Supporting Online Services in Environments Constrained by Communication

Arup Mukherjee

November 4, 1998

CMU-CS-98-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Daniel P. Siewiorek, Chair
Adam Beguelin
James H. Morris
Asim Smailagic, Institute for Complex Engineered Systems

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement numbers DABT 63-95-C-0026 and DASG 60-96-C-0068, and by AT&T.
The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or AT&T.

Keywords: Mobile computing, Control-oriented programming, World Wide Web, proxies, constrained communication

ABSTRACT

With the recent rapid growth of the World Wide Web, the advent of commodity internet access via modems, and the slower but steady proliferation of mobile computing devices, more and more users are accessing internet services from computationally capable machines connected via low-speed communication links. While many such services only offer access to static, unchanging documents, the number of "online" services offering access to frequently updated information is growing rapidly. While systems have been developed to optimize document access in the presence of constrained communication, their approaches do not handle access to online services well.

This thesis examines the problem of designing online services to be accessed over constrained communication links by computationally capable clients. A taxonomy of application classes has been developed to define the distinguishing characteristics of online services, and recognizes that clients and applications often interact not just to exchange data, but also to control each other's actions or resources. This recognition is the basis for two new models of application structure. The two models are proposed as alternatives to established models and form the basis for the application structuring techniques developed and evaluated.

Support for these application structures has been implemented by the Oasis system, which has been designed to allow the easy deployment of online services as applications within the existing framework of the World Wide Web. An examination of some Oasis applications has demonstrated the qualitative and quantitative advantages of control-oriented design as a flexible approach to reducing the communication requirements of online services. For example, a weather data browser designed explicitly to take advantage of Oasis, when partitioned in a control-oriented manner, was able to eliminate 95% of the steady-state communication usage of alternative implementations. Opportunities for smaller reductions in communication usage have also been observed by proxying communications and automatically leveraging the Oasis infrastructure without explicit application support.

Acknowledgements

This thesis represents an achievement made possible thanks to the help, support, advice and input of many people over the last several years. First and foremost, I thank my advisor, Dan Siewiorek, for his guidance, support, and professionalism throughout my graduate school experience. I have learned much from him through our joint exploration of new technical frontiers, and through my watching him competently, patiently, and effectively guide several research efforts through good times and bad.

I am also grateful to the members of my thesis committee: Adam Beguelin, Jim Morris, and Asim Smailagic. My discussions with Adam over the years produced many insights that helped shape my research direction. In all of my interactions with Jim, he has never failed to ask a thought-provoking and original question relevant to the situation at hand. Time and time again, Asim has proven himself to be one of the most helpful and energetic people I have met. I really appreciate all of the occasions on which he has come through for me on short notice.

While faculty featured prominently in my graduate school experience, I have learned and benefitted just as much from my interactions with several fellow graduate students. Claudson Bornstein, who for many years was my officemate, best friend, and confidant is at the top of that list of friends. I also owe much to other close friends I met at CMU (Rujith DeSilva, Darrell Kindred, and Po-Jen Yang) and to my former officemate Sergio Campos. I have shared many educational and extracurricular experiences with all of them. It would be very difficult to list all of my friends that added to the fun and richness of my experiences at CMU. From amongst them, however, I am particularly glad to have met Somesh Jha, Manish Pandey, Girija Narlikar, Rich McDaniel, Eka Ginting, Erik Seligman, Henry Rowley, Karen Haigh, Rob Driskill, and Hao-Chi Wong.

From day to day, the computer science department at CMU has often seemed to run by magic. This means special thanks for making the nontechnical aspects of my life as trouble-free as possible are due to Sharon Burks, Laura Forsyth, Catherine Copetas, Karen Olack, and all of the amazingly accomodating CS computing facilities staff that I have had occasion to deal with.

I might not have come to CMU if not for the encouragement of my friend and ex-boss, Murthy Devarakonda. I am fortunate to have worked with someone as committed as he was to helping me achieve my own goals in life.

Finally, much credit is due to my family. My parents inspired me to initiate my quest for a thesis, and supported and encouraged me in all of my endeavors. More recently, my in-laws have added their encouragement to that of my parents. Most important of all, the love and support of my wife, Nita, have never faltered in the face of somewhat more intimate exposure to the realities of finishing one's doctoral research. The creation of this thesis, like all efforts we will share in life, is as much her accomplishment as it is mine.

Contents

Chapter 1

Introduction

Over the last few years, the explosive growth of the internet and World Wide Web have significantly increased the accessibility of networking to technologically unsophisticated users. Many of these newcomers to networking do not use their computers for data creation. Browsing (61%), information gathering (81%), education (52 %), and entertainment (46%) were among the most frequently mentioned Web activities in a recent survey [24] of Web users — and these statistics may be underestimates, given the self-selecting nature of the survey. Almost 70% of the participants in that survey were connected to the internet by relatively slow (33.6 Kbits/sec or slower) links, running PPP [46] or SLIP [40] over dialup phone lines. As a greater segment of the general population is brought online, both the proportion of users that want to access (but not to modify) online data and services, and the proportion of users that are connected by relatively slow links, continues to increase.

A second distinct community reflects a similar pattern. Users of mobile computers typically cannot use their machines for data creation due to the limited input capabilities of most truly mobile hardware. However, mobile computing devices exhibit much promise as communication devices, and as interfaces to services and data provided by a surrounding network of static servers [51, 56, 44]. However, the wireless networking technology available to mobile computers today is also relatively slow compared to that of wired networks. Users of mobile computing thus also contribute to the growing set of network users, connected via relatively slow links, who do not primarily perform data creation tasks.

Traditionally, applications built for bandwidth-constrained environments usually reflect either the *terminal* or the *workstation* models [13]. Systems such as the Berkeley Infopad [51] and Xerox

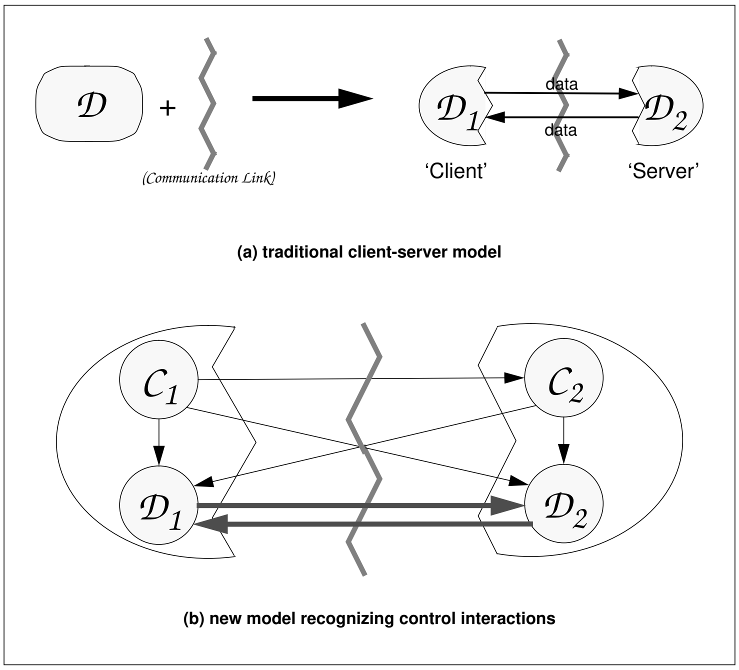


Figure 1.1: The introduction of a communication link partitions an application. Earlier models have treated the communication across this link as homogeneous movement of "data", as in (a). The work described herein treats the partitioned application as consisting of Data Manipulation (D) and Control Processing (C) components. Communication between data manipulation components (i.e. the movement of pure application data), represented by the thick arrows, is distinguished from interactions between control processing components (i.e. "movement of control"), represented by the thin arrows.

PARCtabs [44] embody the terminal model, which suggests that a mobile machine should be regarded solely as an output device, that all manipulation of data should be performed on the server network, and that the communications link be used to return the resultant output. This model is also typical of most online services currently deployed on the World Wide Web. On the other hand, the workstation model, exemplified by systems such as Coda [43], suggests that a mobile machine be the site of all data manipulation, and that its communications link be used solely to access remote data. Web browsers originally did not allow a remote service to take advantage of the computational capabilities of the machine the browser was running on; today, major browsers have this ability in a limited form although most online services fail to leverage it extensively. The ability to partition applications across a communications link warrants the study of new application models.

This research differs from previous work in recognising that not only does communication *partition* the *data manipulation* (or "computation") of an application, but that it also partitions both internal and external *control* interactions of that application. Control interactions are those interactions between a controller entity and its target entity to produce an effect in the environment of the target entity. This distinction from prior work is depicted graphically in Figure 1.1. For example, a weather database service might control the operation of a map browser client running remotely, instructing the browser to display a specific map before instructing it how to augment that map with current meteorological data. An examination of the partitioning of control interactions by communication produces two new models of applications and services deployed in a communication-constrained environment. Accordingly, two new application models have been defined, supported in a Web-based application framework called Oasis, and operationally evaluated through the measurement of sample applications.

A summary of related work is presented in Section 1.1, followed by an outline of the thesis in Section 1.2, and a summary of research contributions in Section 1.3.

1.1 Summary of Related Work

Current and past research efforts of relevance to the design and deployment of online services in communication-constrained environments can be classified into three categories:

1. Efforts to support the downloading and execution of untrusted mobile code, e.g. [49, 50, 54, 16, 57].
2. Efforts to partition applications based on changing the data communicated, e.g. [56, 15, 9, 19, 60].
3. Ad-hoc manual application-specific optimization of code layout, e.g. [7].

The research presented here encompasses and extends the goals of the first and third items, and is complementary to the efforts of the second. Important differences between this work and previous efforts are:

1. Earlier projects to support the execution of untrusted downloaded code usually have considered their efforts neither a mechanism for the partitioning of an application nor a means to manage the use of shared resources on the service–client path.

2. No prior effort has focussed on the use of control interactions as the hints to be used in deciding on application partitions.

3. Prior efforts at application-specific optimization have occasionally yielded control-oriented partitions. However, this result has been achieved through ad-hoc design rather than through the use of general building blocks.

The purpose and functionality of the Oasis system differs from all other systems that have attempted to support downloaded code, or to provide proxy-based or network-based computation. The structure of Oasis reflects these differences. Efforts that have considered similar application scenarios have not investigated these applications in the context of a proxy-based network of computationally-capable resources. As a result, these efforts focussed only on the data transfer aspects of the applications studied, and have typically produced results that are subsumed by, or could complement, the work presented.

These differences are documented in greater detail in Section 2.3.

1.2 Thesis Outline

This thesis examines current approaches to implementing applications and services in communication-constrained environments, presents two new models of application structure, and documents the implementation and evaluation of an infrastructure to support the two models in the context of the World Wide Web.

Chapter 2 presents the motivation for, and the models underlying, the thesis research. The target computing environment is defined, and existing approaches to application deployment are examined. Two novel models of application structure are then described and analyzed, producing a list of benefits that may be realizable by applying them. A set of three important building blocks (augmented by two lesser ones, and three that were not within the scope of this study) is presented as a basis from which application structures can be constructed. Finally, the chapter presents a discussion of work related to this research, i.e. work that has produced artifacts that are similar to Oasis, work related to application partitioning and work addressing models of application structure.

Chapter 3 describes the architecture of the Oasis system, and the facilities that Oasis provides for the construction of agent-based applications for access of WWW services over a communication-constrained link.

Chapter 4 documents the sample applications that were chosen for the evaluation of the Oasis architecture, and their use of the Oasis building blocks.

Chapter 5 analyzes the performance of the Oasis infrastructure in the context of the demonstration applications.

Chapter 6 summarizes the results of the research, and suggests areas for further exploration. Although the Oasis system provides a flexible platform upon which Web applications can easily be deployed in a control-oriented manner, there is potential for further ease of use by developing higher level components based on the Oasis foundation.

1.3 Summary of Research Contributions

The research described in this thesis makes conceptual contributions to the existing body of knowledge as well as practical contributions, arising from the implementation of the prototype system, to the body of tools for experimenting with online services in the context of the World Wide Web.

Conceptual Contributions

The foremost conceptual contribution of this work is an understanding of the importance of control interactions when partitioning an application across multiple computationally capable entities separated by constrained communication channels. This understanding is embodied in design principles and in a proposed minimal set of simple building blocks from which services can be constructed in a control-oriented manner. The use of these building blocks has been demonstrated through the construction of a number of example services.

In addition, this work provides insight into how to harness client-side and intermediary computation capabilities in any distributed system, particularly in one that is communication-constrained at some points. The research approach taken demonstrates that proxy-based interaction is a viable and flexible mechanism for the compatible addition of control-oriented partitioning to an existing distributed system. The work also exemplifies one possible set of core proxy services that form an extensible framework of sufficient flexibility to support the control interactions of arbitrary client applications.

Practical Contributions

The main practical contribution of this research is the implemented artifact itself: the Oasis proxy server which is capable of hosting, in an isolated environment, specialized agent code supplied by service providers, end-users, or other agent code. The proxy is capable of recognising other similar proxies. As such, the proxy forms the basis for a network of computationally capable entities inserted between users and service providers. This infrastructure is suitable for the implementation of any web-based system that needs to strategically place "transducers" into the network. In the case of the research described, the transducers are mainly agents designed to localize adaptations needed to accomodate a slow network link, and to minimize the effects thereof. The Oasis proxy also provides services necessary to support inter-agent interaction, agent interaction with resources available in proxy nodes, and some agent interaction with the processing of the HTTP protocol itself.

In addition to the Oasis proxy server, Oasis includes a small library of utility functions and interfaces useful for the construction of agents. This library is external to the Oasis proxy itself, but provides functionality that would otherwise need to be reimplemented in most nontrivial agent designs.

Finally, a number of the implemented examples are useful even to those with no research interest in Oasis. For example, the Oasis proxy, initialized appropriately, can be configured to intercept and redirect FTP requests, filter out unwanted graphics (particularly commercial messages), provide client-side caching with difference-based updates, and protect the end-user's identity. All of these capabilities are of interest to WWW users online via modem connections; they may save several seconds of access latency for every graphical advertisement eliminated and for every page that can be updated by diffs rather than completely refetched.

Chapter 2

Background

This chapter presents the motivation, reasoning, and prior work relevant to the research in this thesis. Section 2.1 opens by examining and classifying applications in some communication-constrained environments, using the resultant taxonomy to generate four models of application system structure. It concludes that two of the models have been widely employed in past work, whereas two others, arising from a consideration of the *control* interactions between application components, have been overlooked or hitherto unidentified. The section ends by arguing that, in a communication-constrained environment, application design using the two new models may have many benefits. Section 2.2 then presents a set of building blocks useful for constructing applications having the structure specified by the new models. These building blocks are the ones that drove the design of the Oasis infrastructure. Finally, Section 2.3 presents related prior and contemporary work relevant to this research.

2.1 Models and Motivation

This section presents definitions and models underlying the research approach. The defining characteristics of the environment studied are first listed. It is then argued that applications in the target environment span one or more of three types of functionality: *data manipulation*, *communication* and *control*. These three types of functionality are defined, and used to generate a taxonomy of application classes which in turn generates four system models. Two of these models are the traditional *terminal* and *workstation* models which are briefly presented. Two new models, the *user agent* and *server agent* models, are then described. Support for these two models defines the the building blocks for agent-based applications enabled by the Oasis framework. This section concludes by examining some of the benefits that control-oriented applications are able to realize in the target environment.

2.1.1 The Target Environment

The target environment is a distributed information system composed of service providers and clients. The service providers run servers interconnected by a (relatively) high speed network, and provide services to the pool of users. Users connect to the provider network over links that are slower than those of the backbone network, usually significantly so. Each user connects to the network using a *computationally capable* unit – a workstation or mobile machine with processing ability comparable to that of a small workstation. Each user's unit has some storage space (cache) and may be connected to other resources (such as a CD-ROM drive). Machines are typically not shared between users, and each user owns the resources associated with his unit(s). Users are not primarily concerned with creating data or services, but rather with accessing information and services available to them.

Several examples of such environments currently exist. The World Wide Web exhibits this architecture, as do many experimental mobile computing environments. Most commercial online services in existence today (e.g. America Online) also fall into the same category. All of these examples have experienced phenomenal growth in the past few years. Networking has become part of mainstream personal computing, and this scenario has been brought to prevalence with it.

Oasis has been constructed in the context of the World Wide Web. The Web consists of a multitude of independent information servers, accessed by users who run a Web browser to contact servers and retrieve information. A browser usually presents a server with a request for information, specified via a Uniform Resource Locator, or URL [29], and processes the returned information appropriately. Depending on the type of information returned, the browser may format the data and display it to the user immediately (as is usually the case for an HTML [30] document), or it may hand the data to an appropriate helper application (e.g., a postcript document would usually be given to a postscript previewer for display). A Web browser serves primarily as a mechanism for the the retrieval and presentation of data, which is able to recognize and follow hypertext links in that data. Major graphical Web browsers (i.e. Netscape [32] and Microsoft's Internet Explorer [31]) also offer a limited environment for the execution of safe "applet" code downloaded from Web servers.

The remainder of Section 2.1 presents four application models. In presenting these models, mobile computing is often used as a source of example applications and scenarios. This is because the issues of application structure have been considered to a greater extent in the mobile computing literature. The Web was originally conceived as a mechanism for the dissemination of data. Many interactive service applications (e.g. cataloging services) have nevertheless appeared on the Web, and are encouraging the development of an infrastructure to support them. The development of forms support for Web browsers was the first such step, and the appearance of mechanisms to support interactive content was the second. Although many examples are drawn from mobile computing, the models presented also apply to the World Wide Web and to all of the other aforementioned environments.

2.1.2 Application Functionalities

Mobile computing technology is often (e.g., in [26]) regarded as the convergence of tools from two distinct walks of life, namely *communications* (e.g., telephones and pagers) and *data manipulation* or *computation*[1] (e.g., laptop computers). Many mobile applications built to date reflect this heritage, and much effort has been devoted to enhancing traditional "desktop" applications to reap the benefits of mobility (e.g. via the Coda filesystem [43]), and to extending pagers and mobile telephones to exploit additional computing power (e.g., PARCtabs used to read e-mail [44]). Even the computational models underlying these applications usually focus on how exactly the data manipulation performed by an application is partitioned by the communication it performs. However, mobile computation can also be viewed as the convergence of remote controls and laptop computers. This view leads to a richer set of computational models and applications.

> *'Control' is the human or automated guidance of a process that affects its environment.*

Control is an aspect of everyday life that is just as pervasive as communication or computation. Mechanical control interfaces are found on most nontrivial human tools, in the form of buttons, dials, sliders, LCDs, etc. Some control interfaces have been integrated with communications technology, leading to devices such as remote controllers for televisions. When computational capability is associated with an end-user of a tool, or with the tool itself, the possibility of integrating computation and control arises.

[1] The term "data manipulation" is often used in preference to "computation" to avoid confusion with the multitude of definitions that have been applied to the latter term in other contexts.

Mobile computation offers the possibility of such integration, leading to a new realm of applications. These new applications are not restricted to mechanical objects. Rather, control-oriented technology offers an alternative paradigm for accessing any object capable of digital communication, whether it is an intelligent toaster, or a network-accessible vehicle guidance system implemented entirely as a software database application.

Applications can be grouped into a taxonomy based on the three converging technologies.

2.1.3 A Taxonomy of Applications

Communication, data manipulation, and control are three application domains for which many mechanical and electronic tools exist, including most computing applications constructed to date. The three domains are not mutually exclusive, and some applications serve purposes arising from the intersection of two or of all three. The relationship between the domains, as depicted in Figure 2.1, generates seven application classes:

- I **Pure Data Manipulation**: e.g., a sorting program.
- II **Pure Communication**: e.g., a pager, or mobile telephone. Simple fetches of Web pages are also pure communication.
- III **Pure Control**: e.g. operating an oven, driving a car.
- IV **Document Passing**: Two data-manipulating entities exchanging data, e.g. a system running Coda [43]. Most server-client scenarios fall into this category.
- V **Remote Control**: Simple control occurring over a communications link, e.g., a television's remote controller, which just encodes keypresses onto its infrared output.
- VI **Guided Data Manipulation**: A data manipulation that responds to input instructions, e.g., most interactive programs, such as an interactive database browser.
- VII **Collaborating Services:** Applications consisting of at least two communicating entities, wherein at least one entity is controlling another. Control commands and/or data may be flowing across the communications link, e.g., a postscript program downloaded to a postscript printer, which executes the program and prints the output. In this application, the control is unidirectional only, and flows from a workstation to the printer. Bidirectional control is also possible. Currently, this class of application is not very common in the

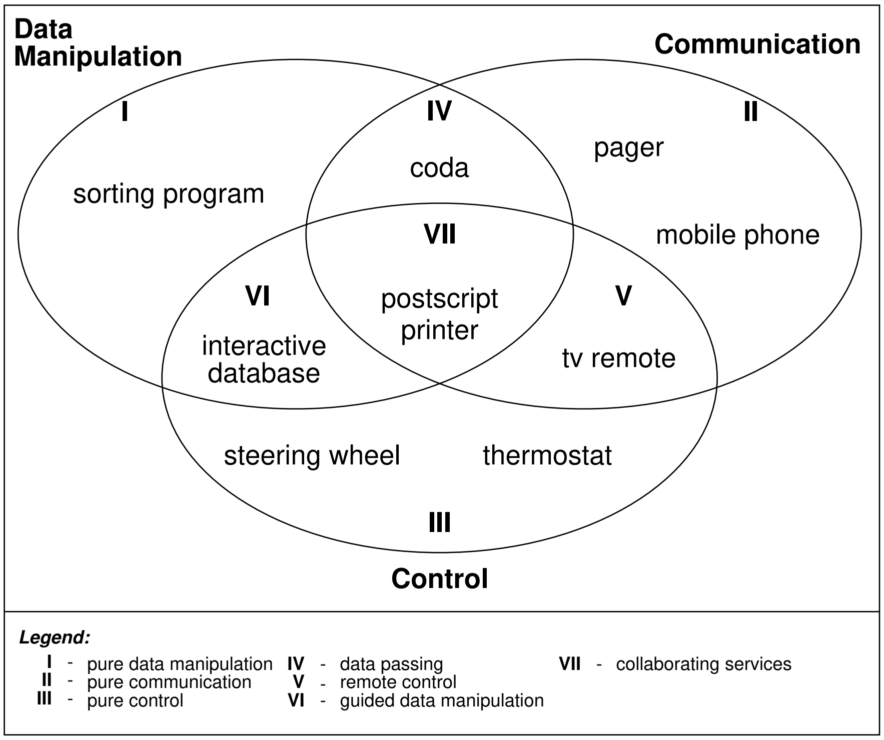


Figure 2.1: Three intersecting application arenas, and the resultant application classes.

world of mobile computing. In the context of the World Wide Web, the advent of browser support for Java applets has resulted in the appearance of a number of limited examples of this application class. The Oasis system described in this thesis is an enabling technology for more complex collaborating services applications in the context of the Web.

Applications are not constrained to being of a single class, and may be composed of multiple phases of distinct classes. In fact, those that initially appear to be a system of collaborating services (class VII) often fall into this category. For example, the use of Netscape to query a database on the World Wide Web often consists of a guided data manipulation (class VI) phase (a search that takes some parameters) followed by a pure communication (class II) phase (transfer of the result to the user).

The class of an application may also vary with the granularity at which it is examined. Complex systems can usually be broken down into several independent subsystems, each of which is an "application" in its own right. Each subsystem thus falls into one of the seven classes, which is not necessarily that of the composite application. In decomposing a system this way, care must be taken to select a level of granularity useful for characterizing behavior in terms of the system model being employed. At the finest granularity, every application is composed of pure communication, computation, and data manipulation phases. At the coarsest granularity, every application has only a single class. For simple applications, the coarsest level may be appropriate. Usually, however, neither of the extremes conveys any useful information, and a point in between is appropriate. For the purposes of applications developed for Oasis, the application developer can choose to draw subsystem boundaries along those between classes, or at a coarser level, along boundaries delineated by interfaces.

The evaluation of the Oasis system, presented in Chapter 4, examines Oasis in the context of pure communication and collaborating services applications. The evaluation first examines the overhead incurred by unmodified pure communication Web accesses and document passing Web accesses when using the Oasis proxy. It subsequently presents the benefits obtainable by automatically converting pure communication and document passing Web applications to collaborating services applications through the use of controllable caching, i.e. through the introduction of an intermediate control and data manipulation step. Lastly, Oasis is evaluated in terms of its ability to support Web applications that are explicitly designed to be collaborating services applications, and in terms of the benefits those applications realize relative to alternative implementations.

2.1.4 Models of System Structure

A "system model," or "model of computation" arises when attempting to realize an abstract application class in the presence of the constraints of the physical world. For example, many existing models of mobile computation arose from attempts to realize document passing (class IV) applications on current mobile computing hardware.

An application that falls into the communications domain as well as another domain is immediately faced with the issue of how the communication partitions the other components of the application. For example, a document passing (class IV) application's designer must decide exactly how much manipulation of data is done at each end of the communication channel. In an environment with infinite bandwidth, zero-latency communication, this issue would not have been very important. However, most clients of the World Wide Web, and most current mobile computing devices, have limited communications capability, and the resolution of this issue is the characteristic that differentiates models.

> *In a communication-constrained environment, the communication* **partitions** *the rest of the application.*

Two commonly cited models, the "terminal" and "workstation" models, as described by [13], represent the two extremes of partitioning. The terminal model of mobile computing views mobile machines as purely input and/or output devices, which relay all data to a remote server (i.e. on the other end of the communication channel), and display any results returned. No data manipulation is done on the mobile unit itself. The workstation model, on the other hand, describes an autonomous mobile unit that communicates with remote servers only for the purposes of obtaining data. No manipulation of data is performed on the servers themselves. Other models lying in between the two extremes, such as "terminals with the possibility of downloading some code," [17] have been considered, but have not become very popular. The workstation model has also spurred the development of submodels converting pure data manipulation (class I) desktop applications to document passing (class IV) mobile applications, with varying degrees of application awareness (e.g., low awareness in the case of Coda [43], and high awareness in [17]).

2.1.5 Control-oriented Models

Control-oriented applications, like communications-oriented applications, must, by definition, span at least two entities, although the two entities need not be physically separated. However, the distinction between controller and controllee ensures that there is a well-defined interface, or control protocol, between the two. The distinction also allows the two entities to be separated by a communications channel, producing a partitioning of control processing in much the same way that the introduction of a communications channel can create a partitioning of data manipulation. In applications that perform both data manipulation and control processing, the introduction of finite-capacity communication channels produces a two-dimensional range of computing models. The extreme points of that range, as applied to an application running in a mobile environment, are delineated in Table 2.1.

The "terminal" and "workstation" models shown in the table are functionally equivalent to the more commonly referenced models of the same name (see Section 2.1.4), although the original models were derived with no consideration given to control processing. The two new models are characterized in terms of *agent*s.

> *An agent is a mechanism to allow tranfer of control across a communication link.*

Site of control processing	Site of data manipulation	
	Mobile Machine	Static Server
Mobile Machine	`Workstation`	`User agent`
Static Server	`Server agent`	`Terminal`

Table 2.1: The models at the four extremes in the partitioning of control processing and data manipulation.

Basically, an agent allows execution of a sequence of actions on one side of a communications channel under the control of the entity at the other end of the channel. For example, the "user agent" model is a scenario in which a mobile machine (the "user") acts only to guide an application whose data manipulation occurs entirely on the server system (e.g., the mobile machine might instruct the server - a microwave oven - to determine via database lookup how long it takes to cook Swedish meatballs, and to then run the oven for that length of time.)

The "user agent" and "server agent" models represent diametrically opposite extremes in the division of control processing among two application components, i.e. unidirectional control. Many of the example control applications given are also unidirectional, and indeed most control applications developed to date are unidirectional. However, this is probably due to the lack of computational power in current controllers – a situation which is changing as mobile computers become more prevalent. Control-oriented applications exploiting bidirectional control are well suited to current mobile computing hardware, as described in the next section.

Bidirectional control is the relationship between two entities that both control, and are controlled by, each other. The relationship is best visualized as a partitioning of the control processing in an application, once again arising from the introduction of communication channels, as depicted in Figure 2.2. In practice, bidirectional control can be treated adequately by modelling it as the concurrent existence of user and server agents in an application.

Lastly, it should be noted that although control-oriented applications are characterized by the existence of a controller/controllee interface, the nature of this interface is unspecified. Specific implementations further distinguish models of control-oriented computation. For example, the controller might employ a "single action at a time" (RPC-like) interface to the controllee, or it might employ a "remote-execution" model, wherein arbitrary amounts of control code are shipped to the controllee to regulate its actions. The Oasis architecture enables the separation of controller and controllee, leaving the control mechanism unspecified. Accordingly, either type of control relationship may be found between Oasis applications and agents.

