Upgrading Distributed Applications with the Version Manager

Mukesh Agrawal **Suman Nath** **Srinivasan Seshan**

March 2005
CMU-CS-05-117

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Developers and managers of distributed systems today face a difficult choice. To provide for the evolution of their systems, they can either require that the software running on all nodes is inter-operable, or they can shut down the entire system when an upgrade is required. The former complicates the life of system developers, while the latter annoys users. We propose Version Manager, a middle ground between these approaches: an upgrade framework that limits the the extent of support required for interoperability, while also providing availability during upgrades. We evaluate the feasibility of our approach by applying it to two existing distributed systems: the Cooperative File System [11], and IrisLog [3]. Our proposal enables the upgrade of these systems while maintaining system availability, and without requiring code modifications.

This work was supported by the Army Research Office under grant number DAAD19-02-1-0389.

Keywords: Distributed systems, upgrades

1 Introduction

Many networking researchers have bemoaned [10] that the difficulty of upgrading the Internet infrastructure has lead to the ossification of both Internet research and Internet functionality. Several promising technologies, such as IP Multicast, IntServ, DiffServ, RED, and Fair Queuing, have all failed to be widely deployed at least partly due to the difficulty of changing or upgrading router functionality.

Recently, researchers have proposed a number of new application-layer infrastructures for distributed applications [21, 19], distributed storage [11, 17, 14], and sensor systems [3]. However, these infrastructures may share the Internet's weakness since few (if any) of these infrastructures accommodate easy upgrades. In this paper, we explore techniques that simplify the upgrade of any distributed application, including these critical application-layer infrastructures.

With respect to upgradability, the fundamental design choice faced by developers of distributed applications is whether to require versions to interoperate directly, or to require that the application be shut down as part of the upgrade process. However, we believe that a better alternative is available for many cases.

In our approach, we view different versions of an application as separate deployments hosted on the same physical infrastructure. Two versions of an overlay network service, for example, are treated as separate overlay networks. Interoperability is managed by gateway proxies, which translate incoming requests between versions.

The key challenges in this design are preventing the different versions on a node from interfering with each other, and providing users and services unmodified and uninterrupted access to the infrastructure. We solve these challenges by using virtual machines to isolation application versions, and by providing a message router to manage communication between nodes. The message router also manages distribution of new software code, and data migration between versions.

In this paper, we argue for the general applicability of our approach, by demonstrating its ability to support upgrades of both the CFS distributed storage system [11] and the IrisLog distributed infrastructure monitoring system [2]. These recently proposed distributed infrastructures differ substantially in their designs. For example, the CFS design relies on an underlying DHT to organize and store write-once data, whereas, IrisLog uses an underlying tree-based structure to store read-write sensor data.

Our experimental results with this service show that with our parallel execution approach, we are able to upgrade an application without disrupting availability. In addition, simple applications incur a 6% increase in response time when simultaneous versions are not executing. More complex applications such as CFS and IrisLog incur $\approx 100\%$ and $6-36\%$ overhead respectively. However, we believe this overhead could be easily reduced by switching our choice of VM from User-mode Linux to a more efficient system such as [6, 24].

Finally, we are able to propagate new versions into a system relatively quickly – ranging from 2.3 minutes to propagate a 9 MB update across 3 nodes, to 7.1 minutes to propagate 9 MB update across 12 nodes.

The remainder of this paper is structured as follows: we formally define the problem in Section 2. In Section 3, we discuss the challenges of the upgrade problem and present a high level design. Section 4 explains the architecture of our upgrade system. In Section 5, we describe two existing applications which we have modified to work with our system. Section 6 presents experimental results about the performance of our system. Sections 7 and 8 discuss related work and future work respectively, and Section 9 concludes.

2 The Software Upgrade Problem

As noted in the introduction, one approach to upgrades is to have the application provide for upgradability directly. Herein, we consider a simple design template for providing upgradability, and then discuss the evaluation of the design template to our example applications.

2.1 Design Template

We begin with three brief definitions. In defining the terms, we assume that we are upgrading a distributed application from version v_1 to version v_2.

Interface Compatibility v_2 is *interface-compatible* (IC) with v_1 if all the messages correctly interpreted by v_1 are also correctly interpreted by v_2. Furthermore, v_2 must be able to generate replies that can be correctly interpreted by v_1.

Message Equivalence v_1 and v_2 are *message equivalent* (ME) if, for any user request, they both generate the same number of messages between any two hosts and in the same causal order. Intuitively, two ME versions have similar message exchange patterns (the contents of the messages may be different).

k-fault tolerant A system is k-fault tolerant if it maintains service in the face of k node failures.

These terms outline an obvious approach that application developers might take to providing upgrade support. Namely, if two applications versions are IC and ME, and the system is k-fault tolerant, we can upgrade the system by crashing k nodes at a time, upgrading them to the new software release, and bringing them back online.

2.2 Applicability to CFS and IrisLog

CFS implements a filesystem using the Chord [21] distributed hash table (DHT). A DHT maps keys to value using an overlay network. Clearly, changing the hash function – the mapping of a key to its location – would result in a non-message-equivalent upgrade. More abstractly, any change to the routing of messages in the overlay would violate message equivalence.

IrisLog is a network monitoring application based on IrisNet [3], a hierarchical and distributed XML database. The initial version of IrisLog did not support caching. However, later versions added caching support. Unfortunately, caching also changes communication patterns, rendering such upgrades non-message-equivalent.

We find that this design template for upgradability, while appealing in its conceptual simplicity, is incapable of supporting reasonable upgrades we might want to make in these recent distributed application infrastructures.

3 High-Level Design and Challenges

As explained in the previous section, simple application-based approaches to providing upgradability severely constrain the type of upgrades that are possible. On the other hand, requiring all applications without built-in upgrade support to be shut down entirely, and be unavailable for the entire duration of an upgrade is untenable.

Our design takes inspiration from how Internet routing has been upgraded. In moving from IPv4 to IPv6, the Internet has allowed both protocols to operate simultaneously on nodes. Similarly, we begin an upgrade by running multiple versions of an application to run simultaneously on a single node. The simultaneous execution period allows the system to deploy and switch to the new version while still providing availability to the previous version. After the new version is ready to run, we terminate the old instance.

