CUSTOMIZABLE DEVICE REGISTRATION AND RESERVATION
IN UBIQUITOUS ENVIRONMENTS

BY

LEUGIM A. BUSTELO

B.S., University of Puerto Rico, Mayaguez, 1999
B.S., University of Puerto Rico, Mayaguez, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois
ABSTRACT

In a multimedia multicasting environment, an overlaid network of nodes connects the sender with a group of receivers forming a multicast tree. The media content travels through the edges of this tree until it reaches a Multimedia Gateway, which relays the content to the subscribed receivers. The Multimedia Gateway is responsible of managing the different receivers by acquiring information about each of them, performing admission control, and allocating the necessary resources. By managing the available resources, the Media Gateway can make Quality of Service guarantees to the receiver. Our work focuses on the interaction between the Multimedia Gateway and the receivers. The main problem is the direct interaction with a variety of end systems and mediums of connectivity. Our approach is to implement a flexible and customizable Multimedia Gateway that can be easily integrated into any network environment and upgraded to support changes to its surroundings. Our Gateway is composed of two systems, registration and reservation. The registration system is responsible for profiling each receiver by gathering a set of attributes and registering them in a table, while the reservation system is responsible of allocating the required resources to establish a particular Quality of Service for the receiver.
To my family and friends
ACKNOWLEDGEMENTS

I will like to extend my gratitude to Prof. Klara Nahrstedt for her advice and guidance, and for introducing me to the field of Multimedia. In addition, I thank Yi Cui, member of the MONET team, who so willingly helped me through the development of my work. I’m also thankful to Maria Conrad and Anda Ohlsson for their assistance.

I’m grateful to my family. They have always provided me with love and support. They are my number 1 fans. I especially thank my friends; they made my two years of Graduate School enjoyable. I will never forget the endless hours at the Sherman Hall Courtyard and all the different people I met there.

Finally, I thank the GEM Fellowship program and IBM Corporation for providing me with opportunity of assisting to Graduate School. Without their aid, this would have been impossible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Description</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Our Approach</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Thesis Outline</td>
<td>5</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Background</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Service Models and Protocols</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Mechanisms and Techniques</td>
<td>11</td>
</tr>
<tr>
<td>2.4 QoS Enabling Architectures</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Configurable Architectures</td>
<td>13</td>
</tr>
<tr>
<td>3 System Design</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Overall Design</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1 Assumptions</td>
<td>17</td>
</tr>
<tr>
<td>3.1.2 Design Goals</td>
<td>17</td>
</tr>
<tr>
<td>3.1.3 Design Approach</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Lookup Service</td>
<td>19</td>
</tr>
<tr>
<td>3.3 Registration System</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1 Registration Protocol</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2 Customizability and Flexibility</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Reservation Table</td>
<td>22</td>
</tr>
<tr>
<td>3.5 Reservation System</td>
<td>23</td>
</tr>
<tr>
<td>3.5.1 Reservation Protocol</td>
<td>23</td>
</tr>
<tr>
<td>3.5.2 Resource Table and Soft State Reservation</td>
<td>25</td>
</tr>
<tr>
<td>3.5.3 Customizability and Flexibility</td>
<td>26</td>
</tr>
<tr>
<td>4 System Implementation</td>
<td>28</td>
</tr>
<tr>
<td>4.1 System Model</td>
<td>28</td>
</tr>
<tr>
<td>4.2 Customizable Components</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1 Plug-able Framework Overview</td>
<td>30</td>
</tr>
</tbody>
</table>
4.2.2 Custom Strategy Framework Overview .................................................. 31
4.3 Programming Environment ................................................................. 32
4.4 Shared Components ............................................................................. 32
  4.4.1 Network Abstraction .................................................................... 32
  4.4.2 Configuration Framework .............................................................. 34
  4.4.3 Singleton Objects .......................................................................... 36
  4.4.4 Server Model ................................................................................. 36
4.5 Lookup System ..................................................................................... 37
4.6 Reservation Table .................................................................................. 39
4.7 Registration System Components ......................................................... 40
  4.7.1 Registration Protocol .................................................................... 40
  4.7.2 Plug-in Framework Implementation ................................................. 42
4.8 Reservation System Components .......................................................... 45
  4.8.1 Reservation Protocol .................................................................... 45
  4.8.2 Strategy Framework Implementation ................................................. 49
  4.8.3 Service Provision .......................................................................... 51

5 System Integration .................................................................................... 52
  5.1 Programming Interface ....................................................................... 52
    5.1.1 Java API ...................................................................................... 53
    5.1.2 System Packaging ....................................................................... 53
  5.2 Plug-in Implementation ....................................................................... 55
    5.2.1 Plug-in Class Naming .................................................................. 55
    5.2.2 Adding the Plug-in Code ............................................................... 56
  5.3 Custom Strategies Implementation ....................................................... 58
    5.3.1 Resource Table Management ......................................................... 59
    5.3.2 Admission Control ..................................................................... 62
    5.3.3 Service and Contract Delivery ..................................................... 63
  5.4 Component Installation ...................................................................... 65
    5.4.1 Plug-in Installation ..................................................................... 65
    5.4.2 Custom Strategy Installation ...................................................... 65

6 Experimental Results ............................................................................... 67
  6.1 Participating Devices .......................................................................... 67
  6.2 Experimental Plug-ins and Custom Strategies ..................................... 68
    6.2.1 Test Plug-ins .............................................................................. 68
    6.2.2 Test Custom Strategies ............................................................... 69
  6.3 Experiments ........................................................................................ 71
    6.3.1 Lookup Experiment ..................................................................... 71
    6.3.2 Synchronization and Registration Experiment ................................ 72
    6.3.3 Admission Control and Resource Management Experiment ............ 74
    6.3.4 Contract Creation Experiment .................................................... 76
    6.3.5 Timing Experiment ...................................................................... 76

7 Conclusions and Future Works ................................................................. 79
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>79</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>80</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Registration System</td>
<td>80</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Reservation System</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>APPENDIX A Configurable Properties</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>APPENDIX B System Installation</td>
<td>87</td>
</tr>
<tr>
<td>B.1</td>
<td>Common Requirements</td>
<td>87</td>
</tr>
<tr>
<td>B.2</td>
<td>Lookup Server Installation</td>
<td>88</td>
</tr>
<tr>
<td>B.3</td>
<td>Server Installation</td>
<td>88</td>
</tr>
<tr>
<td>B.4</td>
<td>Client Installation</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>APPENDIX C Plugin Class Documentation</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>APPENDIX D Strategy Classes Documentation</td>
<td>96</td>
</tr>
<tr>
<td>D.1</td>
<td>ReservationStrategy Class Documentation</td>
<td>96</td>
</tr>
<tr>
<td>D.2</td>
<td>ContractStrategy Class Documentation</td>
<td>99</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1: Multimedia multicasting tree</td>
<td>2</td>
</tr>
<tr>
<td>2.1: QoS friendly service model spectrum</td>
<td>8</td>
</tr>
<tr>
<td>3.1: Multimedia distribution environment</td>
<td>15</td>
</tr>
<tr>
<td>3.2: Area of interest of the multimedia distribution system</td>
<td>16</td>
</tr>
<tr>
<td>3.3: Overview of the gateway system</td>
<td>18</td>
</tr>
<tr>
<td>3.4: Lookup service protocol</td>
<td>20</td>
</tr>
<tr>
<td>3.5: Registration protocol</td>
<td>21</td>
</tr>
<tr>
<td>3.6: Reservation protocol</td>
<td>23</td>
</tr>
<tr>
<td>3.7: Resource Table</td>
<td>25</td>
</tr>
<tr>
<td>4.1: Registration and Reservation System details</td>
<td>29</td>
</tr>
<tr>
<td>4.2: Plug-able Framework</td>
<td>31</td>
</tr>
<tr>
<td>4.3: Network abstraction used in the system</td>
<td>33</td>
</tr>
<tr>
<td>4.4: Network class diagram</td>
<td>34</td>
</tr>
<tr>
<td>4.5: Configuration framework</td>
<td>35</td>
</tr>
<tr>
<td>4.6: Configuration file names and example</td>
<td>35</td>
</tr>
<tr>
<td>4.7: Server model class diagram</td>
<td>37</td>
</tr>
<tr>
<td>4.8: Lookup system protocol</td>
<td>38</td>
</tr>
<tr>
<td>4.9a: Reservation Table class diagram</td>
<td>39</td>
</tr>
<tr>
<td>4.9b: Structure of the Reservation Table</td>
<td>39</td>
</tr>
<tr>
<td>4.10: Initiation and synchronization phase of the registration protocol</td>
<td>41</td>
</tr>
<tr>
<td>4.11: Registration phase of the registration protocol</td>
<td>42</td>
</tr>
<tr>
<td>4.12: Plug-in architecture class diagram</td>
<td>43</td>
</tr>
<tr>
<td>4.13: Initiation and admission phase of the reservation protocol</td>
<td>46</td>
</tr>
<tr>
<td>4.14: Atomic Transaction</td>
<td>47</td>
</tr>
<tr>
<td>4.15: Refresh phase of the reservation protocol</td>
<td>48</td>
</tr>
<tr>
<td>4.16: Strategy architecture class diagram</td>
<td>50</td>
</tr>
<tr>
<td>5.1: Package organization of the registration/reservation system</td>
<td>54</td>
</tr>
<tr>
<td>5.2: Plug-in code without native call</td>
<td>56</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Background

Video conferencing, web radio, video on demand, and web telephony, to name a few, are some of the Multimedia applications that have invaded the Internet and intranet world. They bring dynamic content to an otherwise static text-based environment. A subset of real-time applications, Multimedia systems deliver continuous media, like audio and/or video, through the network, requiring a higher degree of service than the usual Internet traffic. The Internet infrastructure was not build to provide such demanding service levels. To support these applications and provide them with Quality of Service, new software and hardware architectures have been created and added to the Internet infrastructure. These architectures concentrate on providing the end systems with quality guarantees by means of managing the resources in the network.

In addition to the changes in the services provided by the Internet, the next generation of computing devices has concentrated on portability and ubiquity. Ranging from PDA’s to laptops, these new devices are becoming end systems to the already eclectic Internet. Aside from using
new means of connectivity to the network, like wireless, these devices are running distinct platforms and have variable capabilities.

1.2 Problem Description

Multicasting is an efficient method of distributing multimedia content to a group of receivers. In a multimedia multicasting environment, an overlaid network of nodes connects the sender with a group of receivers forming a multicast tree. Figure 1.1 depicts the multimedia multicasting tree. The media content travels through the edges of this tree until it reaches a Multimedia Gateway, which relays the content to the subscribed receivers.

![Multimedia Multicasting Tree](image-url)

Figure 1.1: Multimedia multicasting tree
This Multimedia Gateway is responsible for managing the different receivers by acquiring information about each of them, performing admission control, and allocating the necessary resources. By managing the available resources, the Gateway can make Quality of Service guarantees to the receiver. The Gateway also has capabilities of altering the multimedia content to custom fit it to the receiver’s requirements.

Our work focuses on the edge of the multicast tree, the interaction between the Multimedia Gateway and the receivers. The main problem in this section of the multicast tree is the direct interaction with a variety of end systems and mediums of connectivity. For example, the Gateway might support an HP Jornada running Windows CE connecting through a wireless interface and a Windows PC connecting through a 100Mbit Ethernet. The heterogeneity present in this section of the tree demands that the communication protocols and policies between the Gateway and the receivers be flexible and customizable. The system implementing the protocols and service policies must be easily upgradeable to provide support for new types of receiver devices and remain ubiquitous.

1.3 Our Approach

Our approach is to implement a flexible and customizable Multimedia Gateway that can be easily integrated into any network environment and upgraded to support changes to its surroundings. The main goal is to externalize some of the policies and mechanisms used to manage the receivers and make them programmable, while maintaining a constant communication interface between the Gateway and the receivers.
To manage the group of receivers that connect to the Gateway, we implement two systems, registration and reservation. The registration system is responsible for profiling each receiver by gathering a set of attributes and registering them in a table. Examples of these attributes are: the type of multimedia service that the receiver is requesting, the type of network connection that the receiver device is using, the type of media players installed, etc. Each attribute is collected by a plug-able component that implements the functionality required to gather the information. These components are active and programmable. By active we refer to their ability of being distributed by the Gateway to each receiver, dynamically configuring the receiver side of the registration system. By programmable we refer to being able to implement new components and customize the set of attributes profiled by the Gateway.

The reservation system is responsible for allocating the required resources to establish a particular Quality of Service for the receiver. The resources, required to service a specific receiver, are calculated from the information gathered about the receiver by the registration system. At the Gateway, the reservation system performs admission control by determining if enough resources are available to service the receiver’s request. If so, the required resources are reserved and used to exclusively service the receiver. The reservation system flexibility comes from allowing the programmability of the policies, utilized to perform admission control and resource management. There are no fixed admission control and resource management policies, hence the administrator of the Media Gateway can decide which resources to monitor and how they are affected by the characteristics of the receiver.

The principle idea is that customizing both systems, registration and reservation, can efficiently integrate the Media Gateway into its network environment. Re-configuring the
Gateway implementation, by exchanging the programmable components, allows changes to the network infrastructure and the introduction of new receiver devices.

1.4 Thesis Outline

The thesis is divided as follows. Chapter 2 describes the related work in the area of resource reservation and Quality of Service. It points out the different mechanisms and protocols used to provide a Quality of Service network model. In addition, the chapter presents work done on Component-Based framework. The framework emphasizes flexible and configurable network applications. In Chapter 3, the system design is presented. It introduces the registration and reservation system and explains how they work together. Chapter 4 provides the implementation details of both systems. It includes the components used to build the registration and reservation system, the protocols used for communication between the Gateway and the receivers, and the programmable component framework. Chapter 5 explains how the system can be integrated to the network environment through customization. It provides instructions on how to implement the programmable components used by the system. Chapter 6 concentrates on testing the system and running experiments to validate all features in the registration-reservation system. Finally, chapter 7 concludes the thesis. It summarizes the accomplishments and identifies future work.
CHAPTER 2

Related Work

2.1 Background

With the best effort service model that the Internet is built upon, real-time applications need specialized protocols, techniques, and entire systems that provide the end host with optimal Quality of Service (QoS). In a best effort service model, there are no service guarantees; it limits itself to creating a point-to-point connection. In order for real-time applications to work over the IP centric network, other service models are desirable. Two service models that opted to provide these applications with QoS are Differentiated Service (DiffServ) and Integrated Service (IntServ). New QoS friendly protocols were proposed to implement the ideas stated by these two new service models.