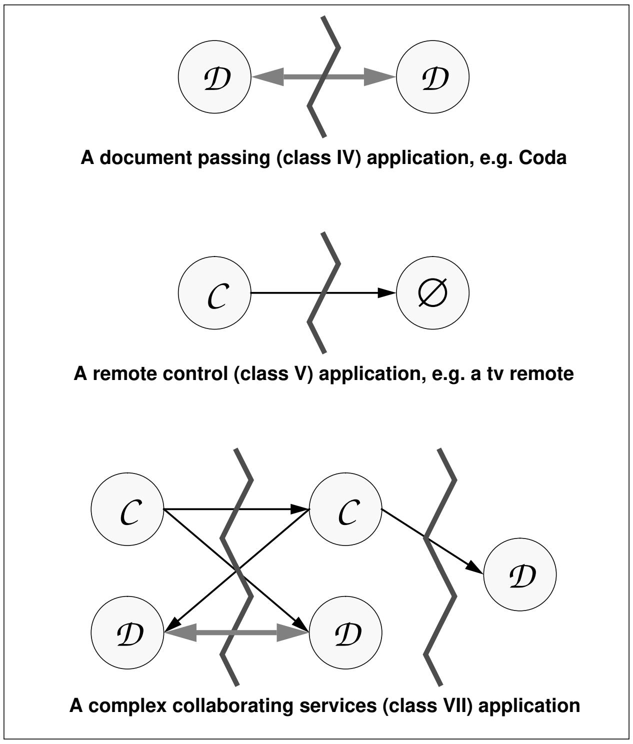


Figure 2.2: A more general model of an application: control(C) and data(D) processing can be regarded as the nodes of a graph, with an arbitrary number of partitions induced by communication. Communication-induced partitions are indicated by jagged grey lines. Nodes marked $\emptyset$ are nodes at which nothing other than I/O operations occur, i.e., no control processing or data manipulation occurs, but interactions may be forwarded to another node, or, as in the tv remote exemplified above, forwarded to the external environment. The edges of the graph represent interactions between components, either local (e.g. procedure call), or across a communication boundary (e.g. remote procedure call, message passing). Control interactions (emanating from a control node) are represented as thin black edges, whereas data interactions (between data manipulation nodes) are represented by thicker grey edges.

2.1.6 A Case for Control-oriented Applications

Design of applications in a control-oriented manner differs from traditional, data-directed design mainly in that the application designer is encouraged to exploit the computational abilities of both the server and the client hardware (and potentially even the resources provided by the network in between them). True control-oriented designs are possible when the application can be split into two or more appropriately interacting pieces. The target environments differ from local area networks in that communication is often slow, has a high latency, and may be unreliable – so much so that appropriate application partitioning can significantly affect the usability of the result. An application partitioned based on control-oriented considerations may realize the following benefits:

1. Reduced bandwidth requirements. The controller/controllee separation should reflect the spatial locality of data accesses, allowing movement of instructions to be traded against the movement of larger amounts of data. In addition, some types of data are algorithmically generatable from more compact representations; a control-oriented design can take advantage of such a characteristic, employing the resources of the data recipient to generate the actual data when required. For example, a Web server allowing a user to navigate a maze might download a textual representation of the maze, along with an algorithm to generate graphical views when required, in lieu of the more traditional approach wherein the server returns a complete image for every move the user makes.

2. Reduced latency in interactions. By allowing the local machine, or another machine accessible with low overhead, to perform some of the computation, unnecessary roundtrips to the server can be eliminated, improving overall latency of interaction. For example, the maze application above could eliminate a roundtrip to the server on every move made by the user. Rather, it would require a refresh of the textual representation whenever the user wandered beyond the confines of the textual description made available to it.

3. Improved service scalability. By moving computation from the servers to the clients, the workload on each server is reduced, potentially allowing each server to handle greater numbers of clients.

4. Isolation of resource-aware code. Client platforms often supply very different resources to fulfill similar roles. This is especially true in mobile computing, where a great amount of diversity exists in the input and output devices available. Control-oriented partitioning tends to separate resource-specific code from the bulk of the application. This partitioning provides a clean abstraction to allow seamless support for a multitude of interface devices – only the machine attached to the device need know the specifics of its own capabilities.

5. Data-directed utilization of client resources. Clients that are designed without provisions for server control often benefit from simple heuristic-based resources available at the client, such as caching. However, these resources can be exploited more efficiently (to reduce communication) if the server is allowed to supply hints for their control. This is especially so in the presence of data that is at least partially algorithmically described. For example,

an "intelligent" cache manager might be allowed to understand the fact that in a weather map, only the weather data itself expires after a certain amount of time – the image of the map itself is unchanged. Upon a cache miss, only the weather data needs to be refetched, rather than the entire image. Depending on the format of the weather data, such intelligent control of client-side resources could produce a substantial reduction in communication.

Of course, other benefits are possible depending on the specific application and/or application domain. For example, user and server agents, in a mobile environment with an unreliable communications link, might be used to define error-recovery schemes that are unnecessary for a client on a local area network. The benefits mentioned above, however, are realizable in *any* communication-constrained environment with computionally-capable clients.

2.2 Application Structures

This section presents a set of techniques that applications can employ under the Oasis infrastructure to achieve some of the benefits listed in Section 2.1. The Oasis prototype enables all of these techniques, and has been evaluated partly in terms of its ability to support most of them. Sections 2.2.1, 2.2.2, and 2.2.3 describe techniques that collectively form a core set providing the flexibility to support a wide variety of control-based applications. Section 2.2.4 describes other techniques that are secondary to the core techniques because they have proven particularly useful within the Oasis framework but would not necessarily be as important in every control-oriented infrastructure, because they can be regarded as a useful extension to one of the Oasis core techniques, or because they would be useful to a class of applications that is noteworthy, but smaller than the class of applications that can exploit the core techniques. The Oasis evaluation applications exploit all of the core techniques described and the first two of the five techniques listed in Section 2.2.4.

2.2.1 Caches of Proxy Objects

Proxy objects represent a simple type of agent that can be used to build services in a control-directed manner. Most often, a proxy object is an object created at the user's machine by a service. From the point of view of the service, it is an interface that the service is aware of, and can manipulate or destroy as necessary. From the point of view of the client, the object presents an interface to one or more capabilities of the service. For example, a trivial proxy object might serve only to accept requests on behalf of the service, forward the requests to the service, and return the result when available. A more complicated object might perform local computation and return a result itself in situations where contact with the server is unnecessary. Proxy objects are cached on the machine where they exist, and may be destroyed, if necessary, to reclaim resources. Java applets [10], supported by many Web browsers, represent a mechanism by which simple proxy objects, isolated from all other services in the network, can be implemented.

Although proxy objects are most useful to services, the duality of the user and server agent models implies that a client machine could also create a proxy object at a server. For example, a resource (such as a Web browser) or local application running on a user's machine may wish to establish an object at a server's machine in order to allow the service to manipulate the resource directly. Such a scenario might arise if a browser wished to be instructed to fetch a document automatically upon the satisfaction of some condition at the server (such as the receipt of some data, or drop in load to an acceptable level).

It should be noted that proxy objects are best suited to their role as client-side service interfaces, because they enable the user, and agents acting on behalf of the user, to interact with multiple services and resources simultaneously, using the set of objects known to the user's machine. Such interaction might, for example, allow services to share data with one another, or to augment each other. One such scheme is the notion of *composable* overlay services.

2.2.2 Composable Services

The success of the Web as a hypermedia[2] information system suggests that users often wish to access several related services simultaneously. Services that are related to one another usually share one or more underlying data types. For example, a map server, a weather service, and a service that computes the distance between two points all share the underlying notion of a geographic location, perhaps specified by a latitude and a longitude. If all three services are accessed as part of the same task (e.g. planning a road trip), it is likely that that some of the information access may be unnecessarily duplicated. In the example, an access to the weather server might wastefully fetch a map that is essentially the same as a map accessed earlier by the map service. In addition, the distance server might not provide a map at all, but the user may wish to see the shortest distance between two cities plotted on the map. In order to solve these two problems, it is desirable that services that share an underlying data type be able to interact

[2] *hypermedia* is used to indicate a hypertext system whose documents contain references to other documents, cf [14].

with, or to be able to control, each other. Services that can cooperate in this manner to manipulate data of a commonly known type are known as *composable* services.

The services presented in the example above actually represent a special case of composable services, wherein each service is able to simply augment the information produced by a previous one. For example, a weather service may simply be able to overlay information on the output of the map server. Such groups of services are known as *composable overlays*. However, services may wish to interact even if they are not overlays. For example, an encylopedia may wish to instruct a map browser service, already present on the user's machine, to display a map of the area to show the location of a historical monument the user just inquired about.

Service composability is thus a desirable characteristic for two reasons. Firstly, it helps to reduce the amount of communication that is performed, by allowing reuse of data, and of code (e.g. the procedure needed to draw a line between two points on a displayed map) that is already cached on the client machine. Secondly, it allows the client to construct composite applications built using client-side aggregating agents to construct capabilities exceeding those of any of the components. Such "client-side scripts" are one type of user agent that may be constructed by the user, or possibly supplied by a third-party service provider.

2.2.3 Controllable Resource Managers

The computationally capable clients of the target environment provide a set of resources available for use by the owner of that client. Traditionally, such resources are managed by algorithms that attempt to optimize their use over a wide variety of scenarios. Examples of such resources include memory and disk space for caching, power, communication bandwidth, display real estate, etc. However, the managers of these resources are often able to benefit from advice provided by the services that use them. For example, a cache manager that is aware that only the weather data in a weather map expires after a few hours, but that the map itself always remains valid, may be able to use this information to eliminate unnecessary data fetches. Similarly, a meeting-scheduling service may wish to inform the client that the result of a request will not be available for several hours, thereby allowing a mobile client to save valuable power by turning off its receiver in the interim, if it so desires.

To a service or agent, resource managers can simply be viewed as "services" located on the client machine itself. Resource managers can thus be represented by controllable objects much like the proxy objects created and managed by real services. These objects serve as a mechanism to allow input of control hints (such as an algorithm for computing the time at which a piece of data expires) by server agents from services that use the resources. Although the management of such control objects is somewhat different from that of proxy objects (e.g. they cannot be cached at remote locations, and they cannot be expelled from the client machine's namespace), this scheme allows for an interface to the user's resources that is consistent with other interfaces to remote services that an agent might encounter.

2.2.4 Other Techniques

User and server agents, as defined, are capable of arbitrary computation, although their access to system resources may be limited in the interests of security. They can therefore be used to construct arbitrary customized communication paradigms. Some such paradigms are listed below:

- *Filtering* allows the application of an arbitary encoding or transform to the data being exchanged by the server and user. For example, a server agent, exported to the user, could decode data compressed in an arbitrary and perhaps type-specific manner by the server. A user agent, exported to the server, might eliminate multiple round trips to the server by allowing a client to to make packaged queries, such that the entire series of queries would be executed at the server, and only the final result would be returned to the user.
- *Precondition verification* might be performed by a server agent on input supplied from the user, to remove invalid requests before they are sent to the server.
- *Shared workspaces* could be implemented by a service by exporting appropriate server agents to each of the users of the workspace. The server agents might interact with the service to ensure that the user's copy of the shared document is kept up to date.
- *Fault tolerance* in the presence of unreliable communications could be provided at a service-specific level by use of appropriate server agents. Such agents might encode predetermined strategies for error recovery in the face of a dropped connection, such as the ability to fall back to a backup server "recommended" by the primary.
- *Reconnectability* and resynchronization of user and server states after a disconnection could also be handled through the use of agents to negotiate the reconnection. Such disconnections may arise in contexts other than that of unreliable communication. For example, agents could be used to implement functionality similar to that of the teleporting system [8]

wherein the user may intentionally "disconnect" from his environment and later reestablish the connection from a different site.

2.3 Related Work

This research investigates the hypothesis that "control relationships" between application components constitute an important factor in partitioning an application across one or more low-bandwidth links. The choice of control as a discriminant is unique to this study. Each related work described falls into one of three categories: projects that involve a similar artifact, but for a different purpose; projects that address the problem of application partitioning, but decide on application structure via a different scheme; or projects that embody a similar, but much more limited, model of application structure in a communication-constrained environment.

2.3.1 Similar Artifacts

There have been a number of efforts, contemporary to the Oasis research, to produce computationally enabled Web browsers, and these comprise the most significant portion of the related work. The most successful such browsers are HotJava [49], Netscape Navigator[32], and Microsoft's Internet Explorer[31]. As Netscape Navigator and Internet Explorer simply integrate technology developed for HotJava, they are not described further. Other, lesser known, early efforts at computationally capable Web browsers include those of the dynamic documents group at MIT [23], and the Mobisaic project at the University of Washington [54].

The Hot Java project aims to provide interactive content via the Web. Dynamic documents allow retrieval that adapts to the available resources of a mobile client, whereas Mobisaic provides documents that react to contextual information, such as the location of the client. None of these systems treat the client host, or its resources, as being shared among applications. No resources, other than the Web browser itself, are available for control by external service providers. As they target different application classes, all three systems differ in one important way from Oasis – support for computationally active documents is provided in the Web browser itself, rather than in a separate entity, such as the Oasis proxy. The Oasis architecture better supports the role of the client machine as the point of convergence between multiple clients, network services, and resources, as is required if control interactions between these entities are to be feasible.

The three systems mentioned above thus provide an enabling technology for limited *server* agents, wherein limitations arise from differences in the envisioned functions of the agents. However, none of them offer any support for *user* agents. All three systems embrace the notion of a monolithic computationally-enabled Web browser. As browsers, by definition, are usually not executing on the part of the network where user agents would ideally be hosted, this approach is incompatible with the Oasis goal of providing a single, homogeneous environment to accomodate both user and server agents.

Another architecture that is similar to Oasis at an ideological level is the system once advocated by General Magic, whose visions are described in [59] and [58]. General Magic proposed an agent-based world where each item of data was to be managed by a computationally active entity. These entities were written in a language called Telescript, and moved from host to host to negotiate with other entities. Although General Magic's system might eventually have provided all the capabilities necessary to support both agent models, the system was abandoned as it became commercially irrelevant – General Magic's universe for agents was completely unrelated to the World Wide Web, and thus the system was unable to exploit its largest source of potential applications. It is unclear how much of the original system was implemented and evaluated. Furthermore, General Magic did not investigate the importance of control interactions in application partitioning, as advocated by this thesis. General Magic recently released a new agent environment called Odyssey [16] featuring better integration with the Web. However, Odyssey is more limited than its predecessor, is still less tightly integrated with service deployment over the Web than Oasis, and fails to consider the issues of control interactions or communication-constrained environments. The IBM Aglets Workbench [27], another platform similar to Odyssey, differs from Oasis in essentially the same respects.

The AT&T GeoPlex [5] system is one that is contemporary with and structurally similar to Oasis. Under GeoPlex, clients and servers are separated by a network of proxies. The network of proxies mediates all communication, and a secure "core" of proxies and services imposes network security, monitoring, load management, and enhanced network services. It is noteworthy that the GeoPlex environment was intended neither for application partitioning nor to facilitate the adaptation of online services to communication-constrained environments. However, the architectural similarity between GeoPlex and Oasis suggests that the infrastructure required for these systems has benefits other than the ones investigated by this research. Consequently, the independent evolution of GeoPlex increases the likelihood that the internet will one day host the kind of network infrastructure necessary for universal acceptance and deployment of the capabilities of both systems.

The use of protocol proxies to filter traffic destined for low-bandwidth or resource-poor environments has been examined by a number of efforts. Many, such as OreO [9], WebExpress [19] and GloMop [15] have targetted or examined World Wide Web access. “OreOs” built from the OreO toolkit are filtering HTTP proxies that act as “HTTP stream transducers,” and may modify accessed data in ways appropriate to the needs of the requesting client. WebExpress consists of a pair of HTTP proxy servers designed to proxy Web access from wireless clients. Its filtering operations and specialized network protocol between proxies attempt to reduce the bandwidth required for page retrieval. GloMop proxies “distill” various data types to approximations requiring lower bandwidth to retrieve. A more general approach is taken by [60] in a framework to allow filtering of any IP-based protocol. All of these approaches are conceptually equivalent to the use of filter agents in Oasis. Although Oasis applications make extensive use of filter agent capabilities as an implementation convenience, Oasis filters are just one of a set of control-oriented building blocks available to Oasis applications. In addition to hosting filters, Oasis proxies act as points of application composition, as points of application interaction with resources available within the network, and as hosts of application code and data. The aforementioned filtering approaches either do not allow or do not encourage (and therefore do not facilitate) the diverse use of proxies in the manner intended by the Oasis infrastructure.

Lastly, the ANTS framework from the Active Networks group at MIT [57] takes the approach that network routers should be computationally capable, to enable rapid deployment and testing of new network protocols. Although this is theoretically comparable to the introduction of computationally capable proxies into the network, the implementation details of the system differentiate the environment considerably from that of Oasis. ANTS is targeted at network protocols, and exported code is thus constrained to operating at a very low level, with severe performance and resource constraints. Application developers are consequently burdened with much additional complexity and a comparative lack of resources. Network routers are usually critical, heavily-loaded environments that cannot afford to allocate many resources to the needs of an individual application. Thus, while this system can support limited server agent capability, it is not well suited to the requirements of application partitioning on a long-term (i.e. more than a few seconds) basis.

2.3.2 Other Approaches to Application Partitioning

A small number of projects are examining the issue of partitioning an application across a wireless network. These projects focus on, and attempt to modify, the data access characteristics of applications through techniques such as prefetching [28], selective data transmission [55, 23], or prioritization [55, 4]. Thus, these projects focus on a problem similar to that addressed by Oasis, but for a distinct class of applications. This research is complementary to these approaches, and addresses those applications, or components of applications, that can be optimized by avoiding or reducing data movement through control-oriented partitioning. Techniques such as prefetching and prioritization could still be applied to those applications or components that are fundamentally pure communication (class II) components (See Section 2.1.3.), and thus cannot be partitioned along control-oriented lines. It is interesting to note that these earlier application partitioning models, which concentrate on data movement in the context of mobile computing, have often partially incorporated proxies (e.g., [56, 6]) as a means of addressing problems that are strictly issues of control rather than of data movement. For example, proxies in [56] and [6] both serve to address situations wherein a repository of application state is best held on the server network and manipulated remotely by the client.

The principle of application partitioning according to control considerations has itself arisen in limited contexts in prior work. The most general such instance is Oracle's "Oracle in Motion" product [38, 20] which supports access to databases from mobile hosts under Microsoft Windows, through their "client-agent-server" model. The model provides support for basic user agents, and points out that user agents can often lead to reduced communication overhead and better application latency. This is because a user agent can sometimes substitute a single roundtrip over a wireless network where multiple ones would have been required by a client-server model in performing the same transaction.

A similar approach, in the context of application deployment in a mobile computing environment, is that of Hokimoto et al [18], who propose the control of applications partitioned via the use of "object graphs." An object graph is an abstraction describing an arrangement of data filters that can be controlled from a mobile computer via a "control object." Feedback from the part of the application on the mobile computer is used to configure interactions with the server-side part of the application to be appropriate for the resources available to the mobile device. In effect, this architecture advocates the partitioning of the application into relocatable filter blocks which are manipulated by a user agent and may be composable. Server agents and local resource interaction are not addressed.

2.3.3 Similar Models of Application Structure

The Rover Toolkit for mobile applications [22] provides some abstractions that are similar to those available to Oasis applications. In particular, Rover requires applications to encapsulate important data in "relocatable dynamic objects" (RDOs) which can be moved between a client and a server upon request by the application. All updates to data contained in RDOs are handled through Rover's queued RPC system (QRPC) which provides operation logging, rollback, and replay and supports a particular concurrency control mechanism. The RDOs may contain methods by which the encapsulated data is manipulated at the client or server – in effect, data is often relocated along with a control interface to that data. The Rover Toolkit can be thought of a more elaborate counterpart to the combination of Oasis proxy objects and the controllable cache manager described in Section 4.4.1. The Rover Toolkit is not integrated with the HTTP protocol, and facilitates coherent data access and update handling that is not supplied by Oasis. Rover's functionality is centered around a paradigm similar only to that of Oasis proxy objects, as Rover does not address the other control-oriented techniques that Oasis supports.

Partitioning along control-oriented lines is also sometimes found in carefully designed applications that deal with large data sets and regard current wired networks as a bottleneck, or as a resource to which access should be optimized to ensure scalability. An example of such a system is the Weather Anchordesk [7]. The partitioning that has been chosen by the designers of the Anchordesk is well suited to wired networks and to the server technology underlying the environments in which the Anchordesk has been deployed. In the Anchordesk system, an "integrator" situated close to the sources of weather data merges the various data sets into a weather map, forwarding the composite result to client browsers. Each weather browser acts as a means to control actions occuring at the integrator. The Anchordesk partitioning reduces aggregate network load by transmitting only composite data to clients. On the other hand, this partitioning prevents client browsers from caching any components of the data received beyond the validity of the shortest-lived component of the composite result. This limitation is acceptable due to the high speed at which an updated composite weather map can be retrieved over the wired network. As described earlier, a different partitioning might have been better suited to this application in a mobile or other communication-constrained scenario. Oasis represents a framework that supports either type of partitioning. In Section 4.5 a reconfigurable Oasis weather browser, similar in structure to the Weather Anchordesk, is used to demonstrate the benefits of repartitioning for communication-constrained scenarios.

2.3.4 Object-oriented Technology and Control

Lastly, it should be mentioned that "control-orientedness" is related to the notion of "object-orientedness," about which there is a huge volume of literature. The latter suggests the encapsulation of data entities such that they can only be accessed by the appropriate methods on that data. The former is a higher-level notion dealing with the division of functionality among the components (and therefore among the objects) of one or more applications. This partitioning is based upon interactions between the various entities. Object-orientedness provides a low level partitioning of data manipulation along object lines. Control considerations, when applied to an object-oriented design, will thus usually suggest a partitioning of the objects. Therefore, the two schemes are compatible with each other. It is hardly surprising that the Oasis architecture incorporates a small object system, based largely upon the object system of the Java language used to implement Oasis, whose objects provide a means to interact with the agent code they encapsulate. A commercial implementation could even have used a system conforming to the well-known Common Object Request Broker Architecture (CORBA) [35, 47], although such an implementation is beyond the scope of this thesis.

Chapter 3

Architecture of Oasis

This chapter describes the architecture of the Oasis system, and its support of the agent models. Section 3.1 presents a short overview of the system, and subsequent sections present design and implementation details. Section 3.2 details the basic agent support provided by Oasis. Section 3.3 describes agent composition. The use of local resource managers is covered in Section 3.4. Section 3.5 describes the integration of the Oasis environment with the World Wide Web, and documents the use of filters, which prove particularly useful in the context of the Web. Lastly, Section 3.6 illustrates the construction of agents through the example implementation of a simple but useful agent.

3.1 Introduction

Oasis was designed to provide a completely homogeneous networked framework to support the partitioning of Web-based applications into user and server agents spanning multiple machines. This goal has been achieved through the construction of a single ubiquitous HTTP proxy server enhanced to accomodate agent functionality. The server is interposed between the client's Web browser and the service provider, at the client's host, on the server network, and possibly at points in between, as shown in Figure 3.1.

The Oasis proxy server, in addition to acting as a normal HTTP proxy, must also support agent recognition and execution, composition of agents, agents dedicated to managing physical resources, and services to support agent integration with the World Wide Web service paradigm.

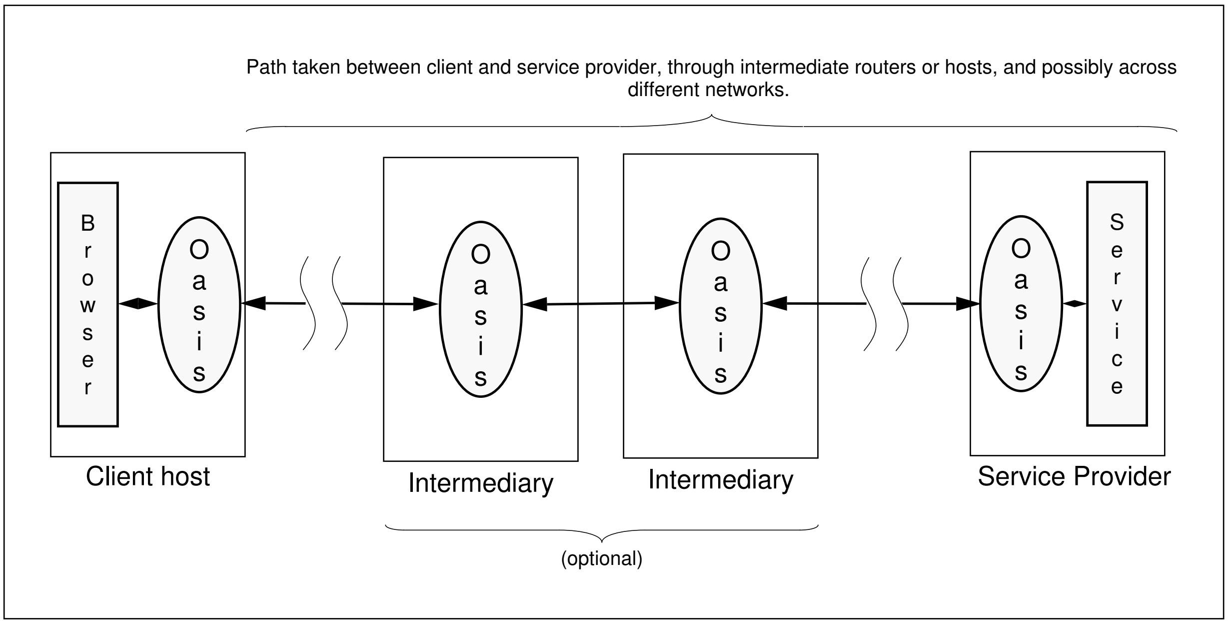


Figure 3.1: Oasis proxies may be located at the client's host, the service provider's host, and/or at various points on the path between the two.

The core proxy server consists of a minimal set of facilities necessary to host agent functionality and provide integration with Web applications. The rest of the server's functionality is implemented by trusted agent modules, as shown in Figure 3.2. The figure shows an Oasis proxy with caching and advertisement elimination provided by executing Oasis filter agents. As shown, the proxy examines data received from external data sources, and intercepts agent code when appropriate, creating Oasis agents. Some such agents are filter agents, like the caching and advertisement filter agents illustrated. Filter agents, once initialized, may examine data requests received from a data consumer and may satisfy or modify these requests themselves rather than allow the proxy to forward them to the appropriate data producers. Although not illustrated in the figure, filter agents may also modify or consume responses received from data producers prior to the forwarding of those responses to the requesting data consumers.

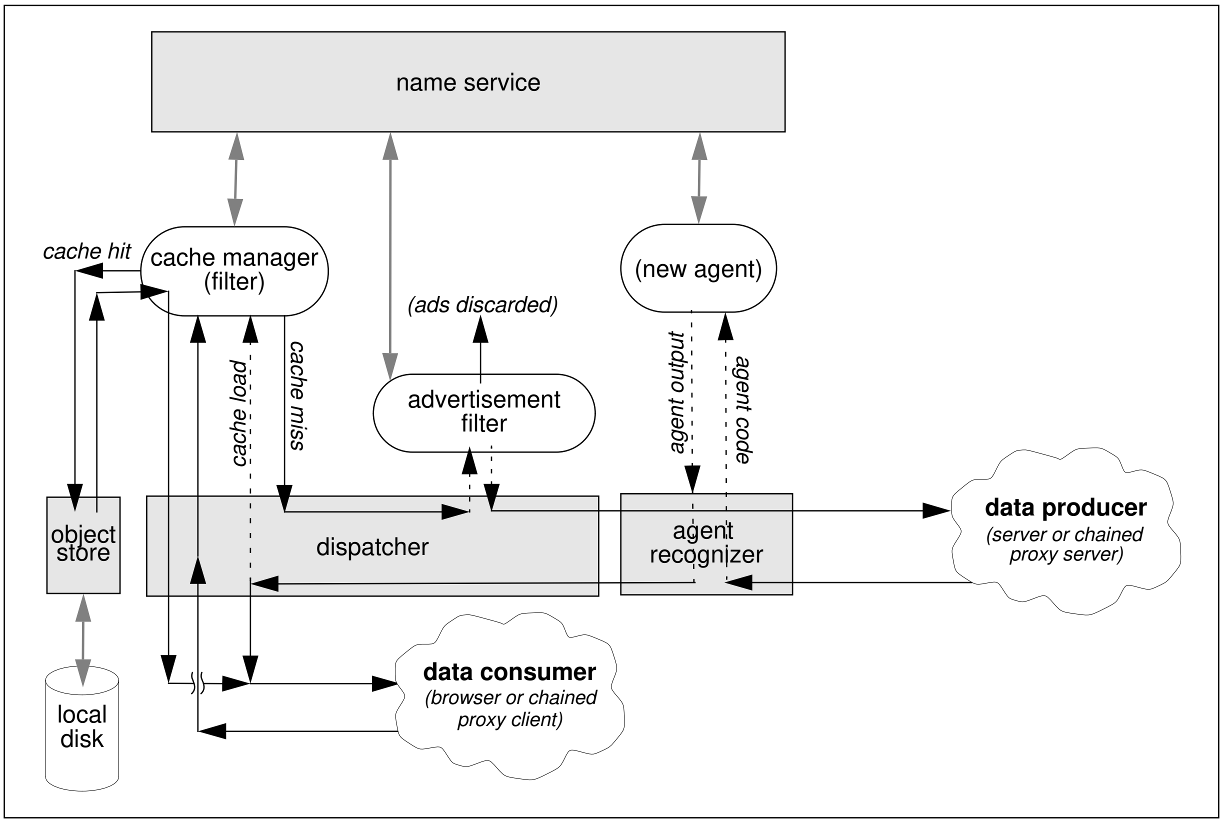


Figure 3.2: The structure of an Oasis proxy, shown with two agents loaded – a trusted cache manager, and an untrusted advertisement filter. The grey components are those of the basic proxy. (Other functional components are dynamically loaded as needed at runtime.) The solid black arrows represent data paths, whereas the dotted arrows represent additional optional data paths. The grey arrows represent interactions between subsystems.

3.2 Basic Agent Mechanisms

Oasis agents consist of compiled Java [50] bytecodes exported by a service, a client, or possibly even another agent. The movement of agents is integrated with the standard World Wide Web paradigms for downloading data from server (i.e. fetching a URL) and uploading data to a server (i.e. posting form data). Oasis proxies on the data path between client and server examine the data passing through, and, when appropriate, intercept agent code and allow it to execute.

