1 Introduction

Distributed applications, and the communication networks that connect them, have changed the way people work and play. To maintain their success over time, these systems must accommodate change with minimal disruption and risk. The state-of-the-art, however, is seriously deficient in meeting this crucial requirement. This deficiency both slows the pace of innovation, and leads to critical failures.

Examples of innovations that have failed to reach users include new network services such as IP Multicast [Dec88], Integrated Services [CSZ92], Differentiated Services [Kil99], and active queueing mechanisms such as RED [FJ93]. The consequences of failed upgrades include losses of tens of millions of dollars in revenue [Koc04]; flight cancellations at busy airports [BBC04]; a day-long total failure of the AT&T frame relay network [McC99], affecting thousands of direct customers; and week-long intermittent failures of one of MCI’s frame relay networks [Roh99], also affecting thousands of direct customers.

Distributed systems can differ substantially in their utilization of available resources. Some systems, such as cooperative backup or sensor networks, may require little more than the idle resources of personal computers. In contrast, the hardware for communications networks is immensely expensive. Thus, such systems are likely to have less headroom. Accordingly, our work tackles the challenges of upgrades for two types of systems, which differ greatly in the abundance of resources available to them. These systems are distributed applications, and Internet service provider (“ISP”) networks.

For distributed applications, which may live in resource rich environments, we propose and evaluate a methodology, simultaneous execution, which exploits the abundance of resources to execute multiple versions of an application simultaneously, but independently. This simultaneous execution methodology dramatically reduces interoperability requirements, enabling upgrades with minimal service disruption and minimal developer effort.

For ISP networks, in which resources are less plentiful, we pursue lighter-weight methods for maintaining service availability during the change process. Our approach here is guided by the observation that, while there are insufficient resources to replicate the entire network, it is still possible to exploit some idle
Figure 1: Distributed Application Architecture

or backup resources to reduce the disruption caused by change. The key idea is to minimize disruption by moving or re-homing customers away from the equipment scheduled for maintenance.

2 Evolving Distributed Applications

As illustrated in Figure 1, a distributed application is composed of a collection of nodes, possibly geographically distributed, which cooperate (through some set of protocols) to provide a service. Clients access the provided service through a client protocol, but do not concern themselves with the server-to-server interactions. We focus on changes to the server protocols.

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. We elaborate on each of these points.

Interoperability To avoid shutting down a system completely for upgrades, new software must be designed to interoperate with extant software. However, this approach to system evolution severely constrains the nature of feasible changes, prohibiting changes to algorithms that control message routing, load-balancing, and cooperative caching. Additionally, this approach imposes a heavy implementation and maintenance burden on software developers.

Testing 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.”
**Recovery** 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.

**Deployment process** Large scale systems require robust, automated processes for the deployment of new software throughout the system. Such processes must (i) provide recovery mechanisms to deal with the possibility that the installation of a new version fails on some nodes, and (ii) cope with the possibility that some nodes are offline at the time an upgrade is initiated.

### 2.1 Related Work

#### 2.1.1 Naive Approaches

A straightforward approach that permits arbitrary change is to stop the entire system, back up any persistent state, and then restart the new version. This approach eliminates the interoperability requirement. However, stopping a large-scale production system is too disruptive to be practical. An alternate approach is motivated by the observation that many distributed applications have facilities for coping with node failure. Taking advantage of this property, we can upgrade the system one node at a time. Doing so would be less disruptive than the “stop-the-world” approach described above. However, the node-at-a-time strategy does little to simplify the interoperability, testing, recovery, or deployment process problems.

#### 2.1.2 Object-Oriented Databases

Work on upgrades in object-oriented databases can be classified into two groups: scheme evolution and schema versioning. Scheme evolution focuses on the problem of migrating data to a new database schema. Key challenges addressed by this body of work are how data fields are translated: either through generic or programmer-defined conversion functions [PS87, BKKK87, LH90], and the management of dependences between objects undergoing translation [FMZ+95, BLS+03]. Our work assumes that the application developer will provide a data conversion tool (if necessary), and focuses instead on the communications between nodes implementing a distributed service.