Scalability and heterogeneity are also important issues that need to be dealt with when implementing network solutions. In a multicasting environment, scalability becomes a concern as the overhead on the network increases drastically with an increasing number of receivers. Additionally, by allowing multiple heterogeneous receivers, it is critical that these QoS enabling systems be flexible. Flexibility refers to the protocol’s capability of supporting a wide variety of
machines at the client side. This is important because an increasing number of smaller devices are accessing the Internet with hopes of getting more content. The systems discussed in this chapter attempt to solve these issues.

2.2 Service Models and Protocols

There are two main architectures defined by the Internet Engineering Task Force (IETF) to provide QoS guarantees to the end users; these are DiffServ and IntServ. If we picture a spectrum of architectures developed to provide QoS guarantees to applications (Figure 2.1), DiffServ will be on one end and IntServ on the other end of the continuum.

The DiffServ architecture uses a small number of classes that are used to classify each flow. These classes guarantee a certain level of service. A higher-level class of service means that the flow will have priority over any other flow with a lower-level class of service. This information is stored in a set of bits in the header of every packet and its length is dependent on the number of classes of service that the architecture provides. One of the advantages of DiffServ is that there is no need to store any per flow information at the routers since the flows are aggregated, allowing it to be more scalable. The overhead at the routers only exists to discriminate between packets based on their class to service them accordingly. Unfortunately, DiffServ leads to a less flexible implementation, with less utilization and assurance levels than its counterpart IntServ ([1]).

On the other side of the spectrum lies the IntServ architecture. It relies on the storage of per flow state information at every enabled router. This state information is set by a signaling protocol that creates the reservation based on the result of an Admission Control mechanism.
Every flow is handled individually with per flow information, making the level of service granted to the flow equivalent to the allocated resources. It provides a more comprehensive approach since there is no pre-established limit on the number of flow classes. In other words, the application is given the exact resources necessary or what was negotiated during Admission Control. The disadvantage of this system is that it is expensive to implement since every router needs to store Per-flow State, resulting in scalability problems. Additionally, the overhead of the signaling protocol needs to coexist with regular traffic in order to maintain the reservations. Figure 2.1 shows where the discussed protocols fall within the QoS friendly service model spectrum.

![QoS friendly service model spectrum](image)

The protocols implemented to provide a QoS model are along this spectrum. On the IntServ side the most popular is the Resource ReSerVation Protocol (RSVP). It is a receiver-initiated protocol, which allows for the existence of heterogeneous receivers. This is a desirable characteristic in a multicasting environment. It gracefully deals with changes in multicast groups. ([2]) RSVP allows the end user to specify its application needs and the resources needed are reserved accordingly. The protocol works as follows: There is a PATH message that goes from the sender to the receiver, which traces the path that the data is going to traverse. Then the receiver sends a RESV message on the reverse path of the PATH message in order to ask for the required resources. This RESV message contains the specification of the flow and the service requested by the receiver. If the resources can be allocated along the entire reverse path, then the communication starts. The reservations made are stored in each router along the path in “soft
state” form. This means that the state information is only valid for a certain amount of time and that it needs to be refreshed in order for it to remain valid. Thus, with RSVP, the receiver needs to keep sending RESV messages up the reverse path to refresh the reservation it made. This prevents the need for an explicit teardown message to remove the reservation made. The use of the PATH message allows for changes in the path that the packets take down the multicast tree, and provides to the receiver with the necessary information so that it can establish the reservation of resources along the new path.

Other efforts have concentrated on improving the performance of RSVP. Reduced Overhead RSVP (REDO RSVP) aims at reducing the overhead of the protocol. It rethinks soft state scalability and control message overhead. It allows flows to be aggregated on a per host basis rather than in different classes, maintaining the desired reservation flexibility. This is important because there are no limitations imposed by a fixed number of classes. Each host has the specific resources needed to provide the requested QoS.

The Flow Initiated and ReServation Tree (FIRST) protocol ([3]) is an attempt to combine the best characteristics of DiffServ and IntServ. It lies somewhere in the middle of the spectrum mentioned above. It stores flow state, similar to RSVP, but instead it uses “hard state” to store the reservation information in the routers. This avoids the need to send refresh RESV messages that occupy bandwidth, but requires an explicit teardown message to terminate the flow and de-allocate the resources. Similar to DiffServ, FIRST aggregates the flow into several service levels making it a more scalable solution. It is also a receiver driven protocol, which gives the ability to handle heterogeneous receivers. An advantage of FIRST is that as the flow initiation message, another name for a RESV message in RSVP, goes toward the sender up the multicast tree, it searches at every router to see if the service flow is already available. If the flow is available at
that node, the reservation stops there; otherwise it keeps going up until it reaches the sender. At the same time it reserves the resources for the requested flow. This is referred to as aggregating flows, which makes the system more scalable.

This idea of aggregating flows is also explored in the Scalable Resource Reservation Protocol (SRP) ([4]). SRP tries to automatically aggregate flows so there is no knowledge of individual flows inside the network but only at the edge of the network. This makes the integration of SRP into the network infrastructure easier since only the edge routers need to know about the protocol. There is also no need for a signaling protocol since it works similar to DiffServ by giving each packet a type. These types are “reserved”, “request”, and “best-effort”, and they serve as the basis of the protocol. The sender initiates the reservation by sending the flow with all packet types as “request”. The SRP enabled routers, at the edge of the network, use an Estimator to examine each packet to predict if accepting it will exceed the available resources. If the available resources are not exceeded, then the packet is forwarded with the same type. If accommodating the packet is predicted to exceed the available resources, then the packet type is downgraded to the “best-effort” type. At the flow destination, the amount of “request” type packets received is analyzed, and this information is sent back to the sender as feedback. After this, the sender has an estimate of the accepted rate and can send the amount of packets with the “reserved” type, which are automatically forwarded by the SRP routers. This protocol relies on a good estimation algorithm to work correctly. The sender is the only one that keeps resource information for each of its receivers. The SRP routers have a total of resources used by all flows. This makes the protocol scalable. One of the disadvantages of SRP, however, is the need for feedback from each receiver. In a multicasting environment, this leaves the sender vulnerable of being flooded by upstream messages.
2.3 Mechanisms and Techniques

Other ways of providing QoS to the clients is by manipulating the data stream. This is crucial in a multicasting scenario where multiple clients receive the same flow but at different qualities because of their individual capabilities. A server-based approach performs poorly in this heterogeneous environment because the same data stream cannot satisfy all receivers. By moving the responsibility down to the receiver, heterogeneity is better accommodated. This is because the receiver has control of the quality of service received. Receiver Driven Layered Multicasting (RLM) is one of the methods suggested. It divides a data stream into different layers with incremental quality. A receiver can choose to subscribe up to the maximum layer that it can effectively handle. RLM easily adapts to congestion situations by dropping layers and improves scalability by aggregation of the flows. This is all possible using the existing best effort network and adding an extra protocol to subscribe and unsubscribe from layers.

A disadvantage of RLM is that it can waste bandwidth when the higher quality layers are not used and the quality is limited to the highest layer. An alternative method could be to provide transcoding capabilities inside the network, either in routers or in gateways at the edge of the network. The process of transcoding involves transforming a data stream by effectively reducing its bit rate. Of course, this alteration has to be performed rapidly to prevent delays. In a Video Conferencing System these delays are crucial. Transcoding involves decoding, re-quantising, and re-encoding the stream. With video, instead of varying the bit rates, the frame rate could be varied. The idea behind dynamic frame rate control is to send the highest quality video along with a file that contains motion vectors for lower frame rates. It works with video compression
algorithms that use motion estimation like MPEG and H.263. As with transcoding, when necessary the frame rate is lowered, thereby lowering the bit rate.

2.4 QoS Enabling Architectures

Another research area develops complete systems that inherently provide QoS. They consist of protocols and techniques that enhance the end-to-end performance. In [13], a Video Gateway is implemented that combines the Real-Time Transport Protocol (RTP) and transcoding mechanisms. In a simple scenario, it serves as a relay of RTP packets. In case that adaptation is necessary, the gateway transcodes the ingress flow using one of the methods explained above and sends the resulting lower rate flow out the egress. The Globus Architecture for Reservation and Allocation (GARA) is a QoS architecture that aims to manage resources by separating the reservation and allocation process ([14], [15]). This allows applications to make advance reservation for resources even if they are not available. The application can wait for high demand resources to be released rather than failing immediately. In [14] GARA is extended to explore the combination of reservation with adaptation. It uses admission control at the edge of the network along with flow aggregation through a DiffServ network. The system requires actuators, which allow a reservation to be dynamically modified when changes in the network are detected by sensors. When sensors fire, an array of decision procedures is in place to decide which actuator to enable.

For multicasting applications, an application level multicasting tree, made of virtual nodes, is often laid over the existing network. A virtual network refers to a group of nodes that may not be physically connected to each other, but exchange messages. These messages are part
of a protocol that is QoS aware. The Multicast Heterogeneous Packet Flow (MHPF) is composed of MHPF servers that form the multicast tree ([16]). Each MHPF server uses rate adaptation and priority filtering to manipulate the flow and abide to the quality requirements. The servers communicate with each other through Heterogeneous Packet Flow protocol (HPF), an end-to-end unicast transport protocol that assigns priorities to flows, in essence aggregating them into different classes like DiffServ.

2.5 Configurable Architectures

The fast dynamism of network computing demands flexible software components that can be easily customized to support new services and new devices. Kon et al [17, 18] propose a Multimedia Distribution Framework built on the idea of a component-based architecture. The key component is a network of Reflectors. These Reflectors are responsible of distributing the data through the network in an efficient way [18]. They are implemented in software providing the flexibility of reconfiguration. By reconfiguring the Reflectors, their internal policies and interconnections can be modified to improve their performance. In addition, plug-able components can be combined in different arrangements, creating new applications. The ever-changing nature of the Internet demands the development of flexible and customizable components.

This chapter provided an overview of research related to the implementation of the proposed registration-reservation system. The following chapters, describe the design and
implementation of this system, focussing on building a protocol with enough flexibility to support new devices and services.
CHAPTER 3

System Design

Multicasting is an efficient mechanism for distributing multimedia data through the network. The data travels from the senders, through the multicasting tree, until it reaches the receivers. Figure 3.1 illustrates the multimedia distribution environment.
The multimedia distribution environment is composed of senders, a multicast tree, Media Gateways, and receivers. The senders are the service provider. They are the source of the multimedia data. The multicast tree is composed of an overlaid network of nodes that are responsible of distributing the data in an efficient manner. The Media Gateway sits at the edge of the multicast network and manages the receiver’s request. Finally, the receivers are the recipients of the data transmitted. Our work focuses on the interaction between the Media Gateway and the receiver. Figure 3.2 depicts the area of interest of the multimedia distribution system.

![Figure 3.2: Area of interest of the multimedia distribution system](image)

The Media Gateway directly interacts with the receivers. It gathers information about each receiver, performs admission control to the network, and manages the network resources. By performing admission control and establishing reservation of resources, the Media Gateway provides the receiver with its required Quality of Service. The main problem encountered by the Media Gateway is in dealing with the heterogeneous nature of the receivers. It needs to provide its services to receivers that run on different types of devices and use different modes of
connectivity. This heterogeneity demands the design of a flexible and customizable Media Gateway that can be easily modified to support the changes to its surroundings.

This chapter explains the design elements that compose a Media Gateway. It emphasizes the protocols and policies used to interact with the receivers. First, the chapter describes the assumptions made to simplify the design of the gateway system. Then, it establishes the main goal followed by the details of the collaborating components.

### 3.1 Overall Design

#### 3.1.1 Assumptions

Our work focuses on the interaction between the Media Gateway and the receivers (see Figure 3.2). We assume that a multicast tree is in place and that the multimedia data is somehow available at the Gateway. Additionally, by managing the resources at the Media Gateway, the receiver is provided with the Quality of Service requested. Finally, we assume that each Gateway services the receivers present in a particular physical area. For example, in a building, each room is managed by an individual Gateway and all the receivers in that room must connect to that particular Gateway. With knowledge of its location, a receiver queries a lookup service to determine the Gateway it must interact with. These assumptions allow the design and the development of our Gateway system to be done independently from the rest of the multimedia distribution environment.

#### 3.1.2 Design Goals
The interaction between the Media Gateway and the receivers consists of a series of protocols for communications as well as a set of policies that ultimately work together to provide a multimedia service to the receiver. The main goal of our design is to develop a customizable and flexible Gateway system that can support a wide variety of receivers. Customizable refers to the ability to modify the Gateway system to integrate it into its environment. Flexible refers to ease of upgrading to support changes in the environment.

3.1.3 Design Approach

The overall design of the Gateway system is divided into two major parts: the Registration System and the Reservation System. Figure 3.3 illustrates the overall view of the system.

![Figure 3.3: Overview of the gateway system](image)

The Registration System is responsible for profiling each receiver by gathering a set of attributes and registering them in the Reservation Table. Examples of these attributes are: the type of service requested, the type of network connection used, the media decoders installed, etc. This information is then accessed by the Reservation System to calculate the required resources.
needed to service the receiver’s request. In addition, the Reservation System manages the access to the multimedia distribution system as well as the resources used to service the receivers. Each system implements a standard protocol to communicate with the receivers.

To accomplish customizability, both the Registration and Reservation system are designed around the Strategy design pattern. The idea behind the Strategy pattern is to encapsulate a set of algorithms individually and make them interchangeable. In the Registration System, the mechanisms used to gather the receiver’s attributes are individually encapsulated into plug-able components. These components are distributed to the receiver, effectively reconfiguring the receiver end of the Registration System. In the Reservation System, the resource management and admission control policies performed at the Gateway are also interchangeable. They can be customized to successfully integrate the Reservation System into the network scenario.