Oasis supports two basic agent types – "User agents" and "Server agents" – which correspond directly to the user and server agent models discussed earlier. User agents usually perform work under control of a client, and wish to execute as close to the service provider as possible. Server agents, on the other hand, usually act under control of a service provider, and execute as close to the client as possible. Both types of agent are tagged, at their source, with an appropriate `Content-Type` header. Every Oasis proxy examines the headers of objects passing through it in order to recognise agent code and decide whether or not hosting it is appropriate.

Whether or not an agent is hosted by a proxy is determined by the type of the agent and the position of the proxy in the chain of proxies between the client and service. By appropriately augmenting requests and responses passed through, proxies communicate their existence to other proxies in the chain. Each proxy is thus able to determine whether or not it is at either end of this chain. Proxies at the ends of the chain host user and server agents; the agent loading paradigm allows for proxies elsewhere to host other agent types (such as global agents, described as an example in Section 3.2.4) built from the primitive agent types.

3.2.1 Agent Recognition and Loading

By default, an HTTP proxy server accepts requests from clients, requests data from remote servers on behalf of those clients, and then returns the data received to the requestors. Similarly, it also allows clients to make requests for data to be uploaded to servers, and passes that data on to its destination. The Oasis proxy, in the course of performing these actions, also takes the following additional action in order to support hosted agents:

1. Headers of an incoming request are checked to see whether `Accept: app/JavaAgent` is specified. If this header is present, the Oasis proxy knows that the requestor is also capable of hosting agent code. Thus, the proxy itself is not the agent host closest to the client.
2. If the request is for data download (i.e. HTTP "GET") the Oasis proxy adds `Accept: app/JavaAgent` to the request headers, and passes the request through to the next server in the chain. Once the response is received, its `Content-Type` header is checked to see if the response is tagged `app/JavaUserAgent`. If so, the agent code (i.e. the body of the response) is intercepted by the Oasis proxy and becomes a live agent, executing as close to the service provider as possible. (If closer execution had been possible, the response would have been intercepted prior to arrival at the proxy.) If the requestor of the data has not specified (in the request) that it is capable of hosting agents, the Oasis proxy must also similarly recognise server agents (`app/JavaServerAgent`) and intercept them,

as it happens to be the agent-capable site closest to the client.

3. If the request is for data upload (i.e. HTTP "POST") the Oasis proxy checks the `Content-Type` header to determine whether or not the uploaded data is tagged as being either type of agent. Any such agents are intercepted and executed.

4. All intercepted agents are supplied with a handle to the unsatisfied data request pending due to their having been intercepted. The newly constructed agent is responsible for generating the response to the request that caused it to be loaded. Often the response is simply a text message to confirm that the agent was started successfully. This response propagates back through the chain of proxies to the original requestor.

Note that this scheme is not completely symmetric due to the need to remain compatible with standard HTTP. In particular, no *automatic* mechanism exists by which a user agent being uploaded can reach the agent host closest to the service provider's site. Rather, a user agent is executed by the next available agent-capable host in the chain of hosts between uploader and service provider. This limitation is not restrictive and can be overcome in a multitude of ways by any agents needing to do so. For example, an agent might simply choose to propagate itself hop by hop until it found the last agent-capable host in the chain. Alternatively, the agent, having been started on the first proxy in the chain, could request a fetch of itself via the regular HTTP mechanisms, causing a regular HTTP download request to propagate up the chain of proxies — the download request itself would be satisfied and intercepted by the last proxy in the chain, as desired. Most of the Oasis evaluation applications implemented use the latter technique in situations where an agent wishes to execute as close to a service provider as possible.

Every agent handled by an Oasis proxy must be a subclass of the `oasis.agent.Agent` class, which serves to provide a standardized means of agent initialization, as well as to serve as a repository for a few convenience routines needed by all agents.

3.2.2 Minimal Agent API

The `oasis.agent.Agent` superclass is the superclass of all agents. As such, it enforces that all agents will have a number of methods in their definition, and provides default definitions of some methods that are useful to an agent, or to other agents that wish to interact with it.

The following methods can be implemented in the body of an agent:

- `main` – The body of the agent. This is the entry point for the agent after initialization completes.
- `getAgentInfo` – This method is called by anyone who has a handle to the agent and needs to find out more information about it. The information is usually returned as an identification string.

The following methods are provided for convenience and agent administration:

- `setNameservice` – This method is used to give the new agent a handle to a name service, through which it may locate other agents, register itself, etc. Prior to invocation of `main`, this method is called with a handle to the very simple name service provided by the Oasis proxy server. (Nameservers are described later, and agents may choose to provide a more complicated name service themselves.)
- `sendHttpTextHeader` – After loading, many agents return a text document to the original requestor, usually to indicate successful loading or to provide other information of interest. This immutable convenience routine sends an HTTP response header to the requestor indicating that a text document follows. After calling this, an agent can simply output the text strings that constitute the body of the response.
- `getURLSource` – This method returns the URL from which the agent code was downloaded, if it is known. This may be useful to the agent itself, as well as to any services wishing to implement access control lists based on agent origin. This information is recorded securely by the Oasis proxy prior to the invocation of `main` and cannot be altered once set.
- `createTopLevelWindow` – This method is provided to allow agents a mechanism by which they can display graphical output. Windows created via this mechanism are destroyed, along with the threads of the agent, when the agent's main thread exits or is destroyed. The automatic destruction of created windows is done via a companion call, `deleteWindows`, that is called by the Oasis proxy.
- `dispatchNewURL` – This method allows an agent to obtain a URLConnection object for access to URL data via the proxy. It is purely a convenience method, as other means of accessing URL data via the proxy are also provided.

Other agent APIs, such as the `FilterAgent` API, described later, are subclasses of `oasis.agent.Agent`.

3.2.3 Inter-Agent Interaction and Security

The Oasis proxy provides a mechanism for agent interaction, but does not set any policies to govern it. It is expected that policies, where required, will be set by developers whose agents must interact with each other. Oasis agent interaction is based upon the existence of a name service coupled with Java method invocation. The Oasis proxy provides a simple name service with which agents may choose to register under a well-known name. Other agents may then contact the registered agent by resolving that agent's name and using the returned handle to invoke methods on public interfaces of that agent. The functions provided by the name service are minimal – for example, the service does not enforce or exploit hierarchical namespace structure. This is because, in practice, it is expected that the name service will be used for bootstrapping purposes only – more sophisticated name services can easily be implemented by an agent; the native name service need only serve to provide a means for locating a more comprehensive name service, and for registering basic host resources made available by the base proxy itself. For example, the Oasis cache manager, which is implemented as an agent (and can act as either user or server agent as needed), utilizes the simple name service to locate the proxy host's public object store, which it uses to cache objects fetched.

Name services and agent interaction in general necessitate the introduction of an access-control system in Oasis, as it is often not desirable that all agents have equal access to all the services and agents registered. The Oasis proxy resolves this problem by tagging all agent code with the URL from which it was downloaded. The source of the agent code can be used to determine the level of trust accorded that agent. In theory, this permits an access-control list security infrastructure; in practice, however, only two levels of access are currently enforced – the untrusted access level, for agents downloaded from the network, and the trusted access level, for agents loaded from the local disk. The latter level is necessary for agents that need greater access to Oasis resources, e.g. an agent to manage the local disk cache. In the current Oasis implementation, all agent interaction performed via method invocation is constrained to occuring only through trusted interfaces, i.e. each Oasis proxy must choose to accept as trusted any interfaces via which two agents may communicate. This restriction eliminates arbitrary agent communication by default. However, any proxy that wishes to allow less restrictive inter-agent communication need only accept the security of an interface defining methods for arbitrary message passing or some variant thereof.

A name service should also implement a garbage collection policy for the name space it maintains. If an agent does not register with any name service, that agent will run to completion, and will thereafter be marked for garbage collection by the Java runtime because no references to the agent's code remain. However, name spaces retain a reference to agents registered with them, and thus provide a means for agent code to remain dormant, in the absence of a running agent thread, until invoked by another agent or by the proxy itself. Ordinarily, well-written agents should unregister from the name service when they are no longer needed. Occasionally, however, it may be necessary to remove code from agents that terminated without unregistering properly. A policy, such as selection according to the time elapsed since the last resolution of an agent's name, is necessary for automatic detection and elimination of such agents. Alternatively, agents known to have terminated improperly can be removed specifically if their registered names are unregistered by a proxy administrator, or by an another agent with appropriate authority. If a name service does not implement a garbage collection policy, the proxy administrator can force garbage collection of all agents registered with that name service by unregistering the entire name service from the native name space implemented by the Oasis proxy. Unregistering a name service disconnects all of its clients from the tree of reachable objects, thereby marking them all for garbage collection unless they are referenced elsewhere.

3.2.4 Arbitrary Agent Placement

The Oasis agent recognition paradigm has been optimized for the two most common agent types, namely user and server agents. However, the tag-based agent recognition mechanism does not restrict agent implementations to only those two models. Other agent positions can be obtained when explicitly desired by the implementor. In general, this is accomplished by constructing the agent to be aware of its own progress through the network, and to take charge in determining where it is to execute. As an example, consider the implementation of global agents, which execute on every agent-capable host between the service provider and the client, e.g. in a multi-level caching scheme. Such an agent can be implemented by an agent injected into the network by a service provider yet tagged as being a user agent. Any agent-capable host on the path between service and client would attempt to intercept and host such an agent. Every time such an interception occurs, the agent need only initialize itself, and then fulfill the pending I/O fetch request by returning its own code (rather than the usual textual notice of successful completion), allowing subsequent interceptions by links further down the chain.

3.3 Agent Composability

Software agents can be composed in two basic ways, both of which are permitted by Oasis. Firstly, an agent may encapsulate, or overlay, another agent. The encapsulated agent continues to function and to provide its original services, but the data input to the encapsulated agent may be preprocessed by the encapsulator, and the data output by the encapsulated agent may be post-processed by the encapsulator. The encapsulated agent may, if desired, simultaneously continue to be available in its original form, without the added effects of the encapsulator. Alternatively, agent composability can be achieved by requiring agents to explicitly advertise their input and output types. When composite functionality is required, a new agent is constructed to stream data into and between the appropriate component agents.

Although both types of agent composition require type-compatibility checking at interfaces between components, the development of data-type characterization and enforcement for the Web is beyond the scope of Oasis and is being addressed by efforts such as [37]. Type compatibility for agent encapsulations is enforced in practice by the Java class hierarchy – encapsulating agents usually must either be a subclass of the encapsulated class or share an ancestor with that class, thereby ensuring type compatibility between layers. If this relationship does not hold, encapsulation can still occur, but the result is likely to be of limited usefulness as the encapsulation is not transparent to entities expecting the interface presented by the formerly unencapsulated agent. Type compatibility for the second agent composition scheme is not enforced by any aspect of Oasis; the agent composing the components is responsible for checking compatibility when setting up the data stream.

Many agent encapsulations occur with the cooperation of a name service. One of the simplest ways for an encapsulation to occur is for an encapsulating agent to register itself under a name formerly held by the encapsulated agent. As the Oasis proxy is a proxy for World Wide Web information access, the proxy provides specialized support, and a specialized namespace for an important class of composable agents tailored to encapsulating WWW data access requests. These agents, called filter agents, are described in Section 3.5.1. Filter agents are a class of agents that control a resource that is local to the machine hosting the Oasis proxy. In the case of filter agents, the local resource happens to be the proxy itself; however, the Oasis proxy also allows for other agents that wish to provide a controllable interface to a local resource.

3.4 Management of a Host's Physical Resources

The management of local physical resources fits naturally into the Oasis paradigm of modular agents providing incremental functionality to the system. Java code can very easily be written to encapsulate access to a physical resource, and integrated with the Oasis proxy by embodying that code as a local agent, i.e. a trusted agent loaded from the local filesystem. Only two security levels are available at present, and all local resources are protected from direct access by downloaded agents. A local agent written to manage a resource need only register itself with the nameservice (either the Oasis nameservice or a more comprehensive one, if available) and make available the appropriate control interface. An example of a controllable physical resource managed by a local agent is the Oasis cache manager, which is able to consult server agents from the appropriate servers when making cache management decisions relating to downloaded objects.

3.5 Agent Integration with the World Wide Web

The Oasis proxy is an HTTP proxy. As such, it is a service that can benefit from exposing control interfaces that permit behavior customization by agents. It is also a conduit for data that constitutes the communication component of Web-based services; in order to optimize communication or customize content, agents often require access to this data. In order to satisfy both of these needs, the Oasis proxy supports a class of agents known as *filter agents*, which operate according to the `oasis.agent.FilterAgent` API.

3.5.1 Filter Agents

Filter agents are agents that encapsulate accesses to World Wide Web data and services via the Oasis proxy (i.e. they can be regarded as "filters" acting in between the consumer and producer of the data). A plethora of useful services has been implemented by filter agents, including services such as transparent access anonymization[2], data adaptation based on available bandwidth, data redirection, intelligent data caching, etc. Oasis supports this important class of agents by exposing a limited control interface that can be used to encapulate the processing that is performed to service a request, as well as by providing a limited namespace to allow for agent encapsulation of less than the complete URL space (a performance optimization).

Filter agents are loaded and initialized in the same manner as any other agent. They are recognized by the proxy, activated, and typically return a text document notifying the original requestor of successful loading. Once activated, a filter agent registers itself with the Oasis proxy's *dispatcher*, which manages accesses to URLs. At this time, the filter agent may also choose that its registration be assigned the default priority used by most agents, or a higher priority used by caching filters, which require access to requests before other filters that may forward a data fetch request to an external data source. Upon demand by the dispatcher, the filter agent must be prepared to return an agent which is a subclass of `FilterAgentUC` (the superclass of all filter agent URL connections) to be used in encapsulating a URL data connection.

Whenever a request to fetch data from, or post data to, a URL is received, the dispatcher determines the set of agents that have registered interest in encapsulating that URL. The dispatcher then creates a simple URL connection (an interface to all potential interactions with the URL or its data), and allows it to be encapsulated by all appropriate filter agents in increasing order of their registration priority. Thus the filter with the outermost encapsulation, and therefore first access to data requests and last access to returned responses, is the filter of the highest priority from amongst those that registered interest for a particular URL. Once all appropriate encapsulations have been performed, the resultant URL connection is used for all interaction with the URL by the proxy. Because each encapsulation allows filter agent control of all operations that can be performed on a URL (e.g. opening a connection to the URL's host server, reading the data for the URL, or posting data to the URL) it enables a rich set of completely composable agent functionality.

Figure 3.3 depicts the use of two composable filters (a caching filter and a compressing/uncompressing filter pair) to implement a caching proxy server that supports compressed data transfers. In this example, every URL requested by a client is examined by the dispatcher. The dispatcher may determine that the request is to a part of the URL space not managed by the caching or compression/uncompression filters, and if so, will exclude one or both of those filters encapsulating the URL. Most URLs are encapsulated first by the compression/uncompression filter, and then by the caching filter, i.e., all operations on those URLs are passed first to the caching filter, and may then be passed on to the compression/uncompression filter, before eventually being passed onto the basic URL fetcher implemented in the Oasis proxy. Thus, a request to read the contents of the URL is handed first to the caching filter, which may return the contents from its cache, if present. If not, the request is passed through to the compression/uncompression layer, which simply forwards the request. The request is eventually handled by the Oasis internal URL fetcher. The URL fetcher of the first proxy in the chain simply forwards the request to the second proxy, where the encapsulation and forwarding process repeats. Eventually, the request reaches the URL fetcher in the second proxy, which contacts the server providing the data for that URL. As the data flows back to the requestor from an internal URL fetcher, it first passes back through the compression/uncompression layer. On the proxy that is closer to the data source, this layer observes the arrival of raw data and compresses it, marking it as compressed, before passing the data onward. On the proxy closer to the user, compressed data received is uncompressed before being passed onward. Back at the first proxy, data returned from the uncompressing filter flows back through the caching filter. The caching filter passes this data back to the requestor, while

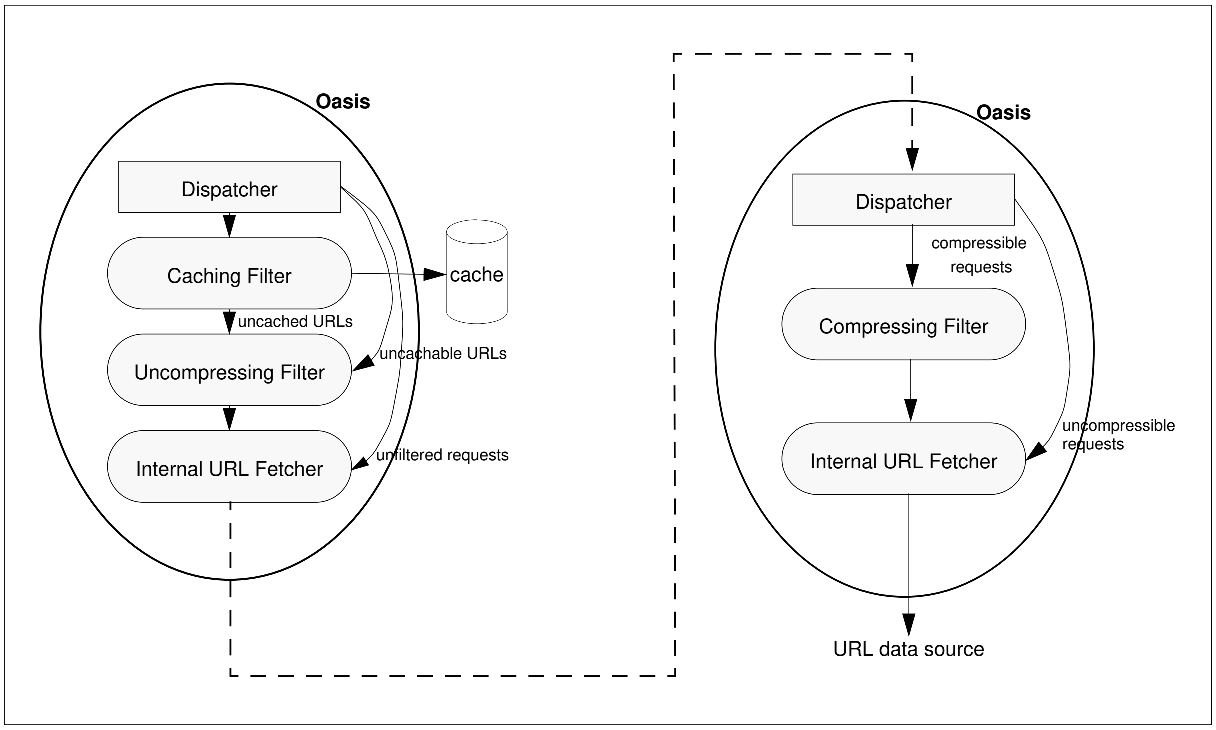


Figure 3.3: The path of a request through a pair of Oasis proxies running filters that implement a caching Web service with compressed data transfers. The path of the returned data (not shown) is just the reverse of the path followed by the request.

simultaneously recording the data in its cache, if appropriate.

Filter Agent APIs

Filter agents, as described earlier, are composed of two parts — an administrator and an implementation. The administrator is an agent that is loaded to activate the filter. It serves to register interest in the appropriate parts of the URL space, and to provide a method which, upon request, will take a URL Connection object from the Oasis proxy and encapsulate it with an implementation agent, which is the implementation of the filter itself. Each filter only requires one instance of the administrator, which remains active until it chooses to deactivate itself, whereas a unique instance of the implementation is created for every URL of interest, and remains active only while that URL is being accessed, as determined by the continued presence of a source and sink for the data retrieved. An administrator agent also usually provides a mechanism by which the filter agent as a whole can be deactivated. It may also provide other administrative functions, such as detecting and avoiding the existence of multiple instances of the same agent in a given proxy.

Administrator agents must be a subclass of `oasis.agent.FilterAgent`. This class provides predefined methods for interaction with the dispatcher, and a predefined method which can be used, by an administrator or by any authorized agent, to deactivate the filter; it also enforces the requirement that administrators provide a method to be used in encapsulation a URL data connection upon request. Since the administrator API is fairly simple, it is not presented in detail here.

Filter implementations must be a subclass of `oasis.agent.FilterAgentUC` (i.e. "Filter Agent URL Connection"). This class mirrors the interface of java's own `java.net.URLConnection` and serves to provide default implementations of all the methods required for a null encapsulation of a URL data connection, as well as a few convenience functions (such as access to the name of the URL associated with the filtered connection). Filter implementations are constructed by overriding the default implementations as necessary. Overrideable default methods are provided to allow reading or modification of every HTTP header field as well as of the data being transferred.

Although filter agents normally deactivate themselves when they are no longer necessary, it may occasionally be necessary to forcibly remove a filter agent, e.g. if that agent did not terminate properly. In order to have such a filter agent marked for garbage collection, not only must the agent's registered names be freed as described in Section 3.2.3, but the filter agent must also be unregistered from the Oasis proxy dispatcher. As in the case in which an agent's registered names must be forcible unregistered, a proxy administrator, or his agent with appropriate permissions, can unregister an agent from the dispatcher when necessary.

3.6 An Example Agent

This section presents an example of a user agent filter that recognizes ftp URLs in documents, and intercepts any attempt to transfer the data they reference. This functionality is useful to users searching for information using a weakly connected browser that is not on their usual desktop. For example, a user might be logged in from home over a slow modem link, searching for references to be examined the following day in the office. Typically, he would not wish to retrieve the large amount of data over the slow link to his home, but would rather instruct an agent, executing on a machine in his office with a fast link to the network, to intercept and save such data on his behalf.

Like all filters, this filter is implemented as a pair of agents. The administrator part of this agent is implemented by class `FtpRedirect` and the implementation part of the agent is implemented by class `FtpRedirectUC`. The skeletons of their implementations are presented below.

3.6.1 Class `FtpRedirect`

`FtpRedirect`, when loaded, registers with the dispatcher indicating an interest in all URL data fetches occurring via the Oasis proxy. This interest is registered at the priority accorded all filter agents by default. It then examines data fetch requests as they occur, and interprets a fetch of "`http://unload_redirect/'`" to mean that the ftp redirection filter is to terminate. It also interprets fetches of URLs beginning with the redirection prefix "`ftp://redirected_`" as URLs whose data is to be fetched and stored into a local file. Intercepted URL data is stored in the `/tmp` directory of the machine hosting the filter, in a file whose name is the same as the final component of the name of the URL. After an interception occurs, the oasis utility class `AgentMessageUC` is used to construct a "fake" URL data connection to satisfy the original request – the connection simply produces a message indicating that the data of the URL has been intercepted and saved. Finally, all other URLs (i.e. non-special URLs) are encapulated with the filter implementation class, `FtpRedirectUC` and allowed to proceed as usual.

The code for `FtpRedirect` is as follows:

```
import java.io.*;
import java.net.URLConnection;
import java.net.URL;

import oasis.agent.*;

public class FtpRedirect extends FilterAgent {
    public FtpRedirect()    {};
    public final String redirect_prefix = "ftp://redirected_";

    public void main (OutputStream os) {
        // After loading, just register with the dispatcher and
```

```
        // print a message to inform the loader of successful initialization.

        PrintStream p = (PrintStream) os;

        registerWithDispatcher("*");

        try {
            sendHttpTextHeader(p);
            p.println("Ftp redirect filter loaded");
            p.flush();
        } catch (Error e) {
            e.printStackTrace();
        }
    }

    public void copyToTarget(URLConnection uc, String target)
        throws IOException
        // Read data from URL connection and copy to target file
    {
        ...
    }

    public URLConnection openConnection (URLConnection uc)
    {
        // This code is called once for every request processed by the
        // proxy, after the agents registers interest with the
        // dispatcher.

        FtpUC ftp_uc = null;
        String url, new_url;
        URL new_URL = null;
        URLConnection new_uc = null;
        String ftp_targetdir = "/tmp";
        String unreg_msg = "FTP URL Redirect mgr now unregistered.\r\n\r\n";


        // Check for a special URL that indicates termination
        url = uc.getURL().toExternalForm();

        if (url.equals("http://unload_redirect/")) {
            unregister();
            return new AgentMessageUC(unreg_msg);
        }
```

```
// If the URL doesn't need to be redirected, wrap it with an
// FtpRedirectUC (the agent implementation) and proceed

if (!url.startsWith(redirect_prefix)) {
    try {
        new_uc = new FtpRedirectUC(uc);
    } catch (java.net.MalformedURLException e) {
        System.err.println ("Exception creating new_uc.");
    }
    return new_uc;
}

// URL is a redirected URL

// Construct the original URL by removing the redirection prefix

new_url = "ftp://" + url.substring(redirect_prefix.length(),
                                   url.length());

System.err.println ("redirected URL is " + new_url);

// Open a data connection for the original URL

try {
    new_URL = new URL (new_url);
    new_uc =  dispatchNewURL(new_URL);
} catch (Exception e) {
}

// If we got an html file back, it's probably the description
// of a directory – don't intercept it.

String ct = new_uc.getHeaderField("Content-Type");
if (ct != null && ct.equals("text/html"))
  return new_uc;


// File is to be intercepted. Figure out where to store the
// intercepted data.

String URL_file = new_URL.getFile();
```

```
        String ftp_target_file;
        int i;

        if ((i = URL_file.lastIndexOf('/')) < 0) {
            ftp_target_file = ftp_targetdir + URL_file;
        } else {
            ftp_target_file = ftp_targetdir +
                URL_file.substring(i, URL_file.length());
        }


        // Finally, fetch the data and save it to the file in question
        try {
            copyToTarget(new_uc, ftp_target_file);
        } catch (IOException e) {
            e.printStackTrace(System.err);
        }

        // Satisfy the pending data request by returning a "fake"
        // URL connection that will notify the original requestor that
        // the data has been intercepted.

        return new AgentMessageUC ("URL " + new_url +
                                   " saved to " + ftp_target_file + ".\n");
    }
}
```

3.6.2 Class `FtpRedirectUC`

The implementation part of the filter examines html files being fetched, modifying their data so that all `ftp://` URLs are prefixed with `ftp://redirected_`, allowing the administrator part of the filter to recognise and intercept any subsequent fetches. The implementation in this case is fairly trivial, as it makes use of the Oasis utility class `FindInputStream`, which allows a filter to specify a search string and its replacement, to be applied to data streaming through a URL data connection.

```
import java.net.URLConnection;
import java.io.InputStream;
```

```
import java.io.StringBufferInputStream;

import oasis.agent.*;

public class FtpRedirectUC extends FilterAgentUC {

    public FtpRedirectUC(URLConnection underlying)
        throws java.net.MalformedURLException
    {
        // upon initialization, make sure the constructor for the encapsulated
        // stream gets called

        super(underlying);
    }

    public InputStream getInputStream() throws java.io.IOException
        // This method is called when it's time to read the data from
        // URL. If appropriate, set up a search/replace pair to modify
        // the data as it passes through.
    {
        String type;
        InputStream is;


        is = uc.getInputStream();
        type = getHeaderField("Content-type");

        // We only want to search for and modify document data if the
        // document is an HTML document.

        if (type != null && type.equals("text/html"))
        {
            return (InputStream)
                new FindInputStream(is, "ftp://", "ftp://redirected_");
        } else {
            return is;
        }
    }
}
```

3.7 Implementation Summary and Limitations

Oasis provides a flexible, extensible architecture for the construction of agent-based online services within the context of the existing World Wide Web. The Oasis infrastructure was intended to provide a platform for experimentation with the two main agent models defined in this thesis, but is flexible enough to easily support other agent models (as exemplified earlier in this chapter) as well as to be adapted to many agent-based uses other than control-oriented application deployment.

The first generation Oasis implementation supports the full Oasis architecture, subject to a small number of simplifying implementation choices. These choices were irrelevant to the successful validation of the architecture as a platform for efficient implementation of control-based applications. However, as they may affect the flexibility of Oasis in a production environment, these choices are briefly listed below:

- Security Model – The Oasis architecture provides for a flexible security model based on varying levels of trust, as expressed by access control lists, capabilities, or via a security manager agent. The current implementation implements only two levels of trust (i.e., trusted and untrusted agents, differentiated based on point of origin), and provides the hooks necessary for the implementation of a more complex security model.

- Name Service – The current Oasis implementation includes only a simple local name service that implements a flat name space. This has been adequate for all of the experimental validation described. However, in practice, it is expected that the native Oasis name service will serve only to bootstrap a more comprehensive, possibly distributed, name service implemented by a trusted agent. The current implementation provides the facilities needed for the addition of such a name service without modifications to the base system.

- Garbage Collection – The Oasis implementation tracks the usage of threads and windows by Oasis agents, and reclaims them after an agent's main thread exits. The Java runtime automatically reclaims memory space used by agent code, once that code is no longer referenced, and Oasis provides primitives to allow an administrator agent to forcibly remove Oasis' references to agents, marking them for garbage collection. Oasis does not currently track the use of object store space, nor does it implement automated garbage collection of agent references bound into the simple native namespace. Both of these functions can be provided by agents implementing more sophisticated replacements of the object store and name space provided in the first Oasis implementation.

- Agent Loading – Oasis does not currently support the loading of agents in Java Archive format [48]. Addition of this support would require minor modifications to the Oasis proxy itself, and result in slightly more efficient agent loading, while allowing for the addition of digital signatures to agents. These are features would make noteworthy additions to a production deployment of Oasis, but were irrelevant to the validation of the thesis.
- Filter Agent Support – The Oasis architecture allows for the assignment of arbitrary priorities to filter agents, affecting the order in which accesses are encapsulated when multiple filters indicate interest in a URL. The current implementation only allows the use of two distinct priorities – one for most filters, and one for filters such as caching filters that require early access to requests, possibly in order to prevent external data fetching.

