# Development Tools for Distributed Applications

Mukesh Agrawal Srinivasan Seshan
*Carnegie Mellon University*

## Abstract

The emergence of the global Internet has dramatically broadened and changed the computing landscape. In particular, much of the value in contemporary computing systems derives from networked applications. Prominent examples include e-mail, Usenet news, the World Wide Web, and the many varieties of peer-to-peer networks.

However, the number of successful, large-scale, truly distributed, applications is exceedingly small. We argue that a major reason for this is that tools and other facilities available to aid the developers of these applications are inadequate. We propose a life-cycle for these applications, identify challenges that must be met to make the model viable, and detail our initial work towards meeting these challenges.

## 1 Introduction

The Internet is not living up to its potential. While the Web has been a tremendous success, providing millions of non-technical users with convenient access to information and the ability to perform transactions on-line, the number of truly distributed applications that have succeeded on the Internet is shockingly small. Even those few applications that support interaction among clients, such as chat rooms, auctions, and file-sharing services, require all operations to pass through a common server. As the success of content distribution networks (CDNs) and peer-to-peer (P2P) applications have shown, there is clearly a great demand for large-scale distributed applications. The major barrier to supporting these, and even richer, applications on the Internet is the difficulty of designing, building, testing, and maintaining distributed applications using the tools that comprise the state-of-the-art today.

We can draw a parallel between the complex task of product development and computer application development. In general, manufacturers explicitly manage the life-cycle of their products. Moreover, they typically have specific tools to support different life-cycle phases, e.g., monitoring tools and statistical packages for quality control, and specialized CAD tools for product design. We believe that the analogous life-cycle for an application would have five stages: the design stage, the implementation stage, the testing stage, the deployment and operation stage, and the maintenance and evolution stage. While good tools exist for the life-cycle stages of traditional non-networked applications (e.g., debuggers, profilers and logging tools), no such tools exist for distributed applications. The goal of our work is to provide a tool chain that supports each of the stages in the life-cycle of Internet applications.

Supporting the life-cycle stages of applications in traditional distributed environments has received a great deal of attention in the past. For example, a wide variety of tools are available for traditional distributed systems. These range from simple communication libraries such as MPI (for scientific computing) to comprehensive environments such as Corba (for enterprise applications). However, these tools target smaller scale, mostly closed environments, which are fundamentally different from the Internet.

Recent efforts have begun to address these same challenges in the Internet context. For example, efforts in DHTs [**?**] and self-organized overlays [2] are developing a collection of building blocks that help in the implementation of large P2P applications. Similarly, simulation tools like ns-2 [3] and open testbeds like Emulab [5] and Planetlab [**?**] have provided excellent platforms for the comparison of different designs choices. However, the research community has largely overlooked the later stages of the life-cycle – specifically, testing, deployment and evolution of these applications. Additionally [1], possibilities for integration of tools from different life-cycle stages are as yet unexploited. For example, the models of system behavior generated during the design stage may prove useful in debugging errant behavior during operation.

In this paper, we describe some of the challenges in addressing the needs of distributed applications in

[1] We thank Margo Seltzer for this observation.

the later life-cycle stages. Our initial work in supporting distributed applications has concentrated on the problem of maintenance and evolution – specifically the problem of upgrading a distributed application and possibly rolling back an upgrade. We describe the challenges in addressing this problem as well as some of our initial solutions in Section 2. In Section 3, we discuss some of the issues in our next area of focus – testing and debugging deployments of distributed applications. We summarize our observations and conclude in Section 4.

# 2 Maintenance and Evolution Stage: Software Upgrade/Rollback

An important capability that our distributed application life-cycle requires is the ability to upgrade to new software versions, and when necessary, to revert to previous software rapidly and easily. We refer to this as the software upgrade/rollback problem. In this section, we explore the challenges in addressing this problem and describe part of the design space of possible solutions.

Broadly, upgrades can be classified as *synchronous* or *asynchronous*. A synchronous upgrade is one in which the software on all nodes must be upgraded near-simultaneously. An asynchronous upgrade, in contrast, does not require coordination amongst nodes of the distributed application. Synchronous upgrades make more demands on the upgrade/rollback system than asynchronous upgrades.

An example of a distributed application that might use asynchronous upgrades is a web server cluster. The prototypical architecture for a web server cluster might consist of a load-balancer, a number of identical web servers, and a back-end database. Because the web servers are, in some sense, replicas, the unavailability of a single web server node does not greatly impact the overall service. Also, the web servers do not need to run the same software release in order to maintain the correctness of the overall service. Accordingly, upgrade and rollback need not be coordinated amongst the web server nodes and can be done asynchronously.