3.2 Lookup Service

Before explaining the communication between the Gateway and the receivers, it is important to understand how the receivers find the appropriate Gateway. The system assumes that every receiver resides in a physical location. Each location is serviced by a particular Gateway. All the locations make up a domain managed by the Lookup Service. For example, we could visualize a domain as a building and each room in the building as a specific location. The Lookup Service is a well-known service provider used by the Gateway to register itself as the manager of a location. Figure 3.4 illustrates the Lookup Service protocol.
A Gateway registers with the Lookup Service to advertise itself as the manager of a particular location. Within the Lookup Service, the address of the Gateway is mapped to the name of the specific location. This location must lie within the domain managed by the Lookup Service. With knowledge of its location, the receiver queries the Lookup Service to find the particular Gateway that it must connect to. In a sense, the Lookup Service is like the DNS, which converts an URL into the address of the host machine. The difference between a Lookup Service and a DNS lies in that the mappings made by the Lookup Service are completely geographical.

3.3 Registration System

The Registration System implements the first phase of communication between the receiver and the Gateway. The registration agents of the Gateway and the receiver communicate using the registration protocol. During this phase of communication, the Gateway profiles the receiver and enters the receiver’s information into the Reservation Table.

3.3.1 Registration Protocol

The registration protocol is used to communicate between the registration agent of the receiver and the Gateway. In general, it is used to pull information from the receiver regarding the service requested, the Quality of Service parameters, the host resources, etc. This information
is then stored in the Reservation Table for later use by the reservation protocol. The sole purpose of this protocol is to establish a standard interface between the receiver and the Gateway and make an entry for the receiver in the Reservation Table. Figure 3.5 illustrates the registration protocol.

The registration protocol is receiver-initiated. The receiver sends a request to the Gateway to start the registration protocol along with a client identification string (ID). This ID is any form of identification, different for all receivers, which is used to create an entry for the receiver in the Reservation Table. The Gateway acknowledges the request to signal that it is ready to proceed. The protocol enters into the synchronization phase. During synchronization, the Gateway remotely reconfigures the receiver side of the registration system by distributing the necessary plug-able components to the receiver. These components implement the functionality required to perform a successful registration. After the receiver is synchronized, it executes all the plug-able components and gathers the profiled attributes. This information is then sent to the Gateway, which creates an entry in the Reservation Table for the receiver. The final
acknowledgement from the Gateway finalizes the protocol. The information is kept in the Reservation Table and used by the Reservation System.

### 3.3.2 Customizability and Flexibility

The Registration System provides flexibility by permitting the Gateway to remotely reconfigure the receiver side of the registration system. At the receiver, the registration system only implements the communication protocol but not the profiling mechanisms. These profiling mechanisms are encapsulated into plug-able components that reside at the Gateway. During the synchronization phase of the registration protocol, the receiver requests for these components. By executing a plug-able component, a specific attribute of the receiver is profiled. For example, a component may implement the profiling functionality to determine the type of connectivity used by the receiver.

This feature allows the customization of the Registration System. Each Gateway can distribute components to the receiver that make sense with the location managed. In essence, the receiver’s profiled information is reliant on the available components. To extend the functionality of the entire Registration System, new components can be implemented.

### 3.4 Reservation Table

The Reservation Table is the link between the reservation and registration protocols. Each protocol has a different use for the table. The Registration System writes into the Reservation Table while the Reservation System reads from it. The information gathered from each receiver during the reservation protocol is stored in the table and mapped to the receiver’s
ID. This ID is then used by the Reservation System to access the registered receiver data. This data is used to calculate the resources required to provide a service guarantee to the receiver.

3.5 **Reservation System**

The Reservation System implements phase following a successful receiver registration. It uses the registered information to calculate the resources required to service the receiver.

3.5.1 **Reservation Protocol**

The reservation protocol is used for communication between the reservation agents of the Gateway and the receiver. The protocol reads an entry for the connected receiver from the Reservation Table and makes a decision about the admissibility of the service requested. If the service request passes admissions, then the Gateway will allocate the resources needed to deliver the service. The reservation protocol also manages the soft state reservation and refresh messages coming from the receiver. Figure 3.6 illustrates the reservation protocol.
The reservation protocol is also receiver-initiated. The receiver sends a START request along with its ID string. This process is repeated because the Registration and Reservation Systems are independent entities. The Gateway then retrieves the receiver’s information stored in the Reservation Table. This information was gathered during the registration phase. After the Gateway acknowledges, the receiver sends an ADMISSION request. The Gateway performs Admission Control by calculating the required resources and verifying their availability. It uses the following policy to determine if the receiver can be admitted.

\[
\text{Allocated Resources} + \text{Necessary Resources} \leq \text{Total Resources}
\]

The Allocated Resources is the amount of resources that are not available because they are reserved for other receivers. The Necessary Resources is the amount of resources needed to service the current receiver. Finally, the Total Resources is the total amount of resources available at the Gateway. For a successful admission, the above statement needs to be true for each of the resources that the Gateway manages.

If the receiver request is admitted, the Gateway acknowledges by delivering a Contract that specifies the refresh rate for keeping the resource reservation alive. If the receiver agrees with the Contract, it signs the Contract, binding itself to the terms of the reservation. The signed contract is sent along with the SERVICE request. At the Gateway the contract is verified for signature. If the contract is signed, then the necessary resources are allocated, the service begins, and an acknowledgement is sent back to the receiver. Once the service has begun, the receiver needs to send REFRESH messages at the regular intervals specified in the Contract. A failure to send a REFRESH message on time voids the reservation. When this happens, the resources previously being used for the receiver’s service are freed and available for use by other receivers.
### 3.5.2 Resource Table and Soft State Reservation

The Gateway manages a Resource Table used to keep track of the resources that are available for reservation. For each resource managed by the Gateway, the table maintains the total amount and the allocated amount. Figure 3.7 illustrates a sample state of the Resource Table.

![Resource Table](image)

The total amount of a resource is not necessarily 100% because some of the resource may be used by the Gateway itself, so it is not available for reservation. From the table, the available amount of resources is determined by the following equation.

\[
\text{Available Resource} = \text{Total Resource} - \text{Allocated Resource}
\]

For example, Resource A is 100% available for reservation, but 30% has already been allocated. This means that only 70% of Resource A is available to other receivers.

To maintain its resources allocated, the receiver has to send REFRESH messages to the Gateway. Each REFRESH message is sent periodically at a specified time interval. When failed to send a REFRESH message on time, the Gateway frees the resources and cancels the...
reservation. This is known as soft state reservation and is based on the reservation mechanism implemented by RSVP. In soft state reservation system, the resources are not allocated permanently. Thus, the reservation needs to be refreshed by the receiver at periodic intervals.

Our Reservation System goes a bit further than RSVP. It allows the refresh intervals to be decided at runtime according to the individual characteristics of the receiver. RSVP implements a static refresh rate. A static refresh rate does not accommodate the heterogeneous nature of the clients. By gathering registration information from each receiver, the system can decide on an optimal refresh rate. Thus, depending on the type of device hosting the receiver, the refresh rate can be assigned accordingly. For example, in a particular scenario where two receivers have established a reservation, one wired and the other wireless, the refresh interval is dependent on the reliability and on the bandwidth capacity of the connection. A wireless connected receiver is less reliable than a wired one. A wireless receiver would need to notify the Gateway more frequently to maintain a successful reservation. At the same time, a wireless connected receiver has less bandwidth capacity. Ideally, bandwidth capacity should be used for data rather than control messages, suggesting the need for a lower notification frequency. Thus, the system must find a balance for the optimal refresh rate.

3.5.3 Customizability and Flexibility

The Reservation System provides flexibility by implementing customizable policies. These policies include admission control, resource management, and reservation management. The policies are programmable and interchangeable. The gateway administrator can implement custom policies tailoring the Reservation System to its network environment. In the admission control and resource management policies, the administrator can select the resources managed by
the Gateway. The reservation management policies allow the Gateway to create a customized Contract for the receiver based on the receiver’s registered information. The Contract contains the refresh interval for the REFRESH messages. The Registration System does not implement a constant refresh interval for all receivers; rather, it can determine the appropriate interval from the information gathered from the receiver during registration.

This chapter presented the main idea behind the design of the system: the ability to create a customizable and programmable protocol, by allowing it to be tailored to the needs of the environment. The next chapter discusses the implementation issues of the system. It focuses on the lower level details of the components of the system as well as the elements that enable the customizable framework.
CHAPTER 4

System Implementation

This chapter provides the implementation details of the development of the Registration and Reservation Systems. It starts with an explanation of the overall system model followed an introduction into the customizability of both systems. The remainder of the chapter goes more in depth into the components that make up each system.

4.1 System Model

The interaction between the Media Gateway and the receivers is implemented in two separate systems, registration and reservation. Both systems utilize the client-server model, the servers residing at the Media Gateway, the clients at the receivers, and the protocols to provide communication. The Registration System is composed of the Registration Server, the Registration Client, and the Registration Protocol. The Registration Protocol provides a standard communication scheme between the Registration Server and the Registration Client. During the registration phase of the Registration Server profiles the Registration Client and makes an entry for the client in the Reservation Table. With an entry already in the Reservation Table, the Reservation System can calculate the resources required to service the receiver’s request. During
the reservation phase, the Reservation Server and the Registration Client interact through the Reservation Protocol. The Reservation Server determines, through admission control, if enough resources are available to service the receiver. The Registration Client requests for the admission and once the resources are reserved, it periodically signals the Registration Server to maintain the reservation alive. Figure 4.1 depicts a detailed view of the Registration and Reservation Systems and how they fit in the overall picture.

In addition to the client-server model, the Media Gateway is a multi-threaded entity. Both the Registration and Reservation Servers service the clients through exclusive threads of communication. This allows the Media Gateway to treat receivers concurrently. When the receiver first connects to the Media Gateway, it interacts with the registration thread. After the
registration, the receiver is hand-off to a reservation thread to continue the communication. The protocols that communicate between the threads with the corresponding clients run on top TCP/IP Sockets, providing delivery reliability and making the protocol design simpler.

4.2 Customizable Components

The main problem that this implementation of the Media Gateway attempts to solve is the heterogeneity of the receivers. The Media Gateway sits at the edge of the multimedia distribution system interaction with a variety of receivers running on different platforms and using different modes of connectivity. By implementing a Gateway system that is both customizable and flexible, the multimedia distribution system becomes ubiquitous.

Both the Registration and the Reservation system attempt to provide a solution to the problem. They use the Strategy Design Pattern to allow the interchangeability of certain components.

4.2.1 Plug-able Framework Overview

The Reservation System uses a Plug-able Framework to provide customizability. The algorithms used to profile the receiver are encapsulated into a plug-able component referred as plug-ins. These plug-ins are programmable and are distributed from the Reservation Server to the Registration Client. They implement the functionality needed to gather a particular attribute of the receiver. For example, a plug-in may be implemented to find out what is the total memory that the receiver has available. The Registration Client is a thin agent that has knowledge of requesting plug-ins, executing them, and returning their results to the Registration Server. The
Registration Server, by distributing the plug-ins, effectively reconfigures the functionality of the Registration Client. Figure 4.2 illustrates the Plug-able Framework.

![Figure 4.2: Plug-able Framework](image)

### 4.2.2 Custom Strategy Framework Overview

The Reservation System also utilizes the strategy pattern but strictly in the Reservation Server. The Custom Strategy Framework allows some of the policies used by the Reservation Server be encapsulated as interchangeable components, referred as custom strategies. Among these policies is admission control, allocation of resources, de-allocation of resources. The framework enables the administrator of the gateway to implements custom mechanisms to better integrate the Reservation System into the network environment. These strategies decide what resources are managed and how they are affected by receiver’s attributes. For example, one environment may demand the management of the CPU and the Memory, while another may demand management of the Bandwidth and the number of concurrent connections. This flexibility makes the system suitable for an environment that is constantly changing.
4.3 Programming Environment

Implementation decisions are based upon building a system that is flexible and capable of being integrated into the select environment. Java\(^1\) is the programming language of choice because of its ability to run on many platforms. For this version, the target platforms are Windows NT/9X and Windows CE. The server is targeted only for a Windows NT/9X machine but the client must run on both platforms. These constraints determine which version of Java should be use. For the server implementation, Sun’s JDK 1.2.1 or 1.1.8 are suitable, but for the client the code needs to respect the JDK 1.1 specification. As of the implementation of the system, the version of Sun’s Personal Java\(^2\) for Windows CE only supports the JDK 1.1. To avoid confusion, the JDK 1.1.8 is used to develop and run the different parts of the system. In addition, the Swing 1.1.1 Java Foundation Classes are included for implementation of the GUI and use of Timers.

4.4 Shared Components

Both the Registration and the Reservation Systems share the components explained in this section. Among these components are the network abstraction, the configuration framework, the singleton objects, and the server model.

4.4.1 Network Abstraction

---

\(^1\) http://java.sun.com
\(^2\) http://java.sun.com/products/personaljava/
The Registration and Reservation Systems are implemented using the client-server model for network communication. The *ObjectExchangeNetwork* is the class that provides the network abstraction used for communication between the agents of the system. At the core, communication is achieved through TCP sockets. By using Java’s stream objects, *ObjectOutputStream* and *ObjectInputStream*, entire objects are exchanged rather than byte streams. This provides a superior Object Oriented design. The network abstraction adds simplicity to the implementation of the protocol by dealing only with objects going back and forth through the network. Any object implementing the Java *Serializable* interface can be sent through the *ObjectExchangeNetwork*. To transmit objects through the network the send and receive methods are called on the instance of the *ObjectExchangeNetwork*.

The registration and reservation protocols use only two objects to encapsulate anything transmitted through the network. These are instances of the *Result* and the *Request* class. Figure 4.3 demonstrates the network abstraction of the system.

Sending an instance of *Request* will return an instance of *Result*. A *Result* object may contain information or it may be used as an acknowledgement of the *Request* sent. In this system, the client sends *Request* objects and receives *Result* objects. On the other hand, the server receives *Request* objects and sends *Result* objects. These classes are composed of a type and piggybacked data. Figure 4.4 illustrates the class diagram for network classes.
The data is stored in a Hashtable, mapping the data object with a name. The name is used to identify the data object in order for the receiver to retrieve the object. The type attribute describes the instance of Request or Result in terms of the protocol. The types are defined in the Type interface, implemented by both classes. A protocol is defined by a sequence of Request and Result objects with different types and data.

