Adaptive Applications in Mobile Environments

Karen Henricksen

Supervisor: Dr Jadwiga Indulska

October 15, 1999

Abstract

Mobile computing involves the presence of object mobility within a computing environment. This mobility may take several forms. Computer mobility involves the movement of a computing device, such as a laptop or PDA, from one location to another. Application mobility refers to the migration of a user program between different computing platforms. Finally, user mobility refers to the movement of a computer user between different computers.

The presence of any of these types of mobility can introduce a number of problems. For instance, mobility of computers can have a dramatic impact on the network bandwidth and other resources that are available to users and applications, and can cause the availability of these resources can vary greatly over time. The mobility of users may require facilities for migration of user applications or means for transferring user configurations between computers. Finally, the mobility of applications requires that applications are capable of operating in possibly diverse operating environments, for instance, under different operating systems or using different system resources.

Application adaptation is a means by which applications can respond to a changing operating environment. This project considers the types of application adaptation that are possible, and the ways in which adaptation can be achieved. In particular, it focuses on the role of the distributed system infrastructure in supporting adaptability.

In order for adaptability of applications to be possible, there are two essential requirements. First, the requirements of users and applications must be known. That is, requirements relating to resources such as network bandwidth, memory, disk space, security and reliability must be available to the party responsible for adaptation decisions. These requirements are typically formulated as a Quality of Service (QoS) description. Secondly, mechanisms must exist for obtaining information about the current state of the environment. For example, it should be possible to determine the current availability of resources such as those listed earlier.

The first part of the project examined the means by which these two requirements can be met by considering a number of distributed system architectures. In addition, it aimed to evaluate the success of each approach in its support for adaptation.

A set of criteria for determining the level of support a distributed system provides for adaptability was formulated. These addressed three different areas. First, they considered the types of Quality of Service that could be supported. Next, they addressed support for types of adaptation mechanisms. Adaptation mechanisms were classified according to the level at which they occurred. Three levels that were considered were the network level, distributed system level and application level. Finally, the criteria addressed support for various types of heterogeneity within distributed systems.

The criteria were applied to eight distributed systems described in recent literature. The systems were found to provide a diverse range of solutions to the problem of supporting adaptability.

The remainder of the project focussed on incorporating adaptability into an example application, consisting of a Web browser and server.

The architecture of Grail, an extensible Web browser, was studied in detail, and approaches to incorporating support for adaptation were examined. Grail's support for extension, based on the use of plug-in modules, ensured that changes to the overall architecture of the browser were minimal.

The provision of a distributed system infrastructure supporting adaptation was also addressed. A resource monitor that supported adaptation by monitoring resources and triggering adaptation mechanisms via callbacks was described.

Implementations of the resource monitor and extensions to the browser were constructed using Python, a powerful scripting language suitable for rapid prototyping. Fnorb, a Python-based CORBA implementation developed at the DSTC, was used for communication between the browser and resource monitor.

The resource monitor was implemented to allow the user to simulate various environmental conditions. The browser was extended to incorporate a support for adaptation in general, as well for a number of particular adaptation policies. The policies focussed on providing dynamic adaptation to changes in network bandwidth and the user display type.

Acknowledgements

I would like to thank my supervisor, Dr Jadwiga Induska, for her guidance and insight throughout the project, and for her assistance in the writing of this report.

Additionally, I would like to thank my husband Matt, for his unfailing support during the year.

Contents

Figures

Chapter 1. Introduction

Mobile computing involves the presence of object mobility within a computing environment. This mobility may take several forms. Computer mobility involves the movement of a computing device, such as a laptop or PDA, from one location to another. Application mobility refers to the migration of a user program between different computing platforms. Finally, user mobility refers to the movement of a computer user between different computers.

The presence of any of these types of mobility introduces a number of problems. For instance, mobility of computers can have a dramatic impact on the network bandwidth and other resources that are available to users and applications, and availability of these resources can vary greatly over time. The mobility of users may require facilities for migration of user applications or means for transferring user configurations between computers. Finally, the mobility of applications requires that applications are capable of operating in possibly diverse operating environments, for instance, under different operating systems or using different system resources.

In general, mobility introduces the possibility of radical change in the conditions under which applications must operate. The ability of applications to respond to these changes is known as application adaptability.

This report considers various types of application adaptation and the means by which they can be achieved. In particular, it focuses on adaptability in the context of distributed systems.

Distributed systems are currently enjoying widespread interest, due in part to recent standardisation efforts which have produced accepted models for distributed computing and led to the development of standard platforms such as OMG-CORBA. Distributed system platforms offer a number of benefits that will ensure their continuing importance in the future. The platforms offer a layer of abstraction over the underlying network, greatly simplifying the task of programming distributed applications. Additionally, many platforms provide support for heterogeneity within distributed systems by concealing differences in hardware and software from application components.

In order for adaptability of applications to be possible, there are two essential requirements. First, the requirements of users and applications must be known. That is, requirements relating to resources such as network bandwidth, memory, disk space, security and reliability must be available to the party responsible for adaptation decisions. These requirements are typically formulated as a Quality of Service (QoS) description. Secondly, mechanisms must exist for obtaining information about the current state of the environment. For example, it should be possible to determine the current availability of resources such as those listed earlier.

The information about resource requirements and resource availability can together be used to detect failures to meet the application and user requirements, and to trigger adaptation mechanisms for coping with the changed conditions.

These requirements can be fulfilled in several different ways. One possibility is to satisfy both of the requirements within applications. This approach requires that applications be responsible for monitoring resource availability and tracking resource requirements. This leads to complex applications.

Another approach involves providing an infrastructure that supports one or both of the requirements. This allows functionality to be shared between different applications and distributed environments, and is less wasteful than the approach in which each application implements its own resource monitoring and tracking.

A number of distributed system architectures incorporating such an infrastructure have been described by recent literature. These have ranged from simple architectures that provide mechanisms for determining the current state of the environment, to complex architectures that are capable of performing certain types of adaptation on behalf of applications.

The first part of this report is concerned with evaluating these architectures and the level of support they provide for adaptability. Chapter 2 characterises adaptation mechanisms in detail, and introduces a set of criteria for evaluating a distributed system's level of support for adaptability. Chapter 3 introduces a number of distributed systems that have been proposed, and provides an evaluation of these systems using the criteria outlined in Chapter 2.

Chapter 4 considers the use of RM-ODP and OMG-CORBA frameworks in the construction of adaptable applications. It addresses the models for describing applications used in each of these frameworks and the impact of these models on the types of adaptation that can be supported.

The remainder of the report discusses a case study in application adaptation. It focuses on an adaptable Web application, consisting of a Web browser and HTTP server. An adaptable Web client based on Grail, a Python-based browser, is described, and mechanisms for adapting the browser to changing network bandwidth and user display type are implemented by way of demonstration. Additionally, the design of a HTTP server that supports adaptable clients is outlined.

Chapter 5 describes the application and its design, while Chapter 6 outlines the implementation. Finally, Chapter 7 provides an illustration of the forms of adaptation supported by the application using a number of concrete examples.

Chapter 8 provides a summary of the work presented in this report.

Chapter 2. Supporting adaptation

In the context of this report, adaptation can be viewed as a response to changes in an application's operating environment. The response typically aims to allow the application to make the best or most efficient use of available resources and to perform in the manner that will be most satisfactory to the user.

The adaptation mechanisms that are appropriate for an application will depend upon the nature of the application and the resources it requires, as well as on the preferences of the user.

Concrete examples of adaptation can be seen by considering a real-time video player that downloads video over a network from a remote server. A number of different types of adaptation mechanism may be appropriate for responding to changes in the availability of bandwidth over the network. For example, when bandwidth is scarce, a number of methods may be used to reduce the bandwidth requirements of the application. One approach is to compress video frames before transmission over the network and decompress them again at the receiving side. This idea is illustrated in Figure 2.1, which assumes that the compression is performed by filter processes at the sending and receiving sides. A second approach involves the transformation of video frames from the original representation to a more compact one, for instance, the transformation of colour frames to black-and-white. This is illustrated in Figure 2.2. A conversion filter is responsible for carrying out the transformation.

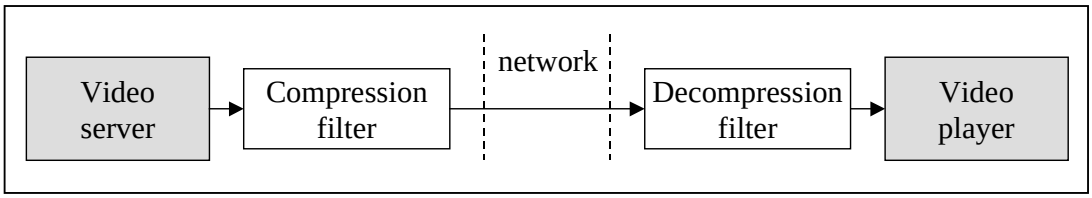

Figure 2.1. Adaptation to scarce bandwidth using compression

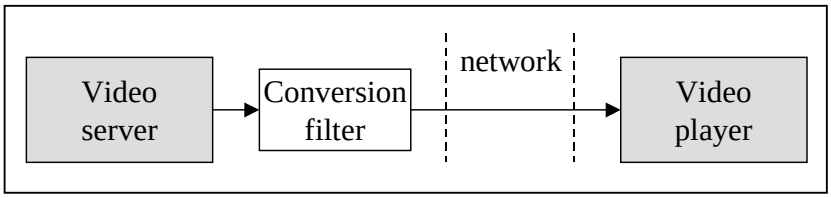

Figure 2.2. Adaptation to scarce bandwidth using a conversion filter

This chapter considers how these and other types of adaptation can be supported in a distributed system. Regardless of the approach that is used, there are two important requirements that must be satisfied in order to achieve adaptation. First, the resource requirements of the application and user must be known. For instance, for the video player application it is necessary to know the

bandwidth requirements of the application, as well as requirements for other resources, such as the memory used for buffering. The second requirement is the ability to determine the current state of resources used by the application. In the video player example it must be possible to determine the status of the communications network as well as resources that are used.

Provided that these two requirements are met, there are a number of models by which adaptation can be achieved. Several possibilities, differing primarily in the place at which adaptation is triggered and carried out, are outlined in the next part of this chapter.

2.1 Models for application adaptation

The adaptation model determines the location at which the triggering and execution of adaptation mechanisms occur. There are two different possibilities: within the application, or within an external infrastructure that supports the adaptation of applications.

The situation in which both initiation and execution of adaptation mechanisms occur externally to the application is known as application-transparent adaptation. In this case, an external party has access to the resource requirements of the application, and endeavours to ensure that the requirements are met. The resource requirements are typically formulated by the application as a Quality of Service (QoS) description, which describes the types of resource required and associated levels.

The external party carries out adaptation when required without the knowledge or input of the applications affected. For instance, adaptation to a shortage of bandwidth can be achieved in a manner that is transparent to applications by the re-routing of network connections to make better use of resources, or the insertion of compression or other filters to reduce the network load.

The scenario in which the application is involved in the execution of adaptation mechanisms is known as application-aware adaptation, regardless of whether adaptation is triggered within the application or without. In the former case, the application must supply its QoS requirements to the external party, as in application-transparent adaptation.

The types of adaptation that can be carried out when application-aware adaptation is used differ to those that are possible with application-transparent adaptation. If the QoS requirements of the application cannot be met, it is possible with application-aware adaptation to adapt the behaviour of the application to match the lowered resource availability. For instance, if the bandwidth available to an application drops, the application may respond by modifying the type of data it transmits in response.

Adaptation need not necessarily occur exclusively with an application or within an external party. Approaches in which the parties collaborate are also possible. For example, the external party may carry out adaptation in order to deliver the best possible QoS to the client, while in addition the client adapts to the changed QoS levels.

The schemes described in this section can offer differing levels of support for adaptation. For instance, the application-transparent approach can be more restrictive than an approach that involves the application, as it cannot alter an application's requirement for resources. Rather, it can only manipulate resources to ensure their efficient usage.

The next part of this chapter outlines a number of criteria for evaluating a particular distributed system's degree of support for adaptation.

2.2 Evaluating support for adaptation

As stated previously, knowledge of both the resource requirements of an application and the current state of resources are essential for adaptation. The first part of this section addresses the ability to describe and monitor resources within a system, and presents criteria that determine the types of resources that can be supported.

The middle part of this section looks at the particular adaptation mechanisms that can be supported by a system. In particular, it considers the places at which adaptation can be carried out, and the policies that determine the type of adaptation that occurs in given situations. Criteria are presented that address both the support for different classes of adaptation, and support for flexible adaptation policies.

The section concludes by considering various forms of heterogeneity that can be present in a distributed system. Criteria are outlined for determining the level of support for interaction between diverse types of hardware and software, evaluating the ability to employ a range of interaction patterns between distributed components, and evaluating support for different types of mobile object.

2.2.1 Support for QoS description

The criteria in this section address the range of resources that can be described and monitored by applications. The description of an application's resource requirements is referred to as a QoS description.

Many distributed systems allow negotiation of resource levels using QoS descriptions, or perform monitoring of resource levels on behalf of applications. These systems may provide differing levels of support for resources.

Some systems may support only one particular type of resource, such as network resources. Others may support a larger but fixed set of resources. Finally, systems may provide an extensible infrastructure for resource management, which allows new types of resources to be added as required. This last approach is clearly the most flexible.

Systems can also differ in the classes of resource that can be supported. Two classes of QoS are described by Manola [12], which he collectively refers to as IQoS.

The first class of IQoS includes the functional requirements of an application, that is, requirements that are directly related to the function or purpose of the application. These include properties such as bandwidth, main memory, and disk space. The second group includes the non-functional requirements, which are sometimes referred to as ilities. These are systemic properties, meaning that they are properties of a system as a whole, rather than an isolated component. They include properties such as reliability, availability, security, scalability, and so on.

The criteria for determining support for description of resources address both the number and types of resource that can be supported. That is, they look at whether a system can support a range of functional properties, and whether it can support different types of ilities. The criteria are summarised below.

- *Support for a broad definition of functional QoS*
- *Support for ilities*

2.2.2 Support for adaptability

The second set of criteria considers the mechanisms that perform adaptation. These mechanisms can be classified according to the place at which they occur. Three different types are identified by Crawley et al. [2].

Network adaptation is concerned with the allocation of communications resources, and occurs at the level of the communication network. Strategies falling within this class include re-routing of connections to make better use of resources, and pre-empting network resources in order to re-allocate them to higher-priority tasks.

Binding adaptation occurs at the level of the distributed system platform. It is concerned primarily with associations between distributed objects. Strategies in this class include replacing service objects requested by clients with others that will consume fewer resources, limiting the set of interactions permitted between objects, or deploying filters to reduce the bandwidth requirements across critical links.

The term ‘binding’ is particular to the ODP model, and therefore a more general name will be used for this class of adaptation in the context of this report. All forms of adaptation that occurs at the level of the distributed system platform will be grouped under the term **distributed platform-level adaptation**.

Application adaptation is adaptation that acts within a distributed application. Adaptation mechanisms that fall into this category are largely dependent on the type of the application. They

may include modification of the application behaviour to match resource availability, substituting alternative resources for those that become unavailable, and so on.

The first of the criteria for evaluating adaptability support considers the levels at which adaptation can occur. Greater flexibility in the types of adaptation that can occur will be achieved by allowing adaptation at multiple levels.

A second way to achieve flexibility of adaptation mechanisms it to allow flexible policies. Adaptation policies are the rules that determine the type of adaptation that will occur in particular circumstances. If policies are easily manipulated, the adaptation mechanisms that are employed can likewise be changed with little difficulty. To achieve this, policies should be separated from the functionality that implements them. The second criterion addresses this.

The criteria that assess support for adaptability are summarised below.

- *Support for adaptation at multiple levels*
- *Support for flexible adaptation policies*

2.2.3 Support for heterogeneity

This section addresses types of support for heterogeneity that can be provided by distributed systems. These are not essential for adaptation, however are highly desirable in a mobile computing environment.

Distributed systems frequently comprise a range of different hardware and software components. This is especially true in mobile computing systems, in which there are frequently both mobile and stationary components. These typically differ in nature due to different performance expectations. Mobile components are typically resource-poor and designed for portability rather than high performance. The opposite is usually true for stationary components. The difference in hardware also tends to lead to differences in the software used on each type of platform.

Ideally, a distributed system platform should support heterogeneity of system components. For instance, it should support the seamless exchange of data between machines with different internal data representations, communication between application components that use different programming languages, and so on. Current distributed system platforms such as CORBA already provide support for this form of heterogeneity.

It is also desirable in a mobile environment to support a range of interaction types for communication between distributed components. One of the most frequently supported types of communication pattern in distributed systems is the Remote Procedure Call (RPC). However, this type of communication is not necessarily suitable for all types of applications. For instance

synchronous communication is not appropriate for the real-time transmission of media such as audio and video. Moreover, synchronous types of communication, such as the RPC, have been criticised as unsuitable for mobile environments in which QoS can degrade to the point where messages are not delivered for long periods [3]. Finally, peer-to-peer models of communication can be overly restrictive for certain types of applications. Garcia et al. have motivated the need for multipeer communication for multimedia applications [4].

The special requirements introduced by mobility and particular applications make it desirable for a distributed system platform to provide flexibility in the types of communication pattern it supports. The second criterion addressing heterogeneity therefore considers the ability to support a heterogeneous set of interaction types.

The final criterion addresses the level of support for mobile objects. At least three different types of object mobility are possible within a distributed system. First, computers may be mobile, and may change their location within the communications network. Next, computer users may move within the system by moving between computers. Finally, applications can be migrated between machines in the distributed system.

The evaluation of support for mobile objects should consider not only the types of mobility that are supported, but also the degree of support for each type of object mobility.

A summary of the criteria determining support for heterogeneity is presented below.

- *Support for heterogeneous computing environments*
- *Support for heterogeneous interactions between application components*
- *Support for heterogeneous mobile objects*

2.3 Summary

This chapter began by defining adaptation and outlining some of the models that can be used to achieve it. Next, it described a set of criteria that can be used to determine the level of support a distributed system provides for adaptation. The criteria dealt with the ability of a system to support QoS, adaptability and heterogeneity; they are listed in Figure 2.1 for easy reference. The following chapter will apply the criteria to a set of distributed system architectures that have been described by recent literature.

Support for QoS description	• *Support for a broad definition of functional QoS* • *Support for ilities*
Support for adaptability	• *Support for adaptation at multiple levels* • *Support for flexible adaptation policies*
Support for heterogeneity	• *Support for heterogeneous computing environments* • *Support for heterogeneous interactions between application components* • *Support for heterogeneous mobile objects*

Figure 2.3. Summary of evaluation criteria

Chapter 3. A survey and evaluation of distributed system architectures

This chapter first introduces a number of distributed system architectures that have been described in recent literature, and then evaluates the architectures according to the criteria described in the previous chapter.

3.1 Architectures

3.1.1 Coda

Coda [5] is a file system that was created by researchers at Carnegie Mellon University. It is designed for use on workstations that are subject to frequent disconnections, and provides high availability of files using two mechanisms, namely server replication and disconnected operation.

Within Coda, files are grouped into units known as volumes, which may be replicated on a number of servers. The Volume Storage Group (VSG) for a volume is the group of servers that possess a replica of that volume. A subset, called the Available Volume Storage Group (AVSG), is the set of servers that are currently accessible. The replication strategy follows a "read-one, write-all" approach, meaning that data is obtained for a client from a single member of the AVSG, however, modified data is sent to all members.

During periods of disconnection, the client can continue to access cached data. Upon reconnection, modified data is propagated to the AVSG.

The support for adaptability provided by Coda is very limited, and is restricted to the mechanisms for coping during periods of disconnection that have been discussed.

3.1.2 Odyssey

The work on Coda led to a more general adaptive architecture known as Odyssey [6, 7]. Odyssey is a software platform that supports adaptation of mobile information-access applications, which are applications that run on mobile clients and access data on remote servers. It is realised as a set of extensions to the NetBSD operating system.

The main components of the Odyssey architecture are the viceroy and wardens. The viceroy implements a centralised resource management function, while the wardens are responsible for providing system-level support for specific types of resources.

Odyssey is integrated with the file system, so that Odyssey services are invoked as file system calls. Applications request resources through a *request* system call, indicating the type of

resource required and a window of tolerance, specified in terms of upper and lower bounds. A request is granted if the resource can be allocated within the window of tolerance. Subsequently, whenever the resource levels fall outside of the window's bounds, the application is notified using an upcall handler, provided by the application. The upcall allows the application to respond appropriately to changing resource availability.

While Odyssey provides much more general adaptability mechanisms than its predecessor does, it is designed with minimality, rather than flexibility, in mind. Because of this, the implementation of new types of Odyssey resources requires a significant effort. Conversely, an architecture known as Mobiware is much more highly engineered, and is specifically designed to support easy modification.

3.1.3 Mobiware

Mobiware, developed by the COMET group at Columbia University, is a middleware toolkit that supports the adaptation of applications to varying mobile network conditions [8]. It is based on CORBA and Java distributed object technology.

Mobiware focuses on bandwidth and the handover phase as the keys to providing mobile QoS. Thus, the architecture is centred on the underlying network infrastructure, providing abstractions of objects at different network layers, and mechanisms for manipulating them.

A programmable MAC allows new services to be created and installed on the fly. At the network level, hardware devices are abstracted as CORBA objects, and can be programmed through a set of network interfaces to support adaptive QoS assurances. The transport layer is built on a set of Java classes. End-to-end adaptation services are provided by binding active and static transport objects at mobile devices and access points. Static transport objects perform functions such as segmentation and reassemble, rate control, flow control, playout control and buffer management. Active transport objects are dynamically dispatched to support value-added QoS functions, such as filtering.