Work on schema versioning, is closer to our own in that it enables applications to access a database simultaneously through both old and new schemas. This is accomplished by limiting write access to a single version or copying objects on writes [KC88], or invoking conversion functions every time an object is accessed through the new schema [FL96]. Key differences in our work are that we support a broader range of applications (beyond databases), and provide a framework that supports testing and recovery.

---

1 Or in groups, depending on the degree of fault-tolerance of the system.
2.1.3 Runtime Patching

A number of researchers have worked on mechanisms for patching running applications [Dmi01, LKKL01, HMN01]. While this is a useful facility in some contexts, we believe it is inappropriate for large changes to complex applications.

2.1.4 Risk Mitigation

The embedded systems community has proposed a number of methods for reducing the risk of introducing new software into a running system [SRG95, TTA+02, CD99]. These methods retain old software after an upgrade, and use the old software to sanity check or complement the new software. In contrast to our work, these methods assume that the interface between the components to be replaced and the rest of the system are unchanged as a result of the upgrade.

2.2 Our Solution: Simultaneous Execution

Having identified the challenges posed by upgrades for distributed applications, and outlined related work, we now give an overview of our solution, and the challenges posed by our approach.

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 simultaneous execution period allows the system to deploy and switch to the new version while still providing availability through the previous version. After the new version is ready to run, we terminate the old instance.
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 designs. Because simultaneous execution permits the deployment of new versions without disrupting existing versions, simultaneous execution can be used to test new software under field conditions, thereby improving testing. Simultaneous execution also reduces the risk of upgrades. As long as the old version is kept running, we can revert a failed upgrade by sending client requests to the old version rather than the new. Finally, 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.

While treating the versions as separate systems simplifies many aspects of upgrading a distributed system, it introduces a problem of its own. Specifically, the existence of simultaneous execution must be masked from clients, so they see a consistent system, regardless of which version is running on the server they contact. The problems are threefold. First, data present in the old version before the upgrade must be made available during the new version. Second, any modifications made during simultaneous execution must be visible in all versions. Third, the system must either prohibit different versions from accepting conflicting writes, or must provide a mechanism for resolving such conflicts. Our experience with existing applications, however, suggests that addressing these problems are not a significant burden.

2.3 Status

2.3.1 Completed Work

In order to evaluate the practicality of simultaneous execution, and to demonstrate its benefits, we have created a prototype system to support simultaneous execution, called Version Manager, and used it to support upgrades of two existing distributed infrastructures that differ substantially in their designs. These infrastructures are the CFS distributed storage system \(\text{DKK}^+01\) and the IRISLog distributed infrastructure monitoring system \[\text{iria}\]. Here, we describe our experience in using Version Manager to upgrade IRISLog in a testbed environment. More detailed results are presented in \[\text{ANS05}\].

**IrisLog 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 \[\text{irib}\], 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\(^2\) (e.g., CPU load, per node bandwidth usage, per slice memory usage etc.) of those nodes.

\(^2\)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).
At each PlanetLab node, IRISLog uses different PlanetLab sensors [RPK+03] 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 [DNGS03]. Both IRISLog and IRISNet are written using Java.

**Applying Version Manager to IrisLog**

As explained in Section 2.2, applying Version Manager to an application requires the resolution of three problems: data migration, write visibility, and write conflicts. Herein, we explain how we resolved these problems for IRISLog.

We resolved the data migration problem for IRISLog by writing a small tool which uses existing IRISLog APIs to retrieve the database from the old version, and insert it into the new version. The tool was written as a thin wrapper over the IRISLog command-line client, and required about 50 lines of shell and Perl code that calls the command-line client.

To ensure that writes are visible to all clients, regardless of the version of the service they contact, we developed a proxy which interposes between clients and servers. The proxy runs on every server node. The proxy processes writes by submitting them to all versions, and returning success if and when all versions return success. The application-specific portion of this proxy is approximately 300 lines of C++.