The next two chapters demonstrate that the Oasis implementation is efficiently able to support all of the basic requirements for deployment of control-oriented services, without imposing undue overhead on traditional Web access.

Chapter 4

Evaluation

The Oasis prototype has been evaluated in the context of representative example applications. These applications have been chosen to demonstrate both the qualitative and quantitative aspects of control-oriented application design, as deployed in the context of the World Wide Web using Oasis. This chapter presents the goals of the evaluation methodology, the rationale for the selection of the evaluation applications, and the detailed structure of the evaluation applications themselves.

4.1 Goals

The applications chosen for the evaluation of the Oasis prototype were selected with the following qualitative goals in mind:

- The applications should produce scenarios in which user and server agents exist alone, as well as scenarios in which they exist and interact concurrently.
- The applications should, where possible, provide experience with control-oriented models deployed in the context of real Web-based services. To this end, two of the applications selected have been chosen to augment and improve existing Web-based services.
- The applications should demonstrate the applicability of control-based techniques to services based on a range of data types.

- The applications should provide experience with Oasis' support of all of the basic control-oriented application structures, namely proxy objects, composable services, and controllable resource managers, as described in Section 2.2. The applications chosen also demonstrate derivative application structures, of which the most notable is the support of filtering.

The applications selected also demonstrate the quantitative advantages of control-oriented design by quantifying the range of reduction in communication that can be obtained. This reduction has been measured and is characterized both in terms of a reduction in the total bandwidth required, as well as in terms of the elimination of high-latency interactions over a communication-constrained link.

4.2 Rationale

Oasis was designed to support the deployment of control-oriented applications in the context of the World Wide Web, which has been evolving over time. Originally, the Web was intended solely to support document retrieval (i.e. communication), and this is reflected in the majority of Web applications and in the infrastructure and protocols supporting them. The subsequent advent of Java support in Web browsers has permitted a limited degree of control expressed as "applets" (server agents) executing within the browser. Oasis extends current Web infrastructure and completes the framework necessary for the support of arbitrary control-oriented applications. This progression is reflected, using the symbology of Figure 2.2, in Figure 4.1.

The progression of applications used to evaluate the Oasis infrastructure reflects this evolutionary path. The first part of the evaluation characterizes both the positive and negative performance implications of Oasis as a control-oriented extension of the current infrastructure. The second part of the evaluation measures the benefits obtainable by identifying and exploiting control inherent in older applications that are constrained, by the current Web infrastructure, to being communication-oriented rather than control-oriented. The last part of the evaluation then proceeds to evaluate the benefits available to applications designed specifically to exploit the Oasis infrastructure.

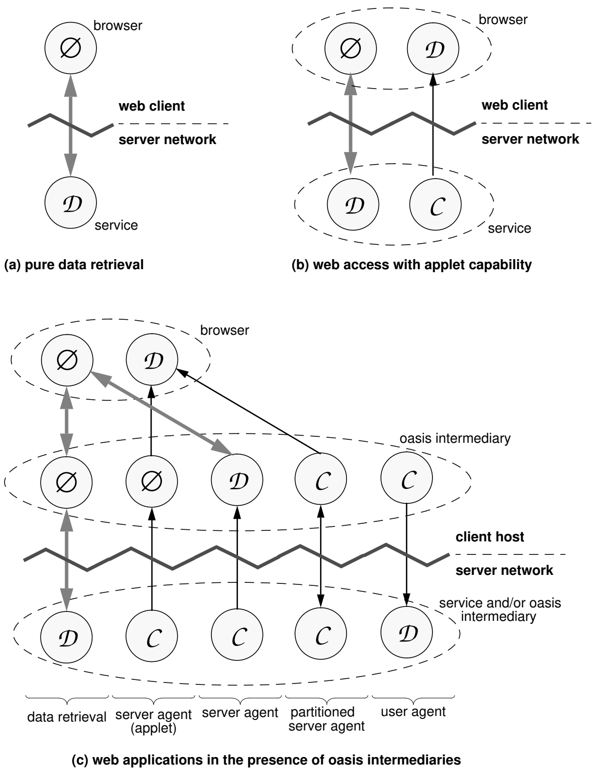


Figure 4.1: The evolution of the World Wide Web: (a) Initially, only pure communication was possible. (b) The introduction of Java-capable browsers allowed the construction of limited server agents, whereas the use of a framework like Oasis in (c) allows the support of arbitrary application structures. Partitioned user agents are also possible in the Oasis environment, but are not depicted here as they require the presence of more than two Oasis proxies. Note that these diagrams are rotated 90 degrees with respect to the orientation used for similar diagrams later in this chapter. In later diagrams, the boundary between the user and the server network is vertical.

4.2.1 The Oasis Communication Benchmark

The first Oasis evaluation "application" is a group of synthetic benchmark agents that collectively measure the costs of communication through an Oasis proxy, and characterize some of the benefits that applications may obtain by using Oasis. Because Oasis is implemented as a network of special HTTP proxies, the communications performance of those proxies affects all applications using them; the very existence of the proxy mandates some overhead to any client communications. Similarly, the overhead of the Oasis mechanism for agent composition is a critical factor in determining the finest granularity at which applications can be split into multiple composed agents without incurring excessive performance penalties. These two factors are the costs of application deployment using the Oasis infrastructure.

On the other hand, every eliminated round trip to the server network and every reduction in communication made possible through the presence of the Oasis infrastructure produce performance benefits to the application. The first experiment characterises the maximum cost and minimum benefit per operation, as constrained by the infrastructure, that a client will obtain.

4.2.2 The Controllable Cache Manager

The second evaluation application is a controllable cache manager, implemented as an Oasis filter agent. This application is a nontrivial composable filter that implements a controllable resource manager. The controllable cache provides a means for servers to control the caching of data they supply. In the absence of server guidance, a default guidance scheme, similar to that implemented by other common HTTP proxies, is invoked. This controllable cache provides both a composable building block that can be deployed in any Oasis proxy to enable caching, and also provides a control interface that can be used by any service to obtain some of the benefits of server-influenced cache management at the proxy; these benefits have been leveraged in the construction of the third evaluation application. Additionally, when augmented with a suitable cache management agent, the controllable cache provides a platform suitable for transparent, mechanical identification and exploitation of control characteristics inherent in some ordinary Web traffic. The second part of the evaluation presents the application of this technique to normal Web traffic flowing through an Oasis proxy, and demonstrates the gains that may be obtained by some applications when they are analyzed and adapted to exploit a control-aware infrastructure. The results obtained represent a lower bound on the improvements obtainable because they permit further optimization, as discussed in Chapter 5.

Conceptually, the second experiment involves transparently inserting a control-relationship, implemented by a pair of controllable caches, in the communication path of scenario (a) in Figure 4.1, resulting in Figure 4.2. The controllable caches exist on either side of a constrained communication link, and a control relationship between them (specifically, diff-based data update) is exploited to reduce traffic whenever possible. This experiment measures the cost of performing control-based communication in lieu of simple data transfer when the former is an option.

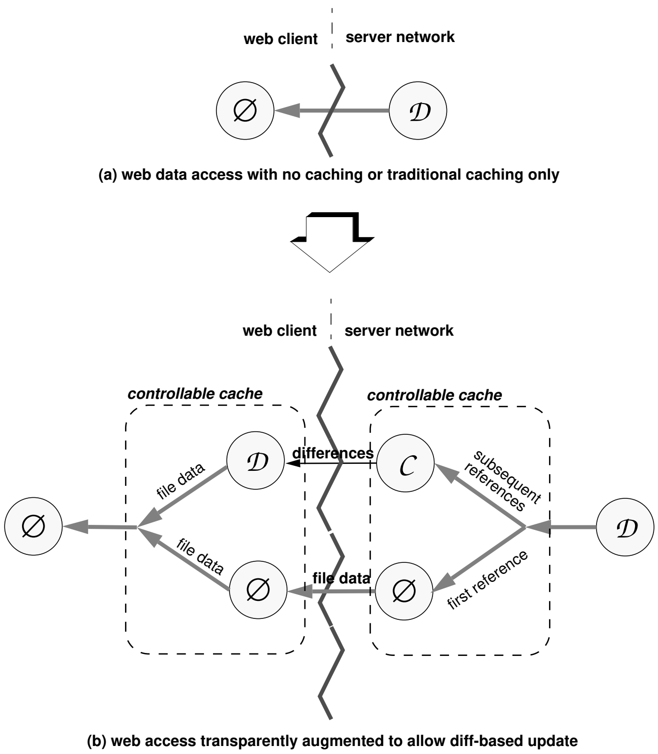


Figure 4.2: The controllable cache experiment involves encapsulating the constrained communication link with a pair of controllable caches. Note that (a) corresponds to Figure 4.1(a). In scenario (b), the cache on the user side can manipulate cached data under control of the cache on the server network. In the experiment, this capability is used to implement diff-based update for certain data types.

4.2.3 The Weather Browser

The third experiment, in contrast to the second, is based on an application that has been constructed from several components designed to control one another. The architecture of the third application is presented in Figure 4.3. The components of the application can be partitioned along multiple boundaries; the experiment focusses on comparing the costs of partitioning along a communication-oriented boundary versus that of partitioning along control-oriented boundaries.

The third evaluation application is a weather data presentation system. Its implementation exemplifies the use of proxy objects and composability, and is described later in this chapter. The measurements of this system serve to demonstrate the benefits that can be obtained when a system

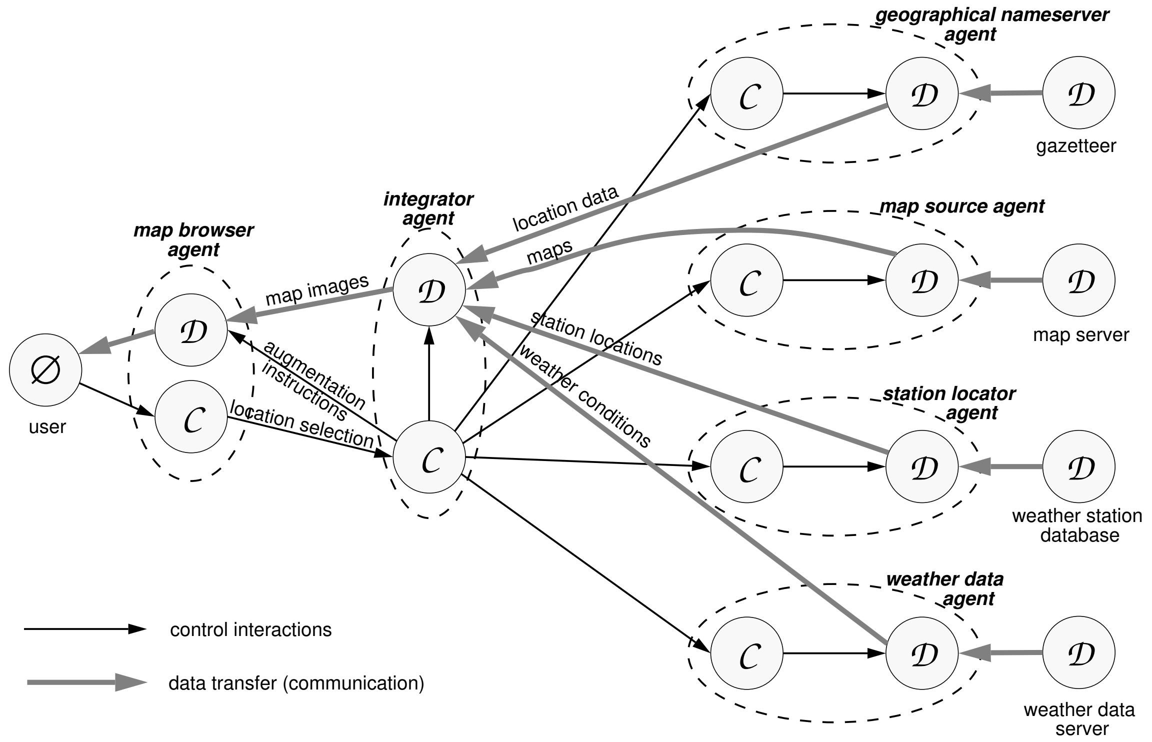


Figure 4.3: Structure of the weather browser evaluation application. This application presents many possibilities for partitioning corresponding to different types of interaction among parts. This application and its associated partitionings are discussed in detail in Section 4.5.

is designed specifically to exploit control-enabling infrastructure where available.

The next three sections describe the evaluation applications and their roles in the experiments performed.

4.3 Experiment 1

4.3.1 Architecture of the Communication Benchmark

The communication benchmark is composed of a data referencing agent that measures the communication performance of data fetches in the environment being tested, and a group of agents created to measure the cost of agent composition on a particular Oasis proxy.

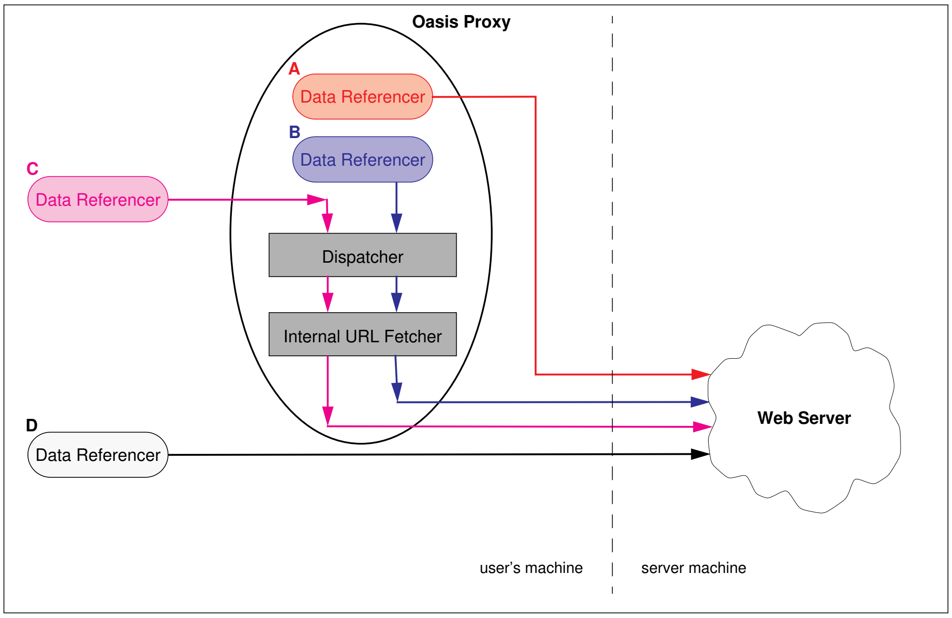


Figure 4.4: The various test configurations used to evaluate Oasis communication performance. In configurations A and B, the data referencer executes as an agent within an Oasis proxy. In configuration A, network I/O is performed directly via the Java network primitives. In configuration B, I/O is dispatched through the proxy. In configuration C, the data referencer runs as a standalone Java program, dispatching its I/O requests through the proxy. Configuration D, in which the data referencer runs as a standalone Java program using Java network I/O primitives directly, is analogous to configuration A, and has not been measured separately. Each test configuration was used to produce specific test scenarios, described in the next chapter. Configuration A was used to measure the *standalone java* scenario. Configuration B was used to measure the *cache miss* and *cache hit (agent)* scenarios. Configuration C was used to measure the *cache hit (external prog)* scenario.

The data referencing agent is an Oasis agent that can also be executed as a standalone Java application. This agent measures the time needed to access data files of varying size from a Web server. Each file is fetched multiple times, and the transfer rate obtained is calculated. When executing as an agent within an Oasis proxy, this agent uses the hosting proxy to service its data requests, or, if permitted by the hosting proxy, can be configured to fetch data directly using Java network I/O. In the former configuration, the agent incurs the overheads of dispatching through the proxy, and reaps the benefits thereof, such as access to locally cached data when the hosting proxy has enabled caching. It thus measures communication performance as available to Oasis

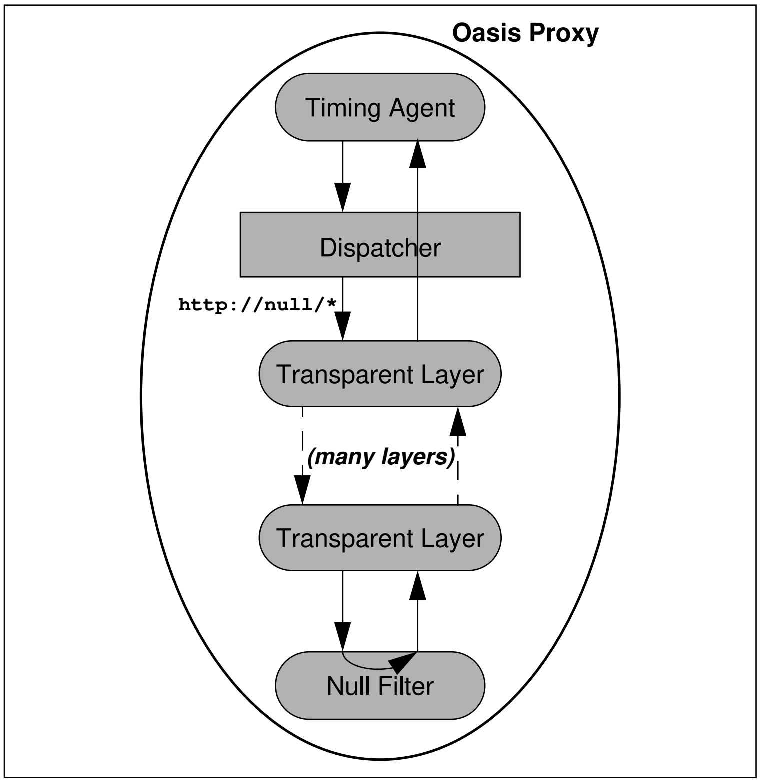


Figure 4.5: The relationship between the timer agent, the transparent layer filters, and the null filter. The timer agent issues requests to the `http://null/` address space. These requests are dispatched through the transparent layer filters, and ultimately handled by the null filter. Composition overhead is measured by measuring the total round trip time taken to complete the request.

agents. In the latter configuration, the data referencer measures the performance of Java network I/O primitives when used from an agent. In this configuration, the data referencer encounters no overhead and derives almost no benefit from the hosting Oasis infrastructure; a small benefit results because the hosting Oasis runtime has preinitialized some Java data structures and warmed caches that are shared by the loaded agent. Lastly, when executed as a truly standalone application, the data referencer is configured to use an Oasis proxy to service its requests. It thereby measures the communication performance available to external client applications using the proxy for data retrieval. Although the standalone data referencer can also be configured not to use the proxy at all, the resultant configuration is equivalent to the second scenario already presented, i.e. the measurement of Java network I/O primitives, uninfluenced by the existence of the proxy. Figure 4.4 illustrates the differences between the test configurations measured.

The measurement of composition overhead is accomplished using three cooperating Oasis agents: a null filter, a transparent layer filter, and a timing agent. The relationship of these three agents is depicted in Figure 4.5. The null filter is a filter agent that registers with an Oasis proxy's dispatcher to filter accesses under the `http://null/` address space. Any requests it receives are consumed, and a zero-length data object is returned in response[1]. Any fetches to part of the null filter space thus amount to the cheapest possible object fetches. The transparent layer filter is a filter agent that overlays the null address space, passing requests through to the layer below it, and returning any responses received. Thus, the addition of a null layer filter is the addition of the cheapest possible agent composition. Multiple null layer filters can be installed in a proxy, and requests to the null filter space pass through them all before reaching the null filter. The timing agent makes several requests to the null address space, and computes the cost per request. As transparent filter layers are added to a proxy, the measured incremental increase in the cost of a fetch to the null address space yields the cost of agent composition under Oasis.

4.3.2 Characterization of Costs and Benefits

As mentioned above, any application deployed in a control-oriented infrastructure is subject to certain costs and benefits. The benefits of control-orientedness arise from the change in the nature of the communication performed, whereas the costs typically arise from the control-oriented infrastructure itself. The maximum costs of a control-oriented infrastructure are easily characterized in terms of overhead in the unchanged communications operations they affect. The minimum benefits realized, however, require greater care, because they arise from changes in the nature of communication of an application, and those changes are usually application-specific. For example, a control-oriented system may be able to completely eliminate requests to the server network that are made by a communication-oriented equivalent, resulting in an "infinite" benefit, if the two requests themselves are compared. Needless to say, the control-oriented implementation is unlikely to see an infinite speedup. In order to avoid this difficulty, measurements of minimal benefit are presented per change in communication operation, e.g. by specifying that every removal of a roundtrip to the server network in the test environment corresponds to an improvement of *at least* 400 ms, because a minimal-cost roundtrip to the server network requires 400ms to process. Of course, in almost all real applications, the realized benefit is greater due to the application specific and possibly highly variable amount of time required by the remote server to process and respond to the request, and, in some cases, also due to significant network latency in communicating with distant servers. (See Experiment 3 for an example in which eliminated roundtrips have a

[1]Note that fetching a zero-length data object requires the issuance of a request of approximately 80 bytes, and the return of an object header of approximately 100 bytes, and is therefore not a null request.

cost that is much greater than that of a minimal-cost interaction.) Similarly, savings in bandwidth are presented by characterizing the communication bandwidth of various test environments, as the amount of communication eliminated will depend on the specific application.

Note that this experiment differs from the next two evaluation experiments in that costs and benefits are characterized by *timed* performance in a particular test environment. The rest of the evaluation measures communications *usage* reduction in a particular application, and applies to any environment in which the application may be deployed. However, the results of this experiment, presented in the next chapter, show that the constrained communications link is a bottleneck in that both the server and the client are capable of saturating it. Consequently, the minimal benefits and maximal costs measured establish bounds on any similarly communication-constrained environment, independent of improvements in processors used at the server and client.

4.3.3 Experimental Setup

In order to evaluate the communication overhead of using the Oasis infrastructure, as well as to determine the impact per operation of eliminated roundtrips and reduced communication, the latency and bandwidth of information retrieval was measured under a number of different scenarios. The scenarios measured were:

- *modem, standalone java*, wherein communication occurred over a 28.8 Kbps modem line and the data referencing agent directly accessed Java network primitives.
- *modem, cache miss*, wherein communication occurred over a 28.8 Kbps modem line and the data referencing agent dispatched its communication requests through the proxy. The proxy was configured not to cache the results of data accesses.
- *adsl, standalone java*, wherein communication occurred over an Asymmetric Digital Subscriber Line (ADSL) connection (1.5 Mbps downstream, 64 Kbps upstream) and the data referencing agent accessed Java network primitives directly.
- *adsl, standalone C*, wherein communication occurred over an ADSL line, using a standalone C equivalent of the data referencing agent. This scenario was measured to ensure that java performance constraints were not limiting the speed of communication over the ADSL line.
- *adsl, cache miss*, wherein communication occured over an ADSL line and the data referencing agent dispatched its communication requests through the proxy. The proxy was

configured not to cache the results of data accesses.

- *cache hit (agent)*, wherein the data referencing agent dispatched all its requests through the hosting Oasis proxy, and the requests were all satisfied from the proxy cache. The proxy cache was provided by the caching filter layer descibed in Section 4.4.1.
- *cache hit (external prog)*, wherein the data referencing agent ran as a standalone java program dispatching requests through the Oasis proxy, and those requests were satisfied from cache hits.

In all cases, the server hosting the accessed test data was a machine capable of saturating the communications link and was not concurrently engaged for any other purposes. The client was an average, slower personal computer concurrently executing other desktop applications, including a Web browser. Specifics of the test environment are presented in the next chapter.

The slower personal computer was also used to measure the cost of Oasis object composition by repeated fetched to null data, as described earlier.

In brief, the results demonstrate that the use of Oasis adds negligible overhead to communication over modem and ADSL links and that, even on a relatively slow personal computer running java, agent composition overhead is low. The use of Oasis as a communications intermediary produced a reduction of no more than 0.04% in peak bandwidth, and an increase in access latency of between 0 and 2.5 ms. The overhead incurred per agent composition is 1.2 milliseconds per composed layer, or no more than 2% of the fastest possible agent access in the environment tested.

On the other hand, remote accesses that can be converted to cached accesses as a result of control-oriented design experience access latency reduced by a factor of between 3.75 (ADSL) and 16.7 (modem), and retrieval bandwidth increased by a factor of between 71.5 (ADSL) and 3915 (modem). Each access that is eliminated completely produces a reduction in latency of at least 90 ms (ADSL) or 400 ms (modem), and a potentially large reduction in retrieval time dependent upon the size of the eliminated access. The potential for these benefits, which can be obtained through the use of control-oriented design in program structuring, vastly outweighs the minor costs of using Oasis. The measurements and analysis of this experiment are presented in detail in the next chapter.

4.4 Experiment 2

4.4.1 Architecture of the Controllable Cache Manager

The controllable cache manager is a composable filter agent that can be loaded into any Oasis proxy to provide caching of Web accesses made through that proxy. The cache manager utilizes the persistent object store facility provided by the Oasis proxy to implement an on-disk cache of data retrieved. As one might expect, URL fetch requests received by the proxy are checked against the cache in the object store. If unexpired data from an earlier access is present in the cache, it is returned to the requestor, avoiding a data retrieval over the network. If a cache miss occurs, the retrieved data is examined. If the data is deemed cachable, the data is stored in the cache as it is fed back to the requestor.

Traditionally, Web caching has been directed at caching static documents, as might be expected of a system originally targetted at document retrieval. Caching Web proxies usually cache data only if the server provides an explicit expiration date via an `Expires` object header, or if the server indicates the time at which a document was last modified, via a `Last-Modified` document header. Unfortunately, Web servers usually provide an expiration header only when manually configured to do so by the administrator of the server; this is fairly uncommon at present. However, for data served from files, a last modification date header is often added automatically by the Web server. Many caching proxies heuristically cache each document for some fraction of its age at the time of retrieval, using the time of last modification to determine the age. For documents, this approach works well and requires minimal effort.

The controllable cache manager differs from the cache in a traditional Web proxy in that retrieved documents are allowed to specify a *cache validator* – a server agent which provides assistance in making caching decisions about the cachability and validity of that document. Documents are allowed to specify, via headers, the name of a preferred cache validator, and a URL from which the validator may be obtained if it is unknown to the proxy. In the event that a cache validator is not specified, a default validator embodying the caching policy discussed in the previous paragraph is invoked.

The cache validator of a document is explicitly invoked when the cache manager:

- needs to determine whether a cached copy of the document has expired,
- is about to delete the object from the object store,

- needs to determine the cachability of an item being retrieved,
- has downloaded the headers of a document being retrieved, or
- has downloaded the body of a document being retrieved.

When invoked at the last two entry points, the cache validator is permitted to modify the data cached for the access being handled. This allows for the implementation of custom data update protocols via a cache validator. In addition, a cache validator, if implemented as a filter agent, may request notification of all accesses to specific URLs. Collectively, these entry points allow a considerable degree of flexibility in the implementation of caching and update schemes implemented for any downloaded object. For example, much of the optimal configuration of the weather browser application used in the third experiment could easily have been implemented as a cache validator operating on the underlying weather data. Note that this approach was not chosen in practice to allow for easier reconfiguration of the application into the non-control-oriented partitions used for evaluation by comparison.

4.4.2 Static and Dynamic Data

One reason that online services may benefit from control-oriented application structures is that a service's client can often be configured to retain memory of interactions with the server, and to use the retained information, under guidance from the server, to reduce subsequent data transfer. For online services, which often provide data that is periodically updated, this approach is superior to simple caching. For example, it permits the separation, and subsequent recombination at the client, of the static and dynamic components of the data served – later accesses may require refreshing of any changed dynamic components. The weather browser, used in the third experiment, employs this approach, separating a weather map into the static underlying geography, and dynamically changing weather data. The cache validators of the controllable cache manager are well suited to exploiting such a separation and recombination when aided by the server in making the separation. However, for certain data types *automatic* separation is also possible without server assistance. For example, in the case of text documents a usable, though not necessarily optimal, separation of the dynamic components can be obtained by computing the text diff between the new document and the old one.