4.4.2 Configuration Framework

Agents in the system can be configurable, allowing properties to be changed by editing a configuration file, resulting in improved usability. Examples of these properties are the address and port of the Lookup Service, the location of the plug-ins, etc. (For a complete list of all the properties refer to Appendix A). The configuration framework is composed of the ConfigurationAgent class and the Configurable interface (see Figure 4.5).
The `ConfigurationAgent` class extends the functionality of Java’s `Properties` class by making the name of the configuration file, where the properties are loaded from and saved to, part of the class. A configurable object implements the `Configurable` interface. The interface includes a method to access the instance of the `ConfigurationAgent` and a `configure` method to shape the configurable instance. In this system, `RegistrationServer`, `RegistrationClient`, `ReservationServer`, `ReservationClient`, and `LookupServer` are the classes that implement the `Configurable` interface. The `configure` method is called within the constructor of these classes. The names of their configuration file are hard-coded in each class. Figure 4.6 shows a table with the names of each file and an example configuration file.

<table>
<thead>
<tr>
<th>class name</th>
<th>configuration file name</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegistrationServer</td>
<td>gateway.cfg</td>
</tr>
<tr>
<td>RegistrationClient</td>
<td>client.cfg</td>
</tr>
<tr>
<td>ReservationServer</td>
<td>gateway.cfg</td>
</tr>
<tr>
<td>ReservationClient</td>
<td>client.cfg</td>
</tr>
<tr>
<td>LookupServer</td>
<td>gatewaylookup.cfg</td>
</tr>
</tbody>
</table>

Figure 4.6: Configuration file names and example
The *ConfigurationAgent* allows the file to be modified. New properties can be added and existing ones can be edited. Appendix A contains tables for all configurable classes defining the properties that each class uses.

### 4.4.3 Singleton Objects

A singleton object is an object of which only one instance exists. The pattern only allows the instantiation of the object through the `getInstance` static method. The method returns the singleton instance or creates one if missing. It is the only way of instantiating one of these classes since the constructors are protected, hence not accessible outside the class. All main classes in the system implement the Singleton pattern. They are: `RegistrationServer`, `ReservationServer`, `RegistrationClient`, `ReservationClient`, and `ReservationTable`. This allows easy access to the object within an instance of the Java Virtual Machine.

### 4.4.4 Server Model

The classes that implement the servers of the system follow a similar pattern. These are the `RegistrationServer`, `ReservationServer`, and `LookupServer`. Servers are capable of servicing multiple clients concurrently and every time that a client connection is accepted, a new thread originates to service the client exclusively. The super class of these connection threads is the abstract class `ServerConnection`, which extends Java’s `Thread` class. It instantiates the `ObjectExchangeNetwork`, used for communication, and provides the abstract `run` method which is overridden by subclasses to implement the specifics of the protocol. Figure 4.7 shows a class diagram with all the associated classes.
The subclasses of `ServerConnection` are, `RegistrationConnection`, `ReservationConnection`, and `LookupConnection`. When a new connection is accepted, an instance of one of these classes is created and from then on, it carries out the communication between the server and the client. For example, in the registration subsystem, the instance of `RegistrationServer` is responsible for listening to new connections from instances of `RegistrationClient`. When a new connection is established the client continues the communication with a new instance of `RegistrationConnection`. The `RegistrationConnection` class implements the server-side of the registration protocol.

### 4.5 Lookup System

Before any communication initiates between the Registration Server and the Registration Client, the entities need to find each other. This is done indirectly through an instance of the `LookupServer` class. The Lookup Server behaves as a phone book. It manages a domain, composed of several locations, each serviced by a Registration Server. For example, a Lookup Server’s domain might be a building and each room inside the building is a location serviced by
a Registration Server. The Lookup Server runs in a well-known machine and port. When either a Registration Server or Registration Client connects to it, an instance of LookupConnection is created, which services the connection. It waits for two different types of requests, REGISTER and QUERY. Figure 4.8 depicts the specifics of the communication.

![Diagram of Lookup system protocol]

Figure 4.8: Lookup system protocol

A Registration Server will send an instance of Request with the type set to REGISTER. Along with the request is the host machine name and port where the Registration Server waits for clients to connect and the location that it services. After mapping the location name with the address of the server, the lookup server completes the communication. When a Registration Client starts, it finds the Registration Server by sending to the Lookup Server an instance of Request with the type set to QUERY. Along with the request is the location name where the client resides. The Lookup Server looks in the table and resolves the query and sends an instance of Result with the address and port of the Registration Server for the location. With this
information, the Registration Client is able to initiate the communication with the registration client.

4.6 Reservation Table

The ReservationTable class implements a storage data structure that keeps the per-client information profiled. Composed of a Hashtable of Hashtables, it allows the storage of different pieces of information depending on the client’s type of host machine. Figure 4.9a illustrated the class diagram of the ReservationTable. To create a registration entry or set a client’s profiled value a call to setValue is made to the Reservation Table. If there is no entry in the table for the client, a new Hashtable is created and mapped to the clients identification string. This new Hashtable is then inserted into the root Hashtable. Any new item of information is then stored in its client’s Hashtable, mapped to the display name of the plug-in used to profile the value. Figure 4.9b demonstrate the structure of the reservation table. To access the information stored in the table, getValue or getClientReservation are used. These methods return the value of one piece of information or the entire client’s entry, respectively.

```
ReservationTable
getValue
setValue
getClientReservation
getClientList
getKeys
clientExists
keyExists
removeClient
removeKey
```

Figure 4.9a: Reservation Table class diagram
Figure 4.9b: Structure of the Reservation Table

39
The importance of the Reservation Table goes beyond the storage of information. It is the link between the Registration and Reservation subsystems. Only after the registration protocol is done, and an entry for the client is in the table, the reservation protocol is able to proceed successfully. The Registration Server writes to the Reservation Table and the Reservation Server reads from it.

4.7 Registration System Components

4.7.1 Registration Protocol

The registration protocol provides communication between the Registration Server and the Registration Client. The core of the protocol connects a Registration Server servicing a location and one or more Registration Clients present in the location. An instance of the RegistrationServer starts by registering with the lookup server. It then waits for connections from instances of RegistrationClient, which finds the Registration Server by querying the lookup server. When a connection is made at the server, a new RegistrationConnection thread is created to handle the client. The communication is exclusive between a reservation client and an instance of RegistrationConnection.

The protocol is divided into three parts: the initiation phase, the synchronization phase and the registration phase. Figure 4.10 details the initiation and synchronization phase.
Figure 4.10: Initiation and synchronization phase of the registration protocol

The protocol is client-initiated by sending a Request instance with type START along with the client identification string. The identification string is cached to create an entry in the Reservation Table during the registration phase. The initiation phase ends with an acknowledgement from the server. After a successful initiation, the client’s communication is continued with the synchronization phase of the protocol. It updates the set of plug-ins necessary at the client-side. The client sends a request with type PLUGIN_LIST and the name of the operating system that the client host machine runs. From this, the server fills a list with the class names of the plug-ins that the client is required to run. It sends the list in a result of type PLUGIN_LIST. The client then filters this list by removing the plug-ins already installed and continues by sending a request for each missing class file. This is done through a request of type PLUGIN_FILE, which piggybacks the requested filename. The server reads the class file into a PluginFile instance along with any native library required. This is sent back in a result of type PLUGIN_FILE. After all plug-ins are installed at the client-side, the synchronization phase ends.
Now that the client has all the plug-ins it needs to run, it proceeds with the registration phase. During the registration phase, the server makes an entry for the client using the cached client identification string. This entry contains all the attributed gathered by the plug-ins. Figure 4.11 illustrates the registration phase. The client runs all the installed plug-ins and places the information in a Request instance with type REGISTER. The argument list of the request contains the value returned from the plug-in mapped to the display name plug-in. The entry is made in the Reservation Table with the values received in the registration request. After a successful registration, the server acknowledges with a result of type REGISTER. The client then finalizes the protocol by sending a request of type END.

![Figure 4.11: Registration phase of the registration protocol](image)

### 4.7.2 Plug-in Framework Implementation

The Plug-in architecture is what provides flexibility to the system. It is composed of the following classes: Plugin, PluginRepository, Plugable, PluginFile, PluginFilter, and
PluginLoader. Figure 4.12 shows the plug-in architecture related classes and the relationship between them.

This architecture is what allows the Registration subsystem to exchange pieces of code, the plug-ins, to extend the profiling capabilities of the client. These plug-ins are Java class files that are loaded at runtime from the plug-in directory. When loaded, an instance is created and it is executed to retrieve some information needed for registration. For example, individual plug-ins can be implemented to determine the amount of memory and the encoders available at the client.

The super class of a plug-in is the Plugin class. It is a template class that declares the essential methods that any plug-in class needs to implement. The main idea is that each plug-in is used to retrieve some information from the client’s machine. This is done with the getData method. Every newly implemented plug-in needs to override this method to carry the functionality used to profile some attribute from the client. Since there are things that cannot be
done purely in Java, the Plugin class also allows the usage of native code through JNI\(^3\). The programmer of a plug-in can specify the name of the helper library, which is also loaded and used by the plug-in. By implementing new plug-ins, the behavior of the client is altered in terms of the data that it can register.

A plug-in follows a strict naming convention in order to identify the system it is aimed for and what it does. Their names are in the form `<OS><NAME>Plugin`. The prefix `<OS>` is the name of the operation system\(^4\) that it is built to support. It can also be set to SHARED, which denotes that the plug-in is capable of being used in any client. The sub-prefix `<NAME>` identifies the data that the plug-in is built to retrieve. This naming convention facilitates the identification of the plug-in. When selecting the plug-ins needed by a client, the name of the client’s operating system is used to filter the necessary plug-ins for that specific client. The PluginFilter class implements this functionality. It implements Java’s FilenameFilter interface, which is used by an instance of a File class to filter filenames by certain criteria.

Another important part of the plug-in architecture is the delivery of the plug-in code through the network. The necessary code is packaged in an instance of a PluginFile. This class reads the content of the plug-in class file and buffers it inside the object. It identifies if the plug-in uses native library and also buffers them. It is a Serializable object, meaning that it can be sent through the network. It has capabilities of reading and writing a file to the file system in a platform independent manner.

Since the plug-in classes are located in a directory that is not part of the CLASSPATH, they are not loaded with the default class loader of the Java Virtual Machine. The PluginLoader

---

\(^3\) JNI is Java Native Interface

\(^4\) This version of the system only supports WINDOWS, WINCE, and SHARED as values of `<OS>`
is a custom class loader used exclusively to load the plug-in classes into the system so that they can be instantiated. Any class file not residing in any of the directories specified by the CLASSPATH are considered by Java as un-trusted. The custom loader is necessary to allow the user to decide where the plug-ins are stored and through a configuration property, the loader can find and load the class files.

The final but essential components of the plug-in architecture are the *Plugable* and *PluginRepository* interfaces. An agent that houses the plug-in class files implements the *PluginRepository* interface. A repository has methods to access the file system to read and write class file with the help of the *PluginFile* class. It can also list the available plug-ins in the repository location. An agent whose behavior is extended by adding plug-ins implements the *Plugable* interface. It provides the necessary methods that are used to add, remove, and access the plug-ins present in the agent as well as loading the class file to create an instance. A *Plugable* object is also a *PluginRepository* because it ends up holding plug-ins. In this system the *RegistrationServer* is a *PluginRepository* and the *RegistrationClient* is *Plugable*.

This framework facilitates the functionality of the registration client to be extended. The new behavior is obtained by synchronizing with the registration server. Depending on the plug-ins available at the server, the set of registered attributed in the client changes. These are reflected on the information that is ultimately registered about the client.

### 4.8 Reservation System Components

#### 4.8.1 Reservation Protocol
The reservation protocol is the counterpart of the Registration subsystem. It depends on a successful registration creating an entry for the client in the Reservation Table. The protocol provides communication between the Reservation Server and the Reservation Client. The protocol is divided into the initiation phase, the admission phase, and the refresh phase. An instance of ReservationClient connects to the Reservation Server, which is assumed to lie in the same machine as the Registration Server. Before the Registration Client hands off to the Reservation Client, it writes into the configuration file the name of the server’s host machine. If the connection is accepted, an instance of ReservationConnection exclusively communicates with the reservation client. The ReservationConnection class implements the server-side of the protocol. The ReservationServer keeps track of the available resources through two tables. One table has the total resources of the server machine and the other maintains the amount of allocated resources.

The initiation phase of the reservation protocol is similar to the initiation phase of the registration protocol. The client sends a request of type START along with the client...
identification string. The server checks if there is an entry in the Reservation Table for the client. If there is one, the server caches the entry and sends an acknowledgement with a result of type START. Figure 4.13 illustrates the initiation and admission phase.

The protocol now enters the admission phase. In this phase, the message exchange forms an atomic transaction. Figure 4.14 demonstrates the need for an atomic transaction.

![Atomic Transaction](image)

Since the server allows concurrent client connections, it needs to maintain the consistency of its resource table by allowing only one client in this phase at a time. This is done through Java’s synchronization model based on monitors. First the client sends a request of type ADMISSION. At the server, admission control is done. A lock is acquired for the resource table and then the admission test. No other client can access the admission phase after this; they need to wait until the lock is released. During admission control, the reservation server calculated the necessary resources to service the client. Admission is successful if the necessary resources plus the allocated resources are less than or equal to the total resources. If the admission passes, an instance of *Contract* is created for the client. The contract contains the interval time for refreshing the reservation. This is sent to the client in a result of type CONTRACT that piggybacks a copy of the contract that the client needs to agree with. The client “signs” the contract in acceptance with the refresh interval time and sends it in a request of type SERVICE. The servers checks that the contract is accepted and if so, the desirable resources are allocated.
The resources are allocated by adding the necessary resources to the allocated resources table. At this point the admission phase is over and the lock for the resource table is released, allowing other clients to go through the admission test.

Once a reservation is accepted and allocated, the protocol enters the refresh phase. Figure 4.15 shows the refresh phase.

