Simplifying Distributed Application Upgrades with Simultaneous Execution

Mukesh Agrawal **Suman Nath**
Srinivasan Seshan

November 2005
CMU-CS-05-190

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Developers and operators 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 be interoperable, or they can shut down the entire system when an upgrade is required. Requiring interoperability complicates the life of system developers, while shutting down the system annoys users. We propose *simultaneous execution*, a middle ground between these approaches: an upgrade methodology that reduces the extent of support required for interoperability, while also providing system availability during upgrades. We demonstrate the benefits of our approach by applying it to two widely different distributed systems: the Cooperative File System [1], and IRISLOG [2]. Our proposal enables easy upgrade of these systems while maintaining system availability, and requiring only minimal support for interoperability.

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 that the difficulty of upgrading the Internet infrastructure has lead to the ossification of both Internet research and Internet functionality [3]. 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 [4, 5], distributed storage [1, 6, 7], and sensor systems [2]. However, these infrastructures may share the Internet's weakness since few (if any) of these infrastructures accommodate easy upgrades. In this paper, we present techniques that simplify the upgrade of many distributed applications, 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 inter-operate 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: one that maintains system availability, while minimizing the support required for interoperability.

In our simultaneous execution approach, we execute multiple versions of an application simultaneously, on the same hardware. We then treat these different versions as separate deployments. Two versions of an overlay network service, for example, are treated as separate overlay networks.

To enable multiple application versions to run on the same physical infrastructure, we must prevent interference between multiple versions on a single node, and carefully manage communication between nodes. We prevent interference within a node through the use of virtual machines (VMs). With respect to inter-node communication, we isolate versions, so that only nodes of the same version communicate with each other. We manage messages between clients and the application using application proxies, which mediate client access to the service, masking the existence of two versions.

In this paper, we argue for the general applicability of our approach, by describing how a prototype implementation of our design, called the Version Manager, supports two recently proposed distributed infrastructures that differ substantially in their designs: the CFS distributed storage system [1] and the IrisLog distributed infrastructure monitoring system [8]. As an example of their differences, CFS 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 prototype show that with our parallel execution approach, we are able to upgrade both applications without disrupting availability.

The Version Manager dramatically reduces, but does not eliminate interoperability requirements. The specific components that Version Manager requires are the application proxy, and a tool to copy state from one version to another. Using a proxy framework which we provide, along with existing CFS library routines, the application proxy for CFS is about 500 lines of C++ code. The state copying tool is another 500 lines of C++. For IrisLog, the application proxy and state copying tool require 300 lines of C++, and 50 lines of scripts (bash and perl) respectively.

In addition to simplifying the complexity of upgrades and eliminating downtime, the Version Manager is also able to propagate new versions of software relatively quickly. For example, with

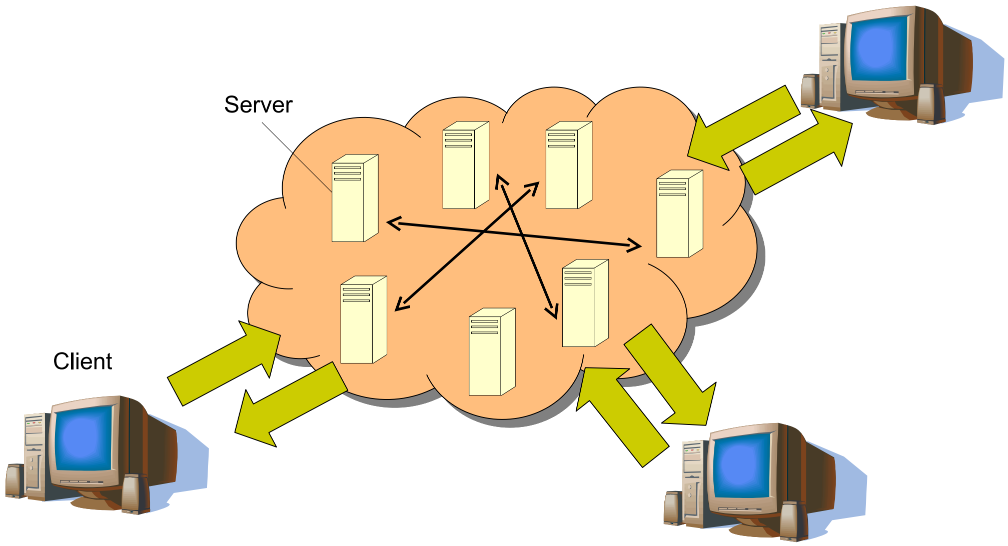

Figure 1: Targeted Application Architecture

a 48 MB software update, Version Manager upgrades a 12 node IRISLOG system in 7-8 minutes. With CFS, and a 6 MB software update, Version Manager completes an upgrade of 48 nodes in under 6 minutes.

The primary cost of using our system is the additional overhead from the virtual execution environment, and from inter-positioning on inter-node communication. During normal execution, when only a single version is running, CFS and IRISLOG incur overheads of up to 300% and 14% respectively. The overhead for CFS is, admittedly, large. However, we believe this overhead can be reduced through optimizations or architectural changes to the implementation, as we discuss in Section 7.

The remainder of this paper is structured as follows: we explain the obstacles to distributed application upgrades in Section 2. In Section 3, we present the design and implementation of our system. In Section 4, we describe our experience in applying our methodology to two existing applications. Section 5 presents experimental results about the performance of our system. Section 6 presents related work, and Section 7 discusses the benefits of our system, and possibilities for performance improvements.

2 Upgrade Obstacles

We consider distributed applications with the architecture depicted in Figure 1. Such applications consist of a number of servers, possibly widely distributed, which cooperate to implement a service. Clients utilize the provided service, but do not concern themselves with the server-to-server interactions.

Upgrading such applications is difficult due to the numerous hurdles that developers and operators of such systems must overcome. The most significant of these are: designing and implementing inter-operable software; testing upgrades before deployment; planning for recovery, in case the upgrade fails; and deploying the new software.

Interoperability The state of the art presents a stark choice for distributed application developers and operators. Either developers must design and implement mechanisms for interoperability, or the operators must shut down the system completely when upgrades need to be made.

Given only these options, new software is generally designed to inter-operate with old software, which is replaced in-situ. However, this approach to system evolution suffers from two significant problems. First, it severely constrains the nature of feasible changes. Second, it imposes a heavy implementation and maintenance burden on software developers.

With respect to precluded changes, requiring interoperability prohibits changes to important distributed algorithms such as those that control message routing, load-balancing, and cooperative caching. With regard to the developers' burden, it may be possible to make small changes without significant difficulty. But for large changes, interoperability may essentially require the developer to implement two very different programs in a single process.

