Self-updating Software
9807-15

Progress Report: July 1, 1998—December 31, 1998

Barbara Liskov and Daniel Jackson

Project Overview

Self-updating software updates itself at runtime, either autonomously or under the control of a remote authority. We plan to develop the basic infrastructure that is needed to make self-updating software efficient, economical and safe. The problem of self-updating software arises in several important areas:

• System administration. The total cost of ownership of software is now recognized as a major problem; there is a desperate need for technology that reduces the burden of installing, upgrading and removing software. The pervasiveness of PDAs, needing their own administration as well as synchronization with desktop machines, exacerbates the problem.

• Geo-adaptation. As hand-held devices become more common, there will be opportunities to adapt them on the fly to the context in which they are being used. In an airport, the device becomes a flight reminder and booking system; in a university, it becomes a campus map and navigation aid; in a government office issuing driving licenses or passports, it becomes a system for entering form data.

• Remote devices. Infrastructure systems, in particular, will increasingly incorporate large numbers of remote sensors and actuators. Updating the software on these devices in concert with one another will be vital for safe and efficient functioning.

There are many factors that make the problem a hard one:

• No user interface. Some devices will have no user interface at all (in the case of sensors and actuators, for example), or a very limited interface (in the case of pagers and phones). Moreover, in many applications it will be undesirable to require user interaction even when possible: a person should not need to actively download software on entering an airport, for example.

• Computational bounds. Updates are likely to be expensive despite improvements in network technology. Software can be assumed to grow as fast as network bandwidth, and there will always be devices with inferior connectivity. The timing of updates must

be thus be carefully determined; updates might be obtained lazily, just before the computation that requires them.

• Safety. Updates must not cause the device to malfunction; they must not interfere with software whose replacement is not intended; they must not make exhorbitant use of resources; and the updating process itself must not impede other activities or prevent their timely completion.

• Security. In infrastructure systems, it will be important to ensure that rogue updates are rejected. In personal devices, users will want assurance that updates will not install software that reveals personal information across the network.

• Dependencies. One update might require software to be present on the device that has not been previously installed, and will therefore call for further updates. Chain reactions must be controlled. Also, users should be protected from a series of updates that leaves the device unable to operate without an additional update that turns out not to be available. For these reasons, it seems likely that updates should be reversible and should be applicable in any order.

• Incrementality. Updates might concern multiple devices and in this case they must be done incrementally since not all devices affected by an update may be up and communicating simultaneously.

Progress

Our work so far has focused on understanding the problem better (with scenarios) and developing a candidate solution (with an architecture).

• Scenarios. The goal for the work on developing scenarios is to come up with a small set of application examples that we can use to evaluate the usefulness of mechanisms that we propose. Usefulness includes both performance (how costly is it to install a particular update?) and completeness (can all updates of interest be accomplished?). We have identified a large number of examples of interesting updates and we are attempting to categorize them so that we can understand what technical issues they raise.

• Architecture. We have developed a candidate architecture for self-updating. The system consists of an change database, device platforms, and a transport. The change database stores information about updates; the device platforms run on devices and handle the installing of updates on the device; and the transport allows two-way communication between the device platforms and the change database.

The change database is stored at servers; it can be distributed among many servers, which might be maintained by different organizations. In the case of a multi-organization database, there would need to be communication standards established. We have not thought yet about what these standards would be and for now we assume a single organization, but possibly multiple servers nevertheless.

We assume that every device runs our platform. We believe this platform is a lightweight entity, e.g., a small amount of code running on JVM. The platform is capable of being notified of pending updates; requesting updates; checking updates for appropriateness; and installing updates at that platform. The platform maintains a local database, containing information about the local environment, including what classes are running there as well as information about specialized equipment. It uses this information to determine whether updates are appropriate for its device, and it keeps the database consistent with the actual code running locally.

The transport might simply be a network, or there might be additional infrastructure. For example, the change database might push updates to platforms and the pushed information might be cached at a nearby server until the platform is ready to receive it.