![Refresh phase of the reservation protocol](image)

During this phase the requested service is also being delivered. The client needs to send Request objects of type REFRESH in order to keep the reservation alive. The time interval for each request is specified in the Contract object “signed” by the Reservation Client. Like RSVP, the resources allocated by the server are only valid for a period of time. A failure to REFRESH the reservation before the time expires de-allocates the resources of the now invalid reservation. This is an instance of a soft state reservation. The mechanism for detecting the validity of the reservation in the Reservation Server is implemented using an instance of class Timer. The server’s timer has an amount of time until it expires. At the client side, another instance of Timer is used to send a REFRESH request at a regular interval. When the Reservation Server receives the REFRESH request, its timer is reset. It is only when the server’s timer expires, that the
reservation is canceled and the resources are freed. The resources are de-allocated by subtracting
the necessary resources used to service the client from the allocated resources table.

The protocol reaches its closing stage under several conditions. At the initiation phase the
connection is closed if there is no entry for the client in the reservation table. At the admission
phase, the client can fail the admission test and the reservation request is then denied. If the
client does not “sign” the contract, the server interprets it also as a failure to admit. During the
refresh phase, the client might fail to send a refresh request in time or the service stops. In both
cases, the resources are de-allocated and the protocol ends.

4.8.2 Strategy Framework Implementation

The ReservationServer class uses the Strategy pattern to externalize some of its
operations. This is done to provide the ability to customize the Reservation Server’s behavior by
strapping new strategies. The strategy framework works similarly to the plug-in framework
except the class files of the strategies are not distributed through the network. They are assumed
to be stored locally in a directory specified in the configuration file. The framework is composed
of the classes ReservationStrategy, ContractStrategy, and StrategyLoader. Figure 4.16 depicts
the related class diagram.

The strategy pattern exists when a service supposedly done by the Reservation Server is
delegated to a different object that is interchangeable. Changing this object essentially changes
the strategy used by the server. In the ReservationServer class the methods initResourceTable,
doAdmissionControl, allocateReservation, deallocateReservation, and createService are
externalized and delegated to an instance of ReservationStrategy. The method createContract is
externalized and delegated to an instance of ContractStrategy.
Creating a custom strategy involves extending either ReservationStrategy or ContractStrategy. These custom strategies need to be loaded into the system and an instance created to substitute the default ones. As with the plug-ins, the strategies location is known through a property in the configuration file. This directory might not be in the CLASSPATH, which means that a custom class loader is needed. The StrategyLoader implements a custom class loader that knows where the strategy class files are located. When an instance of ReservationServer is created, the strategy loader searches the specified directory for valid custom strategies. The names of the strategy class files are also properties in the configuration file. If it is able to load them, an instance is created, and it is set as the current strategy for the reservation server. The Strategy Framework enables the Reservation System to have a standard communication protocol with a dynamic server implementation.
4.8.3 Service Provision

The reservation system attempts to generalize any service requested into the super class Service. This class provides a standard interface to start and stop a requested service. The idea is to create a wrapper object around any existing service. This way, new services can be added by simply wrapping it with a subclass of Service. For example, a Video on Demand application can be wrapped with a subclass called VODService. The implementer will override the start and stop methods as needed. What allows adding new services without changing the reservation server is that the instance of Service is created in the instance of ReservationStrategy, which is a strategy object. The method createService returns a Service object. From the client’s information entry, the service requested can be determined, so the appropriate subclass of Service can be instantiated and returned.

This chapter explained in details the inner workings of the system. It explained the components and protocols used for communication. It also provided the details of the plug-in and strategy framework used to upgrade and, most importantly, integrate the system. The next chapters will provide insights on how to take advantage of the programmability of the system and also some sample runs of the system.
CHAPTER 5

System Integration

The previous two chapters explained the design and implementation of the system. They emphasized the need of a customizable system that can be easily modified in order to integrate it into the application scenario. The two frameworks that allow customization are the plug-in architecture of the registration system and the strategy architecture of the reservation system. This chapter provides the necessary information to create classes for these two frameworks. It is important to point out that the development of plug-ins and custom strategies go hand in hand. Creating new plug-ins might require new custom strategies and vice versa. These frameworks are interdependent due to their relationship to the reservation table. The plug-ins determine the content of the table, whereas its content is used to make decisions in the custom strategies. This chapter introduces the programming interface and provides instructions on how to create the plug-ins and custom strategies.

5.1 Programming Interface

The tools available to the programmer to create the plug-ins and custom strategies are explained in this section. In general, the programmer will use Java API to create the additional
components, however, the system is flexible. For the implementation of plug-ins, the programmer has the option to use other programming languages. Additionally, the programmer has access to the classes used to implement the registration-reservation system.

5.1.1 Java API

Java is the programming environment used to implement and run the system; therefore, it is also used to implement the additional components of the system. As a result, the entire Java API is available for usage to the programmer. Implementation is restricted, however, to Java version 1.1 due to the fact that the Java Virtual Machine, which runs in the Windows CE system, is developed to support up to Java version 1.1 only. Additionally, the plug-in architecture supports the use of native code through Java Native Interface (JNI). JNI allows the programmer to access native methods as if they were regular Java methods, regardless of the programming language used. This is an advantage because a different programming language may be necessary when the functionality of the plug-in cannot be implemented in Java alone. For example, accessing the Windows Registry is done through system specific calls that cannot be accessed through Java.

5.1.2 System Packaging

The programmer also has access to the system’s package of classes. This means that the plug-ins and the custom strategies can use any class implemented to create the registration-reservation system. The registration-reservation system is divided into a set of packages that organize the classes into different groups according to their functionality. Figure 5.1 illustrates the package organization of the system.
The `ResourceReservation.Client` and `ResourceReservation.Server` packages contain the client-side and server-side entities, respectively. These classes implement the registration and reservation protocol. The `ResourceReservation.Protocol` package contains the classes used exclusively for both protocol. It includes the classes used for network communication and the classes for the objects exchanged during the protocol.

The `ResourceReservation.Configurable` package is used by the configurable entities in the system. The `ResourceReservation.Main` package contains the classes that have a `main` method used to run the different entities in the system. The rest of the packages are used internally by the system, including the exceptions and graphical user interface. This chapter will focus on the
ResourceReservation.Plugin and ResourceReservation.Strategy packages. These packages implement the super classes, which are extended to create plug-ins and custom strategy components.

5.2 Plug-in Implementation

New plug-in classes are implemented by sub-classing the Plugin class in the ResourceReservation.Plugin package. Refer to Appendix C for detailed documentation of the Plugin class. To implement a plug-in, the programmer needs to identify the target system, the data to be profiled, and assess the need for native code. The following section explains the process of naming the plug-in classes, as well as the code necessary for proper implementation of plug-in.

5.2.1 Plug-in Class Naming

The target system and the profiled data are used to construct the name of the plug-in class. The plug-in naming convention stipulates that the plug-in class name has to be of the form <OS><NAME>Plugin. In this version, the supported values for <OS> are WINCE, WINDOWS, and SHARED. SHARED is used when the plug-in is entirely implemented in Java without additional native code, and can be distributed to all clients. WINDOWS and WINCE are used when the plug-in is dependent on the client’s platform, whether it is Windows or Windows CE. The platform dependence may exist because of the use of native code that is built to run on a specific platform. The <NAME> sub-prefix is used for convenience. It serves as a simple
identifier of the plug-in function. For example, a plug-in that reads the amount of available RAM in a Windows machine would have a class name similar to “WINDOWSAvailableRAMPlugin”.

5.2.2 Adding the Plug-in Code

The next step is implementing the body of the plug-in. There are two cases: a pure Java plug-in and a plug-in that uses Java and native code. The former is the simpler case, as it does not require the help of any native code. Figure 5.2 presents the code for a plug-in class that does not require any native library.

```java
import ResourceReservationPlugin.*;

public class SHAREDTestPlugin extends Plugin{

    // CONSTRUCTOR
    public SHAREDTestPlugin(){
        super( "SHAREDTestPlugin.class", "TEST");
    }

    // PROTECTED DATA ACCESS METHOD
    public Object getData(){
        // CODE
        return object
    }
}
```

Figure 5.2: Plug-in code without native call

To begin, a default constructor is implemented. It must call the `Plugin` constructor with two `String` objects: the class name and the display name of the plug-in. The class name is used to access the class file from the file system. The display name is used as a key to a value in the reservation table. The display name is mapped to the value, which is returned by the execution of the plug-in during the registration phase. When the registration client executes the plug-in, a call is made to the `getData` method inherited from the `Plugin` class. The programmer implements this
method according to the functionality required to profile the specific information. This method returns a Java Object containing the profiled data.

A more complex case of implementing the body of a plug-in is when it requires the assistance of a native method call. This means that part of the functionality of the plug-in is implemented externally in a programming language other than Java. Figure 5.3 presents the code for a plug-in class that uses native calls through Java’s JNI.\(^5\)

```java
import ResourceReservationPlugin.*;
import java.io.*;

public class WINDOWSTotalRAMPlugin extends Plugin{

    //CONSTRUCTOR
    public WINDOWSTotalRAMPlugin()
        super("WINDOWSTotalRAMPlugin.class", "TOTAL_RAM");
    }

    //STATIC BLOCK
    static{
        libraryName = "WINDOWSTotalRAMPlugin.dll";
        needNativeLibrary = true;

        if( !isLibraryLoaded()){
            System.load(PluginPlugin.buildLibraryName( libraryName ));
            setLibraryLoaded( true);
        }
    }

    //NATIVE METHOD DECLARATION
    public native String nativeGetData(),

    //PROFILED DATA ACCESS METHOD
    public Object getData() throws PluginException{
        //CODE TO PROFILE THE DATA
        try{
            return nativeGetData();
        }
        catch( Exception e ){
            throw new PluginException( "native call failed" );
        }
    }

    //OVERWRITE IF USING NATIVE LIBRARY
    public String getLibraryName(){
        return WINDOWSTotalRAMPlugin.libraryName;
    }

    //OVERWRITE IF USING NATIVE LIBRARY
    public boolean needsNativeLibrary(){
        return WINDOWSTotalRAMPlugin.needNativeLibrary;
    }
}
```

Figure 5.3: Plug-in code with native call

\(^5\) JNI is outside the scope of this chapter. Consult Sun’s Website for a tutorial.
The constructor declaration is the same as the previous example. An essential difference is the static block. This block is only executed once when the class is first loaded into the Java Virtual Machine (JVM). Three important things occur within the static block. First, the attribute \textit{libraryName} is set with the name of the native library. In this example the library name is \textit{WINDOWSTotalRAMPlugin.dll}. This file contains the C++ helping code for the plug-in. Second, the attribute \textit{needNativeLibrary} is set to TRUE to alert the registration system that the plug-in uses a helping library and that it should be sent to the client along with the plug-in code. Third, the library is loaded into the JVM with the call \textit{System.load}.

The native code is accessed by the plug-in through a call to a method declared native. A native method has no body because its functionality is declared elsewhere in a programming language other than Java, which is supported by JNI. In this example the native method name is \textit{nativeGetData} and its body is declared in \textit{WINDOWSTotalRAMPlugin.dll}. This method is called from within the \textit{getData} method as if it were another object method, thus effectively delegating the responsibility of profiling to the native code.

Finally, the \textit{getLibraryName} and \textit{needsNativeLibrary} method are overridden to return the correct values. With these changes made, the plug-in is ready for compilation and distribution to the registration clients.

5.3 Custom Strategies Implementation

Custom strategies are built by sub-classing either the \textit{ReservationStrategy} class or the \textit{ContractStrategy} class. Refer to Appendix D for documentation on the two super classes. From a
programming point of view, it is simpler to implement new strategies than to implement new plug-ins. Difficulty arises in that implementing custom strategies requires knowledge regarding the usage of resources in the server. The programmer must decide which resources are going to be managed by the reservation system as well as the effect that every client or flow property has on each resource\textsuperscript{6}. There are three types of methods defined on the strategy’s super classes: resource table management, admission control, and service and contract delivery. The functionality of these methods depends on the available client information.

5.3.1 Resource Table Management

The Resource Table is composed of two individual tables, *Total Resources Table* and *Allocated Resources Table*. The *Total Resources Table* contains the total amount of resources the server machine has available for reservations. The *Allocated Resources Table* maintains the amount of resources currently in use for clients that have been admitted by the reservation server. Managing the Resource Table involves initializing the table, allocating resources to the table, and de-allocating resources from the table. These functionalities are defined by methods in the *ReservationStrategy* class.

When the reservation server starts, the Resource Table is initialized. The total amount of available resources are calculated and placed on the *Total Resources Table* and the *Allocated Resources Table* is set to zero. This process is achieved by a call to *initResourceTable* method. Figure 5.4 presents a sample implementation of the method. The example manages two fictitious resources, RESOURCE\_A and RESOURCE\_B, as well as the total number of connections the

\textsuperscript{6} The effect that the client’s and flow’s properties have on the resources are outside the scope of this thesis.
server can manage. Figure 5.5 illustrates the resulting Resource Table after initialization with the code shown in Figure 5.4.

```java
public void initResourceTable(Hashtable totalResources, Hashtable allocatedResources)
{
    super.initResourceTable( totalResources, allocatedResources );
    System.out.println( "TestReservationStrategy initializing..." );

    //Resource A
    totalResources.put("RESOURCE_A", new Integer(100));
    allocatedResources.put("RESOURCE_A", new Integer(0));

    //Resource B
    totalResources.put("RESOURCE_B", new Integer(100));
    allocatedResources.put("RESOURCE_B", new Integer(0));
}
```

Figure 5.4: Example implementation of initResourceTable

<table>
<thead>
<tr>
<th>Total Resources Table</th>
<th>Allocated Resources Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key</strong></td>
<td><strong>value</strong></td>
</tr>
<tr>
<td>RESOURCE_A</td>
<td>100</td>
</tr>
<tr>
<td>RESOURCE_B</td>
<td>100</td>
</tr>
<tr>
<td>CONNECTIONS</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>key</strong></td>
<td><strong>value</strong></td>
</tr>
<tr>
<td>RESOURCE_A</td>
<td>0</td>
</tr>
<tr>
<td>RESOURCE_B</td>
<td>0</td>
</tr>
<tr>
<td>CONNECTIONS</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.5: Total Resources Table and Allocated Resources Table after initiation

RESOURCE_A and RESOURCE_B are both 100% available and the server can service 5 concurrent connections. None of the resources are allocated for usage.

