Software Upgrades in Distributed Systems
MIT2002-02

Progress Report: July 1, 2002—December 31, 2002

Barbara Liskov

Project Overview

With the help of support from NTT, we have been working on two projects: support for automatic software upgrades in distributed systems, and work on Byzantine fault tolerance.

Software Upgrades:
Our work on software upgrades is aimed at solving the problem of how to upgrade large scale distributed systems that are intended to provide continuous service over a very long lifetime. Upgrades are needed to correct software errors, to improve performance, or to change system behavior, e.g., to support new features.
Upgrades must propagate automatically: for the systems of interest, it is impractical (or impossible) for a human to control upgrades, e.g., by doing a remote login to a node from some centralized management console and installing the upgrade, taking care of any problems as they arise. Furthermore, upgrades must not interrupt service: upgrades must be installed without bringing the system down.
However, upgrades cannot happen instantaneously. It isn't possible to freeze the system and then cause all nodes to upgrade while the system is frozen. For one thing, this would take too long, since at any moment some nodes may be powered down or not communicating. Furthermore, the system designer may not be willing to have an upgrade happen all at once. Instead, providing continuous service may require that just a few nodes upgrade at a time. Also, the designer may want to try out the upgrade on a few nodes to see that the new code is working satisfactorily before upgrading other nodes.
Upgrades might correct errors or extend interfaces. However, they might also change interfaces in *incompatible* ways: a node might no longer support some behavior or store some state that it used to. Incompatible changes are the most difficult to deal with because they can lead to long periods when nodes that need to communicate assume different interfaces. Yet because of the requirement for continuous service, the system must continue to run (possibly in a somewhat degraded mode) even when such incompatibilities exist.

Byzantine Fault Tolerance:
This project is aimed at developing algorithms and implementation techniques to build practical Byzantine-fault-tolerant systems, that is, systems that work correctly even when some components are faulty and exhibit arbitrary behavior. We believe that these systems will be increasingly important in the future because malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior.
We developed a new replication algorithm, BFT, that can be used to build highly available systems that tolerate Byzantine faults. This work shows, for the first time, how to build Byzantine-fault-tolerant systems that can be used in practice to implement real services because they do not rely on unrealistic assumptions and they perform well. BFT works in asynchronous environments like the Internet incorporates mechanisms to defend against Byzantine-faulty clients, and recovers replicas proactively. The recovery mechanism allows the algorithm to tolerate any number of faults over the lifetime of the system provided fewer than 1/3 of the replicas become faulty within a small window of vulnerability. The window may increase under a denial-of-service attack but the algorithm can detect and respond to such attacks and it can also detect when the state of a replica is corrupted by an attacker.
BFT has been implemented as a generic program library with a simple interface. However, this library requires all replicas to run the same service implementation and to update their state in a deterministic way. Therefore, it cannot tolerate deterministic software errors that cause all replicas to fail concurrently and it complicates reuse of

existing service implementations because it requires extensive modifications to ensure identical values for the state of each replica.
To overcome this, we developed a replication technique, BASE (BFT with Abstract Specification Encapsulation), that corrects these problems. This technique is based on the concepts of *abstract specification* and *abstraction function* from work on data abstraction. We start by defining a common *abstract specification* for the service, which specifies an *abstract state* and describes how each operation manipulates the state. Then we implement a *conformance wrapper* for each distinct implementation to make it behave according to the common specification. The last step is to implement an *abstraction function* (and one of its inverses) to map from the concrete state of each implementation to the common abstract state (and vice versa).
BASE reduces the cost of Byzantine fault tolerance because it enables reuse of off-the-shelf service implementations. It improves availability because each replica can be repaired periodically using an abstract view of the state stored by correct replicas, and because each replica can run distinct or non-deterministic service implementations, which reduces the probability of correlated failures.
BFT and BASE assume that each replica holds a complete copy of the service state. We are also looking at the problem of applying Byzantine fault tolerance to larger services, where the entire service state cannot be stored at each replica. In such a setting it would be desirable to add more nodes to the system as increasing load or storage needs require it, and it is likely that nodes will fail and need to be evicted from the system. This is similar to what happens in a peer-to-peer system, and like these systems we would like to be able to self-organize our system in the presence of a dynamic membership with essentially zero manual intervention. This motivates our new system called Rosebud.

Progress Through December 2002

Software Upgrades:
We have defined an architecture to support automated upgrades. Our approach:

- provides an automatic way to control the scheduling of node upgrades
- enables the system to provide service when nodes are running at different versions
- provides a way to upgrade nodes from one version to the next

To support upgrade scheduling, we provide *scheduling functions* (SFs): procedures that run on nodes and let them coordinate the timing of their upgrades. SFs are defined by the upgrader and support a wide range of behaviors, from upgrading nodes in rapid succession to fix a critical bug to upgrading just a few nodes a day to minimize service disruption. SFs can also communicate with a centralized upgrade database to enable an administrator to monitor and control upgrade progress.
To support communication between nodes running at different versions, we provide *simulation objects* (SOs): adapters that convert calls from one version to the next higher or lower version. SOs let nodes support not only past versions' behavior (i.e., backward compatibility) but also future versions' behavior (i.e., forward compatibility). This is vital for asynchronous upgrades, since upgraded nodes may make calls on non-upgraded ones, and vice versa. And unlike the adapters used in previous work, SOs can contain their own state and so can implement behaviors that are outside the scope of a node's current version.
To support the actual upgrades of nodes, we provide *transform functions*: procedures that convert a node's state from one version to the next.
Our upgrade architecture has the following components. A logically centralized *upgrade server* publishes upgrades for download. An *upgrade database* lets human operators monitor and control upgrade progress and also lets nodes coordinate global upgrade schedules. Per-node *upgrade managers* subscribe to the upgrade server to learn about new upgrades and install them on nodes. Per-node *upgrade layers* gossip with their counterparts on other nodes to learn about new upgrades and handle cross-version calls using simulation objects. The upgrade layer is transparent to applications, so that developers can write their software as if every node in the system were running the same version.

Byzantine Fault Tolerance:
We have presented a preliminary design for Rosebud. This design resembles existing peer-to-peer systems: nodes are assigned random node IDs, and data is partitioned among the participants based on those IDs. Rosebud also provides a simple read/write interface that is common to the existing distributed hash tables that are built on top of current peer-to-peer overlays.
We differ from peer-to-peer distributed hash tables in that our algorithms to determine the current membership of the system and to store and retrieve data are resilient to arbitrary failures of a subset of the nodes.

Our system has a hybrid architecture, consisting of a set of servers – not unreliable client machines that participate in the system intermittently – acting as the peer-to-peer nodes, and a *configuration service* (CS). The CS is responsible for computing the current configuration (including removing faulty peer-to-peer nodes from the system), and informing the peer-to-peer nodes about the current configuration.

Research Plan for the Next Six Months

Software Upgrades:
We plan to start work on an implementation of our architecture. In addition we plan to work on correctness conditions for simulation objects and transform functions.

Byzantine Fault Tolerance:
We have started to work on the implementation of Rosebud, which should be finished in the next six months. We intend to deploy and test the system on a wide-area testbed (such as PlanetLab).