No special effort was required to resolve write conflicts for IRISLog. This is because IRISLog data has single writer: the corresponding sensor. Accordingly, write conflicts will not occur.

**2.3.2 Future Work**

We see three areas of future work with Version Manager. First, while simultaneous execution and Version Manager enable greater testing coverage, we have not prototyped and demonstrated such use of our system. The same is true for recovery from failed upgrades. The final area for future work is improving the performance of our system, which currently exhibits slowdowns up to a factor of four for CFS.

---

3 These APIs are used mainly for replication and load-balancing purposes.
We believe that recent work on comparison-based testing [TWSG05] nicely demonstrates the principles necessary for testing distributed applications using simultaneous execution. Additionally, we believe the principles behind recovery in our system are relatively simple. Accordingly, our future work will focus on understanding the factors with account for the performance of our system.

3 Evolving ISP Networks

3.1 The problem

Figure 3 depicts a typical Internet service provider network, and customer connections to that network. Customers connect to the ISP network through access links. The primary components within the ISP network are backbone routers, access routers, and links connecting routers. Links between routers may be wide area links, connecting routers in different points of presence (POPs), or local area links, connecting routers within a single POP. Significantly, the typical connection between a customer and the ISP is non-redundant. Thus, access links and access routers are single points of failure.

Software upgrades for routers are designed to interoperate with older versions, and other changes to the network are generally undertaken with interoperability in mind. Accordingly, upgrades and other changes proceed on a node-by-node basis. Unfortunately, because access routers are single points of failure, this node-by-node upgrade process is highly disruptive. In particular, activities such as access router software upgrades cause downtime for customers on the affected routers.
The problem is exacerbated by the frequency of failures for IP networking equipment, as compared to other networking technologies; the frequency of patches for IP router software; and the time required to complete activities such as upgrades. In particular, whereas the service goal of “five-nines” of reliability permits only $5\frac{1}{4}$ minutes of annual down-time, access router software upgrades typically require several times that duration.

### 3.2 Related Work

**Operational Practice** It is common practice today to reduce the impact of maintenance activities on backbone routers by the use of a *cost-out* procedure. This procedure increases the routing protocol metric for links incident to a backbone router before planned maintenance actions are taken on that router. The increase in the metric causes other routers in the network to choose alternate paths, as if the router for which maintenance is planned were already out of service. The cost-out procedure is not applicable to access routers, however, because customer routers are typically connected to a single access router.

**Vendor Offerings** Cisco, the dominant vendor of routers used in ISP networks, offers a variety of high-availability options. These include Redundant Processor Support [Cisa], Route Processor Redundancy Plus (RPR+) [Cisb], and Stateful Switchover [Cisc]. These options use redundant hardware to reduce the time required to recover from the failure of the route processor. While such features can reduce the outage time by keeping the router in service while the new software boots, the switch-over process causes significant disruption, as line cards are reset, forwarding tables are cleared, and BGP sessions are lost. An alternative solution is 1+1 redundancy, as described in [Cis04]. However, such a solution doubles capital and operational costs.

**Research Approaches** In [SYL+03], Sebos, Yates, et al. propose a scheme for economical recovery from access router failures. The scheme assumes that customers’ access links to the ISP are reconfigurable. When an access router fails, the access links for customers homed on that router are reconfigured to terminate at a spare router in the ISP network. The customer configurations are retrieved from a configuration server, and installed on the spare router, thereby restoring service.

### 3.3 Our Solution: RouterFarm

To minimize the disruption caused by network upgrades, and to enable more dynamic management of the network edge, we propose *RouterFarm*, a network architecture with re-homing support. In RouterFarm, customers are decoupled

---

4The route processor is used to execute the control plane of the router. On lower-end routers, the route processor executes the data plane as well.
from routers, and routers are treated as interchangeable boxes, similar to servers in a server farm.

