Jiffy: A Lightweight Jini File System Service

Edward Hogan
ehogan@cs.cmu.edu

Julio López
jclopez@cs.cmu.edu

Abstract

Jiffy is a distributed file system implemented to take advantage of Sun Microsystems's Jini Technology. This paper contains quantitative results gathered from an experiment that compared the performance of Jiffy against the Andrew File System. The experiment compared the two file systems using several workloads and several simultaneous clients in order to determine the scalability of the system and to locate places that lead to possible overhead. Based on the results of the experiments, we argue that Jiffy is indeed an efficient file system, because it is able to take advantage of client-side caching, callbacks, and session semantics for sharing, each contributing to reduced traffic with the server. Our results show that in many cases Jiffy is able to outperform AFS. We examine possible influences on the data and conclude that for a specific class of clients, a file system built on top of Jini Technology can be extremely efficient and robust.

1 Introduction

An important breakthrough in the field of computer systems is the ability to rapidly bring network technologies to newly developed devices. Sun Microsystems built Jini Technology™ as a toolkit that could help simplify issues that arise when integrating network technologies with these new devices [1]. Jini provides a variety of services to the programmer, such as discovery, lookup, leasing, distributed events, and transactions. Although the goal of Jini is to facilitate the rapid development of distributed applications, few existing applications have adopted this toolkit because it has been released only recently to the public. We argue that Jini is a powerful toolkit based on experimental results acquired from a comparison of the Jiffy File System, a network file system that we implemented using Jini, against the Andrew File System.

This paper analyzes the Jiffy File System in two ways. First, through implementation, Jiffy can expose the costs and benefits of using the Jini toolkit to develop a network application from a programming perspective. Second, through benchmarking the performance of Jiffy against the Andrew File System, after which Jiffy is modeled, we can determine the performance overhead incurred by using the Jini toolkit in distributed applications. From the results of these two analyses, we are able to gain qualitative and quantitative data about Jini Technology.

Since there is much research on the development of distributed file systems, we used a number of existing optimizations while still relying on Jini for the infrastructure. Our Jiffy File System takes advantage of client-side caching, uses server callbacks, and makes concurrent file sharing efficient by using AFS [12] or private-copy-until-close semantics.

In addition to a discussion of design and implementation of Jiffy, this paper reports the results of a benchmarking experiment. From the results of these experiments, we see that Jiffy performs significantly faster than Andrew in almost all experiments. We conclude that there

are several reasons for these outcomes. Primarily, the caching mechanism in Jiffy is located in the process space of the client application, and the physical location of servers.

Our analysis of creating applications with Jini begins with a background section introducing the Andrew File System and the Jini Toolkit. Next, we describe our problem approach, followed by a description of the architecture of the Jiffy system. Finally, we describe the results from our analyses of Jiffy and Jini and conclude with an overview of future work that can be done.

2 Background

Before looking at the details of the Jiffy implementation, it is first necessary to take a look at the technologies upon which Jiffy builds. These two technologies are the Andrew File System and the Jini Toolkit.

2.1 Andrew File System

Developed at Carnegie Mellon University, this distributed file system has become the largest deployed distributed file system, largely because of its ability to scale to thousands of users [13]. The scalability of AFS can be attributed to two features that reduce network traffic with the file server: the separation of file system branches into volumes located on separate volume servers, and the ability to cache and use the entire contents of a file unless contacted by the server [5]. This whole-file caching can be done because AFS uses a consistency semantic called 'private copy until close', also called session semantics. This semantic specifies that if multiple users on separate machines are concurrently writing to the same file, writes will occur to a private cached copy of the file. Therefore, each user's copy of the file will differ from the copy stored at the server, each user will not be able to tell that another user is writing to the file, and the server will store the last version of the file that it receives from the writers. However, if two users are concurrently writing to a file on the same machine, they will be able to see the other user's changes, because they are writing to the same file in the cache. This was done because the designers of AFS noticed that concurrent write-sharing was done very rarely in real world systems [5].

The Jiffy File System was built to provide the user with the same session semantics as AFS. In Jiffy, multiple users on separate Java Virtual Machines (JVM) [7] concurrently writing to a file will modify private copies of the file because each JVM is using a separate cache. In this way, Jiffy uses session semantics regardless of whether the JVM are located the same or separate physical machines. In both file systems, a cached version of a file can be used as long as the file server has not broken the callback on the file.

2.2 Jini Technology

There are five major services that the Jini Toolkit provides to the application programmer. They are discovery and join, lookup, leases, distributed events, and transactions. Jiffy uses of all of these except the transaction service.