By encapsulating a constrained communication link with a pair of controllable caches, as outlined earlier in Figure 4.2 and presented in detail in Figure 4.6, automatically separatable data can be exploited in a control-oriented manner. On the first access by a client, both caches are loaded

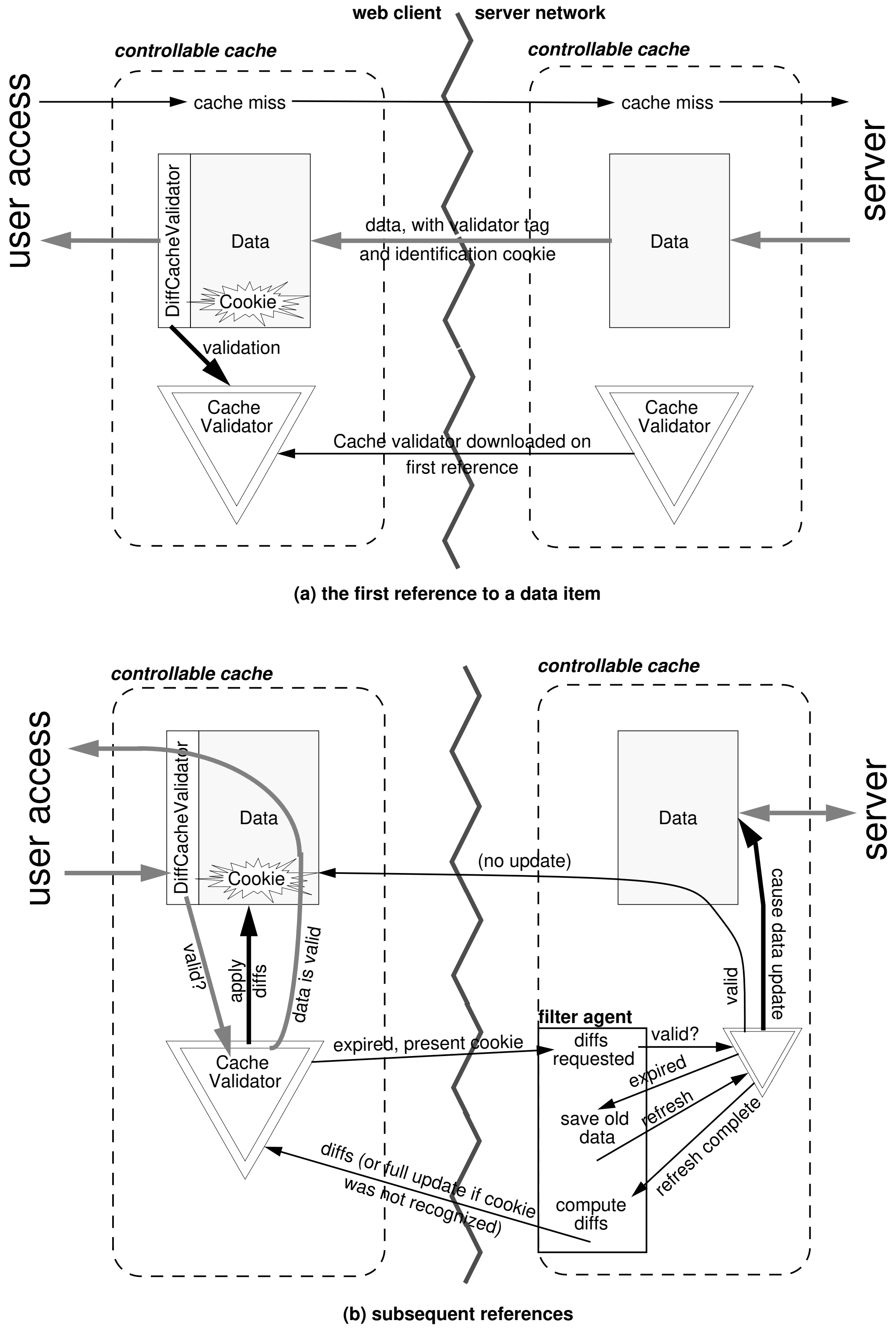


Figure 4.6: Operation of the controllable cache in implementing diff-based cache update.

with the data referenced; this is a pure communication operation. However, the server-side cache adds a token to the outbound data and marks the data such that it will be refreshed, in the client side cache, via a custom validator. The server-side cache supplies the custom validator on request from the client. On a subsequent cache miss at the client, the validator is invoked, and presents the server-side cache with the token from the original, expired document. The server proceeds to obtain a current copy of the document requested. Once the new document has been obtained the server computes the differences between the updated copy and the original, assuming it retains a copy of the original associated with the client's token. If the differences are small, it may then choose to send the client only the differences rather than a fresh copy of the original document. These differences constitute an approximate determination of the truly dynamic components of the document.

4.4.3 Experimental Setup

The dual-cache scenario just described represents a means to automatically convert the communication between some Web clients and servers into control-based interactions – as such, this approach has been studied to compare the communication requirements of a control-based application to that of the conventional, deployed, communication-oriented equivalent.

An Oasis cache validator supporting the the following types of update was implemented: diff-updates of textual data (using the diff format of "diff -f" under Unix), recognition of unchanged nontextual data being refreshed after expiration (i.e. a null diff), and full document update of changed nontextual data. The applicability and effectiveness of this scheme was measured by examining data collected from a number of Web proxy caches containing all of their users' cachable Web accesses for a period of two months. All of the caches collected data using a controllable cache layer and default validator as described earlier. The caches were modified slightly to record the occurrence of refreshes, and to save the old and new values of the most recent refresh of every document. As described in detail in the next chapter, the control-oriented approach was applicable to 89% of the refresh traffic, reducing that segment by about 90%, for an overall elimination of about 80% of the refresh traffic. (i.e., Refreshes were reduced in size by a factor of 5.)

This experiment examines a control-based approach to Web access involving cachable refreshes in comparison to the conventional alternative. Note that it does *not* attempt to measure benefits that could be obtained by specifically tailoring caches or services to the control-based architecture. That is the focus of the third experiment. In particular, this experiment does not measure the total reduction in traffic that could be obtained for all normal Web access, mainly because the normal definition of cachability (see Section 4.4.1) usually excludes data from those services that are most likely to provide frequent data updates, and are therefore likely to benefit most from control-based update. It is also in part due to service providers' efforts to thwart normal caching schemes (to maximize the access count on Web sites, for the benefit of advertisers), and in part because only the most recent refresh of each data item is recorded. The reduction in communication that can be realized with application assistance is addressed by the third experiment.

4.5 Experiment 3

4.5.1 Architecture of the Weather Browser

Weather data is presently available via the internet from a multitude of services. Most of these services provide weather maps that are periodically updated and must be downloaded in their entirety by the user whenever he wishes to ensure that his information is current. In the presence of constrained communication, this approach is wasteful of bandwidth, and is hampered by high latencies in interactions with the user. Users accessing such a service may resort instead to accessing individual components of the weather maps, such as textual presentations of the pressure or temperature data presented by location. In doing so, they are fetching only the dynamic component of the data, i.e. the weather data, while relying on memory to provide the static geographical context provided by a map. However, existing internet services provide access to both the static geographical context underlying weather maps as well as to textual versions of the dynamic weather data. The weather browser demonstrates the construction of an Oasis application, built by composing these existing services, that is able to provide undegraded service in the presence of constrained communication, because it incurs no more communication overhead in the steady state than that minimally required to fetch the dynamic weather data itself.

The weather browser thus presents a unified interface to several sources of meteorological and geographical data available via the internet. It is modelled after other weather browsers such as the Weather Anchordesk[7], but has been implemented to take advantage of the Oasis framework, by exposing the control and communication relationships between major application components. As such, it illustrates the advantages and flexibility that can be realized by applications that are designed to exploit an infrastructure that supports control. The weather browser employs proxy objects, service composability, and filter agents, while leveraging the availability of composable filters and controllable resource management as embodied in the controllable cache manager.

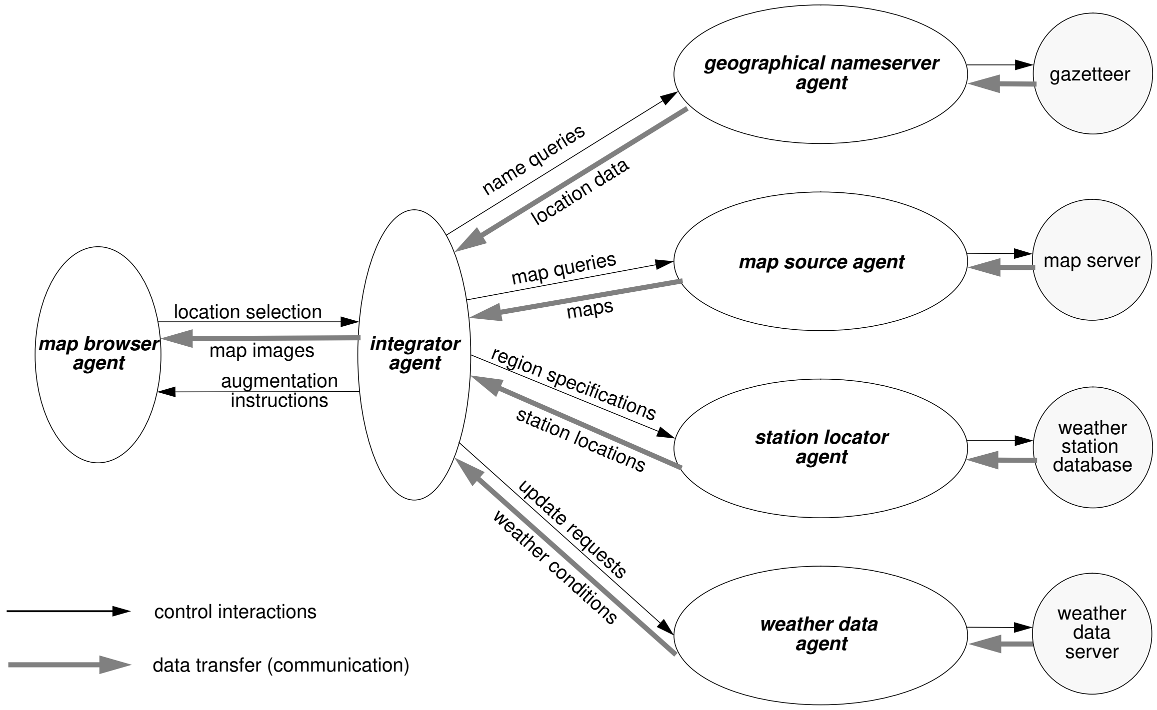


Figure 4.7: The components of the weather browser application. This is a higher level view of Figure 4.3.

Figure 4.7 presents a high-level view of the architecture of the weather browser. As can be seen, the weather browser is implemented as a set of interacting Oasis agents. The browser agent interacts with the user, allowing selection of a geographical region for which weather data is desired. This selection information is conveyed to the integrator agent, which queries the available data sources, merges their results, and presents them to the browser agent for display to the user. Each of the data sources is encapsulated using a data agent, which serves to interface each traditional Web weather service to the Oasis environment, and to the environment of this application, in particular.

4.5.2 Application Structure and Partitioning

The weather browser application must manage a number of simple and composite data types, namely geographical maps, geographic names, weather station location data, temperature data, weather maps, etc. Further, all of these data items can be composed with each other by correlating them using the underlying geographical context, i.e. the latitude(s) and longitudes(s) at which the data applies. This structure suggests a natural control-oriented partitioning of the application: one agent is allocated to manipulate each of the major data types, and allows this manipulation to be performed under control of other agents; agents exploit geographical context to reference each other's data. Thus, the map browser displays and augments geographical maps while the data agents for geographic names, weather station location data and weather data manage access to their respective data sources, and the integrator controls the interactions amongst the agents, embodying the knowledge to create a weather map. All of the agents implement control interfaces appropriate to a provider and/or to a consumer of geographical data, and these interfaces are the primary mechanism for control interaction.

In this scenario, the boundaries between these agents represent coarse-grained control-oriented lines along which the application can be partitioned. Any such partitioning can be realized at runtime, and depends simply on the location, relative to the constrained link, at which each agent is started. However, further partitioning options also lie within the agents delineated above.

Separation of control and data processing may be possible even within an agent manipulating just one data type. Of the weather browser agents, many of the data agents can be further split into a control processing part and a data manipulation part. For example, consider the geographical name server agent which is a server agent encapsulating access to a geographical name service on the server network. Its function is to accept a query string from the map browser (or from anyone else), and to return the location(s), represented by latitude and longitude, that may match the name query. Because the name server agent encapsulates access to an ordinary Web server, it must perform a simple data processing function in addition to controlling that server: it has to filter out extraneous information and HTML markup from the server's output. As the percentage of extraneous output may be as high as 90-95% of the total for geographical names that generate many matches, it is desirable to allow the data processing to occur on the server side of the network, reducing the size of the results communicated over the constrained link. By separating the control processing and the data manipulation, the name server agent can be split into a proxy object operating on the client side of the constrained link, controlling a data manipulation agent on the server network. The proxy object represents the name server in the user's environment, accepts requests on behalf of the user's map browser, instructs the data manipulation "sub-agent" on the server network to perform the appropriate queries and to process the results, and eventually returns a response to the map browser. Other data agents employed in the weather browser application also benefit from a similar separation of control and data under certain conditions.

With a small amount of application programmer assistance in structuring each agent, the separation of control and data processing within the components of the agent is easily realized within the Oasis infrastructure. To this end, all of the data agents described have been implemented explicitly as the combination of a proxy object and a filter agent. (This implementation is reflected in Figure 4.3.) The control processing of the agent lies within the proxy object, whereas the data manipulation lies within the filter agent. The proxy object and the filter agent interact using a URL-based interface, allowing the two to transparently reside, if necessary, in different Oasis proxies, as Oasis will forward the HTTP accesses from one proxy to the next. When such a data agent is first loaded, the proxy object initializes and checks for the existence of its companion filter agent. This check is performed simply by attempting to access the URL-based control interface of the filter– if the access fails, the proxy object concludes that no companion filter exists further upstream (i.e. closer to the server network), and initializes one on the Oasis proxy where it is running. Thus, a data agent may be run purely as a server agent when no upstream filter has been started or as a user agent (as the user-side proxy object controls the filter) when partitioned across multiple proxies. Note that this approach has the advantage that the user and server agent implementations of the data agents are identical, and need only be tagged with the desired agent type when they are loaded. This approach also means that in the server agent scenario some extra code is downloaded to the client to test for the upstream companion filter. As the code for the filter itself need not be downloaded unless needed, the penalty is small, is paid only infrequently due to caching at the client, and is well worth the reduction in effort required to support different application structures.

4.5.3 Implementation Overview

As mentioned earlier, the weather browser is implemented by a group of cooperating agents. Listed below are the main component agents, their functions, and implementation issues they present. Note that the first two agents are indivisible entities, whereas the rest are all data agents, partitionable as described in the previous section.

- The *map browser agent* presents the end user with a graphical interface to the application. The map browser allows the user to query the geographical name server for a location to map, and presents the integrator with the location selected. Thereafter, it displays maps and markup information under control of the integrator until the user chooses a new location to browse. The map browser can be run as either a user agent or a server agent.
- The *integrator* obtains a map reference for the map browser to display, and interacts with the data agents to obtain the information used to mark up the map. Although the integrator

can be run as either type of agent, it must execute in the same Oasis proxy as that hosting the map browser. This is because the interface between the map browser and the integrator is based on procedure call rather than URL-access, causing the two to be tightly coupled. This tightly coupled implementation was chosen for simplicity given that separating the browser from the integrator is not significantly different, in terms of communication cost, from separating all of the data agents from the integrator.

- The *geographical name server agent* is a data agent encapsulating access to a Web gazetteer provided by the U.S. Census Bureau [53]. As described earlier, it can executed as a server agent, or as a user agent when partitioned. Partitioning is usually beneficial for this agent, as it allows for filtering of extraneous data before that data is transmitted over the constrained link. Results returned from the geographical name server queries are tagged, via a custom validator in the controllable Oasis cache, as never expiring.

- The *map source agent* is a data agent encapsulating access to a Web-based mapping service. The implemented example utilizes the U.S. Census Bureau's TIGER mapping service [52]. The map server agent interacts with the controllable cache to indicate to the cache that map data never expires. Although the map source agent can be partitioned if desired, partitioning does not produce a significant difference in communication cost, as the map server does not perform much data manipulation. However, a practical advantage of the partitioned scenario is that any accessed maps are cached at the Oasis proxy on the server side as well as at the one closer to the user; if multiple users are accessing the same service from different client machines, they benefit from maps that are already in the cache of the server-side proxy. Note that maps would not ordinarily be cached at the server side in the absence of the partitioned map source agent because the maps from the Web server are dynamically generated, and not identified as cachable by standard Web proxy caching algorithms.

- The *weather station locator agent* encapsulates access to a database of weather stations in the United States. The database is available as a text file via the Web. As such, this agent performs a significant data manipulation operation – given a rectangular area within the US, the agent searches the text file and generates a list of the stations falling within that rectangular area. If this agent is partitioned, the data file need not be downloaded in its entirety. Rather, only short queries and responses cross the communication link. If the agent runs as a server agent, it needs to download the entire database on the first access, but subsequent accesses require no more communication, as the controllable cache is instructed that the database does not expire. In practice, the mode in which this agent executes depends largely on the number of queries that an application expects to produce.

If only a few distinct regions are to be queried, downloading the entire database is not warranted, but if a great diversity of regions is to be queried, the initial cost may be justified. Of course, the agent could have been implemented to dynamically decide to switch from user agent mode to server agent mode dynamically, depending on the access pattern. This is easily implementable in practice but has not been necessary for the purposes of the evaluation.

- The *weather data agent* encapsulates access to a database of weather data. Weather data agents have been implemented for multiple weather databases on the Web, and can be selected at runtime. As with other data agents, these too can be partitioned. Partitioning of weather data agents into a user agent relationship is generally favorable due to two factors. Firstly, weather data agents generally need to process the retrieved data to eliminate extraneous data and HTML markup. As with the geographical name server agent, communications can be reduced by performing this data manipulation on the server network. Secondly, the weather data agents accept compound queries requiring data from several weather stations, usually corresponding to the set of weather stations to be queried for a given map. These aggregate queries must typically be decomposed into individual queries when the weather data server is queried. When the weather data agent runs as a server agent, this means that multiple queries must be issued to the weather server over the constrained link, involving multiple high-latency round trips, and increased communication due to HTTP protocol overhead on each request. When the weather data agent runs as a user agent, the aggregate query can be transferred to the server network and executed there, eliminating the multiple wasteful round trips. Thus, unless there is a need to cache weather data for individual weather stations (which might be required by an usual access pattern producing several partially overlapping aggregate queries) partitioning is usually beneficial. The weather data agent demonstrates the use of a custom cache validator that implements a value-based expiration scheme for weather data. In particular, aggregate queries and individual queries returning valid data are cached until they reach a certain age. However, responses wherein the weather data server indicates that a station is unknown are recognized and are cached for a different, much longer, period of time. These responses arise because weather servers usually do not carry data for all known weather stations. As the unknown status of a station at a server is unlikely to change, the retention of this status conserves bandwidth and elminates costly uninformative round trips to the server.

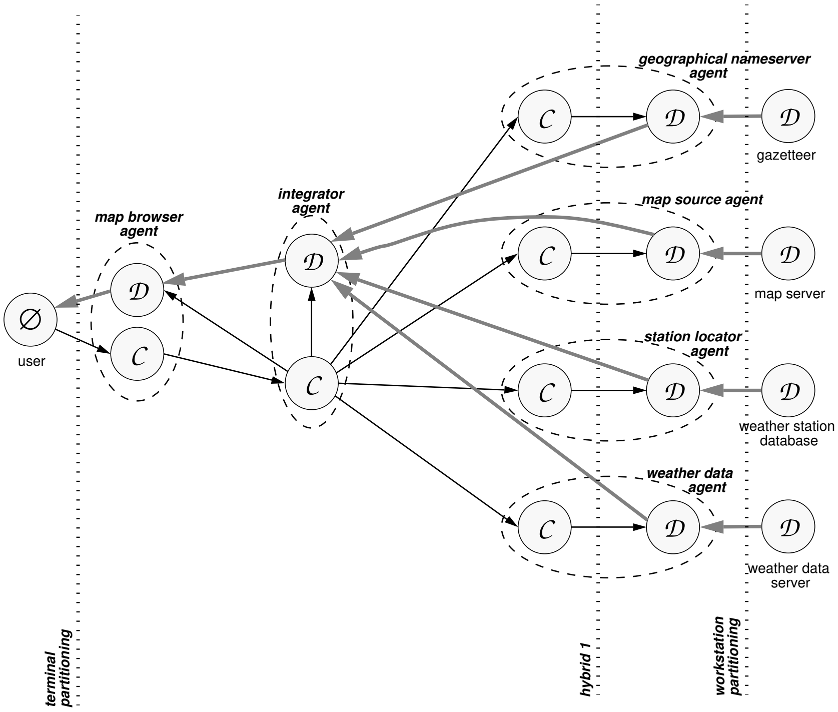
4.5.4 Experimental Setup

The weather data browser provides a means to compare the communication performance of an application designed explicitly to take advantage of control-based partitions to that of the same application as it might have been implemented using the terminal or workstation models. In order to perform this comparison, the weather browser was partitioned across a communications link in different ways, and all communication across the link was measured while selecting, querying, and updating the geographic data for a particular region. Four different partitionings of the weather browser were measured. These partitionings are compared graphically in Figure 4.8, and depicted in detail in Figures 4.9, 4.10, 4.11, and 4.12. The four partitionings were:

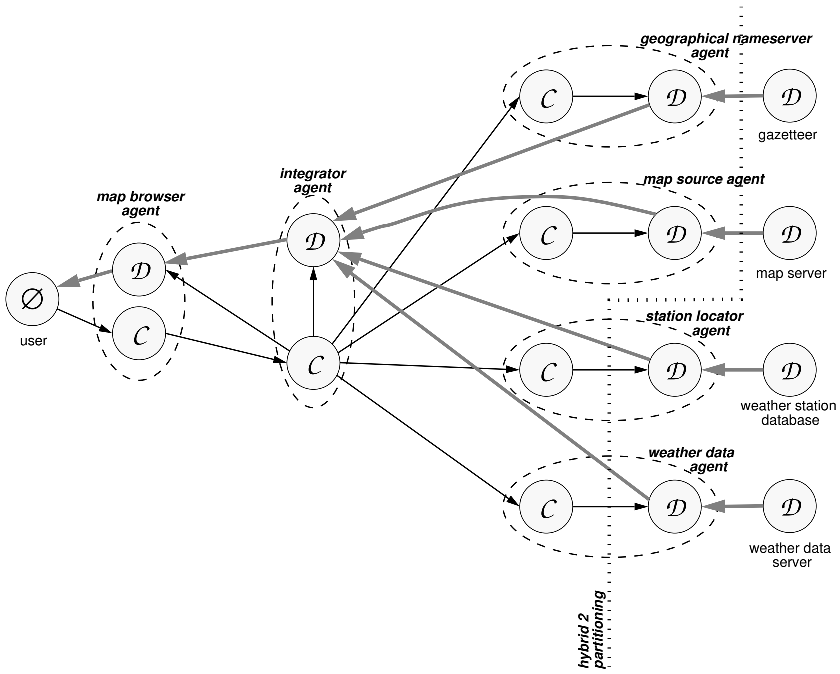
- the *terminal partitioning*, in which all of the agents executed on the server network, and the user's machine acted only as an input/output device. Communication to and from the user was via the X-window system, and consisted primarily of transfers of pure data, as depicted by Figure 4.9.
- the *workstation partitioning*, in which all of the agents were configured to run as server agents running on the user's machine. Communication with the server network primarily represented accesses to the underlying databases, as can be seen from Figure 4.10.
- *hybrid scenario 1*, wherein the map browser and integrator agents were configured as server agents, but all data agents were partitioned into a pair of agents consisting of a user-side proxy object controlling a server-side data manipulation agent. Communication occurring across the constrained link in this scenario represented control interactions and the unavoidable pure communication components of the original data transfers. This situation is depicted in detail in Figure 4.11.
- *hybrid scenario 2*, wherein the map browser, integrator, geographical name server and map source were configured as server agents, while the station locator and weather data agents were partitioned as in hybrid scenario 1. As might be expected, the resultant communication across the constrained link was composed primarily of data interactions required by the geographical name server and map source plus the control and pure communication interactions required by the partitioned station locator and weather data agents. This scenario is illustrated in Figure 4.12.

Because the X window system protocol is not optimized for use over constrained links, the steady-state map update cost of an "ideal" terminal implementation was also measured for comparison purposes. The update cost of an ideal implementation was approximated as the cost of transferring, on every update, a compressed GIF [11] format image containing a screenshot of the image presented by the map browser's GUI. The ideal model assumes that there is no protocol overhead, and no cost for transmission of user keystrokes back to the server.

In brief, these measurements demonstrate that this application, when partitioned along control-oriented lines, is able to achieve a substantial reduction in traffic. In the steady state, about 95% of all update traffic was eliminated, as compared to a workstation or ideal terminal implementation of the same application. (i.e. update traffic was reduced in size by a factor of about 20). These results are presented in detail in the next chapter.



(a) The terminal, workstation, and first hybrid partitioning schemes



(b) The second hybrid partitioning scheme

Figure 4.8: Partitions of the weather browser application that were studied in the experiment. Note that the data path labels from Figure 4.3 have been omitted here for clarity.

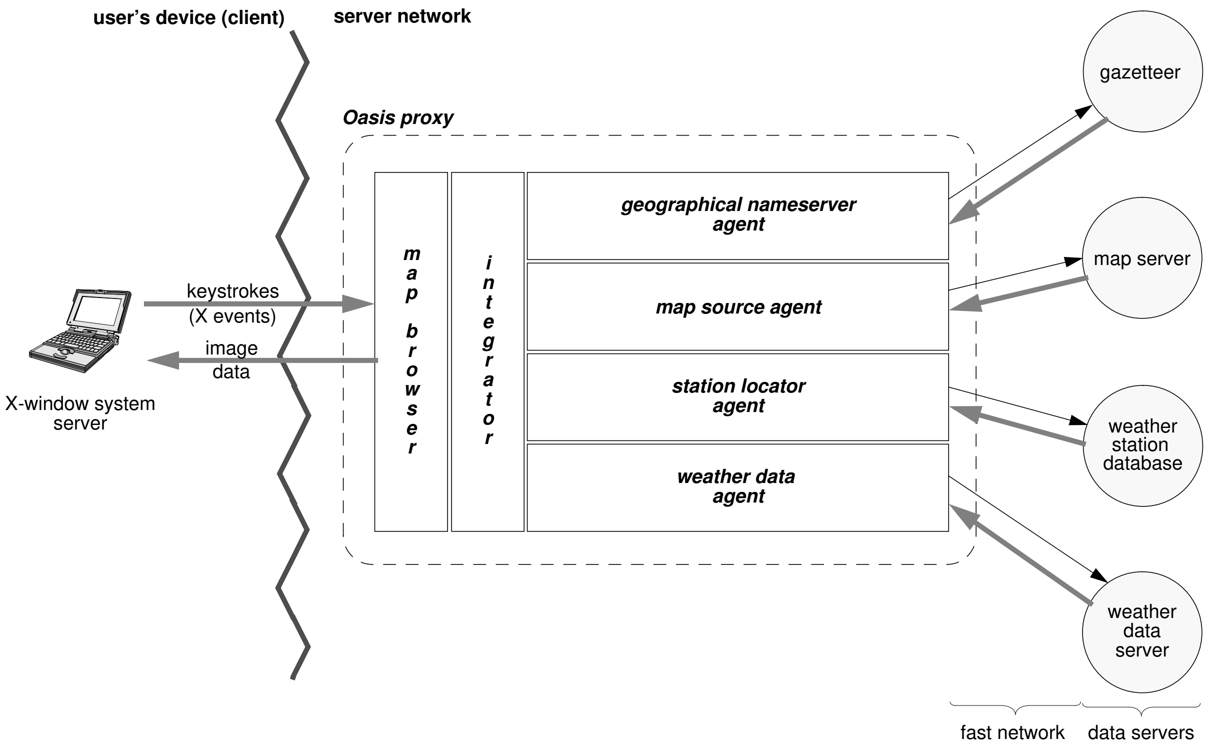


Figure 4.9: The terminal partitioning is characterized by transfers of pure data between the application and the terminal. These transfers correspond to input keystrokes from the user and application output to be displayed. In the evaluated example, the X window system was used for application interaction with the user, and thus the data transfers are encapsulated as X window system key events and image drawing requests.

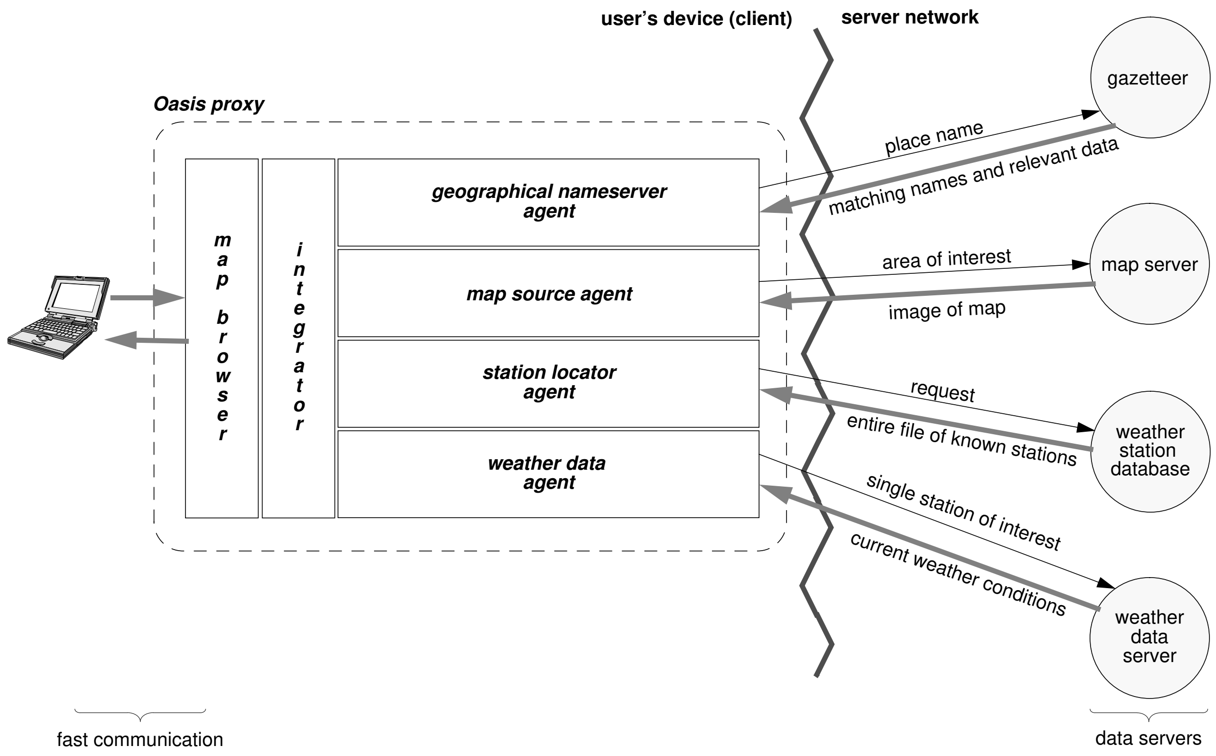


Figure 4.10: The workstation partitioning is characterized by communication with data servers on the server network using the native query and response formats of those servers, which may not be well suited to interaction over a constrained link. In this example, pure data communication occurs in response to queries sent to the data servers involved. The gazetteer responds to a geographical name query with a list of all matching names and data related to those matches, including geographic location data; the map server dynamically generates a map image for a specified location; the weather station database is simply a published data file, and, in this scenario, is downloaded in its entirety when first referenced; the weather data server, when queried, provides a variety of meterological measurements for a single weather station.

