Improving the Deployability of Diamond

Adam Wolbach

CMU-CS-08-158
September 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
M. Satyanarayanan, Chair
David A. Eckhardt

*Submitted in partial fulfillment of the requirements
for the degree of Master's of Science.*

This research was supported by the National Science Foundation (NSF) under grant numbers CNS-0614679 and CNS-0509004.

Keywords: Virtual machines, mobile computing, nomadic computing, pervasive computing, transient use, VirtualBox, VMM, performance, Diamond, OpenDiamond®, discard-based search, content-based search, metadata scoping

To my family

Abstract

This document describes three engineering contributions made to Diamond, a system for discard-based search, to improve its portability and maintainability, and add new functionality. First, core engineering work on Diamond's RPC and content management subsystems improves the system's maintainability. Secondly, a new mechanism supports "scoping" a Diamond search through the use of external metadata sources. Scoping selects a subset of objects to perform content-based search on by executing a query on an external metadata source related to the data. After query execution, the scope is set for all subsequent searches performed by Diamond applications. The final contribution is *Kimberley*, a system that enables mobile application use by leveraging virtual machine technology. Kimberley separates application state from a base virtual machine by differencing the VM before and after application customization. The much smaller application state can be carried with the user and quickly applied in a mobile setting to provision infrastructure hardware. Experiments confirm that the startup and teardown delays experienced by a Kimberley user are acceptable for mobile usage scenarios.

Acknowledgments

First, I would like to thank Satya. Satya has been a great advisor and mentor and has guided my academic career from its conception four years ago to where it is today. He always found time to meet with me despite a busy schedule, and could always be counted on for helpful, prompt feedback. I take great pride in having worked with a researcher of his caliber and deeply appreciate all of the effort he has dedicated to me. Thank you, Satya.

I'd like to thank Dave Eckhardt for serving as the second half of my thesis committee and reading and reviewing my document during a very hectic time of the year. I will always owe my interest in computing systems to spending two fantastic semesters in Operating Systems class with Dave. He also provided many insights and suggestions during the course of my graduate study, and always lent an ear to any problem I had.

The past and present members of Satya's research group provided an incalculable amount of help to me: Rajesh Balan, Jason Ganetsky, Benjamin Gilbert, Adam Goode, Jan Harkes, Shiva Kaul, Niraj Tolia, and Matt Toups. They are an extremely talented group of people and it has been a pleasure working with them. Two names bear special mention: Jan has mentored me from the very start through several projects and thanks to his supreme patience I am here today; and Adam has tirelessly provided help since the onset of my work in Diamond, and was available for consultation at all hours without fail. I'd also like to thank Tracy Farbacher for her hard work coordinating meetings and always ensuring things ran smoothly.

I also owe thanks to the Diamond researchers at Intel who took time out of their schedules to meet with me and provided invaluable advice and feedback, including Mei Chen, Richard Gass, Lily Mummert, and Rahul Sukthankar.

My good friends and colleagues, Ajay Surie, Cinar Sahin, and Zhi Qiao were always there to lend a shoulder when I needed help and motivate me when I struggled. I owe thanks to more friends than I can name here for providing entertainment along the way. Two special groups bear mentioning: my Carnegie Mellon friends, including Pete Beutler, Nik Bonaddio, Jeff Bourke, George Brown, Christian D'Andrea, Hugh Dunn, Patrick Fisher, Dana Irrer, Veronique Lee, Anu Melville, Laura Moussa, Jen Perreira, Ben Tilton, Eric Vanderson, Justin Weisz, and the Kappa Delta Rho fraternity; and my friends from home, including Austin Bleam, Nick Dower, Nick Friday, Matt Givler, Roland Kern, Ray Keshel, Justin Metzger, Curt Schillinger, Michael Way, and Ryan Wukitsch. They have always been there to provide a distraction when I needed it.

On a lighter note, I would like to acknowledge Matthew Broderick for his role in *Ferris Bueller's Day Off*, the Beastie Boys for recording *Paul's Boutique*, and Gregg Easterbrook for regularly penning the column *Tuesday Morning Quarterback*.

Finally, I'd like to thank my family, especially my parents, Carla and Donald, and my grandparents Ruth and Carl Rohrbach and Fay and Donald Wolbach, for their unending emotional support. Without it, I would not be the person I am today.

Adam Wolbach
Pittsburgh, Pennsylvania
September 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Enabling users to interactively explore large sets of unindexed, complex data presents a diverse set of challenges to a system designer. A successful system must cleanly separate domain-specific resources from a domain-independent framework or risk compromising its flexibility. It must also scale with ever-increasing dataset sizes. *Diamond* is a system designed since 2002 at Carnegie Mellon University and the Intel Corporation with these goals in mind.

Diamond takes a *discard-based* approach that rejects irrelevant objects as close to the data source as possible. It executes searches on content servers that operate independently, exploiting object-level parallelism and thus allowing the system to scale linearly. It provides a facility to fine-tune the load balance of search execution between client and server, maximizing performance. In addition, it provides a clear separation of these search mechanisms from searchable data through the *OpenDiamond*® Platform, a library to which all Diamond applications link. The OpenDiamond library standardizes the search interface by presenting Diamond applications with a general search API. Many Diamond applications have been built on top of OpenDiamond in a variety of domains, including generic photographic images (Snapfind), adipocyte cell microscopy images (FatFind), mammography images (MassFind), and a general interactive search-assisted diagnosis tool for medical case files (ISAD).

Diamond's initial, most difficult goal was to validate its discard-based approach, and the majority of early development worked towards that goal. Placeholder implementations were added for non-critical portions of the design with the expectation of later improvement. By late 2006, sufficient empirical evidence had been gathered to validate the concept of discard-based search, and many potential improvements to the system increased in priority. For instance, in many domains such as healthcare, large metadata repositories with indices annotate the data, providing an opportunity to better target a Diamond search and narrow its scope. Diamond also demanded at least laptop-class hardware for clients and could not operate satisfactorily on PDAs, smartphones, or other mobile devices. The focus of this work was on extending Diamond along these axes while respecting the original motivating goals of its designers.

This chapter begins with an overview of the Diamond system. Section 1.2 examines shortcomings of the Diamond system at the onset of this work. Section 1.3 describes the scope and validation of the thesis work. The chapter concludes with a document roadmap.

1.1 Diamond

Diamond is a system designed to solve the problems involved in the interactive exploration of complex, non-indexed data. It uses the concept of discard-based search as the basis of its approach to this problem. Examples of such data include large distributed repositories of digital photographs, medical images, surveillance images, speech clips or music clips. The emphasis on "interactive" is important: Diamond assumes that the most precious resource is the time and attention of the person conducting the search rather than system resources such as network bandwidth, CPU cycles or disk bandwidth. That person is assumed to be a high-value expert such as a doctor, pharmaceutical researcher, military planner, or law enforcement official, rather than a mass-market consumer.

In contrast to classic search strategies that precompute indices for all anticipated queries, discard-based search uses an on-demand computing strategy that performs content-based computation in response to a specific query. This simple change in strategy has deep consequences for flexibility and user control. Server workloads in discard-based search exhibit coarse-grained storage and CPU parallelism that is easy to exploit. Further, discard-based search can take advantage of result caching at servers. This can be viewed as a form of just-in-time indexing that is performed incrementally at run time rather than in bulk a priori. Diamond has explored this new search paradigm since late 2002, and resulted in gains of extensive experience with both software infrastructure and domain-specific applications. This experience helped to cleanly separate the domain-specific and domain-independent functionality. The latter is encapsulated into Linux middleware called the *OpenDiamond*® platform. Based on standard Internet component technologies, it is distributed in open-source form under the Eclipse Public License (http://diamond.cs.cmu.edu/). [32] If you are unfamiliar with Diamond, read [26] and [32].

1.2 Motivation

Prior to this work, Diamond ignored several potential improvements to its design in favor of achieving its critical goal: a validation of its discard-based approach. At the time this work began, empirical results of the discard-based approach had validated its importance, and a shift in priorities allowed the identification of new areas for improvements. The challenges associated with these improvements motivated and guided the work of this thesis.

The early discard-based agenda ignored metadata sources, including those with valuable prebuilt indices, that often accompanied data in many domains. As a result, the Diamond system design possessed no mechanism to execute metadata queries that could narrow the scope of discard-based search. For instance, in the healthcare industry large amounts of metadata in the form of patient records are stored with the actual medical image data. Such a metadata source provides the opportunity to improve the relevance of results by restricting the scope of a search with a query over age, gender, or any of number of types of patient information.

An additional limitation was that the Diamond system required laptop-class hardware or better for a Diamond client to execute. The majority of the work in Diamond searches was performed on content servers, with clients often only responsible for controlling searches and retrieving and displaying results to the user. Despite this the client still required more powerful hardware than what was necessary for a variety of reasons. The shared processing of Diamond searches between a client and a server under heavy load imposed the requirement of a shared processor architecture between machines. A Diamond client also failed to perform adequately when subjected to severe limitations, such as a handheld device with a strained network connection which impacted search result retrieval, poor compute power which limited client-side search execution, limited memory for holding search results, or poor display capabilities for rendering results to the user. These limitations excluded the use of entire classes of less powerful devices as Diamond clients and ruled out many attractive usage scenarios.

1.3 The Thesis

When this work commenced, Diamond existed as a working system with a clear separation between domain-independent and domain-specific components. The thesis work aimed to improve the deployability of Diamond while preserving this essential property. With respect to scoping Diamond searches with metadata queries, the new system enables Diamond to use external metadata sources while imposing few restrictions on those sources themselves. With respect to executing Diamond searches on resource-limited clients, a new system, *Kimberley*, enables Diamond execution on new classes of less powerful hardware. Kimberley is a system with its origin in Diamond but with a general design that allows the execution of applications on mobile clients, exploiting virtual machine technology and the possibility of powerful machines available in local infrastructure. The thesis statement is thus:

> *The versatility and deployability of Diamond can be significantly improved by the ability to incorporate indexed metadata sources into Diamond searches and by the ability to execute Diamond searches from resource-limited mobile clients. These improvements in functionality can be achieved while preserving clear separation between domain-independent and domain-specific functionality in discard-based search.*

1.3.1 Scope of Thesis

This thesis focuses on improving the deployability of the Diamond system. Its techniques are domain-agnostic, and while inspired by certain Diamond applications, they do not attempt to address the shortcomings of any specific one. These techniques thus add to the core functionality of the Diamond system. This thesis makes the following assumptions about the Diamond system and future deployments in a variety of domains:

- In many situations, indexed metadata sources exist which richly describe and annotate the content searched by Diamond. Incorporating these metadata sources into Diamond to reduce the size of the search set can improve the relevance of a search. The querying of metadata sources for this purpose is called "scoping" a search.
- Along with scoping functionality, the a Diamond realm provides authentication, access control, query authorization, auditing, and data location services.
- Executing Diamond searches on weak handheld devices is possible by exploiting superior hardware available in the local infrastructure in mobile environments.

1.3.2 Validation of Thesis

This thesis was investigated through the design, implementation and test of new mechanisms in the Diamond system. These mechanisms then became official releases of the

Diamond system. Diamond has been under development since 2002, involving a collaboration between Carnegie Mellon University, the Intel Corporation, the Merck Corporation, the University of Pittsburgh, and the University of Pittsburgh Medical Center, and provides a large user community from which to draw ideas and personal experiences. Empirical measurements and controlled experiments were conducted to validate the thesis statement.

1.4 Document Roadmap

The remainder of the thesis consists of five chapters. Chapter 2 describes early engineering improvements to Carnegie Mellon's implementation of Diamond. Chapter 3 describes a new mechanism to incorporate querying indexed metadata sources to scope a Diamond search, including both proof of concept and complete implementations. Chapter 4 describes the *Kimberley* system designed to support use of Diamond in mobile environments by utilizing infrastructure resources. Empirical results are presented at the conclusions of chapters 3 and 4. Chapter 5 discusses work related to this thesis. The document concludes with a discussion of the contributions of this thesis and future work in Chapter 6.

Chapter 2

Reengineering Diamond

By late 2006, the Diamond system had been under development for four years. Its code base evolved contemporaneously with the system design, and several non-critical sections of the system design relied on incomplete, placeholder implementations. This was a conscious choice of the developers with the expectation that a later redesign of these components, combined with additional user and developer experience, might lead to an improved overall design.

One instance was in the implementation of an asynchronous networking library. The library gave developers a large amount of flexibility in the network topology but provided little support in terms of marshalling data, use of generated stub functions, or external data representation formats, and its asynchronous nature increased the difficulty of error reporting. This was the correct decision early in the design because it prevented the client from blocking while waiting for search results from a server, a risk due to unbounded searchlet execution time during any given search. The key benefit was a high level of interactivity for the user, which fit the envisioned Diamond usage model of rapid search refinement. However, many of the server calls made by Diamond clients contained small amounts of data or served as control signals for which an asynchronous approach was unnecessary and complicated the programming model. The addition of new types of servers into the system architecture in early 2007, covered in Chapter 3.3, prompted a revisiting of the original networking design choices.

Another example was in the administrative management and distribution of data in the Diamond system. When originally visioned, it was thought Diamond might rely on "active disk" technology, [**?**] executing searchlets on disks themselves to minimize the bottleneck of I/O bandwidth. After four years of experience it was found that the most useful model for many Diamond applications was to simply make a subset of each collection of data objects available in the local filesystem of the content servers tasked with searching those images. This unfortunately involved the tedious maintenance of many configuration files on both client and server that obstructed the deployment of new collections, and made changes to existing collections difficult. No administrative tools were available to document or manage content.

This chapter provides an overview of two opportunities to improve the engineering of Diamond as it existed in late 2006. The following section describes improvements made to the communications layer of Diamond. Section 2.2 describes a tool created to ease content management. The chapter concludes with a summary.

