Software Deployment Using Mobile Agents

Nils P. Sudmann and Dag Johansen*

Department of Computer Science, University of Tromsø, Norway,
nilss@cs.uit.no, dag@cs.uit.no

Abstract. In this paper we show how mobile agents can be applied to software component updates in a distributed environment. One important aspect is that the control code for component updates can be distributed at run time. Also, by selecting the right agent system one gains flexibility in the choice of configuration languages.

1 Introduction

The principle of deploying web components over the Internet is currently touted by key industrial players as *the* structuring approach for software systems. Companies like, for instance, Microsoft (.NET), IBM (WebSpheres), Sun (JXTA), and BEA (WebLogic) are all advocating web service architectures. The fundamental idea is to download web service components over the network and install and extend a core software base already running on the computers.

Fundamentally, this is not a new idea. Web components, application updates, and similar extensibility techniques have been applied in, for instance, micro kernel operating systems [Lie95,BSP+95], in Java environments with applets and servlets, and in mobile agent technologies [GKCR98,AO98,BHRS98]. The difference, though, is the massive focus this type of deployment architectures currently gets.

In this paper, we present the latest in a series of middleware toolkits, which, in principle, is similar to these emerging web services platforms. This is TACOMA v2.2, a mobile agent system built to support software deployment over the network. In TACOMA, in contrast to all other early mobile agent systems, general software installation has always been a key requirement to support. In this paper, we will show how to actually ship install logic along with the software components to be installed.

The rest of this paper is organized as follows. In section 2, we introduce the key concepts of TACOMA, the mobile agent system we use as infrastructure for component deployment. In section 3, we discuss our software component deployment application. In section 4, we discuss various design options for component deployment using mobile agents, and finally, section 5 concludes this paper.

* This work was supported by NSF (Norges forskningsråd), DITS program, Norway grant no. 112578/431 and 126107/431.

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 97–107, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 Concepts in TACOMA

Software component updates are inherently complex because components often are interdependent. Hence, care must be taken during maintenance to avoid breaking the system while it is in a inconsistent state. It is difficult and inflexible to build a generic software deployment system. Rather, we need support code that controls the actual update or install process and is supplied dynamically as the software component is updated. Mobile agents are computations that are able to relocate themselves from one host to another, and they are a convenient way to dynamically supply control code for the update process in a distributed setting.

In the TACOMA project [JvRS95,JLvR+02], we started out almost 10 years ago to build extensible servers with mobile code being deployed over the network. This was initially used to extend remote weather servers in the Arctic with client software [JH94]. We generalized and built several versions of our deployment middleware, but with focus on mobile agent support. However, in contrast to all other early mobile agent systems, our focus was on supporting agents written in multiple languages. A typical mobile agent in TACOMA was a simple Perl script that needed to be deployed throughout the network, or a C++ server extension that had to be added to a single remote server.

Hence, a TACOMA mobile agent is basically a software component that has to be deployed in a network of extensible servers. In principle, this is exactly the same as what emerging industrial web service infrastructures provide, where our mobile agent resembles a web service.

In TACOMA, a mobile agent collects initial state and state changes in a *briefcase*. Once an agent is ready to move, its briefcase should contain a sufficient snapshot of the state needed to pick up execution at another host. Briefcases are further divided into a set of named *folders*, which themselves consist of a list of *elements*. The element is the basic data type in this structure; it is interpreted by the TACOMA system as an sequence of bytes. By convention, the actual interpretation beyond this is left to the application receiving it.

A *cabinet* is a persistent site-bound briefcase that a mobile agent may allocate and store data in for future visits to the host. Cabinets are used by our system as repositories to store software packages and information about them.

An agent *core* is a digitally signed immutable collection of initial state. It is used to authenticate an agent. Agents without a core are considered anonymous, and are severely restricted in what they are authorized to do in a general TACOMA configuration. In fact, in the configuration of TACOMA to general component deployment, agents are rejected if they are not signed by a trusted repository. The core is also used to validate the authenticity of the update control code and the software package. Only if the update control code is part of the core, is it allowed to execute with the needed permissions to perform software updates. This mechanism ensures that the code has not been modified while in transit. In effect, we turned a general mobile agent system into a safe installation toolkit. For more details, see [SJ00,SJ01].

An important aspect of Tacoma is the ability agents have to carry with them other agents. This is done by taking a briefcase representation of an agent, archiving it and incorporating it into the core of another agent. This agent may at a later point fetch the archive from its core and activate the initial agent.

