and whenever it is not an interface nor a primitive type recurs downwards.

---

\(^2\) As stated before, all elements having attributes have a complex type definition.

\(^3\) [http://www.jdom.org](http://www.jdom.org)
These steps result in the Java representation of the child which then can be used as an argument to set the appropriate property of the root element.

An activity diagram for this process is shown in figure E.5 on page 88. The next three subsections will describe the three main procedures in detail. First the mapping from an complex XML type to its Java representation is discussed. Section 3.3 will then explain how to deal with complex types representing a Java interface class and section 3.4 will explain how to set the properties after constructing the values.

![Activity diagram of UnMarshalling.java](image)

**Figure E.5: Activity diagram of UnMarshalling.java**

### E.3.2 Mapping element types to classes

The first question to answer is how to map XML types to Java classes. In this approach the name of all non-primitive types in the XML Schema represent the
E.3 Binding XML to Java

Java classes. The XMLSchema in figure E.6 on page 89 will demonstrate how to construct an XMLSchema for the Nikon example of figure E.4.

```xml
<complexType name='com.trade.Order'>
  <sequence>
    <element name='camera' type='com.trade.products.Camera'/>
  </sequence>
  <attribute name='currency' type='string'/>
</complexType>

<complexType name='com.trade.products.Camera'>
  <sequence>
    <element name='brand' type='string'/>
    <element name='type' type='string'/>
    <element name='price' type='decimal'/>
  </sequence>
</complexType>
```

Figure E.6: XML Schema with Java references

So far most other developers have tried to map an XML namespace to a Java package structure or disregarded the Java package structure all together. The namespace approach is though it attractive elegance not wise. Using namespaces to resemble Java package structures would enforce a consistency between the Java application and the XML instance that is often not acceptable. What would happen to XML instances shared by several different actors or applications of which only some are written in Java. Using the namespace approach would enforce one identical package e.g. namespace structure for all parties.

The approach described here has left the structure of the data completely untouched. Every actor can simply build his version of the schema by replacing the name of the type. The XML instance is in other words kept portable (That is what XML is all about after all). It prevents endless debates between partners on how to setup a portable XML Schema.

Figure E.7 shows how to construct a Java object based on this approach.

This approach does not work for interfaces, because they cannot be instantiated themselves. The next subsection describes how this process is defined.

E.3.3 Interfaces

To illustrate binding interfaces, let's consider the example shown in figure E.8.

Every object implementing the Transporter interface should provide the drive method to other objects. Now let's assume there are three objects implementing
E.3 Binding XML to Java

```java
//find the schema of the element called xmlElement
String elementType = findElementType(xmlElement.getName(), schema);

//construct a new instance
Object javaObject = Class.forName(elementType).newInstance();
```

Figure E.7: Construct new Java instance based on XMLSchema

```java
public interface Transporter {
    public void drive(Direction direction, Double speed);
}
```

Figure E.8: Java interface

This interface: a car, a bike and a boat. Though their implementations of the drive method are probably totally different, their appearance to other objects is equal. Figure E.9 how this could be represented in an XML Schema.

```xml
<complexType name='transporter'>
    <sequence>
        <choice>
            <element name='boat'/>
        </choice>
        <choice>
            <element name='car'/>
        </choice>
        <choice>
            <element name='bike'/>
        </choice>
    </sequence>
</complexType>
```

Figure E.9: XML schema based on Java interface

An XML instance compliant with the Schema could be the following:

```xml
<transporter>
    <boat>....</boat>
</transporter>
```

Figure E.10: XML instance based on Java interface

What is seen here is that the boat is defined as an attribute of the transporter. This is not what is meant. The boat must implement the transporter and is the transporter. Though the boat is a nested element in XML it is most certainly not in Java. To prevent this the unmarshall class replaces the transporter element by its first and only child element. This approach is illustrated by the Java code of figure E.11
E.3 Binding XML to Java

```java
elementType = findElement(bodyElement.getChildren().get(0) \n    .getName(), schema);
// now we have the class implementing the interface.
```

Figure E.11: Java code parsing interface

E.3.4 Setting the bean properties

The previous subsections described how new Java instances are constructed based on the type definition in the XML Schema. This is just half of the job. A collection of new instances is just not what is expected as the final result. Looking again at the camera example it is clear that after creating a new brand instance, the brand setting method must be invoked on its parent (the camera).

The approach chosen here is to make use of Java's Bean framework. Though this framework is designed for the construction of ready made modules in integrated development environments like Forte for Java, its characteristics are very useful for this case as well. Though a complete discussion of the bean framework is far beyond the scope of this article, the basic idea is that a bean is a class with an empty constructor and implicit public variables. Implicit public variables are private attributes accompanied by public get and/or set methods.

Together with Java's reflection package the Java code capable of setting the correct property is shown in figure E.12.

```java
BeanInfo beanInfo = Introspector.getBeanInfo(javaObject.\n    getClass());
PropertyDescriptor[] propertyDescriptors = beanInfo.\n    getPropertyDescriptors();
while((found==false) && (i<propertyDescriptors.length-1))
{
    if(propertyDescriptor[i].getName().equals('brand'))
    {
        Object args='nikon';
        propertyDescriptor[i].getWriteMethod().\n            invoke(javaObject, args);
    }
    ..
}
```

Figure E.12: Setting the bean property

4In this example the brand of the camera is set to nikon.
E.4 Conclusions

The first conclusion that can be made is that due to XML Schemas the road is finally paved for a more intense relationship between Java objects and XML instances. A second conclusion is that this version of the Unmarshall class is though it is fairly complete, still has some limitations.