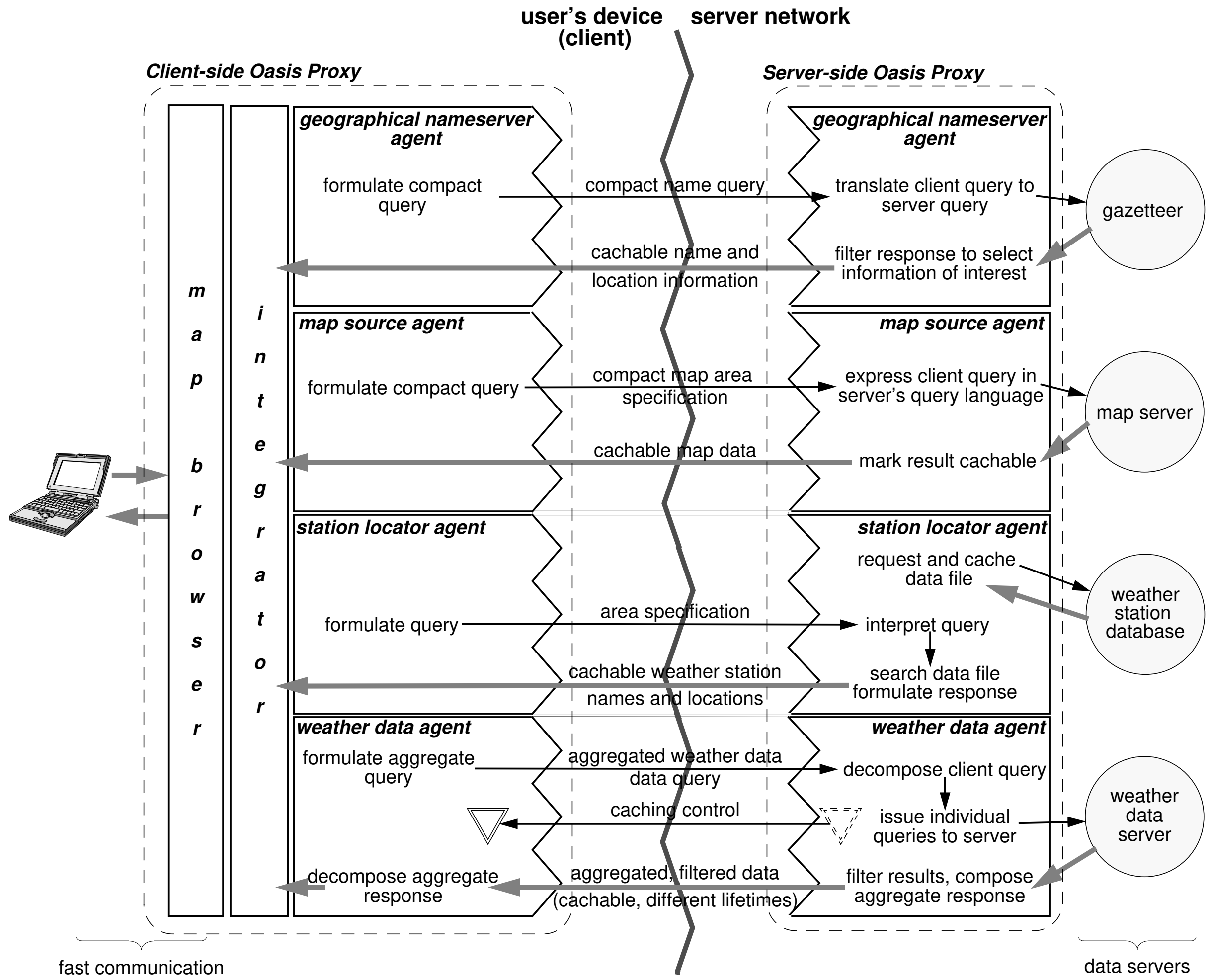


Figure 4.11: Communication occurring across the constrained link in hybrid scenario 1 reflects the benefits of allowing control interactions to occur in both directions across the constrained communication link. Each data agent is partitioned into a pair of components controlling one another, and every such component presents an interface for controlling the manipulation of data it accesses. In effect, each data server is thus augmented with a control interface. The control-oriented nature of the interface imposes a service partitioning requiring optimization of communication before it occurs. In the reverse direction, server agents such as the weather data validator create a similar mechanism to optimize cache-related communication before it occurs. The resultant optimizations are simply effects of requiring each service to be partitioned such that data-reducing operations occur before communication where possible. In this example, these effects correspond to the discarding of irrelevant inputs (prefiltering), query aggregation producing a reduction in protocol overhead, and elimination of unnecessary communication operations through awareness of the underlying data (e.g. through the use of the controllable cache). The data transfers that remain in this control-oriented partitioning are smaller than their counterparts in the workstation model, and correspond to those parts of the original transfers that are pure communication and cannot be eliminated simply by use of a control-oriented partitioning.

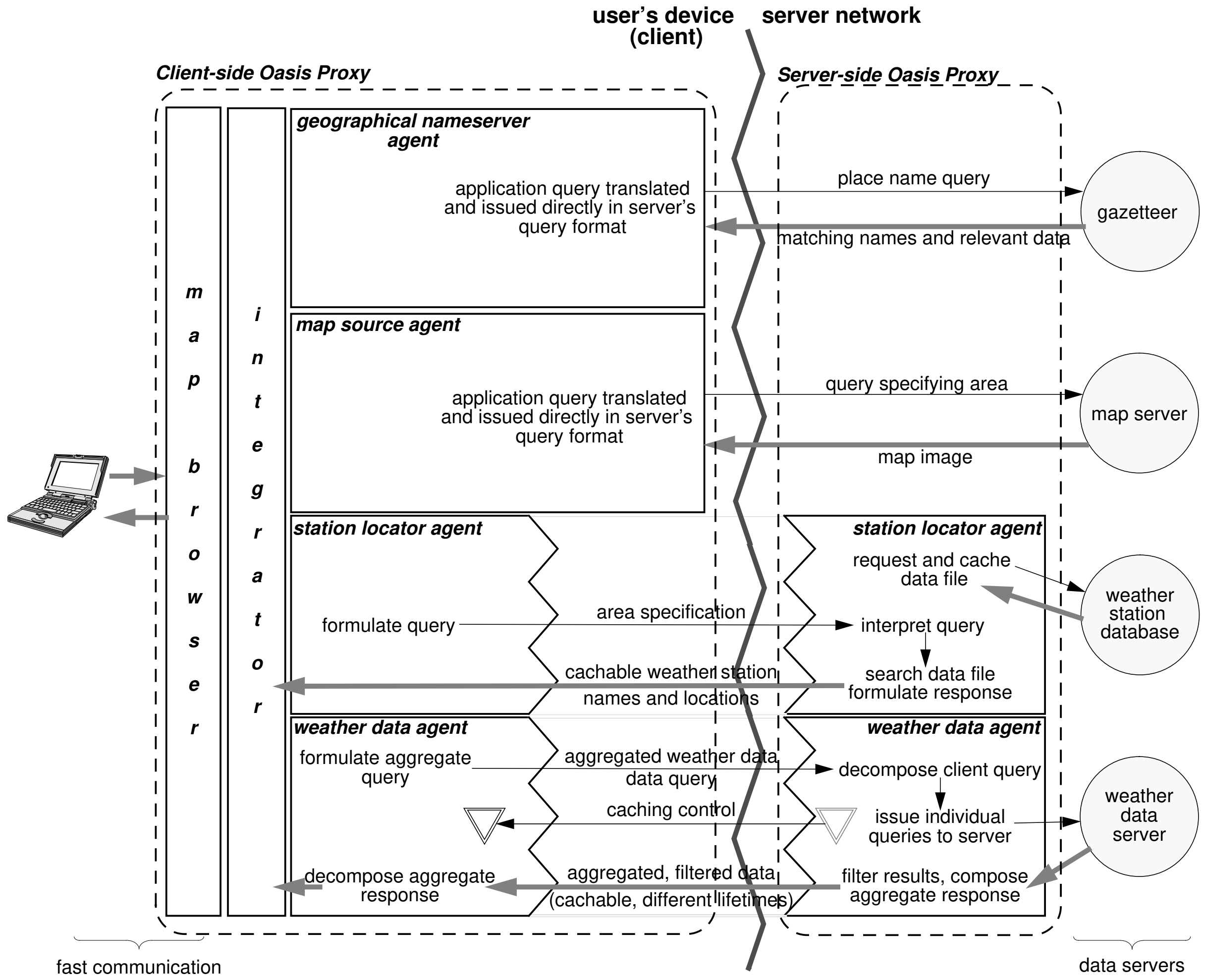


Figure 4.12: In the second hybrid partitioning, the geographical name server agent and the map source agent are not partitioned, and reside entirely on the client-side proxy, as in the workstation scenario. The station locator and weather data agents are partitioned just as in the first hybrid scenario. As might be expected, the traffic across the constrained link in this situation is simply composed of the respective traffic components from the workstation partitioning and from the first hybrid partitioning.

Chapter 5

Experimental Results

This chapter presents the results of the experiments that were described in the previous chapter. For each experiment, a description of the measurements taken is presented, followed by the measurements themselves, and an analysis of the results.

5.1 Experiment 1: Oasis Costs and Benefits

5.1.1 Experimental Synopsis

In order to characterize the costs and benefits that affect Oasis applications, the Oasis framework has been evaluated in the context of a test environment using two different constrained communications links of varying capacity – a modem link, typical of those commonplace in households today, and an ADSL (Asymmetric Digital Subscriber Link) connection, of the type expected to become common for Web access over the next several years. In the tests, both the user's machine and the server machine were running the Linux 2.0 kernel and version 1.1.3 of the Java Development Kit [21], augmented with the TYA 1.0 just-in-time compiler [25]. The server machine was a dedicated 300 MHz Pentium II processor running the Apache [3] Web server, whereas the user's machine was an older and slower 66 Mhz Pentium processor concurrently running a user's browsing and editing software in addition to the test code. In each test, an Oasis proxy or standalone program executing on the user's machine communicated with a Web server executing on the server machine via either a 28.8 Kbps modem, or via an ADSL line. The ADSL line tested was rated to provide maximum data transfer rates of 1.5 Mbits/second downstream (towards the user's machine) and 64 Kbits/second upstream. In all tests involving an Oasis proxy, the proxy was preloaded with 5 commonly used filter agents in addition to the test agent, and where applicable, the caching layer filter. The other preloaded filters provide functionality such as Web access anonymization, commercial message removal, and ftp redirection as described in Section 3.6.

As described earlier in Section 4.3.3, both the modem and ADSL links were tested using a data referencing agent run as an Oasis agent, and as a standalone Java program. The ADSL line was also evaluated using a C program that was equivalent to the standalone Java program, in order to ensure that the overhead imposed by the Java runtime was not limiting the bandwidth measured. The data retrieval characteristics for objects in cache were also measured for accesses from both Oasis agents running within a caching proxy, and for external programs using the proxy to dispatch data requests. Lastly, a separate set of agents was used to measure the overhead of composing two agents in an Oasis proxy. Note that in every test involving an Oasis proxy, the proxy ran on the user's 66 MHz Pentium machine. Thus, **all costs and benefits measured are those of deploying the proxy on the relatively slow user machine**. If the user's hardware is replaced by a faster processor on an equivalent communication link, the overheads of using the proxy, currently dominated by processor time, are reduced. The benefits of using the proxy are correspondingly increased.

5.1.2 Results and Analysis

Communications Constraints

As described earlier, communications performance in the various test environments has been evaluated by measuring the time needed to access files of various sizes in each scenario. Performance is thus evaluated while including all protocol overhead necessary to perform the file accesses. For example, the protocol overhead incurred by a proxied agent accessing a single byte test file is the transmission of a 156 byte request over the underlying network from the requestor to the server, and the reception of 173 bytes of response headers in addition to the accessed data. This overhead is almost constant for all other scenarios and file sizes, varying only by a few bytes that depend on the filename and scenario in question. Although the overhead is negligible for large file accesses, it is significant for smaller accesses. In particular, this overhead affects the measured access latency greatly, because a full request must be sent, and a full set of response headers read back, before the first data byte from a URL can be accessed. Access latency is thus treated as the time elapsed before the first data byte can be read from a URL. The protocol

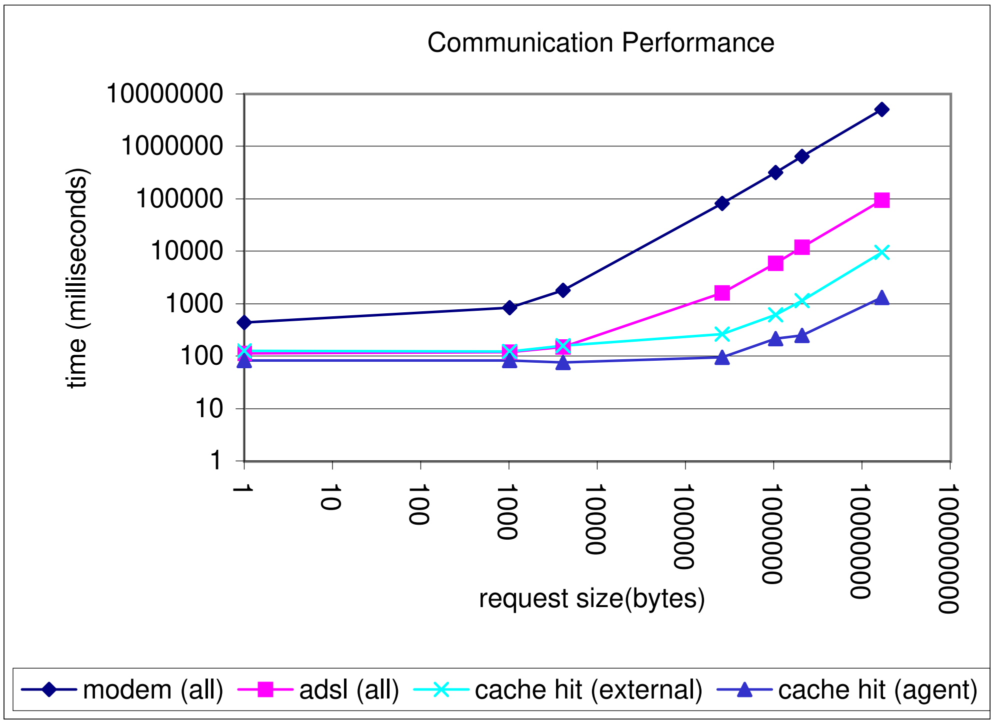


Figure 5.1: Summary of the communications performance measurements. Measurements were taken for URL accesses of size 1 byte, 1 KB, 4 KB, 256 KB, 1 MB, 2 MB and 16 MB. The two modem data sets (*standalone java* and *cache miss*) are presented as a single line, as are the three ADSL data sets (*standalone java*, *cache miss* and *standalone C*). This has been done because the members of each group are indistinguishable from one another on the scale of this graph.

overhead similarly affects the measured bandwidth for accesses to small files. Note, therefore, that the measurements of bandwidth and latency presented here do not simply correspond to the bandwidth and latency obtainable by the underlying TCP/IP on the communication medium in question.

Figure 5.1 summarizes the communications measurements from the various test scenarios. As expected, the throughput obtained by each request increases with request size until the data referencing agent is able to saturate the communications channel, whereafter data retrieval time increases linearly. On the scale of the figure, there are no visible differences between the different scenarios examined for each communication medium, so the scenarios for modem access and ADSL access have been collapsed into two lines in the figure. Figure 5.2 and Figure 5.3 present subsets of these measurements to allow comparison of the file retrieval bandwidth and latency

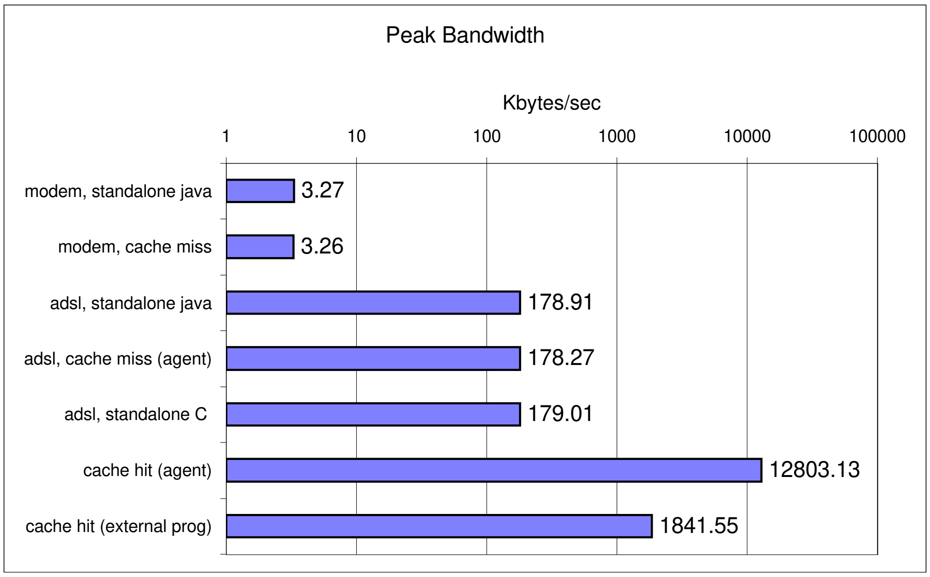


Figure 5.2: The peak bandwidth obtained by each experimental scenario. These numbers determine the rate of linear increase in retrieval time, once the size of the file being retrieved exceeds a minimum dependent upon the communication medium. In all scenarios, variation in the measured peak bandwidth was less than 2%.

characteristics obtained with and without the proxy.

Figure 5.2 demonstrates that the use of an Oasis proxy on a 28.8 Kbps modem connection or on an ADSL connection does not result in more than 0.04% reduction in peak communication bandwidth obtained. However, this tiny reduction in bandwidth may be due to variations in network conditions rather than to the use of the proxy. The peak bandwidth obtained on a cache hit demonstrates that the Oasis proxy can be used without becoming a bandwidth bottleneck even on much faster connections. The performance of the external C client, which is essentially identical to that of the Java clients, also confirms that the Java implementation was not a bottleneck in the ADSL scenario. Lastly, the difference between an internal agent and external client in bandwidth obtained on a cache hit is mainly due to the fact that an agent executes, in the Linux Java environment, within the same address space as the Oasis proxy and the caching layer whereas an external client does not. As such, the agent is able to exchange data with the proxy and with other agents simply by moving data around with no operating system involvement. Transmission of data between the proxy and the external client, on the other hand, requires repeated expensive crossing of the protection boundaries between the two address spaces and the kernel, resulting in

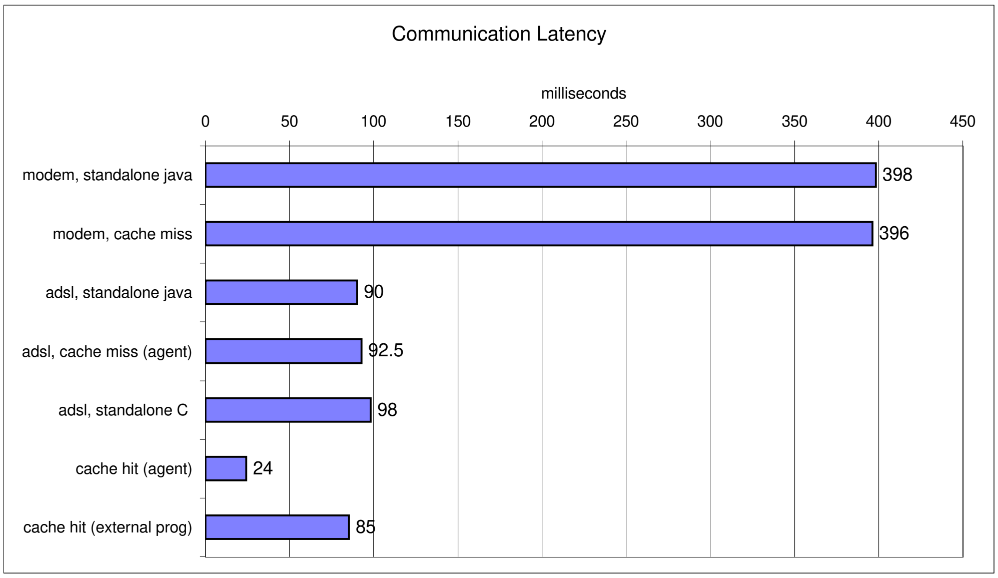


Figure 5.3: The time taken by each tested scenario to retrieve a 1-byte file. The measurements presented are median values obtained from 200 trials. In all scenarios, the median values measured varied by approximately 2% between experimental runs.

substantial data copying at each step.

Figure 5.3 shows the measured median latency of reading a single byte file in the various test scenarios. The figure compares median measured latency rather than mean latency in order to exclude the effects of certain external sources of variation, as discussed at length later. The charted data demonstrate that the use of the Oasis proxy does not add significant overhead to access latency. For modem accesses, the overhead of the proxy was completely undetectable. The ADSL measurements, on the other hand, reflect an increase of 2.5 ms in the median access latency when using the proxy. However, the actual cost of using the proxy is likely to have been too small for this experiment to accurately determine, given the presence of numerous external factors described below. As an examination of the ADSL data sets shows a difference of only 1 ms between the minimum access latencies measured with and without the proxy, the true cost of utilizing the proxy is likely to be between 1ms and 2.5ms; 2.5ms is henceforth treated as an upper bound on the increase in access latency that can be attributed to use of the proxy.

Figure 5.3 also demonstrates that a C equivalent of the data referencing agent is unable to achieve better access latency than the Java agents. The higher latency is incurred because the C language

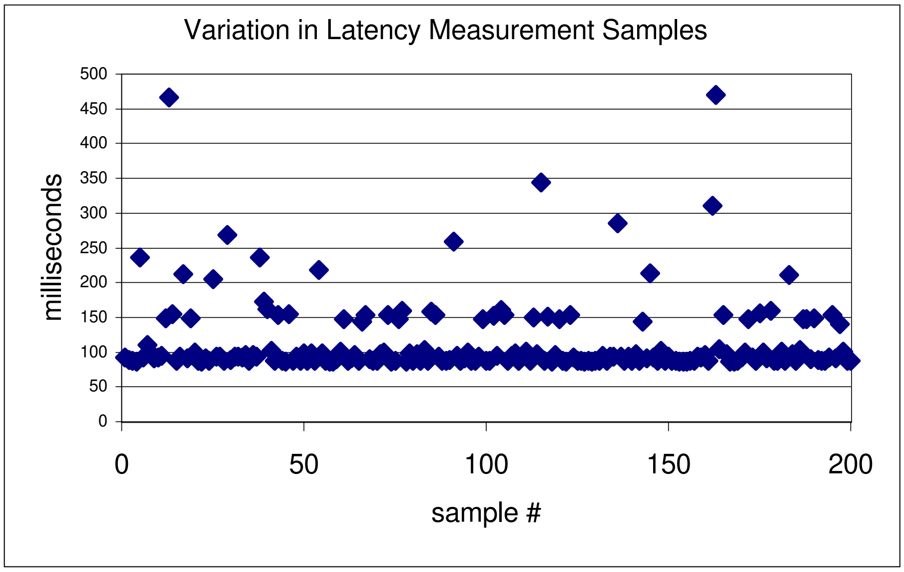


Figure 5.4: The variation in access latency experienced by the data referencing agent using the proxy to access a single byte file. These samples are those of the *adsl, cache miss (agent)* data set. Over half of the 200 accesses cluster around a single value, but occasionally, external events influence the measurements, causing significant outliers. The outliers cluster around specific values, probably corresponding to the events that caused them.

runtime, unlike the Java runtime, does not cache the results of hostname lookups that are performed while accessing the test URL. If the C code is reorganized (in a manner specific to this test) to eliminate duplicated hostname lookups, an improvement of 20 ms in access latency results, resulting in slightly better overall performance compared to the standalone and agent Java implementations.

It is noteworthy that all comparisons of access latency measurements collected in this experiment have been carried out by comparing median, rather than mean access times, in order to factor out the effect of external influences. Although the measurement iterations of this test were spaced out carefully to avoid local resource contention, the test is very susceptible to external events occurring in the underlying shared computing and network resources. Each measured access is relatively short and factors such as thread scheduling within the Java runtime, page faults, processor scheduling, availability of network buffers, delays at network routers, network errors and resultant retransmissions (significant for modem access) or simply network contention can all add a significant amount to measured access latency. Figure 5.4 illustrates the variation encoun-

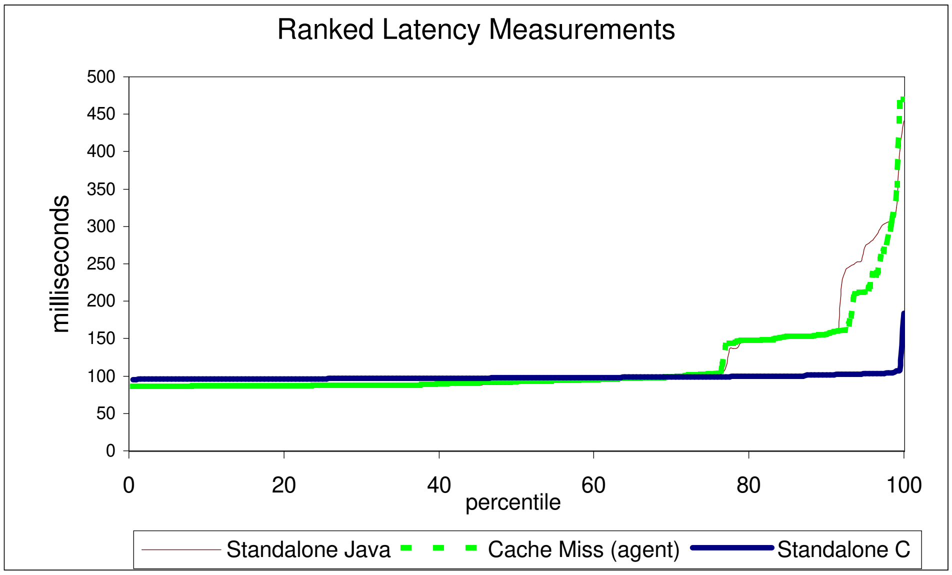


Figure 5.5: Ranked latency measurement samples for the three ADSL data sets. As more than half of the samples in each data set are uninfluenced by external events, the median is a good basis for comparison of latency measurements.

tered in a typical set of latency measurements. Notice that outlying measurements are clustered at certain values, almost certainly corresponding to the additional overhead of specific events that created them. This distribution is *not* a normal distribution. Measurements cluster around the true latency, and all outliers lie above the true value. Thus, the number and magnitude of outlying measurements, and therefore the arithmetic mean of the data set, depend on uncontrollable events external to the experiment. Consequently, the arithmetic mean is not a metric suitable for comparing the results of the various tests. For example, the arithmetic mean of the data presented in Figure 5.4 is 114.56 ms whereas the median is 92.5 ms. Between the data set presented, and other data sets of 200 latency measurements from identical scenarios, the arithmetic mean was observed to vary by as much as 20%, while the variation in the median was about 2%.

Figure 5.5 depicts the ranked measurements from the three ADSL data sets, and embodies the rationale for choosing the median, rather than the mean, as a metric for comparison. Notice that in all three of the data sets, well over half of the measurements are uninfluenced by external factors. The "steps" in the ranked data sets correspond to effects of different types of external event; differences in the number of steps represent differences in the number of events to which a particular scenario is susceptible. For example, the C program is not susceptible to the effects of

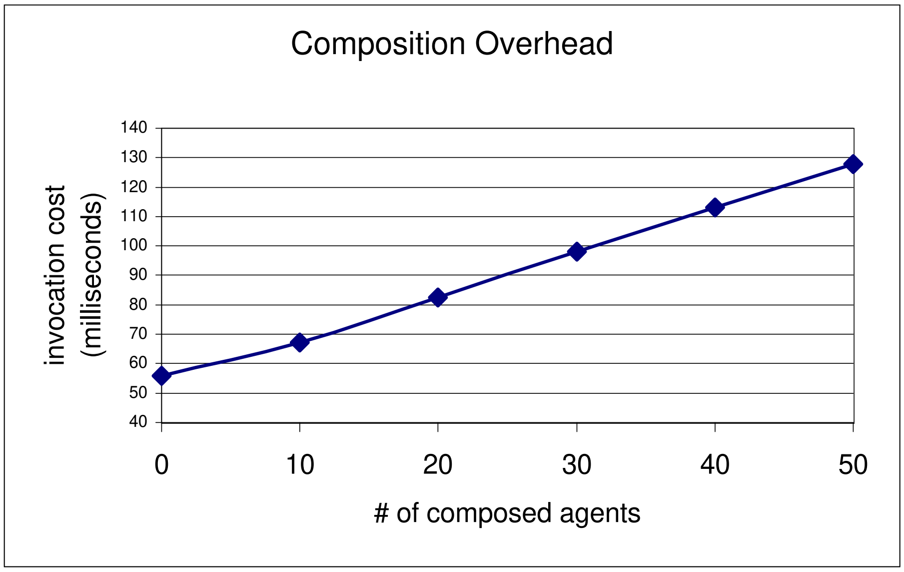


Figure 5.6: The increase in the cost of invoking a method on an agent, as a function of the number of composed layers that must be traversed to reach that agent. These measurements were taken from a proxy running on the user's machine.