An application specifies its QoS requirements through a QoS API, in the form of a utility function and an adaptation policy. The utility function expresses the level of satisfaction of the application with different levels of bandwidth, while the adaptation policy determines how the application's bandwidth allocation will vary as resource availability changes.

A set of distributed CORBA network objects is responsible for delivering QoS to mobile hosts. Each mobile device has two associated proxy objects that support adaptation and handoff; these are the QoS Adaptation Proxy (QAP) and Routing Anchor Proxy (RAP), respectively. An abstraction of the mobile device is provided by the mobile device object, which provides APIs for obtaining beaconing information, registering with new access points, establishing flows, renegotiating QoS and handing off flows.

A mobility agent object performs a centralised management function, and is responsible for flow, adaptation and mobility management services. It achieves this through interactions with RAP and QAP proxies, using network topology information retrieved from router objects.

3.1.4 Jini

Unlike Mobiware, Jini [9] hides the details of the underlying network, and provides a simple, bare bones infrastructure. Its aim is to provide a flexible and easily administered environment, in which new services can be added and removed as required. Jini is based on the Java programming language, and, like Java, was developed by Sun Microsystems.

The architecture consists of three components:

- an infrastructure for federating services in a distributed system;
- a programming model for providing distributed services; and
- services that provide functionality for the benefit of users.

A lookup service forms the basis for the infrastructure, and provides a mechanism for locating services within a Jini system. This service matches interfaces that describe the functionality of a service with objects that implement that service. Services can be added to the lookup service on the fly, using a discovery and join protocol.

A service is requested by a client through the invocation of the appropriate method on the object returned by the lookup service. The object may contain code that executes the service locally, may make calls to remote objects using the Java Remote Method Invocation (RMI), or both.

The Jini architecture does not support adaptation in the sense that has been discussed, but rather provides a form of adaptability that permits the topology of the distributed system to be extremely flexible and dynamic.

3.1.5 An ODP-based adaptive management architecture

An architecture designed to support the special requirements present in defence networks is described by Crawley et al. [2]. Here, the importance of providing service to the most critical traffic necessitates the use of global policies for the allocation of resources.

The architecture is based on the ODP model, and realised using CORBA. This allows the architecture to cope with various forms of heterogeneity.

The architecture supports three kinds of adaptation, namely network adaptation, binding adaptation and application adaptation. The first two forms of adaptation are used if possible to

provide adaptation that is transparent to applications, and otherwise application-aware adaptation must occur.

At the centre of the architecture is the Adaptive QoS Manager (AQM), which manages the connections between applications and service objects (known as bindings), and controls resource usage throughout the system according to global policies. The AQM has access to the following forms of information, provided by a set of supporting services:

- management policies, which are maintained by the Policy Service
- interface specifications and QoS requirements of applications and services, which are the responsibility of the Type Manager
- network topology and QoS measures, which are available via the Network QoS Manager (NQM)

When clients require a service, they contact the adaptive QoS manager, specifying the interface type required, the service properties the server must have, and the QoS that is requested. The AQM, in conjunction with the Trader, selects an appropriate binding type and set of service objects, and chooses the most suitable object for minimising resource usage. To satisfy the request, existing bindings may be changed, communications paths may be altered, or filters may be deployed as necessary, according to the adaptation policies.

3.1.6 Nomadic computing architecture

The nomadic computing architecture [10] bears many resemblances to the previous one, however adds support for the mobility of users and applications within the system, as well of computers. Its goal is to provide users with access to services independent of their location, motion, computing platform, communication device and bandwidth.

The main component of the architecture is the Mobility Manager, which is responsible for managing adaptation, and plays a similar role to that of the AQM within the previous architecture. In addition, the Mobility Manager plays an active role in the mobility of objects.

The Mobility Manager uses policies specified in the form of relationships between objects, as well as user, computer and application profiles maintained by the Type Manager, to control mobility. The relationships specify rules that must be adhered to, for instance that certain applications must run on particular computers, and can only be used by particular people.

3.1.7 Context-Aware Computing Architecture

Unlike the architectures discussed so far, the context-aware computing architecture [11, 12] does not specifically target distributed applications, but rather provides mechanisms for any application to gather knowledge about its environment. It aims to support the adaptation of any

application to its context. Context-aware software is software that modifies its behaviour according to factors such as the location of use, the people nearby, the set of accessible hosts and devices, and so on.

The framework for supporting application adaptation is based around the dynamic environment server, which manages a set of environment variables, and delivers updates to clients who have subscribed to the server, as these variables change. There is typically one server for each context, where a context may be a room, work group, or other environment.

User agents work in conjunction with the collection of servers. These maintain environment variables pertaining to a user, and track the user's current location. Applications subscribing to the user's environment are automatically notified of changes, such as the owner's location.

The architecture has been realised using hand-held devices known as ParcTabs, which communicate with fixed base-stations using infrared signals. The devices function primarily as graphics terminals, with most computation being performed remotely on stationary hosts.

There are four types of adaptation to context that have been identified as relevant for applications in this system:

- *Proximate selection*. Under this scheme, nearby objects are emphasised over distant ones, and are made easier for the user to select.

- *Automatic contextual reconfiguration.* Components, such as connections to servers, loaded device drivers, and so on, are added, deleted or altered according to the context.

- *Contextual information and commands*. This scheme is based on the idea that queries and commands can be "parameterised" by the context, so that responses vary according to the situation.

- *Context-triggered actions*. Simple if-then rules are used to specify how the system should adapt; information about the context may then trigger commands according to these rules.

3.1.8 Limbo

The final architecture differs significantly to those already discussed due to its use of a radically different communication paradigm. The originators of Limbo suggest that connection-oriented, synchronous communication is unsuited for mobile contexts, for the reason that it is difficult to provide this type of communication in circumstances of extremely poor QoS [3]. Limbo is instead based on a tuple space paradigm. It provides a platform for mobile computing that is capable of supporting QoS monitoring and control by adaptive applications.

Under the tuple space model, processes communicate through a form of virtual shared memory known as a tuple space. Information is stored in data structures known as tuples, which consist of a collection of typed data fields.

The tuple space approach was popularised by the language Linda, which provides the following four basic operations:

- `out`, which inserts a tuple into the tuple space.
- `in`, which extracts a tuple from the tuple space. The argument to this operation provides a template for the tuple that is to be read.
- `rd`, which behaves identically to `in`, except that the tuple that is read remains in the tuple space following the operation.
- `eval`, which creates an active tuple that spawns processes to evaluate each tuple field, and subsequently places this tuple in the tuple space.

Limbo adds the following features to the basic model provided by Linda:

- Multiple tuple spaces, providing support for security and scalability
- A tuple type hierarchy, with support for sub-typing
- Tuples with explicit QoS attributes, such as deadlines, which dictate the lifetime of a tuple
- System agents supporting QoS monitoring, creation of tuple spaces, and propagation of tuples between tuple spaces. QoS monitors watch over aspects of the system such as connectivity, power availability and cost, and propagate the information gathered to clients through the tuple space.

The platform is implemented in a distributed fashion. A daemon process runs on each host, and, together, the daemons in the Limbo system manage the tuple spaces. A stub library is linked into each Limbo application process, providing the means for performing Limbo operations using the daemon process.

The tuple space approach has not been extensively used in distributed systems, and therefore its viability is still open to discussion. The communication overhead associated with maintaining the consistency of distributed tuple spaces may be large, and therefore the approach may prove infeasible in practice. Several optimisations to deal with this problem are discussed by Davies et al. in their paper.

3.2 Evaluation

In this section, the criteria described in Chapter 2 are applied to the eight distributed system architectures. This evaluation aims to highlight the similarities and differences among the architectures, and to determine which of approaches prove the most successful.

3.2.1 Support for QoS description

The first part of the evaluation addresses the support provided by the architectures for functional and non-functional QoS descriptions.

3.2.1.1 Support for a broad definition of functional QoS

The eight architectures demonstrate that support for functional QoS can vary dramatically across different distributed system architectures. At one extreme, there are architectures that do not support any form of QoS, while at the other there are those that support QoS relating to a broad and extensible set of resources.

Jini and Coda fall into the former category. They do not support any form of negotiated QoS, but rather offer the same service to all clients.

Mobiware supports only QoS related to bandwidth. An application specifies its QoS requirements in terms of a utility function, which expresses the application or user's level of satisfaction with particular levels of bandwidth. QoS is negotiated with a CORBA mobile device object, which maintains a table of QoS specifications for each flow entering or leaving a mobile device. The provision of QoS is achieved by CORBA objects that run on mobile devices, access points and mobile-capable switch/routers. These objects support operations that allow resources to be reserved for flows according to their QoS requirements.

The ODP-based adaptive management architecture provides a broader view of QoS. This architecture uses a specification language known as NQSL (Network QoS Specification Language) to capture information about the QoS measures of the network. The measures can be associated with both links and domains, and typically characterise network attributes such as bandwidth, delay, jitter and error rates. Domains can also have QoS measures that are not communications-related. All QoS measures are expressed as name/value pairs, where the value is possibly a complex type.

QoS requirements and constraints can be captured in various ways. QoS characteristics may be associated with service and application descriptions, as well as binding types; this information is maintained by the Type Manager. Additionally, QoS requirements can be specified by an application at the time a binding request is made.

The architectural support for this QoS model is largely provided by a service known as the Network QoS Manager (NQM). The NQM maintains the NQSL descriptions of the network,

monitors the network in order to determine QoS measures, and selects routes for flows based on QoS requirements. It supports the provision of QoS to applications by reserving the resources that are required within the network.

The nomadic computing architecture likewise uses a NQM service to monitor the QoS of the underlying network. However, the architecture also adds separate mechanisms for maintaining QoS information pertaining to resources that are not communication-related. The QoS of a computer can be captured within an object description, which characterises computer resources such as CPU type and memory. This computer QoS description allows the Mobility Manager to determine which applications will run on which computers, and which computers are compatible with which users. Object descriptions relating to computers, as well as users and applications, are maintained by the Trader.

The context-aware computing system does not support QoS description, however it does provide a mechanism for applications to monitor resources and respond to changes. Information relating to resources, and other aspects of the environment, is maintained by servers known as dynamic environment servers. Environment variables may be added to the servers as required. Thus, monitoring agents can be added to the system to monitor the environment, and can communicate the data collected to interested parties, by placing the information in an environment server.

While this scheme is flexible in that many kinds of environmental conditions can be monitored, it has a number of disadvantages. First, clients cannot request resources and receive a guaranteed level of service; instead, they can only respond to changing resource availability. This may be undesirable for applications that must have certain resources in order to function. Second, clients cannot be selectively notified about changing conditions. In a number of other architectures, including Odyssey, clients are only notified when their requested QoS cannot be met. The scheme used by the context-aware architecture is less efficient in its use of communication, and therefore may prove to be less scalable.

Limbo's approach is similar to that taken by the context-aware system in several respects. QoS monitoring agents can be placed into the system as required, and can communicate their information to interested applications through the tuple space. Tuple spaces play a similar role to the dynamic environment servers discussed previously, in that they are used to disseminate information about QoS to clients, and can act as a form of shared memory. However, the tuple space does not provide a notification service to which interested clients can subscribe; instead, clients must check the tuple space when QoS information is required. The two problems that were discussed in relation the context-aware are also applicable to this architecture.

Finally, the scheme used by Odyssey allows any type of resource to be monitored, as in the last two architectures; however, it also provides a means for negotiating QoS. Each form of resource, other than the generic resources, bandwidth, network latency, disk cache space, CPU, battery power, and cost, is managed by a component known as a warden. In contrast, the generic resources are managed by the viceroy, which is an object responsible for centralised resource

management. Each type of resource must be requested by the client application, which specifies QoS requirements in terms of a window of tolerance. This window expresses the range of resource levels that are acceptable to the client. When the requested levels can no longer be met, the application is notified via an upcall.

A disadvantage of this scheme is that only resources that can be characterised by quantifiable levels can be supported.

Each of last three solutions discussed can support new types of QoS through the addition of new objects to monitor these types. This typically involves a significant amount of effort. The common approach taken by the ODP-based adaptive management architecture and the nomadic computing architecture is preferable, as it allows new types of QoS to be added without the addition of new code. This flexibility is achieved by the use of a resource management function capable of operating on the generic data supplied by the ancillary servers, and the use of a Type Manager that supports the dynamic manipulation of object types.

3.2.1.2 Support for ilities

While the majority of the architectures support some form of functional QoS, there is very little support among the architectures for ilities.

Jini addresses the provision of systemic properties, such as security and reliability, to applications by incorporating the support for these into the Jini infrastructure. For instance, a security model based on permissions is in place in all Jini systems. However, although these ilities are supported in Jini systems, they are not present as a form of QoS, as they are constant and non-negotiable.

Limbo provides the sole example of an architecture that supports the dynamic negotiation of systemic properties, however this support is of a rather limited nature. Limbo permits the negotiation of tuple space properties at creation time. These properties include ilities such as security and consistency.

The ability to support these systemic properties within tuple spaces can be used to a limited extent to enable the construction of distributed applications that possess the same attributes. For example, the presence of security within the tuple space used for communication between application components may be used to support security within the application. Unfortunately, whether or not this is actually possible is largely dependent on the nature of the application concerned and its security requirements.

The lack of support for dynamically negotiable ilities may reflect the fact that it is non-trivial to achieve. Manola proposes several approaches that can be used to solve this problem [1]. However, he also notes that all of the current solutions are insufficient and more work in this area is required.

3.2.1.3 Summary

The majority of the architectures discussed provide some support for functional QoS, with the most general architectures providing the ability to extend the scope of the QoS definition as required.

Conversely, there is very little support for non-functional QoS, and wide scope for future work in this area.

3.2.2 Support for adaptability

Having addressed the forms of QoS that various systems support, this evaluation next examines the ways in which changing QoS levels are handled.

The types of adaptability mechanism supported by each of the architectures are first addressed, followed by the policies for adaptation.

3.2.2.1 Support for adaptation at multiple levels

Three levels at which adaptation may occur have been identified. Each of these levels is addressed in turn.

Application adaptation

Support for adaptation within applications is widespread among the eight architectures. The architectures demonstrate four different approaches to supporting application adaptation.

Odyssey uses upcalls to implement application-adaptation. An upcall is an operation provided by an application that can be invoked by Odyssey when the QoS changes. The upcall is responsible for carrying the adaptation that is dictated by the application's adaptation policy.

Mobiware and, the nomadic computing architecture and the ODP-based adaptive management architecture enable application adaptation by providing QoS information through APIs. Applications can be programmed to monitor the QoS through these interfaces, and to respond to changes as they are detected.

Limbo allows QoS information to be obtained by applications through the tuple space. The QoS monitoring agents deposit their information periodically into the tuple space, and this information is accessed by clients as required.

Finally, the context-aware computing system provides clients with access to context information through a set of servers that maintain relevant environment variables. There are two mechanisms for clients to obtain information on changes in context. The first method uses polling, and requires that the client periodically check the environment server for changes. The second

method involves the delivery of events indicating changes in the environment to clients on a subscription basis.

The approaches based on upcalls and event notification have the advantage that they do not require that clients continually monitor QoS levels. These approaches will typically involve less communication than polling-based approaches, and therefore will be more scalable.

Network adaptation

At the network layer, there is considerably less support for adaptation. Only three of the architectures support adaptation at this layer. There are two distinct approaches taken by these architectures.

Mobiware implements four adaptation policies that determine when bandwidth allocations will be adjusted to reflect bandwidth availability. Mobile devices compete for bandwidth in the manner defined by the adaptation policy. Support for the adaptation policies is programmed into CORBA objects that run at various locations in the network.

Another form of adaptation can be provided by deploying filters and media scaling as necessary. Flows are scaled during periods of resource contention to improve resource utilisation, and reduce the probability of handoff dropping.

The ODP-based adaptive management architecture and the nomadic computing architecture each take a different approach, using a resource management function to support adaptation. Rather than adaptation policies being supported at network devices such as switches and access points, there is an object that monitors the entire network and performs functions such as requesting re-routing of flows as required. In both architectures, this object is known as the Network QoS Manager (NQM), and its task is to track the network topology and the QoS measures associated with different parts of the topology. The NQM supplies this information to the object responsible for managing adaptation, which may then instruct the NQM to enforce adaptation decisions that relate to network connections.

The types of adaptation that can be supported using this approach include adjustment of network resource allocations, re-routing and insertion of filter objects to reduce bandwidth requirements when necessary.

The advantage of the first approach is that it is entirely decentralised, and therefore may be a more scalable solution. The second approach is conceptually centralised; however, to achieve scalability the NQM and other objects may be implemented in a distributed fashion.

A second advantage of the approach taken by Mobiware is that functionality supporting the provision of QoS can be implemented at the network layer, allowing greater control of the network itself. The second approach controls the network from the outside, and therefore in a

more limited manner. The provision of QoS must be achieved by external actions, such as requests for re-routing.

One disadvantage of the scheme used by Mobiware is that there is no mechanism for making decisions that are globally optimal for the system based on system-wide policies.

Distributed platform level adaptation

Two significantly different approaches for supporting adaptation at the distributed platform level are displayed by the architectures. Odyssey and Limbo allow distributed platform objects to be inserted in order to support various forms of adaptation. The ODP-based adaptive management architecture and nomadic computing architecture support a form of centralised adaptation management based on bindings.

Odyssey allows adaptation policies to be built into the distributed platform level objects known as wardens. Each warden supports the management of a particular kind of resource; the types of adaptation that can be performed by a warden are particular to the type of resource managed.

An example that illustrates the type of adaptation that can be performed involves a speech-synthesizer application [6]. The warden for this application determines the relative levels of computation carried out at the mobile client and at the server, depending upon available resources.

In Limbo, adaptation mechanisms can be programmed into system agents. One of the most important forms of adaptation that is supported is filtering adaptation. Bridging agents, which propagate tuples between tuple spaces, can selectively forward tuples, or can perform transformations on tuples. For instance, if bandwidth availability is limited, only certain types of tuples may be forwarded, or tuples may be first transformed to another representation that is more compact before they are forwarded.

By using filtering agents, it is possible to support a set of tuple spaces that offer the same service at radically different levels of QoS.

The ODP-based adaptive management architecture and the nomadic computing architecture support a form of adaptation known as binding adaptation. This is based on the concept of a binding as an association between a set of objects that allows those objects to interact. Bindings may involve multiple objects in different roles, where the role dictates the interface that the object supports.

In these two architectures, the types of binding involved in each application are specified, and stored by the ODP Type Manager. Objects may request bindings from the adaptation manager, which is known as the AQM in the ODP-based adaptive management architecture, and the Mobility Manager in the nomadic computing architecture. The adaptation manager consults the trader in order to determine a set of appropriate objects to participate in the binding.

If required, the adaptation manager can adjust bindings subsequently. For example, objects participating in a binding may be replaced by other objects as required, the set of interactions offered by an interface may be limited when bandwidth is scarce, and so on.

The first two systems that were discussed, Odyssey and Limbo, support only localised, resource-specific adaptation. The objects that implement the adaptation policies are specially programmed for the purpose, and inserted at the distributed-platform layer. Thus, this type of adaptation management is difficult to add or modify.

The shared approach taken by the ODP-based adaptive management architecture and the nomadic computing architecture is more general, and supports forms of adaptation that are globally optimal according to system-wide policies. Trade-offs can be performed which would not be possible with the localised approach to resource allocation. The result is that forms of adaptation which benefit the system as a whole can be effected, rather than just those types of adaptation that benefit individual applications or users.

3.2.2.2 Support for flexible adaptation policies

All of the architectures that support adaptation also support adaptation policies. This group includes all of the architectures other than Jini. However, in many cases the support for policies is not apparent, because the policies are static and hidden amongst the implementation of the adaptation mechanisms. It is only when the policies are separated from implementation that policies are truly flexible, as they can be modified without changes to code.

Support for adaptation policies varies greatly between the architectures. Some of the architectures do not support flexibility of policies at all, whereas others support extremely general policies that are easily modifiable.

Coda, which supports a minimal form of adaptation, supports only one adaptation policy. This policy dictates the manner in which disconnections are handled, and is the basis for the design of Coda. It cannot be changed without altering the design and implementation of Coda, and thus is extremely inflexible.

In the context-aware computing system, adaptation is the responsibility of individual applications, as are the policies for adaptation. There is no support at the system layer for policies. Therefore, provision of flexible policies is the task of the application programmer.

Both Odyssey and Limbo allow adaptation policies to be implemented within platform-level objects, however this is a non-trivial task, and therefore policies would be expected to change infrequently.

Odyssey allows policies for adaptation, relating to a particular type of resource, to be programmed into wardens. For instance, the warden that supports the speech synthesizer

application, which was discussed earlier, encapsulates the policy that dictates when computation is performed locally, and when it is carried out remotely.

In Limbo, policies can be implemented within the bridging agents. For instance, a bridging agent can be programmed to propagate only certain tuples when the bandwidth drops below a certain level.

In both of these architectures, the support for new adaptation policies is provided by allowing new platform-level objects that implement these policies to be inserted into the architecture, as they are required. This approach limits the types of policies can be supported. For instance, policies that dictate system-wide behaviour cannot be implemented, as there is no centralised resource management.