3.1 Simultaneous Execution

While the different versions run on the same set of nodes, our version management approach treats different versions of an application as separate deployments. For example, for an overlay network application, different versions of the application are conceptually separate overlay networks. This is necessary, because, although the versions run over the same set of nodes, differences in routing between versions may generate different overlay topologies[1]

To enable two versions of an application to run simultaneous on the same node, we must carefully isolate the versions from each other. The isolation environment needs to provide isolation in two dimensions: intra-node, and inter-node. Intra-node isolation is required to support applications which are structured as multiple processes, possibly communicating via IPC to well-known addresses (e.g. `/tmp/.X11-unix/X0`).

[1] Note that even the same routing protocol with different parameters – such as link failure detection timer, or random seed for neighbor probing – might generate different topologies.

Inter-node isolation is required to prevent version 1 on node A from erroneously communicating with version 2 on node B (and vice versa). We implement this isolation through the use of virtual machines and careful network message routing, as detailed in Secs. 4.

3.2 Upgrade Propagation

Before a node can run a new version, it must learn that the version exists, and obtain the code and related files for that version. We handle the former task by piggybacking advertisements of running versions on the application's communication. To obtain the new version, a node simply retrieves it from any other node that has advertised the version.

3.3 Non-Participation

As a practical matter, we cannot expect that all nodes associated with a service will run our version management system. This may be due to concerns about overhead, or simply because some nodes are beyond our control. In particular, while we expect that nodes providing services such as file storage or distributed sensing will use version management, we do not expect the clients of these systems to run version management as well.

To accommodate this reality, we advocate that clients access services through a narrower *gateway* interface than servers use to communicate with each other. The application developer then provides, with each software release, a gateway proxy that can translate the gateway interface requests to accommodate clients which are running different versions than the servers. As we will see in Sec. 5.1, such an architecture is already employed by distributed applications.

3.4 Application-Specific Functionality

While we have enumerated many application-independent aspects to version management, some tasks remain application dependent. For example, most applications will require that persistent state available in the currently running version of the application is also available in any new versions. However, this state migration will likely differ significantly between applications. We handle these cases by providing hooks which call in to the application, as we detail in Sec. 5.

In the remainder of the paper, we describe our resolution of these challenges, and our experience in applying our solution to the example applications.

4 The Version Manager System for Upgrades

As outlined in Sec. 3, to enable simultaneous execution, we must provide intra-node isolation and inter-node isolation. Additionally, to propagate upgrades, we need mechanisms for disseminating information about new versions, and distributing the new software. Here, we explain our implementation of each of these functionalities.

4.1 Intra-node Isolation

To provide intra-node isolation between simultaneously executing application instances, a number of well-known isolation techniques might be used. We begin this subsection by considering several possibilities, and identifying their limitations. We then detail our chosen isolation technique: the User Mode Linux virtual machine.

As a first strawman, we consider using separate processes, possibly using `chroot()` environments to isolate their filesystem namespaces. However, such a solution is deficient in two respects: it does not provide isolation of network resources (in particular TCP and UDP ports), and it does not provide support for multiple users. These deficiencies cause two difficulties. First, the failure to isolate ports means that the

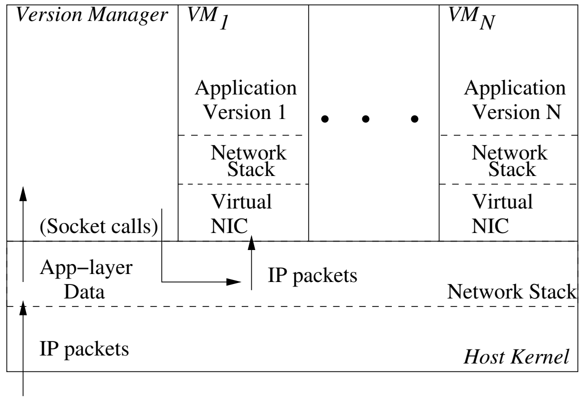


Figure 1: Routing of messages between the virtual machines and Version Manager. The diagram shows data flow for incoming requests. The flow for outgoing replies is simply the reverse.

different application instances must use different ports. Second, application that rely on the use of multiple userids, such as to isolate privileged operations, need to be modified.

A more promising possibility is to use BSD `jail()` environments, or Linux *vservers*. These facilities provide support for multiple users inside an isolation environment, as well as support for isolation of network traffic. While these facilities would be sufficient for some applications, they are limited in one important respect: the IP address seen by processes running inside the jail/vserver differs from the IP address seen by processes running outside isolation.

The fact that the isolation environment can not use the same IP address as native processes raises a complication. Namely, unless the host can dedicate routable address to application versions, the host must use network address translation (NAT) to provide connectivity between isolated processes and the public network.

As a consequence of this use of NAT, the application running inside the jail or vserver will believe it has one address, while its communications peers believe it has a different address. This disrupts the functioning of applications which communicate their IP address to other nodes. Note that even if the host can dedicate routable addresses, the fact that IP addresses differ between versions may require application reconfiguration.

To support a broader range of applications and deployment scenarios, we chose the UML VM[13] as our isolation environment. UML provides the illusion of an independent Linux kernel while running as a process on some other machine. In this context, the illusory machine is referred to as a *guest*, and the real machine is called the *host*. Access to network communication for all VMs is handled by a single process, called the *Version Manager* [2] in the host machine. An example of this high-level design is shown in Figure 1.

Having decided to use UML to provide intra-node isolation, we must provide for network connectivity between the application (running inside the virtual machine), and the public network. UML provides each guest VM access to network communication by using a virtual network device (a *tap* interface) to exchange packets with the host kernel. In a typical configuration, the host is configured to forward these packets to the Internet using standard IP forwarding.

In order to successfully provide isolation for the application instances, as well as the illusion that the application is running natively on the host system, we must resolve the problem of IP address conflicts. Recall that we want the ability to run unmodified applications inside the isolation environment. That is, the same code and configuration can be used both natively on the host, and also (unchanged) inside the VM. Accordingly, we want applications running inside the VM to believe the VM has the same IP address as the host.