2.1 Diamond's Communications Subsystem

2.1.1 Background

Diamond's communications system existed in 2006 as an asynchronous message passing library. A client issued a call to a content server to supply some information or send a signal, such as supplying a searchlet or starting a new search. If a response was needed, the content server would make a callback when the corresponding operation was complete. From the client perspective, an asynchronous approach maximized interactivity by avoiding long, blocking calls such as those to execute searches and return results. On content servers, it allowed search results to be sent to clients as they became available, without delay. To avoid the situation where large object results might congest a network connection between client and content server, the library split communications across two network connections: a control connection carried search parameters and control signals from client to server; and a data connection carried object results returned from a content server to a client. The data connection was called the "blast channel" as its sole purpose was to pass results back as soon as they became available. The use of separate control and data network connections remains in place today.

This approach achieved good performance, but also resulted in several drawbacks. For one, the library provided few of the features of well-known Remote Procedure Call systems. Each call needed specialized code to convert its arguments into network byte order and then marshall them into a custom Diamond protocol packet. Then, the library sent the packet along a TCP connection, which required the application constantly monitor the connection to ensure the kernel's send buffer did not overfill and cause the system to block or drop data. On the far end, the receiving library also needed specialized code to perform the corresponding inverse operations. This sequence created a large overhead for developers wishing to create new network calls. The overhead involved became evident in the source over time, with developers opting to use generic calls such as "device_set_blob" to pass an arbitrary blob of data to a content server rather than creating a new, separate library call. This reduced the portability of the system by exposing the platform dependencies of many data types and structures. The asynchronous approach also confounded error reporting, since if a call failed, a callback from content server to client needed to be made to alert the client. Finally, the library's code base became difficult to maintain once its primary developer left.

2.1.2 Design

The goal of reengineering Diamond's communications library was to improve its maintainability, reliability, and ease of use with minimal performance cost and without causing changes at the OpenDiamond library API level. Early on in the design, the blast channel's ability to asynchronously return search results was identified as the key component that determined performance. This approach minimized user delay by avoiding blocking calls on slow servers. As a result, the new design retains a separate network connection dedicated solely to the end goal of asynchronously returning search results. However, the control channel often sent much smaller messages signalling the start and stop of a Diamond search, searchlet descriptions, names of filters to execute, and many other calls related to search control and search statistics. These calls were handled very quickly and often returned error codes or small pieces of data through client callbacks.

A well-established mechanism employed to enable a heterogeneous network of computers to communicate small messages effectively is Remote Procedure Call, or RPC. RPC systems attempt to replicate the semantics of local procedure calls as closely as possible. They generally provide some subset of the following features: user authentication; secure end-to-end communication; client and server stub generation; argument marshalling and unmarshalling; conversion to machine-independent data types; reliable data transport and exception handling; and the binding of clients and servers [19]. The idea of a programmatic RPC interface is over thirty years old [38] and a variety of open-source implementations exist today for many different high-level programming languages. Among the most popular are Sun Microsystems' RPC (now called Open Network Computing/ONC RPC) [27], ONC+ RPC/TI-RPC [13], DCE/RPC [3], ICE [5], SOAP [10] and the related XML-RPC [16], Java Remote Method Invocation (RMI) [6], CORBA [1], and the most recent addition, Facebook's Thrift [35].

A survey of existing RPC mechanisms was conducted with several constraints. Diamond is released under the Eclipse Public License (EPL) version 1.0 and therefore is incompatible with all versions of the GNU General Public License, eliminating a large number of open source RPC systems. The EPL requires a more lenient license such as the GNU Lesser General Public License or the BSD family of licenses. Further, Diamond is developed with the C programming language and an effort was made to avoid object-based RPC systems such as SOAP, CORBA, and Java RMI which did not match the programming model well. The most applicable systems were ONC RPC, ONC+ RPC and XML-RPC, all providing a compatible license, client and server stub generation and a C interface.

XML-RPC over the BEEP Core protocol [25], supplied through the Vortex library [15] developed by Advanced Software Production Line, initially appeared to be the most attractive fit. It encodes data into a standard XML format and uses the BEEP Core protocol to transfer data between machines. BEEP itself provides a framework to developers that allows application protocols to be layered on asynchronous communications mechanisms with transport layer security (TLS), peer authentication, multiplexing of connections and many other features. However, a closer analysis of the library's source revealed a rapid pace of development and the lack of a complete implementation suggested the system was not ready for production-level deployments.

The decision then fell to ONC RPC and ONC+ RPC, which provided the best possible feature sets to developers. They used the XDR External Data Representation Standard [20] for platform-independent data transfer and provided stub generation, argument marshalling, error handling and were licensed under the GNU Lesser General Public License (LGPL). The key difference between the two RPC systems was that ONC+ RPC extended ONC RPC by separating the transport layer. This allowed developers to associate new transport protocols with the other RPC mechanisms, including traditional TCP and UDP protocols with IPv6 support, and earned the library the additional name Transport-Independent RPC (TI-RPC). However, Linux support for ONC+ RPC was found to be incomplete. In contrast, ONC RPC, which was called Transport-Specific RPC (TS-RPC) in response to TI-RPC, had been under development for over two decades as Sun RPC. It had established a place in the standard C library and was not likely to change drastically. In addition, over time its widespread use had generated volumes of documentation and a vast knowledge base.

The connection pairing mechanism of the original Diamond communications library remains untouched by the incorporation of RPC, since the need for a separate blast channel remains. Content servers accept control connections and blast channel connections on separate well-known ports. They pair connections from the same client application using a nonce generated on the server, which is sent to the client along the control connection and returns on the data connection. The servers pair the connections by forking a child process which obtains exclusive control of the connections. In the new design, the child server transforms the control connection into an ONC RPC server instead of managing it itself. All calls into the transport mechanism occur through the same transport library API as before.

2.1.3 Implementation

ONC RPC was fitted to the OpenDiamond library by replacing the ad-hoc set of client and server procedures with a protocol specified in Sun's standard interface definition language XDR, an acronym for eXternal Data Representation. From this protocol, the ONC RPC stub generator *rpcgen* created client and server stub procedures, as well as network formatting and memory management procedures for various XDR data types. The procedure stubs received the same arguments as the corresponding calls from the previous implementation, but removed the burdens of marshalling data, managing connections at the socket level, and handling network errors.

After defining the client-server RPC interface, initialization code was introduced to transform the connected sockets into corresponding ONC RPC clients and servers. The pairing protocol executed when a client first contacts a server meant the ONC RPC client and server interfaces must use sockets already bound and connected. The client-side implementation of ONC RPC fit this model well, but the GNU libc server-side implementation of ONC RPC did not provide a facility for creating a new server on top of an already-connected socket. As a result, when a pair of connections are made to a content server, a separate thread must be forked to create an ONC RPC server. It is created on a random port known to both child and parent threads, and listens only for connections coming from the local host. This measure was taken to prevent an external attacker intercepting the setup of a control connection. The parent thread waits briefly after the fork for an indication that the server is running, and then connects on the port. From that point it serves only as a tunnel, forwarding bytes in both directions between the external connection and the ONC RPC server.

This implementation artifact introduces an extra hop in the control connection between the client Diamond application and the content server. However, any additional latency in the connection was not visible in actual use. In fact, users noted no visible difference in performance between the original asynchronous communications library and the replacement ONC RPC implementation. In addition, error reporting became more standardized with the new implementation, assisting in the discovery of many hidden bugs. Due to the foresight of the original developer, this addition to the system only required changes beneath the OpenDiamond API layer. As a result, an upgrade required no changes to the

Diamond applications themselves.

2.1.4 Validation

The introduction of ONC RPC as the transport layer for Diamond was officially released in August 2007 as OpenDiamond 3.0. The result of switching from the original to ONC RPC was a net removal of 653 lines of code (when compared between OpenDiamond version 3.0.2 and its previous version 2.1.0). Since the release of version 3.0.2, many new RPC calls have been easily added to the system, confirming the engineering benefits of this change.

2.2 Content Management

2.2.1 Background

The content management system in Diamond as it existed in late 2006 relied on the concept of object groups each spread across a number of content servers, identified by a unique 64-bit identifier (often represented in hexadecimal). To ease the identification of specific groups by a user, an additional mapping of collection names, represented by ASCII strings, to group identifiers was added on the client-side. Collections known to the Diamond client applications were listed in a drop-down box in the clients' GUIs, from which the user would select a subset of them to search. The group identifiers also did not include any information on which content servers owned those groups, so an additional mapping of group identifier to content server hostname existed on each client. Both maps existed as configuration files stored in a user's home directory.

Content servers received a list of group identifiers from a client prior to executing a series of Diamond searches. This established the scope of a Diamond search at a very coarse granularity. When a new search request was received from a client, the content server attempted to open an index file specific to the group identifier which contained pathnames of objects in the group, relative to a data directory specified in a Diamond configuration file. These pathnames were fed one at a time to the search threads to execute a Diamond search.

The design choices made with respect to object groups allowed Diamond users to easily refer to large collections of objects used in system demonstrations and when obtaining experimental results. The key tradeoff in gaining this ease of specificity was the loss of the ability to select subsets of objects based on additional object data. Another shortcoming was that GUI support to select collections had to be provided by each Diamond application. Further, each user's configuration files fell out of date whenever new collections were added or existing collections redistributed to new content servers, adding to the difficulty of content management in Diamond.

2.2.2 Detailed Design and Implementation

In response to the problems experienced with content management on Diamond clients and content servers, a new utility called *volcano* was created. Volcano provided the ability for Diamond administrators to distribute a new collection of objects to the servers without tediously creating each configuration file by hand. The manual page for volcano can be found in Appendix A.

On the command line, volcano takes a new collection's user-visible name, the pathname of an index file containing object locations, and one or more content server hostnames to distribute the new collection amongst. The index file contains a list of pathnames of files or directories, one per line. A file is taken to be a single object, while a directory is descended into recursively to enumerate all file descendants as objects. This allows the administrator to specify a large number of objects very compactly. The objects are then distributed in a round-robin fashion amongst the content servers listed. Each object is hashed into, and named by, a 64-bit number using the object's data, which allows the detection of (but does not prevent) duplicate objects.

The result of a successful volcano execution is the creation of a series of files. For each content server listed, a new group index file is created, containing relative pathnames of new objects to be distributed to that server. The group identifier used is a random 64-bit number. In addition, volcano creates new mappings from collection name to group identifier, and group identifier to content server hostnames, which an administrator can then distribute to users. The final output of volcano is a generic shell script which contains the actual distribution commands. This is done to allow the administrator to review the actions to be performed before a possibly lengthy content distribution period. When they are satisfied with the commands, they simply execute the generated script.

2.2.3 Validation

Volcano was developed in early 2007 and became an official part of OpenDiamond when version 3.1 was released. It has been used successfully by system administrators at CMU, Intel and IBM to distribute new collections of objects across content servers.

2.3 Chapter Summary

ONC RPC replaced Diamond's original ad hoc communication mechanism. This serves to improve the maintainability of the Diamond communications subsystem, as well as improve portability, error reporting and decrease the complexity of the code. Volcano is a utility created to assist system administrators in distributing and managing new collections of objects across many content servers, by removing the overhead related to configuration file management. Both of these improvements were included as part of separate official releases of the OpenDiamond system.

Chapter 3

Using Metadata Sources to Scope Diamond Searches

In many potential deployments of Diamond, rich metadata sources accompany the data on which Diamond performs content-based search. For example, in a clinical setting large patient-record database systems often store not only the raw data produced by laboratory equipment but also the patient's relevant personal information, the date and time, the name of the attending physician, the primary and differential diagnoses, and many other possible fields. The use of prebuilt indices built on this metadata provides an opportunity to select a much more relevant subset of objects to search. The improvement in search quality may be substantial.

Early Diamond research focused on discard-based search and treated index-based search as a solved problem. Hence, the original Diamond architecture ignored external metadata sources. At the beginning of this work the feasibility of discard-based search had been established, and attention could now be paid to this aspect of search. The content servers of the previous system treated all object data opaquely. In this way they exploited object-level independence and provided a parallel architecture that scaled linearly with the number of content servers available. The introduction of external metadata sources into this Diamond architecture allows searches to discard irrelevant objects prior to the execution of discard-based search. This approach preserves the core Diamond search strategy while leveraging the well-known benefits of indexed search.

In this chapter, improvements to the Diamond system are presented which enable the use of external metadata sources to define the scope of a Diamond search. The improvements do not affect the core Diamond search protocol itself, instead introducing several new types of servers to perform the scoping operation. The following section describes usage scenarios possible in the new system. Section 3.2 describes design goals of this system. Section 3.3 outlines changes to the core architecture with the addition of new servers and describes their purposes within a Diamond realm. The following section describes high-level changes to the system architecture. Sections 3.4 and 3.5 show two separate designs and implementations of metadata scoping in Diamond: a preliminary proof-of-concept that provided the same functionality as the previous system; and a fully-featured final version. A short evaluation and discussion follow in Sections 3.6 and 3.7. The chapter concludes with a summary.

3.1 Usage Scenarios

Two hypothetical examples below illustrate the kinds of usage scenarios envisioned for metadata scoping in Diamond.

