9807-15
Self-updating Software

Progress Report
January 1 — June 30, 1999

Barbara Liskov and Daniel Jackson

Project Overview

Loading and updating software has become a dominant cost in the administration of PCs -- some estimate $5k/machine-year — and a major source of fragility and unreliability. It is also the key obstacle that stands in the way of pervasive computing: the environment of the future in which large numbers of small devices will operate and update themselves without human intervention.

Our research aims to find a systematic and practical solution to this problem. Our current approach has two prongs.

At one end of the spectrum, we are investigating self-updating in the realm of PC-like devices, in which updates come both from the desires of users for new functionality (in "pull mode") and from manufacturers who offer versions that add new features and fix old problems (in "push mode"). The configurations on the devices are large, complex, and heterogeneous: the same application may well be installed in different configurations on different machines. The updating infrastructure cannot assume total control over the system's configuration, but must compensate automatically for changes made by the user, such as accidental deletion of files.

At the other end of the spectrum, we are investigating self-updating in the realm of embedded devices, such as elevator controllers, in which updates come primarily from the manufacturer. The configurations on the devices are smaller and less complex, and may be assumed to be homogeneous; in this respect the problem is easier. On the other hand, "hot updates" must be possible, in which normal operation is not noticeably interrupted by the updating, and in which state is retained across updates, appropriately transformed for the new code. Demands of reliability and performance are higher, and the scheduling of updates across large numbers of networked devices must be addressed.

Progress

Until now, our work has focused on developing the models and architecture that underlie these infrastructures. We have developed a dependency model in which the essential structure of a software system is represented as a graph over components, and a scheme for hot updates that involves filters that identify objects to be updated and transformers to execute the updates.

The key notion in the dependency model is that a component's needs are expressed not by naming the components on which it depends, but by giving specifications they should satisfy. This allows a need to be met more flexibly (since there may be several components that satisfy a given specification) and allows components to be replaced smoothly. We have designed an initial infrastructure based on the model for assembling a systems of Java classes. Currently, specifications are simply uninterpreted strings. The Java package is the unit of granularity; the specification and dependence information is stored in a manifest file in

the package's jar archive. Updating may be initiated in two ways: either by a request for new functionality (eg, the user wants a word processor and none is installed), or by a request to update a particular component or all components.

The dependence model is represented in three places. In the client machine, there is a local database that stores the configurations of all installed systems; from this database, inconsistencies can be automatically detected and repairs can be initiated. In the server machine, there is a database that maps specifications to components: in response to a request for a component that satisfies a particular specification, it provides a location (eg, as a URL) from which the component may be obtained. Finally, components themselves carry information about the specifications they satisfy and their needs. This reduces reliance on the server database, and allows components to be safely downloaded from untrusted sites (since the component's manifest, in which this information is stored, is cryptographically sealed).

Our work on hot upgrades has been conducted in the context of Thor, an object-oriented database with persistence and transactions. We view an upgrade as a transaction that, in a single, atomic step, transforms all objects of one type to a new type. Since an instance of the database may contain trillions of objects distributed over thousands of servers and millions of clients, making the upgrade literally atomic is infeasible. So instead we are investigating a lazy approach that will provide the right semantics but with good performance. We have developed two key ideas.

First, we have a correctness condition. Each upgrade is viewed as a transaction like any other. When upgrades are done lazily, we want this transactional view to still hold as far as user transactions are concerned: in other words, laziness cannot be detected by user transactions.

Second, we have outlined an implementation technique. Its essence is to propagate upgrade information quickly enough that if a user transaction makes use of an object affected by an upgrade, the system knows about the upgrade and can ensure that every object used by that transaction also reflects the effects of that upgrade. We believe our approach will cause no degradation in overall system performance, nor cause delays to user transactions.

Plans for July - Dec 1999

In the next six months, we plan to continue to refine these schemes and to implement them.

In the realm of PC-like devices, we will focus on the consistency problem: ensuring that configurations are consistent, and generating appropriate actions to establish, maintain and restore consistency. The initial infrastructure will consist of a centralized database that stores information about components, and on the local machine, a registry that stores configuration data and a utility that performs installations and repairs configurations automatically. We will begin with a rudimentary server database, and focus on the automatic management of configurations on the local machine. By representing dependence information as simple ASCII files accompanying components in a compressed archive, we hope to be able to wrap existing components and demonstrate the infrastructure on small systems, such as shareware programs like Ghostview and Winzip, and Java programs composed of tens of packages. We expect to refine our dependence model as we learn more about the structure of downloaded packages.

In the realm of embedded devices, we will focus on the problem of lazy updates. We plan to implement on top of Thor, our object-oriented database system that provides persistence and transactions. Using Thor will allow us to leverage our previous work, allowing us to provide an object store in which persistent objects can be upgraded automatically. By the end of the year, we expect to have completed the design of the lazy update scheme, as well as an initial implementation.