Unfortunately, simply configuring the VM with the host's IP address does not work. The reason is simple:

[2] Note that while both "virtual machine" and "version manager" can be abbreviated as VM, we always use VM to mean virtual machine.

network messages sent by applications inside the VM will be sent with a source address field set to the host's IP address. But replies to these messages are first seen by the host's network stack. The host's network stack, recognizing the destination address of the replies as its own address, attempts to deliver the data to processes running natively on the host, rather than forwarding the packet to the VM.

To solve this problem, we first take advantage of aliased interfaces inside the VM. Aliased interfaces provide a way to configure a single network interfaces with multiple IP addresses. We use aliases to configure the VM's tap interface with two addresses. One is a private address, which enables communication between the host and the VM. Note that we must ensure that all packets sent from the VM to the host (whether destined to the host or to the public network) are sent with the source address field set to the VM's private address.

In addition to the private address, we also configure an alias with the host's IP address. The existence of this aliased address permits applications inside the VM to `bind()`, or `listen()` on the host's IP address, as they would if they were running natively. However, as noted above, we must ensure that despite a `bind()` to the public address, packets are still sent with the source sent to the private address. Similarly, we must ensure that incoming connections, which will have their destination address set to the VM's private address, are delivered to processes `listen()`-ing on the host's IP address.

We can meet both of these needs using NAT inside the VM. Specifically, we configure the VM's network stack to translate packets that are destined to the VM's private address, so that the packets are delivered to its public address instead.

Note that while we have resolved the IP address conflict problem, we still need to provide connectivity between the VM and the public network. And our solution must ensure that communication peers of the isolated application see the same address as the application itself sees. Namely, the communication peers must see the source field of the packet header set to the host's IP address. Contrary to the situation with jails and vservers, however, we can actually use NAT to help us. Specifically, we configure the host's network stack to translate packets coming from the VM and going to the public network so that the source address is that of the host (and analogously for incoming packets).

While this configuration meets many of our goals, one further problem arises. We detail and resolve this problem in the next section.

4.2 Inter-node Isolation

The inter-node isolation problem arises from the fact that the upgraded version of an application will use the same TCP or UDP port as prior versions. Thus, when a packet arrives from a remote system for the application's port, it is not clear to which version of the application the packet ought to be delivered.

To resolve this contention for transport-layer ports, we interpose a transparent proxy on the communication between the application and its peer nodes. To guarantee that application traffic is routed to correct versions at peers, we prepend a header to each outbound request, identifying the version number of the sender, and the maximal version running on the sending host. The sender version field of this header is examined and stripped by the proxy on the peer node, which routes the request to the appropriate application version. To accommodate the fact that the application may also communicate with services that do not use version management (e.g. public web servers, mail servers, etc.), the proxy does not interpose on such services.

In order to facilitate this differentiated treatment of managed and unmanaged traffic, we require the application to register the network ports used to implement the application's protocols. This is accomplished with a simple protocol similar to *portmap*. Rather than modifying applications to implement this protocol, however, we have written a simple stand-alone tool for registering the application's port number. The tool is run as part of the bootstrap process inside the isolation environment.

Given these approaches to transport stack and port conflict handling, the overall data flow for incoming requests is illustrated in Figure 1.

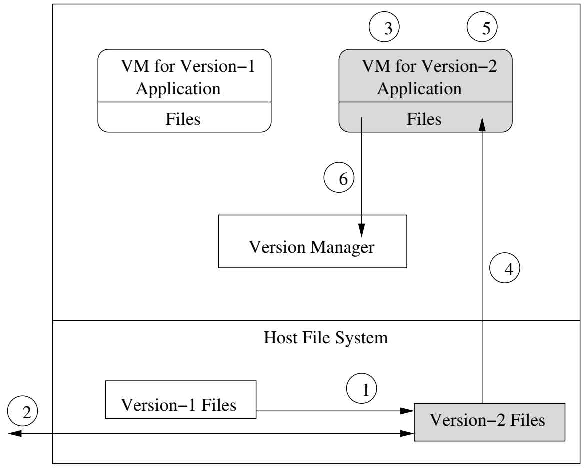


Figure 2: Steps of upgrading a host with a new version of an application. The shaded components are specific to Version 2 and are installed after the query for Version 2 was received.

4.3 Version Propagation & Control

Because every request contains a version header indicating the maximal version running on the sending host, any time host A communicates with another host B, A will learn if B is running a newer version[3]. If B is running a new version, then A creates a copy of the application-specific portion of the filesystem, synchronizes this copy with the files on B, and boots the application inside the isolation environment.

This approach imposes two requirements. The more obvious of these is that application-specific files must be identified and separated into a subtree. The less obvious requirement is that node-specific state (e.g. configuration files containing the node's IP address) must also be kept outside of the application tree, so that it is not overwritten with information from the peer node. However, this latter requirement is in line with existing practice. For example, configuration files are usually kept in `/etc` on Unix systems.

Once a new version is identified, the Version Manager first copies the set of files related to the most recent version the host already has (step 1 in Figure 2), and then performs a `rsync` between these newly copied files and the files of the target newer version on a remote host (step 2). These operations are performed within the host OS filesystem. Much like active network implementations [23], we choose the originator of the higher version message as the remote host to perform the `rsync` with, since this node is guaranteed to have had recent access to this code. Note that other strategies to choose the remote host are possible. Once the `rsync` is complete, the host has the application files required for the new version of the application.

At this point, the Version Manager executes a new VM (step 3), and copies the files required by the new version from the host file system to the VM (step 4). Finally the new version of the application is run inside the VM (step 5) which then registers itself to the Version Manager (step 6). The version manager keeps track of the ongoing local upgrades and, thus, avoids creating multiple instances of the same version of the application.

4.4 Additional Issues: Availability and Consistency

As noted above, until we have copied the persistent state from the old version to the new, some data will be unavailable in the new version. We believe that, in most cases, this can be remedied by having the gateway resubmit queries that fail in the new version to the old version.

A further concern is incoming requests that modify the persistent state. For most applications, these changes will need to be reflected in all versions. To handle this case, we simply have the gateway submit

[3]We omit the symmetric case for ease of exposition.