Testing and recovery The essence of the testing problem is coverage. While simulation and testbed testing may uncover some problems, it is overly optimistic to expect such testing to anticipate problems that will occur "in the wild." The recovery problem consists of two parts: replacement of faulty software with a new version, and "undoing" the consequences, such as data corruption, of the buggy software. This paper does not focus on these challenges. However, we briefly discuss how the Version Manager could be used to address these issues in Section 7.

Deployment process Conceptually, the deployment process might be quite simple. An operator simply logs on to each machine, downloads the new software, and runs some installation script. For large systems, however, this simple process is complicated by the reality that, at any given time, some nodes will be offline. Furthermore, given recent studies which show that operator error is a significant source of outages in Internet services, it is essential to automate the deployment process [9, 10]. Today, automation is hindered by the complexity of error handling. For example: what should be done if a new version fails to start, after the old version has been overwritten?

3 Design and Implementation

The key idea behind our design is *simultaneous execution*. We borrow and adapt the idea from the deployment of IPv6 (and other protocols) in the Internet. Figure 2 provides a logical view of simultaneous execution. The essence of the technique is to allow multiple versions of an application to run simultaneously on a single server node (without interference), and to route client traffic to the appropriate version or versions. The arrows in the diagram indicate the server-to-server communication, highlighting the fact that inter-server routing need not be consistent in old and new versions.

Simultaneous execution addresses the interoperability problem by eliminating the need for servers to inter-operate amongst versions. This enables more radical changes in server to server

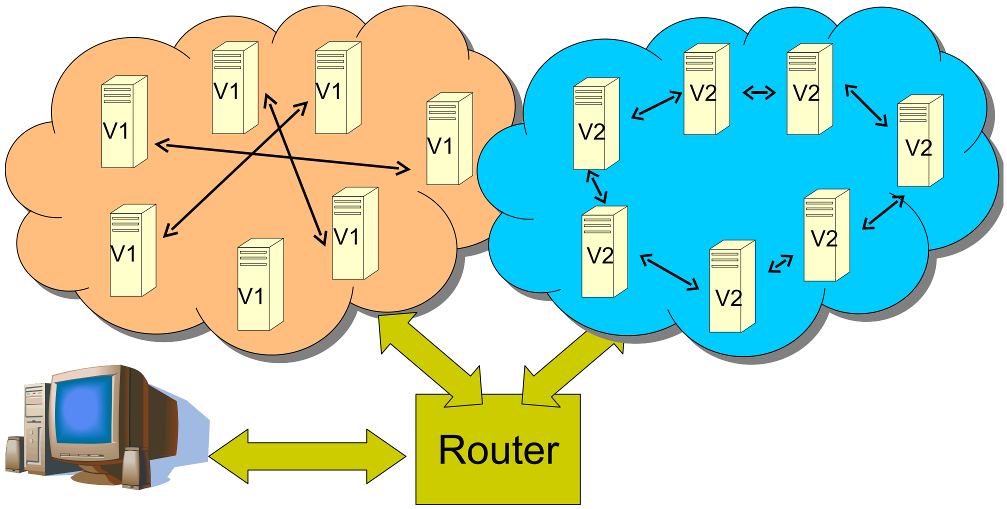

Figure 2: Logical View of Simultaneous Execution

designs. Simultaneous execution simplifies deployment as well. Because new versions do not interfere with old ones, an automated deployment system can simply kill a failed new version, without needing to provide elaborate recovery mechanisms.

Figure 3 illustrates the realization of simultaneous execution in the Version Manager architecture. While simultaneous execution simplifies distributed application upgrades, it does require some application specific components, such as the Application Proxy. Next, we explain the details of our Version Manager implementation, and the demands it makes of distributed application developers. We present case studies that quantify the costs of meeting these demands in Section 4.

3.1 Isolating Versions on a Node via Virtual Machines

In order to enable different versions of an application to run on a single physical node, as in Figure 3, we must prevent different versions of the application from interfering with each other.

To prevent this interference, a number of well-known isolation techniques, ranging from the use of multiple processes to full-blown hypervisors, might be used. At one extreme, processes provide the least isolation, with the lowest performance impact. At the other, hypervisors provide strong isolation, with a greater performance cost.

We believe the more limited isolation techniques such as multiple processes, or `chroot()` environments are insufficient. The multiple process approach suffers from the inability to readily support applications that are themselves structured as multiple processes. Moreover, neither the use of multiple processes, nor `chroot()` supports multiple user ids, which might be needed by applications that employ "privilege separation" to protect against malicious users. They also do not permit the multiple versions of an application to listen on the same transport protocol (e.g., TCP or UDP) port.

A more promising possibility is to use BSD `jail()` environments, or Linux *vservers*, as both of these facilities eliminate the user id and listening port limitations. However, both preclude

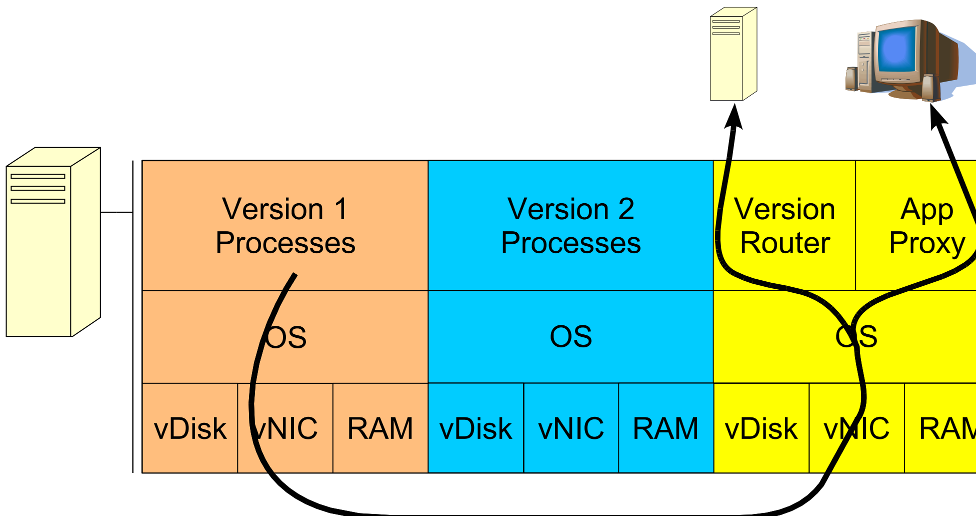

Figure 3: Version Manager Architecture. Colors designate distinct Virtual Machines.

software upgrades that require new kernels. A new kernel might be desired, for example, due to improvements in filesystem or virtual memory algorithms.