Java thread scheduling, and, as a smaller, task-specific, program, is less likely to encounter page faults or processor context switches. As such, the ranked data from the C program exhibits fewer steps, but still reflects the infrequent occurrence of large deviations from the base latency. Because more than half of the measurements in each data set are uninfluenced by external events, the median is also uninfluenced by external events, and thus serves as a good metric for comparison between data sets.

Composition Cost

Figure 5.6 shows the measured linear increase in agent invocation time as null layers were added to the experiment. The slope of the graph is the incremental cost per composition, 1.47 milliseconds per composition. This is only 2.5% of the cheapest possible (i.e. null) agent invocation in the tested configuration, and would be an even smaller fraction of any real invocation. (Real invocations require additional time to process the actions invoked.) As applications usually only have a small number of composed layers, this overhead is acceptable and negligible.

Guaranteed Minimal Gains

The above results characterize the major costs and constraints affecting communication and composition via the Oasis proxy. However, these results are also the basis for the minimum gains guaranteed by the infrastructure. In particular, the communication-related benefits of control can be regarded, at the lowest level, as consisting of the elimination of some round trips to the server, the reduction in volume of some data transfers, and the conversion of some data transfers to cachable accesses. In conjuction with the measurements obtained:

1. Each completely eliminated round trip to the server network reduces application latency by at least 400 ms while communicating over a 28.8 Kbps modem network, or by at least 90 ms while communicating over a 1.5 Mbit/64 Kbit ADSL network. It is important to note that, in the context of a real application deployed over the internet, the realized latency benefit due to an eliminated round trip is likely to be far greater. This is because the network latency in accessing a distant server may be much greater than that of the test scenarios, and because server queries to an online service usually involve server processing delay in addition to file retrieval time. The realized latency benefit is thus likely to range from a few seconds to a few minutes in many applications.

2. Reductions in data transfer produce an application communication delay reduction of at least 306 ms per kilobyte in a 28.8 Kbps modem environment. In the ADSL environment, reductions in *downstream* data transfer (from the server network to the user's machine) produce a communication delay reduction of at least 5.6 ms per kilobyte. Reductions in upstream data transfer have not been explicitly measured by this experiment, but can be extrapolated from the measured results; they should produce a communication delay reduction of at least 130 ms per kilobyte.

3. Communication requests that are converted to cached accesses produce a reduction of at least 375 ms in access latency in the modem environment, or a reduction of at least 66 ms in the ADSL environment. Once again, these latency savings are likely to be considerably higher in a real application due to eliminated processing time and network delays. In addition, every 28.8 Kbps modem network access converted to a cached access produces a saving of approximately $(375 \ + \ 305.7\ k)$ ms, where k is the size of the converted access, in kilobytes. Similary, converted requests for ADSL downstream communication save approximately $(66 + 5.5k)$ ms, and converted requests for ADSL upstream communication save approximately $(66 \ + \ 130\ k)$ ms.

Thus, control-oriented applications constructed using Oasis stand to obtain substantial benefits in communication-constrained environments within a wide range of constraining parameters. The degree of improvement depends upon the application environment, and on the extent to which control-oriented structuring changes the communication behavior of the application. On the other hand, the communication penalties for any application using the Oasis framework are low — less than 0.04% change in obtainable peak bandwidth, and between 1 and 2.5 ms additional communication latency, whether or not the application is control-oriented. The Oasis environment is therefore a practical infrastructure for the deployment of online service applications, providing the potential for large benefits without extracting a significant penalty.

The next two experiments evaluate the extent to which the communication characteristics of an online service are improved by application transparent and application-assisted introduction of control respectively. While this experiment has evaluated timed communication performance within particular test environments and provides the means to relate communication characteristics to timing improvements, the results of the next two experiments are presented in terms of their impact upon application communication characteristics. As such, those results are applicable to any test communication environment in which Oasis is deployed.

5.2 Experiment 2: Diff-based Cache Update

5.2.1 Experimental Synopsis

In order to evaluate the communication requirements of a control-based approach to document refreshing in a Web proxy cache, three proxy caches were analyzed. Each proxy cache used the conservative algorithm, as described in the previous chapter, to select cachable accesses and to determine the lifetime of the accessed documents in the cache. When a document was refreshed (due to expiry from the cache, or due to an explicit refresh request) the original and new values were recorded. For documents that were refreshed multiple times, only the most recent refresh was recorded.

The data in each of the three caches was the result of channelling all of the Web accesses of one or two users through an Oasis proxy over a two month period. The number of users in each case was restricted in order to ensure capture of the maximum possible variety of refreshes, some of which might otherwise have been lost due to capacity constraints combined with the tendency of a cache to favor accesses duplicated by multiple users. Once the proxies had run for two months, each cache was examined to determine the documents that had been cached and subsequently refreshed. Every such document was then examined to determine the type of data accessed in order to select document fetches that would have been made more efficient, or eliminated altogether, using the diff-based update.

The diff algorithm implemented allows diff-based update of textual data using diffs in the format of Unix's "diff -f," and recognition of null updates (i.e., unchanged data) of any type. Thus, refetches of unchanged data and updates to textual data are replaced with a diff update in this scenario. The size of the replacement diffs and associated headers was computed and compared to the size of the original refreshes.

The three caches studied were configured as follows:

- "doublecache" – a cache, shared by two users, in a proxy placed in between the browsers of the users and the server network. This cache was thus secondary only to the on-disk cache of each user's browser.
- "primary" – the first in a chain of two proxies placed in between the browser of a single user and the server network. This user had configured his browser to use only a small in-memory cache. Thus, this cache was essentially the user's primary cache.
- "secondary" – the second in the chain of two proxies behind "primary" above. This cache was secondary to the cache of the first chained proxy.

5.2.2 Results and Analysis

The contents of the three caches studied are detailed in Table 5.1. As can be seen from the tables, all three caches exhibited similar distributions of the types of cache updates. For the "doublecache" cache, Figure 5.7 depicts the classification of the updates that occurred in the two months observed. In this and the other cases, about half of the updates that occurred could have been reduced to null diffs ("unchanged"), and another 39% of the updates ("diffable") could have been replaced by text diffs, leaving only 11% of the updates unchanged. In the diff-based update scheme implemented, the size of the updates replacing the unchanged and diffable updates respectively amounted to 2–3.5% and 3–6% of the of the original total update traffic. In this experiment, null diffs were thus, on average, 6–6.5% of the size of the refreshes they replaced, whereas text diffs were 8–18% of the size of the refreshes they replaced.

"doublecache"					
	# files	headers (bytes)	content (bytes)	total (bytes)	percentage
refreshed	1537	529022	8139832	8668854	100.0
diffable	185	60917	3297767	3358684	38.7
same	1256	435708	3880534	4316242	49.8
Text diffs	185	37000	234440	271440	3.1
Null diffs	1256	251200	0	251200	2.9

"primary"					
	# files	headers (bytes)	content (bytes)	total (bytes)	percentage
refreshed	909	308593	3161942	3470535	100.0
diffable	245	79342	1180393	1259735	36.3
same	620	214880	1724564	1939444	55.9
Text diffs	245	49000	174451	223451	6.4
Null diffs	620	124000	0	124000	3.6

"secondary"					
	# files	headers (bytes)	content (bytes)	total (bytes)	percentage
refreshed	936	313207	3058105	3371312	100.0
diffable	284	89264	1226767	1316031	39.0
same	612	211287	1663602	1874889	55.6
Text Diffs	284	56800	88592	145392	4.3
Null Diffs	612	122400	0	122400	3.6

Table 5.1: Analysis of the refresh traffic observed in each of the three caches.

These results demonstrate the large savings in communication that may be obtained simply by identifying and exploiting periodic rereferences to certain data sources on a server network. The savings identified represent only those applicable to conservatively cachable accesses whose static and dynamic components can be mechanically identified and separated. Note that within this domain, gains even greater than those demonstrated are possible through further optimization. For example, the reductions in traffic could be improved further by utilizing a more compact diff format. Such optimizations have not been explored further, as this experiment serves to demonstrate the potential benefits of introducing control-based partitioning where there was none before, rather than to evaluate the performance of any specific implemented partitioning.

As mentioned earlier, the retention of the conservative caching algorithm allows this experiment to be carried out in the context of a traditional Web proxy cache, but is also a major limiting factor in the identification of eligible data access patterns. The conservative caching algorithm usually eliminates the caching of accesses to sources of data that are constantly updated, i.e. accesses to "online services," and favors the caching of unchanging documents that are likely to be valid indefinitely, or not to be rereferenced after expiry from the cache. In fact, data that were cached and later reloaded due to expiration of the copy or update at the source constituted no more than 10% of the total contents of each cache at the end of the experiment.

In order to mechanically capture accesses to online services, the conservative caching algorithm could be replaced by one that caches *all* accesses, while on each access verifying the validity of cached data that is not deemed conservatively cachable. This approach is no more expensive than that of refetching such data on every access, and is less expensive whenever memory of a prior access permits the application of diff-based update. While the modified algorithm would correctly identify more access patterns that can benefit from diff-based update, it would still be hampered by its inability to handle references to non-textual data. However, if the controllable cache is allowed to receive application guidance as to the data types being accessed, ready identification of all data access patterns that are optimizable using diff-based update is possible.

With application guidance, the frequency of updates and validity checks as well as the size of resultant updates can be reduced. In a control-oriented application, the provision of application guidance is easy in the presence of a controllable cache, while the control-oriented nature of the application itself serves to eliminate many updates entirely. Necessary updates, i.e. communication, are still performed, but unnecessary data updates are replaced by more compact transfers of control. This approach is evaluated in the next experiment.

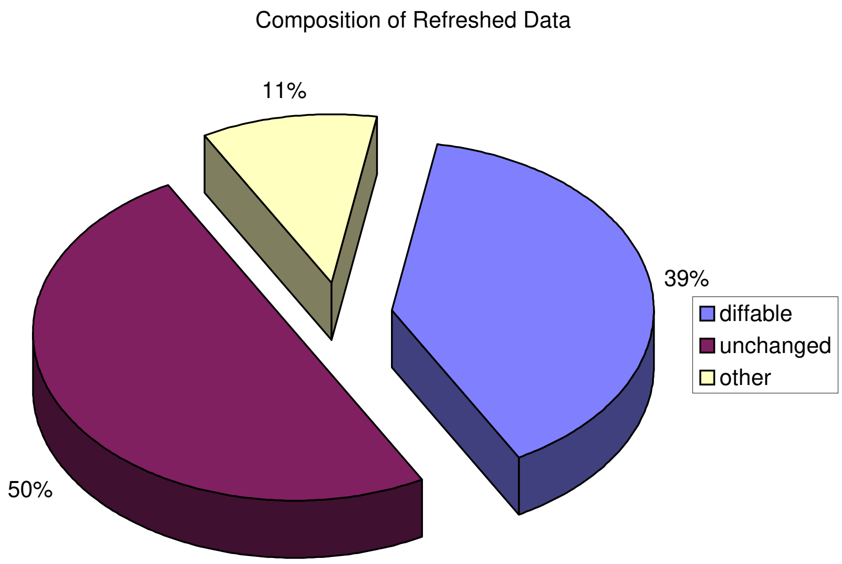


Figure 5.7: Breakdown of the types refresh traffic that occurred in "doublecache." The breakdown of the refresh traffic in the other two caches was very similar.

5.3 Experiment 3: Weather Browsing

5.3.1 Experimental Synopsis

In order to compare the communication costs of the four different weather browser partitionings, each partitioning was initialized and guided through an identical sequence of data retrieval actions. A program interposed between the user-side and server-side Oasis proxies hosting the browser agents measured the amount of communication occurring at each step. These results document the communication bandwidth utilized by the various partitionings. The measured results are compared to each other, and to an "ideal" terminal partitioning defined as described in Section 4.5.4. Communications latency, on the other hand, has not been measured precisely due to the difficulty of characterising the beginning and end of each measured step in the absence of modifications to the application, and due to the wide variation in latency caused by variation in the response times of the underlying data sources. However, it has been noted that some partitionings exhibit significantly better latency than others due to the elimination of high-latency roundtrips to the server network. The performance advantage of those partitionings is characterised by presenting the costs of the roundtrips that are eliminated.

Each of the partitionings was evaluated by guiding the system through the steps necessary to retrieve the map for an urban area (Pittsburgh, PA) and to augment that map with weather conditions at all known weather stations on the map, as provided by the weather data server at Texas A&M University. Measurements of the cumulative amount of data sent and received were taken at the following points:

- *initialization* – just after the initialization of the weather browser application, including downloading any uncached agent code.
- *GNS query* – after the geographic name server has been queried for, and returned, all possible matches for the name "Pittsburgh, PA".
- *map load* – after the unaugmented geographical map of the Pittsburgh area has been retrieved and displayed.
- *weather stns* – after the weather station server has been queried to determine all weather stations within the viewable area of the map.
- *complete* – after all the weather data for the map have been retrieved and have been used to augment the weather map appropriately.
- *first update* – after the processing of a user-requested update occuring soon after the weather data query is complete, and before the weather data has expired from the cache.
- *refresh* – after the expiration, and subsequent update of, the weather data. The incremental cost of the refresh is representative of the size of the cost of subsequent updates to the data. As such, it represents the steady state cost of running the weather browser, and is the most important of the measurements taken.

For all partitionings other than that of the terminal partitioning, the measurements have been performed twice. This was done in order to demonstrate that there is no change, other than the change in initialization costs, in going from a cold cache to a warm cache scenario. The terminal partitioning does not involve any caching at the user side, and has hence only been measured once. In addition, the measurements of the implemented terminal partitioning include two additional costs that are not incurred by any of the other partitionings. These are *X init* and *quit*, and they respectively reflect the cost of initializing the connection of the Oasis proxy to the X window system at the user's host prior to the initialization of the weather browser, and the cost of deleting the X window corresponding to the weather browser upon termination of the application.

All of the measured results are presented in the next section.

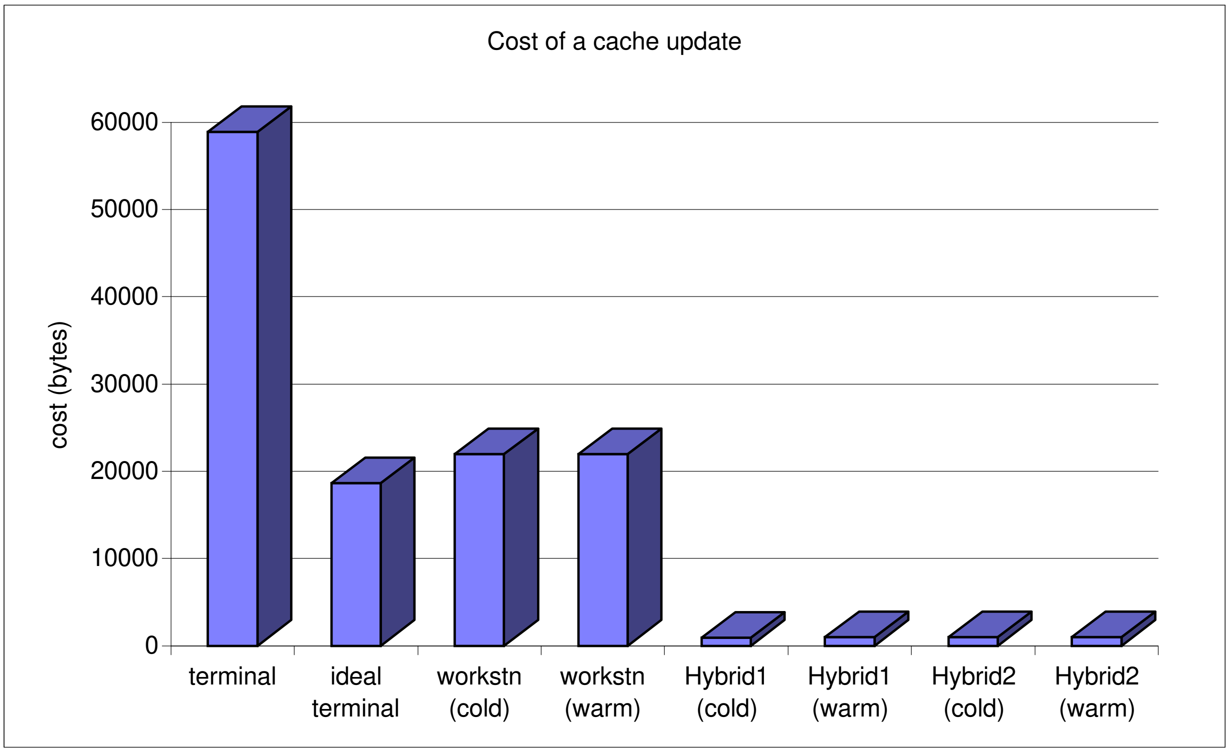


Figure 5.8: Steady-state communications cost of each partitioning. These figures represent the cost of updating the weather map presented to the user. Measurements designated "(warm)" are from experiments in which the cache of the Oasis proxy was preloaded by a previous execution of the weather browser. Those designated "cold" are from experiments where the cache had not been preloaded.

5.3.2 Results and Analysis

Tables 5.2 and 5.3 present the cumulative communication utilization and incremental communication cost of every operation that was measured for each of the weather browser partitionings. These measurements are compared in Figures 5.8, 5.9, 5.10, and 5.11.

Figure 5.8 presents the steady-state cost of an update when running the weather browser in each of the measured configurations. This cost is the most important of all of the measured costs because all of the other costs, for an appropriately partitioned application, are one-time startup costs and could be incurred at a time when bandwidth is plentiful, such as when a mobile client is physically plugged into the server network. The update cost, on the other hand, must be incurred periodically, no matter what the cost of bandwidth, if the data presented are to be up to date. As

Workstation Partitioning - Cold Cache							
	initialization	GNS query	map load	weather stns	complete	first update	refresh
bytes sent	4475	4697	5851	6085	17193	17193	26123
bytes rcvd	73431	77952	95404	138421	161528	161528	174579
total	77906	82649	101255	144506	178721	178721	200702
Cost	77906	4743	18606	43251	34215	0	21981

Workstation Partitioning - Warm Cache							
	initialization	GNS query	map load	weather stns	complete	first update	refresh
bytes sent	805	805	805	805	9735	9735	18665
bytes rcvd	5653	5653	5653	5653	18704	18704	31755
total	6458	6458	6458	6458	28439	28439	50420
Cost	6458	0	0	0	21981	0	21981

Hybrid 1 - Cold Cache							
	initialization	GNS query	map load	weather stns	complete	first update	refresh
bytes sent	5741	5949	7041	7275	8319	8319	8595
bytes rcvd	68976	69340	86975	87771	97480	97480	98129
total	74717	75289	94016	95046	105799	105799	106724
Cost	74717	572	18747	1030	10753	0	925

Hybrid 1 - Warm Cache							
	initialization	GNS query	map load	weather stns	complete	first update	refresh
bytes sent	2071	2071	2071	2071	2347	2347	2623
bytes rcvd	1198	1198	1198	1198	1887	1887	2592
total	3269	3269	3269	3269	4234	4234	5215
Cost	3269	0	0	0	965	0	981

Hybrid 2 - Cold Cache							
	initialization	GNS query	map load	weather stns	complete	first update	refresh
bytes sent	5115	5337	6491	6725	7960	7960	8236
bytes rcvd	71195	75716	93168	93964	104744	104744	105441
total	76310	81053	99659	100689	112704	112704	113677
Cost	76310	4743	18606	1030	12015	0	973

Hybrid 2 - Warm Cache							
	initialization	GNS query	map load	weather stns	complete	first update	refresh
bytes sent	1445	1445	1445	1445	1721	1721	1997
bytes rcvd	3417	3417	3417	3417	4114	4114	4811
total	4862	4862	4862	4862	5835	5835	6808
Cost	4862	0	0	0	973	0	973

Table 5.2: Communication measurements of the various partitionings. In each group, all rows other than the “Cost” row represent cumulative byte counts measured from the start of the experiment.

Terminal Partitioning									
	X init	initial-ization	GNS query	map load	weather stns	complete	first up-date	refresh	quit
bytes sent	36156	65008	68272	76668	78172	84636	98492	110940	112924
bytes rcvd	8020	24060	27968	599116	604340	635356	685208	731632	733184
total	44176	89068	96240	675784	682512	719992	783700	842572	846108
Cost	44176	44892	7172	579544	6728	37480	63708	58872	3536

Table 5.3: Communications measurements of the terminal partitioning. This partitioning required additional communication prior to initialization of, and upon termination of, the application.

is apparent from the figure. the agent-based partitionings of the weather browser require a much smaller amount of update communication than any of the other measured scenarios do. In particular, the hybrid scenarios perform equally well, and require less than 5% of the communication required by the workstation or ideal terminal partitionings on an update. An update performed by the implemented terminal partitioning, as expected, is significantly more expensive, requiring about three times the amount of communication required by the ideal terminal and workstation partitionings.

This result reflects the fact that the hybrid partitionings of the weather browser permit improved separation of the dynamic and static components of the data being presented, and that both hybrid partitionings are able to exploit the use of data filtration on the server network. In each hybrid scenario, the weather update involves only the data managed by the weather data agent, whereas data supplied by each of the other agents are identified as static and are cached. Both of the hybrid scenarios partition the weather data agent identically, and as a result the two produce the same update cost in the experimental results. In each hybrid scenario, the filter agent component of the weather data agent reduces the cost of updating the weather data by preprocessing the data before it leaves the server network. On the other hand, the poor performance of the terminal partitionings reflects the fact that the static and dynamic components of the data are not separated, and thus that the entire weather map must be fetched on each update. Although the workstation partitioning is able to exploit the separation of dynamic and static data due to its implementation as a set of cooperating agents, it is still unable to reap the benefits of preprocessing the weather data on the server network.

Figure 5.9 demonstrates that the agent-based partitionings are competitive with the workstation and terminal approaches in initialization costs. For the terminal model, the cost of initialization

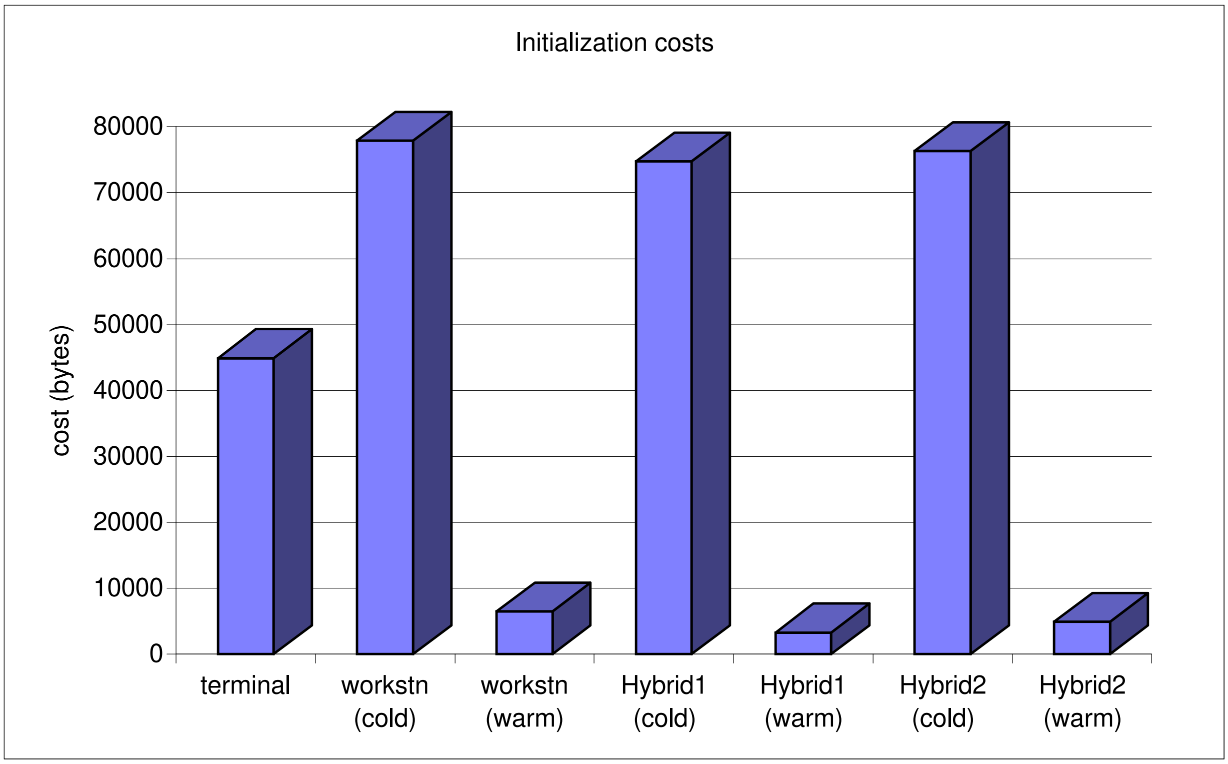


Figure 5.9: The costs of initializing each partitioning of the weather browser application. The initialization costs of the terminal partitioning do not include the substantial "X init" cost presented in Table 5.3.

is the protocol cost of requesting that the window system on the user's terminal create and initialize a window for the weather browser. For all the hybrid partitionings and the workstation partitioning, the cost of initialization is the cost of downloading the code to effect the window creation and initialization. As might be expected, all three of these initialization costs are greatly reduced by caching. Notice that the Hybrid 1 partitioning is cheaper to initialize than Hybrid 2, and that the initialization costs of the two differ by the same amount in both warm and cold cache scenarios. This small difference is due to the extra overhead, incurred by Hybrid 2, of detecting the absence of a server-side filter component for the geographical name server agent and the map source agent. As mentioned earlier, Hybrid 1 could also theoretically profit from the downloading from a smaller amount of agent code, as the user-side proxy need not retrieve the filter agent component of the partitioned geographical name server and map source agents. However, the implementation approach chosen foregoes this advantage in favor of the flexibility to allow easy implementation of all the tested scenarios using identical agent code. In practice,

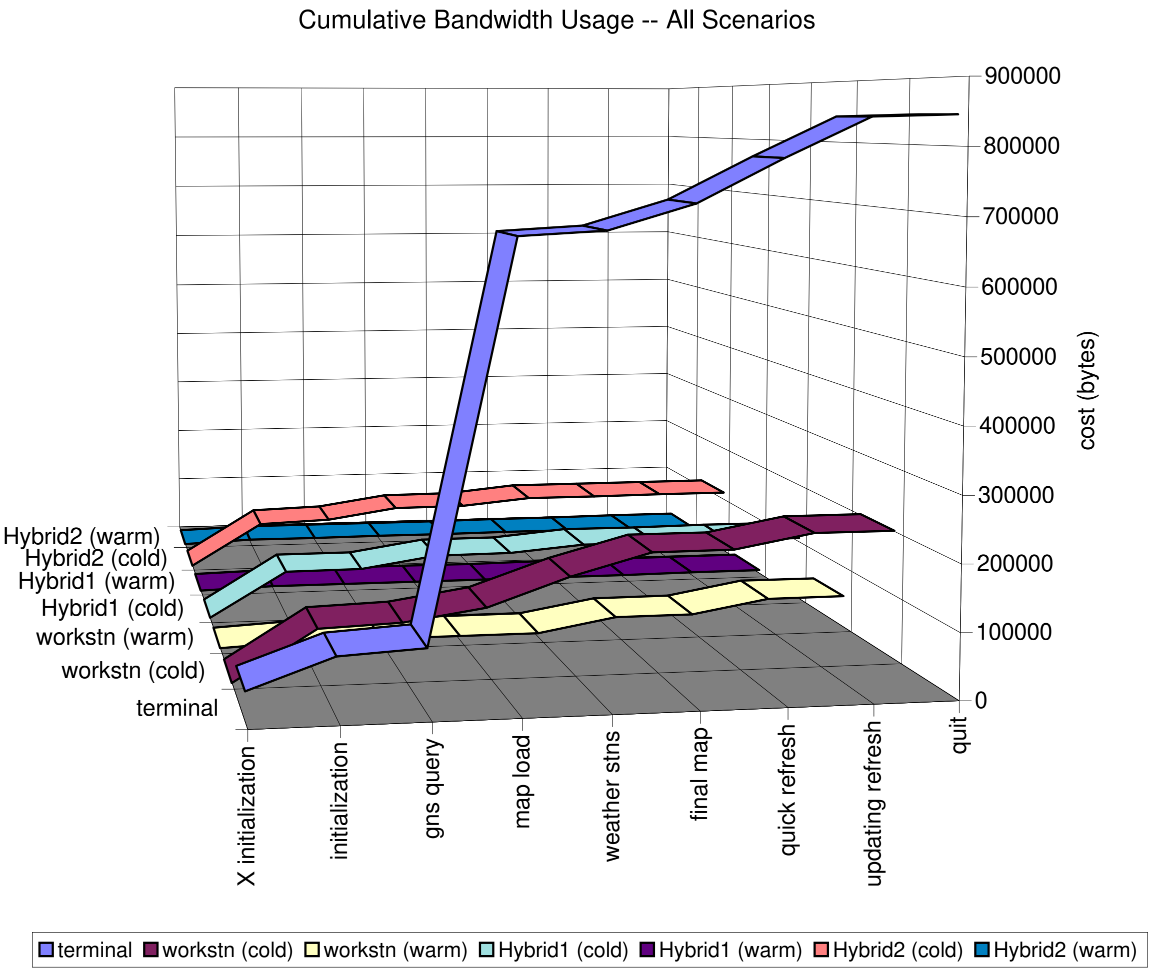
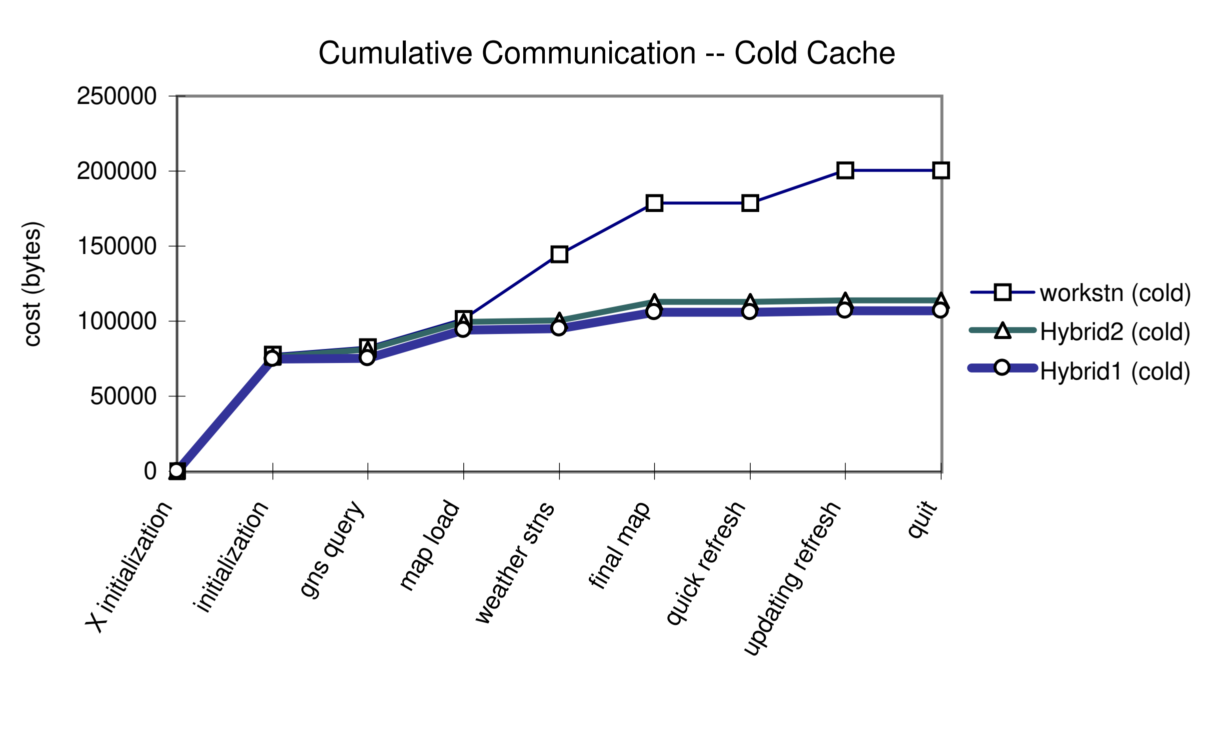


Figure 5.10: The cumulative communication costs of the various partitionings. This figure depicts the significantly higher cost of the implemented terminal partitioning compared to that of any other partitioning. Figure 5.11 presents the same data in the absence of that of the terminal partitioning.

caching ensures that the benefits of any optimization to remedy this effect benefits only the very first run of the application. Caching of the agent code already ensures that subsequent runs are not affected.