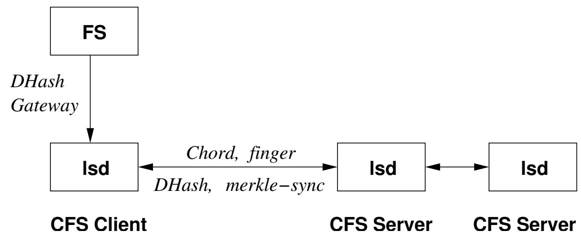


Figure 3: CFS software architecture. Vertical links are local APIs; horizontal links are RPC APIs.

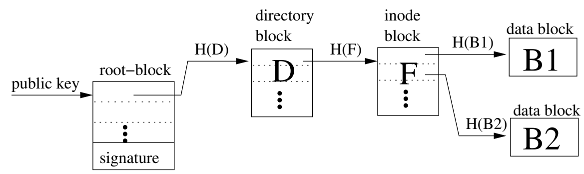


Figure 4: A simple CFS file system structure example. The root-block is identified by a public key and signed by the corresponding private key. The other blocks are identified by cryptographic hashes of their contents.

write requests to all running versions. The gateway returns success to the application only if all versions commit the write.

4.5 Termination

To avoid wasting resources on versions that have outlived their usefulness, as well as to kill off buggy versions, the Version Manager must provide some support for terminating versions. The decision of when to terminate a version, however, is a matter of policy. Some service providers may have little patience for clients running old versions, whereas other providers may be more forgiving.

Accordingly, we focus here on the termination mechanism, rather than the termination policy. We believe that two mechanisms are required. One requisite mechanism is a data reporting mechanism that provides the application service provider with information about the usage of old versions. We believe this mechanism is best implemented by exploiting existing distributed monitoring systems, such as IrisLog[2], Sophia[22], or Ganglia[16].

The other requisite mechanism is the interface for terminating a version. Our current interface is, admittedly, crude. The service provider logs in to a node via ssh and runs a script to terminate the obsolete version. The script also informs the Version Manager that the version has been obsoleted. This is required to handle the case of a newly deployed version, which has been found to be buggy. If the Version Manager did not record that the version had been obsoleted, it might re-install the version from another node.

5 Case Studies

In this section we describe our experience of incorporating our Version Manager with two applications, CFS and IrisLog.

5.1 Cooperative File System (CFS)

CFS implements a distributed filesystem over the Chord DHT. Conceptually, when a user writes a file into CFS, the file is broken into blocks [4]. Each block is stored in the DHT, with the hash of the block's content used as the storage key. The filesystem directory is also stored in the DHT.

All blocks (whether belonging to a file or a directory node) are pointed at via content hashes, with the exception of the root block for a filesystem. A filesystem root is named using a public key from a public/private key pair. The root block is signed using the private key corresponding to the public key under which the root block is stored in the DHT. The use of content-hashes and public-key cryptography provides a self-certifying filesystem. Note that because any modification of a filesystem requires changing meta-data all the way up to the root block, only a holder of the private-key for a filesystem can modify the filesystem's content .

As illustrated in Fig 3, a node participating in CFS typically runs two daemons: the CFS user-space NFS server, and the CFS block server (*lsd*). Additionally, the node runs an NFS client, to mount the filesystem. The operation of the complete system is as follows: when a process attempts to read a file from CFS, the kernel's NFS client issues RPCs to the CFS user-space NFS server. The NFS server in turn requests blocks from *lsd*. *lsd* issues RPCs to Chord peers, to retrieve the requested block from the DHT. After *lsd* receives the block, it returns the result to the NFS server, which replies to the NFS client. The kernel then returns the requested data to the reading process.

Fig 4 diagrams the protocol interactions between *lsd* and other entities. From this diagram, we note that *lsd* communicates with the NFS server using the *dhashgateway* protocol, which is implemented as RPCs over either a Unix socket or a TCP socket. *lsd* communicates with its peers using the *chord fingers*, *dhash*, and *merkle_sync* protocols, typically as RPCs over UDP. The *chord* and *fingers* protocols implement Chord routing, while the *dhash* and *merkle_sync* protocols provide block insertion/retrieval and block availability maintenance in the face of node join/leave [5].

With this understanding of CFS in hand, we proceed to describe how we apply our framework to CFS.

5.1.1 Incorporating Version Manager with CFS

For CFS, we choose to provide version management for the protocols implemented by CFS, but not for the NFS protocol for it is expected to remain stable over different versions of CFS. In the following we describe the tasks required to modify CFS to incorporate our Version Manager.

Identifying application- and host-specific files. CFS runs from a single binary, so identifying its files and keeping them separate is trivial. Its node-specific state consists only of the IP address on which the daemon should listen. We store this information in the boot-time script (stored in `/etc`) which we use to start CFS.

State migration. We have implemented a simple program to copy the blocks from one version to another. The program is implemented in 500 lines of `C++` code using the CFS libraries. The program functions as follows. It uses the *merkle_sync* protocol to query the local server for the current version about the list of blocks stored locally. The copy program then attempts to read the same block from the new version. If the read fails, the copier reads the block from the current version, and writes the block to the new version.

Availability and consistency. Our handling of writes, however, raises an additional concern. The newly initiated writes may conflict with the writes for the copying of data from the old version to the new. Fortunately, CFS already has mechanisms for dealing with write conflicts. Namely, the content of any block written (except a filesystem root block) must hash to the same value as the name (DHT key) under which the block is written. Thus, two writes to the same block with different content are highly unlikely. For root

[4] Physically, blocks are further broken into fragments, which are actually stored on the DHT nodes. The fragmentation of blocks improves fault tolerance [9].

[5] In practice, these protocols are encapsulated in CFS' *transport* protocol, which multiplex/demultiplexes messages between virtual nodes. We omit this detail in further discussion.

block changes, CFS requires that the root block include a timestamp, and that the timestamp is greater than that of the existing block.

While we provide no mechanism for resolving the write conflicts that our upgrade system introduces, we hope that we will be able to reuse the application's existing mechanisms to provide a similar semantics during upgrade as during normal operation, as we have for CFS.