First, the discovery and join service is the means by which applications that wish to use a Jini framework can obtain references to other network applications [8]. For example, in Jiffy, the file server can use the discovery protocol to find the location of the lookup server. It will then join the lookup server, meaning that the lookup server will know the location and kinds of services that the Jiffy server can provide. A Jiffy client will use the same discovery protocol to determine the location of the lookup server.

Second, the lookup service is the means that

Jini provides for applications to find the location of services that they require [11]. In Jiffy, once a client has discovered the lookup server it can query the lookup server if there are any Jiffy file servers available to contact. In this way, the client does not need to know the location of the file server since it will retrieve this information from the lookup service.

Next, the lease facility is the means that Jini provides for servers to grant resources to their clients [10, 2]. Leases are promises that a client will be notified in the event that the server revokes access to the resource. By granting a lease to a client, the client can recognize that it is able to use the resource as long as the lease has not expired and the server has not informed it of a broken lease. This is used in Jiffy to grant clients access to files. The Jiffy clients receive a lease on a file that they are interested in using, and are able to open and write to the file as long as the lease remains valid. Each lease has a finite lifetime configured by the server. This ensures that leases will eventually expire even in the event of a client crash.

Finally, the distributed event facility in Jini allows a server to alert a set of clients on the occurrence of a particular event [9]. Upon event notification, the clients handle the arrival of the event. In Jiffy, the server uses this to handle the breaking of a lease. This occurs when another client writes and closes a file. In the provided implementation of event notification in Jini, notifications are synchronous events, meaning that to notify four clients, the server must contact them one at a time. Jini's implementation of event notifications is based on Java's RMI: Remote Method Invocation [3] calls; therefore, event deliveries cannot proceed until the remote method has returned. In our implementation, we improve on this situation by spawning a separate thread to manage the notification and to reduce the time blocking the client.

3 JIFFY FILE SYSTEM

To approach the problem we built a distributed system on top of the infrastructure and programming model offered by Jini. We did this because building and testing a system using Jini is a good way to identify the strengths and weaknesses of deploying distributed systems and services using the toolkit. Our service, named Jiffy, is a lightweight distributed file system service for Jini. With our implementation, we are able to evaluate the performance of the Jini technology. We chose to implement a file system for several reasons. First, file systems are a standard service in a computer system that have a well-defined interface (open, close, read, write, create, delete, etc.). Second, with the exception of some variations in the case of distributed file systems, most file systems, specifically AFS, have well-defined semantics. Finally, a distributed file system service for Java clients and Jini-enabled devices is useful if it does not require many resources in the client.

3.1 Implementation Goals

Our implementation of Jiffy guarantees the client access to files using AFS semantics. It ensures the consistency of the directories at all times. It allows temporarily inconsistent views of files that are resolved on close. The content of the file corresponds to the copy written by the last client that closed the file, not a mix of updates made to the file by multiple clients.

In order to make Jiffy usable it needed to offer acceptable performance. Ideally, the performance should to be similar to that of local storage. However, we were aware that this was difficult to achieve given the fact that this service was built on top of general tools that were not optimized for this purpose. In addition, since the service is executed from Java's intermediate code (bytecodes), it is not optimized for any

particular hardware platform and therefore is unable to get the best execution performance of each architecture. Despite the performance penalty of the bytecode execution, we expect that as processor speeds increase with respect to I/O devices the Jiffy file system will be able to take advantage of these additional CPU cycles.

It is also desirable to keep the resource requirements (CPU and memory) of the client low, so that Jiffy can be used by a wide variety of clients, this is particularly desirable to allow usage by Jini-enabled devices. On the server side, the small footprint requirement can be relaxed, allowing the servers to have more resources than their thinner Jini device clients.

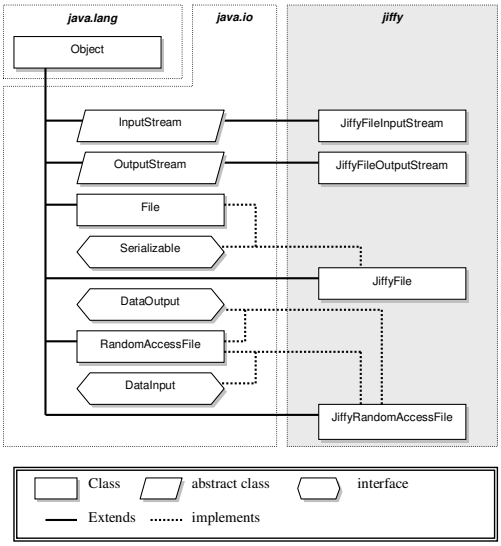

Figure 2: Jiffy API

3.2 Jiffy Implementation

Because one of our goals was to use as many Jini-provided services as needed, we chose to implement an approach that takes the most advantage of the Jini infrastructure and still provides the desired semantics.