Mobiware provides built-in support for various adaptation policies that can be selected by applications at run-time, however the set of adaptation policies supported by the system is fixed. The adaptation policies dictate the manner in which the bandwidth allocated to a flow will vary according to bandwidth availability. The policies supported are:

- *fast*, which means that the bandwidth allocation will be increased as soon as additional bandwidth becomes available

- *smooth*, which uses a damping period to ensure that bandwidth is not changed after brief fluctuations, however is changed in response to longer lasting bandwidth variations

- *handoff*, which changes the bandwidth allocation only after handoff

- *never*, which corresponds to a policy of static bandwidth allocation

The adaptation policies are realised by a set of adaptation proxies that run on mobile devices, access points, and mobile capable switch/routers. A CORBA mobile device object provides the mobile devices with the ability to request a QoS policy, and stores the requested policies for each flow that is transferred to or from the mobile device. A CORBA mobility agent object is responsible for managing flows and their allocated resources, and performs re-routing and re-allocation of resources as necessary, by communicating with switches via the CORBA switch servers.

While it is trivial for an application to switch between the adaptation policies that are supported, it is less trivial to support new policies. The currently available policies are implemented within the set of CORBA network objects, which would need to be re-implemented to support additional policies.

The remaining two architectures offer the most flexible type of policy.

In the nomadic computing architecture, adaptation decisions are made by the Mobility Manager, based on the range of information that is available. Descriptions of computers, users and applications are available through the Type Manager and Trader. Mobility constraints are specified in the form of relationships that hold between these objects. The take the form of rules, which may be obligations, permissions or prohibitions. One particular type of constraint that can be specified relates to co-residency requirements, which dictate which computers and users, applications and computers, and pairs of applications, can co-exist.

The set of relationship rules is maintained by the Relationship Repository. Relationships can be added and deleted as required, so that adaptation behaviour can be altered on the fly.

As well, the architecture supports adaptation policies for application bindings. These policies describe the actions that should be performed in response to environmental changes. Policy types are stored in the Type Manager, and a particular policy type for a binding can be selected by an application when it contacts the Mobility Manager with the binding request.

The ODP-based adaptive management architecture uses Sloman's policy for the specification of policies. Adaptation policies are maintained by a service known as the Policy Service. The policies specify how and when the Adaptive QoS Manager should respond to changing resource availability.

To summarise, there are four basic approaches to supporting policies that are used by the architectures considered here:

- The architecture is build around a set of policies, and hence the policies can never change. This is the approach used by Coda.

- The architecture allows resource-specific policies to be added as new platform objects are added. Odyssey and Limbo use this approach.

- The architecture provides a fixed set of policies that can be adopted dynamically. This approach is used by Mobiware.

- The architecture allows new policies in the form of rules governing system-wide adaptation to be inserted on the fly. This is the approach used by the nomadic computing architecture and the ODP-based adaptive management architecture.

It is clear that the last approach is the most flexible. This approach relies on the presence of a specification technique for policies, and a type manager for maintaining policy descriptions.

3.2.2.3 Summary

The architectures exhibit wide variation in their levels of support for adaptation.

Five of the eight architectures – namely, Mobiware, the ODP-based adaptive management architecture, the nomadic computing architecture, Limbo and Odyssey – support multiple forms of adaptation. Coda and the context-aware system each support only one form of adaptation, while Jini does not support any of the types of adaptation that have been discussed.

Of the architectures supporting adaptation, all apart from Coda support some form of flexibility with respect to adaptation policies. However, only the nomadic computing architecture and the ODP-based adaptive management architecture achieve the separation of policy from the mechanisms for enforcement. This separation is essential in allowing new types of policy to be added without the necessity of re-coding.

3.2.3 Support for heterogeneity

The final stage of this evaluation considers the support each of the architectures provides for various forms of heterogeneity.

3.2.3.1 Support for heterogeneous computing environments

In order to provide full support for heterogeneous computing environments, it is necessary to support heterogeneity of programming languages, operating systems and hardware. Three of the architectures provide this full support for heterogeneity, while the remainder all impose some constraints on the types of heterogeneity that can be tolerated.

The solutions fully supporting heterogeneity of computing environments have in common that they are based on the CORBA or ODP frameworks. Mobiware uses CORBA objects to provide the infrastructure that supports adaptability. The nomadic computing and ODP-based adaptive management architectures are both designed using the ODP framework, while the prototypes of these systems are implemented using CORBA.

ODP and CORBA support heterogeneity in at least two different ways. First, they allow application interfaces to be specified in a manner that is independent of programming language, allowing seamless interaction between objects written in different languages or running under different operating environments. Additionally, the operation of the underlying communications infrastructure is hidden, allowing different types of hardware to be used in a way that is transparent to applications.

Mobiware employs CORBA technology in a radically different manner to the other two architectures, and, consequently, its support for heterogeneity differs somewhat. In its application within Mobiware, CORBA does not form the middleware between applications itself, but supports a different form of middleware toolkit. CORBA objects are used by mobile devices to obtain communication services from the underlying network. Thus, there is a CORBA interface between the client and the network, rather than between pairs of client applications. Therefore, other mechanisms must be used for clients to interact and exchange data in a standard way that is independent of the programming language used.

Each of the remaining five architectures, other than Coda, places significant constraints upon the application programming languages that can be used.

Jini requires that applications use the Java programming language, or another language that can be complied into Java bytecodes. Additionally, user applications must be written using the Jini programming model. A set of interfaces provides support for this programming model. These are:

- The leasing interface, which provides mechanisms for obtaining resources for fixed periods.
- The event and notification interface, which provides support for event-based communication between Jini services.
- The transaction interfaces, which support atomic transactions between entities.

Odyssey, Limbo, and the context-aware computing system require that applications be capable of using libraries that are provided. This may require applications to be written in particular programming languages.

The Odyssey library is in the form of an API that is invoked through the operating system kernel. The data structures passed to the API must be of a particular format, which may require data type conversion to be carried out.

In the context-aware system, applications for ParcTabs are written using a library that supports basic functions, such as reading and writing to external memory, drawing to the display, and so on. All applications must be written in C/C++ or Modula-3 to use the library.

Limbo requires that applications be written in a language that can be linked with the Limbo stub library. However, components written in different languages can interact seamlessly. The use of a distributed system platform known as MOST is responsible for enabling this seamless interaction.

At the operating system level, the architectures exhibit a more widespread support for heterogeneity. However, Coda and Odyssey, which are implemented at the operating system layer, rather than above it, each require the use of a particular operating system.

Coda is designed to operate exclusively in an environment composed of Unix workstations. Odyssey is implemented as a set of extensions to the NetBSD operating system, a variant of Unix, and therefore can only be used in this context.

The remaining systems achieve operating system independence by offering services at a higher level than the operating system level, removing the need for applications to communicate directly with the operating system.

Finally, the all of the systems provide support for heterogeneous collections of hardware, although in some cases there are requirements imposed upon the hardware by constraints at higher levels. For instance, the requirement that computers run the NetBSD operating system that is enforced by Odyssey may place restrictions on the hardware platforms that can be used.

3.2.3.2 Support for heterogeneous interactions between application components

This section considers the types of interactions that are possible between application components. Most of the architectures support at least type of interaction. The exception is Coda, which is designed as a file system rather than a middleware toolkit.

In the remaining architectures, the most common form of communication supported is the remote procedure call (RPC).

Jini applications communicate using remote method invocation (RMI), which is a Java extension of the RPC that allows entire objects to be passed across the system, rather than just simple data. RMI provides a mechanism for clients to invoke services transparently on remote server objects. Jini is most suited to supporting applications that follow the client-server model. Complex types of application interaction that are more complex are not well supported. Moreover, the model is not well suited for systems in which there is poor connectivity.

The context-aware computing system also employs the RPC as the model for communication between client applications. The difficulties associated with using synchronous communication in situations in which there is poor connectivity are overcome using a special mechanism for delivering requests to intermittently connected hosts. A non-terminating back-off-retry technique is used to handle temporary disconnections.

A second form of communication is also permitted in the context-aware system for the dissemination of context information. A distributed event mechanism is used to notify clients of changing context.

Finally, the prototypes of the ODP-based adaptive management architecture and nomadic computing architecture are implemented using communication based on RPC semantics. However, the architectures themselves support a more general model of communication based on ODP bindings. ODP permits numerous types of interactions to be used in bindings, such as operations (RPC), notifications and stream communication.

Odyssey does not dictate a communication model that must be used by applications, but rather, allows wardens to implement communication with other parties as required. This approach means that the patterns of interactions are flexible, however they are expensive to modify, as the modification of wardens can involve significant implementation effort

Mobiware operates at the network, rather than the distributed platform, level, and therefore provides a service that is similar to that of the underlying network. Because of this, clients are

not restricted by a communication model introduced at the distributed platform layer. A disadvantage of this approach is that client applications must operate at a lower level, and be aware of network layer details, such as network addresses.

Finally, Limbo uses a form of communication based around the tuple space. This allows very general forms of interactions between clients, from peer-to-peer, to multipeer-to-multipeer communication. However, this model does not permit QoS guarantees to be made about interactions. Because communication channels are not explicitly created for interactions, there is no possibility of anticipating network requirements or allocating adequate resources, in advance.

In summary, the most flexible types of interactions are provided by the ODP-based adaptive management architecture, the nomadic computing architecture, Odyssey and Limbo. The other systems that offer communication between clients are based around the RPC.

However, although the approaches taken by Odyssey and Limbo support many types of interaction, both have significant disadvantages that are not present in the two architectures that are based on ODP. Odyssey requires communication to be carried out within platform-level objects called wardens, which introduces a significant implementation overhead when new types of interactions must be introduced. Limbo permits very flexible interactions, but the price for the lack of structure imposed on communication is that QoS guarantees cannot be made about communications.

3.2.3.3 Support for heterogeneous mobile objects

This section considers support provided for mobile objects, including mobile users, computers and applications.

Most of the architectures are designed for use in mobile computing environments, and therefore they support mobility of computers. However, support for other forms of mobile objects is provided by only two architectures. These are the nomadic computing architecture, and the context-aware computing system.

The nomadic computing architecture supports the mobility of users and applications, as well as of computers. Users may move between computers, taking with them their computing environments. Individual applications can also be migrated from one computer to another.

The architectural support for these types of mobility is provided by a set of ODP objects. The Application Migration Manager supports the migration of applications between computers. The Type Manager and Trader maintain type information and descriptions of applications, computers and users, providing the Mobility Manager with the information necessary for supporting the mobility of these objects. The Relationship Repository maintains the rules that constrain mobility of objects, in the form of co-residency requirements. These determine which pairs of mobile objects can co-reside, allowing special requirements to be captured. Finally, the Location

Manager tracks the location of users within the system, while the Network QoS Manager tracks the location of computers.

The context-aware computing system supports mobility of computers known as ParcTabs, and additionally provides the potential for mobility of users and their applications.

The architecture provides mechanisms for tracking the location of users, which can be used to support the migration of objects according to their owner's position. This approach is demonstrated by Schilit et al. using an example application that allows a user to migrate a window to a nearby display [12]. The context information is provided by the infrastructure of the context-aware system. However, the actual migration of the window to the new display is performed by the application itself, and is not supported by the system.

Thus, the extent to which the nomadic computing and context-aware architectures support mobility of users and applications varies greatly. The first architecture enables the migration of users and applications to be invoked and controlled by a centralised body that manages mobility. The second, however, provides only the potential for these types of mobility. It is the responsibility of individual applications to effect the migration of applications and user environments; the support provided by the system extends no further than a provision of information about the location and context.

Finally, it should be mentioned that Jini, while not directly supporting the forms of mobility that have been described, has potential in this area. The Jini architecture is based on Java, which provides built-in support for the migration of code between computers. This feature could be exploited to allow the migration of applications within a Jini system.

3.2.3.4 Summary

The provision of support for heterogeneity of computing environments within a distributed system has been addressed by frameworks such as ODP and CORBA. Thus, the architectures built on these foundations, namely, Mobiware, the ODP-based adaptive management architecture and the nomadic computing architecture, all achieve a high degree of support for this type of heterogeneity.

Other architectures achieve a lesser degree of support by providing their own abstractions of the underlying communications and other infrastructure.

Support for heterogeneous application interactions is achieved using the ODP concept of bindings, using a tuple space paradigm, or by allowing platform-level objects to implement new types of interactions.

Finally, support for heterogeneous mobile objects is achieved to differing degrees by the nomadic computing architecture and context-aware system, which can support mobility of users and applications, as well as of computers.

3.3 Conclusions

This chapter has addressed the support for adaptability that is provided by current architectures. This support was found to vary greatly among the architectures that were considered.

Some architectures were discovered to provide very little support for adaptability, and among these were Jini and Coda. Odyssey provided greater support, allowing objects to be added to the architecture to provide support for new types of resource with associated adaptation policies.

Mobiware provided mechanisms for adaptation to changing bandwidth, supporting both application-adaptation, and policies for controlling the form of adaptation occurring within the network.

The context-aware system took a unique approach, providing support for a form of adaptation based on context. Context information was disseminated to applications by environment servers, thus allowing applications to adapt to changing conditions.

Limbo followed an approach based on tuple spaces. Support for application adaptation was achieved using QoS monitoring agents, which communicated QoS information to applications through the tuple space. Additionally, tuple spaces could be configured for different levels of QoS as required.

A common approach was shared by the remaining architectures, the ODP-based adaptive management architecture and the nomadic computing architecture. These were based around a resource management function which controlled adaptation across the system based on global policies. These architectures were in many respects the most general and flexible, allowing the adaptation policies to change on the fly, and using a Type Management scheme to allow new types of objects and QoS to be added dynamically.

The evaluation has highlighted some areas that currently are inadequately supported by distributed system architectures. In particular, support for ilities was found to be poorly addressed by the architectures surveyed, and more work in this area is required.

Chapter 4. Modelling distributed applications in ODP and CORBA

This chapter describes two distinct approaches to modeling distributed applications that are used by RM-ODP [13] and OMG-CORBA [14]. The first uses the concept of a binding type to describe associations between application objects. The second approach describes applications in terms of components, which are objects that offer a number of ports to which other components may bind.

4.1 Bindings in ODP

In ODP, a binding is an association between a set of objects that defines the manner in which these objects can interact. Bindings are described by binding types. A binding type defines the roles of objects that participate in the binding, and the interactions that can occur between these objects. Other information can also be associated with binding types, such as QoS requirements for the binding.

As an example, the RPC could be described by a binding type with client and server roles. This is illustrated in Figure 4.1. An interface type is specified for each role. In order for the binding to be instantiated, the objects intended to fill each role must be able to support an interface that is compatible with the one specified for the role.

The behaviour of the binding can be specified in terms of the sequence of interactions that may occur. In the case of the RPC binding, the interactions must follow a set pattern, namely a request transmitted by the client to the server, followed by a response from the server to the client.

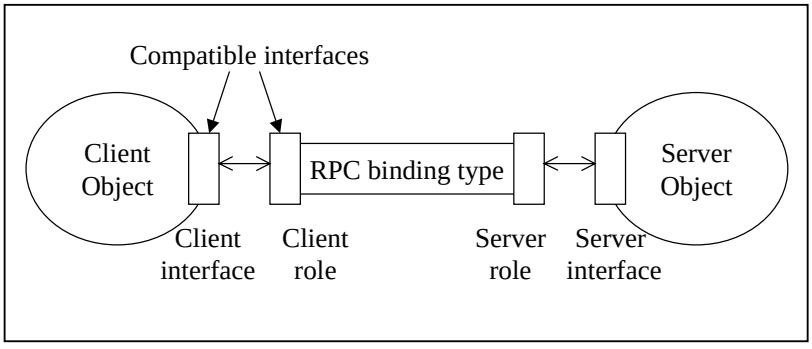

Figure 4.1. Instantiation of an RPC binding type in ODP

There are three stages involved in the establishment of a binding.

First, all of the objects that will be involved in the binding make assertions about the type of interfaces that they are able to support. Next, a suitable binding type is identified, and, finally, this is instantiated with each of the objects bound in a particular role.

A binding may be instantiated either by one of the objects involved (for instance, the client in the RPC example presented earlier), or by a third party, such as a trader.

4.2 The CORBA Component Model

The ODP approach to describing applications uses a macroscopic view of application objects and their interactions. A binding type captures relevant information about the associations between objects, including the interfaces provided by each of the objects, and the pattern of interactions that may occur. The CORBA component model follows a different approach, describing associations in terms of the types of interactions that can be supported by individual component objects. Components, unlike ODP objects, can be defined compositionally. That is, a component is not necessarily a simple object, but may provide an abstraction of several objects or components bound together.

The CORBA component model has not yet been standardised by the OMG, however is currently undergoing review.

The component type that has been proposed is a new CORBA meta-type that is an extension and specialisation of the object meta-type. Component types describe implementation objects that provide functionality in a manner that hides the detail of implementation from the user. Interaction with a component typically occurs through one of the following five types of ports:

- facets
- receptacles
- event sources
- event sinks
- attributes

A component's facets are the distinct interfaces it provides for interaction with other CORBA objects. Facets are intended to provide the means by which client objects obtain service from a component. A client can obtain a reference to a facet by invoking the `provide_<name>` operation on a component, where `<name>` corresponds to an interface provided by the component.

Receptacles are points at which the component accepts connections to other objects upon with the component can invoke operations. Receptacles may accept one or more object references

simultaneously. Receptacles are implemented as sets of operations for establishing and managing connections. In particular, the `connect_<receptacle_name>` operation accepts an object reference as a parameter, and attempts to create a connection between the named receptacle and the object represented by the reference. The operation `disconnect_<receptacle_name>` is used to terminate an existing connection.

CORBA events enter and leave components through event sinks and sources respectively. Component event sources hold references to consumer interfaces into which events are pushed. Conversely, component event sinks provide interfaces into which events can be pushed by event sources.

Component attributes are intended primarily for configuration purposes. Attributes may be read-only, in which case an accessor operation is provided in order for clients to obtain the attribute value. Attributes may also support modification of the attribute value through a mutator operation.

The CORBA component type forms an abstraction of an implementation object that captures information about the interactions supported by the object. This information can be used to determine whether application objects can be bound together at run time, and thus can perform a similar role to the information associated with ODP binding types.

An illustration of the use of CORBA components in practice can be obtained by returning to the RPC example introduced earlier. The client and server objects involved in the interaction can each be model by CORBA components. The server object provides an interface, through which the client invokes the RPC request. The client requests an instantiation of this interface, which is returned as a reference to a facet of the server component. This is illustrated in Figure 4.2.

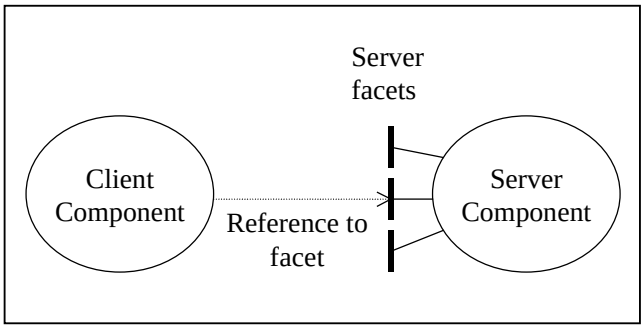

Figure 4.2. CORBA facets example

4.3 Conclusions

The two models discussed both provide mechanisms for modeling information about potential associations between objects in distributed applications. In the ODP model, this information is maintained within binding types, which provide centralised repositories for information about associations. In the CORBA component model, the information is distributed across component

types, which model application objects and the ports they provide for interaction with other objects.

This difference of approach may have an impact on the support for adaptability that can be achieved under each scheme. For example, the description of QoS may differ depending on the model. Using ODP bindings, it may be more natural to describe QoS that is associated with the overall binding rather than with individual interfaces. However, the reverse will probably be true in a CORBA environment.

The differences may have the result that adaptation types that are possible using one of the models have no equivalent in the other. For instance, the CORBA component model is amenable to a type of adaptability that is supported by the ability to configure components. Components can be adapted through reconfiguration in response to changing requirements of clients. Configuration of components may also provide a means for achieving systemic properties, such as security or reliability, which must be supported uniformly across a system.

Chapter 5. An adaptive Web application

The practical component of the project considers adaptability in the context of an example application. The Web application has been chosen for study, for the reason that it has wide applicability and is familiar to most computer users.

This chapter describes the design for an adaptive Web application. The application consists of an adaptive Web browser, and a HTTP server that supports adaptation within the browser.

The chapter begins by characterising a typical Web client/server application, before focussing on an example Web client. The application considered is Grail [15], an extensible Web browser created by the Corporation for National Research Initiatives (CNRI)[1]. The architecture of Grail is described with particular focus on features that facilitate the extension of the browser.

The remainder of the chapter considers the incorporation of adaptability into Grail. A number of different types of adaptability that are suitable for use in Web applications are described. Subsequently, some modifications to Grail's architecture to enable adaptation are outlined.

Finally, the design of a HTTP server that supports some types of client-side adaptation is briefly discussed.

5.1 A Web application

A typical Web application consists of client and server entities that communicate using a common protocol. The protocol defines the formats for messages exchanged between the communicating entities, the valid message exchange sequences, and so on. A large number of protocols have been defined, including the File Transfer Protocol (FTP), and the Hypertext Transfer Protocol (HTTP), among others. This discussion will focus only on HTTP, which is one of the most widely used protocols.

5.1.1 HTTP

HTTP is an Internet standard that has been in use on the Web since 1990, and has undergone several revisions since that time. The current version is HTTP/1.1 [16].

The protocol defines two types of message: *request* and *response*. Both conform to a standard message format. A request is sent from a HTTP client to a server, and indicates the type of

[1] http://www.cnri.reston.va.us/

service that is requested. A response is sent by the server in reply to a client's request. The response message includes the request status and the result of the request, if applicable.