Figures 5.10 and 5.11 present the cumulative use of bandwidth by each scenario evaluated. In the latter figure, the costs of the implemented terminal model, which dwarf those of the other approaches in the former, have been removed to permit better comparison of the remaining approaches. In these figures, note that the slope of the graphs between "quick refresh" and "updating



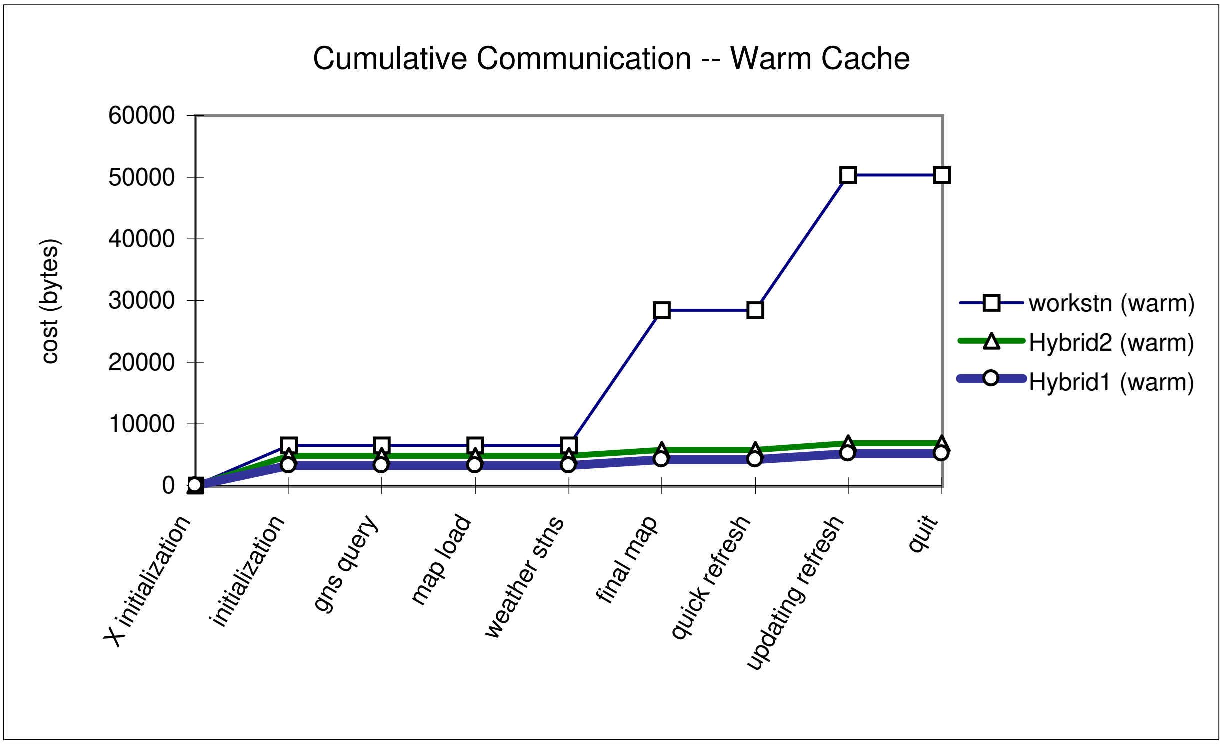


Figure 5.11: Cumulative communication costs of partitionings other than the terminal partitioning. The upper graph depicts measurements taken from experiments run on a proxy that had not already hosted the application, i.e. had a cold cache. The lower graph depicts the same experiments run on a proxy with a warm cache.

refresh" represents the cost of an update, as graphed in Figure 5.8. This is the rate at which the cumulative communication cost would increase with continued viewing of the same, periodically updating weather data set.

Observed Latency Benefits

By changing the nature of the communication across the constrained link, agent-based applications are often able to obtain improvements in application latency that cannot be duplicated by other implementation approaches. In the case of the weather browser, both hybrid architectures, each using a partitioned weather data agent, eliminate multiple high-latency queries of weather data for individual weather stations. Multiple queries of individual weather stations are exported to the server network as a single compound query. On the server network, the server-side part of the weather data agent decomposes this query into multiple queries issued to the weather data server, collates the result, and returns a single composite result. As mentioned before, this reduces communication bandwith used due to reduced protocol overhead, but it also reduces application latency by eliminating high-latency round trips over the constrained link. In the experiement conducted, every round trip eliminated corresponds to a reduction of about half a second in total data retrieval time. This figure has not been measured precisely due to the high variation in response times of the internet-based weather data sources, combined with the difficulty of measuring the latency of individual parts of the data retrieval process.

The hybrid weather browser architectures also reduce latency as they decompose the accessed data into cachable and uncachable components. This allows the caching of data that would otherwise not have been cached because it would have been part of a larger uncachable data type. For example, both hybrid architectures are able to cache the geographical maps underlying the weather maps that are accessed. While a similar combination of data from cachable and uncachable data sources may be achieved in a workstation architecture, support for control permits much greater flexibility in designing this type of decomposition. In the absence of server agents, the data source on a server network has no means to provide any assistance in the handling of its data, e.g., it cannot aid the user in making caching decisions, or in augmenting or interpreting the data supplied. In the context of the weather browser, the hybrid partitionings eliminate duplicate fetches of geographical maps, and where possible, duplicate queries of weather station data. At times of high load, each request made to the map source may take several minutes to process, and each weather station query may take several seconds. Elimination of expensive requests like these whenever possible produces a noticeable improvement in application responsiveness.

5.4 Lessons Learned

The results presented in this chapter demonstrate the potential benefits of deploying a control-oriented infrastructure, the use of control-oriented techniques to reduce communication in unmodifiable legacy applications, and the benefits available to an application constructed specifically to leverage an infrastructure supporting control. However, these experiments also provide experience with the design of control-oriented services and applications, yielding procedural insight that is valuable to application designers.

Control-oriented considerations of importance in the early design and modularization stages of building an online service application are similar to those important in building an application as a distributed object system. In particular, control-oriented design suggests that an application be built around the data-types and data objects that will be manipulated by the application, and that modularization be data-centric. Data-centric modularization encourages the development of modules that interact by controlling one another, exchanging data only when no alternative exists. As demonstrated by the development of the weather browser, one technique by which this modularization can be achieved is by implementing the application as a set of simpler cooperating services, each of which is primarily responsible for a single data type, which are ultimately composed together to form the final product. Note that this type of modularization is often not a requirement in traditional server-client or 3-tier server-middleware-client system designs where an object system is not involved.

Appropriately modularized applications present many opportunities for control-oriented partitioning along module boundaries. An ideal partitioning should be picked to minimize the remaining data movement, and to ensure that the remaining data movement is accompanied by the movement, if necessary, of a control interface to that data. Although the set of possibilities for application partitioning may be large in applications with many subsystems, the bulk of the benefit of control-oriented partitioning can be obtained by identifying and applying these principles to just those subsystems that are responsible for most of the dynamically changing content in the data flow. In the case of the weather browser application, this simplification is exemplified by the Hybrid 2, which in practice performs almost as well as the Hybrid 1 partitioning.

Lastly, it is worth noting that the repetitive development effort required to produce a control-oriented service can be reduced greatly by the identification and development of appropriate domain-specific building blocks beyond those provided by a general-purpose infrastructure. For example, filter agents represent the most specialized building block provided by Oasis, but the structure of the weather browser exhibits some redundancy that can be embodied in a more specialized abstraction. In particular, each of the weather browser's data agents consist of a proxy object acting in cooperation with a data manipulation entity. Although not explored in the implemented weather browser, structural similarities like this provide an opportunity for the construction of application-specific building blocks to increase code reuse and reduce development effort. In the case of the weather browser, an appropriate building block would have been a structure encapsulating a proxy object plus a data manipulating entity and handling the communication and partitioning issues inherent in this separatable arrangement. An application developer deploying several control-oriented services is likely to profit from observing and exploiting opportunities for the construction of similar application-specific building blocks.

Chapter 6

Conclusions

As internet access becomes a household commodity, the number of users connected to its extremities via comparatively slow links continues to grow rapidly. At the same time, the explosive growth of the World Wide Web and its resultant importance as an information resource is encouraging the development of devices that allow ubiquitous, but often limited, internet access. Both of these trends mean that for the foreseeable future, there will be rapid growth in the number of users accessing the internet via constrained communication links. Meanwhile, the nature of the data available on the internet is also evolving. In addition to the static or slowly changing documents that the Web originally contained, more and more online services are appearing, presenting frequently updated data that users will need to access, even over constrained communication links. Addressing this problem has been the motivation for this thesis.

Section 6.1 summarizes the thesis research that has been presented and lists the resultant contributions. Opportunities for future research that arise as a consequence of this work are presented in Section 6.2.

6.1 Contributions

This thesis has proposed and evaluated the use of control-oriented application design and deployment as a means to reduce the communication requirements necessary to deploy an online service in the face of constrained communication. The nature of control has been explored through the development of an infrastructure to support control-oriented application partitioning. The measurement of a set of sample applications has quantified some of the benefits realizable through control-oriented partitioning.

The contributions of this thesis research can be thought of as conceptual contributions, practical contributions arising from the implemented artifacts, and observations based on insights gained in the course of demonstrating the thesis.

Conceptual Contributions

- This work introduces the concept of control as a fundamental application functionality. The thesis defines control and employs it as a central factor in the development of many of the application models presented.
- A taxonomy of applications, arising from the analysis of application functionalities. This taxonomy identifies various application classes, and provides key insight into the nature of the communication employed by communicating applications. It also serves to delineate the application classes to which the techniques presented in the thesis are best applied.
- Basic design principles, including formalized agent models, that an application designer can employ in order to ensure that an online service can be partitioned to leverage control-oriented infrastructure whenever one is available.
- A set of building blocks for control-based applications. These building blocks are proposed as a minimal set that must be supported by a control-oriented infrastructure in order to enable a wide variety of applications. They are not necessarily the only possible such set, but form a set that has proven easy to support without imposing undue constraints on the applications built upon them.
- A mechanism for the integration of control-oriented infrastructure and services with the existing infrastructure of the World Wide Web. The use of a network of computationally capable proxies ensures backward compatibility and easy deployment of control-oriented applications.

Practical Contributions

- Implementation of the Oasis Web proxy, which is an easily extensible Web proxy server as well as an infrastructure for the support of control-oriented applications.

- Implementation of the building blocks for agent support, including Java classes to aid in the development of agent code, and classes to allow agent code to interface to the Web-based functions of the Oasis proxy.
- Implementation of many agent-based extensions to the Oasis proxy that provide functions that are useful to end users. For example, example extensions have been written to support selection and removal of commercial advertising from Web content, access anonymization, traffic monitoring, proxy caching with potential for personalized indexing of referenced data, etc.
- Implementation, as an example application, of a system that allows diff-based cache update in the Oasis proxy systems.
- The development of a system that can be used to compose, or to add value to, existing Web services in a practical manner, producing integrated applications. This is not a distinct implementation component, but rather a practical use of the Oasis system as a whole that is distinct from the use being investigated in this research.

Observations

- Application guidance can have tremendous positive impact on the management of accessed data by intermediaries, especially when composite or opaque data types are being managed. The thesis work has demonstrated this in the context of caching, but it is likely to be true in other arenas too. For example, [33] has obtained similar results in the realm of managing application data access in an environment with variable bandwidth.
- The ability to partition a client of a data server is a mechanism that allows for greater flexibility in overcoming network limitations, simply by allowing for changes in the communication protocol utilized over the remote link. For example, flexible partitioning may allow the aggregation of multiple queries into a single compound query, reducing the protocol overhead incurred by the client.
- Web proxy caches that are currently in common use are not well suited to the types of data access involved in deploying an online service. Growth of the internet and contention at popular online services will eventually require that this problem be addressed at an infrastructure level.
- At a point at which resource availability changes dramatically in a network, it appears to be advantageous to deploy transducers that are aware of, and can help manage, the impact

of the change. This research work demonstrates the effectiveness of transducers placed around the endpoints of a constrained communication link. It mirrors the trend in efforts that address other discontinuities, such as those addressing the large variations in display and input mechanisms that arise when mobile clients share access to a system that also serves desktop workstations.

6.2 Future Directions

This research has opened up several avenues for further exploration. Efforts for which this work may serve as a starting point can be thought of as

1. extensions to the application design and development models,
2. extensions to and alternative uses of the implemented system, or
3. new directions.

Application Design and Development

- The system models and building blocks proposed by this research form one set from which applications can be built in a control-oriented manner. However, it is possible that other combinations of basic agent types may also prove to be more convenient in practice. Further research will determine whether or not an alternative set of basic agent types is more practical for application designers.
- All of the control-oriented structures supported are very simple, in order to ensure applicability that is almost universal. However, if large numbers of control-oriented applications are deployed, it is likely that more complicated, and therefore more specialized, application structures built from simpler ones will become desirable as standard components. For example, as mentioned earlier, standard structures to support fault tolerance and reconnectability would be useful to a wide variety of applications. Similarly, particular structures may be useful for interfacing with underlying protocols or infrastructure. For example, filter agents have proven extremely useful, in the studies described here, for the purposes of interfacing Oasis agents to the Web infrastructure underlying them. Thus, the need for standardization of new application structures will arise with experience; these structures need to be defined and eventually supported in Oasis or other infrastructure supporting control.

Extensions to the implemented system.

- The implemented Oasis artifact presents many opportunities for extension. The most important such opportunity is in the development of security policies to govern trust issues relating to, verification of, and access granted to, downloaded code. The current Oasis implementation provides many hooks for the institution of flexible security policies. However, the policies implemented are very simple, such as the use of access control based on the server from which agent code is obtained. Research that is being carried out in security policy as applicable to other systems based on mobile code is also likely to be applicable to Oasis.

- Oasis could also be extended to handle alternative sets of agent building blocks, as mentioned in the previous section, or to provide a standard means for locating and downloading support for unknown structures. These conveniences will aid the development of large reusable components for rapid construction of applications.

- A network of Oasis proxies presents a flexible framework that can be used for many purposes other than simply for experimentation with control-oriented applications. For example, if the proxy were modified to handle protocols other than HTTP, it could be used as an experimental platform for many "active networking" applications. In the context of the Web, the existing Oasis proxies can be used as a platform for adding value to any Web service, content monitoring and analysis (e.g. indexing all Web sites visited by each user, and using those statistics to predict the user's interest in other sites), automatic translation of Web pages, etc.

New Directions

Oasis provides an enabling technology for a number of broader research topics to build upon. In particular, Oasis enables the creation of networks of controllable objects and applications. In deploying these networks, the following questions arise: How are networks of controllable objects managed? What sorts of resource discovery and adaptation to network changes should be supported in such a network? Does control easily provide all the functionality desired by those developing "active" technologies? How are interfaces to existing "passive" entities constructed for the purposes of interfacing to the network?

- Management of controllable objects is a topic that has been explored only superficially. It differs from the management of passive entities in that controllable objects can be involved in management functions. This allows for the implementation of management capabilities that far surpass simple monitoring. For example, centralized policies and guidelines can be chosen dynamically for the network and effected at each controllable object. The evolution and exploitation of standard networking hardware towards a high degree of SNMP [41] controllability is indicative of a fraction of convenience that may be realized by software controllability. In general, software applications are more complex than hardware applications, and thus offer controllable parameters. Thus, controllable software represents even greater opportunity for management convenience.

- Resource discovery and network adaptation rely on the development of mechanisms by which network entities can collaborate to find one another, or to optimize performance. Although both of these issues have been considered in the context of data-driven infrastructures, their implementation to exploit the capabilities of a control-based framework has not yet been addressed.

- The recent development of many "active" technologies seems to reflect the need for control in many software and hardware arenas. However, the perceived motivation, in many cases, has not been described as a need for control, but rather as a need for something else, such as for centralized security and accountability [5] or as for device independence [1]. A belief underlying the Oasis infrastructure is that control is an enabling technology for such goals, and that these goals can all be easily expressed in terms of control. Future work within the control oriented frameworks will determine whether or not control is a convenient technology for expressing "active" requirements.

- Lastly, the interfacing of programs and devices to a network of controllable entities is an area that merits further investigation. The development of control interfaces to legacy applications and devices still relies greatly upon the intuition of the interface designer. By contrast, it is quite easy for the original developer of an application to design around control-oriented communication boundaries, leading to a trivial control-oriented decomposition of the application into its components. Further examination of this issue may yield more mechanized ways to interface to older, data-driven applications.

Bibliography

[1] ADOBE, INC. *Postscript Language Reference Manual*. No. ISBN 0-201-18127-4. Addison-Wesley Publishing Company, December 1990.

[2] ANONYMIZER, INC. The handy anonymizer guide. *http://www.anonymizer.com/guide.shtml*, 1998.

[3] APACHE PROJECT. Apache HTTP server project. *http://www.apache.org/*, 1998.

[4] ATHAN, A., AND DUCHAMP, D. Agent-mediated message passing for constrained environments. In *Proceedings of the Mobile and Location-Independent Computing Symposium* (August 1993), USENIX.

[5] AT&T LABS. The Geoplex project. *http://www.geoplex.com/*, 1998.

[6] BADRINATH, B. R., BAKRE, A., IMIELINSKI, T., AND MARANTZ, R. Handling mobile clients: A case for indirect interaction. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems* (Napa, October 1993), IEEE, pp. 91–97.

[7] BASS, L., CLEMENTS, P., AND KAZMAN, R. *Software Architecture in Practice*. No. ISBN 0-201-19930-0. Addison-Wesley Publishing Company, January 1998, ch. 18 – The Meteorological Anchor Desk System: A Case Study in Building a Web-Based System from Off-the-Shelf Components.

[8] BENNETT, F., RICHARDSON, T., AND HARTER, A. Teleporting – making applications mobile. In *Proceedings of the Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, December 1994), IEEE, pp. 82–84.

[9] BROOKS, C., MAZER, M., MEEKS, S., AND MILLER, J. Application-specific proxy servers as http stream transducers. In *World Wide Web Journal: 4th International World Wide Web Conference* (December 1995).

[10] CAMPIONE, M., AND WALRATH, K. *The Java Tutorial: Object-Oriented Programming for the Internet*, 2nd ed. No. ISBN 0-201-31007-4. Addison-Wesley Publishing Company, 1998, ch. 8: Overview of Applets.

[11] COMPUSERVE, INC. Graphics interchange format(sm), version 89a(modified). Tech. rep., Compuserve, 1990. Compuserve filename: GIF89M.TXT.

[12] DEVARAKONDA, M., MUKHERJEE, A., AND KISH, W. Meta-scripts as a mechanism for complex web services. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems* (Orcas Island, WA, May 1995), IEEE.

[13] DUCHAMP, D. Issues in wireless mobile computing. In *Proceedings of the Third Workshop on Workstation Operating Systems* (Key Biscayne, FL, April 1992), IEEE.

[14] FORDE, C. Publishing on the infobahn. *http://www.bcsc.gov.bc.ca/ cforde/xplor95.html*, February 1995.

[15] FOX, A., GRIBBLE, S. D., BREWER, E. A., AND AMIR, E. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), ACM.

[16] GENERAL MAGIC. Odyssey frequently asked questions. *http://www.genmagic.com/technology/odyssey_faq.html*, 1998.

[17] GOLDBERG, D., AND TSO, M. How to program networked portable computers. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems* (Napa, CA, October 1993), IEEE, pp. 30–33.

[18] HOKIMOTO, A., KURIHARA, K., AND NAKAJIMA, T. An approach for constructing mobile appplications using service proxies. In *Proceedings of the 16th International Conferencce on Distributed Computing Systems*.

[19] HOUSEL, B. C., AND LINDQUIST, D. B. WebExpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking* (November 1996), ACM.

[20] J. PURDY, ED. Oracle in motion: Managing database applications remotely. *Mobile Letter* (January 1995).

[21] JAVASOFT. The java development kit (JDK). *http://java.sun.com/products/jdk/*, 1998.

[22] JOSEPH, A. D., TAUBER, J. A., AND KAASHOEK., M. F. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing* (March 1997).

[23] KAASHOEK, M. F., PINCKNEY, T., AND TAUBER, J. A. Dynamic documents: Mobile wireless access to the WWW. In *Proceedings of the Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, December 1994), IEEE, pp. 179–184.

[24] KEHOE, C., AND PITKOW, J. E. GVU's seventh WWW user survey. Tech. rep., Georgia Institute of Technology, June 1997.

[25] KLEINE, A. TYA archive. *http://www.dragon1.net/software/tya/*, 1998.

[26] LANDAY, J. A., AND KAUFMANN, T. User interface issues in mobile computing. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems* (Napa, CA, October 1993), IEEE, pp. 40–47.

[27] LANGE, D., AND OSHIMA, M. *Programming and Deploying Mobile Agents With Java Aglets*. No. ISBN 0-201-32582-9. Addison-Wesley Publishing Company, August 1998.

[28] LEI, H., AND DUCHAMP, D. An analytical approach to file prefetching. In *Proceedings of the 1997 USENIX Annual Technical Conference* (Anaheim, CA, January 1997).

[29] W3 CONSORTIUM. WWW names and addresses, URIs, URLs, URNs. *http://www.w3.org//hypertext/WWW/Addressing/Addressing.html*, November 1994.

[30] W3 CONSORTIUM. HyperText Markup Language (HTML). *http://www.w3.org/hypertext/WWW/MarkUp/MarkUp.html*, March 1995.

[31] Microsoft internet explorer. *http://www.microsoft.com/windows/ie/*, 1998.

[32] NETSCAPE. Netscape communicator and netscape navigator news. *http://home.netscape.com/browsers/index.html*, 1998.

[33] NOBLE, B. D. *Mobile Data Access*. PhD thesis, Carnegie Mellon University, 1998.

[34] NOBLE, B. D., PRICE, M., AND SATYANARAYANAN, M. A programming interface for application-aware adaptation in mobile computing. In *Proceedings of the Second Symposium on Mobile and Location-Independent Computing* (April 1995), USENIX.

[35] OBJECT MANAGEMENT GROUP AND X/OPEN. *The Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1, Revision 1.1.

[36] OCKERBLOOM, J. Introducing structured data types into internet-scale information systems. Ph.D. thesis proposal, Carnegie Mellon University, May 1993. *http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/user/spok/www/proposal.html*.

[37] OCKERBLOOM, J. *Mediating Among Diverse Data Formats*. PhD thesis, Carnegie Mellon University, 1998.

[38] ORACLE. Oracle in motion. Technical product summary, Oracle, September 1994.

[39] OUSTERHOUT, J. Tcl: An embeddable command language. In *Proceedings of the Winter 1990 Usenix Conference* (Washington, DC, January 1990), Usenix, pp. 133–146.

[40] ROMKEY, J. A nonstandard for transmission of IP datagrams over serial lines: SLIP. RFC 1055, Network Working Group, June 1988.

[41] ROSE, M. T. *The Simple Book: An Introduction to Networking Management*, 2nd ed. No. ISBN 0-134-51659-1. Prentice-Hall Press, 1996. April.

[42] SATYANARAYANAN, M. Mobile computing: Past, present, & future. In *Proceedings of the IBM Workshop on Mobile Computing* (Austin, TX, January 1994), IBM.

[43] SATYANARAYANAN, M., KISTLER, J. J., MUMMERT, L. B., EBLING, M. R., KUMAR, P., AND LU, Q. Experience with disconnected operation in a mobile environment. In *Proceedings of the Usenix Mobile and Location-Independent Computing Symposium* (Cambridge, MA, August 1993), Usenix.

[44] SCHILIT, B. N., ADAMS, N., GOLD, R., TSO, M. M., AND WANT, R. The PARCtab mobile computing system. In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems* (Napa, CA, Oct 1993), IEEE.

[45] SCHILIT, B. N., THEIMER, M. M., AND WELCH, B. B. Customizing mobile applications. In *Proceedings of the Usenix Mobile and Location-Independent Computing Symposium* (Cambridge, MA, August 1993), Usenix, pp. 129–138.

[46] SIMPSON, W. The point-to-point protocol (PPP). RFC 1661, Network Working Group, July 1994.

[47] SOLEY, R. M. *The Common Object Request Broker: Architecture and Specification*, second ed. Object Management Group and X/Open, 1992. OMG Document Number 92.11.1, Revision 2.0.

[48] SOMMERER, A. The java archive (jar) file format. *http://www.javasoft.com/docs/books/tutorial/jar/index.html*, 1998.

[49] SUNSOFT. The HOTJAVA browser: A white paper. *http://java.sun.com/1.0alpha2/doc/overview/hotjava/index.html*, 1995.

[50] SUNSOFT. The java language: A white paper. *http://java.sun.com/1.0alpha2/doc/overview/java/index.html*, 1995.

[51] TRUMAN, T., ET AL. Infopad: A system design for portable multimedia access. In *Proceedings of the Seventh Annual International Conference on Wireless Communications* (Calgary, Canada, 1994).

[52] US CENSUS BUREAU. TIGER mapping service: The "coast to coast" digital map database. *http://tiger.census.gov/instruct.html*, 1998.

[53] US CENSUS BUREAU. U.S. gazetteer. *http://www.census.gov/cgi-bin/gazetteer*, 1998.

[54] VOELKER, G. M., AND BERSHAD, B. N. Mobisaic: An information system for a mobile wireless computing environment. In *Proceedings of the Workshop on Mobile Computing Systems and Applications* (Santa Cruz, CA, December 1994), IEEE, pp. 185–190.

[55] WATSON, T. Effective wireless communication through application partitioning. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems* (Orcas Island, WA, May 1995), IEEE.

[56] WATSON, T. Wit: An infrastructure for wireless palmtop computing. Ph.D. qualifier paper, University of Washington, 1995.

[57] WETHERALL, D. J., GUTTAG, J. V., AND TENNENHOUSE, D. L. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of the First IEEE Conference on Open Architectures and Network Programming* (San Francisco, CA, April 1998), IEEE.

[58] WHITE, J. E. Telescript technology: Scenes from the electronic marketplace. General magic white paper, General Magic, 1994.

[59] WHITE, J. E. Telescript technology: The foundation for the electronic marketplace. General magic white paper, General Magic, 1994.

[60] ZENEL, B. *A Proxy Based Filtering Mechanism for the Mobile Environment*. PhD thesis, Columbia University, 1998.