Ideally, the clients should be able to bootstrap and access Jiffy with minimum initial knowledge of the service. Jiffy is implemented as a Jini service that clients can locate using the lookup service and the discovery protocols provided by the Jini toolkit. In our implementation the client program loads a set of Java classes into its JVM in order to interact with the Jiffy Server.

For the objects in the file system, we rely on the name space provided by server's local file system at the directory and file level. To avoid confusion with a client local file system, the client must specify a prefix such as "/jiffyfs" to access files on our server.

Each client maintains a local cache that is managed by a Jiffy client component that runs in its address space. The Jiffy client coordinates requests and forwards them to the Jiffy service provider. Our current implementation has one non-replicated server.

3.3 System Architecture

To show precisely the layout of our Jiffy project it is necessary to specify the architecture that we used in the project. Figure 1 depicts the System Architecture and the relationships between the components.

There are seven main components in the Jiffy client. They are the Client Application, the Jiffy Client API, the Client Coordinator, the Cache Manager, the Jiffy Service Interface, the Callback Listener, and the Local Cache. The Jiffy Server is composed of four main components the Jiffy Server, the Callback Manager, the I/O Interface and the File System. An additional component is the Jini toolkit that is used by both client and server.

In order to describe the architecture of the Jiffy File System, we will describe how the operations and primitives implemented by the Jiffy API complete their task. The Jiffy API consists of the following functions Open, Close, Read,

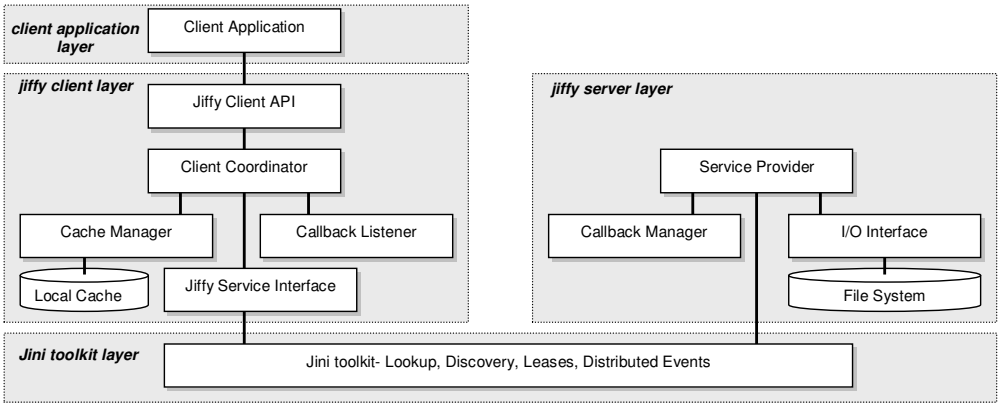

Figure 1: Jiffy architecture

Write, Seek, Stat, Mkdir, Delete, Rename, and Cache Validation. In the descriptions of the API below, the names of objects are italicized, while the names of operations are in a fixed width font. Figure 2 depicts the Jiffy client API.

It is important to notice from Figure 2 that the Jiffy Client API implements the same interface as the Java 2 API Platform Specification [4] for interacting with files. In order to achieve portability among various architectures the Java 2 specification does not allow the creation of hard or soft links.

Open File: A file is opened implicitly when the client program creates a new stream object. The type of stream object created depends on the desired operations to the file. For instance, `JiffyFileInputStream` provides read access to the file, `JiffyFileOutputStream` provides write access to the file, and `JiffyRandomAccessFile` object is used in the case of random accesses to the file. The stream object passes the requests from the client application to the `ClientCoordinator` object. The function of the client coordinator is to multiplex and coordinate the requests of multiple threads in a single client application. The coordinator checks with the `CacheManager` if a valid version of the file already exists in the client's cache, in which case that file is used for the session and further operations. If the file is not in the cache, then the coordinater contacts the server through the JiffyServerInterface using RMI. The Service Provider receives the request, looks up the file, and uses the I/O Interface to read the file from the storage facility. The Service Provider then registers the client/file tuple with the `CallbackManager` and creates a `Lease` object that is passed back to the client along with the entire contents of the file. The lease object serves as a callback promise that the server gives to the client to ensure that the server will notify the client in the event that the file is modified, as long as the lease has not already expired. The client stores the content of the file in its cache for further operations and passes the Lease to the Callback Listener, which maintains a collection of all the Leases held by the client. This mechanism allows the client to access the file later without contacting the server if the lease is still valid.