Scenario 1: *Dr. Jones is sitting at his desktop computer attempting to identify points of interest in a pathology slide of one of his patients. To aid his diagnosis, he opts to use Diamond to search a dataset of previously identified images and find comparable examples. Knowing that the age of a patient is a risk factor for possible diseases, he decides to restrict the scope of his Diamond searches to patient images in a certain age range. This information is held in a patient record database available to the Diamond backend. With a web browser, Dr. Jones navigates to a Diamond webpage which presents an interface for this specific metadata source. After authenticating himself, he uses the web interface to craft and execute a query that selects a subset of objects to search. Then, he switches to his Diamond application and executes a search on the restricted set of data, which increases the relevancy of search results and improves the final diagnosis.*

Scenario 2: *Later that day, Dr. Jones is again presented with a difficult pathology slide from another patient. In this case, the likely disease is a rare pathogen and is one of only a few cases seen at this hospital. However, a clinic a thousand miles away exclusively treats the disease and owns large repositories of relevant patient data. It provides physicians access to this data by charging a monthly subscription fee and operates a Diamond backend to search the data. Dr. Jones, in addition to the small amount of data available within his hospital's backend, wishes to include the clinic's dataset in his Diamond search. He navigates to a Diamond webpage interfacing to the foreign metadata source. He crafts a query and submits it to his hospital's Diamond backend, which forwards the query to the clinic on his behalf. The clinic's Diamond backend executes the query over its metadata which selects a subset of its objects to search. Dr. Jones then switches to his Diamond application and executes a search over a much more relevant dataset and improving the* *final diagnosis.*

3.2 Design Goals

The goal of this work is to improve the relevance of search results, by restricting the scope of a Diamond search with queries to external metadata sources. This allows the system to discard irrelevant results by taking advantage of the annotations and prebuilt indices that accompany data in many situations. It aims to accomplish this without disturbing the core Diamond search protocol or affecting the performance of Diamond's discard-based search strategy.

This work also defines a *realm* as the unit of Diamond administration. A realm may choose the most appropriate search and storage policies for its metadata sources. It may choose storage policies for the actual data as well. It provides a means of secure authentication for clients, between servers within a realm, and between separate realms. It also enables secure communication between all components of a Diamond realm. Finally, this work enables users to execute searches across multiple realms simultaneously. Realms make external business and security arrangements to recognize and trust other realms.

3.3 System Architecture

In the prior system architecture, as seen in Figure 3.1, the only components in the Diamond backend were content servers operating independently to exploit object-level parallelism. As a result, each server was maintained separately and the concept of a realm did not provide a useful abstraction for Diamond system administrators. Content servers did not use privacy or security mechanisms or allow access to external metadata sources. Additionally, the location of data on content servers generally remained static after an initial distribution, due to the overhead involved in maintaining group index files on content servers.

The new system increases the significance of the realm concept in Diamond. In the new architecture, seen in Figure 3.2, a Diamond realm consists of five different components: content server(s), scope server, authorization server, location server, and web server. One or more content servers remain as the primary execution points of Diamond searches. A *scope server* is responsible for executing queries on external metadata sources. An *authorization server* audits and approves metadata queries prior to their execution, providing a trail for system administrators. A *location server* contains the distribution of objects across content servers. Finally, a *web server* generates a dynamic web interface to the

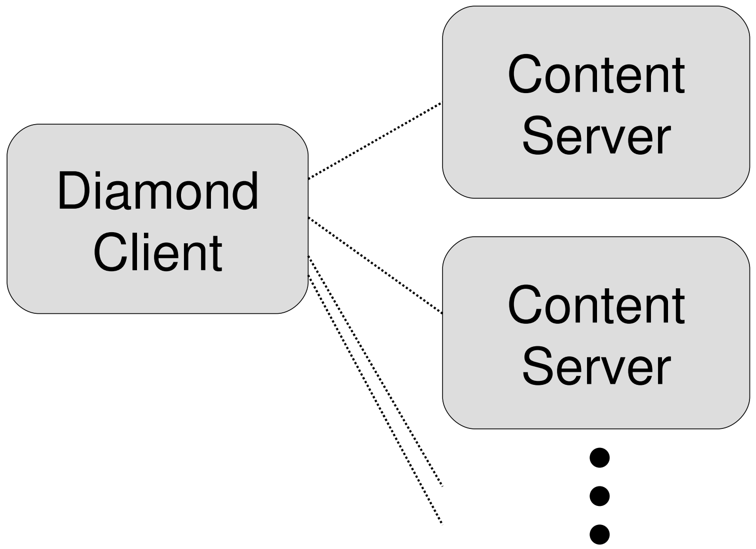


Figure 3.1: Prior Diamond System Architecture

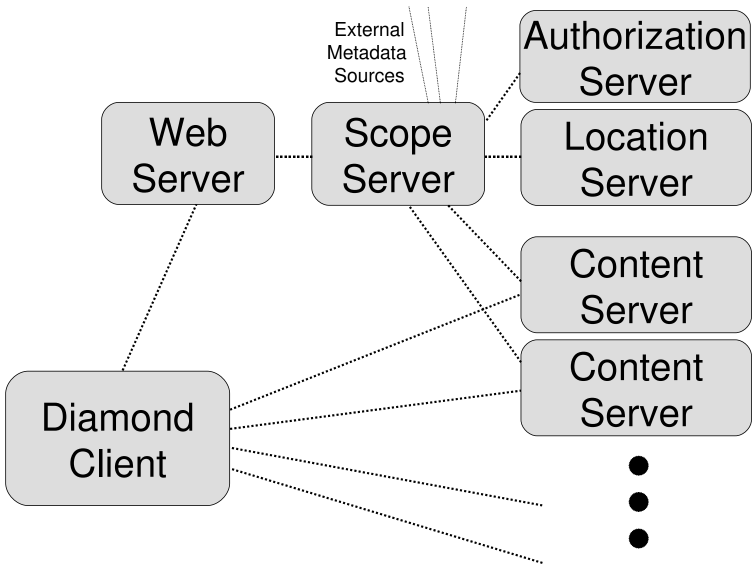


Figure 3.2: New Diamond System Architecture

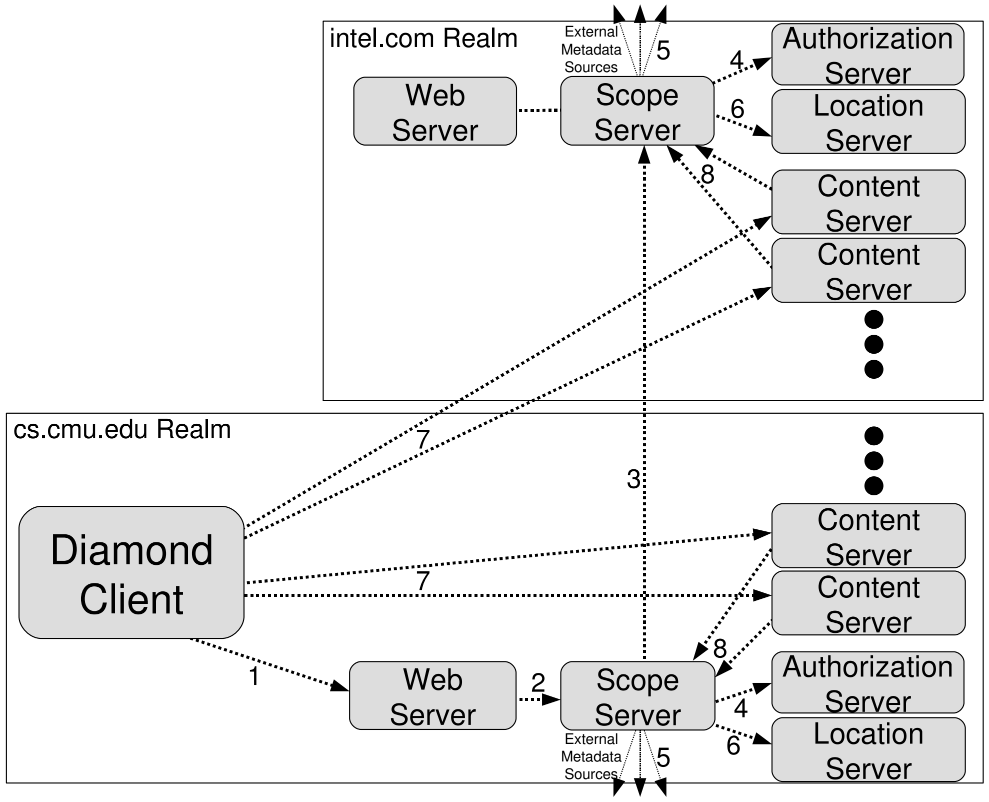


Figure 3.3: Searching Multiple Realms

scope server to provide a common user interface for all Diamond applications. While the scope, authorization, location, and web servers each represent a single logical server in a realm, the implementation of well-known failover and replication mechanisms could be used to increase failure resiliency.

3.3.1 Scope Server

A scope server provides access control and metadata scoping mechanisms for a Diamond realm. Users must authenticate to a scope server prior to defining the scope of or executing a search. The scope server controls access to metadata sources in its own realm and in foreign realms through role-based authentication, and provides a mechanism for users to enable the roles in which perform their actions. All other servers within a realm trust the realm's scope server to verify a user's identity and enabled roles.

A scope server receives metadata queries from a user, via a web interface, and executes the query on the corresponding metadata source on the user's behalf. If the metadata source exists in a foreign realm, the scope server contacts the foreign realm's scope server to forward the query. Figure 3.3 depicts this situation. The scope servers perform cross-realm authentication and through external business and security arrangements recognize and trust each other. This is represented by step 3 of the diagram. With this pairwise mutual agreement in place, scope servers may federate user identity and execute search requests in both realms. The scope servers request query execution to the authorization server in their realms, which step 4 represents. If granted, the scope servers execute the query on their metadata sources, visible as step 5.

Each scope server also then retrieves the query's result, a list of object names stored on the metadata source which satisfy the query. They locate the objects by forwarding their names to location servers, as in step 6, which match each object's name with the content server that stores its data. The returned list of object names paired with locations is called a *scope list* and is stored temporarily on disk. The scope servers return unique handles representing their scope lists to the client via the client realm's scope server. These unique handles are called *scope cookies*. A scope cookie is very small, in the range of a few kilobytes, as compared to a scope list potentially containing tens of thousands of object locations and hence megabytes or more of data. The client supplies the scope cookie to content servers as proof of the right to access its corresponding objects, seen in step 7. The content servers contact their realm's scope server to fetch the relevant portions of the scope list in step 8, and a Diamond search executes. As discussed in Section 3.5.5, a scope cookie is an X.509 certificate and thus possesses the cryptographically-based security properties of this certificate type. The activation and expiration times of the certificate are set to indicate the lifetime of the scope cookie and associated scope list. It is important to note that a user must query a metadata source even if he wishes to search all objects available, in order to generate a scope list.

3.3.2 Authorization Server

A realm's authorization server performs a more specific purpose than the scope server. It only authenticates and communicates with the scope server in its realm. It receives query requests, scrutinizes each request to see if the user is attempting to illegally access metadata resources, and grants the right to execute those queries which pass. The authorization server also audits the query, including whether it was granted, the user requesting and his roles, and other related context.

3.3.3 Location Server

The use of object names and a location server in Diamond logically separates object identification from object location by mapping an object's name to the content server hosting its data. Depending on a realm's implementation, an object identifier may be a filename, an object disk identifier, or any other unique character string. The location server then logically acts as a single point of data management within a realm. The object name or identifier is fixed at tuple insertaion and is not affected by data movement between different content servers. This provides the ability for system administrators to efficiently reorganize large numbers of objects without the tedious maintenance of configuration files distributed across many servers. As a result, the location server extends a great deal of storage flexibility to system administrators, allowing the incorporation of dynamic load-balancing strategies such as Self-* storage systems [22]. Further, only one server logically exists within a realm, but the actual implementation may use replication mechanisms to provide higher availability.

Within a realm, only the scope server may consult the location server. The location server first authenticates the scope server to prevent the loss of potentially private location information. Then, the scope server requests the location of a list of objects. The location server locates each object and pins its location for the duration of the scope. By pinning the objects, the user receives a guarantee that all relevant objects will be searched, as long as the cookie has not yet expired.

3.3.4 Web Server

The final component of the system architecture is the dynamic interface that a web server generates. The previous architecture required that each Diamond application implement an interface to select the collections in a Diamond search. Using a web server to dynamically generate this interface provides an application-independent platform for defining the scope of Diamond searches. The web application then communicates with the scope server on behalf of the user.