Our work takes inspiration from [SYL⁺03], exploring the use of the technique in the realm of network evolution, and delving more deeply into the challenges that must be resolved to make the technique practical. The expected benefits of RouterFarm include:

- reduced down-time due to planned maintenance activities (such as software upgrades), by moving customers to spare or idle routers during the maintenance activities
- reduced down-time due to router failures, in the same manner
- interface protection (maintenance on, or failure of, individual line cards)
- faster network evolution, as maintenance activities are no longer limited by customer schedules
- load balancing, such as when the route processor is overloaded with BGP sessions, or a router’s uplink is saturated
- masking of faults, by migrating customers when a failure is imminent (such as when the chassis temperature exceeds a threshold)
- simplified migration of customers from legacy hardware

Realizing these benefits requires resolving important challenges. In the case where customers to re-homed to spare resources with the same POP, the primary issues that arise are management of router configuration as customers are re-homed, and understanding the performance (e.g., down-time and time to completion) of rehoming. Rehoming customers across POPs introduces additional challenges, including the geographic placement of backup resources, mapping of customers to backup resources, route de-aggregation, traffic demand shifts, and cross-layer coordination of link-layer transport capacity. In this thesis, we intend to focus on intra-POP rehoming. We explain the problems to be addressed in Section 3.4. We may additionally investigate some of the inter-POP rehoming issues, as time permits (Section 3.5).

3.4 Challenges to be Addressed

In this thesis, we plan to focus on the configuration and performance modeling issues that arise in realizing the RouterFarm concept. We next detail these challenges.

3.4.1 Configuration

As part of the re-homing process, the new home router for a customer must be configured to provide the customer with appropriate service. This requires three specific capabilities: the ability to identify a customer’s services (as different
services may have different re-homing constraints), the ability to extract the customer configuration from the customer’s current home router, and the ability to patch the customer configuration into the new home router.

A significant obstacle to providing these capabilities is the router configuration language itself. The most common configuration language, used by Cisco and most other vendors, lacks any formal grammar or underlying model. Consequently, given a router configuration, it is difficult to identify the portions relevant to a customer. Similarly, it is difficult to guarantee that the configuration to be patched into the new home router will function as desired.

Given the difficulty of automating configuration extraction and patching with today’s routers, our work on rehoming support will begin with the manual creation of procedures to handle these tasks for a few common services and options. In addition to providing rehoming solutions for these services, this portion of the work will give us insight into the problems that the current configuration language poses for rehoming.

The identification of common services and options is itself an important and interesting problem, which we aim to address in our work. While ISPs are likely to have engineering rules that limit the possible services and options, such rules may not be comprehensive. Furthermore, these rules may be deliberately violated in special cases. Accordingly, we plan to use automated classification and clustering techniques to identify services and configuration options used in a large provider network.

The third component of our work on configuration issues will be the design of a configuration language which is safe for re-homing. Our initial thinking is that the key to such a language is to keep customer configurations separated from each other, and from general router configuration, as much as possible. To maintain the aggregation that is necessary for high performance, a compiler will generate optimized configurations.

The expected contributions of this work are the tools for classification of common configurations, the tools supporting re-homing of these configurations, and a configuration language design which simplifies the re-homing process.

3.4.2 Performance Modeling

At the heart of the RouterFarm concept is the capability to re-home customers. As enumerated in Section 3.1, such a capability would ease many network maintenance tasks. However, each such task may have different requirements. While it will be important, in all tasks, to minimize the customer impact, the requirements for other metrics will differ. In particular, tasks such as failure recovery will place emphasis on minimal time to completion, whereas tasks such as planned maintenance will be more concerned about the risk of additional service disruption from the re-homing process.

Just as there are many applications for the re-homing primitive, there are varied procedures for realizing re-homing. For example, when multiple customers must be re-homed, it is possible to re-home them either one-at-a-time (serially), or many-at-a-time (in parallel). These two approaches will naturally
differ in their completion times and levels of risk.