To support the broadest set of possible changes, we choose the Xen hypervisor as our isolation environment. The Xen hypervisor runs directly on the hardware, and provides an x86-like interface to which operating system kernels are ported.

Under Xen, each kernel is executed in a separate *domain*, consisting of resources such as memory and (possibly virtualized) disks and network interfaces. The system is partitioned into a root domain, used for managing the system, and a number of user domains, which host applications. Typically, physical devices are managed by the root domain, while user domains see virtual disks and network interfaces.

We run each application version in a separate Xen domain, with a private virtual disk (backed by storage in the root domain), and a virtual network interface connected to a counterpart in the root domain. The user domains are configured with unique private IP addresses. They are provided with access to the public network via the network address translation feature of the root domain.

This approach isolates the application versions by providing near-complete isolation of namespaces. One remaining resource, however, is not isolated: communication channels used by the application. Specifically, these are the channels used for inter-server, and client-server communication. Our inter-node isolation strategy, implemented by the version router (Section 3.2) and application proxy (Section 3.3), provide isolation for these resources.

3.2 Routing Inter-Server Messages with a Version Router

The need to manage inter-server communication arises from the fact that the server-to-server protocols for new versions of an application are likely to 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, which we call a version router, on the communication between an application and its peers on other nodes. To guarantee that application traffic is routed to correct versions at peers, the Version Router prepends a header to each outbound request, identifying the version number of the sender. 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*. We provide a tool that is run as part of the bootstrap process in the isolation environment, which accepts the application port numbers as a command-line argument, and registers the application with the version router.

3.3 Mediating Client Access through an Application Proxy

The role of the application proxy is two-fold. First, if the client protocol has changed, the proxy may need to translate client-to-server and server-to-client messages. Second, the proxy must provide a unified view of the system to clients, masking the existence of multiple versions.

The translation of requests and responses will necessarily be application specific. We hope that separating message translation and the implementation of the new server will simplify both. However, we acknowledge that the requirement for translation will discourage changes to the client protocols.

The task of making multiple independent systems appear to behave as a single consistent system seems, at first blush, to be quite arduous. To provide this illusion, we must solve three subproblems. First, any data available in the system at the time the upgrade is initiated must remain available to all clients during and after the upgrade, regardless of which version of the system they access. Second, any changes made to the client-visible state must be made to all versions. Third, despite the fact that state may be changing even during the upgrade, clients of the old and new versions should be able to agree on what the system state is.

As daunting as all this seems, for the applications we have studied, we have accomplished these tasks using very simple approaches. To provide access to old data in new versions, we have written migrators that copy data from old versions to new[1]. To ensure that clients of new versions can access this data before migration is complete, we have the application proxy resubmit failed reads to older versions. To guarantee that all changes are visible in all versions, the application proxy submits writes to all versions, returning success to the client if (and only if) all running versions return success. To provide meaningful semantics across clients accessing different versions, we exploit the self-certifying nature of the results returned by the applications. We discuss the implementation costs of these solutions in Section 4.

[1] These are run from the upgrade script described in Section 3.5.

```
typedef int (*data_cb_t)
   (vers_t, databuf&, conn_handler&);

struct proxy_callbacks {
   data_cb_t handle_request;
   data_cb_t handle_reply;
};
```

Figure 4: Framework/application Interface

3.4 Supporting Application Developers with a Proxy Framework

Developing an application proxy is a non-trivial task, particularly for stream protocols (such as TCP) that do not preserve message framing. To assist the developer, we provide a proxy framework in C++, which manages network communications and data buffering, calling into application specific code for message translation and dispatch.

Specifically, the framework manages each client connection, and its associated server connections, using a connection handler. The connection handler interfaces with the application specific code via two callbacks: `handle_request()`, and `handle_reply()`. The former is called when a message is received from a client connection, the latter is invoked on receipt of data from a server connection. The arguments supplied in these callbacks are the version number of the sender, a buffer containing the data, and a reference to the invoking connection handler. Figure 4 gives the C++ declaration for this callback interface.

In addition to the standard read and write operations, `databuf` supports a `peek()` operation. This operation returns data from the buffer, without consuming it. Subsequent calls to `peek()` will return the data following the peeked data. Peeked data is consumed when the handler returns, with the return value of the handler indicating the number of bytes to be consumed. The effects of intermingling `read()` and `peek()` calls are undefined.

The motivation for peekable buffers is to support parsers that need to examine arbitrary amounts of data before they can determine if the message is complete. Without peekable buffers, the parser would have to support incremental parsing. With peekable buffers, if the message is incomplete, we can simply discard the parser state, and try again when more data is available. Figure 5 details the buffer API.

The last argument to the callback is a reference to the connection handler. The connection handler provides calls for enqueuing data either to a server (by specifying the protocol version implemented by the server), or to a client. Ownership of enqueued buffers passes to the connection handler. The connection handler also provides the application code access to a list of running versions. We list the connection handler interface in Figure 6.

The version list is required so that the application code can decide where to dispatch a request, and what translation (if any) is necessary (similarly for replies). The versions list provides `min()`, and `max()` methods, which return pointers to the least and greatest available server versions. `vers_list` also provides the usual calls (e.g. `operator++()`, `operator--()`, etc.) used

```
class databuf {
public:
   databuf(const databuf& orig);
   databuf(const databuf& orig,
           size_t len);

   int read(char *buffer, int len);
   int peek(char *buffer, int len);
   void reset_peek();

   void append(char *buffer, int len);
};
```

Figure 5: Buffer API

```
class conn_handler {
public:
   void enqueue(vers_t version,
      databuf *buf);
   void enqueue_client(databuf *buf);

   vers_list& versions();
};
```

Figure 6: Connection Handler API

to iterate over lists.

3.5 Upgrade Discovery, Distribution, and Installation

As part of the upgrade process, nodes must learn that a new version is available, and they must retrieve the software for the new version. We solve the first problem by using the Version Router to piggyback version advertisement messages on server-to-server communication, and solve the second problem using the BitTorrent file transfer protocol.

To support upgrade discovery, the Version Router adds a second field to the header it prepends on outgoing server-to-server messages. Whereas the first field of the header identifies the sending application version, the second field advertises any one of the other versions (randomly selected) running on the same host as the sender. This enables the Version Router on other nodes to learn of new versions from the sender, or to inform the sender that one of its running versions is obsolete.