When a reservation is admitted or freed, the Allocated Resources Table must be updated. After a successful admission, the Allocated Resources Table is updated to reflect the increased use of resources. The update is done by a call to allocateReservation. Figure 5.6 presents a sample implementation of this method.
public void allocateReservation( Hashtable clientInfo, Hashtable allocatedResources )
{
    super.allocateReservation( clientInfo, allocatedResources );
    System.out.println( "TestReservationStrategy allocating. " );

    //Calculate Resource A and Resource B usage
    int neededA = 0, neededB = 0;
    Enumeration enum = clientInfo.keys();
    while ( enum.hasMoreElements() )
    {
        String propName = (String)enum.nextElement();
        neededA += calculateResourceA( propName, clientInfo.get(propName) );
        neededB += calculateResourceB( propName, clientInfo.get(propName) );
    }

    //Allocating resource A
    int allocatedA = ((Integer)allocatedResources.get("RESOURCE_A")).intValue();
    allocatedA += neededA;
    allocatedResources.put("RESOURCE_A", new Integer(allocatedA));

    //Allocating resource B
    int allocatedB = ((Integer)allocatedResources.get("RESOURCE_B")).intValue();
    allocatedB += neededB;
    allocatedResources.put("RESOURCE_B", new Integer(allocatedB));

    System.out.println( "Allocated: " + allocatedResources );
}

Figure 5.6: Example implementation of allocateReservation

The method calculates the amount of each resource required to service the client. This is calculated from the available client information in the Reservation Table. The required resources are then added to the Allocated Resources Table. If the client's reservation is voided after its required resources have been allocated, the Allocated Resource Table must be updated to free the resources. The freed resources are available for consumption by other clients. This process is achieved by a call to deallocateReservation. Figure 5.7 demonstrates an example implementation of this method. This method also calculates the allocated resources for the specific client. The allocated resources are subtracted them from the Allocated Resources Table, thus allowing the resources to be available for later use.
5.3.2 Admission Control

Admission Control is used to determine whether or not sufficient resources are available resources to service the client adequately. The Reservation Server carries out admission control by calling the method `doAdmissionControl`. The method is declared on the `ReservationStrategy` class. Figure 5.8 presents a sample implementation of this method. The method does not change the content of the Resource Table; rather it uses the content to determine the admissibility of the client. The client’s information is used to determine the resources necessary to service the client. The following rule is applied:

Allocated Resources + Necessary Resources <= Total Resources.

If the above statement is true for every resource, then the client is successfully admitted. If the statement is not true, the client is denied admission.

```java
public void deallocateReservation(Hashtable clientInfo, Hashtable allocatedResources )
{
    super.deallocateReservation( clientInfo, allocatedResources );
    System.out.println("TestReservationStrategy deallocating...");

    //Calculate Resource A and Resource B usage
    int neededA = 0, neededB = 0;
    Enumeration enum = clientInfo.keys();
    while ( enum.hasMoreElements() )
    {
        String propName = (String)enum.nextElement();
        neededA += calculateResourceA( propName, clientInfo.get(propName) );
        neededB += calculateResourceB( propName, clientInfo.get(propName) );
    }

    //allocating resource A
    int allocatedA = ((Integer)allocatedResources.get("RESOURCE_A")).intValue();
    allocatedA -= neededA;
    allocatedResources.put("RESOURCE_A", new Integer(allocatedA));

    //allocating resource B
    int allocatedB = ((Integer)allocatedResources.get("RESOURCE_B")).intValue();
    allocatedB -= neededB;
    allocatedResources.put("RESOURCE_B", new Integer(allocatedB));

    System.out.println("Allocated: "+allocatedResources);
}
```

Figure 5.7: Example implementation of `deallocateReservation`
public boolean doAdmissionControl( HashMap clientInfo, HashMap allocatedResources, HashMap totalResources )
{
    boolean ret = super.doAdmissionControl( clientInfo, allocatedResources, totalResources );
    System.out.println( "TestReservationStrategy doing admission control." );

    System.out.println( "Allocated:");
    System.out.println( "Total: ");

    //return immediately when it fails
    if( ret )
        return false;

    //check Resource A and Resource B usage
    int neededA = 0, neededB = 0;
    Enumeration enuma = clientInfo.keys();
    while( enuma.hasMoreElements() )
    {
        String propName = (String)enuma.nextElement();
        neededA += calculateResourceA( propName, clientInfo.get( propName ) );
        neededB += calculateResourceB( propName, clientInfo.get( propName ) );
    }

    //check Resource A availability
    int allocatedA = ((Integer)allocatedResources.get( "RESOURCE_A" ));
    int totalA = ((Integer)totalResources.get( "RESOURCE_A" ));
    if( allocatedA+neededA > totalA )
    {
        System.out.println( "Not enough Resource A available!" );
        return false;
    }

    //check Resource B availability
    int allocatedB = ((Integer)allocatedResources.get( "RESOURCE_B" ));
    int totalB = ((Integer)totalResources.get( "RESOURCE_B" ));
    if( allocatedB+neededB > totalB )
    {
        System.out.println( "Not enough Resource B available!" );
        return false;
    }

    return true;
}

Figure 5.8: Example implementation of doAdmissionControl

5.3.3 Service and Contract Delivery

The client information, collected during the registration phase, is also used to determine the requested service and to create a custom contract for the client. To create an instance of a subclass of Service, the Reservation Server calls the createService method. This method is
defined in the ReservationStrategy class. For example, a plug-in can query the client regarding the service it is requesting. This information is then mapped in the Reservation Table to the key SERVICE_TYPE. The method looks for the value mapped to SERVICE_TYPE and returns the appropriate instance of a subclass of Service. For example, if the value mapped to SERVICE_TYPE is VOD, the appropriate subclass of Service would be VODService.

To create an instance of Contract, the server calls createContract, defined in the ContractStrategy class.

```java
public Contract createContract(Hashtable clientInfo)
{
    Object obj = clientInfo.get("CONNECTION_TYPE");
    if (obj == null)
        return new Contract();
    String contractType = (String)obj;
    Contract c;
    if (contractType.equals("WIRED"))
    {
        c = new Contract(5);
        System.out.println("WIRED client created "+c);
    }
    else if (contractType.equals("WIRELESS"))
    {
        c = new Contract(10);
        System.out.println("WIRELESS client created "+c);
    }
    else
    {
        c = new Contract();
        System.out.println("UNKNOWN client created "+c);
    }
    return c;
}
```

Figure 5.9: Example implementation of createContract

By calling this method, the server creates a contract based on the available client information. The programmer can implement any criteria to select the proper refresh interval time carried by the Contract instance. Figure 5.9 illustrates a sample implementation of this method. It creates a different instance of Contract depending on the value mapped to the key
CONNECTION_TYPE. If a client has a value of WIRED, it is given a contract with a refresh interval of 5 seconds. If the value is WIRELESS, then the refresh rate of the contract is 10 seconds.

5.4 Component Installation

In order for the system to recognize and use the additional components, the components must be appropriately installed into the system. The process is fairly similar for both plug-ins and custom strategies (see Appendix B for details on the system installation).

5.4.1 Plug-in Installation

Installation of the plug-ins simply requires making a copy of the class files and additional libraries into a directory in the machine running the registration server. The path to this directory is then entered into the reservation server’s configuration file as the value of ‘server.plugin.path’ property. The server will only distribute the plug-ins present in this directory.

5.4.2 Custom Strategy Installation

Custom strategy installation is similar to the plug-in installation. Class files are copied into a directory and the path of this directory is set as the value of the ‘server.strategy.path’ property. This property is defined in the reservation server’s configuration file. In addition to knowing the location of the class files, the reservation server must know the names of the custom strategy classes. The name of the strategy class that extends ReservationStrategy is set as value

---

7 The name of the registration server’s configuration file is defined in Figure 4.6
8 The name of the reservation server’s configuration file is defined in Figure 4.6
to the ‘server.strategy.resv.name’ property. The name of the strategy class that extends
\textit{ContractStrategy} is set as value to the ‘server.strategy.contract.name’ property. If the server fails
to load any of the custom strategies, it loads and uses the default implementation.

This chapter provided the knowledge required to extend the system and enhance
integration into its setting. The next chapter presents a series of experimental results from
running the system. A set of test plug-ins and custom strategies are implemented and used to
prove that the plug-in and strategy framework behave properly.
CHAPTER 6

Experimental Results

This chapter presents the results of experimental runs of the registration-reservation system. Experiments are aimed at testing the different features of the systems and to assess their behavior. A set of plug-ins and custom strategies were implemented to illustrate and examine the extension capabilities of the registration and reservation protocol. In addition, the clients were hosted on distinct machines to demonstrate the capability of supporting heterogeneous receivers.

6.1 Participating Devices

Development of the registration-reservation system considers two platforms: Windows PC and Windows CE. Servers in the registration-reservation system are hosted on Windows workstations, whereas, clients may be hosted on any of the two platforms.

<table>
<thead>
<tr>
<th>High Resource Capacity Machine</th>
<th>Low Resource Capacity Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>CPU</td>
<td>CPU</td>
</tr>
<tr>
<td>RAM</td>
<td>RAM</td>
</tr>
<tr>
<td>Operation System</td>
<td>Operation System</td>
</tr>
<tr>
<td>JVM</td>
<td>JVM</td>
</tr>
<tr>
<td>PC or Laptop</td>
<td>HP Jornada</td>
</tr>
<tr>
<td>PIII: 500–800 MHz</td>
<td>Hitachi SuperH SH-3</td>
</tr>
<tr>
<td>64 MB – 128 MB</td>
<td>16 MB</td>
</tr>
<tr>
<td>Windows NT,9X</td>
<td>Windows CE 3.01</td>
</tr>
<tr>
<td>JDK 1.1.8</td>
<td>Personal Java 1.0 for Windows CE</td>
</tr>
</tbody>
</table>

Figure 6.1: Characteristics of the two types of clients in the experiment
Hosting the client on distinct platforms models the heterogeneity of the receivers. Client heterogeneity is modeled by hosting the client on devices that have dissimilar resource capacity. Figure 6.1 depicts the characteristics of the two types of clients used during the experiment. A client hosted on a Windows PC or a Laptop represents a High Resource Capacity machine, and a client hosted on an HP Jornada, running Windows CE, represents a Low Resource Capacity machine.

Hosting the client on different platforms, permit analysis of the synchronization phase of the registration protocol. During synchronization, the proper plug-ins are delivered to the proper platform. Information gathered about the clients is then utilized by the custom strategies for, among other things, create custom contracts for each client and calculate the resources required for servicing each client.

6.2 Experimental Plug-ins and Custom Strategies

Experimental plug-ins were developed to test the extensibility of the registration system (see Figure 6.2). Having a set of plug-ins permits testing of the synchronization phase of the protocol as well as the ability to profile the clients and register the information in the Reservation Table. The custom strategies developed will test admission control, the correct allocation and deallocation of resources, and the creation of a custom contract.

6.2.1 Test Plug-ins

To test the correctness of the plug-in framework, a set of plug-ins was implemented. Figure 6.2 lists and describes the implemented plug-ins for the experiment. Together, the plug-
ins test the Registration Server’s ability of selecting the appropriate plug-ins, delivering them to the client based upon its operating system, and of creating entries in the Reservation Table with the information gathered by the plug-ins. Individually, the plug-ins test the features of the plug-in framework.

<table>
<thead>
<tr>
<th>Plug-in Name</th>
<th>Target Platform</th>
<th>Display Name</th>
<th>Profiled Data</th>
<th>Native Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOWSTestPlugin</td>
<td>Windows</td>
<td>WINDOWS_TEST</td>
<td>“SUCCESS”</td>
<td>none</td>
</tr>
<tr>
<td>SHAREDConnectionTypePlugin</td>
<td>All</td>
<td>CONNECTION_TYPE</td>
<td>Value of property ‘client.connection’</td>
<td>none</td>
</tr>
<tr>
<td>SHAREDServicePlugin</td>
<td>All</td>
<td>SERVICE_TYPE</td>
<td>Value of property ‘client.service’</td>
<td>none</td>
</tr>
<tr>
<td>WINCETestPlugin</td>
<td>Windows CE</td>
<td>WINCE_TYPE</td>
<td>“SUCCESS”</td>
<td>none</td>
</tr>
<tr>
<td>WINDOWSTotalRAMPlugin</td>
<td>Windows</td>
<td>TOTAL_RAM</td>
<td>“HELLO”</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 6.2: Plug-ins developed for the experiment

For example, WINDOWSTestPlugin and WINCETestPlugin test that the correct plug-in is delivered to the correct platform. WINDOWSTotalRAMPlugin tests the usage of a native library through JNI. The library code returns “HELLO”, but the mechanisms, used to get this String, are not trivial. To execute this plug-in successfully, the registration server must recognize that the plug-in uses a native library. This native library must be sent to the client along with the plug-in. At the client, the native code is loaded and executed from within the plug-in. SHAREDConnectionTypePlugin and SHAREDServicePlugin test the distribution of a plug-in aimed at all platform. These plug-ins should be delivered to all clients, regardless platform.

6.2.2 Test Custom Strategies

The test custom strategies were implemented with knowledge of the test plug-ins. These are TestReservationStrategy and TestContractStrategy. TestReservationStrategy extends ReservationStrategy and implements custom resource management and admission control
methods by managing two fictitious resources, RESOURCE_A and RESOURCE_B. TestContractStrategy extends ContractStrategy and overrides the contract creation mechanism.

To test resource management and admission control, TestReservationStrategy manages two fictitious resources, RESOURCE_A and RESOURCE_B. These resources are both initiated to be 100% available. RESOURCE_A is affected by the value mapped to the key CONNECTION_TYPE in the Reservation Table. The plug-in SHAREDConnectionTypePlugin gathers this information from the client’s configuration file. Figure 6.3 illustrates how RESOURCE_A is affected by the different values of CONNECTION_TYPE.