3 The Software Deployment Application

Our infrastructure contains one or more repositories that contain the latest versions of software packages. These packages can be used to update or install new software at a number of hosts that are subscribing to the repository service. However, we do not require that all the subscribing hosts share the same software base after updates. That is, some hosts may be unavailable at update time, and may lag behind in updates. Furthermore, impatient host owners may update their software manually without waiting for a repository update. This implies that repositories cannot use their knowledge of previous package updates to determine future updates.

The repositories are Tacoma sites, containing software packages, and running a restricted version of our mobile agent system. A software package is a self contained archive with a version number. Each package is associated with a *probe agent* and a *update control agent*, which are package specific. In our system we currently use Redhat `rpm` packages, since most of our implementation has been centered around the Linux platform. These packages are stored in the Tacoma cabinet of the local repository. Furthermore, we store a list of hosts subscribing to the service.

Next, the software deployment application itself consists of two different agents, the state collector agent (`State.Coll`) and the installer agent (`Pack.Install`). The `State.Coll` agent travels in an itinerant style and collects information about hosts. The second type consists of several copies that work in parallel and does the actual software installation, see figure 1.

Like all Tacoma agents the snapshot of the `State.Coll` and `Pack.Install` agent consists of a core that is digitally signed by the repository. The core contains among others the command and control code and the list of nodes to visit. As mentioned, clients will only accept agents whose core is signed by parties they trust.

3.1 Step 1: The `State.Coll` Agent

The software update process starts at the repository. The repository regularly collects the list of packages available, and the list of hosts that are subscribing to its service. Next, the repository creates an agent core for the `State.Coll` agent containing the list of updates, the client host list, the `State.Coll` code, and a set of probe agents, one for each package, that determine the state of that particular software package at the client. The core is digitally signed by the repository with its key.

Now, the newly created `State.Coll` agent is activated. It examines its list of hosts that it has to investigate. It then tries to relocate itself to a host on

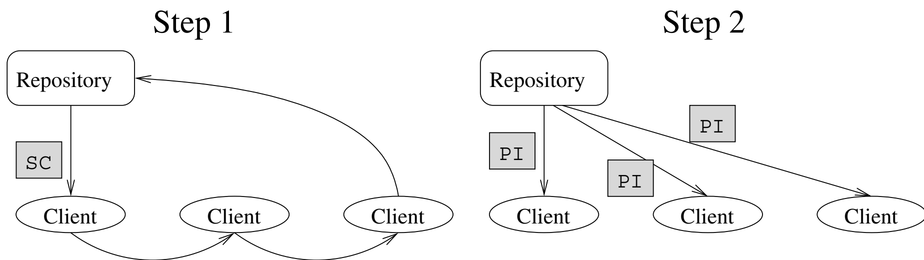

Fig. 1. TACOMA Software distribution Architecture. `SC` = `State.Coll`, `PI` = `Pack.Install`.

this list. Hosts unreachable at the moment will be re-tried later. Once it has been relocated to and authenticated by the receiving host, it is activated and the probe begins.

The `State.Coll` agent then activates each of the probe agents in sequence, supplying them with the package version available at the repository. The probe agent then determines the installed version of package it is designed for, and returns a simple binary answer to the question of whether an update to the `State.Coll` agent is needed or not.

Since all our updates currently come in the form of Red Hat rpm packages, all our probe agents are currently identical. In order to probe the clients installation, they simply execute the local rpm command to collect version numbers of installed packages. However, since the probe agent is dynamically supplied by the repository, it can equally well determine package versions using other methods. For instance, for packages not installed by rpm, it can locate a compiled binary by searching default installation paths for this package. Once a binary is found, it searches through it for a version string.

After the `State.Coll` agent has executed all of its probe agents, it creates a new entry in its briefcase that describes the the needed updates at the current host. From this list of needed updates, it subtracts packages that are locally cached, something which may happen because of unresolved dependencies, see section 3.3 below.

Finally, it picks a new unvisited host from its host list and repeats, until it has visited every host in its list. Hosts that are unreachable at any given time are rescheduled to the end of the host list. However, once three attempts have been made to reach a certain host, the `State.Coll` agent gives up, and deletes the host from its list. The reasoning behind this is that the host will probably be unavailable for some time, and will be caught by later runs of the `State.Coll` agent. When the host list is exhausted, the `State.Coll` agent returns to the repository.

3.2 Step 2: The `Pack.Install` Agent

The next step in our process is to disseminate the needed updates from the repository to clients. In this step, we need to allow the package supplier some control over the actual update process. This is achieved in the following way.