When a receiving Version Router learns of a new version, it executes an upgrade script. The script's arguments include the address of the Version Router that advertised the new version. The script is responsible for retrieving the new software, and creating a new domain to host the new

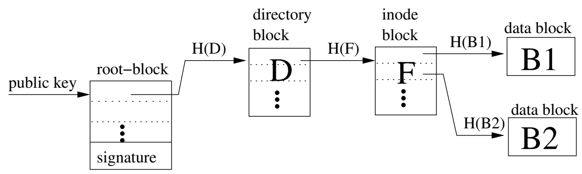

Figure 7: A simple CFS file system structure example. The root-block is identified by a public key and signed by the corresponding private key. The other blocks are identified by cryptographic hashes of their contents.

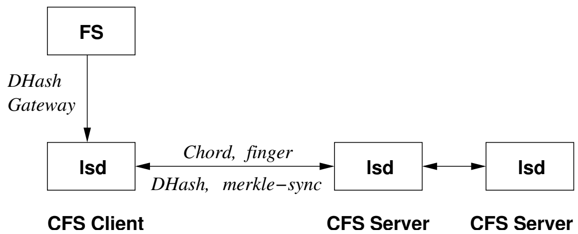

Figure 8: CFS software architecture. Vertical links are local APIs; horizontal links are RPC APIs.

application version. The specific steps taken by the upgrade script are as follows:

Download The script copies a `.torrent` file from the node advertising the new version, and starts a BitTorrent client to transfer the file.

Prep FS The script creates a filesystem image for the new application version.

Create VM The script creates a Xen domain to host the new version.

A boot-time script executed in the new VM manages the remaining tasks:

Copy to VM Copies new software from root domain to application domain.

Config app Unpack the software and run any installation procedure.

Running Run the application.

4 Case Studies

In this section, we describe our experience in applying our Version Manager methodology and tools to two applications: CFS and IrisLog.

4.1 Cooperative File System (CFS)

4.1.1 Application Overview

CFS implements a distributed filesystem over the Chord DHT. Conceptually, when a user writes a file into CFS, the file is broken into blocks [2]. 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.

[2] Physically, blocks are further broken into fragments, which are actually stored on the DHT nodes. The fragmentation of blocks improves fault tolerance [11].

Figure 7 illustrates the structure of the CFS filesystem in more detail. 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 Figure 8, 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 its 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.

Figure 8 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 [3].

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

4.1.2 Applying Version Manager to CFS

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

Application Proxy As described earlier, the application developer must provide a proxy which mediates client access during simultaneous execution. Using our proxy framework, and library routines from the CFS source code, we have written a proxy which implements the mediation strategy described in Section 3.3. This proxy is approximately 500 lines of C++ code.

State migration To use Version Manager, a developer must also provide a mechanism for copying state from old versions to new. To meet this requirement for CFS, we have implemented a simple program to copy the blocks from one version to another. The program uses the *merkle_sync*

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

protocol to query the old version of the application for a 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. This program is also about 500 lines of `C++` code, making use of the CFS library routines.

Consistency As noted in Section 3.3, clients of old and new versions should be able to agree on the system state, even though the state may be changing. Given that clients may be modifying the filesystem at the same time that our state migration tool copies blocks from the old version to the new, it would seem that inconsistencies might arise.

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

4.2 IRISLOG

4.2.1 Application Overview

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 [2], 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[4] (*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 [12] 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 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. Upon receiving a query, each IRISNET node queries its local database, determines which parts of the answer cannot be obtained from the local database, and recursively issues additional sub-queries to gather the missing data. Finally, the data is combined and the aggregate answer is sent to the client. IRISNET also uses in-network aggregation and caching to make the query processing more efficient [13].

Both IRISLOG and IRISNET are written using Java.

[4] 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).

4.2.2 Applying Version Manager to 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. In this section, we highlight the changes IRISLOG requires to incorporate our Version Manager.

Application proxy The IRISLOG application proxy processes queries by submitting them in parallel to all versions, and returning the longest result. The longest result is chosen in order to mask data that is missing from a new version during the interval between the new version being brought online, and the time when all old data has been copied to the new version. Data updates are processed by issuing the update to all versions, and returning success if and when all versions return success. The IRISLOG application proxy is approximately 300 lines of C++, making use of our proxy framework.

State migration IRISLOG uses a database-centric approach, and hence all its persistent state is 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 the IRISLOG command-line client to copy the whole local database of the old version running on the same host to the new version. The migration tool is implemented with about 50 lines of shell and perl code that calls the command-line client.

Consistency 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, thus, there are no write conflicts.

5 Performance Evaluation

In this section, we examine three key aspects of the performance of our system:

- How long does an upgrade take to complete, and how does this scale with the number of nodes in the system?
- How much overhead does our system impose during normal execution (i.e., when no upgrade is in progress)?
- How much disruption is caused during an upgrade?

We first describe our experimental setup, and then answer each question posed.

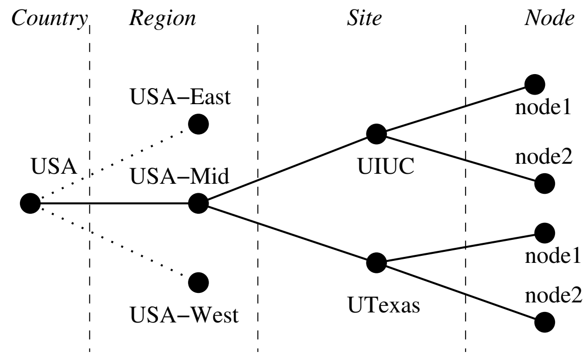

Figure 9: Part of the data hierarchy used in the IRISLOG evaluation. Subtrees omitted from the diagram are indicated by dashed lines.

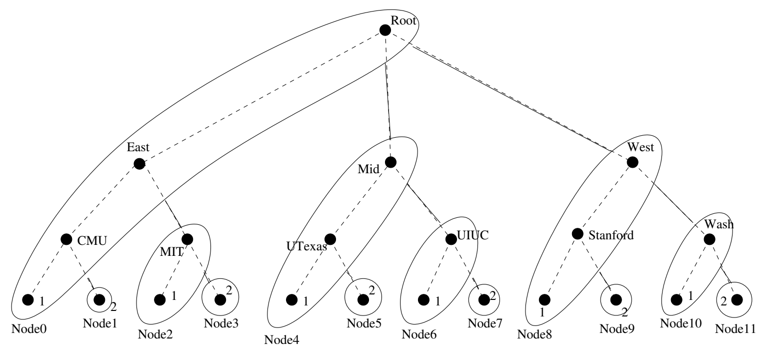

Figure 10: Mapping of IRISLOG data hierarchy to node topology. Bounded regions represent physical nodes containing the data items (shown as smaller solid circles) within the regions.