3.4 Preliminary Implementation: The Gatekeeper

The *Gatekeeper* was a preliminary design and implementation of metadata scoping which provided most of the major elements of the new system architecture in order to validate the use of a web-based interface. The decision was made to adapt the existing group identifier

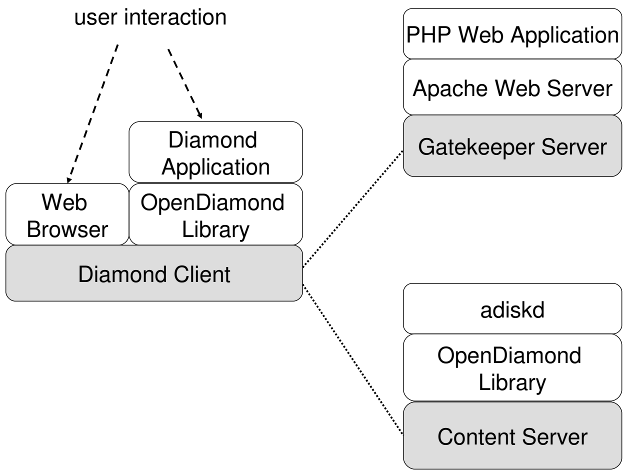
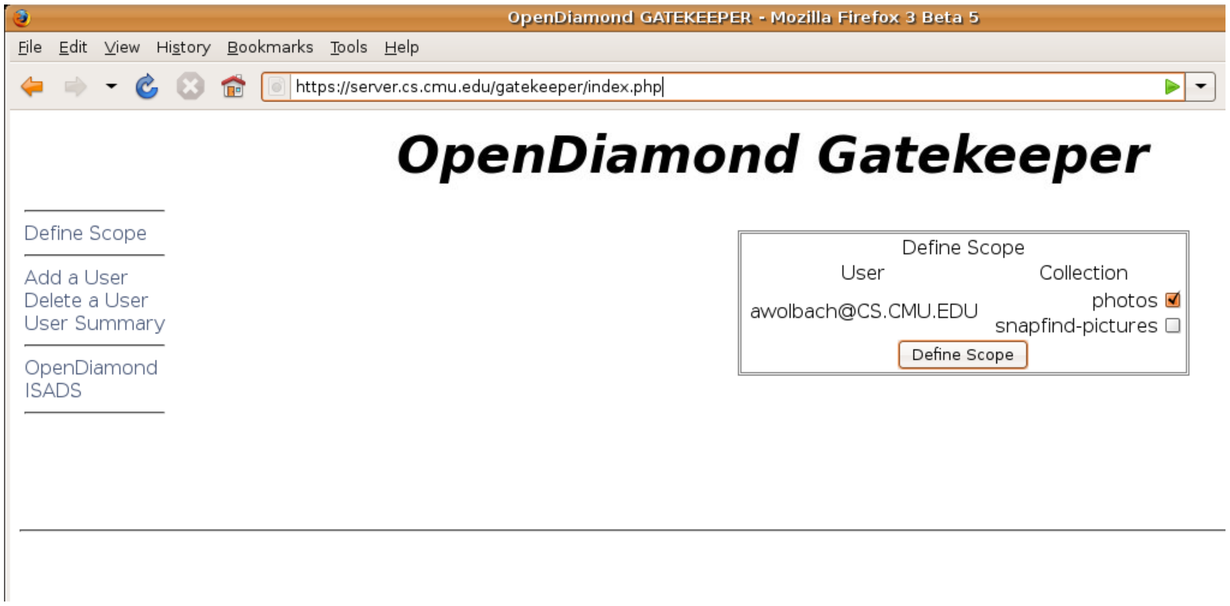


Figure 3.4: Gatekeeper Design

scoping mechanism to fit this model. This achieved the same server-side functionality that the collection/group model already provided while greatly reducing the difficulty of configuration file maintainance on the client. Its system design is presented in Figure 3.4.

The Gatekeeper introduced a web/scope server combination which contained access control, collection and group information in SQLite [11] databases stored in its local filesystem. It provided a dynamically generated web interface through the Apache 2 httpd server [4] using the mod_php module, which executed the scope server's PHP application logic. Figure 3.5 shows a screenshot of the Gatekeeper web application in action. The web interface prompted the user to authenticate through a two different mechanisms, selected through a configuration file. An administrator could choose either the built-in Apache *htaccess* username/password mechanism, or use the Apache Pubcookie module [9] which interfaces to a number of different authentication services such as Kerberos, LDAP, or NIS.

After a user authenticated, the Gatekeeper retrieved the collections he was granted access to and generated a webpage containing a checkbox for each. Figure 3.5 shows a example of a user with access to two collections. The user chose the collections he wished to search and clicked a button to submit his selection. The PHP code queried the SQLite database to find the group identifiers and hostnames associated with each collection and built the configuration files a Diamond client would need to search them. It then concatenated and returned these configuration files as one large ASCII text file to the client's web browser with a special "diamond/x-scope-file" MIME type. A MIME type


OpenDiamond GATEKEEPER - Mozilla Firefox 3 Beta 5
File Edit View History Bookmarks Tools Help
https://server.cs.cmu.edu/gatekeeper/index.php
Define Scope
Add a User
Delete a User
User Summary
OpenDiamond
ISADS
Define Scope

OpenDiamond Gatekeeper

Define Scope

User	Collection
awolbach@CS.CMU.EDU	photos ✓
	snapfind-pictures ☐

Figure 3.5: Gatekeeper Web Interface

handler installed with the OpenDiamond library moved this file into the user's Diamond configuration directory. Importantly, it did not overwrite the existing configuration files to avoid disrupting a user's current search session.

Each of the Diamond applications was fitted with a ”define scope” button that invoked a method in the OpenDiamond library to parse the downloaded file and rewrite the client's actual configuration files. This allowed the user to decide when a newly downloaded scope file would be actively used. Previous configuration files were rotated out to keep a log of past scope definitions. From this point, the previous collection/group mechanism handled scoping subsequent Diamond searches.

In this implementation, the group identifier served as the scope cookie, as it was assumed that any client that knew an identifier was permitted access to the data. Since group identifiers were drawn randomly from a sample space of size 2^{64}, guessing a group identifier was extremely unlikely. The Diamond content servers still served static collections of objects so they were not altered to consult the scope server for scope lists prior to search execution. This implementation choice also preserved the utility of the Volcano administration tool from Section 2.2.2. Positive usage experiences with the Gatekeeper encouraged the move to a full scope server design, presented in Section 3.5.

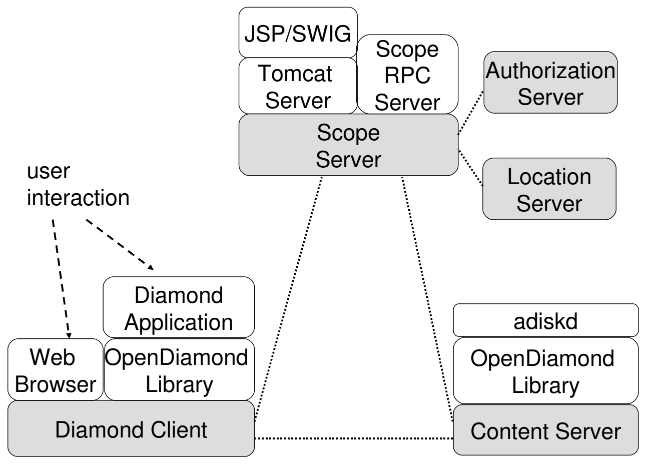


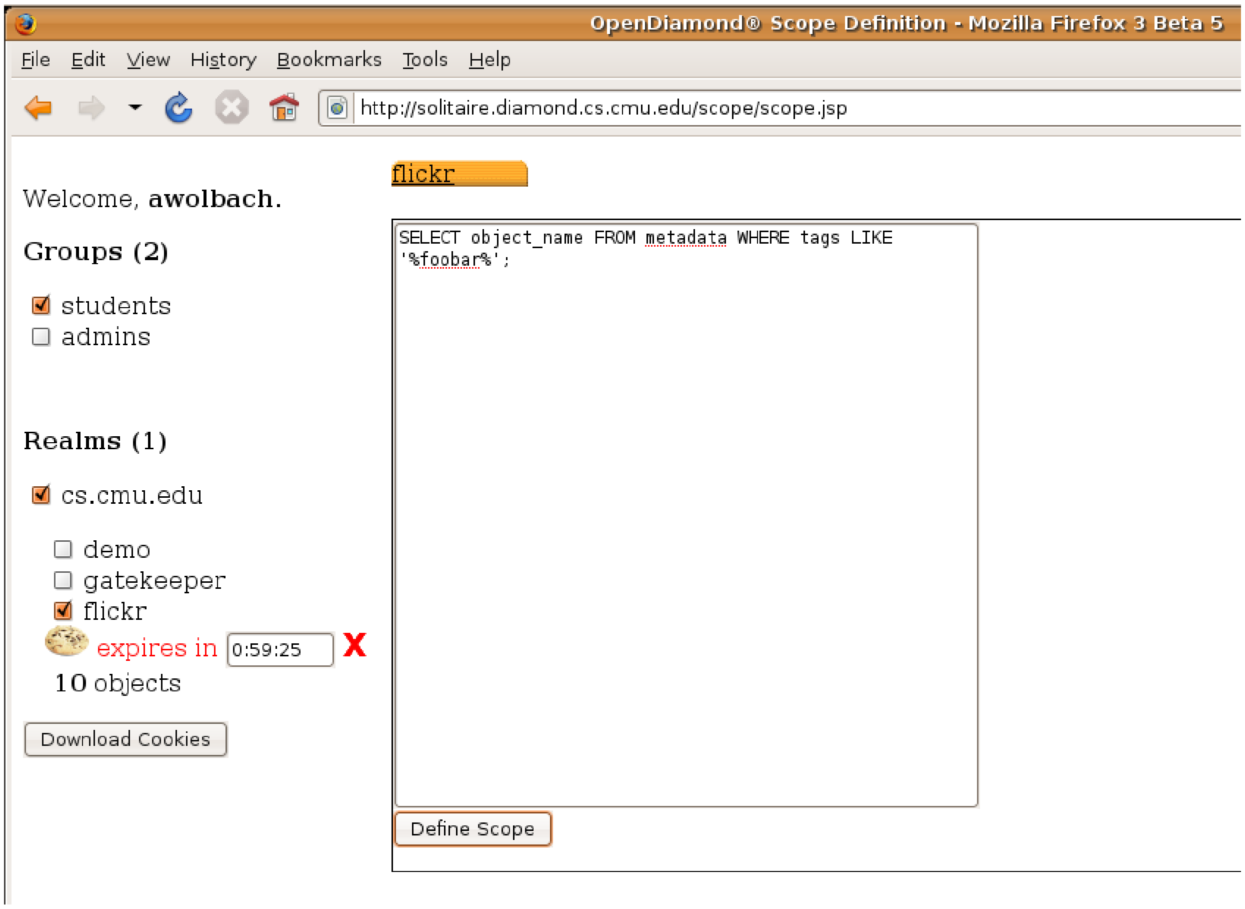
Figure 3.6: Full Scoping Design

3.5 Detailed Design and Implementation

The full addition of metadata scoping to Diamond introduces all of the components of the architecture, as presented in Figure 3.2: a web server that allows users to craft metadata queries; a scope server that authenticates users and controls access to various metadata sources; an authorization server that audits and approves metadata queries; and a location server that stores a mapping of object name to location. Its design is laid out in Figure 3.6.

3.5.1 Walkthrough

A user first navigates to the scoping website, which negotiates a secure connection using a trusted certificate issued from a certificate authority. The user may inspect this certificate to avoid potential phishing attacks and accepts it to continue. The web server first generates a SASL [43] authentication webpage with forms for the user to fill in. The use of SASL is described in section 3.5.3. When submitted, the server sends user information as SASL data to the scope server, which authenticates the user. If authentication succeeds, the web server presents the user with a scope definition webpage. The web server asks the scope server which roles this user may act within and presents this as a form to the user. The user selects a subset of these roles and submits them to the web server, which in turn forwards the selections to the scope server. Each role may access certain realms and metadata sources. The web server presents the user with realm and metadata sources available


OpenDiamond® Scope Definition - Mozilla Firefox 3 Beta 5
File Edit View History Bookmarks Tools Help
http://solitaire.diamond.cs.cmu.edu/scope/scope.jsp
flickr
Welcome, awolbach.
Groups (2)
students
admins
Realms (1)
cs.cmu.edu
demo
gatekeeper
flickr
expires in 0:59:25
10 objects
Download Cookies
Define Scope

```
SELECT object_name FROM metadata WHERE tags LIKE
'%foobar%';
```

Figure 3.7: Full Scoping Web Interface

to query. For each metadata source selection, the web server fetches a query definition webpage fragment tailored to that source. It then aggregates and isolates the fragments on a single, tabbed webpage. Figure 3.7 shows an example scoping web interface.

The user interacts with a single metadata source's webpage at a time, using its specific interface to craft and submit a query. Upon submission, the web server forwards the query to the scope server. The scope server then looks up the realm of the metadata source in a local database. If it is the local realm, the scope server contacts the realm's authorization server for inspection and auditing. Otherwise, it looks up the hostname of the foreign realm's scope server in the same local database and contacts it with the metadata query. In this situation, both realms' authorization servers audit the query, storing the query string, user, role, time-of-day and other contextual information, but only the foreign authorization server inspects the query. Paired with the query is a certificate request that serves as a potential scope cookie for the search. If the inspecting authorization server grants execution of the query, it signs the request and returns the certificate to its realm's scope server. If the query originated from a foreign realm, the scope server then returns it to that realm's scope server.

The scope server then contacts the metadata source, sending the query and receiving a list of object names back as a result. It locates the objects by querying the location server with this list, resulting in a list of object locations. The scope server takes the resulting list and pairs it with the certificate's unique serial identifier (which serves as a handle for the search), forming a scope list. The server stores the scope list for the duration of the search in a local database. When all object locations are stored, the scope server returns the list of content servers and the scope cookie to the web server. If the query is on a foreign realm, the foreign scope server returns first to the local scope server, which subsequently returns it to its web server. The web server holds all of the active cookies generated by the user so far and provides a "Download Cookies" button to retrieve them when scope definition is complete. When it is clicked, the web server concatenates all valid cookies into a single file, called the "cookie jar", and sends it to the web browser.

From there, a user may execute a search from any Diamond application. The OpenDiamond library checks for a cookie jar file when a new search is executed. It connects to all of the content servers paired with each scope cookie and forwards the cookie data. Upon receiving a new scope cookie, a content server verifies the cookie's activation and expiration times and its signatures. It then passes the cookie back to the scope server, which responds with the scope list for this search, and search execution proceeds as in the previous system.

3.5.2 Dynamic Web Interface

All communication between the web browser and the web server is encrypted with the use of SSL/TLS web certificates issued from a certificate authority. As in the Gatekeeper, the Apache 2 httpd web server [4] provides this functionality and forwards data between the web browser and the Apache Tomcat server. The web application then presents the user with a customized scoping interface. The web application uses Sun Microsystems' JavaServer Pages (JSP) technology [8] to dynamically generate a customized user interface. JSP allows developers to embed Java code directly in HTML and XML documents. This code executes on the web server and is responsible for outputting customized webpages for a specific user. User-specific state is defined by a Java class and held in a persistent Java object known as a "bean". The JSP code may access any of the public methods and variables of the bean's class during each HTTP request.