Plans

In the next few months, we plan to work on 3 areas. In the first, archeology, we will extend our work on scenarios to a more detailed analysis of the kinds of dependency that arise in existing complex systems. In the second, architecture, we will be continuing to refine and elaborate our infrastructure to support self-updating. In the third, demonstrations, we will be evaluating our ideas, demonstrating their practical application, and discovering new challenges.

• Archaeology. We are beginning a series of investigations into the dependency structures that hold together the software in large systems. We are using a variety of lightweight techniques to obtain information from which dependences can be extracted. These include: runtime monitoring, by instrumenting the opening and closing of files; examination of scripts and logfiles generated by installers and backup utilities, and of dependency databases held by configuration managers such as the Windows registry and Linux's RPM; periodically surveying file modification dates to determine gross access patterns.

• System Design. We plan to elaborate the architecture by developing a design for a prototype system. Here we discuss some preliminary work on the change database. The database stores information about the structure of programs and how that structure is affected by updates. We assume here that programs are composed of objects.

The database will store information about interfaces, classes, and updates. An interface is a set of methods together with a description of what behavior is provided when those methods are called on objects that "support" the interface. An interface can "extend" other interfaces if it provides their methods and behavior. Extension is transitive, and an interface extends itself.

Classes are implementations: they have objects and the objects have methods. Each class "supports" one or more interfaces; this means it provides their methods and behavior. If a class supports an interface, we also say that objects of the class support the interface. Classes also "depend on" interfaces if their code calls interface methods of objects that support the interface. Note that a class might not depend on all interfaces supported by the object; this is important for reasons discussed below.

An upgrade is a four-tuple <source class, target class, filter, transformer>. The upgrade maps objects of the source class to the target class; the filter determines what subset of objects belonging to the source class should be mapped; and the transformer does the mapping: it is a procedure that takes an object of the source class as an argument and produces an object of the target class as an output.

We intend to support "incompatible" updates in which the target class does not support the same interfaces as the source class. The reason we define behavior in terms of interfaces (rather than in terms of entire classes) is to allow recognition of where behavior has not been affected even when the upgrade is incompatible.

• Demonstrations. Work on several demonstrations is underway. The projects include: a self-updating mechanism for Thor, an object-oriented database; an updatable telephone, implemented on a PC with TAPI; a stock price tracker implemented using Java aglets. Here we describe only the first of these.

Thor is a persistent object store designed and implemented by Prof. Liskov's group. It is implemented as a client/server system in which persistent objects are stored at (one or more) servers, and applications run at client machines. The servers store objects together with their code. As an application runs at a client machine, the objects it uses (together with their code) are moved to the cache at the client machine and all application processing takes place at the client on the cached information. Application computations occur as atomic transactions; when a transaction is ready to commit, any modifications it made to persistent objects are sent back to the servers, which decide if the commit is allowable, and install the new versions of the modified objects if the commit succeeds.

Now suppose that we add the ability to update Thor objects. For example, Thor might contain many CAD objects that should be enhanced by the additionof a new widget.

The update would cause all existing CAD objects to change to include the new widget, plus any new CAD objects would have the new widget.

Since there may be many objects affected by an update, and these objects might be stored at many different servers, we would not want to update the objects simultaneously, since that would be slow, and there might even be problems completing the update, since some servers might be down. Thus it makes sense to do updating incrementally in Thor and we plan to support this. This will allow us to explore the issues that arise from incremental updating. The approach we are exploring is to update objects when they are in client caches. Thus updating is highly incremental and lazy.

Updating in Thor exhibits many other properties of the final system (in addition to incremental updates): not all objects of a source class will be updated and therefore there must be a mechanism to identify those objects that should be updated; updates can involve transformations of existing objects as well as creation of new objects; by the time an update happens, several updates for objects of the source class may have been defined so that the system needs to figure out which one to install. Thus carrying out this project will help us to understand the technical issues that are raised by doing updates, but in a simpler system than what we are ultimately aiming for.