<table>
<thead>
<tr>
<th>Value</th>
<th>Resource Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRED</td>
<td>30%</td>
</tr>
<tr>
<td>WIRELESS</td>
<td>50%</td>
</tr>
<tr>
<td>else</td>
<td>25%</td>
</tr>
</tbody>
</table>

Figure 6.3: RESOURCE_A usage depending on values of CONNECTION_TYPE

RESOURCE_B is affected by the value mapped to SERVICE_TYPE in the Reservation Table. Similarly, the plug-ins SHAREDServicePlugin gathers this information from the client’s configuration file. Figure 6.4 illustrated the amount of RESOURCE_B required by different values of SERVICE_TYPE.

<table>
<thead>
<tr>
<th>Value</th>
<th>Resource Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOD</td>
<td>40%</td>
</tr>
<tr>
<td>VCONF</td>
<td>55%</td>
</tr>
<tr>
<td>else</td>
<td>30%</td>
</tr>
</tbody>
</table>

Figure 6.4: RESOURCE_B usage depending on values of SERVICE_TYPE

Along with these two fictitious resources, the parent strategy implementation treats the maximum number of concurrent connections as a resource. With these three resources and allocation rules presented in Figures 6.3 and 6.5, the resource management and admission control
mechanisms are tested. Only if there is enough of each resource available, can the client be admitted.

Additionally, the custom strategies test the creation of a contract based on the client’s information. TestContractStrategy implements a rule of contract creation based on the value mapped to CONNECTION_TYPE. Figure 6.5 presents the refresh intervals assigned depending on the value of CONNECTION_TYPE.

<table>
<thead>
<tr>
<th>Value</th>
<th>Refresh Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIRED</td>
<td>5 sec</td>
</tr>
<tr>
<td>WIRELESS</td>
<td>10 sec</td>
</tr>
<tr>
<td>else</td>
<td>default</td>
</tr>
</tbody>
</table>

Figure 6.5: Refresh interval based on values of CONNECTION_TYPE

During the experiment, it should be noticed that for every REFRESH request sent by a WIRELESS client, there are two REFRESH request from a WIRED client.

6.3 Experiments

The experiments are divided into sub-experiments that test the individual features of the registration-reservation system. Experiments are setup with the four entities, a registration-reservation server, a lookup server, and two clients. The two clients, CLIENT_A and JORNADA_A, are running on different platforms. CLIENT_A is running on a Windows PC and JORNADA_A is running on a Windows CE HP Jornada. The two servers run on separate Windows workstations.

6.3.1 Lookup Experiment
The first experiment is to test the lookup system. For this experiment, the lookup server is configured to manage domain DOMAIN_X. It runs on host 111.111.111.1\(^9\) and port 6666, where it waits for requests from either a client or a server. Then, the registration-reservation server is started. The registration and reservation servers run on the same machine but on different ports. The host machine for the servers is 111.111.111.2. The registration server runs on port 1234 and the reservation server runs on port 4321. When the registration server starts, it registers with the lookup as the server for location LOCATION_X. When the two clients start, they each query the lookup server and successfully get the address for the server of LOCATION_X. The query for the server of LOCATION_X yields the address 111.111.111.2:1234.

### 6.3.2 Synchronization and Registration Experiment

During this phase of the experiment, the required plug-in files are distributed to the clients so that registration can be accomplished. After all the plug-ins are distributed, each client registers its information. To test the synchronization phase CLIENT_A does not have any of the plug-ins stored locally while JORNADA_A only has a partial set. Figure 6.6 indicates the plug-ins present on each client before and after synchronization. Only the missing plug-ins were distributed by the registration server.

\(^9\) The IP addresses have been changed
## Figure 6.6: Client’s plug-ins before and after synchronization

<table>
<thead>
<tr>
<th>Server Plug-ins</th>
<th>CLIENT_A</th>
<th>JORNADA_A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before</td>
<td>after</td>
</tr>
<tr>
<td>WINDOWSTestPlugin</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SHAREDConnectionTypePlugin</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>SHAREDServicePlugin</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>WINCETestPlugin</td>
<td>no</td>
<td>--</td>
</tr>
<tr>
<td>WINDOWSTotalRAMPlugin(^{10})</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Having all the required plug-ins, each client executes the plug-ins and registers the information with the Reservation Server. Figure 6.7 illustrates tables with the information registered by each client.

### Figure 6.7: Information registered by CLIENT_A and JORNADA_A

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>WINDOWS_SUCCESS</td>
<td>TEST</td>
<td>WINCE_SUCCESS</td>
</tr>
<tr>
<td>SERVICE TYPE</td>
<td>VOD</td>
<td>SERVICE TYPE</td>
<td>VOD</td>
</tr>
<tr>
<td>CONNECTION TYPE</td>
<td>WIRED</td>
<td>CONNECTION TYPE</td>
<td>WIRELESS</td>
</tr>
<tr>
<td>TOTAL_RAM</td>
<td>HELLO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The information reaches the server, which makes an entry for each client in the Reservation Table. After the registration is made, the registration protocol ends. Figure 6.8 depicts a view of the Reservation Table after CLIENT_A and JORNADA_A are through with the registration phase.

---

\(^{10}\) This plug-in has a native library called WINDOWSTotalRAMPlugin.dll
6.3.3 Admission Control and Resource Management Experiment

After both clients are successfully registered, they enter the reservation phase of the system. During the experiment, CLIENT_A was run before JORNADA_A. This means that CLIENT_A enters the reservation phase first. It is the first client to request admission from the server meaning that all resources are available. The reservation server admits the client and allocates the resources. Figure 6.9 is a view of the resource table after CLIENT_A resources are allocated.

CLIENT_A needs 30% of RESOURCE_A because its value for CONNECTION_TYPE is WIRED (see figure 6.3). It also needs 40% of RESOURCE_B because its value for SERVICE_TYPE is VOD (see figure 6.4). The reservation server also allocated a connection to it from its maximum of 5 available.
After CLIENT_A is allocated, JORNADA_A enters admission control. It requires 50% of RESOURCE_A because its value of CONNECTION_TYPE is WIRELESS (see figure 6.3) and 40% of RESOURSE_B because its value of SERVICE_TYPE is VOD (see figure 6.4). Additionally, the server must be able to support the connection. From figure 6.9 it can be observed that the resources are available. JORNADA_A is successfully admitted and its required resources are also allocated. Figure 6.10 is a view of the resource table after allocating the resources of JORNADA_A.

After CLIENT_A is allocated, JORNADA_A enters admission control. It requires 50% of RESOURCE_A because its value of CONNECTION_TYPE is WIRELESS (see figure 6.3) and 40% of RESOURSE_B because its value of SERVICE_TYPE is VOD (see figure 6.4). Additionally, the server must be able to support the connection. From figure 6.9 it can be observed that the resources are available. JORNADA_A is successfully admitted and its required resources are also allocated. Figure 6.10 is a view of the resource table after allocating the resources of JORNADA_A.
Running an additional client, CLIENT_B, tests a failed admission. CLIENT_B is also a Windows client as CLIENT_A. Figure 6.11 details the information registered by CLIENT_B.

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>WINDOWS_SUCCESS</td>
</tr>
<tr>
<td>SERVICE_TYPE</td>
<td>VCONF</td>
</tr>
<tr>
<td>CONNECTION_TYPE</td>
<td>WIRED</td>
</tr>
<tr>
<td>TOTAL_RAM</td>
<td>HELLO</td>
</tr>
</tbody>
</table>

Figure 6.11: Information registered by CLIENT_B

CLIENT_B requires 30% of RESOURCE_A because its value for CONNECTION_TYPE is WIRED (see figure 6.3). It also needs 55% of RESOURCE_B because its value for SERVICE_TYPE is VCONF (see figure 6.4). From figure 6.10 it can be perceived that there is a connection available for CLIENT_B but there are not enough of either RESOURCE_A or RESOURCE_B. This yields a failed admission.

### 6.3.4 Contract Creation Experiment

A feature of the system that must be tested is the creation of a contract based on the client’s information. The test contract strategy creates a different contract for CLIENT_A and JORNADA_A based on their values of CONNECTION_TYPE. The strategy returns a contract with a refresh interval of 5 seconds for CLIENT_A and a contract with refresh interval of 10 seconds for JORNADA_B (see figure 6.5). During the refresh phase of the reservation protocol, CLIENT_A has to send twice as many refresh messages as JORNADA_A.

### 6.3.5 Timing Experiment
This experiment measures the total amount of time it takes to add a client into the system and to provide it with the multimedia service. The major time overhead is produced by the synchronization of the client with the server during the registration protocol. During the synchronization phase, the server and the client exchange the files that implement the plug-ins. This file transfer is the most expensive part of the entire system. To test how much the synchronization phase affects the system, we ran four different tests. Figure 6.12 provides the details of each test.

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Client</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_1</td>
<td>CLIENT_A</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>TEST_2</td>
<td>CLIENT_A</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>TEST_3</td>
<td>JORNADA_A</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>TEST_4</td>
<td>JORNADA_A</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.12: Timing experiment parameters

The tests utilize clients CLIENT_A and JORNADA_A (see Figure 6.7). Each client is tested with all the plug-ins available and without any of the plug-ins available. If the client has all the plug-ins required, there is no need to transfer any files and the overhead of synchronization is removed. Figure 6.13 provides the results of these experiments.

Figure 6.13: Timing of the entire system
This experiment shows that registration of devices in the ubiquitous environment is expensive, especially if new devices are handheld devices, communicating via wireless network with the registration server.

The experiments proved the correct behavior of all features of the developed system. The implemented test plug-ins and custom strategies proved that the system could be easily customized. The system provides the desired flexibility proposed during the design phase.
CHAPTER 7

Conclusions and Future Works

This chapter discusses the accomplishments of the developed registration and reservation protocol. Additionally, the chapter presents ideas for future work and the direction of future efforts to increase the functionality of the developed system.

7.1 Conclusions

Throughout the development of the registration-reservation system, especial attention was granted to the creation of a customizable and flexible system. The communication interface between the client and the server is static but the internal mechanism can be customized. The registration system can be extended by the implementation of plug-ins. These plug-ins are exchanged between the server and the client during communication and executed at the client to profile its host machine. The information gathered during registration is used by the reservation system to manage the server’s resources. The resource management mechanisms are customizable through implementing custom strategies. The developed system was successful in implementing the plug-in and custom strategy framework, which adds the programmability feature to the system. This design allows a system administrator to create a set of plug-ins and
custom strategies to integrate the registration-reservation system to its domain. In the event of the network admitting new end systems or new services, the system can be upgraded by implementing the necessary plug-ins and make changes to the custom strategies.

7.2 Future Work

Future work in network applications must concentrate on developing systems that easily evolve along with changes to its environment. This includes addition of new types of end systems and new services. For this specific project, future efforts should concentrate in determining the proper set of plug-ins to effectively profile the end systems and decide on the best strategies for managing the network resources. In addition, improvements must be applied to increase the performance of the developed system. The next sections specify advised improvements to the registration and reservation system, individually.

7.2.1 Registration System

Performance is the main area of improvement for the registration system. During the experiments, it could be seen that the synchronization phase is expensive in terms of bandwidth and time. This is especially true for end systems connected to the network through low capacity mediums, for example wireless. To improve the delivery of plug-ins to the client, the server can be improved by determining the link capacity and decide to compress the plug-in files before distribution. Plug-in class files are usually small files except when they require the help of a native library. The native libraries are larger compared to plug-in class files and this may justify
the use of compression. It is important to study the effectiveness of compression since compressing a file demands a higher utilization of resources at the server.

Another area of improvement is the addition of versioning to the plug-in files. The current registration system is not able to detect if the plug-ins at the clients are outdated and must be exchanged by newer versions contained at the server. The mechanism to exchange plug-ins is done by name, so a plug-in which implementation has changed but its name remained the same, is not going to be successfully distributed. By adding a versioning mechanism, the administrator can change the implementation of a plug-in and change its version number to effectively update the plug-ins at the client.

Finally, the plug-in architecture does not provide the notion of security. The clients remain vulnerable to whatever functionality the plug-in implements, even malignant implementations. By creating a sandbox, similar to Java’s Applet framework, the system can limit the functionality the plug-ins, effectively protecting the clients. Another approach is to create an authentication process to identify the registration server as a trusted source of plug-ins. The client can be certain that any trusted registration server will behave in the correct manner.

7.2.2 Reservation System

The developed system concentrated in providing a flexible mechanism of implementing custom resource management strategies during reservation. Determining the mechanisms to calculate the required resources and efficient resource management was outside the scope of this work. Future efforts to improve the reservation system must be targeted to determination the correct set of resources that must be managed by the server and implement ways of how to calculate the necessary resources from the information gathered from the client during
registration. Indeed, this is the process that the administrator must go through in order to implement the correct custom strategies and integrate the registration-reservation system to its setting.

Another area of improvement for the reservation protocol is the addition of renegotiation mechanisms. The currently implemented system terminates communication with the client when the resources are not available and the client fails admission control. Future generations of the reservation system must add a methodology for allowing the server and the client to renegotiate the reservation request and prevent an immediate denial of service. This functionality should be implemented into the Contract abstraction. The server and client can renegotiate the reservation request by changing parameters within the contract until both entities agree.

Finally, the service provision part of the reservation system needs improvement. The framework for decoupling the different types of services and the reservation system is in place but future work should explore the functionality in more detail. This decoupling is crucial to allow the system to be independent of the services provides and still be easily upgraded when new services appear.