5.2 IrisLog

IrisLog is a distributed network and host monitoring service that allows users to efficiently query the current state of the network and different nodes in an infrastructure. It is built on IrisNet [3], a wide area sensing service infrastructure. Currently, IrisLog runs on 310 PlanetLab nodes distributed across 150 sites (clusters) spanning five continents and provides distributed query on different node- and slice-statistics[6] (*e.g.,* CPU load, per node bandwidth usage, per slice memory usage etc.) of those nodes. At each PlanetLab node, IrisLog uses different PlanetLab sensors [20] to collect statistics about the node and stores the data in a local XML database. IrisLog organizes the nodes as a logical hierarchy of `country` (*e.g.,* USA), `region` (*e.g.,* USA-East), `site` (*e.g.,* CMU), and `node` (*e.g.,* cmu-node1). A typical query in IrisLog, expressed in the standard XPATH language, selects data from a set of nodes forming a subtree in the hierarchy. IrisLog routes the query to the root of the subtree selected by the query. IrisNet, the underlying infrastructure, then processes the query using its generic distributed XML query processing mechanisms. On receiving a query, each IrisNet node queries its local database, evaluates what part of the answer can not be obtained from the local database, and recursively issues additional subqueries to gather the missing data. Finally, the data is combined and the aggregate answer is sent to IrisLog(client). IrisNet also uses efficient in-network aggregation and caching to make the query processing more efficient [12].

Both IrisLog and IrisNet are written using Java.

5.2.1 Incorporating Version Manager with IrisLog

At a high level, IrisLog consists of two independent components: the module that implements the core IrisLog protocol, and a third-party local XML database. We choose to provide version management for the first component, since the latter is expected to be stable over different versions of IrisLog. We here highlight the changes IrisLog requires to incorporate our Version Manager.

Identifying application- and host-specific files. The IrisLog distribution comes with the source code and all the application specific files are organized in a directory tree. The node specific states are all given in a configuration file.

State migration. IrisLog uses a database-centric approach, and hence all its persistent states are stored in its local XML database. Thus, state migration involves transferring the local XML database of the old version to the new version. IrisLog provides APIs for an IrisLog host to copy or move a portion of the local database to a remote host where it gets merged with the existing local database (used mainly for replication and load-balancing purpose). To copy the latest persistent state from an old version, we start the new version with an empty database and use IrisLog's APIs to copy the whole local database of the old version running on the same host to the new version. Note that, unlike CFS, the data is transferred within the same host, within the versions.

Unlike CFS, we do not need to handle the consistency issues for IrisLog. This is because IrisLog's data has single writer (the corresponding sensor), and there is no write conflicts.

[6] A slice is a horizontal cut of global PlanetLab resources. A slice comprises of a network of virtual machines spanning some set of physical nodes, where each virtual machine (VM) is bound to some set of local per-node resources (*e.g.,* CPU, memory, network, disk).

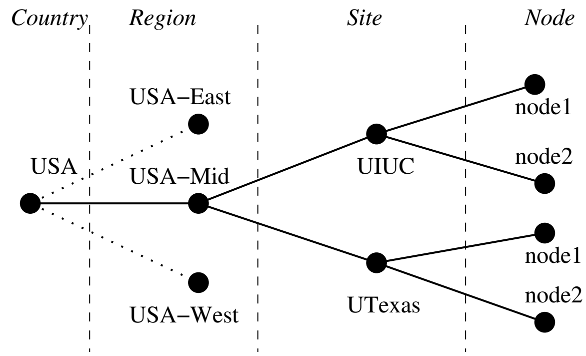


Figure 5: Part of the hierarchy used by the IrisLog Evaluation. Dashed lines indicate subtrees that are omitted for clarity.

6 Experimental Evaluation

In order to ascertain the viability of our approach, we used two sets of experiments. The first set of experiments measured the overhead of running applications inside the UML VM, while the second set of experiments demonstrated the upgrade overhead and its impact on the normal operation of the applications. We present these two sets of results after we describe our experimental setup.

6.1 Experimental Setup

We conducted all our experiments on Emulab [1] using a set of 850 MHz Pentium III machines with 512 MB of RAM. The machines were running RedHat 7.3.

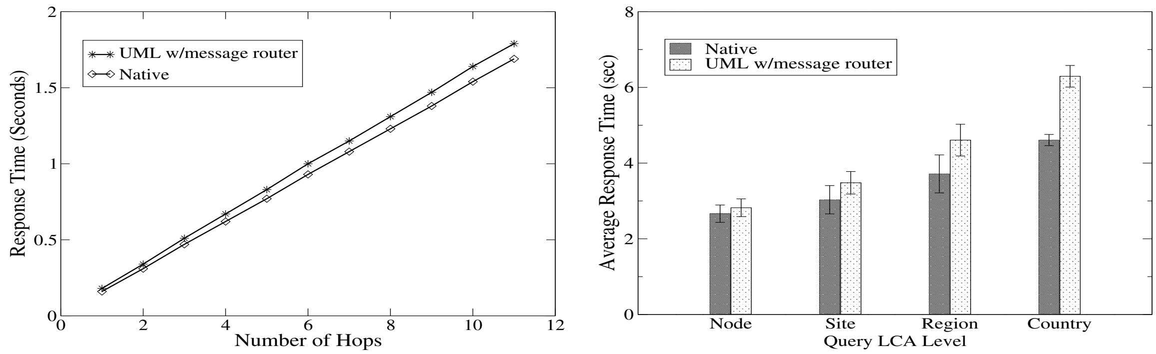
We used three different distributed applications to run on the Version Manager. The applications were chosen so that they had a different degree of CPU and I/O processing. This would provide us an insight of how other CPU intensive or I/O intensive applications would perform with the Version Manager.

The first application, called SourceRouter, is a simple application written in C. It takes from the client a simple lookup request consisting of a destination host and a list of intermediate hops through which the lookup request is routed to the destination and the lookup answer is propagated back to the client. The application is neither CPU intensive nor I/O intensive. The goal of using this simple application was to get the estimation of the overhead of our approach on a minimal application. Nonetheless, we believe that this simple application emulates an important subset of the functionalities of many peer-to-peer applications. In our evaluation, we ran this application on 12 Emulab machines forming a chain topology with the latency and bandwidth between the two adjacent machines being 25 ms and 5 Mbps respectively.