Once the `State.Coll` agent has returned to the repository, it compiles a list of hosts and packages, and generates one `Pack.Install` agent for each. These agents contain those packages that are needed at the host they are destined for. Furthermore, for each `Pack.Install` agent, the repository collects the software packages needed together with update control agents, one for each package. This is inserted into the `Pack.Install` core, and digitally signed by the repository.

Next the `State.Coll` agent launches the `Pack.Install` agents in parallel. Each `Pack.Install` agent travels to its host and begins installation. Once authenticated and activated, the `Pack.Install` agent activates the update control agents in sequence. These agents contain the actual control code that installs the software, and may be application specific or, as in our case, generic. Since we use the rpm package system, the update control agent simply executes the rpm tool locally and does some parsing of its output to detect unresolved dependencies.

However, if the package to be installed consists of, for instance, source code, the update control agent will be more complex, invoking compilers and linkers in the process of updating the installed software. The update control agent may even shut down running services, prior to updating them, and then restart them when the update process is completed.

Once all of the update control agents have finished executing, `Pack.Install` collects the results. However, to avoid flooding the repository, the `Pack.Install` agent does not report back success to the repository once it has finished. It does, however, respond to unresolved dependencies as discussed below.

3.3 Dependencies

Sometimes the `Pack.Install` agent may encounter unresolved dependencies during installation[1]. Such dependencies are duly noted and the installation aborted. However, to avoid transporting the payload twice over the network, the `Pack.Install` agent caches its payload in the local TACOMA cabinet. It then converts itself to a `State.Coll` agent and returns to the repository with a new list of packages that need to be collected.

If packages are needed but are unavailable in the repository, a human administrator is notified. Once the administrator has fetched the missing packages, he may manually start the probing process again, or simply wait until the repository does this at some later point.

[1] A new version of a package may require a new previously uninstalled package, or an update of a package on which it depends.

3.4 Privileges

The probe and update control agents all currently run with administrator privileges, since most software updates in our system require this. Our security framework is based on authentication using digital signatures, which in the end means that client administrators must trust the repositories they subscribe to.

The advantage to this approach is that it can be applied to existing systems, without the need for new security enforcement schemes. The disadvantage is that it has a coarse granularity, basically all or nothing is allowed, and does not follow the principle of least privilege advocated by security experts.

It is possible to extend our system to support a finer security granularity based on different user accounts. For instance, it is possible to map package suppliers to user accounts. This would require our system to authenticate the individual probe and update control agents that `State.Coll` and `Pack.Install` activates at the client. This modification is straightforward, since the probe and update control agents are full-fledged agents; they can be required to have a digitally signed core. Based upon this signature, which is the signature of the package supplier rather than the repository, one can map package suppliers to user accounts.

3.5 A Note on Performance

The `State.Coll` agent needs to do a full probe at clients of every package available at its repository. The reason for this is simple. There is no way to make sure that even the local TACOMA site has a correct view of the state of locally installed packages. Even if we cached earlier probes, a local administrator may have bypassed our system and made an update or downgrade manually.

Each package needs its own probe agent. Creating and activating an agent locally imposes an overhead of about 300 ms, not counting the computation of the probe agent itself. Once the number of packages becomes large, the overhead of creating and activating probe agents will reach unacceptable levels.

For this reason, we give local administrators the ability to turn caching of local probes on. This means that the probe agents will miss changes to the local packages done manually, but it greatly reduces the overhead of the `State.Coll` agent.

When caching is turned on, the results of local probes are stored in the local file cabinet. The cache lists the package name, the last version number probed, and whether an update is needed or not. The `State.Coll` then looks in the cache before activating a probe agent to determine if the probe is necessary. The probe is only launched when the package numbers differ or if the cache indicates that an update is necessary.

4 Design Options for Software Deployment

This section discusses some of the main design alternatives we were faced with when applying the mobile agent paradigm to software deployment problems. As

such, it also constitutes our argument that mobile agents are well suited for software deployment.

4.1 Software Updates as Mobile Agents

One way to do software deployment using mobile agents is to select an appropriate mobile agent system, and deploy software components as mobile agents. The software components can then relocate themselves to another host as a mobile agent, and replace an already running agent there.

The problem with this approach is that software components have to be modeled and implemented as mobile agents. Most software components are not mobile agents, and redesigning a software package as a set of mobile agents still requires substantial work. Furthermore, most mobile agents systems execute their agents in a virtual machine, an execution layer providing safe code execution and a homogeneous environment in a heterogeneous network. This often conflicts with the needs of software upgrades that need to access the underlying operating system.