In this implementation, an Apache Tomcat server executes the JSP code. Tomcat is maintained by the Apache Software Foundation and provides a full implementation of Sun's JSP specifications [14]. It associates separate web requests by generating a unique web cookie for each newly created session, which the web browser supplies on each subsequent request. It serves generated webpages through the Apache HTTP server, which forwards them to the client web browser. The JSP code uses MiniRPC, explained in more detail in Section 6.2.1, to provide communication between web and scope server. However, since MiniRPC generates only C-language bindings for calls, a glue technology known as SWIG [12] was introduced to cross the Java/C language barrier. SWIG is an acronym for Simplified Wrapper and Interface Generator and allows programs written in high-level languages to communicate with programs or, as in this case, libraries written in C or C++. The SWIG stub generator relies on generating Java Native Interface (JNI) [7] code to connect Java to C. JNI is a programming interface from Sun Microsystems for including platform-specific code in Java classes or embedding a Java virtual machine inside of native programs or libraries. SWIG transforms C header files created by the MiniRPC stub generator *minirpcgen* into corresponding JNI interfaces and hence greatly simplifies crossing language boundaries. This enables the scope server to communicate with all components through standard MiniRPC interfaces.

3.5.3 User Authentication and Access Control

The full metadata scoping system provides user authentication using SASL [43]. SASL is an acronym for Simple Authentication and Security Layer and provides a protocol-independent, extensible framework for user authentication. Many implementations of common authentication mechanisms are implemented in the Cyrus SASL library which the scope server relies on, including plain username/password login, one-time passwords, and GSSAPI support for Kerberos V5. The user enters SASL authentication credentials into an authentication webpage which are sent to the Java bean. The bean and the scope server conduct a series of SASL steps as per the authentication mechanism used by using a generic SASL RPC call which forwards and returns base-64 encoded strings. Only the web server and scope server ever see the user's password. The bean uses a Java SASL package included in the Java SDK, while the scope server uses the Cyrus SASL C library. Due to the SASL specification's platform and protocol independence, communication between different SASL libraries is not an issue.

The scope server also manages associations between user and role, and between role, realm, and metadata sources in a locally accessible MySQL database. Role, realm and metadata source information is available to the web server through separate RPC methods. The system assumes that the administrators of a Diamond backend are responsible for actively updating this database with current access control information. When queries reference foreign metadata sources, the local scope server contacts the foreign realm's scope server to forward the query. The foreign scope server implicitly trusts that the local server successfully authenticated the user. The scope servers perform peer authentication using SSL/TLS certificates. This peer authentication mechanism is provided by the MiniRPC library.

3.5.4 Query Creation

The scope server is responsible for fetching the metadata webpages which enable a user to create a query for a specific source. The server contains realm and metadata source location information in a local MySQL database. It is responsible for contacting all metadata sources within its realm, and for contacting the scope servers of foreign realms. Additionally, it also knows the webpage URL of all metadata sources a user may access. It performs a URL access to each metadata source's webpage upon user request, allowing each to dynamically generate its scoping interface. The scope server receives the webpage results, separates the header and body from the webpage, and returns both to the web server. The only requirements imposed on each webpage are that they not include malformed HTML and that they make a specific JavaScript call to request a new scope, which includes the realm name, metadata source name, and metadata query ASCII string. The web server then includes in the combined web page each header fragment within the header section, and each body fragment as a separate tab, and sends the result to the user.

3.5.5 Query Authorization

The scope server first sends queries to an authorization server for approval. The scope server and authorization server communicate using MiniRPC. With each query authorization request is an X.509 certificate request which serves as a potential scope cookie. The request contains a unique serial number and the activation and expiration times of the scope list. Using X.509 certificates allows content servers to independently verify the validity of a scope cookie by checking that the signature of the certificate request matches the scope server's certificate and that the signature of the certificate itself matches the authorization server's certificate. Additionally, a scope server may publish a list of revoked certificates which content servers check prior to each execution of a Diamond search, allowing a Diamond administrator to immediately revoke a user's access to data.

Upon receiving this certificate request, the authorization server may be customized to execute specific inspection guidelines defined in a per-realm policy, or it may default to a basic all-or-nothing access control check as to whether the acting roles may access this metadata source. It is important to note that additional information privacy guidelines, when realm policy dictates their use (e.g., hospitals adhering to HIPAA guidelines), may be set within the metadata source and enforced through the inclusion of user and role information when executing a query. Thus, an authorization server only controls access on an all-or-nothing basis for each metadata source.

3.5.6 Metadata Query Execution

This implementation imposes the constraint on the scope server that metadata sources exist as MySQL databases, but this constraint is an implementation artifact. ODBC, or Open Database Connectivity, was briefly explored as an database-agnostic interface for querying metadata sources, but it was found that this excluded non-relational data sources such as those with XQuery interfaces and other XML data sources. If a specific realm wishes to use a different metadata type, they must customize the scope server to support it specifically. The only design assumption made is that executing a query results in a list of zero or more object names in the scope.

3.5.7 Scope Cookies, Jars, and Lists

When the scope server contacts the location server with a list of object names, the location server looks each object name up in a local MySQL database. It also pins the objects in those locations by setting a do-not-disturb timestamp which expires at the same time as the scope cookie. This avoids the problem of "phantom" objects, where objects disappear when an administrator moves them after a user defines a scope but before he can execute his search. The scope cookie then refers to the list of object locations returned to the scope server, known as a scope list.

The cookie jar file that the web server generates uses a specific "diamond/x-scope-cookie" MIME type. This allows a Diamond MIME type handler installed with the OpenDiamond library to execute upon downloading the cookie and place it in the user's Diamond configuration directory. This completely automates scope cookie management from the user's perspective, and since it occurs outside of or beneath the OpenDiamond library API, requires no changes to Diamond applications.

When content servers receive scope cookies from a user, they first validate and then

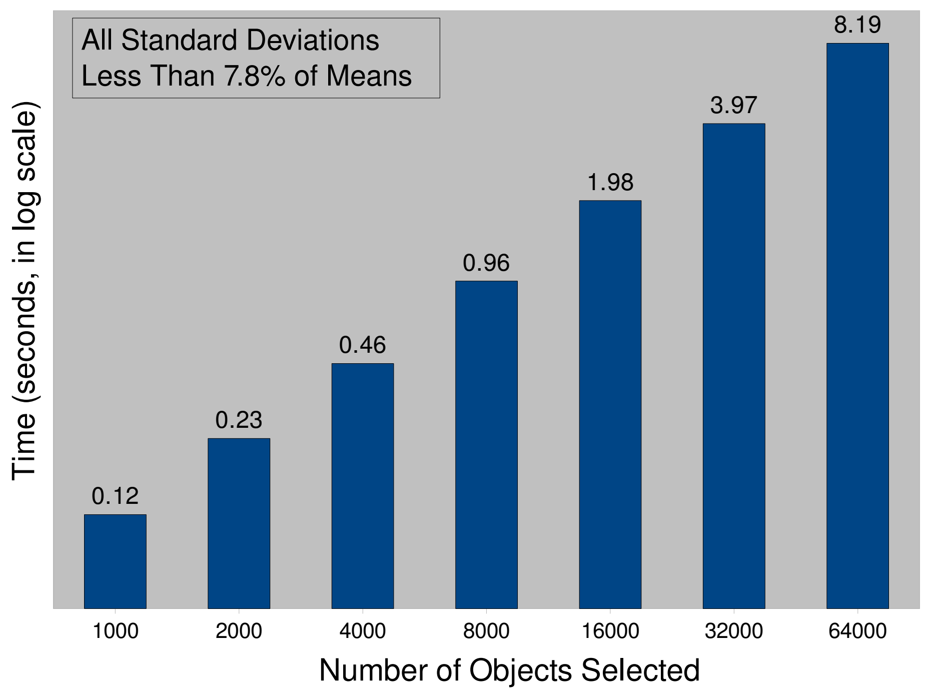


Figure 3.8: Overhead of Metadata Scoping

send the cookies back to their realm's scope server. This serves as a request for their portion of the corresponding scope list. Content servers and scope servers communicate using MiniRPC and perform peer authentication using SSL/TLS certificates to prevent impersonation. The content server requests the scope list in chunks through separate RPC calls and awaits transfer completion before allowing search execution.

3.6 Evaluation

There are two primary sources of delay in the metadata scoping system. The first arises from passing a query to a metadata source and blocking until query execution completes. Obviously, this depends on several factors outside of the control of the system, including the type of metadata source used and the complexity of the query. Due to these factors an evaluation of metadata query execution time is omitted. The second source of delay is the time required for a scope server to create a scope list after query results become available. The following experiments were conducted to measure this delay.

The scope server takes three logical steps to create a scope list. It first executes a metadata query, which takes X amount of time. Secondly, it fetches the result list of object names from the metadata source and uses it to query the location server to generate a list of object locations, which requires time Y. Finally, the scope server fetches the list of object locations and stores them as scope list in a on-disk database, which takes time Z. These experiments measured the delay of Y+Z, taking a result set of object names from a metadata source and forming a scope list. This delay was measured by executing a MySQL metadata query of the following form and measuring the amount of time necessary to create a scope list from its results.

```
SELECT object_name FROM metadata LIMIT N;
```

In these experiments, N began with a value of 1,000 and increased exponentially by powers of two to 64,000. The delays are plotted in Figure 3.8.

The delays experienced by the metadata scoping system ranged from tenths of seconds when a metadata query returned 1,000 results to just over eight seconds for 64,000. The delay scaled linearly with the number of results returned, increasing by roughly a second for each additional 8,000 results. A reasonable user, who might wait a minute for a query to execute, could expect to search nearly half of a million images in this manner.

The dataset used is a collection of 1.3 million JPEG images and its associated metadata, such as owner and image tags, downloaded from the website Flickr. In our prototype, the metadata source, authorization server, location server, and scope server execute simultaneously on a single machine. The metadata source and location server are MySQL databases served by the MySQL 5.0 server daemon. For experimental purposes, query caching was disabled in the MySQL server. The machine is a Dell Precision 380 desktop with a 3.6 GHz Pentium 4 processor, 4 GB RAM, and an 80 GB disk. It runs Ubuntu 8.04, which is based on the Linux 2.6.24-19 kernel.

3.7 Discussion

The metadata scoping mechanism achieves its stated goals. The Gatekeeper served as a proof-of-concept implementation that leveraged the existing collection and group mechanisms to present a user with an application-independent scoping interface. It provided a crude scope server which referenced SQLite metadata databases and presented the user with a PHP-generated web interface. It also removed the requirement that Diamond applications provide an interface for selecting collections to search. Users authenticated using the Kerberos V5 system and interacted over an encrypted connection. The Gatekeeper was successfully used from Summer 2007 to Summer 2008 at Carnegie Mellon University to give many demonstrations of the Diamond system.

The full design and implementation enables a Diamond realm to take advantage of external metadata sources while imposing minimal integration requirements on those sources, a clear improvement in the functionality of the system. It authenticates users through realm-specific SASL mechanisms and provides role-based access control on metadata sources. It also allows users of one realm to execute searches on a foreign realm. The use of TLS encryption in MiniRPC and the web server secures connections between all components in the design. Most importantly, Diamond applications do not require any knowledge of the metadata scoping mechanism; all client and content server scope cookie management occurs within the OpenDiamond library and no changes affected the OpenDiamond API. The new version of scoping has been successfully deployed at Carnegie Mellon and is able to search existing collections of data through an updated Gatekeeper mechanism, and a metadata source referencing 1.3 million images downloaded from Flickr. Users willing to tolerate delays of a few seconds for scope definition can define scopes containing thousands of objects. The metadata scoping mechanism will be released shortly as OpenDiamond version 5.

3.8 Chapter Summary

This chapter presented a system architecture that enables Diamond to scope Diamond searches with queries over external metadata sources, in an effort to improve the relevancy of search results. The design and implementation of an preliminary version of metadata scoping known as the Gatekeeper provided a validation of the system architecture and was successfully used for over a year. A subsequent complete design and implementation provides the full feature set including authentication, role-based access control, query audit, cross-realm search, and object location services. An evaluation showed that a primary source of delay due to the metadata scoping mechanism scaled linearly with the number of objects searched, and that a user could set a scope containing thousands of objects if he waited just a few seconds. The metadata scoping mechanism will be released shortly as OpenDiamond version 5.

Chapter 4

Enabling Mobile Search with Diamond

An attractive set of usage scenarios envisioned early on for Diamond involved users executing searches and receiving results on-the-go with a mobile device. The validation of work on the core Diamond system in late 2006 allowed system developers to revisit this possibility. Diamond applications imposed significant resource requirements on client hardware and software which appeared to present a serious obstacle to mobile system use. Further, since searchlet execution and state was shared between content servers and clients, either a common processor architecture or significant engineering work was required to share searchlet data in a standard way. These obstacles prompted a generalization of the problem with an application-independent approach.

A general approach is important because Diamond is far from the only application with interesting mobile usage scenarios saddled with resource requirements that challenge the limitations of mobile computing devices. Many applications are valuable on-the-go, including word processing, spreadsheet, and presentation applications, even when the user has only a few minutes to spare. Unfortunately, these applications are traditionally designed for desktop or laptop-class machines and are often cumbersome to use with a small form-factor machine or impossible to compile and execute for closed processor architectures.

Usability often suffers when a mobile device is optimized for size, weight and energy efficiency. On a handheld device with a small screen, tiny keyboard and limited compute power, it is a challenge to go beyond a limited repertoire of applications. A possible solution to this problem is to leverage fixed infrastructure to augment the capabilities of a mobile device, using techniques such as dynamically composable computing [37] or cyber foraging [17, 24, 18]. For this approach to work, the infrastructure must be provisioned with exactly the right software needed by the user. This is unlikely to be the case everywhere, especially at global scale. There is an inherent tension between standardizing infrastructure for ease of deployment and maintenance, and customizing that infrastructure to meet specific user needs.