Notice that because Jiffy follows the file I/O API described in the Java 2 specification [4], it is possible for a client to get a reference to a file without receiving the contents of the file sent from the server. The `JiffyFile` object is used as a container for the metadata of the file or directory. In this way, a user can examine the metadata without having the contents of the file. This information is also stored in the cache by the Cache Manager.

Read: The read operation is done through the read method on either the `JiffyFileInputStream` or `JiffyRandomAccessFile` objects. When this method is invoked, the stream object contacts the Client Coordinator to access the file in the local cache. The request can be satisfied without contacting the server, requiring only access to the local storage since the file was retrieved on the open operation. This speeds up the response time for the operation.

Write: The write operation is provided by the write method in the `JiffyFileOutputStream` or `JiffyRandomAccessFile` objects. When this method is invoked, the stream object contacts the Client Coordinator to write the data passed by the application. Again, this operation is carried out without contacting the server because the write actually occurs to a local file stored in the client's cache. This speeds up the response time for the operation.

Seek: The seek operation is used by the skip method in the `JiffyFileInputStream` and by the seek method in the `JiffyRandomAccessFile` object. The seek is an operation performed on the local file object stored in the cache, without contacting the server.

Close File: The close operation is either called explicitly by the application through any of the close methods in the stream objects or when the stream object is garbage collected. The close request is passed to the Client Coordinator, which checks whether any other writer client has this file open. If there are no other writers to this file, the file is sent to the server. Upon receiving the file, the Service Provider stores it on the persistent media through the I/O Interface. Then, through the Callback Manager, the server sends a broken lease notification event to any clients that have a lease on this file. The Callback Listener on the clients will receive these notifications. If the client does not have the file open, the file is marked as invalid by the Cache Manager, and is evicted from the cache. Otherwise, the stale version of the file remains in use until it is closed in the client, at which time it is then evicted from the cache. However, if this file was opened for writing then upon closing, it would be sent to the server as the latest version.

Stat: The stat operation returns the meta-information of the file object. This information is accessed through the `JiffyFile` object. When a new `JiffyFile` object is created, the Client Coordinator checks if the meta-information for the file or directory is in the local cache and returns it. If the information is not in the local cache or is marked as invalid, then the Client Coordinator contacts the Service Provider and retrieves the file or directory information from it. The server returns the meta-information along with a lease on the object. This lease is passed to the Callback Listener, which adds the lease to its collection of leases.

Make Directory: The mkdir method in the `JiffyFile` object creates new directory in the file system. This method passes to the Service Provider the name of the directory to be created. The server then attempts to create the directory using the I/O Interface and returns the result of this operation to the client. If the server succeeds in the creation of the directory then the Callback Manager invalidates any leases on the parent directory and notifies the clients that have a lease on the parent directory.

Delete File/Directory: The delete method in the JiffyFile object removes the specified file or directory from the file system. If the object is a directory, then it must be empty for the deletion to succeed. The Client Coordinator sends the request to delete an object to the Service Provider. The server uses the I/O interface to perform the appropriate checks, attempts to delete the object and returns the result to the client. The server's Callback Manager sends notification events to any clients to invalidate the deleted object and the parent directory

Rename File/Directory: The renameTo method in the JiffyFile object renames the target object or moves it from one directory to another directory. When this method is called, the request is sent from the Client Coordinator to the Service Provider. The server will use the I/O Interface to rename the object and return the result to the client. The server's Callback Manager sends several notification events to the clients invalidating the renamed object, the directory where the object was before the operation (source directory) and the directory where the object was after the operation (target directory).

Client Start-up: Every time a client application initializes it must validate the data in the cache maintained by the Cache Manager. The client sends a list of the objects that it has in its cache along with a version identifier for each object. The server then checks the state of those objects in the server and sends back Leases for the objects that are still valid. The client passes the valid Leases to the Callback Listener and informs the Cache Manager to invalidate and evict all other remaining objects in the cache. Notice that no file content transfer takes place at this stage.