5.1 Experimental Setup

We conducted all our experiments on Emulab [14] using a set of 850 MHz Pentium III machines with 512 MB of RAM, and 100 Mbit Ethernet NICs. The operating system is Fedora Core 2.

Our IRISLOG experiments use a hierarchy of the same depth as the original IRISLOG running on PlanetLab, but consisting of only a subset of the PlanetLab nodes[5]. Specifically, our hierarchy, part of which is shown in Figure 9, represents three regions (`USA-East`, `USA-Mid`, and `USA-West`) of the USA. Under each region, we have 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 has two PlanetLab nodes.

We created a topology in Emulab to represent this hierarchy. The latencies of the links in the topology are assigned according to the expected latencies between the of corresponding real PlanetLab nodes. The bandwidth of links between nodes in the same site is 100 Mbps, while that for the wide area links is 5 Mbps. Figure 10 illustrates the mapping of the IRISLOG data hierarchy to the node topology.

[5]Since different nodes at the same level of the hierarchy process a query in parallel, 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.

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 2 KB is used.

5.2 Time to Complete Upgrade

In this section, we 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. We study the time to upgrade both IRISLOG and CFS. Both IRISLOG and CFS use a 2 GB filesystem image for the application VMs. The size of the software download is 48 MB for IRISLOG, and 6 MB for CFS.

IRISLOG. To measure the time required to complete a system-wide upgrade of IRISLOG, we start a new version of IRISLOG on the root node of the data hierarchy. We then execute a series of queries and data updates from our client node. Queries that pass through nodes having the new version cause children of those nodes to initiate upgrades.

We present the time to complete an IRISLOG upgrade in Figure 11. The vertical axis denotes the steps in the upgrade process, as described in Section 3.5. Each series represents the progress of a single node through the upgrade process. The progress of the IRISLOG root node, which seeds the upgrade, is distinguished with a heavy dashed line. A subtlety to note is that, for the seeding node, the `Download` stage represents time taken in house-keeping tasks, rather than downloading the upgrade.

The IRISLOG upgrade experiment was conducted five times, and the result presented is the *w*orst case amongst these runs. The time to upgrade IRISLOG in this case is 489 seconds. This compares to a maximum of 434 seconds maximum in the other four runs. Comparison of Figure 11 with graphs for the other experiments (not shown) shows that the difference arises from the large time spent for the last node in Figure 11 to learn of the availability of a new version. The time taken in this case, however, is within the expected variation for our benchmark workload.

Across the runs, the preparation of the filesystem for the new version consistently dominates the upgrade time. Even in the case presented, with a higher than typical time spent waiting to learn of an upgrade, the filesystem preparation stage accounts for 210 seconds of the 489 second upgrade process, or about 40%.

The time required to prepare a filesystem for the new version is due to our simplistic approach, which simply creates a new copy of a baseline filesystem on demand. Should it be necessary to complete upgrades more quickly, the time spent preparing the new filesystem could be eliminated through any number of techniques, such as keeping a spare filesystem image around, or by using copy-on-write, or stackable filesystems.

CFS. 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 ring. We 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. Throughout the process,

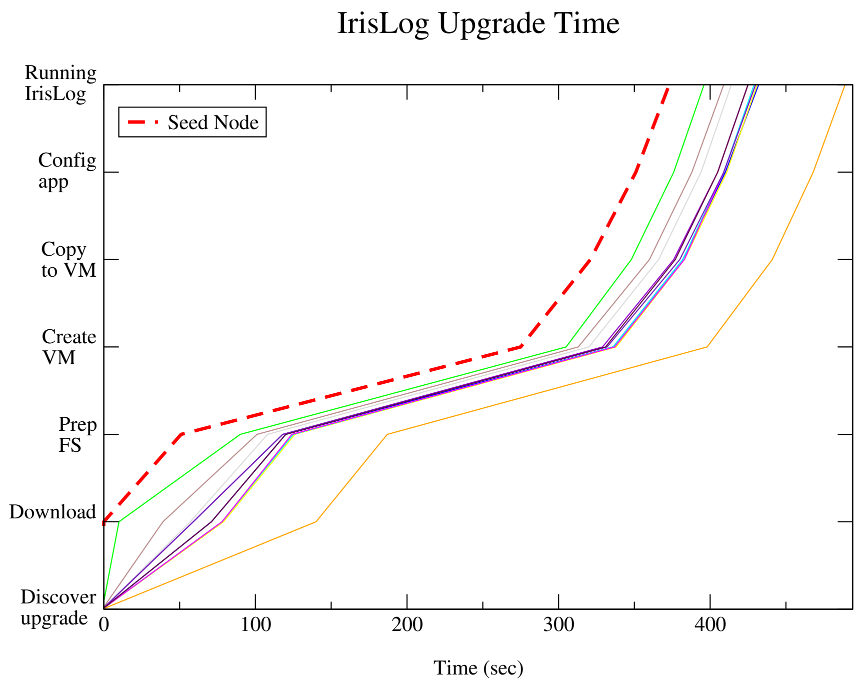

Figure 11: Time required to upgrade IRISLOG. The thick dashed line denotes the progress of the server seeding the upgrade. Other lines each represent one other node in IRISLOG.

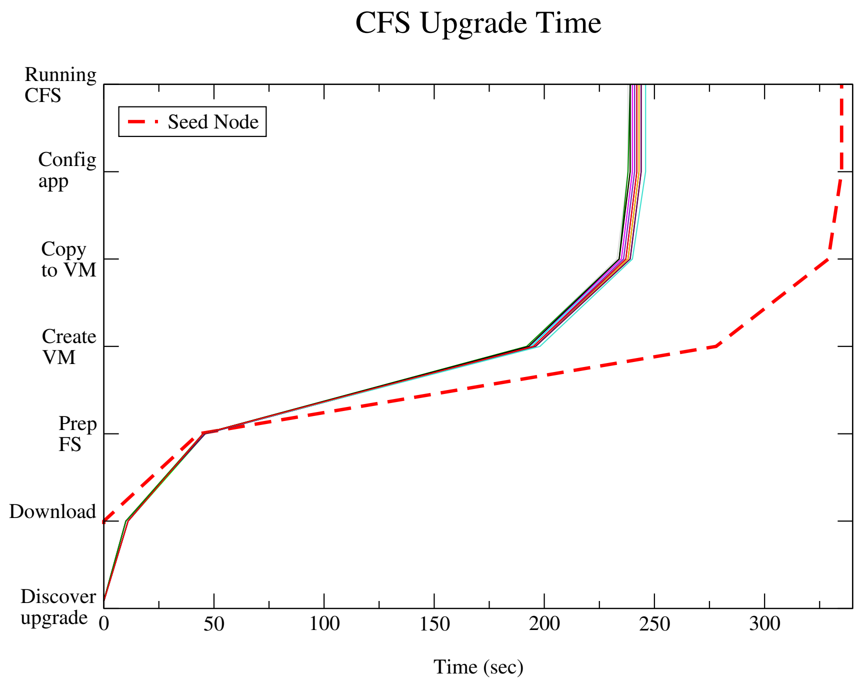

Figure 12: Time required to upgrade CFS. The thick dashed line denotes the progress of the server seeding the upgrade. Other lines each represent one other node in CFS.

a separate client node writes to CFS using the seed node to access CFS.[6]

Figure 12 presents the time required to upgrade a sixteen node CFS system. This experiment was run once. As with IRISLOG, the dominant cost in this upgrade is the time to prepare the filesystem for the new version.[7] The same techniques proposed to accelerate upgrades for IRISLOG would apply for CFS.