Routing applications are examples of distributed applications that may require synchronous upgrades. In contrast to the web server cluster, the nodes are not replicas, so the unavailability of a single node will force the unavailability of any resources unique to that node. In addition, the operation of the routing nodes is not independent. They must cooperate in order to provide service to their clients. Thus, we must either design upgrades to be backwards compatible, or we must coordinate upgrade (and rollback) amongst the routing nodes.

## 2.1 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 [4] and work on upgrading classes in object-oriented databases [**?**].

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 [4] 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.

Liskov et al.'s work [**?**] focuses on upgrading classes in object oriented databases, where it is essential to preserve object state across upgrades. Note that our problem differs significantly from the problem considered by this effort. In our problem, we assume that the distributed application does not require persistent state. Under this assumption, which we believe will hold for many distributed applications, upgrade and rollback can be handled with significantly less developer effort.

## 2.2 Challenges/Requirements for Upgrade/Rollback

Based on the discussion above, a solution to the upgrade/rollback problem requires specific capabilities:

1. An upgrade/rollback solution should not require excessive change from existing processes for the distribution and installation of software.

2. A solution must enable software rollback with minimal operator intervention. Ideally, when an operator initiates the rollback procedure, the system automatically reverts all necessary state, such as program files, configuration files, etc.

3. The upgrade and rollback procedures should operate quickly.

4. The solution should minimize the down time due to upgrades and rollbacks. Note that this requirement can be relaxed for applications such as a web server cluster, because the application architecture itself can compensate for the unavailability of individual service nodes.

Note that the third and fourth requirements are not equivalent. It might be possible, for example, to minimize down time by running the upgrade procedure "in the background". But this might conflict with the goal of upgrading the software quickly.

## 2.3 Solutions to Upgrade/Rollback

We now consider approaches to addressing the challenges of upgrade/rollback. We consider the asynchronous case first, and the synchronous case next.

### 2.3.1 Asynchronous Upgrade and Rollback

In the asynchronous case, no coordination between nodes is required to effect a successful upgrade. Accordingly, assuming that the application architecture can compensate for the unavailability of individual service nodes, techniques used for upgrade of single nodes can be reused. We briefly discuss two of these techniques.

**Packages** Given that software is often distributed as packages, and managed by package management systems, such as the Redhat Package Manager, a natural approach is to leverage these systems to manage software updates and rollbacks. Our upgrade procedure might be as follows: shut down the service, copy system configuration files, upgrade the software (taking care to log the prior version of the upgraded software), and then restart the service. The rollback procedure would be to shut down the service, remove the upgraded software, reinstall prior versions, restore configuration files, and then restart the service.

This approach well achieves the first and second goals of Section 2.2. Depending on the number of packages involved in an upgrade, and the complexity of their installation scripts, however, the corresponding rollback operation may take some time to complete. Thus the approach may not achieve the third goal.

**File System Checkpoints** An alternative approach, that addresses the speed concern (while maintaining the other goals), is to use file system checkpoints. To upgrade the software, we shut down service on a node, checkpoint the filesystem, and then restart service. To roll back the software, we shut down service, roll back the filesystem, and then restart service. Because this approach avoids the need to execute uninstallation and installation code, it may better meet the speed goal.

### 2.3.2 Synchronous Upgrade and Rollback

We now consider the problem of synchronous upgrades. We consider the specific problem of an upgrade which is both synchronous, and not backwards compatible. An example of such an upgrade would be switching an application's routing protocol from distance vector to link state.

In designing a solution, we draw 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, in our approach, the upgrade process begins with simultaneous execution of old and new versions on the application nodes. The simultaneous execution provides an opportunity to to bootstrap the new application instance, and thus, minimize unavailability due to the upgrade. After the new version is ready to run, we terminate the old instance.

To support simultaneous execution, we employ virtual machines (VMs). VMs provide the illusion of an independent computing machine while running as a process on some other machine. In this context, the VM is referred to as a *guest*, and the other machine is called the *host*.

For our VM, we choose User Mode Linux (UML)[**?**], which provides a virtual Linux machine running as a process on a Linux host. Two features of UML are required for our purpose. First, it provides the ability to route network packets between the host and guest. Second, it supports copy-on-write filesystem images. To use the copy-on-write facility, the user specifies a base filesystem image, and a copy-on-write file. Both files are stored in the host filesystem. The base filesystem is treated as read-only, and any changes to the guest file system are written to the copy-on-write file.

To employ UML for upgrade/rollback, the distributed application is installed in a UML VM. Before a software upgrade, we duplicate the copy-on-write file and create a snapshot of the running virtual machine process. These files are saved in case a rollback is later required.