In order to choose appropriate procedures for different network maintenance tasks, it is essential to understand the performance characteristics of the different procedures. The most straightforward approach to this problem is to simply create a testbed, and evaluate the procedures in this testbed. However, it is infeasible to create a testbed which replicates the scale of production systems. In particular, modern IP access routers can support hundreds or thousands of customers. To evaluate the behavior of re-homing procedures in such a system, with full fidelity, would require hundreds or thousands of customer routers.

We propose to attack this problem using a combination of emulation, modeling, and simulation. Specifically, we will begin by emulating the behavior of small scale systems using a router testbed. We will use these experiments to build a model of the factors, such as timers, communication delay, and CPU load, which determine the performance of re-homing procedures. We will evaluate these models by comparing their predictions for the time to restore customer reachability against the behavior observed when routers are rebooted in a large production network. The validated model will then be used to drive simulations at full scale.

The models should be able to answer questions such as: How does time to completion scale in the number of routes and number of customers? How does the maximum outage duration (across customers) scale in the number of routes and customers? How do network-wide metrics, such as defects per million (unavailability weighted by number of affected customers) scale? How do different procedures compare on these metrics? The target scale for the modeling work is the number of customers homed on a single access router. In future work, modeling may be extended to include the effect of larger scale maintenance tasks on the network backbone.

The expected contributions of this work are threefold. First, the model will provide an understanding of the constraints for re-homing procedures given current network equipment. Second, simulation of different procedures at scale will yield guidance about the appropriateness of re-homing procedures for different network maintenance tasks. Third, where simulation suggests that existing procedures are inadequate for the tasks at hand, we will suggest alternate procedures (where possible), or architectural changes which would enable the infeasible tasks.

3.5 Other Directions

As noted in Section 3.3, realizing a general RouterFarm, in which customers may be rehomed between POPs, will require the resolution of additional challenges. We detail two of those challenges here.

Cross-layer coordination of link capacity As part of the re-homing process, we must create new link layer connections between customer routers and access routers. In the case of remote rehoming, these link layer connections will traverse longer spans of the physical network than the connections they replace,
possibly causing congestion. This may hinder the ability of the link layer to adapt to link failures, such as fiber cuts. Accordingly, it is important to provide for coordination between the rehoming system and the link layer.

**Route deaggregation** To minimize the size of routing tables in a provider network, network addresses are assigned by geographic hierarchy. This enables, for example, all customers homed in a particular city to be represented by a single routing table entry in the network backbone. Some applications of RouterFarm, such as disaster recovery for a large city, may require the re-homing of customers of a single POP to backup resources across multiple POPs. A naive splitting of customers across POPs, risks greatly inflating the number of routes present in backbone routers, and thus, causing routing instability [CGH02]. It is important to understand this risk, and, if necessary, develop procedures that minimize it.

4 Timeline

5 Expected contributions

The expected contributions of this work fall into three broad categories: software artifacts, new upgrade methodologies, and insights into the design of upgradeable systems.

- **Software**
  - Version Manager software for upgrading distributed applications in resource rich environments.
  - A rehoming toolkit including tools for the analysis of existing configurations, modeling of rehoming performance, and migration of customers between routers.

- **Methodologies**
  - Simultaneous execution: a methodology for resource rich environments that provides most of the flexibility of a “stop-the-world” upgrade, but with availability similar to a “node-at-a-time” upgrade.
— Re-homing: a methodology for resource constrained environments which reduces the disruption caused by “node-at-a-time” upgrades, by using spare resources to dynamically eliminate single points of failure.

• Insights

— Our work on distributed application upgrades demonstrates that, in resource rich environments, it is possible to evolve server-to-server protocols without requiring the application to provide explicit support for upgradeability.

— Realizing upgrades via re-homing requires an understanding of how the components of a system interact, to ensure that these interactions are preserved after re-homing. We expect our work on configuration challenges (Section 3.4.1) to yield insights into how a system can express these interactions clearly.

References