Virtual machine (VM) technology provides a method to encapsulate and share customized application state with standardized infrastructure. Its use only imposes a single requirement, that the virtual machine monitor which executes a virtual machine be the same on all machines. A VM image contains all of the virtualized RAM and disk state used by the machine. As a result, its size can be in the tens of gigabytes, and transferring an entire VM to provision an infrastructure machine for transient use becomes an exercise in futility. Our solution to this problem drastically decreases the amount of data required to customize a VM executing on an infrastructure machine.

This chapter describes *Kimberley,* a system for rapid software provisioning of fixed infrastructure for transient use by a mobile device. Kimberley decomposes customized virtual machine (VM) state into a widely-available *base VM* and a much smaller, possibly proprietary, *private VM overlay.* These two components are delivered to the site being provisioned in very different ways. The base VM is downloaded by the infrastructure in advance from a publicly accessible web site. The private VM overlay is delivered to a server in the infrastructure just before use, either directly from the mobile device or under its control from a public web site. In the latter case, encryption-based mechanisms can be used to ensure the integrity and privacy of the private VM overlay. Once obtained, the overlay may be optionally cached for future reuse. The server applies the overlay to the base to create and execute a *launch VM.* When the user departs, this VM is terminated and its state is discarded. In some cases, a small part of the VM state may be returned to the mobile device; this is referred to as the *VM residue.* Figure 4.1 shows a typical Kimberley timeline.

It is anticipated that a relatively small number of base VMs (perhaps a dozen or so releases of Linux and Windows configurations) will be popular worldwide in mobile computing infrastructure at any given time. Hence, the chances will be high that a mobile device will find a compatible base for its overlays even far from home. The chances of success can be increased by generating multiple overlays, one for each of a number of base VMs. The collection of popular base VMs can be mirrored worldwide, and a subset can be proactively downloaded by each mobile computing infrastructure site.

The following section describes expected usage scenarios for Kimberley. Section 4.2 describes the design and implementation of Kimberley's components. Section 4.3 describes an experimental evaluation of the size of VM overlays and resume and teardown delays. The chapter concludes with a summary.

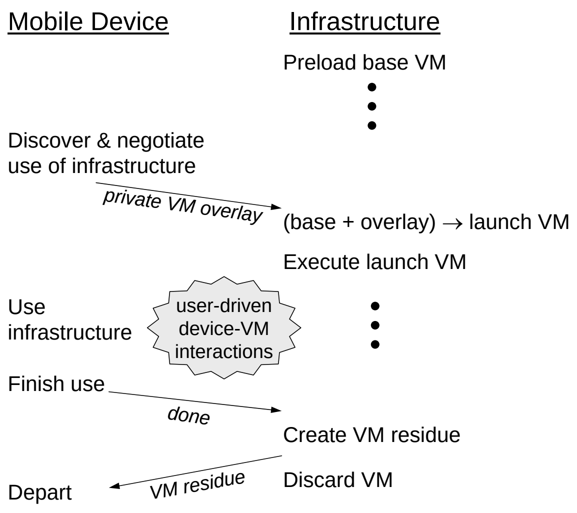


Figure 4.1: Kimberley Timeline

4.1 Usage Scenarios

The two hypothetical examples below illustrate the kinds of usage scenarios envisioned for Kimberley.

Scenario 1: *Dr. Jones is at a restaurant with his family. He is contacted during dinner by his senior resident, who is having difficulty interpreting a pathology slide. Although Dr. Jones could download and view a low-resolution version of the pathology slide on his smart phone, it would be a fruitless exercise because of the tiny screen. Fortunately, the restaurant has a large display with an Internet-connected computer near the entrance. It is sometimes used by customers who are waiting for tables; at other times it displays advertising. Using Kimberley, Dr. Jones is able to temporarily install a whole-slide image viewer, download the 100MB pathology slide from a secure web site, and view the slide at full resolution on the large display. He chooses to view privacy-sensitive information about the patient on his smart phone rather than the large display. He quickly sees the source of the resident's difficulty, helps him resolve the issue over the phone, and then returns to dinner with his family.*

Scenario 2: *While Professor Smith is waiting to board her flight, she receives email asking her to review some budget changes for her proposal. The attached spreadsheet shows a bottom line that is too high. After a few frustrating minutes of trying to manipulate the complex spreadsheet on her small mobile device, Professor Smith looks around* *the gate area and finds an unused computer with a large display. She rapidly customizes this machine using Kimberley, and then works on the spreadsheet. She finishes just before the final boarding call, and retrieves the modified spreadsheet on to her mobile device. On board, Professor Smith barely has time to compose a reply, attach the modified spreadsheet, and send the message before the aircraft door is closed.*

Other possible mobile computing scenarios in which Kimberley may prove useful include:

- viewing a map, possibly with personal annotations.
- impromptu presentations and demonstrations.
- spontaneous collaboration, as in choosing a restaurant.

The need for crisp user interaction in these scenarios deprecates the use of a thin client strategy. Network latency is of particular concern when the interactive application runs on a server that has to be reached across a WAN. Kimberley enables VM-based execution of an application on hardware close to the user, hence reducing network latency and improving the quality of user interaction. Although WAN bandwidth is relevant in determining VM overlay transmission time (and hence startup delay) in Kimberley, it is typically much easier to control and improve than WAN latency. In the worst case, WAN access can be completely avoided by delivering the VM overlay directly from the mobile device over a local network.

4.2 Detailed Design and Implementation

The Kimberley prototype uses a Nokia N810 Internet tablet with a 400MHz TI OMAP processor, 128 MB of DDR RAM and 256 MB flash memory, 2 GB flash internal storage, an attached 8 GB microSD card, and a 4-inch touch-sensitive color display. It supports 802.11b/g and Bluetooth networking, and is equipped with GPS and ambient light sensors. Its software is based on the Maemo 4.0 Linux distribution. The infrastructure machine in our prototype is a Dell Precision 380 desktop with a 3.6 GHz Pentium 4 processor, 4 GB RAM, an 80 GB disk, and a 20-inch 1600x1200 LCD monitor. It runs Ubuntu 7.10, which is based on the Linux 2.6.22-14 kernel. This machine has a 100 Mb/s Ethernet connection to the Internet; it also has 802.11b/g wireless network access via a USB-connected network interface. Three aspects of the implementation are described below.

4.2.1 Creating VM overlays

The VMs used in Kimberley are configured using *VirtualBox* [42] on a Linux host. VirtualBox is an open source virtual machine monitor that can be hosted on a variety of operating systems, including Linux. We provide an overlay creation tool called `kimberlize` that is invoked as follows:

```
kimberlize <baseVM> <install-script> <resume-script>
```

`baseVM` is a VM in which a minimally-configured guest OS has been installed. There are no constraints on the choice of guest OS, except that it must be compatible with `install-script` and `resume-script.` The tool first launches `baseVM,` and then executes `install-script` in the guest OS. The result is a VM that has been configured for use by the mobile device. Next, the tool executes `resume-script` in the guest OS. This launches the desired application, and brings it to a state that is ready for user interaction. The VM is now suspended. It can be resumed rapidly at runtime without the delays of guest reboot or application launch. There are no constraints on `install-script` and `resume-script` except for the obvious caveat that they should not halt or crash the guest.

Once the launch VM has been created, `kimberlize` uses the `xdelta` binary differencing tool to obtain the difference between the launch and and post-install memory images. In principle, `xdelta` could also be used to obtain the difference in disk images. However, this is not necessary since VirtualBox already maintains a compact differencing disk. The VM overlay consists of this differencing disk plus the `xdelta`-generated memory difference.

The VM overlay is then compressed using the Lempel-Ziv-Markov algorithm [41], which is optimized for fast decompression at the price of relatively slow compression. This is the appropriate tradeoff for Kimberley because decompression takes place in the critical path of execution at runtime and contributes to user-perceived delay. Further, compression is only done once but decompression occurs each time a mobile device uses infrastructure.

The final step of `kimberlize` is to encrypt the compressed VM overlay using AES-128 private-key encryption. An optional argument to `kimberlize` can be used to specify the private key; otherwise `kimberlize` generates a random key from `/dev/urandom.`

4.2.2 Binding to infrastructure

Figure 4.2 shows the key runtime components of Kimberley. The controller of the transient binding between mobile device and infrastructure server is a user-level process called *Kim-*

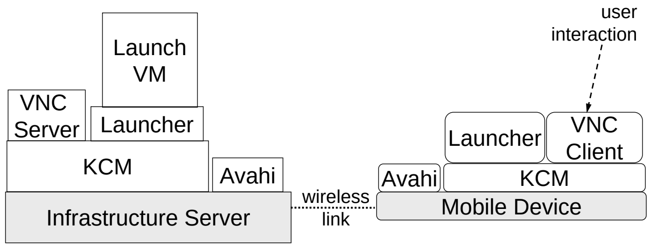


Figure 4.2: Runtime Binding in Kimberley

berley Control Manager (KCM). An instance of KCM runs on the device (“device KCM”) and on the infrastructure server (“server KCM”). The KCM abstracts service discovery and network management from the rest of Kimberley. Over time, we envision Kimberley supporting many different types of wireless and wired networks. While wireless connections are more natural and seamless to the user, there may be situations where the freedom from RF congestion of a dedicated wired link may be preferable for crisp user interaction. The current implementation supports 802.11 (wireless) and USB (wired) connectivity; we expect to add Bluetooth and Ethernet in the future. Note that the mobile device may have a direct connection to the Internet, in addition to the server connection shown in Figure 4.2.

KCM supports the browsing and publishing of services over the D-Bus interprocess communication mechanism [40]. The use of D-Bus simplifies the implementation of extensibility in service discovery by allowing KCM method calls to be defined in XML files. We currently support the Avahi service discovery mechanism [39], and plan to support the Bluetooth Service Discovery Protocol (SDP) in the future. It should also be possible to extend Kimberley to use other service discovery mechanisms.

The first step in the binding sequence is the establishment of a secure TCP connection using SSL between a device KCM and a server KCM. This secure tunnel is then used by the rest of the binding sequence, which typically involves user authentication and optional billing interaction. As in the metadata scoping work in Chapter 3, Kimberley supports the Simple Authentication and Security Layer (SASL) framework, which provides an extensible interface for integrating diverse authentication mechanisms. After successful authentication, a `dekimberlize` command is executed on the server. This reverses the effects of the original `kimberlize` command: it fetches the VM overlay from the mobile device or a Web site, decrypts and decompresses it, and applies the overlay to the base VM. The suspended VM is then launched.

The last step in the binding process is enabling user interaction via VNC [30]. With this setup, all user interactions occur at the mobile device and its screen mirrors a scaled image of the large display on the server. This style of interaction allows the user to remain untethered and to be at some distance from the large server display. However, the ease of interaction is limited by the dimensions of the mobile device. Kimberley also supports interaction through a full-sized keyboard, mouse and other devices attached to the server, as in Scenario 2. This approach may be preferred when the usage context requires more extensive interaction.

4.2.3 Sharing user data

Some usage contexts require data to be shared between the mobile device and infrastructure server. In Scenario 2, for example, the mobile device could be running Windows CE while the launch VM may run Windows XP. Although versions of the Excel spreadsheet application run in both environments, they are not identical: the mobile version is optimized for interaction in a small device. However, both versions share the same spreadsheet file format. Kimberley needs a mechanism to share such files between device and infrastructure. A file system such as Coda [34] would be an obvious choice. However, this adds to the list of dependencies for making infrastructure compatible with Kimberley, possibly restricting its deployment.

Our solution emulates a large FAT32 floppy disk using a part of the device's flash memory storage. When the mobile device is being used in isolation, this virtual floppy disk is accessible as a mounted file system. When binding to an infrastructure server, Kimberley unmounts this virtual floppy disk, transfers its contents to the server, and then mounts it in the launch VM. When unbinding, Kimberley unmounts the virtual floppy disk, computes the state change since it was mounted, and transmits this difference to the mobile device. The state difference is effectively the "VM residue" shown in Figure 4.1. At the device, Kimberley applies the difference and then mounts the virtual floppy disk. The default size of the virtual floppy disk is 20MB, but it can be configured to any size within the storage capacity of the mobile device. Although simple, this solution works well for the usage scenarios that we envision for Kimberley. It is idiot-resistant since the virtual floppy disk is never accessible simultaneously on both the mobile device and the VM.

4.3 Evaluation

The evaluation addresses three questions relative to the use of an infrastructure site that has no cached overlay state:

- How big are typical VM overlays?
- What is the typical startup delay?
- How fast can a user depart?

To answer these questions, six open-source Linux applications were examined. These are listed below, along with their better-known Windows counterparts:

- *AbiWord:* a word processor (Microsoft Word)
- *GIMP:* an image processor (Adobe Photoshop)
- *Gnumeric:* a spreadsheet program (Microsoft Excel)
- *Kpresenter:* a slide tool (Microsoft PowerPoint)
- *PathFind:* a digital pathology tool for whole-slide images based on the OpenDiamond platform and suggestive of Scenario 1.
- *SnapFind:* a tool for examining digital photographs and searching for other similar photographs. It is also based on the OpenDiamond platform.

As a limiting case to explore the intrinsic overhead of Kimberley, we also considered a hypothetical *Null* application by invoking `kimberlize` with empty `install-script` and `resume-script` parameters.