Additional copies of the process snapshot and copy-on-write file are then made for the VM that will run the upgraded application. We initialize this VM, using the copied files and the same base filesystem image as the original VM. Next, we perform the software upgrade inside the second VM. Because the two VMs use different copy-on-write files, changes to the filesystem by either VM are not seen in the other VM.

At this point, the system is ready to begin simultaneous execution. If, however, both versions of the application listen on the same network port, we must arbitrate access to that port. With some assistance from the application developer, we can construct viable approaches for both datagram communication (UDP) and byte-stream communication (TCP).

For UDP, we require that the application have a way of identifying and dropping messages from incompatible versions. The application might, for example, include a version number in each message. We then have the host kernel deliver all datagrams for the application to both VMs.

For TCP communication, we cannot simply deliver packets to both VMs. Spurious packets would be processed by the VM's TCP stack, possibly confusing it. Thus the filtering must be done on the host machine. To support this filtering, the application developer provides the host machine with filters that the host machine can use to route application requests to the appropriate application instance. To handle connection establishment, connection requests (SYN packets) are answered by the host machine. After the first request packet from the remote end is received, the application version is identified. The host machine then spoofs a connection request from the remote end to the appropriate VM, discards the VM's response, and forwards the request packet to the VM.

Once the upgrade is complete, we terminate the VM running the older software. How, though, do we determine that an upgrade is complete? Our present approach is to consider the message rate of the old application. As new application nodes enter the system, they choose to run the new application version. Concurrently, old nodes exit the system, decreasing the rate of messages for the old application version. Thus, over time, the message rate for the new version increases, and the message rate for the old version decreases. When the message rate for the old application version at a node drops below a threshold, the node terminates the old application version.

| | Process Size (in MB) | | |
|---|---|---|---|
| | 32 | 64 | 128 |
| Snapshot | 0.89 | 1.78 | 10.42 |
| Resume | $< 0.01$ | | |

Table 1: Time (in sec) to Snapshot and Resume Processes

In order to ascertain the viability of a VM approach to simultaneous execution, we have conducted some preliminary experiments on the costs of snapshotting and resuming. Because support for resuming UML is not yet complete, we present results based on a process of similar size (in terms of virtual memory image) as a UML VM. Table 1 presents the time to snapshot and resume processes of varying sizes. Experiments were run on a 766 MHz Pentium III with 256 MB of RAM. Snapshot times are short enough that network connections are unlikely to be disrupted. Note that resume times are consistently low, and independent of process size, because resuming only maps the process' data into memory. The data will be faulted in later as needed.

Based on these results, we believe that a virtual machine based approach is appropriate for providing upgrade/rollback facilities for synchronous upgrades. Note that it may be possible to improve the snapshotting time for large processes by performing *lazy snapshots*. For example, we could mark the pages of the virtual machine process as copy-on-write, and then save the snapshot data in the background.

### 2.4 Open Issues

Open issues remain for both asynchronous and synchronous upgrades. 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 [1] may ease the application developer's burden.

Several issues specific to synchronous upgrades remain as well. First, it is not clear that reverting to the prior process state on a rollback is always the right choice. Routing table data, for example, is often transient. Thus, if a large period of time elapses between the upgrade and the initiation of the rollback, it may make more sense to restart the old software with a clean state (an empty routing

table). Also, if the new application version needs to reflect the volatile state of the earlier application, we may need to log and replay messages delivered between the time that the snapshot is taken, and the time that the new software is ready to handle messages.

## 3 Testing Stage

In the previous section, we described our current efforts for supporting the evolution life-stage of distributed applications. Here, we briefly discuss our future plans for supporting the testing life-stage.

### 3.1 Test Queries

We envision two basic capabilities to support the testing of distributed applications. One is the ability to inject test queries into the application, and the other is the ability to capture the data that will enable us to verify the correctness and performance of the upgrade. We call the components responsible for these abilities the *injection component* and the *validation component* respectively, and consider the requirements for each component.

The injection component is responsible for delivering test cases into the distributed application for execution. A complication in the implementation of an injection component is that the behavior of a distributed application may depend on who originated a request, or where in the network a request originated. The injection component must reproduce these attributes of the test cases in order to accurately test the deployed distributed application. In particular, this means that the injection component is itself distributed, as test requests may be injected at different nodes of the application.