4 EVALUATION

Our goal was to evaluate the Jini Toolkit from two perspectives. First, we wanted to determine the overall performance of Jini under a variety of workloads and with multiple clients. The second perspective through which we wanted to evaluate Jini was in terms of the ease of use of the toolkit, this was done through our hands on development with the toolkit. In this evaluation, we were interested in whether Jini succeeds as a tool aiding the rapid development of a distributed application.

4.1 Methodology

To evaluate the performance of the Jini toolkit we performed a series of experiments to compare the performance of Jiffy against AFS. We executed these experiments with up to eight simultaneous clients to discover the change in performance caused by the increase of simultaneous requests.

To carry out each experiment, we use Java client programs instrumented to collect the appropriate metrics. To access AFS and the local file system, the client programs used standard Java classes to execute the appropriate system calls. The OS redirected the calls to the target file system. To access the Jiffy file system, the client programs used classes that redirected the calls to the Jiffy server.

Although we are confident that our testing methods were sufficient to compare AFS and Jiffy, it is necessary to examine the factors that could have influenced the results that we obtained. Primarily, the implementation choices made in the design of each file system affected the performance of the system.

An important factor to be considered when evaluating AFS and Jiffy are the goals of each system. Jiffy is a proof of concept system designed as a lightweight service that could reside on a Jini-enabled device. AFS, on the other hand, is production system designed to be a robust, scalable file system that is capable of serving tens of thousands of users. These design goals lead to three important differences in implementation. First, AFS enforces security using Kerberos to guarantee the privacy of file system, and although this is a necessity in a large shared environment it also introduces overhead that may have detracted from the performance of the system. The current version of the Jiffy File System provides no such service to its clients.

4.2 Experiment Setup

The client used in the experiment were 550 MHz Pentium III processor machines with Ultra SCSI hard disks running version 2.0.36 of the Linux kernel. The client machines were connected using a 100 Mbit Ethernet network. We used the Linux JDK 1.2 pre-release v2 from Blackdown to run the test clients. The load generated by each client machine in this setup corresponds to the load generated by multiple Jini enabled devices.

The AFS and Jiffy server ran on Sun SPARC machines, located in a different ethernet network segment.

4.3 Microbenchmarks

This experiment measured the latency of creating a single directory, creating a zero-length file, reading a zero-length file with cold and warm cache, and deleting a file or a directory. The goal of this experiment was to compare the additional delay introduced by the remote procedure calls of both systems.

	AFS (ms)	Jiffy (ms)	Local (ms)
Create dirs	16.72	30.29	0.47
Create files	13.16	30.74	0.22
Read Warm	1.94	0.57	0.52
Read Cold	8.72	41.86	0.59
Delete	11.39	10.35	0.26

Figure 3: Microbenchmark results

Figure 3 shows the average time in miliseconds to complete each of the different operations. It is clear from Figure 4 that the RMI calls used in Jiffy and Jini introduced significant overhead compared to AFS's RPC2 [14]. The time to create directories and files is double that of AFS, while the time needed to read a zero-length file from the server is slower by a factor of five. The additional overhead introduced by RMI can be attributed to several factors including the TCP connection establishment, TCP slow start mechanism, and the marshalling protocol used by RMI.

RMI uses TCP as its transport protocol affecting the way RMI performs. When a remote method is invoked it may be necessary to establish a new connection with the server, incurring the additional cost of connection setup. Additionally, the necessary communication for the method invocation proceeds while TCP is in the slow start phase [6] preventing the full utilization of the available bandwith.

The Java Object Serialization is the protocol RMI uses to marshall and unmarshall procedure arguments and return values of remote method invocations. The marshalling/unmarshalling process is done based on runtime information rather than staticly linked information. This makes the marshalling mechanism very generic and flexible, however this slows down the process.

Figure 4 shows that Jiffy performs extremely well reading the files from the cache. In this case there's no cost associated with remote method invocations to contact the server. Since the client is located in the same Java virtual machine of the client application, the context switch to contact the local clerk is also avoided.

4.4 Read-only Throughput

To determine how the file size affects the performance of the system we performed each experiment with three directory trees. Each directory tree consisted of 25 MB of data of varying file sizes. For the remainder of our testing scenarios we used the source trees described in Figure 5.

To measure the read-only throughput each client read the target tree completely. The experiment was performed with cold and warm cache. Although the file transfer rate was dominated by the network bandwidth, it was de-

	Total Size	Files	Dirs	Avg. Size	Std. Dev
Small Tree	25 MB	1250	50	20.48 KB	50.4 KB
Medium Tree	25 MB	50	5	512 KB	287.4 KB
Large Tree	25 MB	10	1	2560 KB	385.6 KB

Figure 5: Source Trees.