To determine the scaling behavior of upgrades, we have conducted the CFS upgrade with 32, 48, 64, and 80 nodes. The results are shown in Figure 13. We find that the upgrade time is constant between 16 and 48 nodes, and also between 64 and 80 nodes. The dominant difference between the smaller and larger experiments is the time taken to download the upgrade. Recent studies [15] have shown that the mean download time for BitTorrent clients (in a swarm) is dependent only on the mean upload bandwidth of all the clients, and is independent of the number of clients in the swarm. This leads us to speculate that the larger experiments (64 and 80) force Emulab to use nodes that are separated by one or more switches, while the smaller experiments use nodes connected to a single switch. The bottleneck between the clusters leads to longer transfer times.

---

[6]Even when client traffic is absent, CFS nodes communicate with each other periodically to accomplish tasks like Chord ring maintenance. Thus, the client traffic is not necessary in order to propagate the upgrade. However, we use the client traffic to measure the performance during an upgrade, which is presented in Section 5.4.

[7]The difference in time required to prepare the new filesystem between IRISLOG and CFS is due to a slight change in the procedure for preparing the filesystem. Rather than copying a base filesystem directly, we decompress a

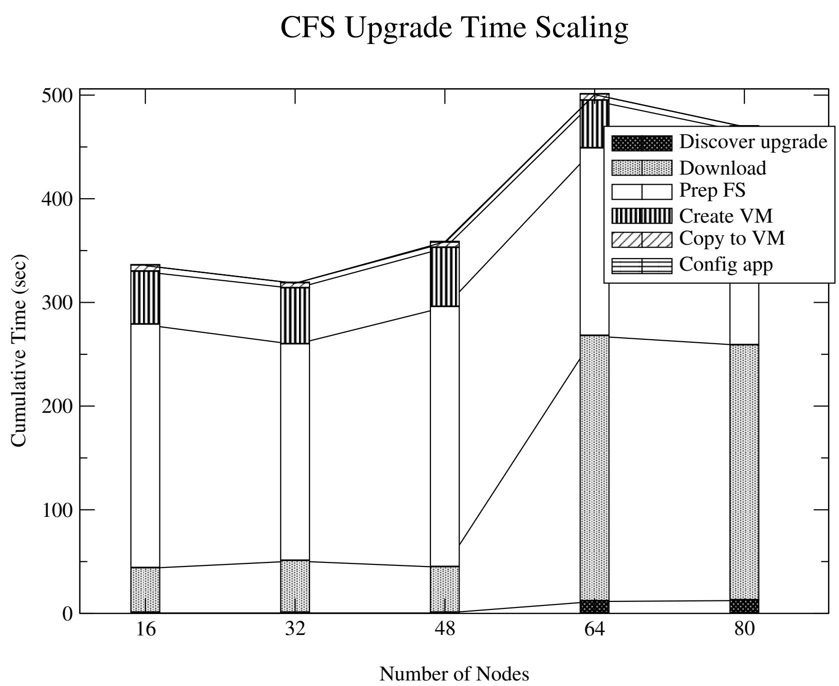

Figure 13: CFS Upgrade Scaling

Scaling in number of nodes. Although the results above are based on relatively small deployments, we argue that our approach has good scaling property and therefore is feasible for large-scale distributed systems. This is due to the observation that the time required for a system-wide upgrade is proportional to the diameter (in number of hops) of the system topology,[8] and almost all large distributed systems use small-diameter topology for scalability. For example, diameters of CFS and DNS (examples of structured topologies) and gossip-based systems (example of a random topology) are logarithmic to the number of nodes in the system and the diameter of the Gnutella (example of an unstructured topology) network consisting of around 400,000 nodes has been measured to be 20 [16]. Our approach can upgrade a large distributed system in a reasonable amount of time as long as it has a reasonably small diameter.

5.3 Virtualization Overhead

IRISLOG To measure the overhead imposed by our system when running IRISLOG, we benchmark the latency to execute update and query operations with and without our system. For queries, we vary the size of the query, in terms of the number of servers that must be contacted to answer the query. This number varies from one, for a `Node` query, to twelve, for a `Country` query. Update

compressed copy.

[8]The upgrade time also depends on traffic pattern, but most systems (e.g., CFS) use periodic background traffic that we can use to trigger a node's upgrade.

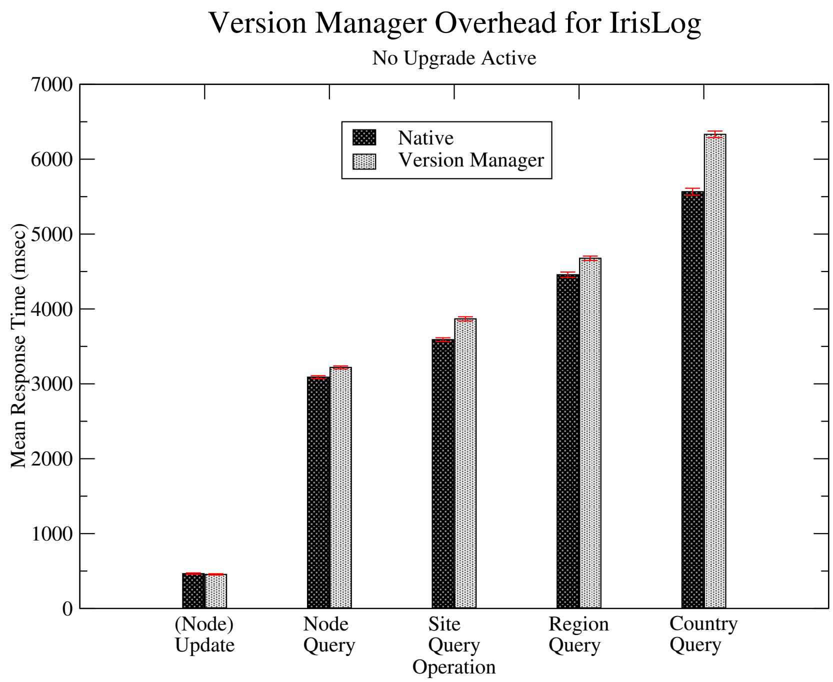

Figure 14: Overhead of running IRISLOG with our system.

operations, in contrast, always involve only a single node, as IRISLOG permits writes only at leaf nodes.