The interaction between client and server entities is shown in Figure 5.1.

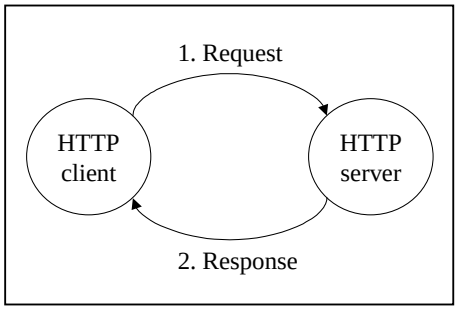

Figure 5.1. Interaction between HTTP client and server

The HTTP specification describes a number of methods and corresponding message headers that indicate the purpose of a request. Methods operate on objects described by URIs (Uniform Resource Identifiers). A URI may be either a location (URL) or a name (URN).

The following methods are defined by HTTP/1.1:

- **Options.** This method represents a request for information about the communications options available along the request/response path for the particular URI indicated.
- **Get.** This is used to retrieve the information associated with a URI.
- **Head.** This is similar to the Get method; however, the response contains only meta-information, and must not contain a message body.
- **Post.** The Post method requests that the entity enclosed in the message be accepted as a subordinate of the object identified by the URI. For instance, if the object is a mailing list, this method represents the posting of a message to the mailing list.
- **Put.** This method requests that the enclosed message body become the entity represented by the URI.
- **Delete.** This is used to delete the entity represented by the URI.
- **Trace.** The Trace method is used for testing purposes. The response should reflect back the message received by including it as the body to the response.
- **Connect.** This method is used with proxies that are capable of operating as tunnels.

The type of method requested by the client, the URI and the HTTP version used are all included in the first line of a HTTP request, known as the request line. Following lines may optionally contain a number of request header fields. These fields provide additional information about the request, such as the content types, languages and encodings that will be accepted by the client, authorisation information, and caching information.

The end of the request header is denoted by a blank line. A request body, containing data pertaining to the request, may optionally be included after the request header.

A HTTP response header consists of a status line followed by optional header fields, and is terminated in the same manner as the request header. The status line contains the HTTP version, a status code indicating the server's ability to fulfil the request, and possibly a text explanation of the status code.

5.1.2 Content Negotiation in HTTP

HTTP permits the existence of several versions under one URI. Each of the versions is known as a variant. HTTP content negotiation provides a facility for the client and/or server to determine which of the available variants should be returned to the user in response to a request.

Several forms of content negotiation exist. Server-driven negotiation involves selection by the HTTP server, while in agent-driven negotiation, the selection of the best variant is performed by a user agent based on variant information contain in the server's response message. Transparent negotiation, described by RFC 2295 [17], combines both server-driven and agent-driven negotiation. It allows caches to perform server-driven negotiation on behalf of the origin server, provided the cache understands the aspects in which the variants may differ.

Several HTTP header fields have been defined to support content negotiation. The Accept family of header fields allows the user to express preferences for particular types of variant. The Accept header field enables the client to indicate the media types that are preferred. For example, the client can request a plain-text type as follows:

Accept: text/plain

It is also possible for the client to specify a number of preferences. A precedence level between zero and one can be placed beside each preference, with a one indicating the highest preference, and zero indicating the lowest. For instance

Accept: audio/*; q=0.5, audio/basic

indicates that the client prefers audio/basic (a non-specified precedence defaults to one), but is moderately satisfied (50%) with any form of audio. Other media types are unacceptable to the client.

Similarly, the Accept-Charset, Accept-Encoding and Accept-Language header fields can be used to request particular character sets, content encodings and languages, respectively.

Server-driven content negotiation often uses the information contained in the Accept header fields to select the variant that will best satisfy the client's requirements.

The Vary header field is a response header field that supports the caching of variants. It indicates the set of request header fields that were used to select the particular variant in the response. This information can be used to determine when a cached variant can be used in response to future requests.

Transparent content negotiation also makes use of the Alternates response header field and Accept-Features request header field.

The Accept-Features header allows the user agent to provide information about features required by the user that cannot be captured using the other Accept header fields. For example, the header field

Accept-Features: colourdepth = 5, dpi = [300-599]

requests a colour depth of 5 and between 300 and 599 dots per inch.

The Alternates header field is used to convey information about variants to the user agent. For example, the following header describes three variants of a paper, including HTML versions in English and French, and a postscript English version.

```
Alternates:  {"paper.1" 0.9 {type text/html} {language en}},
             {"paper.2" 0.7 {type text/html} {language fr}},
             {"paper.3" 1.0 {type application/postscript} {language en}},
             proxy-rvsa="1.0, 2.5"
```

The last line in this header restricts the variant selection algorithms that can be used by proxies. One particular algorithm, known as RVSA/1.0, is intended for use with transparent content negotiation. It is described in RFC 2296 [18]. The algorithm combines the user preferences expressed in the Accept headers with information about the quality and features of variants to determine which variant is best for the user.

The algorithm computes quality values for each of the variants using the following calculation:

$$Q = \text{round5}(qs \times qt \times qc \times ql \times qf)$$

Here, round5 denotes a function that rounds its floating-point argument to five decimal places, and the q-values are defined as follows:

- qs is the quality of the variant source, as defined by the variant description.

- qt is the media type quality factor. Its value is one if no type attribute is provided in the variant description, or there is no Accept header field in the request; otherwise, it is the quality assigned to the media type by the Accept header.

- qc is the charset quality factor. Its value is one if there is a charset attribute in the variant description, or there is no Accept-Charset header; otherwise, it is the value assigned to the charset by the Accept-Charset header.

- ql is the language quality factor. It is one if there is no language attribute in the variant description or no Accept-Language header in the request. Otherwise, it is the highest quality factor assigned by the Accept-Language header to any one of the languages listed in the variant description.

- qf is the features quality factor. It is one if there is no features attribute in the variant description or there is no Accept-Features header in the request. Otherwise, it is the quality degradation factor for the features attribute. Each predicate listed in the Accept-Features header is assigned a true-improvement if it evaluates to true, or a false-degradation if it evaluates to false. The evaluation of predicates uses the features attribute in the variant description. The improvement and degradation factors can be set in the variant description; otherwise, they default to one and zero, respectively.

The overall quality values assigned to variants are classified as definite or indefinite. A value is definite if it was computed without the use of wildcards ('*' characters) in the Accept header fields, and without the need to use the absence of a particular Accept header field. Otherwise, the value is speculative.

The best variant is defined as the variant with the highest overall quality value, or the first of several such variants.

If all of the following conditions are met, the best variant is returned to the client in the response:

- the overall quality value of the best variant is greater than zero

- the overall quality of the best variant is a definite value

- the variant resource is a neighbour of the original negotiable resource

The last of these requirements is present for security reasons.

If the three conditions are not met, the server cannot select a variant. Instead, the server responds with the list of available variants, using the Alternates header to convey the variant information. It then becomes the task of the user agent to select the most appropriate variant.

5.2 Grail architecture

Grail is a prototype Web browser that has been developed with a view to extensibility. It is written using the Python scripting language making heavy use of the Python HTTP libraries and the Tk interface toolkit.

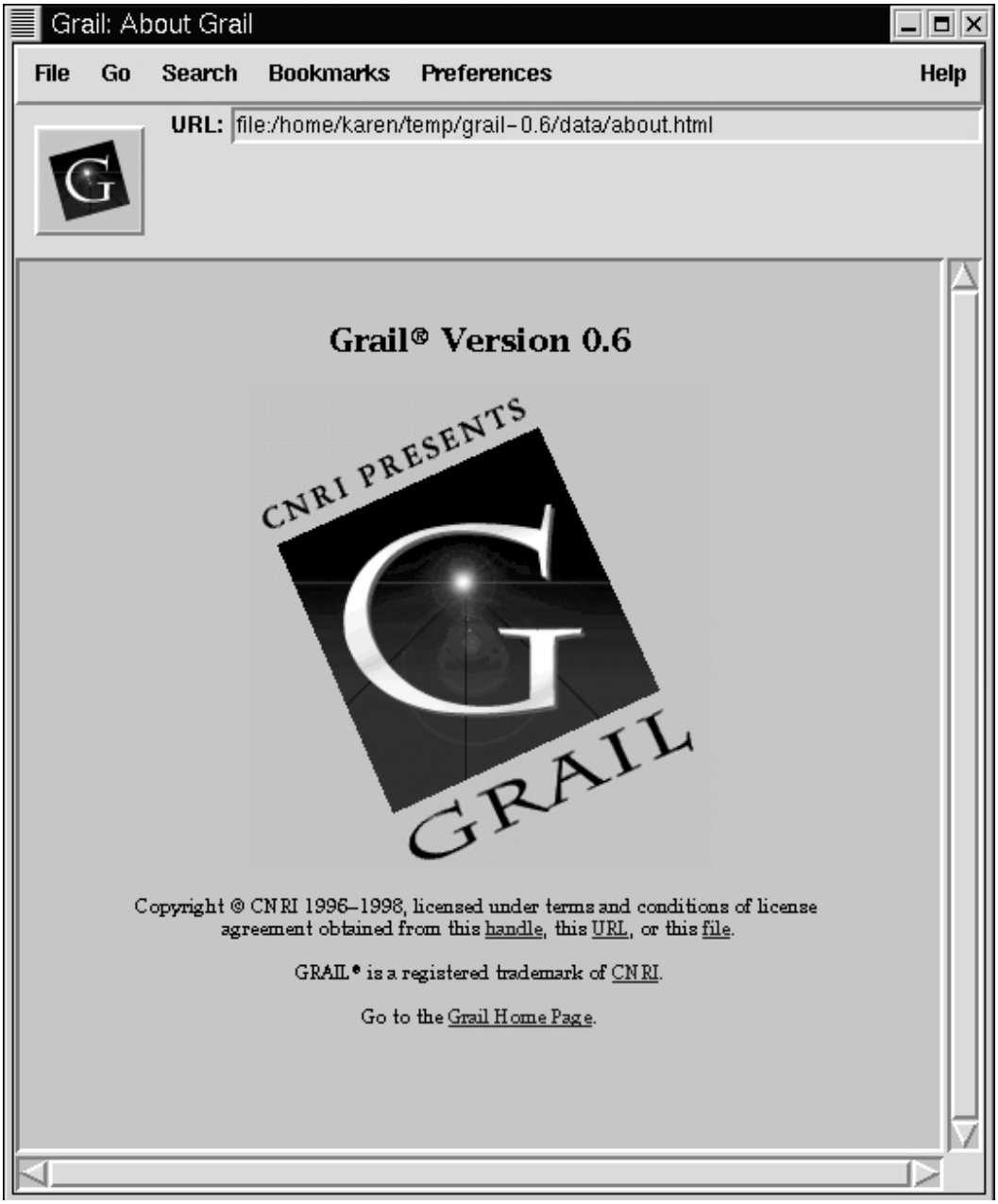

Figure 5.2. The Grail user interface

The interface and functionality provided by Grail are similar to those of popular browsers such as Netscape Navigator. The Grail browser window is shown in Figure 5.2.

Grail supports HTML version 2.0, as well as a number of features from more recent versions. Support for special Python applets is also provided. These are executed in a restricted execution environment in order to prevent malicious attacks on a user's machine.

The architecture of Grail is complex and poorly documented. However, a simplified architecture is shown in Figure 5.3. This diagram was constructed through a detailed analysis of Grail's implementation, comprising in excess of one hundred Python modules and 30 000 lines of code.

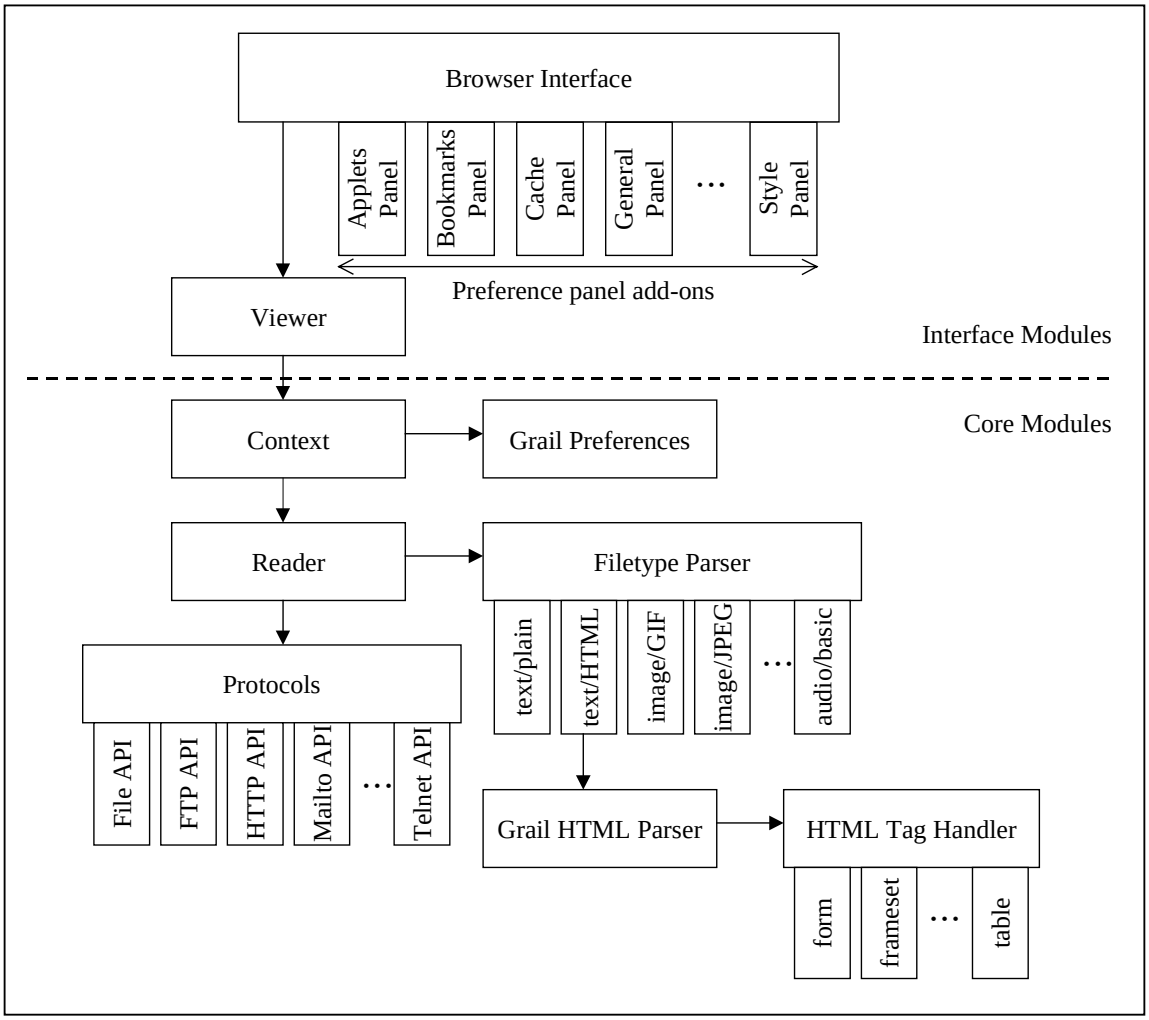

Figure 5.3. Simplified Grail architecture

In this diagram, an arrow from a component *A* to a component *B* indicates that *A* uses *B* to perform its task.

The browser uses an event-driven paradigm. Actions arise through user interaction with the Tk interface. For instance, the code to load a Web page is invoked when a user enters a URL into a text box in the user interface. The loading of a file involves a number of Python modules. The first of these is the Grail Context module, which maintains information about the current browsing context, such as the user's browsing history. The Context module determines whether a file can be reloaded from the cache of recently accessed files. If this is not possible, the requested file is retrieved using a Reader object. The Reader invokes the appropriate protocol handler in order to obtain the file, and then uses a parser to interpret the data contained in the file. The parser used is chosen according to the type of the file concerned. For instance, a HTML file must be treated differently to a plain text file. The parsing of the latter type is straightforward, whereas parsing of a HTML document may involve steps such as the downloading of in-line images and the processing of HTML tags.

The ability to extend Grail arises from the use of plug-in modules to implement parts of the functionality. The following four types of plug-in are supported by Grail's architecture:

- protocol plug-ins
- HTML plug-ins
- file-type plug-ins
- preference panel plug-ins

The positioning of each of the plug-in types within the architecture is depicted in Figure 5.3.

Each protocol plug-in provides support for a type of protocol scheme, such as FTP or HTTP. The protocol support currently provided by Grail is all supplied by extensions, resulting in a flexible set of supported protocols.

HTML plug-ins each support one or more HTML tags. They define how the tags are handled during the parsing of HTML documents, and provide a simple mechanism for extending the HTML parser.

A file-type plug-in is used to support a particular type of file that can be loaded by the browser. A file-type is expressed in the form *major/minor*, where *major* describes a general class of file types, and *minor* is a specific subclass of the general class. Currently supported file-types include text/plain, text/html, image/gif, image/jpeg and audio/basic. Grail applets are also supported using a file-type plug-in.

The implementation of a file-type plug-in should provide a parser for that type, which determines the manner in which a loaded file is handled by the browser.

The final type of plug-in allows new preference panels to be defined for the browser. A Grail preference panel is a type of user interface component that allows the user to manipulate preference settings. Examples of preference panels provided with Grail include a style panel for manipulating attributes of the user interface, such as font size and colour, and a proxy panel for setting browser proxies. The style panel is shown in Figure 5.4.

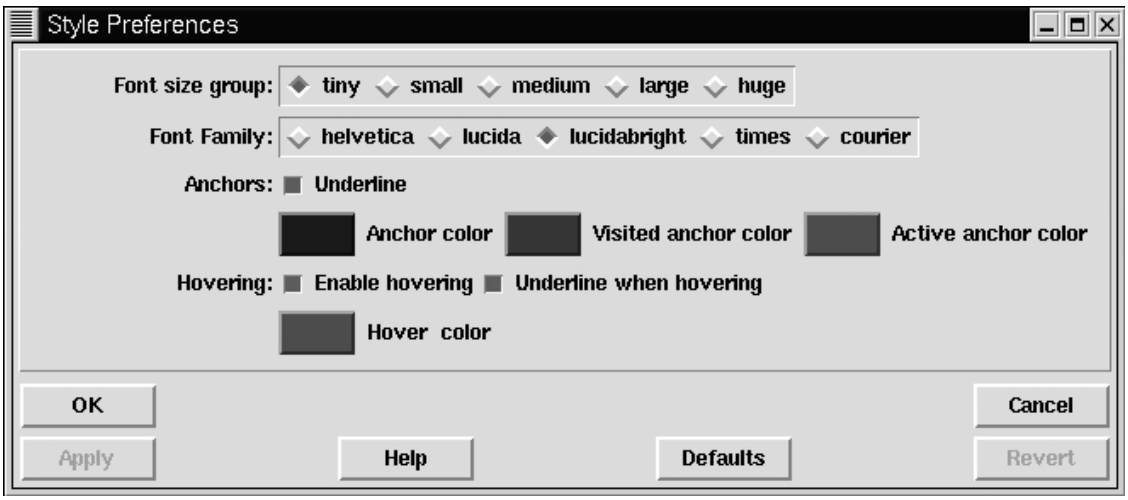

Figure 5.4. Style preference panel

5.3 Adaptation mechanisms for the Web browser

This section considers types of application adaptation that can be incorporated into a Web browser, such as Grail. The types of resource required by the browser are considered, along with the changes these resources might experience due to mobility. Next, mechanisms for responding to these changes are described.

The resources used by the browser are shown in Figure 5.5. There are three general classes of resource used by Grail:

- **Libraries.**
  The libraries required by Grail all belong to the standard Python distribution. They include the Tkinter module, which provides access to the Tk user interface toolkit, the httplib module, which implements a simple HTTP client, the urllib module, which supports manipulation and retrieval of URLs, and the socket library, which provides a low-level network interface.

- **Applications.**
  The applications used by Grail are language interpreters for Python and Tk. The Grail modules directly use the Python interpreter. This interpreter in turn invokes the Tk interpreter in order to render components of the user interface.

- **System resources.**
  Both interpreters make use of a number of system resources, such as CPU, memory and disk space. The Tk interface also makes use of the user's display to render the interface. Finally, the Python interpreter uses the operating system's network services to carry out operations on network sockets.