What we need is a mechanism that allows mobile agents to wrap generic software components, without modification of the software components themselves. In order to support this, the software components have to be separated from the update control and network code of the mobile agent. In this approach, the software component is added as a passive payload to the agent, while the update control and network code becomes the mobile agent.

The problem with this approach is that it requires some support from the agent system. The update control code needs to be granted privileges beyond those of a regular mobile agent in order to affect the system outside the safe sandbox of the agent environment. Either these privileges are granted indirectly through a service of the local agent system, or directly by running the update control code with all privileges needed to perform an update. In either case, some trust of the authenticity of the payload (the software component) and the update control code has to be achieved.

4.2 Single Itinerary Agent or Parallel Agents

The natural mobile agent approach to software installation is a single itinerant mobile agent. This agent is loaded with the software to be installed. It then travels to all the hosts which need to be updated in order, and installs its payload at them. The advantages are linked to the benefits of mobile agents. The update process puts little strain on network resources, since the agent updates one host at a time. The number of updates sent over the network is n, where n is the number of subscribing clients, with one additional status message sent back to the repository. Furthermore, since the mobile agent is autonomous, it operates asynchronously with no need for further control.

This approach has, however, its drawbacks. First, if hosts do not share exactly the same software base, parts of the payload might not be needed at a host. Furthermore, it is prone to failure, since the failure of a single host can potentially

terminate the entire upgrade process. This is a common problem for mobile agents, and solutions have been developed [JMS+99,MvRSS96]. However, these solutions add to the complexity of the system, and are not always suited for our setting.

A second alternative is to perform the update in parallel, using n agents, one for each host that is to be updated. Now, each agent can configure its payload to the needs of the host it is going to update. This may potentially lower payload size, and thus network usage. The number of updates sent over the network is still equal to the number of hosts, n, while the number of status messages rises from 1 to n in this approach. The failure of one site no longer affects the agents updating other sites, so costly fault-tolerance mechanisms for mobile code need not be employed. However, since updates are performed in parallel, care must be taken not to stress the network when the agents are initially shipped, and status messages are sent back to the repository.

4.3 Global and Local Repositories

The second approach above scales poorly for truly large systems. In a setting with not only one local network, but thousands of local networks each containing several hundred hosts, the parallel update approach would need several hundred thousands of update agents that are shipped from a single repository. A supplier of software components may wish to spread this load between several different repositories.

Furthermore, local administrators would probably like additional control over updates emerging from a single global repository. For instance, if the global repository does aggressive updates (using its user base for cheap beta testing), local repositories may elect to delay updates until confidence in stability is achieved.

We used the following approach to solve these problems. Some hosts subscribing to a repository service may in fact act as repository services themselves. Any payload of an update agent from its parent repository is inserted into the local temporary repository. A system administrator may at his/her leisure move updates from the temporary repository to the local repository, making the update available to its local subscribers.

End users are able to subscribe to any repository, unless restricted by local policies. So end users may still live dangerously on the edge, by subscribing to an aggressive repository instead of a possibly more careful local repository.

4.4 Client State

Another problem is getting information about the clients of the repository. This includes what software components we need to update them. Having the clients themselves poll a server for updates is a common method today. The trouble with this approach is that the client has no idea about what updates are available and when. A client would need to get a list of all components available at the repository at frequent intervals, and compare it with the locally installed

components. This wastes network bandwidth and puts additional strain on the repository.

Another approach is to have the repository probe all the clients subscribing to its service once a new component becomes available. Here the problem is that clients may be unavailable since they are disconnected or the network might be partitioned for a while. Clients that missed an update due to partition may not be aware of this fact. These clients will eventually become updated during the next regular repository probe.

We used a hybrid approach that combined the pull and push methods. Clients that are simply disconnected or turned off may issue a probe once the connection is re-established.

4.5 Mandatory and Optional Software

Basically, from a system administrators view, there are two types of software, optional software and mandatory software. Mandatory software consist of software the system operator deems necessary for the correct operation of the local network site. This includes things like security updates, virus checkers, and monitoring software. Other software packages are optional, and tailored toward the needs of the end user, which are not critical to the correct behavior of the target host.

To accommodate for these two classes of software, we have repository entries that are tagged as mandatory by the repository administrator. These are installed without interfering with the subscribing end user. Information about optional software is mailed to the end user, who may elect to include the software upon the next update. Once optional software is installed, further updates will be installed without user interference.