Each operation type, such as an update, or Node query, is executed a minimum of 150 times. Figure 14 presents the averages and 95% confidence intervals of the response times for each operation. For most operations, we observe a modest increase in latency of 8% or less. However, for Country queries, this rises to about 14%.

CFS To evaluate the overhead of Version Manager for CFS, we benchmark the read and write throughput achieved by a single client accessing a CFS service of 16 nodes. In these experiments, 1 MB of data is read from or written to the file system using the `dbm` program from the CFS distribution. The data is written in 2KB blocks, with a concurrency level of 0.[9] Each operation is repeated at least 25 times.

To determine the performance and sources of overhead for CFS under Version Manager, we run our benchmark under five different conditions, as illustrated in Figure 15. In the Native configuration, CFS is running on a standard Linux kernel, with the kernel running directly on the hardware. In the Xen configuration, we run CFS inside a single Linux domain atop of Xen. Xen 2D introduces a second Linux domain. In this configuration, CFS runs inside the second domain, and communicates with peer nodes by passing traffic through a NAT running in the root domain. In the

[9] The concurrency level is the maximum number of outstanding read of write requests.

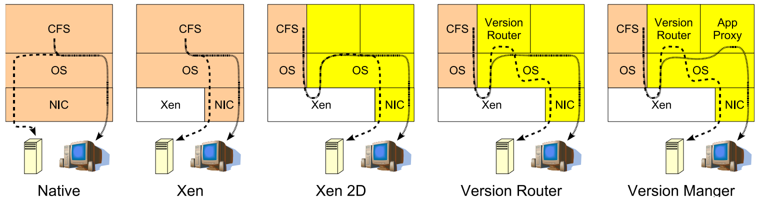

Figure 15: Machine configurations for CFS overhead experiments. Colors designate distinct Virtual Machines.

Version Router configuration, we replace the NAT with our Version Router. Finally, the Version Manager configuration includes two Xen domains, the Version Router, and the Application Proxy. Figure 16 presents the mean and 95% confidence interval for read and write throughput achieved.

We find that running CFS under Xen reduces read throughput by 18% and write throughput by 8%. Introducing another domain reduces read throughput by 50%, and write throughput by 25%, both relative to native execution. The addition of the version router and application proxy causes a further 50% reduction in read throughput, and 40% reduction in write throughput. As compared with native execution, we see a 72% reduction in read throughput, and a 52% reduction in write throughput. We believe these overheads can be reduced, as we explain in Section 7.

5.4 Performance Degradation During Upgrade

We now examine how the normal operation an application is disrupted by an ongoing upgrade.

IRISLOG To measure the impact of an upgrade on IRISLOG performance, we execute our benchmark workload (of Section 5.3) before, during, and after an upgrade.

During an upgrade, the IRISLOG application proxy processes queries by submitting them in parallel to all versions, and returning the longest result. The longest result is chosen in order to mask data that is missing from a new version during the interval between the new version being brought online, and the time when all old data has been copied to the new version. Data updates are processed by issuing the update to all versions, and returning success if and when all versions return success. We do not terminate the old version in this experiment. Hence, during the "post upgrade" period, both versions are executing.

The experiment is repeated 5 times. Figure 17 presents the average query and update latencies during and after an upgrade. The figure repeats the Native execution and Version Manager (non-upgrade) data from Figure 14, for ease of comparison. Results for the upgrade period include a total of 24 country queries, and 72 region queries. All other bars include at least 140 trials[10].

As expected, the figure shows that performance does degrade when two versions are running (i.e., during and post upgrade). The relative overhead during and after upgrade is relatively similar. The overhead during the post-upgrade period ranges from 15% for updates, to 51% for country

[10]The difference in number of operations is due to our choice of workload, and the fact that the number of operations executed during the upgrade period depends on the length of the upgrade process.

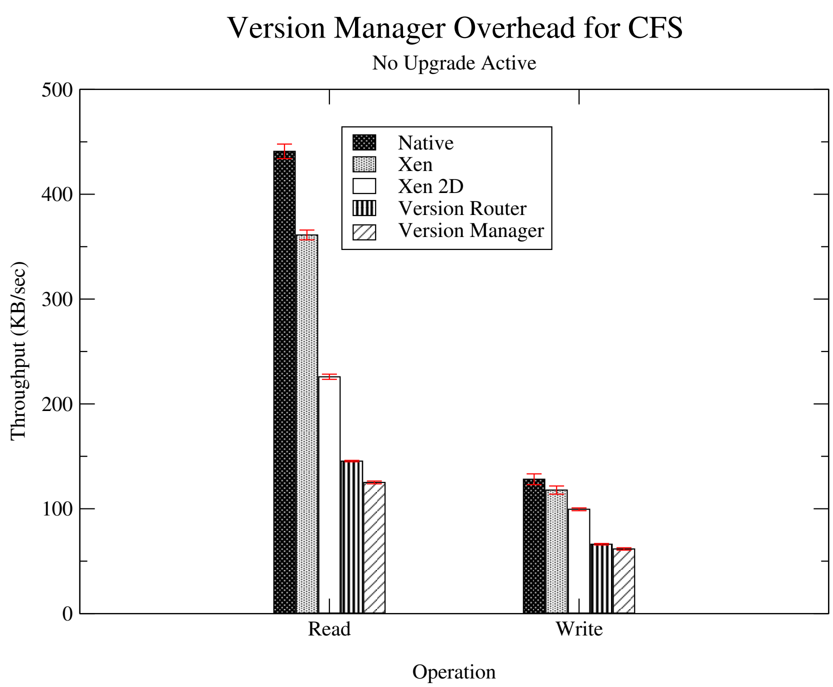

Figure 16: Overhead of running CFS with our system. Configuration in the legend correspond to depictions in Figure 15.