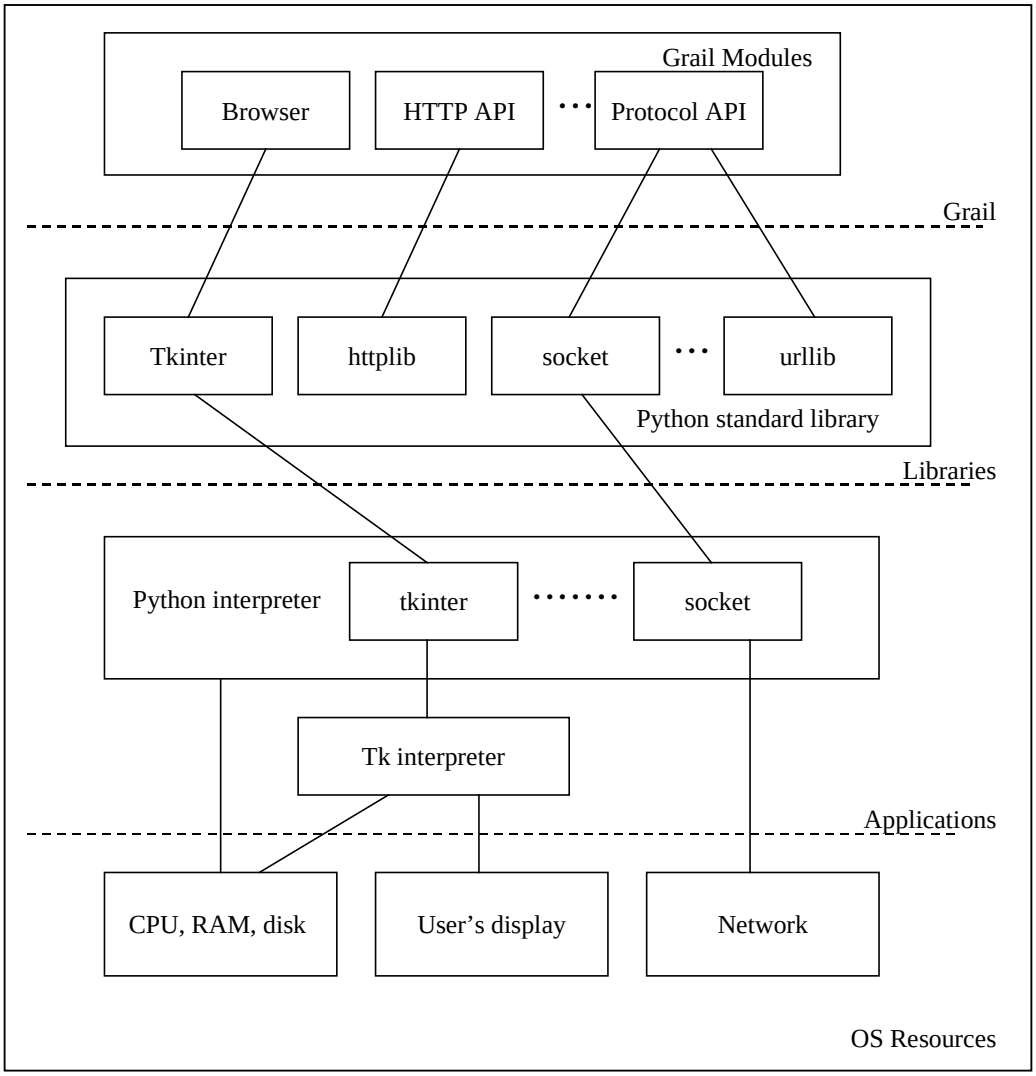

Figure 5.5. Grail resource usage[2]

[2] This figure is adapted from a diagram located on the Grail home page. At the time of writing, the original diagram is available at the following URL: http://grail.cnri.reston.va.us/grail/info/diagram.gif.

Mobility of users, computers and applications can affect any one of these classes of resource. However, changes in the availability of the first two classes of resource cannot be tolerated at all by Grail, and therefore only adaptation to changes in system resources will be considered.

Mobility can result in changes to any of the types of system resource used by Grail. Mobility of the browser between computers can cause changes in CPU processing power, memory and disk space availability, network QoS, and characteristics of the user's display. Mobility of the computer executing the browser can result in variations in the network QoS.

In the next few sections, types of adaptation that could be used to respond to variation in resources are described.

5.3.1 Adapting to changing CPU and primary memory resources

In general, problems due to lack of processing power or main memory rarely arise in browsers executing in Workstation environments. However, problems may arise when execution is performed on resource-poor mobile computers. In this scenario, it may be appropriate for the functionality of the browser to be scaled back to match the capabilities of the local machine.

In order to adapt to limited processing power, CPU-intensive tasks should be identified. These tasks should eliminated if possible when processing power is limited. For instance, it may be appropriate for animations, video and applets to be disabled when the CPU resources are scarce.

There are at least two possibilities for dealing with limited memory. One possibility is to scale back the browser itself by limiting its functionality and thereby reducing the number of modules that are kept within memory. A second possibility is to limit the size of files that are downloaded by the user. For instance, large video and image downloads could be disabled, or could be transformed to more compact representations, such as greyscale rather than colour.

Many browsers support a limited form of user-controlled adaptation that can be used when resources are scarce. The user is able to disable certain types of downloads, such as images and applets. This can be effective in reducing resource requirements, however it is inflexible, as the user is restricted to an all-or-nothing choice.

5.3.2 Adapting to changes in secondary memory availability

A Web-browser's primary use of secondary memory is for caching purposes. The browser typically retains recently read files on disk so that these can be retrieved from the cache, rather than over the network, if the user requires these files again.

Most browsers use a static caching policy that dictates the length of time files can reside in the cache before they become stale and must be purged, and the maximum capacity of the cache. In some browsers, such as Grail, the user can manipulate aspects of the caching policy. The preference panel that Grail provides for this purpose is shown in Figure 5.6.

In order to support dynamic adaptation, a browser can employ an algorithm that modifies the caching strategy according to the current availability of secondary memory.

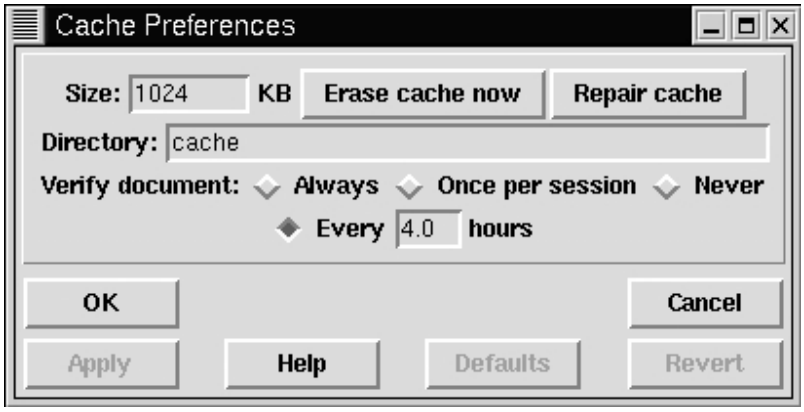

Figure 5.6. Grail cache preferences panel

5.3.3 Adapting to changing network resources

In circumstances in which the available network resources, such as bandwidth, are poor, it may be necessary for an application to reduce its bandwidth requirements. This may be achieved in a number of ways.

One possibility is to disable the downloading of objects that exceed a particular size, such as large images or video. However, this approach has limited applicability, as it is often impossible to determine the size of an object before it is downloaded. A second possibility is to download the most compact available representation of each object, for instance to obtain a compressed JPEG image rather than a larger GIF, or a plain text document rather than a postscript one.

Unfortunately, HTTP currently does not support the specification of complex user requirements. For instance, the client cannot specify a requirement for a file variant shorter than a particular length, or a requirement for the smallest available variant.

At least two different approaches can be used to solve this problem. One approach involves allowing the browser to perform the selection of variants. In this approach, the browser obtains variant information from the server via the Alternate header field in the server's response, and uses this to determine the most appropriate variant.

A second approach involves extending HTTP to allow clients to specify additional requirements in their requests to enable servers to carry out variant selection. This can be achieved through the addition of one or more new HTTP header fields that can carry information about the additional requirements.

The advantage of the first approach is that the browser has full control over the selection of variants. Two disadvantages include increased complexity of the browser and difficulty in re-

using the selection algorithms in other applications. The second approach does not suffer from these disadvantages, however it introduces several disadvantages of its own. First, it may lead to large request headers, as clients need to supply all of their requirements in requests to ensure the optimal response. Secondly, it is unlikely that clients will be able to express all of their requirements in requests, so that the variants selected by servers are likely to be sub-optimal for clients in some cases.

Although the adaptation of downloads to bandwidth availability is likely to be beneficial to clients in the majority of cases, it may be undesirable in others. For instance, in cases in which the user downloads critical data, such as a digitized X-ray to be used for diagnostic purposes, it is inappropriate for the browser to accept a low-fidelity variant. Therefore, the browser should allow the user some control over the adaptation mechanisms and policies.

5.3.4 Adapting to a changing user display

If migration of the Web browser between machines is possible, the ability to respond to changing display type may be necessary. For instance, it may be desirable to present a rich interface to the user on a large workstation display, but not on a hand-held PDA.

The types of adaptation that are suitable are likely to depend on the displays involved. For instance, if the difference between display types is small, changing the font size and the dimensions of the browser window could be adequate. However, if the difference is more pronounced it could be necessary to change the user interface components displayed, such as toolbars or status bars.

In addition to changes in the browser interface, changes to the way in which Web pages are displayed might also be appropriate. For instance, on extremely small displays, it may be preferable to display only text by disabling images.

In certain cases, it may also be appropriate to adapt downloads to the display type. For instance, the smallest variants of images may be downloaded when the user display is small and larger variants obtained when the display is large. Two approaches to achieving the adaptation of downloads were discussed in the previous section.

Several mechanisms for controlling the user interface are typically provided by Web browsers. Most browsers allow the user to set the current font, change the window size, choose colour schemes, enable/disable the displaying of images, and so on. The addition of algorithms for determining suitable settings according to the display type can be used to provide dynamic adaptation. However, these should not be unnecessarily restrictive to the user. For instance, the algorithm should not prevent the user from viewing images if this is desired. Allowing the user to determine the adaptation policies that are used may be an effective way to ensure that the user is not frustrated by undesired forms of adaptation.

5.4 Adaptable Web browser design

This section considers the incorporation of some of the types of adaptability that have been described into a Web browser.

The first part of this section describes a general framework for supporting the adaptability within an application, such as a Web browser. The second considers the incorporation of adaptability into Grail's architecture.

5.4.1 A scheme for supporting application adaptation

In order for an application to be capable of adaptation to its environment, a means for obtaining information about the current environmental conditions is required. The approach described here fulfils this requirement by providing a monitor object that allows client applications to be notified of QoS changes that will affect them.

The scheme is based on concepts used by the Odyssey architecture, which was described in Chapter 2. Odyssey allows clients to describe their QoS requirements in terms of a window of tolerance. Clients define upcall handlers, which are invoked when available resource levels stray outside the window of tolerance.

The approach described here is based around an object called the QoS Monitor (QoSM). This object allows a client application to indicate the resources that are of interest, and a number of predicates that determine when the application will be notified of changes in resource availability. Predicates are expressed as pairs consisting of a binary comparison operator together with a value. A given predicate is considered to evaluate to true whenever

```
current_level op value
```

evaluates to true, where `current_level` is the current level of the resource, and `op` and `value` are the operator and value specified by the predicate, respectively. The client application is notified whenever the state of the resource changes so that the predicate becomes true.

The use of predicates, rather than a window of tolerance, for determining when callbacks should occur allows for greater flexibility in the type of resources that are managed. For instance, a resource that is always in one of a finite set of discrete states, such as *off* or *on*, can be described using predicates, however is not well described by a window.

Another distinction must be made between the approach used by Odyssey and the approach described here. In Odyssey, resource management and resource monitoring are intertwined, and are the concern of the same party. The QoSM described here, however, is concerned only with the *monitoring* of resources.

In order to permit maximum flexibility in the implementation of the QoSM and client applications, the QoSM is abstracted as a CORBA object. This allows the QoSM to be capable of

interworking with applications written in a range of different implementation languages, on a range of different platforms. Additionally, it ensures location-transparency. This is important if the QoSM must be located on a different machine to the application. This may be necessary due to the nature of resources monitored by the QoSM. For instance, if the QoSM is concerned with network QoS, it may need to be located at a node within the network.

The next two sections describe the interfaces and the internal construction of the QoSM, respectively.

5.4.1.1 Interfaces

The QoSM provides an interface to clients that allows them to register interest in resources and indicate the circumstances in which they wish to be notified of changes in those resources. This interface is kept deliberately simple; this is done to allow the basic ideas remain clear from cluttering detail.

The interface is shown below in CORBA IDL.

```
interface RegisterIF
{
    typedef string ResourceType;
    typedef string ResourceValue;
    typedef enum CompOp {greater, less, equal, less_or_equal,
        greater_or_equal} ComparisonOp;
    typedef struct Pred
    {
        ComparisonOp op;
        ResourceValue value;
        ChangeNotificationIF notificationIF;
    } Predicate;
    typedef sequence<Predicate> Predicates;

    exception InvalidResourceType {};
    exception ResourceNotRegistered {};

    void Register(in ResourceType resource, in Predicates preds)
        raises(InvalidResourceType);
    void RemoveRegistration(in ResourceType resource)
        raises(ResourceNotRegistered);
};
```

Resources types and resource values are expressed as strings. It is intended that resources will be described by name, such as *display type*, or *bandwidth*. The strings that are permissible as values for these resource types will depend upon the nature of the resource; for instance, the string values for the *bandwidth* resource might contain numbers of bits per second.

Five simple comparison operators are defined for use in predicates. The operators may be interpreted differently for different resource types, and for certain resource types not all of these operators will be applicable.

The interface defines predicates as triples consisting of an operation, value and notification interface. The notification interface is used by the QoSM to notify the client application whenever the predicate becomes true.

The QoSM provides two operations to its clients. The first operation is used by the client application to register interest in a particular resource type. One or more predicates should be specified to indicate which changes are relevant and should be reported to the client. An application can invoke this operation several times for the same resource type. Each time this occurs, the list of predicates specified by the client will completely replace any predicates specified earlier.

The register operation will raise an exception if the resource specified by the client is not a valid one, that is, if the resource type is not supported. No checking of the predicate list is performed during registration, and predicates that are not properly formed will be ignored.

The second operation provided by the QoSM is used by a client application to indicate discontinued interest in a particular resource. Subsequently, that client will not be notified of any changes in the resource unless the client again registers interest. An exception is raised by this operation if the resource specified is not one that is currently being monitored for the client.

A standard interface is used by the QoSM to notify clients of changes. This is described in CORBA IDL below.

```
interface ChangeNotificationIF
{
   void Notify(RegisterIF::ResourceValue value);
};
```

The `Notify` operation is invoked by the QoSM whenever a predicate becomes true. The parameter indicates the current value of the resource. Clients of the QoSM must provide at least one implementation of this interface, however may provide many. In the case that the client provides only a single instance of the interface, this is used to handle all callbacks from the monitor. Alternatively, the client may choose to provide different instantiations to handle different types of callback.

5.4.1.2 Design of the QoSM

A primary goal of the QoSM is to enable its users to monitor the status of any resource type that is of interest. Therefore, the design is motivated by the need to easily add components to monitor new types of resource.

This is achieved using plug-ins similar to those used by Grail. Each type of resource is handled by a plug-in module that presents a standard interface to the coordinating module. This architecture is shown in Figure 5.7.

The details of plug-in implementation will depend upon the type of resource concerned, however each plug-in should provide a Register operation that takes a list of client-supplied predicates as its parameter.

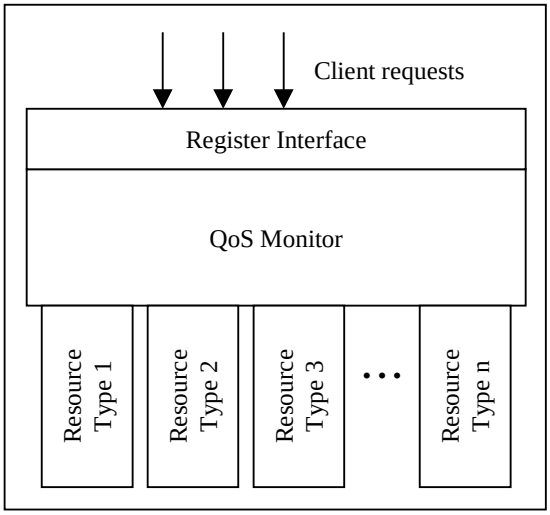

Figure 5.7. QoSM architecture

The types of plug-in that are implemented will depend on the requirements of the client application. The following are some examples of resources that might be monitored by QoSM plug-ins:

- network bandwidth
- memory
- user display

5.4.1.3 Discussion

This section has described an infrastructure for supporting adaptation comprising a resource monitor implemented as a CORBA object. While this infrastructure is sufficient to allow adaptation of applications, it is limited.

First, there are limitations on the use of the monitor, which arise from the nature of the operations it provides. The monitor is intended for use by one application only, as the registration of a set of predicates relating to a resource will completely delete all previously registered predicates. This problem is easily overcome by providing a separate operation to delete predicates relating to a resource.

Another limitation arises from the nature of the predicates that are used by the QoSM. Five predicate types have been described, all of which are binary comparison predicates. Unfortunately, there are circumstances in which these types of predicates are not appropriate. For instance, a user may wish to specify that callbacks should occur every time the resource value changes, without enumerating the entire set of possible values. This limitation can be removed by adopting a more expressive predicate language.

Other problems with the infrastructure described in this section include its limited support for QoS and lack of direct support for object mobility.

Although support for QoS monitoring is provided by the infrastructure, QoS negotiation and resource management are not supported. In other words, while QoS can be tracked by the system, it cannot be controlled. This may make it unsuitable for applications with critical resource requirements.

The mobility of objects, such as applications and users, is also neglected by the infrastructure. This limitation can be eliminated by providing mechanisms for migrating applications and maintaining mobility profiles that can be used in the management of mobile objects. This approach has been adopted by the nomadic computing architecture, described in Section 3.1.6.

5.4.2 Adding adaptability to Grail

This section considers the use of the QoSM to support the adaptability of Grail, focussing on the impact of adaptability on Grail's architecture.

The QoSM can be used by Grail as the means for detecting the need for adaptation. In order to achieve this, the QoSM must be informed of the events that should trigger adaptation. This can be accomplished through the use of predicates.

For instance, suppose that Grail supports a policy in which the user interface is scaled back when the application is migrated to a PDA. The QoSM can be used to trigger this form of adaptation.

To enable this, the application must first register with the QoSM. The resource type that is of interest is *machine type*, and the predicate of interest is *value = PDA*. Assuming that the QoSM has a plug-in for monitoring the machine type, the registration will succeed, and the QoSM will subsequently track the machine type of behalf of the application.

When machine type changes so that the predicate evaluates to true, a callback will occur. Finally, the callback performs the changes to the interface as specified by the adaptation policy.

This scenario is depicted in Figure 5.8.

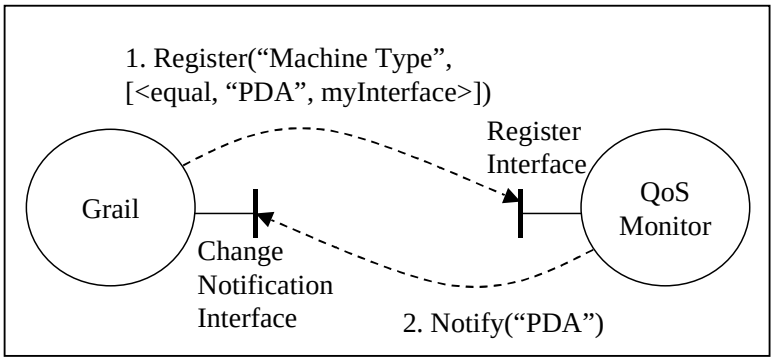

Figure 5.8. Interaction between Grail and the QoS monitor

The most notable impact of this scheme on Grail's architecture is the need to support callback handlers. In order to support easy extension of the set of monitored resources, the handlers can be defined as Grail plug-ins, with one module being provided for each resource type. Note that, although there is a single module for each resource, there may be multiple callback handlers within each module. This modification to Grail's architecture is shown in Figure 5.9.

The notification handlers may make modifications to the current browsing Context to reflect the current state, and carry out other actions, according to the adaptation policies.

A second extension can be easily made to allow the user to control the adaptation policies used by Grail. This can be supported by adding Grail preference panel plug-ins. The types of policy that can be selected by the user will depend on the type of resource concerned. This modification does not require any changes to the browser's architecture

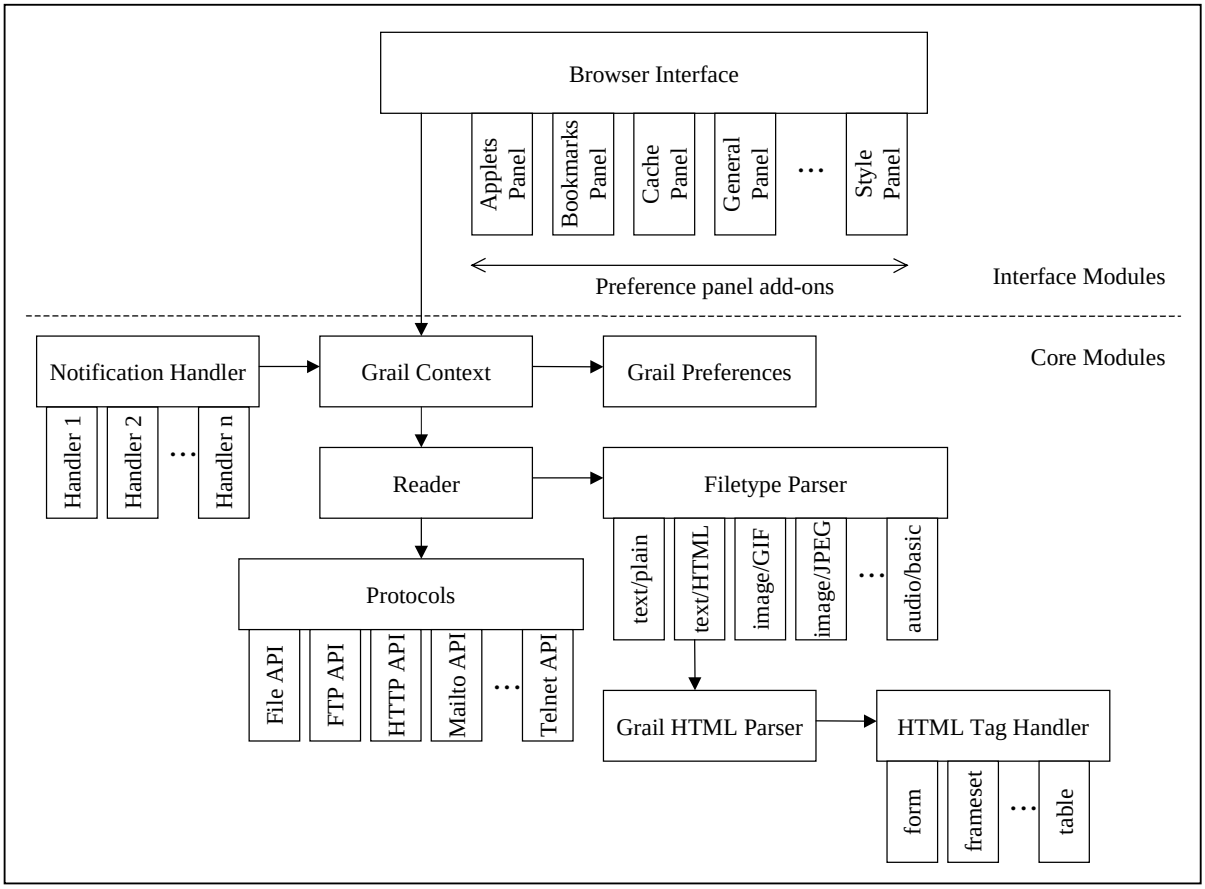

Figure 5.9. Modified Grail architecture

Changes required to support two specific types of adaptation will now be considered. Adaptation to changing display type and network resources have been chosen as interesting examples.