4.6 The Insecurity of Mobile Code

Another problem with the use of mobile agents is the safety requirements a generic mobile agent system must fulfill in order to execute foreign code. This is usually achieved by running the code on safe virtual machines that enforce safety [GM95,WLAG93]. These techniques isolate the system from the mobile agent, and the agent is then restricted to access the system through well defined services.

This voids our argument that mobile agents are useful in software deployment. We argued that the software that is to be deployed needs some control of the update process at a host. This control may be very system specific, and require access to the system beyond that provided by virtual environment offered.

Instead, we found that executing code based upon authentication and sufficient trust is much more suited in this setting. Thus, we have a closed set of virtual machines that are only available to agents whom we trust. Once access to these virtual machines is granted to a mobile agent, it may execute without further limitations. In our context, this means that our system is a closed agent

system. Only mobile agents authenticated as repository agents gain access to the system while regular agents are denied access.

5 Conclusion

In this paper, we have shown how mobile agents can be used to do software component deployment. However, we have just demonstrated the potential in this technique. Currently, there is little reason for the `Pack.Install` agent to be a full-fledged mobile agent. Since the system currently uses only one type of software package (the Redhat rpm format), the agent can be replaced with a simple message from the repository to a service agent at the client. However, by using mobile agents, our model support arbitrary software packages, that may even be vendor specific, since the control code of the update is dynamically downloaded to the client.

The same argument goes for the `State.Coll` agent. The `State.Coll` agent benefits from several of the befits inherent in the mobile agent paradigm, such as asynchronous operation, preservation of bandwidth, and usefulness in disconnected environments. However, critics may say that the advantages gained by using mobile agents to collect state information about client hosts may not match the cost of having to deploy a mobile agent system in the first place. But, since the code that collects the state is supplied by the repository and dynamically uploaded to the client host, using mobile agents provides more flexibility. Thus, our approach demonstrates a fundamental principle in which mobile agents may be used to support application controlled updates.

The base agents are in place, and we are currently working on a tool set that allows repository administrators greater control over the deployment process. With this tool set, administrators can for instance schedule updates to happen at a specific time. Currently, administrators have to manually update repositories, even if the repository itself is a client to a parent repository. The administrator tool set will allow the administrator to automate this process, allowing for a tree of repositories where updates trickle down from a common root.

References

[AO98] Y. Aridor and M. Oshima. Infrastructure for Mobile Agents: Requirements and Design. In *Proceedings of 2nd International Workshop on Mobile Agents (MA '98)*. Springer Verlag, September 1998.

[BHRS98] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole – Concepts of a Mobile Agent System. *World Wide Web*, 1(3):123–137, January 1998.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the *spin* Operating System. In *15th ACM Symposium on Operating System Principles*, December 1995.

[GKCR98] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.

[GM95] J. Gosling and H. McGilton. The java language environment: A white paper. Technical report, Sun Microsystems, Inc, May 1995.

[JH94] D. Johansen and G. Hartvigsen. Architectural issues in the StormCast system. In *Prococeeding of the Dagstuhl Seminar on Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 1–16, Dagstuhl, Germany, 1994. Springer Verlag.

[JLvR+02] D. Johansen, K. J. Lauvset, R. van Renesse, F. B. Schneider, N. P. Sudmann, and K. Jacobsen. A TACOMA Retrospective. *Software Practice & Experience, Wiley*, 2002. To be published.

[JMS+99] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 180–189, Austin, TX, June 1999. IEEE Computer Society.

[JvRS95] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HOTOS-V)*, pages 42–45, Orcas Island, WA, May 1995. IEEE Computer Society.

[Lie95] J. Liedtke. On μ-Kernel. In *15th ACM Symposium on Operating System Principles*, December 1995.

[MvRSS96] Y. Minsky, R. van Renesse, F. B. Schneider, and S. Stoller. Cryptographic Support for Fault-Tolerant Distributed Computing. Unpublished technical report., February 1996.

[SJ00] N. P. Sudmann and D. Johansen. Adding Mobility to Non-mobile Web Robots. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'00) Workshops*, pages F73–F79, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, April 2000. IEEE Computer Society.

[SJ01] N. P. Sudmann and D. Johansen. Supporting Mobile Agent Applications Using Wrappers. In *Proceedings of the 12th International Workshop on Database and Expert Systems Applications (DEXA'01)*, pages 689–695, Munich, Germany, September 2001. IEEE Computer Society.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993.