The second application we used in our evaluation is IrisLog described in Section 5.2. This is an example of both CPU and I/O intensive application. We ran IrisLog using a hierarchy of the same depth as the original IrisLog running on PlanetLab, but consisting of only a subset of the PlanetLab nodes[7]. Specifically, our hierarchy, part of which is shown in Figure 5, represented three regions (`USA-East`, `USA-Mid`, and `USA-West`) of the USA. Under each region, we had two PlanetLab sites (`CMU` and `MIT` in `USA-East`, `Univ. of Texas` and `UIUC` in `USA-Mid`, `Stanford` and `Univ of Washington` in `USA-West`). Finally, each site had two PlanetLab nodes. We created a topology in Emulab to represent this hierarchy. The latencies of the links in the topology were assigned according to the expected latencies between the of corresponding real PlanetLab nodes. The bandwidth of the local area links (*i.e.,* links between nodes in the same site) was 100 Mbps, while that for the wide area links was 5 Mbps.

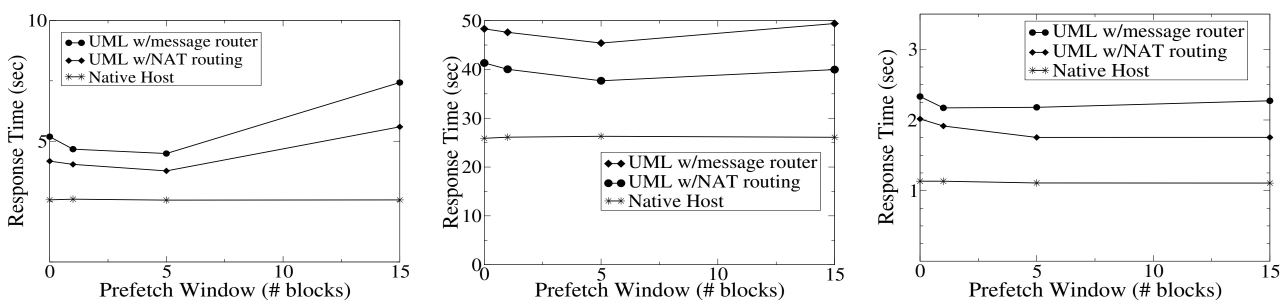
The third application we used in our evaluation is CFS, described in Section 5.1. It is an example of I/O intensive application. We classify CFS as I/O intensive based on its use of the network. Specifically, CFS

[7]Since different nodes at the same level of the hierarchy process a query in parallel, the response times mostly depend on the depth of the hierarchy. Therefore, the response times in our simple setup are very similar to those in the original IrisLog.



(a) SourceRouter (b) IrisLog

Figure 6: Overhead of running the applications inside VM.



(a) Write(1MB file) (b) Write(10MB file) (c) Read(1MB file)

Figure 7: CFS read/write overhead.

requires multiple RPCs per block fetch. And, because it uses UDP as its transport, each block fetch requires multiple system calls. Our experiments with CFS focus on a LAN topology of 16 nodes. The links have a bandwidth of 100 Mb/sec, and a delay of <1 ms. The `lsd` daemons are configured to run one vserver each. For file reads and writes, a block size of 8 KB is used.

6.2 Virtualization Overhead

Figure 6(a) shows the overhead of running the SourceRouter application inside UML VMs. The graph shows how the response time increases with the number of overlay hops required for the lookup. As expected, the response time increases linearly with the number of hops. As the graph shows, running this application inside UML incurs small local overhead ($\approx$ 6%) which ascertains the viability of running such applications inside UML.

Running the Java based application IrisLog inside UML, however, incurs higher overhead. Figure 6(b) shows the averages and 95% confidence intervals of the response times of different types of IrisLog queries. The type of a query is given by the level of the root of the subtree selected by the query (*i.e.,* the lowest common ancestor (LCA) of the selected nodes, in IrisLog's terminology). For example, a query with the `Region` LCA asks information about all the nodes in a region. As the graph shows, IrisLog with VM experiences 6% overhead in response times when the query involves a single node, and around 36% overhead when the query involves a whole region or country. The overhead is higher than that of the SourceRouter. This is expected since IrisLog is more I/O intensive than SourceRouter, and performing I/O from inside the UML is expensive.

Figure 7 shows the overhead of running CFS inside UML VMs. It reports the response times to read and

Delta Size	Execution time
3 MB	152 sec
6 MB	153 sec
9 MB	160 sec
12 MB	162 sec

Table 1: Upgrade execution time as a function of the size of delta. Results are reported for upgrading IrisLog, and are the mean of five runs.

Step	Execution time
1	15 sec
2	9 sec
3	54 sec
4	20 sec
5	60 sec
other	2 sec
Total	160 sec

Table 2: Breakdown of upgrade execution time for IrisLog. Results are reported for upgrading IrisLog, and are the mean of five runs. Different steps refer to the steps in Figure 2

write files of different sizes. For each file size, we vary the prefetch window (the number of simultaneously outstanding block requests) from 0 to 15. Higher values of prefetch window benefit from pipelining up to a point. However, the highest prefetch window size actually increases response time for CFS inside the VM.

We notice that the response times increase significantly (by $\approx$ 100%) when CFS runs inside the UML environment running on top of our message router. This is explained by the fact that CFS has a high I/O to computation ratio, and thus the overheads of processing I/O from inside the VM dominates in the total response time. To further understand how much of this overhead is attributed to our message router, we plot the overhead of running CFS inside UML, but without our message router (shown as "UML w/NAT routing" in the graphs). As shown, most of the overhead comes from the UML, and our message router incurs only around 20% overhead. We believe that the UML overhead can be reduced by using a more efficient VM [6, 24].

6.3 Upgrading Overhead

Amongst the desirable characteristics of an upgrade system are the abilities to upgrade quickly, and to minimize service disruption. Here, we evaluate the ability of our system to meet these requirements.

6.3.1 Time Required to Upgrade a New Version

We here present the time required for a system-wide upgrade, as well as a break-down of the costs of the upgrade procedure for a single node.