- Due to the nature of the recursion and the Bean model only objects can be bind with an empty constructor. All parameters must be set with appropriate set methods. Binding Java objects that include parameters in the constructor is difficult and potentially dangerous. How would a binder know which attributes defined as children of the element are parameters of the constructor?

- In the current release of Java, creating a beaninfo object for an interface does not enforce that a class implementing this interface will provide all methods described in the beaninfo object. This can potentially result in reflection exceptions because the methods invoked in binding the class can be described in the beaninfo object of the interface but not in the beaninfo object of the class. This problem may well be resolved in future versions of Java.

- Inner classes are impossible to bind because it is not possible to construct beaninfo objects of innerclasses. The differences between static and non-static innerclasses complicates the binding of these specific Java objects even further.
Appendix F

Jini HLA overview

Niels Lang [Lang, 2001] has described the High Level Architecture (HLA) used for communicating between and synchronizing of simulations. Peter Jacobs described Jini as a framework for distributed application. This paper describes the joint vision on the overlap and the differences between Jini and HLA. To understand both the overlap as the differences it is essential to first understand their goals.

F.1 Goals

F.1.1 HLA Goals

High Level Architecture (HLA) is used for communication between and synchronizing of simulations. HLA was developed to provide a framework for distributed simulations interacting without affecting their reusability and therefore not affecting their individual code.

F.1.2 Jini Goals

The goal of Jini is to provide mechanisms for an ideal network. An ideal network is a network that is never down, does not rely on changes in physical composition and contains continuously comming and going services that broadcast their services to the network as they join or leave.

The basic idea is that HLA is developed for simulations and Jini is developed for distributing applications. However both standards define a framework for distributed components and therefore share many concepts and definitions. These shared concepts and definitions are described in the next sections.

F.2 Dictionary

This section describes the dictionary that is used in both frameworks.
• Federation: A federation, in both Jini as in HLA, is a group of components working together to get something done. There is therefore a perfect match between HLA and Jini on the meaning of the word federation.

• Federate or service: A component in a HLA federation is called a federate where a component in a Jini federation is called a service. The main difference between federates and services is that communicating in HLA is done indirectly. Jini’s services work together to get work done, where HLA’s federates do all the work themselves and only communicate through the RTI.

• Federation execution: The time between bootstrapping a HLA federation and destroying it. Jini is not familiar with this concept because the goal of a Jini federation is to never breakdown.

• Runtime Infrastructure: the RTI is the component offering all the HLA services. The RTI component bootstraps the federation and is by itself not a federate. Jini is not familiar with the concept of one centralized manager. The services offered by the RTI are served in a Jini federation by many independent services. These services are, even though they maintain system sustainability, technically just regular Jini services.

![Diagram of RTI and federates]

Figure F.1: Runtime infrastructure

Figure F.1 shows the relations between the RTI, the federation and the federates while figure F.2 shows the same relations for a Jini network.
Jini services providing
network services

Jini services providing
user defined services

Figure F.2: Jini Services

- Federation Object Model: the HLA framework stores a complete object and interaction model of the combined federates in what is called the FOM. In the Jini framework there is no centralized object model. Jini clients must be familiar with the classes, operations and attributes of services they can potentially address. The basic difference is that HLA has one combined object model where Jini uses distributed object models. Jini’s object model supports object behavior, inheritance and aggregation, where as HLA’s object meta model only supports object’s inheritance.

F.3 Services

This section maps the services offered by Jini to the services offered by HLA.

- Jini’s discovery & join protocol is unique and not very comparable to any of the services offered by the RTI. Jini services can bootstrap themselves without knowing which lookup service to address. A federate must always know which RTI to address. Though this may look as an omission on the HLA framework, it is necessary to see these remarks in the perspective of
their potential usage. HLA is developed for the interaction of simulations in which worldwide persistence and discovery are less important issues.

- Jini's lookup service enables clients to find and establish direct communication between services on a template based query. Jini's services therefore publish attributes by which they can be searched upon in a federation. These attributes include version, model, etc. This mechanism is not offered by the RTI because the HLA framework forbids direct communication between federates. Communication is done indirectly through the RTI.

- HLA's object management is the service that enables federates to publish and subscribe to events and objects. To publish and subscribe events, Jini components use Jini's event service. JavaSpaces provide a mechanism to share distributed objects among services.

- Jini's JavaSpaces provide a shared data store to other services. JavaSpaces can be used for storage of any kind of object. Services can publish, subscribe and change ownership on Java objects. HLA provides object declaration and management services to share events and states of locally stored objects through subscription and publication mechanisms. Shared storage is not supported by the RTI.

- Jini's distributed leasing concept is one of the core services to support the everlasting sustainability of the network. HLA does not provide this mechanism for the same reason as was described at the discovery & join protocol.

- Jini offers a transaction service that fully complies with the ACID standards used in relational databases. HLA's RTI does not control the individual transactions but does notify federates when a retraction should occur. HLA's RTI provides mechanisms not only to notify retraction on failed transactions, but controls time consistency in the simulation as well. Time consistency between Jini services is not offered by the Jini framework, since this concept is only applicable to discrete event simulation, which is not Jini's native application area.

- HLA's RTI provides time synchronization between different simulations, e.g. optimistic and pessimistic synchronization.
Bibliography


decision support and control. http://www.tbm.tudelft.nl/...

glish. Oxford University Press.