5.4.2.1 Adapting Grail to a changing user display

The changes that are required in order to allow Grail to support adaptation to a changing user display will depend on the adaptation policies that are employed.

Simple policies that involve changing only the types of settings that can currently be manipulated by the user, such font size and window size, require no changes to Grail other than those already discussed.

Policies that manipulate aspects of the user interface that are not currently controllable by the user will require modifications to the browser interface modules.

Policies that dictate changes to the way in which pages are displayed will require changes to the parsing modules for the file types involved.

Finally, policies that dictate adaptation of user downloads to the display type will require changes to the HTTP module, which is responsible for performing downloads using the HTTP protocol.

None of these changes will affect the overall architecture of the browser.

5.4.2.2 Adapting Grail to changing network resources

In order to adapt to changing network resources, the browser must implement flexible download policies. The download mechanisms in Grail are isolated within protocol plug-ins. Each plug-in determines the procedures followed for all downloads that use a particular protocol, such as HTTP or FTP.

Support for adaptation to changes in network resources can be incorporated into Grail without modification to the architecture. New download procedures can be easily added to Grail as protocol plug-ins. For instance, suppose that an adaptation policy is used that dictates that the browser must request the most compact representations of downloaded objects when bandwidth is scarce. HTTP permits this using content negotiation. The policy could be implemented by a HTTP plug-in that has knowledge about the current bandwidth availability, and negotiates the content of downloads according to this information.

5.4.3 Summary

This section first discussed a model for supporting application-controlled adaptation in general, and next addressed required modifications to Grail's architecture that arise from the use of the model. It concluded with an evaluation of the changes required to the browser's architecture to support two specific classes of adaptation. The results of this evaluation were positive; in both cases, the architecture required no additions other than callback handlers. This is a testimony to Grail's success as an extensible Web browser.

5.5 Web server design

This section considers the construction of a HTTP server that supports adaptation of downloads from the server to the current environment.

The adaptation of downloads to user requirements is currently possible in HTTP using variants. The algorithm discussed in Section 5.1.2 performs adaptation of downloads to user preferences relating to attributes such as media-type, language and character set. This section describes modifications to the algorithm that incorporate QoS information into the selection of variants.

A HTTP request header field can be used to convey information about the client's QoS in request messages. A new header field designed to suit this purpose is described in the first part of this section.

5.5.1 QoS header field

The QoS header field conveys information about the status of arbitrary resources. The header consists of a number of status fields separated by commas. Each status field includes the name of the resource type followed by a variable number of values relating to the resource.

For example, the following is a legal QoS header field:

QoS: Bandwidth high, Display small

This header field indicates that the client's bandwidth resources are high and the display is small.

5.5.2 Quality value computation

The computation of variant quality values can be modified to take into account the status of each type of resource listed in the QoS header. This can be achieved by allowing each resource type to contribute a factor to the computation. These factors can be combined with the existing factors using multiplication.

The computation of resource factors should be dependent on the resource types involved. The remainder of this section describes the computation of factors for the bandwidth and display resources.

It should be noted that the formulae used in the computation of factors described here have been chosen solely for illustration purposes. In practice, the methods used to assign factors to variants should be verified through experimentation to determine their effectiveness before they are adopted. In particular, the interplay that occurs between different factors is complex, and it should be ensured that the outcome of combining a number of factors is appropriate.

5.5.2.1 Bandwidth factors

This section illustrates the allocation of bandwidth factors to variants. For the sake of simplicity, this discussion assumes that three bandwidth levels can be specified by clients: high, medium and low.

When the client's bandwidth is high, the variant selection process should return the variant that best matches the user's preferences, regardless of the size of the variants. That is, the bandwidth factor should not influence the selection of variants. Therefore, the bandwidth factor allocated to all variants when the bandwidth is high is one.

Conversely, when bandwidth is low, preference should be given to the variants that are the smallest. This can be achieved by allocating a bandwidth factor to each variant that is inversely proportional to its size. The following computation ensures that all variants are allocated factors in the range zero to one, with the smallest of the variants being allocated a value of one.

$$qb = \frac{\text{size of smallest variant}}{\text{size of variant}}$$

When the client has a medium level of bandwidth, it may be appropriate to give preference to smaller variants. However, this preference should be less pronounced, as the minimisation of bandwidth usage is less critical in this case than when bandwidth is extremely scarce. A reduction in the spread of bandwidth factors can reduce the effect of the bandwidth factor on the overall quality values. The following formula computes a value for each variant that lies between $^2/_3$ and 1.

$$qb = \frac{2}{3} + \frac{\text{size of smallest variant}}{3 * \text{size of variant}}$$

If no bandwidth field is present in the user's request, or no size is available for a particular variant, a default value of one should be used.

5.5.2.2 Display factors

This section demonstrates the allocation of display factors to variants, assuming three user display types: small, medium and large.

The size of downloads can be matched to the user's display size by assigning each variant a display factor that indicates the suitability of the variant for the display size.

When the user display is large, the choice of variants need not take into account the dimensions of variants, because all should be suitable for the particular display type. Therefore, in this case, all variants are allocated a display factor of one.

Conversely, when the user display is small, variants with smaller dimensions should be given preference over those with larger ones. This can be done by allocating each variant a display factor that is inversely proportional to the dimensions. The following formula allocates each variant a factor between zero and one, with the variants of smallest overall area being assigned a factor of one:

$$qd = \frac{\text{smallest variant area}}{\text{variant area}}$$

The area of a variant is computed as the product of the variant's height and width, which are typically listed in pixels.

When the display is of a medium size, it may be appropriate to give some preference to variants with smaller dimensions. However, the dimensions of variants should have a smaller influence on the selection of variants than in the previous case; to achieve this, there should be a smaller degree of variance in display factors. The following formula assigns display factor values in the range 0.75 to 1 to variants:

$$qd = \frac{3}{4} + \frac{\text{smallest variant area}}{4 * \text{variant area}}$$

When no display field is supplied by the user, or the area of a variant is not known, a default value of one should be used.

5.5.3 Modifications to the selection algorithm

This section describes an alteration that is made to the selection algorithm in addition to the modification of the variant quality calculation.

The variant selection algorithm described in section 5.1.2 makes a distinction between definite and indefinite values. When the value of the best variant is indefinite, the server responds with a list of variants, and the user agent is required to perform the selection. This approach requires that the user agent be capable of performing a variant selection process.

In order to remove this requirement, and reduce the number of messages exchanged between the client and server, the modified algorithm ignores the distinction between definite and indefinite quality values. This modification has the advantage of simplifying the selection algorithm substantially.

In order to permit variant selection by the client, the Alternates header can be retained, and returned to the client in addition to the selected variant. This header can be ignored by clients that do not support variant selection, and exploited by those that do in order to allow sub-optimal choices to be rectified.

5.5.4 Variant descriptions

The specification of RVSA/1.0 indicates that one of the inputs to the algorithm is information concerning the file variants. It does not specify the format of this information. This section describes a method for associating variant information with particular resources, based on an approach used by the Apache Web server[3].

Apache Version 1.3 uses a type map to describe the different variants of a resource. The description of a variant consists of a number of header records that are of the same format as HTTP header fields. Each record consists of a name, followed by a colon and a value for that name. The following types of header record are permitted:

- *Content-Type*. Indicates the media type of the variant, such as text/plain or image/jpeg.
- *Content-Encoding*. Describes type of encoding applied to the variant, such as gzip.

[3] http://www.apache.org/httpd.html

- *Content-Language*. Describes the languages associated with the variant, such as en (English) or fr (French).

- *Content-Length*. Indicates the length of the variant.

- *URI*. Indicates the resource locator associated with the variant.

Variant descriptions are separated by blank lines.

The following extension headers can be added to support the description of the dimensions of a variant:

- *Height*

- *Width*

These headers specify the height and width of a variant in pixels.

A variant description can be associated with a resource by storing it in a file named URI.var, where URI is the resource indicator for the resource.

5.6 Summary

This chapter considered adaptability in the context of a Web application. It defined the Web application, and introduced HTTP, a protocol commonly used by Web applications.

The chapter also introduced an example Web client, the Grail browser. The architecture of this browser was studied in detail, with particular emphasis on the features provided for extensibility. The resources required by the browser were also examined with a view to determining which types of adaptability might be appropriate. Adaptability mechanisms for responding to changes in the following resources were considered:

- CPU and primary memory

- secondary memory

- network resources

- user display

The next part of the chapter considered how some of these types of adaptability could be incorporated into Grail. First, a mechanism for obtaining information about the state of resources was described. Next, changes to the browser's architecture to enable support for adaptability were outlined.

Finally, modifications to the HTTP variant selection algorithm to allow QoS information to play a part in the selection of variants were proposed.

Chapter 6. Web application implementation

This chapter describes an implementation of the Web application described in the preceding chapter.

The implementation is written in Python[4] and uses the Fnorb CORBA ORB[5] for communication between the browser and the QoS Monitor.

The QoS Monitor has been implemented to allow the user to simulate various environmental conditions. The Monitor permits the user to set the levels of QoS that are communicated to the Monitor's client.

A number of modifications to Grail to incorporate adaptation have also been implemented. The extensions implement adaptation to changing bandwidth availability and display type.

The HTTP server described in Chapter 4 has also been implemented, however this work was performed in the context of a project for Advanced Networks and Communication (CS433). The implementation is therefore described in a separate report.

6.1 Implementation of the QoSM simulator

The QoS Monitor implements the register interface specified in Appendix A. This interface allows the client to register interest in resources and receive callbacks when the resource levels change in a way that is considered significant by the client.

The implementation of the monitor provides a simulation of a true resource monitor, and obtains resource information from the user rather than through system monitoring. This facilitates easy testing of the adaptive application, as it removes the necessity of placing the system into the states desired for testing. The inputs to the monitor are obtained from the user through a simple graphical user interface (GUI).

The monitor implementation consists of three core modules and a number of plug-ins. A listing of these modules appears in Appendix B.

---

[4] http://www.python.org/

[5] http://www.dstc.edu.au/Products/Fnorb/

6.1.1 Core modules

The first of the core modules is the driver module, `QoSMonitor`, which implements the main procedure. This procedure is responsible for creating the implementation of the `RegisterIF` interface and registering it with the CORBA Basic Object Adaptor (BOA), which is the CORBA object responsible for managing object references and the dispatching of requests to target objects.

The main procedure also creates the QoSM user interface and starts the event loop, which processes incoming service requests from clients as well as user interface events.

The `Resource` module implements a generic resource type, providing the base classes from which resource plug-in implementations should derive. The module is made up of two classes. The `Resource` class provides an abstraction of a resource monitoring function. It defines the `register` operation, which allows the user predicates to be defined. These are used by resource monitor to determine when the client is notified of changes in the resource. They are evaluated at the time the predicates are first registered, and again each time the status of the resource changes.

The second component of the `Resource` module is the `ResourceInterface` class. This class implements a user interface component, which is displayed in the main QoSM window. The role of this component is to present the user with information regarding the status of the resource. The default implementation provided by the class creates an empty frame; subclasses must override this behaviour by implementing their own versions of the `createWidgets` method. The method should create all of the user interface widgets that make up the display.

Finally, the `Predicates` module defines classes for storing and evaluating predicates. The `Predicates` class maintains a set of predicates relating to a particular resource, and provides three operations. New predicates are added to the set using the `add` method, while `deleteAll` empties the set of predicates. Finally, `notify` is used to evaluate the entire set of predicates using the current resource value supplied as the parameter, and to notify clients about predicates that evaluate to true.

The second class implemented by the `Predicates` module is the `GeneralPredicate` class, which defines a general predicate type. The class is abstract, and forms the base from which the concrete predicate classes derive. It provides two methods. The `evaluate` function determines the truth value of the predicate. This is achieved using `compare`, a function that defines the type of comparison that is used in the evaluation of the predicate.

Five subclasses of `GeneralPredicate` are implemented, corresponding to the five predicate types define by the `RegisterIF` interface. Each of the subclasses provides its own version of the comparison function, according to the type of the predicate.

6.1.2 Resource type plug-ins

In addition to the core modules, two plug-in modules are defined.

The `Bandwidth` plug-in is intended to simulate a monitor of network bandwidth. It obtains the current bandwidth value from the user via an entry box in the user interface. Bandwidth values are interpreted as numbers of bits per second.

The Display plug-in provides the simulation of a monitor of user display type. The values used in the simulation are determined by the user through the QoS GUI. Radio buttons allow the user to select from 320 × 200, 640 × 480 and 800 × 600 displays. The display types have been chosen arbitrarily for the simulation, and all display sizes are measured in pixels.

Figure 6.1 illustrates the interface presented by the monitor when both bandwidth and display type plug-ins are loaded.

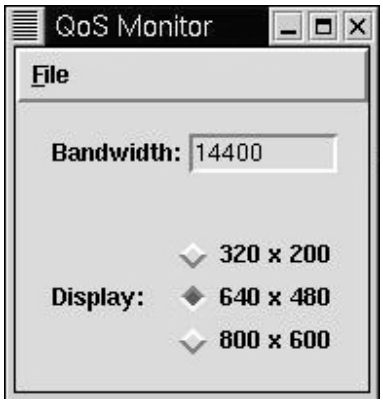

Figure 6.1. The QoSM user interface

6.2 Implementation of the Grail extensions

The extensions to Grail implement adaptation mechanisms for response to changing bandwidth and user display type.

The policies implemented by the adaptation mechanisms are described in the first part of this section.

6.2.1 Adaptation policies

The extended version of Grail provides a number of adaptation policies that can be selected by the user. There are two different policies for adaptation to bandwidth availability, and three policies for adaptation to display type.

6.2.1.1 Adaptation to bandwidth availability

Grail's adaptation to bandwidth availability uses the HTTP extensions defined in Chapter 5. The QoS header is used to convey information about the level of bandwidth available to Grail to the HTTP server. This information can be used by the server to tailor the selection of variants to the current state of the environment.

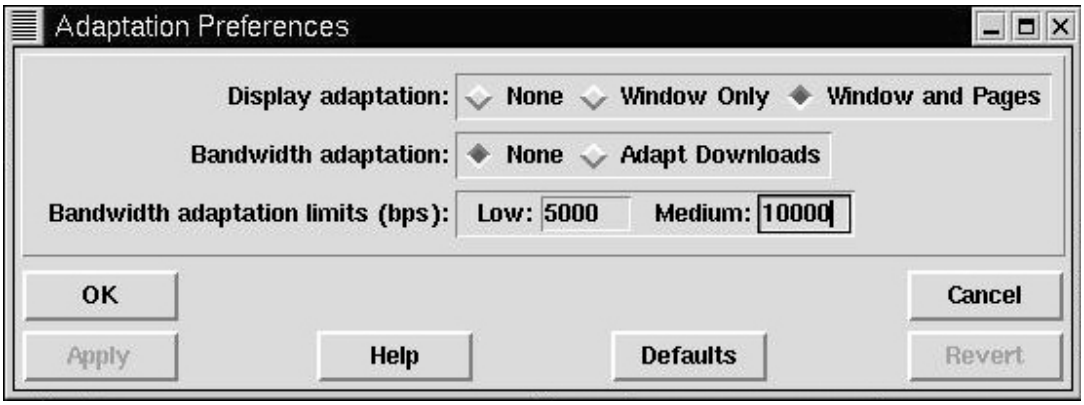

Figure 6.2. Adaptation preference panel

Two simple adaptation policies are possible. The first policy dictates that no adaptation of the browser should occur. In this case, the QoS header is omitted from HTTP requests, and the browser accepts the default file variants selected by the server. The second adaptation policy dictates that the QoS header should be used to indicate the bandwidth QoS to the server. Servers that support this header should respond with a file variant that is appropriate for the bandwidth level.

The user is able to determine the bandwidth ranges that correspond to high, medium and low levels of bandwidth. These levels, as well as the adaptation policy, can be set using the adaptation preference panel, shown in Figure 6.2.

6.2.1.2 Adaptation to display type

Three types of adaptation to user display type are defined. The user can select a policy using the preference panel in Figure 6.2. The first policy corresponds to no adaptation, the second to adaptation of the browser window only, and the third to adaptation of the browser window and the Web pages displayed in the window.

Adaptation of the browser window consists of modification of window fonts according to current display type.

Adaptation of Web pages involves the adaptation of fonts and images to the display type.

Three types of display are considered by the browser. These have been chosen arbitrarily to demonstrate the concepts. In reality, the browser has not been tested on all of these types of monitors. All tests have been carried out on a single machine with 1024×768 resolution.

A set of fonts is associated with each of the three display types. The particular fonts used by the browser window and Web pages are adapted to match the display type whenever a change in the display occurs, in accordance with the adaptation policy. The fonts comprising each set are specified in the Grail preference file, allowing easy modification.

When adaptation of Web pages is enabled, the displaying of images is adapted to the display type according to a simple rule. When the display type is 320×200, the loading of images is automatically disabled. It can be re-enabled using Grail's general preference panel if desired. The loading of images is enabled when the display changes to a higher resolution.

In addition, the adaptation of Web pages involves adaptation of Grail's downloads from the HTTP server. The QoS header is used to convey information regarding the display to the HTTP server, which can use this information in the selection of a variant that is appropriate for the display type.

6.2.2 Implementation

The policies are implemented through the addition of several modules and the modification of a number of the original Grail modules. A listing of new modules is provided in Appendix C, while Appendix D describes the changed modules.

The changes to Grail can be divided into three distinct categories. The first category of changes implements general support for adaptation, while the remaining two categories include the changes that implement policies for adaptation to bandwidth availability and display type.

6.2.2.1 Support for adaptation

The modified browser incorporates an infrastructure that is designed to support the implementation of the policies outlined in the previous section, as well as the implementation of arbitrary types of adaptation that might be incorporated into Grail in the future. The infrastructure comprises two main components: callback mechanisms and a repository for the maintenance of QoS information.

The callback mechanism allows the browser to receive notification of QoS changes. Two modules implement the callback infrastructure.

The `Adaptation` module implements the `AdaptationCallbacks` class, which is responsible for initialising the CORBA BOA and obtaining a reference to the QoS Monitor. The reference is obtained using the `QOS_MONITOR` environment variable, which contains the path of a file containing the server's IOR (Interoperable Object Reference). While this approach has

limitations, as it relies on Grail and the QoSM sharing a common file system, it is simple and sufficient for a prototype browser. The limitation can be removed by the use of the Fnorb name server to locate the Monitor implementation.

The `AdaptationCallbacks` class is also responsible for the creation of all of Grail's callback handlers.

Two methods are supplied by the AdaptationCallbacks class. The first returns a reference to the BOA, while the other closes down the callback handlers upon Grail's termination.

The `CallbackHandler` module also forms part of the adaptation infrastructure. It provides a generic callback handler that forms the base class for all of Grail's callback handler implementations. Each of the resources monitored by the QoS Monitor on behalf of Grail has a corresponding callback handler.

The `CallbackHandler` class implements `ChangeNotificationIF`, which provides the `Notify` method, used by the QoSM to notify the handler of changes in the resource levels. The class' initialisation method registers the callback implementation with the CORBA BOA, and the `Register` method registers the list of predicates for the resource with the QoSM using the monitor's own `Register` method. Both `Register` and `Notify` must be implemented by subclasses of `CallbackHandler`, as the implementations of these methods must be tailored to the particular resources concerned.

The third component of the adaptation infrastructure is the `QualityOfService` module. This module provides a repository for QoS values, allowing arbitrary Grail modules to obtain information about the current environment. The repository is implemented using a Python dictionary, in which the QoS attribute names form the set of keys, and the current QoS levels form the dictionary values.

The repository is currently limited, as it permits only the setting and retrieval of QoS values, and does not allow the association of callbacks with particular QoS values. The applicability of callback mechanisms in Grail has been demonstrated by the preference facility, which permits callback functions to be associated with preference groups. The preference repository automatically invokes the functions associated with a group whenever one of the group's values is altered. This type of callback is already widely used in Grail to allow the browser to respond to preference changes. A QoS callback mechanism could be used to allow automatic response to QoS changes.