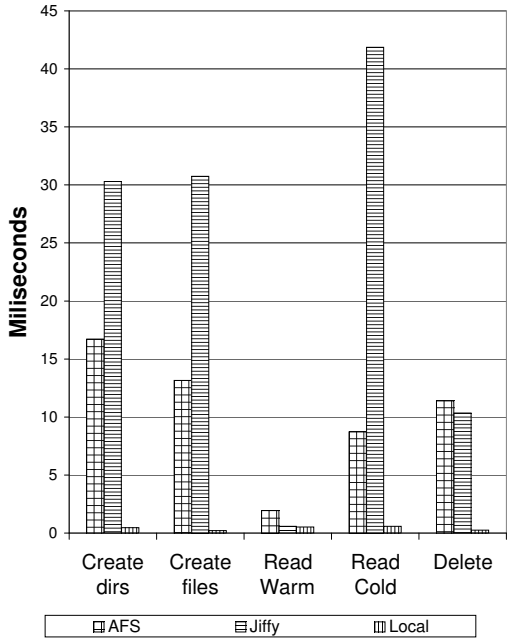

Figure 4: Microbenchmarks

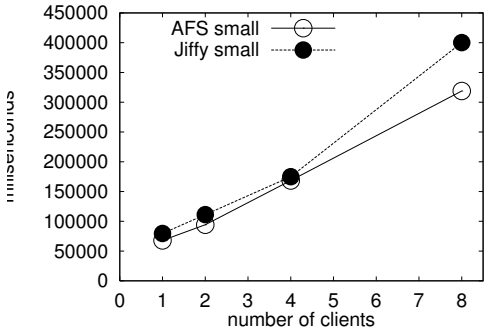

Figure 6: Read with cold cache

sirable to compare the additional overhead introduced by the communication mechanism of each system when performing a bulk transfer.

Figure 6 shows that Jiffy introduced substantial overhead when accessing small files in the cold cache scenario. As the number of client requests to the Jiffy Server increased, the overhead introduced by RMI's marshalling mechanism becomes more evident.

Figure 7 show the transfer time for medium and large files with cold cache. In this situation it is important to note that many implementations of RPC, and most streaming protocols, use UDP as ther foundation in order to avoid both the expense of setting up the network connection and TCP stack's flow-control process. However, in this situation RMI did not intro-