Local Upgrade Local upgrade includes all the steps shown in Figure 2. Table 2 presents a breakdown of the upgrade execution times for IrisLog, as observed at a single node. The steps are numbered as in Figure 2. We omit steps that have an execution time of one second or less. From the table, we find that the upgrade costs are dominated by the time to boot the new VM (step 3 in Figure 2), and the time to start the application inside the VM (step 5). However, the latter component is application dependent. We used IrisLog distribution source code, and after getting the new version from a remote host, the local host needed to run the configuration scripts which took significant portion of this time. Using a binary distribution

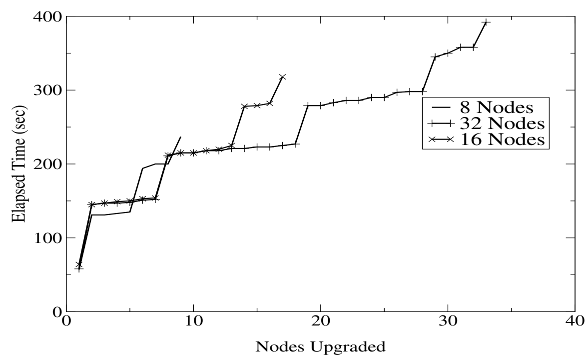


Figure 8: Time required to upgrade CFS.

compiled to run within the VM would avoid this expensive configuration process and significantly reduce this time.

There are two factors that can make the completion time of local upgrade different for other applications. The first factor is *delta*, the size of the data that needs to be transferred from the remote host during the `rsync` operation (Step 2 in Figure 2). Table 1 shows the local upgrade time of IrisLog with different hypothetical delta sizes. The time is measured from when the upgrade is initiated (due to a request from the node running a newer application version) until the new application version is running on the upgraded node. The numbers in Table 2 use a delta of 9 MB, which is the observed delta between two consecutive stable releases of IrisLog.

The second factor affecting the local upgrade time is the time to start the application inside the VM (step 5). As mentioned before, IrisLog took long starting time because of running the configuration scripts. However, CFS comes as a precompiled binary and hence does not require that process. We found that the CFS application takes only 2-5 seconds to start within the VM.

System-Wide Upgrade System-wide upgrade time is defined by the time required to all the (up) nodes in the system to finish upgrading the new version. We use both the IrisLog and the CFS applications to study this time.

To measure the time required to complete a system-wide upgrade of IrisLog, we initiate a query from the root of the IrisLog hierarchy. The query is sent recursively down the tree, from each parent to its children. Note that, because the version field is rewritten by each node that processes the query [8], a child will never attempt to upgrade before its parent. Thus the time to complete a system-wide upgrade is a function of the depth of the hierarchy.

Table 3 presents the time required to upgrade an IrisLog deployment as a function of the hierarchy depth. A depth of three corresponds to the hierarchy of Figure 5, while smaller depths reflect a smaller deployment. As expected, the upgrade time is roughly linear in the hierarchy depth.

To measure the global upgrade time of CFS, we first boot `lsd` on all the nodes in the system. We monitor the routing tables of the nodes to determine when all the nodes have joined the Chord overlay. We then start a new version of `lsd` on one of the nodes in the system. We then measure the time until the new version of `lsd` has started execution on each node. We also measure the time until all nodes have joined the overlay running the new software version.

Figure 8 presents the time required to upgrade CFS instances of 8, 16, and 32 nodes. Each point in a series represents the time at which another node has started execution of `lsd`. The last point in a series represents the time at which all nodes have joined the new overlay. The series show clear stair-step patterns. We believe this occurs as first a single node is running the new version, then that nodes immediate neighbors, then neighbors of immediate neighbors, and so on.

[8]In fact, the query message itself may be rewritten as well.

Depth of Distribution Tree	Global Upgrade Time
1	2:19
2	4:48
3	7:09

Table 3: Time required to upgrade IrisLog.

	Response time
Period	IrisLog
Steady State	6.294 sec
Upgrade	6.769 sec

Table 4: Impact of an ongoing upgrade process on response times.

6.3.2 Performance Degradation During the Upgrade Period

We now determine how much the normal operation of an application is disrupted by the ongoing upgrade process. There are two major reasons for which a distributed application may exhibit degraded performance during the upgrade period. First, the upgrade process may congest the network and overload the hosts performing local upgrade. Second, for the applications that improve their performance by caching data, a newly started version of the application may incur cold cache misses and perform poorly.

Table 4 shows the performance degradation for the SourceRouter and the IrisLog applications. We ran the SourceRouter application with a 10-hop lookup requests, and the IrisLog application with a query that accessed all the nodes in the hierarchy. The SourceRouter application does not cache data, so its performance degradation is mainly because of the first factor mentioned above and is less than 2%, as the table shows. IrisLog however, caches data and compiled query processing programs[12]. If we exclude the caches misses, IrisLog sees a performance degradation of $\approx 4\%$. The queries that experience cache misses, however, see a 40% higher response time.

In a nutshell, the performance degradation during the upgrade period due to the network and CPU overload is minimal. Performance degradation may be significant as a result of cold cache misses, but this degradation is experienced only by the first queries that touch the newly installed versions and gets amortized over time.

7 Related Work

There are two pieces of related work on upgrading large scale distributed systems that are worth noting: the work on upgrading the Internet routing infrastructure [23] and work on upgrading classes in object-oriented databases [7].

The Internet routing infrastructure can be viewed as a large distributed application. As many networking researchers have bemoaned, the difficulty of upgrading or incorporating new functionality into the Internet infrastructure has significantly limited the deployment of new techniques. This difficulty is the result of both a design that does not accommodate automatic deployment of new functionality and the distributed ownership of the Internet (making it difficult to reach consensus about upgrades). The Active Network [23] community spent many years attempting to address these shortcomings with unfortunately little success. However, we believe some of the important lessons from this work and the deployment of new protocols in the Internet do carry over to the area of upgrading distributed applications.