Some simple modifications to the existing `grail` and `app` modules have been necessary. Additional code has been added to the modules to include the creation of `AdaptationCallbacks` and QoS repository instances. A number of small changes have also been required to accommodate the use of Fnorb. The Tk event loop that was used to process user interface events has been replaced by a Fnorb event loop that processes both these events and

Fnorb events. In addition, Grail's `quit` method has been modified so that the Fnorb event loop is stopped and the callback handlers that process notifications from the QoSM are disabled.

The adaptation infrastructure described in this section currently supports two classes of adaptation mechanism, which are described next.

6.2.2.3 Adaptation to bandwidth availability

The monitoring of bandwidth within Grail using the QoSM is performed by the `BandwidthHandler` module. The module implements a callback handler that is responsible for updating the bandwidth attribute of the QoS dictionary in response to bandwidth changes.

The bandwidth level information is used by the HTTP plug-in, which implements a HTTP client responsible for carrying out communication with HTTP servers.

Extensions have been made to the HTTP client's `open` method to incorporate the HTTP extensions that were described in Section 5.5. When the adaptation of bandwidth is enabled, the HTTP client constructs a QoS header field that conveys the bandwidth information to the server. This information can be used by the server to adapt downloads to the QoS levels.

6.2.2.4 Adaptation to display type

The tracking of the user display type within Grail is performed by the `DisplayHandler` module. This module receives display change notifications from the QoSM, and updates information about the display type and current fonts that is stored in the QoS and preference repositories.

Two types of changes to the Grail interface modules have been made to support adaptation of the browser window to the current display. First, the creation of user interface widgets has been modified to use information about the current font settings. In addition, many of the user interface classes have been modified to respond dynamically to changes in font settings. The modules that support this dynamic adaptation all provide an `update_styles` method, which is automatically invoked by the preference callback mechanism when the browser styles are changed. The method updates the font styles used by the interface component to match the new font settings.

While many of the interface modules used by Grail have been modified to support adaptation to display type, there remain user interface components that do not yet support this adaptation. In particular, the interface components provided by the Tk library, such as file selection and warning dialogs, do not allow the manipulation of font types. In order to provide uniform adaptation of the interface, these modules will need to be completely re-implemented to support adaptation.

The adaptation of Web pages is implemented by the `httpAPI` and `DisplayHandler` modules.

The enabling and disabling of images is achieved trivially within the display callback method implemented by `DisplayHandler`. The callback handler simply modifies the Grail preference that controls the loading of images, and pre-existing Grail code uses this information to adapt the downloading of images.

The `httpAPI` plug-in module implements support for the adaptation of downloads to the display type. When the adaptation of Web pages is enabled, the plug-in's `open` method conveys display information to the HTTP server in request messages, using the QoS header. This display information can be used by the server to adapt downloads to the client's QoS levels.

6.3 Summary

This chapter has described implementations of the QoS Monitor and adaptive browser described in the Chapter 5.

The QoS Monitor was implemented as a simulation of resource monitor that obtained resource information from the user.

The extensions to Grail consisted of an infrastructure supporting adaptation, and the mechanisms for two particular types of adaptation. Five new modules were added, and a number of modifications were made to existing modules. The fact that most of the extensions were minor has lent support to the claim made in Chapter 5 that Grail's architecture is successful in supporting extensibility.

Chapter 7. Adaptation examples

This chapter illustrates the browser's adaptation mechanisms using a simple example. The example focuses on a Web page containing an image that can be obtained in several different versions that differ in their levels of fidelity. The image variants are described in the following section.

The original Web page, which has been adapted for use in this chapter, can be found at http://reseau.chebucto.ns.ca/Environment/NHR/lepidoptera.html.

7.1 Image variants

Four image variants are used in the example. These vary in the level of JPEG compression used, number of dots per inch (dpi) and size. The variants are shown in Figures 7.2 to 7.5.

Variants 2 to 4 were created using Paint Shop Pro version 4.10. A JPEG compression level of 90 was used in the creation of variant 2, while variants 3 and 4 used a compression level of 95.

A variant description file provides the information that enables the HTTP server to select the variant that is most suited to the client's current QoS levels. This file is shown in Figure 7.1.

```
URI: Variant1.jpg
Content-Type: image/jpeg; q = 1
Content-Length: 16293
Height: 225
Width: 244

URI: Variant2.jpg
Content-Type: image/jpeg; q = 0.9
Content-Length: 3536
Height: 225
Width: 244

URI: Variant3.jpg
Content-Type: image/jpeg; q = 0.5
Content-Length: 2584
Height: 225
Width: 244

URI: Variant4.jpg
Content-Type: image/jpeg; q = 0.4
Content-Length: 1507
Height: 138
Width: 150
```

Figure 7.1. Variant description file

Figure 7.2. Image variant 1 - Original butterfly image at 16293 bytes

Figure 7.3. Image variant 2 - Compressed to 3536 bytes using JPEG compression

Figure 7.4. Image variant 3 - Compressed to 2584 bytes using JPEG compression

Figure 7.5. Image variant 4 - Size reduced and compressed to 1507 bytes using JPEG compression

The q-values in the Content-Type lines of the variant descriptions express the quality or fidelity of the images. In this example, the values have been chosen subjectively, and are intended to reflect a typical user's level of satisfaction with the variant. The range of possible values extends from zero to one, with a value of zero expressing complete dissatisfaction of the user, and a value of one expressing complete satisfaction. The quality values decrease with increasing levels of lossy JPEG compression and with decreasing image size.

The interpretations of other fields of the variant description file were discussed in Section 5.5.4.

7.2 Adaptation to bandwidth availability

The image discussed in the previous section forms part of a Web page on Lepidoptera, shown in Figure 7.6. In this section, the adaptation to bandwidth availability performed by Grail is illustrated using the page as an example.

Grail attempts to match the bandwidth consumed by user downloads to its available bandwidth by notifying the HTTP server of its current bandwidth level in its request messages. A QoS-aware server is expected to use this information to adapt its responses accordingly. The design for such a server was discussed in section 5.5. This section presents results obtained using a server implemented to this design.

Figure 7.6. The browser window adapted for high bandwidth and a medium-sized display

Variants are selected by the server using quality values computed using of a number of factors. The bandwidth factors for each of the image variants, calculated according to the formulae described in section 5.5, are listed below.

Image Variant	*High bandwidth factor*	*Medium bandwidth factor*	*Low bandwidth factor*
Variant 1	1	0.697	0.092
Variant 2	1	0.809	0.426
Variant 3	1	0.861	0.583
Variant 4	1	1	1

Each of the overall quality values, computed as the product of the variant quality factor and the bandwidth factor, is provided in the table below.

Image Variant	*High bandwidth quality*	*Medium bandwidth quality*	*Low bandwidth quality*
Variant 1	1	0.697	0.092
Variant 2	0.9	0.728	0.3834
Variant 3	0.5	0.431	0.2915
Variant 4	0.4	0.4	0.4

The variant selected in each case is the one with highest overall quality. In this example, variants 1, 2 and 4 are selected in circumstances of high, medium and low bandwidth availability, respectively, assuming display type is not taken into account in the selection. The first case is illustrated in Figure 7.6.

The effect of display type on the variant selection process is discussed in the following section.

7.3 Adaptation to display type

This section demonstrates the adaptation of Grail to changing display type, and again uses the Lepidoptera page by way of illustration.

The adaptation of the browser to display type involves three aspects: adaptation of the browser window, adaptation of pages displayed in the browser window, and adaptation of downloads. Each of these types of adaptation can be enabled or disabled according the user's preferences.

Adaptation of the browser window involves the matching of window fonts to the current display type. The font type and size used for each display type can be determined by the user through an entry in the Grail preference file.

The adaptation of pages displayed by the browser involves the matching of font sets to the bandwidth availability and the disabling of images when the display is small. These attributes can also be directly manipulated by the user through the preference panels.

Finally, Grail downloads are adapted by including display type information in HTTP requests. The design of a server that uses display type information in the selection of file variants was

discussed in Section 5.5. This discussion presents the results that are obtained when a server implemented according to this design is used.

The display factors for each of the image variants, calculated according to the formulae described in section 5.5.2, are listed below.

Image Variant	*Large display factor*	*Medium display factor*	*Small display factor*
Variant 1	1	0.844	0.377
Variant 2	1	0.844	0.377
Variant 3	1	0.844	0.377
Variant 4	1	1	1

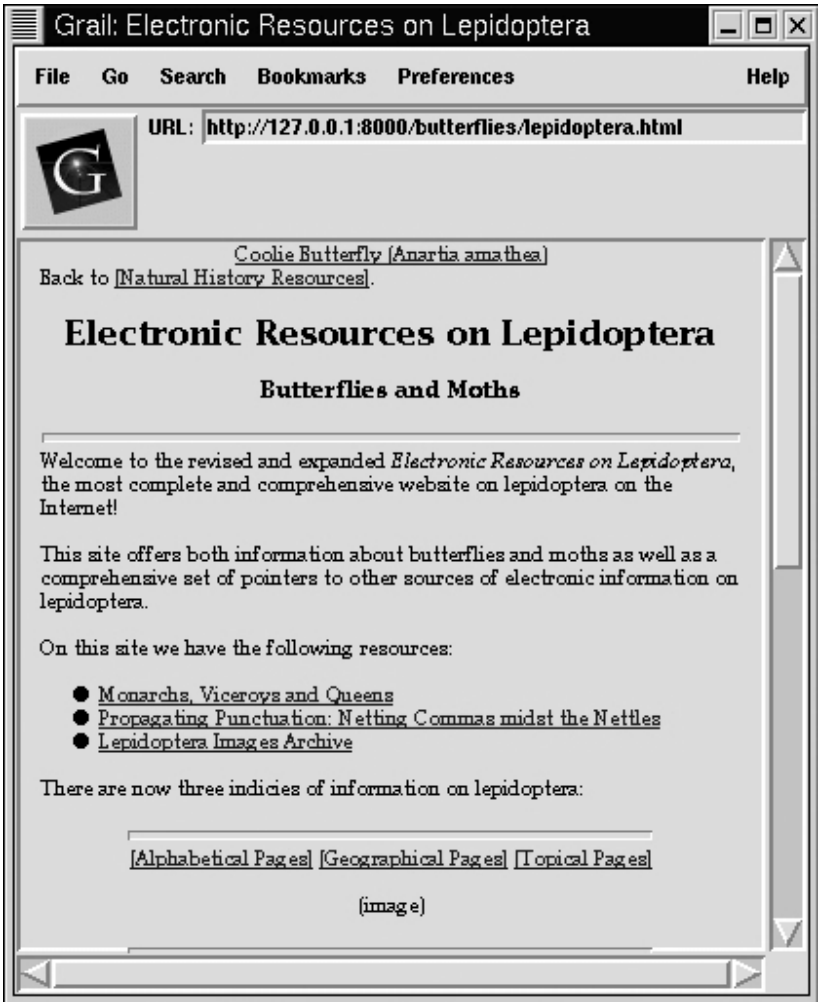

Figure 7.7. The browser window adapted for a small display

The overall quality values, computed as the product of the variant quality factor and the display factor, are provided in the table below.

Image Variant	*Large display factor*	*Medium display factor*	*Small display factor*
Variant 1	1	0.844	0.377
Variant 2	0.9	0.760	0.339
Variant 3	0.5	0.422	0.1885
Variant 4	0.4	0.4	0.4

Assuming no other factors, such as bandwidth factors, come into play, Variant 1 is selected when the display is of large or medium size, and Variant 4 is selected when the display is small.

Figure 7.6. The browser window adapted for high bandwidth and a large display

When there are other factors to take into consideration, these are multiplied by the quality values listed in the above table, and the variant that is selected is the one with highest overall value.

Figures 7.7, 7.6 and 7.8 show the results of applying all three aspects of display adaptation in situations in which the user display is small, medium and large, respectively. Note that all browser windows are sized according to the same scale.

7.4 Summary

This chapter has demonstrated the adaptation of Grail to both bandwidth availability and display type, using an example Web page. The example page contained an image that was available as several different variants, allowing the HTTP server's responses to be matched to the client's particular requirements. The variant selection process described in Section 5.5, involving the calculation of quality values for each variant, was demonstrated using the example.

While the chapter has addressed only image variants and the two types of adaptation currently provided by Grail, the principles are more general. The variant selection algorithm can be applied to variants of any file type and can be extended to incorporate types of QoS factor other than bandwidth and display factors.

Chapter 8. Conclusions

This report has motivated the need for adaptation in the context of mobility, and addressed means by which distributed systems can provide support for adaptive applications. It has presented existing systems that support adaptability, and developed an adaptive application and a simple infrastructure for supporting adaptation.

Chapter 2 described the two elements that are essential for adaptation, and discussed how these can be provided in a distributed system environment. In addition, it presented criteria for determining a distributed system's level of support for adaptation. These criteria addressed the support of the system for QoS description, adaptation mechanisms, and types of heterogeneity.

In Chapter 3, these criteria were applied to a set of distributed systems that have been described by recent literature. The architectures displayed a range of different solutions, and demonstrated a requirement for future work in a number of areas. In particular, the architectures were found to provide little support for the ilities, such as reliability, security and availability. In addition, most of the distributed systems displayed a lack of support for complex interaction types between distributed system components. In some cases, this limitation was imposed by the implementation, rather than the design of the system.

Chapter 4 addressed the differences between ODP and CORBA in the modeling of distributed applications, and attempted to identify the impact of the differences on types of adaptation that can be supported.

The remainder of the report has described an adaptive Web application.

Chapter 5 presented the design for an adaptive browser and supporting HTTP server, as well as a resource monitor used to trigger adaptation within the browser.

The Web browser design is based on that of Grail, an extensible browser developed using Python. The design of Grail was analysed, and extensions enabling adaptation were outlined. The resource monitor was defined as a CORBA object allowing the client to register interest in resources, and receive callbacks when significant changes in those resources occur. Finally, the server design outlined a method for adapting downloads according to QoS information supplied by the client in requests. This method was based on the Remove Variant Selection Algorithm described by RFC 2296 (RVSA/1.0).

Chapter 6 described the implementation of the browser and resource monitor, and Chapter 7 illustrated the adaptation policies implemented by the browser using an example Web page.

The application and supporting infrastructure currently exhibit limitations that could be addressed by future work. In particular, the infrastructure is limited in its support for QoS, and provides no direct support for object mobility.

While support for QoS monitoring is provided, QoS negotiation and resource management are not. This means that while QoS can be tracked by the system, it cannot be controlled. This may make it unsuitable for applications with critical resource requirements. The nomadic computing architecture and ODP-based adaptive management architecture, described in Chapter 3, both solved this problem, and demonstrated how the allocation of resources could be controlled by a resource management function.

The mobility of objects, such as applications and users, is also neglected by the infrastructure. This limitation can be eliminated by providing mechanisms for migrating applications and maintaining mobility profiles that can be used in the management of mobile objects. This approach is adopted by the nomadic computing architecture, described in Section 3.1.6.

In addition, the browser currently supports a very limited set of adaptation mechanisms. These address changes in bandwidth and display type. Other types of adaptation could be incorporated in the future, such as adaptation to available memory, disk space and processing power.

References

[1] Frank Manola. "Providing Systemic Properties (Ilities) and Quality of Service in Component-Based Systems." (Draft) Object Services and Consulting, Inc., February 1999. http://www.objs.com/aits/9901-iquos.html

[2] B. McClure, J.Indulska and S. Crawley. "Adaptive Management of Network Resources in Heterogeneous Defense Networks." *Proceedings 9th IFIP/IEEE International Workshop on Distributed Systems Operations and Management*, pp. 25 – 38. Newark, USA, October 1998.

[3] Nigel Davies, Adrian Friday, Stephen P. Wade and Gordon S. Blair. "L^{2}imbo: A distributed systems platform for mobile computing." *ACM Mobile Networks and Applications (MONET)*, Special Issue on Protocols and Software Paradigms of Mobile Networks, Volume 3, number 2, pp 143-156. August 1998.

[4] F. Garcia, D. Hutchison, A. Mauthe and N. Yeadon. "QoS support for distributed multimedia communications." *Proceedings of the 1st International Conference on Distributed Platforms*. Dresden, Germany, 27 February - 1 March 1996.

[5] M. Satyanarayanan. "Coda: A Highly Available File System for a Distributed Workstation Environment." School of Computer Science, Carnegie Mellon University. *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*. Pacific Grove, CA, September 1989.

[6] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn and Kevin R. Walker. "Agile Application-Aware Adaptation for Mobility." *Proceedings of the 16th ACM Symposium on Operating System Principles*. St. Malo, France, October 1997.

[7] Brian D. Noble, Morgan Price and M. Satyanarayanan. "A Programming Interface for Application-Aware Adaptation in Mobile Computing." *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*. Ann Arbor, MI, April 1995.

[8] Oguz Angin, Andrew T. Campbell, Michael E. Kounavis and Raymond R.-F. Liao. "The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking." *IEEE Personal Communications Magazine*, Special Issue on Adapting to Network and Client Variability, Volume 5, number 4, pp 32-44. August 1998.

[9] Sun Microsystems. "Jini Architectural Overview." Technical White Paper. http://www.sun.com/jini/whitepapers/architecture.html

[10] Jadwiga Indulska, Andy Bond and Matthew Gallagher. “Support for Mobile Computing in Open Distributed Systems.” *Proceedings IEEE Region Ten Conference*, pp 3 – 6. Dehli, India, December 1998.

[11] Bill N. Schilit, Norman Adams and Roy Want. “Context-Aware Computing Applications.” *IEEE Workshop on Mobile Computing Systems and Applications*, December 8-9, 1994.

[12] Bill N. Schilit, Marvin M. Theimer and Brent B. Welch. “Customizing Mobile Applications.” *Proceedings USENIX Symposium on Mobile and Location-Independent Computing*, August 1993.

[13] ITU-T Recommendation X.P03. “ODP Reference Model Part 3, Architecture.” January 1995.

[14] The Object Management Group. "CORBA Components - Joint Revised Submission." OMG TC document orbos/99-02-05. March 1, 1999.

[15] Guido van Rossum. "Grail -- The Browser for the rest of us." (Draft) Corporation for National Research Initiatives. May 30, 1996. http://grail.cnri.reston.va.us/grail/info/papers/restofus.html

[16] The Internet Society, Network Working Group. "RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1" June 1999.

[17] The Internet Society, Network Working Group. "RFC 2295: Transparent Content Negotiation in HTTP." March 1998.

[18] The Internet Society, Network Working Group. "RFC 2296: HTTP Remote Variant Selection Algorithm -- RVSA/1.0." March 1998.