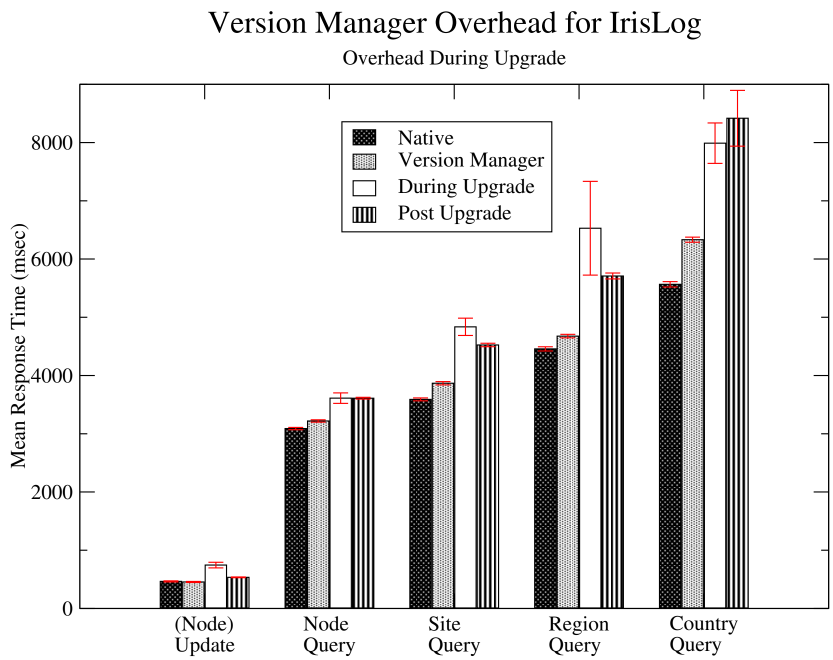

Figure 17: IRISLOG performance during upgrade

queries. We believe that the larger queries incur more overhead for simultaneous queries since there are fewer idle resources in the system for executing the second instance of the system.

6 Related Work

Several researches have looked at the problem of upgrading large-scale distributed systems. The most relevant of these are: (a) work on upgrading the Internet routing infrastructure [17], and (b) work on upgrading classes in object-oriented databases [18].

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 [17] 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 [19] or by limiting the types of changes that can be made [20, 21, 22]. More recently, Boyapati et al.'s work [18], 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 without 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 contexts. 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.

7 Discussion

In this paper, we have proposed the use of *simultaneous execution* in order to ease the challenging task of upgrading a distributed application service. We used our implementation of the concept, the Version Manager, to upgrade CFS and IrisLog, two distributed applications which are substantially different in their design.

While we have not tackled testing or recovery from failed upgrades in this paper, simultaneous execution can ease these tasks as well. Simultaneous execution can improve the coverage of testing by enabling the testing of new software under field conditions. To do so, the software testing team would write an application proxy that sends requests to both the current version, and the version to be tested. The proxy would compare replies from these versions, providing developers feedback on possible errors. Such a proxy would be similar to recent work for NFS testing [23].

Simultaneous execution also offers a straightforward approach to handling buggy upgrades. During the simultaneous execution period, any operation modifying the state of the system is routed to all running versions. If a bug is found in the version before the old version has been retired, we recover by simply retiring the new version. Note that even if the bug is found only after the old version has been retired, some recovery is possible by reverting to a "known good" state.

While simultaneous execution promises these benefits, the performance of our prototype implementation is limiting. In addition to typical profiling and optimization, a number of possibilities exist for improving the performance of our system. First, we believe our heavy use of a non-congestion-controlled transport protocol (UDP) for relaying traffic between multiple domains is unusual, one which Xen is not explicitly tuned for. Second, given the well-defined functionality of the Version Router, it could be re-written as a kernel module. A third possibility is to run the application in the same domain as the Version Router and Application Proxy, eliminating cross-VM communication costs. Finally, as the Application Proxy is only required when multiple versions are executing, the proxy could be loaded as part of the upgrade process.

8 Conclusion

In this paper, we have described the simultaneous execution approach to upgrading large distributed systems. This approach relies on: 1) viewing different versions of a distributed application as separate deployments, and 2) using virtual machine technology to effectively share a common infrastructure between the different active versions. We described a prototype implementation of our design, called the Version Manager. This prototype incorporates a number of techniques to help developers resolve any application specific processing that must be done as part of using simultaneous execution. Finally, to demonstrate the practicality of the simultaneous execution approach to upgrades, we adapted the Version Manager to support upgrades of both the CFS distributed file system and the IRISLOG monitoring system. Our experience with these applications indicates that the Version Manager needs relatively little new code to support new applications and that the Version Manager is effective at propagating updates to different nodes in the system. We also observed that the Version Manager does degrade performance in some scenarios. However, we believe that the additional functionality provided by the system far outweighs its overhead in most situations.

References

[1] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the 18th Symposium on Operating System Principles*, (Chateau Lake Louise, Banff, Canada), Oct. 2001.

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

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

[4] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols*, 2001.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network ," in *Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols*, (San Diego, California), Aug. 2001.

[6] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[7] 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.

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

[9] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?," in *USITS2003*, 2003.

[10] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in internet services," in *OSDI2004*, 2004.

[11] J. Cates, "Robust and efficient data management for a distributed hash table," Master's thesis, Massachusetts Institute of Technology, 2003.

[12] T. Roscoe, L. Peterson, S. Karlin, , and M. Wawrzoniak, "A simple common sensor interface for planetlab." PlanetLab Design Notes PDN-03-010, 2003.

[13] A. Deshpande, S. Nath, P. B. Gibbons, and S. Seshan, "Cache-and-query for wide area sensor databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.

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

[15] A. Bharambe, C. Herley, and V. Padmanabhan, "Understanding and Deconstructing BitTorrent Performance," Tech. Rep. MSR-TR-2005-03, Microsoft Research.

[16] M. Ripeanu and I. Foster, "Mapping gnutella network," in *IPTPS*, 2002.

[17] 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.

[18] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richmann, "lazy modular upgrades in persistent object stores," in *OOPSLA2003*, 2003.

[19] M. Atkinson, M. Dimitriev, C. Hamilton, and T. Printezis, "Scalable and recoverable implementation of object evolution for the $PJama_1$ platform," in *POS9*, (Lilihammer, Norway), Springer Verlag, September 2000.

[20] D. J. Penney and J. Stein, "Class modification in the GemStone object-oriented dbms," in *OOPSLA87*, (Orlando, Florida), pp. 111–117, 1987.

[21] B. S. Lerner and A. N. Habermann, "Beyond schema evolution to database reorganization," in *OOPSLA90*, (Ottawa, Canada), pp. 67–76, 1990.

[22] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth, "Semantics and implementation of schema evolution in object-oritented databases," in *SIGMOD87*, (San Francisco, CA), pp. 311–322, 1987.

[23] Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger, "comparison-based file server verification," in *USENIX-ATC05*, (Anaheim, CA), April 2005.