A variety of work has focused on upgrading classes in object oriented systems. A common approach to this problem relies on either stopping the system to apply the upgrade [4] or by limiting the types of changes that can be made [18, 15, 5]. More recently, Boyapati et al.'s work [7], on the Thor system, identifies a key set of properties for object upgrade transforms that allow object upgrades to be postponed until the object is accessed. Assuming that this object is accessed infrequently, such upgrades might be processed in the background with interrupting service. Our work differs in some important ways. First, we do not require an object-oriented design. Second, we try to hide service interruptions regardless of the service's access pattern. In addition, while their system places restrictions on the type of upgrades, our system places restrictions on the type of applications that we can support. However, our system likely incurs higher computation overhead than Thor since we rely on simultaneous execution. These differences suggest that both systems are useful in different context. The focus of each system might be the result of the fact that our design was motivated by the challenges of maintaining the distributed systems such as those deployed on PlanetLab, while Boyapati's work seems more motivated by the object oriented database community.

8 Future Work

Our system only begins to address the many needs of large distributed applications. For example, as described earlier, rollback is an important need closely related to upgrade. Our current system supports rollback simply by allowing earlier versions to continue executing in parallel with the newest version. Administrators can choose to shutdown more recent versions of the software if bugs are identified and the older versions will continue to provide service. However, one open question is how to handle side effects of buggy software upgrades. For example, in an e-commerce application, the web service software might be responsible for computing the tax on a purchase. While we can revert a software upgrade that has a bug in its tax computation code, we can not automatically correct its tax calculations. Although such corrections are clearly application specific, incorporating an undo facility [8] may ease the application developer's burden.

While we have primarily only explored the usefulness of this tool to support upgrades, we believe that this system can support an number of other uses. For example, we plan to use parallel execution to provide an easy way to perform live testing of the beta software. The basic high level idea is to simultaneously execute the new version, V_2, of the software on all the nodes that already have the old version, $V - 1$. The testing algorithm submits user queries made to V_1 to V_2, and *compares* the outputs of the two versions to detect any disagreement between the two versions[9]. This simple scheme can provide useful debugging information. First, it can identify user requests for which the output of V_2 does not agree with that of V_1. The developer can then evaluate if the disagreement is due to some bug in V_2. Second, for message equivalent upgrades, the particular host causing the disagreement can also be pinpointed by performing the comparison of the outputs of V_1 and V_2 along every hop of an user request.

Another possible use might be to run different versions of the software in parallel (much like n-version programming) and compare output to ensure that no version has become victim to a security breach.

9 Conclusions

It is clear that distributed applications are a significant part of the computing landscape. But to fully realize the potential of such applications, we need tools that help address the complexity in developing and managing them. The particular problem we have addressed in this paper is the onerous and difficult task of upgrading a distributed application – especially, when the upgrade does not maintain full backward compatibility. We have described a novel tool that greatly simplifies this task and attempts to minimize any service disruption as a result of the upgrade.

The core idea behind our design is the simultaneous execution of different application versions on a single node. This allows for upgrades to be deployed without denying access to the service. Our system uses a combination of virtual machine technology, application-specific message routing, and request-driven code propagation to enable this design. Our experience with a prototype upgrade system, shows that: 1) it is able to support large applications with complex interactions (e.g. CFS), 2) it imposes reasonable overheads in normal operation (about 6% for the SourceRouter, 6-36% for IrisLog, and $\approx 100\%$ for CFS), and 3) it is able to propagate upgrades quickly (160 secs to upgrade a single node and 10 minutes to upgrade a 12 node IrisLog system).

[9]Note that the use of V_2 is hidden from the users

References

[1] EmuLab: The Utah Network Emulation Facility. http://www.emulab.net.

[2] IrisLog: A Distributed Syslog. http://www.intel-iris.net/irislog.php.

[3] IrisNet: Internet-scale Resource-Intensive Sensor Network Service. http://www.intel-iris.net.

[4] M. P. Atkinson, M. A. Dmitriev, et al. Scalable and recoverable implementation of object evolution for the PJama 1 platform. In *Persistent Object Systems (POS)*. Sep. 2000.

[5] J. Banerjee, W. Kim, et al. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD International Conference on Management of Data*. May 1987.

[6] P. Barham, B. Dragovic, et al. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*. 2003.

[7] C. Boyapati, B. Liskov, et al. Lazy Modular Upgrades in Persistent Object Stores. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Oct. 2003.

[8] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX 2003 Annual Technical Conference*. San Antonio, TX, Jun. 2003.

[9] J. Cates. *Robust and Efficient Data Management for a Distributed Hash Table*. Master's thesis, Massachusetts Institute of Technology, 2003.

[10] C. S. Committee On Research Horizons in Networking, D. o. E. Telecommunications Board, et al. *Looking Over the Fence at Networks: A Neighbor's View of Networking Research*. National Academy Press, Washington, D.C., 2001.

[11] F. Dabek, M. F. Kaashoek, et al. Wide-area cooperative storage with CFS. In *Proceedings of the 18th Symposium on Operating System Principles*. Chateau Lake Louise, Banff, Canada, Oct. 2001.

[12] A. Deshpande, S. Nath, et al. Cache-and-query for wide area sensor databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2003.

[13] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the Annual Linux Showcase*. Atlanta, GA, Oct. 2000.

[14] P. Druschel and A. Rowstron. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th Symposium on Operating System Principles*. Chateau Lake Louise, Banff, Canada, Oct. 2001.

[15] B. S. Lerner and A. N. Habermann. Beyond schema evolution to database reorganization. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Oct. 1990.

[16] M. L. Massie, B. N. Chun, et al. The ganglia distributed monitoring system: Design, implementation, and experience. Submitted for publication, February 2003, http://berkeley.intel-research.net/bnc/.

[17] A. Muthitacharoen, R. Morris, et al. Ivy: A read/write peer-to-peer file system. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*. 2002.

[18] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Oct. 1987.

[19] S. Ratnasamy, P. Francis, et al. A Scalable Content-Addressable Network . In *Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols*. San Diego, California, Aug. 2001.

[20] T. Roscoe, L. Peterson, et al. A simple common sensor interface for planetlab. PlanetLab Design Notes PDN-03-010, 2003.

[21] I. Stoica, R. Morris, et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols*. 2001.

[22] M. Wawrzoniak, L. Peterson, et al. Sophia : An information plane for networked system. In *Proceedings of the Hotnets-II*. 2003.

[23] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th Symposium on Operating System Principles*. Dec. 1999.

[24] A. Whitaker, M. Shaw, et al. Scale and performance in the denali isolation kernel. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*. 2002.