Appendix A. CORBA IDL

```
//
// Notification.idl
//
// This module describes interfaces used for communication between a
// QoS Monitor and its client.
//

module Notification
{
   interface ChangeNotificationIF;

   interface RegisterIF
   // Interface provided by the QoS Monitor to allow the client
   // application to specify its interest in particular resources.
   {
      typedef string ResourceType;
      typedef string ResourceValue;
      typedef enum CompOp {greater, less, equal, less_or_equal,
         greater_or_equal} ComparisonOp;
      typedef struct Pred
      {
         ComparisonOp op;
         ResourceValue value;
         ChangeNotificationIF notificationIF;
      } Predicate;
      typedef sequence<Predicate> Predicates;

      exception InvalidResourceType {};
      exception ResourceNotRegistered {};

      void Register(in ResourceType resource, in Predicates preds)
         raises(InvalidResourceType);
      void RemoveRegistration(in ResourceType resource)
         raises(ResourceNotRegistered);
   };

   interface ChangeNotificationIF
   // Interface implemented by client applications, and used by the
   // QoS Monitor to notify the client of changes to resources.
   {
      void Notify(in RegisterIF::ResourceValue value);
   };
};
```

Appendix B. Implementation of the QoSM simulator

QoSMonitor.py

```
"""
QoSMonitor.py

This module implements the driver for the QoS Monitor simulator.

The Register class implements the RegisterIF provided to clients, as
defined by Notification.idl.

The QoSMonitorInterface class implements a simple interface that
display the resource types that are currently being monitored for the
client.

The QoSMonitor class provides code for setting up the QoSMonitor.  It
registers the implementation with the BOA, creates the interface, and
starts the event loop.  The IOR of the implementation is written to a
file named server.ref.
"""

import sys
from Fnorb.orb import BOA, CORBA, TkReactor
import Notification, Notification_skel
import resources
from Tkinter import *

class Register(Notification_skel.RegisterIF_skel):
    def __init__(self, masterWindow):
        self.resources = {}
        self.masterWindow = masterWindow

    def Register(self, resource, preds):
        """
        Register interest in a resource.  The list of predicates,
        preds, determines when the client will be notified of
        changes in the resource.  The list of predicates
        completely replaces any predicates that might have been
        specified earlier for that resource.

        If the resource named by the client is not supported by
        the monitor, the InvalidResourceType exception will be
        raised.
        """
        if self.resources.has_key(resource):
```

```
            instance = self.resources[resource]
        else:
            statement = "import resources." + resource
            exec(statement)
            statement = "resources." + resource + "." \
                + resource + "()"
            instance = eval(statement)
            instance.setMaster(self.masterWindow)
            self.resources[resource] = instance
        instance.register(preds)

    def RemoveRegistration(self, resource):
        """
        Indicates that the caller is no longer interested in the
        particular resource.  All predicates relating to that
        resource are deleted.

        If the client has not previously registered an interest in
        the resource, the operation raises a ResourceNotRegistered
        exception.
        """
        if self.resources.has_key(resource):
            try:
                self.resources[resource].register([])
            except:
                raise Notification.RegisterIF.ResourceNotRegistered

class QoSMonitorInterface(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.master.title("QoS Monitor")
        self.master.iconname("Choose QoS Values")
        self.pack()

        # add menu bar
        menubar = Frame(self, relief=RAISED, bd=2)
        menubar.pack(side=TOP, fill=X)
        mbutton = Menubutton(menubar, text='File', underline=0)
        mbutton.pack(side=LEFT)
        menu = Menu(mbutton)
        menu.add_command(label='Quit', command=self.quit)
        mbutton['menu'] = menu

        # add labels
        self.labelframe = Frame(self)
        self.labelframe.pack(pady=40, padx=20)
        Label(self.labelframe, text="No resources currently").pack()
        Label(self.labelframe, text="being monitored").pack()

    def removeFrame(self):
        if self.labelframe is not None:
            self.labelframe.destroy()
```

```
            self.labelframe = None

class QoSMonitor:
    def __init__(self):
        self.masterWindow = QoSMonitorInterface()

    def main(self, argv):
        orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
        TkReactor.TkReactor_init()
        boa = BOA.BOA_init(argv, BOA.BOA_ID)
        obj = boa.create('QoSM', Register._FNORB_ID)
        impl = Register(self.masterWindow)
        boa.obj_is_ready(obj, impl)
        open('server.ref', 'w').write(orb.object_to_string(obj))
        boa._fnorb_mainloop()
        return 0

if __name__ == '__main__':
    monitor = QoSMonitor()
    sys.exit(monitor.main(sys.argv))
```

Resource.py

```
"""
Resource.py

This module implements a generic resource monitored by the QoS Monitor
simulator.

The Resource class implements a generic interface to a resource
handler.  All resource implementations should provide a subclass of
this class.

The ResourceInterface class implements a user interface component
displayed for the resource.  The create_widgets method should be
overriden with code that creates the appropriate widgets for obtaining
the current resource status from the user.  The update method should
also be overriden if the input is not via an entry box named "entry."
"""

from Tkinter import *
from Predicates import *

class Resource:
    def __init__(self):
        self.preds = Predicates(self.convertValue)

    def register(self, predicates):
        """
```

```
        Register a number of predicates that determine when the
        user should be notified of changes.

        Any previously registered predicates are overwritten.
        """
        self.preds.deleteAll()
        for pred in predicates:
            self.preds.add(pred)
        self.interface.update(None)

    def update(self, value):
        """
        The resource has changed - evaluate the predicates, and
        notify the user as required.
        """
        self.preds.notify(value)

    def convertValue(self, value):
        """
        Conversion function that maps from strings to the
        appropriate type for comparision of resource values (eg.
        numbers for bandwidth)

        The default does nothing.
        """
        return value

class ResourceInterface(Frame):
    def __init__(self, callback, parent):
        Frame.__init__(self, parent)
        parent.removeFrame()
        self.callback = callback
        self.pack(anchor='n', expand=YES, fill=X, padx=15, pady=15)
        self.createWidgets()

    def createWidgets(self):
        """
        Create the interface widgets for the resource - subclasses
        should override.
        """
        pass

    def update(self, event):
        """
        Callback for handling changes in the resource value.
        """
        input = self.entry.get()
        self.callback(input)
```

Predicates.py

```
"""
Predicates.py

This module implements mechanisms for storing and evaluating
predicates.

The Predicates class maintains lists of predicates.  New predicates are
added with add(), and a list of predicates is evaluated using notify().
The deleteAll() method removes all of the predicates in the list.

GeneralPredicate provides an abstraction of a predicate.    The
evaluate() operation determines whether a predicate holds using a
comparison function.

The remaining classes define specific types of predicate, and each has
its own version of the comparison function.
"""

import Notification

class Predicates:
   def __init__(self, conversionFunction):
      self.predicates = []
      self.conversionFunction = conversionFunction

   def add(self, predicate):
      """
      Add a new predicate.
      """
      value = self.conversionFunction(predicate.value)
      if predicate.op == Notification.RegisterIF.equal:
         pred = EqualToPredicate(value,
            predicate.notificationIF)
      elif predicate.op == Notification.RegisterIF.greater:
         pred = GreaterThanPredicate(value,
            predicate.notificationIF)
      elif predicate.op == Notification.RegisterIF.less:
         pred = LessThanPredicate(value,
            predicate.notificationIF)
      elif predicate.op == Notification.RegisterIF.less_or_equal:
         pred = LessThanOrEqualToPredicate(value,
            predicate.notificationIF)
      elif predicate.op == Notification.RegisterIF.greater_or_equal:
         pred = GreaterThanOrEqualToPredicate(value,
            predicate.notificationIF)
      self.predicates.append(pred)
```

```
    def deleteAll(self):
        """
        Delete all of the predicates.
        """
        self.predicates = []

    def notify(self, value):
        """
        Evaluate the predicates.  If a predicate evaluates to
        true, the client is notified using the callback interface.
        """
        for pred in self.predicates:
            if pred.evaluate(self.conversionFunction(value)):
                IF = pred.notificationIF
                IF.Notify(value)

class GeneralPredicate:
    def __init__(self, value, IF):
        self.value = value
        self.notificationIF = IF

    def evaluate(self, value):
        """
        Evaluate the predicate.
        """
        return self.compare(value, self.value)

    def compare(self, value1, value2):
        """
        Comparison function that determines whether or not the
        predicate should evaluate to true.  The second parameter
        should be the actual value of the resource, while the
        third paramter should be the value determined by the
        client.

        This function must be implemented by subclasses!
        """
        raise 'Subclass should implement'

class EqualToPredicate(GeneralPredicate):
    def compare(self, value1, value2):
        return value1 == value2

class GreaterThanPredicate(GeneralPredicate):
    def compare(self, value1, value2):
        return value1 > value2

class LessThanPredicate(GeneralPredicate):
    def compare(self, value1, value2):
        return value1 < value2
```

```
class LessThanOrEqualToPredicate(GeneralPredicate):
   def compare(self, value1, value2):
      return value1 <= value2

class GreaterThanOrEqualToPredicate(GeneralPredicate):
   def compare(self, value1, value2):
      return value1 >= value2
```

Bandwidth.py

```
"""
Bandwidth.py

This module implements a plug-in for the bandwidth resource.
"""

from Tkinter import *
from Resource import *
import string

class Bandwidth(Resource):
   def setMaster(self, master):
      self.interface = BandwidthInterface(self.update, master)

   def convertValue(self, value):
      return string.atoi(value)

class BandwidthInterface(ResourceInterface):
   def createWidgets(self):
      """
      Creates the interface widgets for obtaining the current
      (simulated) bandwidth value from the user.  The user
      enters the value through an entry box.  Pressing enter in
      the text box causes the value to be committed.
      """
      l = Label(self, text='Bandwidth:')
      l.pack(side=LEFT)
      self.variable = StringVar(self)
      self.variable.set("14400")
      self.entry = Entry(self, relief=SUNKEN, width=10,
         textvariable = self.variable)
      self.entry.pack(side=RIGHT, expand=NO)
      self.entry.bind("<Key-Return>", self.update)

   def update(self, event):
      self.callback(self.variable.get())
```

Display.py

```
"""
Display.py

This module implements a plug-in for the Display resource.
"""

from Tkinter import *
from Resource import *

class Display(Resource):
    def setMaster(self, master):
        self.interface = DisplayInterface(self.update, master)

class DisplayInterface(ResourceInterface):
    def createWidgets(self):
        """
        Creates the user interface widgets for obtaining
        information about the simulated display type from the
        user.  The user chooses the type using radiobuttons.
        """
        l = Label(self, text='Display:')
        l.pack(side=LEFT)
        ResourceInterface.createWidgets(self)
        frame = Frame(self)
        self.variable = StringVar(frame)
        self.variable.set("640x480")
        b0 = Radiobutton(frame, text="320 x 200",
            variable=self.variable, value = "320x200")
        b0.pack(side=TOP)
        b0.bind('<ButtonRelease>', self.update)
        b1 = Radiobutton(frame, text="640 x 480",
            variable=self.variable, value = "640x480")
        b1.pack(side=TOP)
        b1.bind('<ButtonRelease>', self.update)
        b2 = Radiobutton(frame, text="800 x 600",
            variable=self.variable, value = "800x600")
        b2.pack(side=TOP)
        b2.bind('<ButtonRelease>', self.update)
        frame.pack(side=RIGHT)

    def update(self, event):
        self.callback(self.variable.get())
```

Appendix C. New Grail modules

Adaptation.py

```
"""
Adaptation.py

Implements a callback manager, which is responsible for performing
initialisation required to use Fnorb, finding the QoSM using the
QOS_MONITOR environment variable, and creating the callback
implementations used by Grail.

Additionally, the get_boa operation returns a reference to the CORBA
BOA, and the stop operation terminates the callback handlers.
"""

import sys, os
from Fnorb.orb import BOA, CORBA, TkReactor
from notification import BandwidthHandler
from  notification import DisplayHandler

class AdaptationCallbacks:
    def __init__(self, app):
        """
        Start the callback handlers.
        """

        # Initialise the CORBA orb, TkReactor and boa
        orb = CORBA.ORB_init()
        TkReactor.TkReactor_init()
        self.boa = BOA.BOA_init([], BOA.BOA_ID)

        # Find the server
        if os.environ.has_key('QOS_MONITOR'):
            filename = os.environ['QOS_MONITOR']
        else:
            raise 'QoS Monitor not found'
        stringified_ior = open(filename).read()
        server = orb.string_to_object(stringified_ior)
        if server is None:
            raise 'Nil object reference'
        if not server._is_a('IDL:Notification/RegisterIF:1.0'):
            raise 'Incorrect server type'

        # Create the callback handlers
        self.bandwidthIF = BandwidthHandler.BandwidthCallbackHandler(
            self.boa, app, server)
        self.displayIF = DisplayHandler.DisplayCallbackHandler(
            self.boa, app, server)
```

```
    def get_boa(self):
        return self.boa

    def stop(self):
        """
        Terminate Grail callback handlers - called when the user
        exits Grail to ensure graceful termination.
        """
        self.bandwidthIF.stop()
        self.displayIF.stop()
```

CallbackHandler.py

```
"""
CallbackHandler.py

Implements a generic callback handler.   Subclasses must implement
Notify and Register operations.
"""

from Fnorb.orb import BOA
import Notification, Notification_skel

class CallbackHandler(Notification_skel.ChangeNotificationIF_skel):
    def __init__(self, name, boa, app, QoSM):
        self.app = app
        self.QoSM = QoSM
        # register with the boa
        obj = boa.create(name, CallbackHandler._FNORB_ID)
        boa.obj_is_ready(obj, self)
        self.Register()

    def Notify(self, value):
        """
        This method is called by the QoS monitor when QoS changes
        in a significant way.
        """
        raise 'Subclass should implement'

    def Register(self):
        """
        Register the resource predicates with the QoS Monitor.
        """
        raise 'Subclass should implement'

    def stop(self):
        pass
```

QualityOfService.py

```
"""
QualityOfService.py

Implements a class that manages Quality of Service values.  The class
is implemented as a simple wrapper for the python dictionary.
"""

class QualityOfService:
    """
    Manages Quality of Service information.  QoS attributes are
    retrieved with getQoS and set with setQoS
    """
    def __init__(self):
        self.QoS = {}

    def getQoS(self, attribute):
        if self.QoS.has_key(attribute):
            return self.QoS[attribute]
        else:
            return None

    def setQoS(self, attribute, value):
        self.QoS[attribute] = value
```

BandwidthHandler.py

```
"""
BandwidthHandler.py

Bandwidth callback handler for Grail.  The handler is notified of
changes in bandwidth availability via the Notify operation.  The
Register operation registers the predicates that determine when the
handler will be notified of bandwidth changes.  The predicates depend
on the bandwidth cutoff points in the user preferences.
"""

from Fnorb.orb import BOA
from Notification import *
from CallbackHandler import *
from grailbase import GrailPrefs
import string

class BandwidthCallbackHandler(CallbackHandler):
    def __init__(self, boa, app, QoSM):
```

```
    # when adaptation policy changes, re-register the
    # predicates with the QoS monitor in accordance with the
    # policy
    app.prefs.AddGroupCallback('adapt-policy', self.Register)

    # default bandwidth is high
    app.QoS.setQoS('bandwidth', 'high')
    CallbackHandler.__init__(self, 'Bandwidth Callback Handler',
       boa, app, QoSM)

def Notify(self, value):
    val = self.convertValue(value)

    # look up the current adaptation policy
    low = self.app.prefs.Get('adapt-policy', 'bandwidth-low')
    low = self.convertValue(low)
    medium = self.app.prefs.Get('adapt-policy', 'bandwidth-medium')
    medium = self.convertValue(medium)
    if val < low:
       self.app.QoS.setQoS('bandwidth', 'low')
    elif val < medium:
       self.app.QoS.setQoS('bandwidth', 'medium')
    else:
       self.app.QoS.setQoS('bandwidth', 'high')
    self.Register()

def Register(self):
    """
    Register the bandwidth predicates with the QoS Monitor.
    """
    currentQoS = self.app.QoS.getQoS('bandwidth')
    low = self.app.prefs.Get('adapt-policy', 'bandwidth-low')
    medium = self.app.prefs.Get('adapt-policy', 'bandwidth-medium')
    preds = []
    if currentQoS == 'low':
       preds = [RegisterIF.Pred(RegisterIF.greater_or_equal,
          low, self)]
    elif currentQoS == 'medium':
       pred1 = RegisterIF.Pred(RegisterIF.less, low, self)
       pred2 = RegisterIF.Pred(RegisterIF.greater_or_equal,
          medium, self)
       preds = [pred1, pred2]
    elif currentQoS == 'high':
       preds = [RegisterIF.Pred(RegisterIF.less, medium,
          self)]
    self.QoSM.Register('Bandwidth', preds)

def stop(self):
    """
    Terminate the callback handler - must be called when Grail
    exits to ensure graceful termination.
    """
```

```
        self.QoSM.RemoveRegistration('Bandwidth')
        CallbackHandler.stop(self)

    def convertValue(self, value):
        """
        Conversion function from strings to bandwidth values.
        """
        return string.atoi(value)
```

DisplayHandler.py

```
"""
DisplayHandler.py

Display callback handler for Grail.  The handler is notified of changes
in display type via the Notify operation.   The Register operation
registers the predicates with the QoSM that determine when the handler
will be notified of display changes.
"""

from Fnorb.orb import BOA
from Notification import *
from CallbackHandler import *
from grailbase import GrailPrefs

class DisplayCallbackHandler(CallbackHandler):
    def __init__(self, boa, app, QoSM):
        app.QoS.setQoS('display', '640x480')
        size = app.prefs.Get('browser', 'default-font-size')
        app.QoS.setQoS('font-size', size)
        CallbackHandler.__init__(self, 'Display Callback Handler',
            boa, app, QoSM)

    def Notify(self, value):
        """
        Bring about adaptation of the browser in response to the
        change in display type.

        Adaptation mechanisms are triggered by the changes to the
        Grail preferences, ie. callbacks defined in various Grail
        modules are automatically invoked in response to the
        changes to the browser preference group.
        """
        if value == '320x200':
            self.app.QoS.setQoS('display', 'small')
            if self.app.prefs.Get('adapt', 'display') != 'None':
                self.app.prefs.Set('styles', 'size', 'tiny')
                size = self.app.prefs.Get('browser',
```

```
                'default-320x200-size')
            self.app.QoS.setQoS('font-size', size)
        if self.app.prefs.Get('adapt', 'display') == \
                'Window and Pages':
            self.app.prefs.Set('browser', 'load-images', '0')
    elif value == '640x480':
        self.app.QoS.setQoS('display', 'small')
        if self.app.prefs.Get('adapt', 'display') != 'None':
            self.app.prefs.Set('styles', 'size', 'small')
            size = self.app.prefs.Get('browser',
                'default-640x480-size')
            self.app.QoS.setQoS('font-size', size)
        if self.app.prefs.Get('adapt', 'display') == \
                'Window and Pages':
            self.app.prefs.Set('browser', 'load-images', '1')
    elif value == '800x600':
        self.app.QoS.setQoS('display', 'small')
        if self.app.prefs.Get('adapt', 'display') != 'None':
            self.app.prefs.Set('styles', 'size', 'medium')
            size = self.app.prefs.Get('browser',
                'default-800x600-size')
            self.app.QoS.setQoS('font-size', size)
        if self.app.prefs.Get('adapt', 'display') == \
                'Window and Pages':
            self.app.prefs.Set('browser', 'load-images', '1')
    self.app.prefs.Save()

def Register(self):
    """
    Register the display type predicates with the QoS Monitor.

    Three types of display are supported.
    """
    pred1 = RegisterIF.Pred(RegisterIF.equal, '320x200', self)
    pred2 = RegisterIF.Pred(RegisterIF.equal, '640x480', self)
    pred3 = RegisterIF.Pred(RegisterIF.equal, '800x600', self)
    preds = [pred1, pred2, pred3]
    self.QoSM.Register('Display', preds)

def stop(self):
    """
    Terminate the callback handler - must be called when Grail
    exits to ensure graceful termination.
    """
    self.QoSM.RemoveRegistration('Display')
    CallbackHandler.stop(self)
```

Appendix D. Modifications to existing Grail modules.

This appendix briefly describes modifications that have been made to the original Grail modules. Due to the length and number of these modules, it is infeasible to include full listings.

The changes are divided into modifications to core modules, user interface modules, and protocol modules.

D.1 Modifications to core Grail modules

Module	Class	Method	Modification
grail	Application	__init__	Added the creation of an AdaptationCallbacks instance, which is responsible for creating the callback handlers that receive notifications from the QoS Monitor.
grail	Application	quit	Modified the method so that callbacks from the QoS Monitor and the Fnorb event loop are stopped before Grail terminates. This prevents Fnorb errors.
grail	Application	go	Replaced the original tk mainloop that processed tk events with a Fnorb event loop that processes both tk and Fnorb events.
app	Application	__init__	Added code to create the QoS repository.

D.2 Modifications to user interface modules

The modifications to the user interface modules add support for adaptation of user interface fonts to the display type.

Module	Class	Method	Modification
IOStatusPanel	IOStatusPanel	__init__	Registered a callback handler, update_styles, to be invoked when any of the style settings changes.
PrefsPanels	Framework	__init__	
PrintDialog	RealPrintDialog	__init__	
Browser	Browser	__init__	
IOStatusPanel	IOStatusPanel	create_widgets	Added code to determine the font style that should be used for the interface, using the base font defined by the Grail preference file, and the font size dictated by the current QoS. Modified the creation of widgets to use this font when possible (that is, when the widget creation method supports the specification of a font type).
PrefsPanels	Framework	PrefsWidgetLabel, PrefsEntry, PrefsRadioButtons, PrefsCheckButton, PrefsOptionMenu, create_disposition_bar	
PrintDialog	RealPrintDialog	__init__	
Browser	Browser	create_menubar, create_urlbar, create_statusbar	

Module	Class	Method	Modification
AppletsPanel	AppletsPanel	CreateLayout	Added code to determine the font style that should be used for the interface, using the base font defined by the Grail preference file, and the font size dictated by the current QoS. Modified the creation of widgets to use this font when possible (that is, when the widget creation method supports the specification of a font). Created a list of interface widgets, in order to allow widget fonts to be updated easily by the update_styles method.
BookmarksPanel	BookmarksPanel	CreateLayout	
CachePanel	CachePanel	CreateRadioButtons, CreateLayout	
GeneralPanel	GeneralPanel	CreateLayout	
ProxiesPanel	ProxiesPanel	CreateLayout	
StylePanel	StylePanel	CreateLayout, __add_color	
IOStatusPanel	IOStatusPanel	update_styles	Added a new method that updates the font styles used by interface widgets when the style settings change.
PrefsPanels	Framework	update_styles	
PrintDialog	PrintDialog	update_styles	
OpenURIDialog	OpenURIDialog	__init__	Added an optional font parameter, allowing a font to be specified when the dialog is created. In addition, created code to set the fonts used by interface widgets when a font parameter is supplied.
PrintingPanel	PrintingPanel	CreateLayout	Increased the widths of “vertical separation” and “indentation” entry labels to accommodate larger font sizes.
Browser	Browser	__init__	Initialised the list of menus to the empty list. The menu list stores references to all of the menus used by the browser, and is used by update_styles to update the menus.
Browser	Browser	__init__	Added the url instance variable to track the last page loaded. It is initialised to the page initially loaded by the browser, if such a page exists.

Module	Class	Method	Modification
Browser	Browser	load_from_entry	Changed the local url variable to an instance variable that can be used in a wider scope.
Browser	Browser	update_styles	Added a new method to update the font styles used by interface widgets when the style settings change. In addition, the method reloads the last page loaded in the browser (from the cache where possible), so that the style changes are shown immediately.
Browser	Browser	create_menu	Added code to determine the font style that should be used for the interface, using the base font defined by the Grail preference file, and the font size dictated by the current QoS. Modified the creation of menus so that menu items are displayed using the font. Added the new menu to the menu list.
Browser	Browser	open_uri_command	Added code to determine the currently used font style. This style is passed to OpenURIDialog so that the font can be used in the dialog box.

D.3 Modifications to protocol modules

Module	Class	Method	Modification
httpAPI	MyHTTP	open	Added support for the QoS header. The header carries QoS information that can be used by the HTTP server in the selection of a variant. The QoS information is only included in the header if adaptation of downloads to QoS is enabled. Currently, only QoS relating to bandwidth and display type are supported.