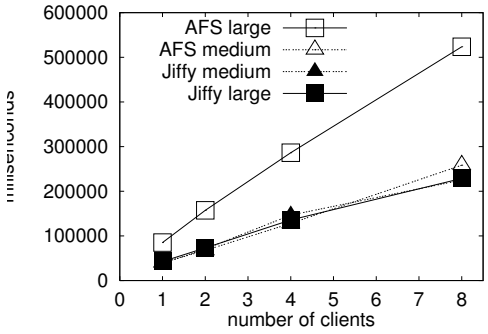

Figure 7: Read with cold cache

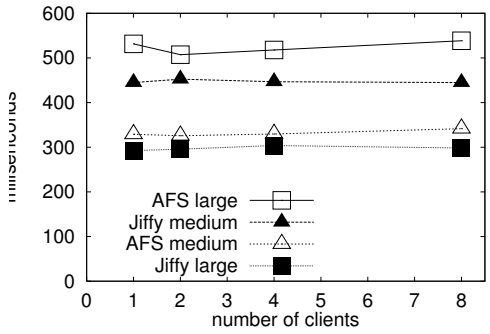

Figure 8: Read Medium and Large tree with warm cache

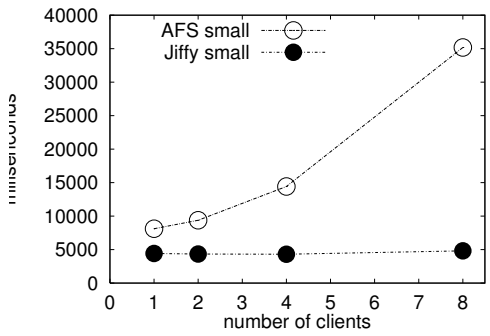

Figure 9: Read Small tree with warm cache

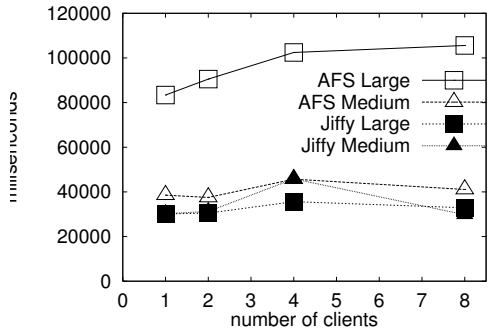

Figure 10: Write latency in the precense of multiple reader

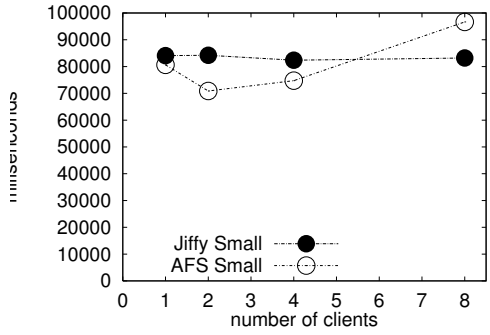

Figure 11: Write latency in the precense of multiple reader

duce additional overhead to perform the bulk transfer. In these cases Jiffy was able to amortize the cost of the initial method invocation over over the cost of the whole file transfer to improve throughput.

We also measured the read-only throughput with warm cache of the two systems. The performance of the two systems is comparable for medium and large size files (See Figure 8.

Figure 9 shows the elapsed time to complete the read the small file directory tree. Directory lookup could be the cause for the slowdown experienced by AFS, requiring the client to contact the server multiple times to access the file. Jiffy caches the directory information, speeding up the operations.

4.5 Callback Mechanism

This experiment tested the write throughput and scalability of the callback mechanism. In this experiment multiple clients read the same target tree on the server. After these clients finished reading the tree, one client then modified all the files in the target tree. The experiment measures the additional latency experienced by this writer due to the callback mechanism.

Figures 10 and 11 depict the results from this experiment. The expected time to complete the writer's request did not increase significantly in AFS as the number of clients that needed to be notified increased. Figure 11 shows that the callback mechanism slows down the writer in the Jiffy system as the number of clients and requests increase. Jiffy uses Jini's leases and

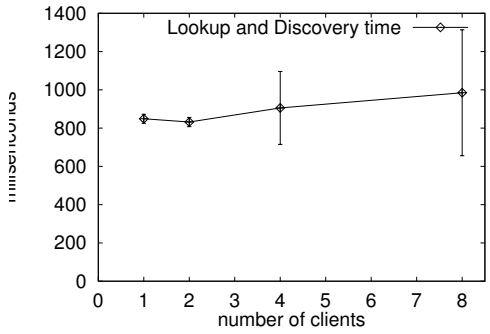

Figure 12: Lookup & Discovery time

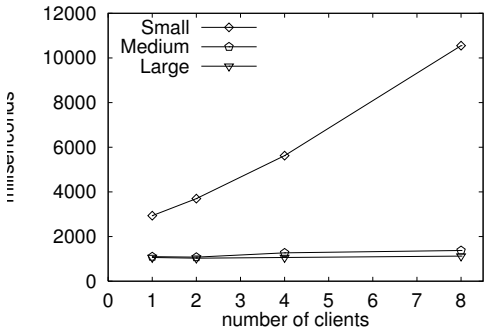

Figure 13: Cache validation

distributed event services to implement its callbacks. This two services are implemented on top of RMI. This means that a new RMI connection may have to be setup to notify the client. RMI's implementation tries to minimize this penalty by reusing established connections. In short, Jiffy's callback mechanism does not appear to be as scalable as Andrew's, despite the optimizations that are performed in the RMI code.

4.6 Lookup and Discovery

This experiment was designed to measure the initialization cost of the Jiffy client startup and test the Jini Lookup service. In this experiment, we measured the time it took to discover the name server and lookup the Jiffy server as the number of clients increased. The time to revalidate the cache was also measured.

From these results shown in Figure 12, we can see that Jini's lookup and discovery mechanism did not scale well as more clients made simultaneous requests to the service. Although the expected time to locate the host did not increase dramatically, the variance increased significantly.

Figure 13 shows that the cache validation time of each client was affected by the number of files in the client's local cache. Therefore, Jiffy incurs large initial cost each time a client application starts. However, this initial overhead saves the client and server time in the future. After this has taken place a client does not need to validate a cache entry before using it. This initial overhead could have been reduced further, had our implementation used a strategy, similar to that done in AFS, of validating entire volumes as a single object.

4.7 Ease of Use

From an ease of use perspective, we noticed several positive and negative aspects regarding the current implementation of Jini.