In conclusion, the implemented registration-reservation system was successful in exploring the implementation of customizable protocols. These protocols have a static interface but a customizable implementation allowing them to evolve along with their environments.
REFERENCES


## APPENDIX A

### Configurable Properties

<table>
<thead>
<tr>
<th>property name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lookup.port</td>
<td>lookup server’s listening port</td>
</tr>
<tr>
<td>lookup.maxConnCount</td>
<td>maximum number of concurrent connections</td>
</tr>
<tr>
<td>lookup.domain</td>
<td>name of the domain of the lookup server</td>
</tr>
</tbody>
</table>

Figure A.1: *LookupServer* properties

<table>
<thead>
<tr>
<th>property name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.reg.port</td>
<td>server’s listening port</td>
</tr>
<tr>
<td>lookup.host</td>
<td>lookup server’s host machine name</td>
</tr>
<tr>
<td>lookup.port</td>
<td>lookup server’s listening port</td>
</tr>
<tr>
<td>server.plugin.path</td>
<td>path to the folder that contains the Plug-ins</td>
</tr>
<tr>
<td>server.location</td>
<td>name of the location serviced by the server</td>
</tr>
<tr>
<td>server.maxConnCount</td>
<td>maximum number of concurrent connections</td>
</tr>
</tbody>
</table>

Figure A.2: *RegistrationServer* properties

<table>
<thead>
<tr>
<th>property name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.reg.port</td>
<td>registration server’s listening port</td>
</tr>
<tr>
<td>server.reg.host</td>
<td>registration server’s host machine name</td>
</tr>
<tr>
<td>lookup.host</td>
<td>lookup server’s host machine name</td>
</tr>
<tr>
<td>lookup.port</td>
<td>lookup server’s listening port</td>
</tr>
<tr>
<td>client.plugin.path</td>
<td>path to the folder that contains the Plug-ins</td>
</tr>
<tr>
<td>client.location</td>
<td>name of the location where the client resides</td>
</tr>
<tr>
<td>client.id</td>
<td>identification string for the client</td>
</tr>
</tbody>
</table>

Figure A.3: *RegistrationClient* properties
<table>
<thead>
<tr>
<th>property name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>server.resv.port</td>
<td>reservation server’s listening port</td>
</tr>
<tr>
<td>server.maxConnCount</td>
<td>maximum number of concurrent connections</td>
</tr>
<tr>
<td>server.strategy.path</td>
<td>path to the folder that contains the Strategies</td>
</tr>
<tr>
<td>server.strategy.contract.name</td>
<td>name of the custom Contract Strategy</td>
</tr>
<tr>
<td>server.strategy.resv.name</td>
<td>name of the custom Reservation Strategy</td>
</tr>
</tbody>
</table>

Figure A.4: ReservationServer properties

<table>
<thead>
<tr>
<th>property name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>client.id</td>
<td>identification string for the client</td>
</tr>
<tr>
<td>server.resv.port</td>
<td>reservation server’s listening port</td>
</tr>
<tr>
<td>server.resv.host</td>
<td>reservation server’s host machine name</td>
</tr>
</tbody>
</table>

Figure A.5: ReservationClient properties
APPENDIX B

System Installation

The registration and reservation system is divided into three entities: the lookup server, the registration-reservation server, and the registration-reservation client. This document explains the installation procedure for each entity. It will first explain the common requirements between the different entities followed by the installation instructions for each entity individually.

B.1 Common Requirements

Before installing any of the entities of the system, it is important that the correct version of Java is present on the machine. For the Windows platform, the Java JDK 1.1.8 or higher must be installed. For the Windows CE platform, the Personal Java 1.0 for Windows CE has to be installed. Additionally, the Swing Foundation Classes 1.1.1 must be installed on both platforms\textsuperscript{11}. For the development of plug-ins, it is important that they are compatible with Java’s JDK 1.1 for them to run on the Windows CE platform.

\textsuperscript{11} Java 1.2 for Windows already include the Swing Foundation Classes
Script files are provided to run each individual entity. This script files need to be edited to include the correct path to the JDK and the system’s classes. Additionally, the script should run from the root directory of the entity. This is the directory that contains all the necessary directories for the entity. An important directory that has to be present is “config”. Inside this directory is the specific configuration file for the entity. See Figure 4.4 for the names of these configuration files. The next sections will discuss more in detail the necessary changes to this file. They will also discussed the other necessary directories and files that each entity must have. The “demos” directory is provided and it includes examples of the different entities, which gives insight of how they should be set up.

B.2 Lookup Server Installation

The lookup server is installed in a well-known machine and port. The address of this machine is used for the installation of the client and the server. The default port is 6666 but it can be changed through the configuration file (see Appendix A). Additionally, the configuration file should be edited to provide the name of the domain managed by the lookup. The system’s administrator must distribute this information for the installation of the clients and the servers.

B.3 Server Installation

Aside from the necessary “config” directory, the server has two other important directories, the plug-in class files location and the custom strategies location. These directories are set in the configuration file and the paths are relative to the root directory of the entity. If custom strategies are implemented, their class names must also be set in the configuration file.
See Appendix A for the relevant property names and descriptions. When the server is run, an initializing GUI allows the user to specify the domain where the server is running as well as the location that it manages. If the domain is not in the list, the user can add a domain by specifying the name and the address of the lookup server for the particular domain.

B.4 Client Installation

The client is installed similar to the server. The only difference is that the client does not use custom strategies. In the client’s case, the plug-in directory is where the plug-ins are going to be copied to once they are delivered by the server. The client also has an initializing GUI that allows the selection of domain and location.
APPENDIX C

Plugin Class Documentation

ResourceReservation.Plugin
Class Plugin
java.lang.Object
    +---ResourceReservation.Plugin.Plugin

This class is the superclass for all Plugins in the system. The basic function of a Plugin is to have a name for internal use and a display name for GUI purposes. The main function of this class is to get some kind of data and the method to access this is getData.

Version:
03/13/2001
Author:
Leugim Bustelo, University of Illinois Urbana-Champaign

Field Summary

| protected | data         | An Object with the data gathered by the Plugin |
| protecte  | dataClass    | The Class type of the data                     |
| protected | displayName |                                             |
### Constructor Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td><code>Plugin()</code></td>
</tr>
<tr>
<td></td>
<td>Default Constructor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>Plugin(java.lang.String name)</code></td>
</tr>
<tr>
<td></td>
<td>Constructor that takes the name of the Plugin.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>Plugin(java.lang.String name, java.lang.String displayName)</code></td>
</tr>
<tr>
<td></td>
<td>A Constructor that takes the name and displayName of the Plugin.</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td><code>java.lang.String buildLibraryName(java.lang.String libraryName)</code></td>
</tr>
<tr>
<td></td>
<td>Helper methods for plugins that need native libraries and are loaded with the PluginLoader.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td><code>java.lang.Object getData()</code></td>
</tr>
<tr>
<td></td>
<td>This method is used to run the Plugin and retrieve the data it was programmed to gather.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>java.lang.String getDisplayName()</code></td>
</tr>
<tr>
<td></td>
<td>This method returns the display name of the Plugin.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>java.lang.String getLibraryName()</code></td>
</tr>
<tr>
<td></td>
<td>This method returns the name of the library needed by the Plugin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>java.lang.String getName()</code></td>
</tr>
<tr>
<td></td>
<td>This method returns the name of the Plugin.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td><code>boolean isLibraryLoaded()</code></td>
</tr>
<tr>
<td></td>
<td>This method returns the display name of the Plugin.</td>
</tr>
</tbody>
</table>
This method tells if the needed library was already loaded.

boolean needsNativeLibrary()
This method tells if the class needs a library.

void setDisplayName(java.lang.String displayName)
This method sets the display name of the Plugin.

static void setLibraryLoaded(boolean libraryLoaded)
This method marks the library as loaded or not

void setName(java.lang.String name)
This method sets the name of the Plugin.

java.lang.String toString()
Overrides the toString of class Object.

<table>
<thead>
<tr>
<th>Methods inherited from class java.lang.Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait</td>
</tr>
</tbody>
</table>

### Field Detail

**name**
protected java.lang.String name
A String with the classname of the Plugin

**displayName**
protected java.lang.String displayName
A String with a display name of the Plugin

**libraryName**
protected static java.lang.String libraryName
A String with the name of an optional native library

**libraryLoaded**
protected static boolean libraryLoaded
A boolean that tells if the library has been loaded or not, default is false
needNativeLibrary
protected static boolean needNativeLibrary
    A boolean that tells if the class need a native library, default is false

data
protected java.lang.Object data
    An Object with the data gathered by the Plugin

dataClass
protected java.lang.Class dataClass
    The Class type of the data

Constructor Detail

Plugin
protected Plugin()
    Default Constructor

Plugin
public Plugin(java.lang.String name)
    Constructor that takes the name of the Plugin.
    Parameters:
    name - A String with the name of the Plugin

Plugin
protected Plugin(java.lang.String name,
                java.lang.String displayName)
    A Constructor that takes the name and displayName of the Plugin.
    Parameters:
    name - A String with the name of the Plugin
    displayName - A String with the display name of the Plugin

Method Detail
gedata
public abstract java.lang.Object getData()
    throws PluginException
    This method is used to run the Plugin and retrieve the data it was programmed to gather.
    Returns:
    An Object that has the data
**getName**

```java
public java.lang.String getName()
```

This method returns the name of the Plugin.

**Returns:**
A String with the name of the Plugin

---

**getDisplayName**

```java
public java.lang.String getDisplayName()
```

This method returns the display name of the Plugin.

**Returns:**
A String with the display name of the Plugin

---

**getLibraryName**

```java
public java.lang.String getLibraryName()
```

This method returns the name of the library needed by the Plugin

**Returns:**
A String with the name of the native library

---

**setName**

```java
public void setName(java.lang.String name)
```

This method sets the name of the Plugin.

**Parameters:**
name - A String with the name of the Plugin

---

**setDisplayName**

```java
public void setDisplayName(java.lang.String displayName)
```

This method sets the display name of the Plugin.

**Parameters:**
displayName - A String with the display name of the Plugin

---

**isLibraryLoaded**

```java
public static boolean isLibraryLoaded()
```

This method tells if the needed library was already loaded.

**Returns:**
A boolean that tells if the library was already loaded

---

**needsNativeLibrary**

```java
public boolean needsNativeLibrary()
```

This method tells if the class needs a library.
Returns:
A boolean that tells if a library is needed

**setLibraryLoaded**

```java
public static void setLibraryLoaded(boolean libraryLoaded)
```

This method marks the library as loaded or not

**Parameters:**
libraryLoaded - A boolean to set the library as been loaded or not

**buildLibraryName**

```java
public static java.lang.String buildLibraryName(java.lang.String libraryName)
```

Helper methods for plugins that need native libraries and are loaded with the PluginLoader. It returns the absolute name of the native library;

**Parameters:**
libraryName - A String with the name of the library

**Returns:**
A String with the absolute path of the native library

**toString**

```java
public java.lang.String toString()
```

Overrrides the toString of class Object.

**Overrides:**
toString in class java.lang.Object

**Returns:**
A String that represents the Plugin
APPENDIX D

Strategy Classes Documentation

D.1 ReservationStrategy Class Documentation

public class ReservationStrategy
extends java.lang.Object

This is the superclass of any strategy used do admission control. The ReservationServer will use this class or a subclass as the definition of how to do the admission control and allocation.

Version: 04/03/2001
Author: Leugim Bustelo, University of Illinois Urbana-Champaign

Constructor Summary

| ReservationStrategy() | Default Constructor |
Method Summary

**void allocateReservation** (java.util.Hashtable clientInfo, java.util.Hashtable allocatedResources)

This method is the strategy used to do resource allocation.

**Service createService** (java.util.Hashtable clientInfo)

This method is the strategy used to create a Service object.

**void deallocateReservation** (java.util.Hashtable clientInfo, java.util.Hashtable allocatedResources)

This method is the strategy used to do resource deallocation.

**boolean doAdmissionControl** (java.util.Hashtable clientInfo, java.util.Hashtable allocatedResources, java.util.Hashtable totalResources)

This method is the strategy used to do admission control.

**void initResourceTable** (java.util.Hashtable totalResources, java.util.Hashtable allocatedResources)

This method is the strategy used to initiate the table of the server with the total amount of resources.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

**ReservationStrategy**

public ReservationStrategy()

Default Constructor

Method Detail

**initResourceTable**

public void initResourceTable (java.util.Hashtable totalResources, java.util.Hashtable allocatedResources)

This method is the strategy used to initiate the table of the server with the total amount of resources. It also initiates the allocatedResources to all 0.

Parameters:

totalResources - A Hasetable with the total resources of the server
allocatedResources - A Hasetable with currently allocated resources
doAdmissionControl

```java
public boolean doAdmissionControl(java.util.Hashtable clientInfo,
                                  java.util.Hashtable allocatedResources,
                                  java.util.Hashtable totalResources)
```

This method is the strategy used to do admission control. This method implements a
default method of doing admission and to change, the new subclass strategy should
override this method.

**Parameters:**
- `clientInfo` - A Hashtable with the registration entry for the client
- `allocatedResources` - A Hashtable allocated resources of the server
- `totalResources` - A Hashtable with the total resources of the server

**Returns:**
A boolean with the result of the admission

allocateReservation

```java
public void allocateReservation(java.util.Hashtable clientInfo,
                                java.util.Hashtable allocatedResources)
```

This method is the strategy used to do resource allocation. This method implements a
default method of doing allocation and to change, the new subclass strategy should
override this method.

**Parameters:**
- `clientInfo` - A Hashtable with the registration entry for the client
- `allocatedResources` - A Hashtable allocated resources of the server

deallocateReservation

```java
public void deallocateReservation(java.util.Hashtable clientInfo,
                                  java.util.Hashtable allocatedResources)
```

This method is the strategy used to do resource deallocation. This method implements a
default method of doing deallocation and to change, the new subclass strategy should
override this method.

**Parameters:**
- `clientInfo` - A Hashtable with the registration entry for the client
- `allocatedResources` - A Hashtable allocated resources of the server

createService

```java
public Service createService(java.util.Hashtable clientInfo)
```

This method is the strategy used to create a Service object.

**Parameters:**
- `clientInfo` - A Hashtable with the registration entry for the client

**Returns:**
A Service object

98
D.2 ContractStrategy Class Documentation

public class ContractStrategy
extends java.lang.Object

This is the superclass of any strategy used to build a contract. The reason for this class is to provide a default mechanism to creating a Contract object. If the server administrator wishes to provide his own strategy, he needs to subclass this class and change the definition of create contract.

Version: 04/03/2001

Author: Leugim Bustelo, University of Illinois Urbana-Champaign

Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContractStrategy()</td>
<td>Default Constructor</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contract createContract(java.util.Hashtable clientInfo)</td>
<td>This method returns a default Contract object.</td>
</tr>
</tbody>
</table>
Constructors inherited from class java.lang.Object

class java.lang.Object
  clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ContractStrategy

public ContractStrategy()  
  Default Constructor

Method Detail

createContract

public Contract createContract(java.util.Hashtable clientInfo)  
  This method returns a default Contract object. To create contracts based on the information of the client, this method needs to be overridden in the subclass.

Parameters:
  clientInfo - A Hashtable with the registration entry for the client

Returns:
  A default Contract object