4.3.1 Overlay Size

Table 4.1 presents data relevant to the first question. The table shows that the compressed overlay size for these seven applications ranges from 5.9 MB for the Null case to 196.6 MB for PathFind. The uncompressed sizes indicate a typical compression factor of about three. We initially expected overlay creation to be a completely deterministic process, and were therefore surprised to see small variations in overlay sizes for the same application in different experimental runs. We conjecture that this is an artifact of VirtualBox's differencing disk implementation. Based on more detailed experiments (not described here), we conjecture that VirtualBox asynchronously preallocates real disk space for its virtual disk. The effect is modest, amounting to just a few percent of total overlay size. The last column gives the total size of the installation packages for an application used in creating the VM overlay. This size is loosely correlated with the application's compressed and uncompressed overlay sizes.

Table 4.2 shows the time taken by `kimberlize` to create overlays. Recall that `kimberlize` sacrifices compression speed for fast decompression at runtime. This accounts for the relatively long times shown in Table 4.2, ranging from just over a minute for Null to nearly 11 minutes for Gnumeric.

Application	Compressed VM Overlay Size (MB)	Uncompressed VM Overlay Size (MB)	Install Package Size (MB)
AbiWord	119.5 (3.8)	364.2 (19.9)	10.0
GIMP	141.0 (5.2)	404.7 (30.7)	16.0
Gnumeric	165.3 (3.6)	519.8 (24.7)	16.0
Kpresenter	149.4 (7.6)	426.8 (32.2)	9.1
PathFind	196.6 (1.0)	437.0 (2.3)	36.8
SnapFind	63.7 (0.9)	222.0 (3.6)	8.8
Null	5.9 (0.0)	24.8 (0.0)	0.0

Note: Small figures in parentheses are standard deviations

Table 4.1: VM Overlay and Install Package Sizes (8 GB Base VM size)

Table 4.3 shows the overlay sizes from a more recent version of `kimberlize`, which contains many implementation-level improvements, most notably a much more efficient version of VirtualBox. As a result, the newer overlay sizes are much smaller. These experiments were run in a different environment, including a newer guest operating system, and thus Table 4.3 should not be compared directly with Table 4.1. The results include sizes for four of the previously mentioned applications. The line titled "Suite" represents the installation of all four applications in a single `kimberlize` run. Particularly of note is that the application suite overlay is much smaller than the sum of the separate overlay sizes. These savings are mostly accounted for by the memory image, whose size stays relatively constant regardless of whether one or more applications are installed. This is likely a result of the reuse of common parts of memory for sequential tasks, such as the stack and the heap. Smaller savings on disk were also seen, likely as a result of overlap in the installation packages.

4.3.2 Startup Delay

To answer the second question ("What is the typical startup delay?"), the time it takes from a user's initial action to use infrastructure to the point at which he starts interacting with the application was measured. This includes the time taken to transmit the VM overlay and virtual floppy disk, to apply the VM overlay, and to launch the VM.

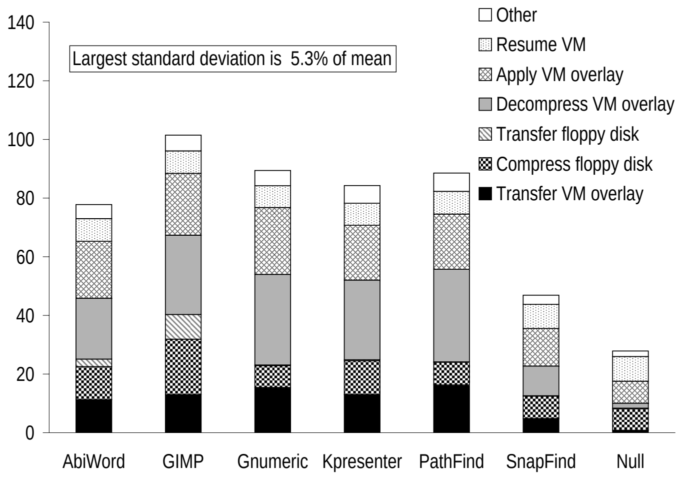
Application	Elapsed time for `Kimberlize` (seconds)
AbiWord	513 (32.4)
GIMP	527 (39.3)
Gnumeric	652 (36.3)
Kpresenter	548 (45.1)
PathFind	518 (4.3)
SnapFind	277 (3.9)
Null	81 (0.7)

Note: Small figures in parentheses are standard deviations

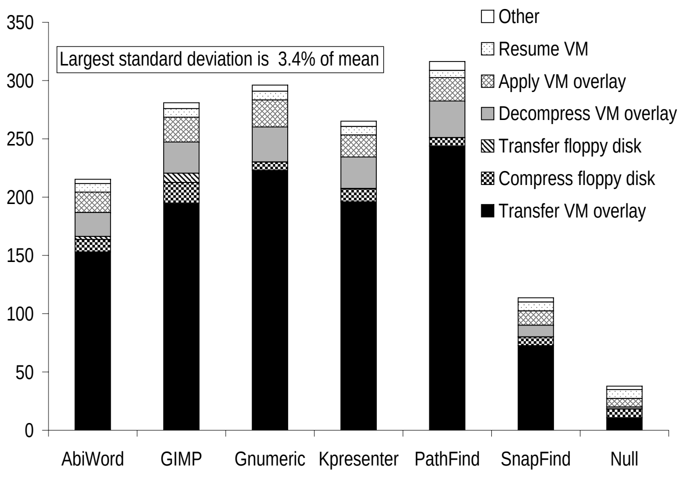
Table 4.2: Creation Time for VM Overlays (8 GB Base VM Size)

Application	Compressed VM Overlay Size (MB)	Uncompressed VM Overlay Size (MB)	Install Package Size (MB)
AbiWord	43.5	105.3	4.1
GIMP	31.7	80.1	0.0
Gnumeric	87.9	308.5	3.4
Kpresenter	186.7	479.4	29.4
Suite	217.0	636.4	37.0

Table 4.3: VM Overlay and Install Package Sizes with Application Suite (8 GB Base VM size)



(a) 100 Mbps



(b) 10 Mbps

Figure 4.3: Startup delay in seconds. (a) and (b) represent possible WAN bandwidths between an infrastrcture machine and an overlay server.

The experiments used the same set of seven applications and the VM configuration discussed in Section 4.3.1. The hardware used in these experiments is the same as that described at the beginning of Section 4.2. Since network bandwidth is a critical factor that affects overlay transmission time, we explored performance at two different bandwidths. The first bandwidth of 100 Mbps is typical of what one might expect within an enterprise. The second bandwidth of 10 Mbps is suggestive of well-connected public infrastructure. For example, in Scenarios 1 and 2, this might be the bandwidth available from the restaurant or airport gate to the overlay server. Note that bandwidth to the overlay server is independent of the wireless connnection between mobile device and infrastructure over which control interactions and transfer of the virtual floppy disk take place.

Figure 4.3 presents the results for two bandwidths: 100 Mbps and 10 Mbps. Five runs of each experiment were performed. At 100 Mbps, the transfer time for VM overlays only accounts for one-fifth or less of the total startup delay for most applications. Decompressing and applying the VM overlay are the dominant contributors to delay, accounting for roughly half of the total startup delay in most cases. The time to compress the virtual floppy disk is also significant. Improving this quantity is complicated by energy considerations for mobile devices. The total startup delay at 100 Mbps is quite acceptable, ranging from under a minute for SnapFind to just over a minute and a half for GIMP.

At 10 Mbps, VM overlay transfer time dominates. It accounts for roughly three-fourths of the total startup delay for most applications in Figure 4.3(b). Decompressing the VM overlay and applying it account for roughly one-fourth of the total time in those cases. The total startup delay ranges from roughly 3 to 5 minutes in most cases, which is at the outer limit of acceptability for transient use of infrastructure. This can be improved by using techniques such as data staging [21] to proactively transfer an overlay close to an anticipated usage site. In that case, the startup delay experienced would be closer to the 100 Mbps case. A cached overlay would, of course, also greatly improve startup delay.

4.3.3 Teardown Delay

Before a user can depart from a site, three steps must occur: the VM must be powered down; the server must generate the VM residue; and the residue must be transferred to the mobile device. Measurements (omitted here) indicate that all of these steps combined require less than one second for all the applications studied. Post-departure cleanup of state at the server and on the mobile device take a few additional seconds.

4.4 Chapter Summary

This chapter presented Kimberley, a system that applies VM technology to the problem of provisioning infrastructure for use by mobile computing devices. A good solution to this problem must respect the resource limitations of mobile devices while also keeping the configuration complexity of infrastructure low. Kimberley accomplishes the first goal by avoiding VM execution on the mobile device (hence reducing CPU and memory demand) and by using indirection to deliver VM overlays to the infrastructure from third-party sites (hence reducing wireless bandwidth demand and energy use on the mobile device). The second goal is accomplished through the use of VM technology. Rather than infrastructure having to be configured for many different applications, it only has to be configured to support Kimberley. The rest of the configuration happens dynamically, through VM overlays. By simplifying configuration, Kimberley lowers a key barrier to the widespread deployment of mobile computing infrastructure.

Likewise, Kimberley also achieves its original goal, to successfully provide the transient use of Diamond applications in mobile environments, through its general approach. Infrastructure machines can be customized with Diamond applications and execute as expected with minimal delay to the user. This enables many usage scenarios for Diamond and further increases its deployability.

Chapter 5

Related Work

The work described in this document focuses on functional improvements to the Diamond system. Many of the techniques described in this document are similar to those used in other systems. However, to the best of our knowledge they have never been applied to content-based search.

This chapter is divided into two sections. The former notes related work that influenced the design of Diamond's metadata scoping mechanism. The latter section describes two projects that laid the ideological groundwork for Kimberley.

5.1 Metadata Scoping Systems

5.1.1 Separation of Data Management from Storage

The introduction of new scope, authorization, and location servers centralizes access to data in a Diamond realm at a small number of points. This allows Diamond to eliminate the use of content server resources for operations not critical to search performance. This system design choice was influenced heavily by the Network-Attached Secure Disk research of Gibson et al. [23] which focused on maximizing performance in I/O-bound systems. The NASD system uses a file manager for controlling access to data and a storage manager for concurrency control and data location. These servers' functionality is reflected in the Diamond system architecture with the presence of scope and location servers which serve similar purposes.

5.1.2 Federation Between Realms

The metadata scoping work also enforces the federation of user identity across realms through a realm's scope server. Federation allows scope servers in different realms to implicitly trust that a search request from a foreign realm originates from the authenticated user requesting access. The choice to federate user identity is similar to the manner in which the Extensible Messaging and Presence Protocol (XMPP) [31] operates. XMPP Jabber servers authenticate clients and speak to foreign servers on their behalf to exchange user messages and presence information. This design choice removes the requirement of a central authentication entity and allows different realms to choose the most appropriate authentication mechanism for their purpose.

5.1.3 Content Management

Daisy [2] provides a general content management system with a focus on extensibility and ease of integration. It provides a generic XML interface to Daisy "documents", which contain one or more fields of opaque or typed data. Users may submit XML queries over these fields to a repository server, which executes the query over a subset of documents and returns those which satisfy it. Daisy was extensively examined as a possible platform for querying metadata sources in Diamond. It provides a general interface to metadata sources that fits Diamond's model well. Ultimately, however, it was decided that imposing XML query and response requirements on metadata sources was too strict. Potential Diamond deployments may not desire or be able to transfer large amounts of data into Daisy repository servers and as a result a more general system architecture was developed.

5.2 Rapid Provisioning of Fixed Infrastructure

5.2.1 Virtual Machines as a Delivery Platform

The Kimberley system relies on virtual machine technology to present a user with his customized application in a mobile environment. The use of VM technology encapsulates all user state in an application-independent manner and standardizes Kimberley installations on fixed infrastructure. This approach was heavily influenced by work on the Internet Suspend/Resume (ISR) [33] system, which takes an infrastructure-based approach to presenting a user with the same customized environment at each machine he uses. The key difference between Kimberley and ISR is that ISR attempts to persist a user's interaction with a virtual machine across hardware and time, while Kimberley assumes that the user's interaction is brief and not valuable enough to save. As a result, Kimberley discards VM state on infrastructure hardware while ISR provides a mechanism for maintaining and updating state on its servers.

5.2.2 Controlling Large Displays with Mobile Devices

The usage scenario Kimberley aims to enable is the transient control of a large display with a handheld mobile device. The attractiveness of this scenario comes from previous research into flexi-modal and multi-modal user interfaces spread across multiple machines within the Command Post of the Future (CPOF) [28] project at Carnegie Mellon University. The CPOF project explored new user interfaces possible when using handheld devices together with PCs [29]. The relationship of these ideas to transient application use in Kimberley was obvious and provided an impetus for exploring mobile Diamond use.

5.2.3 Mobile Computing

Further afield of Kimberley, Want et al. [37] focus on the problem of dynamically composing hardware components to yield a computing device with desired capabilities. Balan et al. [18] focus on the problem of creating mobile computing applications that can use cyber foraging. Goyal and Carter [24] discuss secure cyber foraging using services provided by VMs in the infrastructure; their work assumes that the VMs have been pre-configured for the desired services. Outside mobile computing, a recent commercial product *vizioncore vPackager* uses VM differencing to support a community of collaborative software developers [36].

Chapter 6

Conclusions

This document described two mechanisms that fundamentally improve the deployability of Diamond. It introduced the ability to increase the relevancy of search results by executing a complex query over indexed metadata. It also described *Kimberley*, a system enabling mobile use of Diamond by rapidly customizing infrastructure hardware. Experimental data confirms both mechanisms incur low overheads over the traditional Diamond system.

6.1 Contributions

The two main contibutions of this thesis are: a demonstration that highly-indexed metadata sources can be effectively incorporated into Diamond searches; and a demonstration that fixed infrastructure can be used to facilitate the transient use of Diamond on a mobile device. More specifically, this thesis makes contributions in the following areas:

1. *System architecture and design contributions*

    - *Architectural redesign of Diamond to incorporate metadata scoping.*
      Redesign incorporating a metadata scoping service into the Diamond system architecture which improves the overall quality of search with little overhead (Section 3.3).
    - *Design of a system architecture to facilitate mobile application use by exploiting fixed infrastructure.*
      Design of system architecture which utilizes virtual machine technology to rapidly customize fixed infrastructure with a desired appplication (Section 4.2).