A possible design for a simple injection component is to require every node to support proxy or relay functionality. This would allow remote developers to masquerade as any node in the system. For example, if a test case requires that a message be sent from node A, then the node running the test simply sends/receives messages to a proxy at node A, which forwards them to the distributed applications (and returns messages from the distributed application to the tester). Messages that need to be sent by a particular user are handled in a similar manner. The ability to impersonate a user or machine naturally raises security issues that must be addressed. In addition, while this allows a developer to route via another node, it may not perfectly recreate the tested node's view of the system. For example, the timing of messages might be affected which in turn may affect the behavior of the system. A richer scripting mechanism to control nodes may provide a more accurate remote view of the system but may be much more complicated to implement.

The validation component is responsible for facilitating the checking of the application on the injected test cases. Note that it does not perform the check itself – the responsibility for determining the correctness of application behavior necessarily requires application specific knowledge. The duty of the validation component is to provide the data that enables validation of application behavior. The data gathering must be lightweight in terms of communications cost. The component must also minimize the effort required by the application developer to specify the data to be captured. At the same time, however, it must provide flexibility in the kind of data to be captured.

An important tool to help validation is support for a rich logging system. A developer should be able to specify a set of triggers, a set of data types, and a callback for each data type. Incoming messages at a node can be matched against the set of triggers and the callbacks are invoked to capture data when a message matches a trigger. The triggers can be checked before or after the message is processed by the application. Some key open challenges are the scalable distribution of these triggers and callbacks to the application nodes and the scalable collection of the resulting logs for analysis. It is likely that not all nodes in the system will be involved in the validation, thereby simplifying the task of distribution and collection.

### 3.2 Debugging Support

When developers of traditional applications need to understand application behavior, they often employ source-level debuggers. A debugger provides a developer the ability to study program behavior at a microscopic level. Specific abilities of common debuggers include single-stepping through program execution, live inspection of complete program state, including memory and registers, and post-mortem analysis of program state via *core* files.

How might we provide similar capabilities to developers of networked applications? As a first, naïve, approach, consider building a debugging system for distributed applications using traditional debuggers and a remote invocation facility such as *ssh*. In this approach, the remote invocation facility is used to attach a debugger process to each process of the dis-

tributed application. Unfortunately, this approach approach does not work well. To illustrate, we consider how such a system deals with single-stepping.

How do we implement single-stepping in this system? This is, perhaps, a trick question. While single-stepping is well-defined for a single process, there is no clear analogue for an application with multiple processes. There is *no* obvious order in which we could instruct the processes to execute their next steps, because concurrent execution is rife with the potential for race conditions. Hence, it does not matter which implementation we choose: single-stepping is *inherently* an inappropriate debugging primitive for distributed applications. One important challenge, then, is to find a set of primitives that provide meaningful and useful semantics for distributed applications.

One possibility is to support a less demanding form of application debugging. For example, instead of using traditional debuggers at each node, message logging may provide a useful first approximation to single stepping. Clearly, capturing all the messages generated at all nodes could be very expensive for some applications. Instead, we may limit the log to those messages associated with certain requests, restricted either by message type, or by node using the techniques described for validation in Section 3.1. Sahai et al. [**?**] propose a method of capturing all the messages associated with a request that may be useful for this approach. Either synchronized clocks or Lamport clocks could be used to timestamp events in these logs. Developers could perform real-time visualization of the logs to observe their application in operation. Alternatively, if all messages are collected it might be possible to diagnose problems by replaying the logs within a simulator, where traditional debugging would be possible.

## 4 Conclusions

It is clear that distributed applications are a significant part of the computing landscape. But to fully realize the potential of networked computing, we will need richer development environments that lessen the burden on application developers. We believe that the development of distributed applications can be improved by an appropriate life-cycle model, and tools that support that model.

We have proposed a life-cycle that consists of design, implementation, testing, deployment and operation, and maintenance and evolution stages. Tools to support the first two stages already exist, or are now emerging. For the maintenance and evolution phase, initial work indicates that approaches based on versioning/snapshot filesystems and virtual machines will be appropriate for many distributed applications. To support the testing phase, we propose the message injection and data capture primitives. We also find that traditional approaches to debugging do not map well to distributed applications, and that new primitives will be needed. As a start, we propose rich forms of message logging as an approximation of single-stepping. Finally, while we have not addressed the issue of the remaining life-stages in this paper, we hope to eventually develop a tool chain that helps developers in all stages.

## References


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

[2] Y.-H. Chu, S. G. Rao, et al. A case for end system multicast. In *Proceedings of the ACM Sigmetrics 2000*, pp. 1–12. ACM, Santa Clara, CA, Jun. 2000.

[3] ns-2 Network Simulator. http://www.isi.edu/nsnam/ns/, 2000.

[4] 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.

[5] B. White, J. Lepreau, et al. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 255–270. USENIX Association, Boston, MA, Dec. 2002.