While there were certainly a number of specification documents from Sun available to discuss a high-level view of Jini, many of the toolkit's vital features lacked documentation. This is certainly evident when compared with the documentation available for the standard Java Library. Furthermore, at the time of building Jiffy, there were only single implementations of the Jini toolkit interfaces. Although these were problems with the ease of use of Jini, we expect that as this technology matures, the documentation and implementations will become more robust.

On the positive side, the Java-centric nature of the Jini toolkit allows it to avoid many of the complex platform-centric issues that other distributed object systems must face. Further-

more, while the Jini toolkit's implementations of interfaces appear minimal at first, they in fact allow the programmer to be as flexible or rigid with the interface as desired.

5 CONCLUSIONS

Building a distributed application is a difficult task because the software architect faces the challenge of designing a system that runs transparently on multiple machines. To do this, the designer must implement means of handling many complex issues such as heterogeneous clients, concurrency, fault tolerance, and reliability while preserving the correctness of the system's semantics. This is made even harder since users still demand performance.

Sun Microsystems built Jini technology to address these difficulties. Jini provides a variety of services to the programmer, such as discovery, lookup, leasing, distributed events, and transactions intended to facilitate the rapid development of distributed applications. In fact, the builders of the Jini technology claim that this technology will make applications flexible, resilient, and compatible. In fact, the appearance of Jini caused enthusiasm among Java developers. However, few existing applications have adopted this toolkit because it has been released only recently to the public.

The motivation of this project was to put the promise of Jini technology to the test. We built a distributed file system using the tools provided by Jini. By doing so, we tested the features of Jini in the context of building a conventional distributed application. Since there has been much research on the development of distributed file systems, we looked into implementing some of the standard optimizations while still relying on Jini for the infrastructure. Our system was designed to provide AFS semantics. We then used a set of metrics to compare this system against AFS.

In conclusion, from our use of the Jini toolkit, we have certainly seen room for improvement in terms of ease of use. First, the little amount of documentation is certainly disappointing and frustrating compared to the rich documentation of the Java core libraries. Furthermore, for many aspects of the Jini toolkit, only the interfaces or only a single implementation are provided for these important interfaces. Although providing the interface allows the programmer to be more flexible, a working implementation of each feature is a necessity in a toolkit of this kind. Despite its Java implementation, it does not seem to affect the performance of the Jiffy file system as a whole. It seems apparent that a Java-based file server can provide an adequate performance because the bottleneck is increasingly in I/O accesses while the CPU becomes essentially a free resource. We expect that the performance of Jini will certainly be improved with the advancement of Just-In-Time compiler technologies.

The Jini toolkit provided us with a core set of functionalities and allowed us to focus on the remaining features of the Jiffy File System. The Jini toolkit supplied the tools to quickly build the lookup, leasing, and callback notification capabilities of Jiffy. A testament to this is that over a short few months we have successfully built a distributed file system service that can be used by Jini-enabled devices.

References

[1] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification.* Java series. Addison-Wesley, 1999.

[2] Gray C.G. and D.R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating Systems Principles*, Litchfield Park, Ariz., 1989. ACM.

[3] Troy Bryan Downing. *Java RMI: Remote Method Invocation.* IDG Books, 1998.

[4] J. Gosling, Joy W., and Steele G. *The Java Language Specification*. Addison-Wesley, Menlo Park, California, 1996.

[5] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), Feb 1988.

[6] V. Jacobson. Congestion avoidance and control. In *SIGCOMM' 88*, pages 314–329, Sept 1988.

[7] T. Lidholm and F. Yellin. *The Java Virtual Machine Specification, 2nd Edition*. Addison-Wesley, Menlo Park, California, 1999.

[8] Sun Microsystems. Jini discovery and join specification. `www.sun.com/jini/specs/boot101.ps`, 1999.

[9] Sun Microsystems. Jini distributed event specification. `www.sun.com/jini/specs/event101.ps`, 1999.

[10] Sun Microsystems. Jini distributed leasing specification. `www.sun.com/jini/specs/lease101.ps`, 1999.

[11] Sun Microsystems. Jini lookup service specification. `www.sun.com/jini/specs/lookup101.ps`, 1999.

[12] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S. Rosenthal, and F.D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3), Mar 1986.

[13] M. Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, 18(1), 1992.

[14] M. Satyanarayanan and E.H. Siegel. Multirpc: A parallel remote procedure call mechanism. In *Workshop on Design Principles for Experimental Distributed Systems*, West Lafayette, IN, Oct 1986.