2. *Engineering contributions*

   - *Reengineering of Diamond's communications layer.*

     Research of existing client/server communications libraries and subsequent incorporation of Sun Microsystems' Remote Procedure Call library as Diamond's communications layer (Section 2.1.2).

   - *Design and implementation of a content distribution tool for Diamond.*

     Introduction of *Volcano* to ease the integration of new data into the Diamond system by eliminating much of the bookkeeping associated with maintaining ASCII text configuration files, while retaining valuable information such as the data's provenance (Section 2.2.2). Volcano became part of the official 3.1 release of the Carnegie Mellon implementation of Diamond.

   - *Preliminary and final implementations of metadata scoping.*

     The *OpenDiamond Gatekeeper* served as a usable proof-of-concept implementation of the core pieces of the scoping architecture at a coarse granularity. A subsequent implementation provides the full power and flexibility available when scoping Diamond searches with metadata queries (Sections 3.4 and 3.5). The Gatekeeper became part of version 3.1 of Carnegie Mellon's implementation of Diamond and was officially released. The latter is expected to be released shortly as Diamond version 5.

   - *Implementation of the Kimberley system enabling mobile use of Diamond.*

     Kimberley provides a working implementation of an application-independent system designed to enable the transient use of fixed infrastructure by mobile devices (Section 4.2). Kimberley successfully executed several Diamond applications with short resume and teardown delays.

3. *Evaluation contributions*

   - Controlled experiments demonstrating the ability of the scoping mechanism to create and store large scope lists (Section 3.6).

   - Controlled experiments demonstrating usable, human-scale resume and teardown latencies during mobile use of Diamond and other applications in Kimberley (Section 4.3).

6.2 Future Work

6.2.1 Engineering Work

RPC Subsystem

Since August 2007, a new RPC mechanism called MiniRPC has been developed for another project at Carnegie Mellon known as Internet Suspend/Resume. MiniRPC was created to address the shortcomings of the 25-year-old ONC RPC design and implementation, including improved support for asynchronous communications, event handling, and secure communications. Like ONC RPC, it relies on XDR as an external data specification and has similarly structured interface definition files. It also provides many implementation-level improvements over ONC RPC, including the ability to start an MiniRPC server on a connected socket, and event callbacks to handle network I/O errors.

Due to its prototype status in mid-2007, MiniRPC was not seriously considered as a candidate for Diamond's RPC subsystem. During the past year it has evolved into a stable project and provides the core communications protocol of the Internet Suspend/Resume system. Its stability and attractive feature set prompted a new effort to replace ONC RPC with MiniRPC in mid-2008, culminating in the successful release of OpenDiamond version 4.

Future work on MiniRPC includes the addition of encryption to network traffic through the use of SSL or TLS certificates. It is expected that this work will secure communication in Diamond between clients and servers. However, some engineering work will be required in updating Diamond to take advantage of this feature.

Volcano

Currently, the Volcano tool does not provide utility for system administrators wishing to deploy Diamond with metadata scoping. Since the collection and group scoping mechanisms were retained during this work, it provides some benefit to casual users interested in a small deployment of Diamond to test. Future engineering work may allow it to also help system administrators taking advantage of Diamond's scoping mechanism, by updating the tool to generate scripts which update the location server and scope server databases. The automation of updates to object location and access control would improve the life of a Diamond system administrator considerably.

6.2.2 Metadata Scoping

One potential way in which the design of the metadata scoping mechanism may be improved would be the addition of a standard interface for accessing metadata sources. MySQL does not provide the ability to access other types of databases or other data sources through its network interface, and no universal mechanism was known as the time of this work. It is possible the creation of a dynamically linked plugin architecture with a standard procedural interface would ease the incorporation of new types of metadata sources into Diamond.

An improvement in system performance could be seen by adapting content servers to execute searches on partial scope lists as they become available. By streaming scope lists from scope servers, a content server could execute a Diamond search without waiting for a fetch of the scope list to complete. The time to locate objects and store the list, however, is unavoidable. As scope lists may be sizable in large-scale Diamond deployments, streaming offers a large potential speedup and eliminates a primary source of delay in the backend.

6.2.3 Kimberley

The prototype Kimberley implementation misses several engineering opportunities to improve the performance of the system. During resume, Kimberley writes the full VM overlay to disk after each logical task in the process, including download, decryption, and decompression. Each tool that performs a task also provides streaming output to allow programs to use pipelined architectures and thus avoid disk accesses. The use of a pipelined resume process in Kimberley would avoid the lengthy intermediate writes to disk, a significant source of resume delay.

Additionally, the resume process does not cache VM overlay data on infrastructure hardware after use. Caching popular VM overlays or using lookaside content-addressable storage would minimize the size of future downloads, improving resume delays. This is particularly attractive in bandwidth-constrained scenarios. However, the use of caching techniques must respect privacy concerns and allow worried users to wipe their personal state from disk.

6.3 Final Thoughts

Prior to this work, Diamond existed as a usable system able to search static collections of objects from well-equipped client hardware. Diamond required considerable administrative effort to manually configure content servers and clients correctly. The goal of this thesis was to improve the deployability of Diamond. The addition of a metadata scoping mechanism allows the system to leverage existing metadata repositories nearby actual data at minimal cost. The work on Kimberley enables mobile clients to execute applications, including Diamond client applications, far exceeding the limitations of the mobile devices themselves. Experimental results confirm that the delays imposed by each system are within acceptable ranges and each provides a fundamental extension to the Diamond system. Their introduction enables a broad swath of new applications for Diamond.

Appendix A

Volcano Manual Page

```
volcano(8)

NAME
        volcano - Diamond System Administration Tool

SYNOPSIS
        volcano create <collection> <index-filename> <server1> [server2] ...
        volcano remove <collection>
        volcano list <collection>

DESCRIPTION
        create <collection> <index-filename> <server1> [server2]...
                Create a new collection named collection over the
                content servers listed.  The objects to be
                inserted are listed in the text file index-filename, one per
                line.  Directories may also be listed, which causes
                a recursive search for all file descendants as
                objects.  The tool then generates a script,
                distribute_objects.sh, which distributes the objects
                in a round-robin fashion when it is executed.

                This call also outputs two other files:  an SQLite
                provenance database describing which files were
                distributed to which content server; and an SQLite
                scope database containing collection name to group
                ID to server hostname mappings.

        remove <collection>
        Delete references in collection’s index
        files related to the files listed.  Does not
        actually remove any data.

list <collection>
        List all objects in collection.
```

Bibliography

[1] Common Object Request Broker Architecture: Core Specification. http://www.omg.org/docs/formal/04-03-12.pdf. 2.1.2

[2] Daisy - the open source CMS. http://cocoondev.org/daisy/index.html. 5.1.3

[3] Distributed Computing Environment RPC. http://www.opengroup.org/dce/. 2.1.2

[4] The Apache HTTP Server Project. http://httpd.apache.org/. 3.4, 3.5.2

[5] The Internet Communications Engine. http://zeroc.com/ice.html. 2.1.2

[6] Java™ Remote Method Invocation. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html, 2004. 2.1.2

[7] JDK 6 Java Native Interface-related APIs & Developer Guides. http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html, 2006. 3.5.2

[8] JavaServer Pages Technology. http://java.sun.com/products/jsp/. 3.5.2

[9] Pubcookie. http://www.pubcookie.org/. 3.4

[10] SOAP Specifications. http://www.w3.org/TR/soap/, 2007. 2.1.2

[11] SQLite). http://www.sqlite.org/. 3.4

[12] Simplified Wrapper and Interface Generator. http://www.swig.org/. 3.5.2

[13] Introduction to TI-RPC (ONC+ Developer's Guide). http://docs.sun.com/app/docs/doc/816-1435/rpcintro-46812. 2.1.2

[14] Apache Tomcat. http://tomcat.apache.org/. 3.5.2

[15] Vortex Library. http://www.aspl.es/vortex/. 2.1.2

[16] XML-RPC Specification. `http://www.xmlrpc.com/spec`, 1999. 2.1.2

[17] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang. The Case For Cyber Foraging. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002. 4

[18] Rajesh Balan, Darren Gergle, Mahadev Satyanarayanan, , and James Herbsleb. Simplifying Cyber Foraging for Mobile Devices. In *Proceedings of the 5th International Conference on Mobile Computing Systems, Applications and Services (MobiSys 2007)*, San Juan, Puerto Rico, June 2007. 4, 5.2.3

[19] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/2080.357392. 2.1.2

[20] M. Eisler. XDR: External Data Representation Standard. RFC 4506 (Standard), May 2006. URL `http://www.ietf.org/rfc/rfc4506.txt`. 2.1.2

[21] Jason Flinn, Shafeeq Sinnamohideen, Niraj Tolia, and M. Satyanaryanan. Data Staging on Untrusted Surrogates. In *Proceedings of the FAST '03 Conference on File and Storage Technologies*, San Francisco, CA, March 2003. 4.3.2

[22] Gregory R. Ganger, John D. Strunk, and Andrew J. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical report, 2003. 3.3.3

[23] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Oper. Syst. Rev.*, 32(5):92–103, 1998. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/384265.291029. 5.1.1

[24] Sachin Goyal and John Carter. A Lightweight Secure Cyber Foraging Infrastructure for Resource-Constrained Devices. In *Proceedings of the Sixth Workshop on Mobile Computing Systems and Applications*, English Lake District, UK, December 2004. 4, 5.2.3

[25] W. Harold. Using Extensible Markup Language-Remote Procedure Calling (XML-RPC) in Blocks Extensible Exchange Protocol (BEEP). RFC 3529 (Experimental), April 2003. URL `http://www.ietf.org/rfc/rfc3529.txt`. 2.1.2

[26] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of Usenix File and Storage Technologies (FAST)*, April 2004. 1.1

[27] Sun Microsystems. RPC: Remote Procedure Call Protocol specification: Version 2. RFC 1057 (Informational), June 1988. URL `http://www.ietf.org/rfc/rfc1057.txt`. 2.1.2

[28] Brad Myers, Robert Malkin, Michael Bett, Alex Waibel, Ben Bostwick, Robert C. Miller, Jie Yang, Matthias Denecke, Edgar Seemann, Jie Zhu, Choon Hong Peck, Dave Kong, Jeffrey Nichols, and Bill Scherlis. Flexi-modal and multi-machine user interfaces. In *In IEEE Fourth International Conference on Multimodal Interfaces*, pages 377–382, 2002. 5.2.2

[29] Brad A. Myers. Using handhelds and pcs together. *Commun. ACM*, 44(11):34–41, 2001. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/384150.384159. 5.2.2

[30] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. Virtual Network Computing. *IEEE Internet Computing*, 2(1), Jan/Feb 1998. 4.2.2

[31] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), October 2004. URL `http://www.ietf.org/rfc/rfc3920.txt`. 5.1.2

[32] M. Satyanarayanan, Rahul Sukthankar, Adam Goode, Larry Huston, Lily Mummert, Adam Wolbach, Jan Harkes, Richard Gass, and Steve Schlosser. The OpenDiamond® Platform for Discard-based Search. Technical Report CMU-CS-08-132, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 2008. URL `http://www.diamond.cs.cmu.edu/papers/CMU-CS-08-132.pdf`. 1.1

[33] Mahadev Satyanarayanan, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and H. Andres Lagar-Cavilla. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 11(2):16–25, 2007. ISSN 1089-7801. doi: http://dx.doi.org/10.1109/MIC.2007.46. 5.2.1

[34] Satyanarayanan, M. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2), May 2002. 4.2.3

[35] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Facebook, Palo Alto, CA, April 2007. URL http://developers.facebook.com/thrift/thrift-20070401.pdf. 2.1.2

[36] vizioncore. vPackager Version 1.0 User Manual, 2007. [Online; accessed on 13-April-2008 at http://www.vizioncore.com/vPackager.html]. 5.2.3

[37] Roy Want, Trevor Pering, Shivani Sud, and Barbara Rosario. Dynamic Composable Computing. In *Proceedings of the Ninth Workshop on Mobile Computing Systems and Principles (HotMobile 2008)*, Napa, CA, February 2008. 4, 5.2.3

[38] J.E. White. High-level framework for network-based resource sharing. RFC 707, December 1975. URL http://www.ietf.org/rfc/rfc707.txt. 2.1.2

[39] Wikipedia. Avahi (software) — Wikipedia, The Free Encyclopedia, 2008. [Online; accessed 9-June-2008 at "http://en.wikipedia.org/w/index.php?title=Avahi_89746162"]. 4.2.2

[40] Wikipedia. D-Bus — Wikipedia, The Free Encyclopedia, 2008. [Online; accessed 9-June-2008 at http://en.wikipedia.org/w/index.php?title=D-Bus&oldid=217062865]. 4.2.2

[41] Wikipedia. Lempel-Ziv-Markov chain algorithm — Wikipedia, The Free Encyclopedia, 2008. [Online; accessed 22-April-2008 at http://en.wikipedia.org/w/index.php?title=Lempel-Ziv-Markov_chain_algorithm&oldid=206469040]. 4.2.1

[42] Wikipedia. VirtualBox — Wikipedia, The Free Encyclopedia, 2008. [Online; accessed 21-April-2008 at http://en.wikipedia.org/w/index.php?title=VirtualBox&oldid=206252157]. 4.2.1

[43] K. Zeilenga. Anonymous Simple Authentication and Security Layer (SASL) Mechanism. RFC 4505 (Proposed Standard), June 2006. URL http://www.ietf.org/rfc/rfc4505.txt. 3.5.1, 3.5.3