

A Smart Device Update Web Service

5/24/02

Terry H. Ess
T. H. E. Solution LLC

INTRODUCTION

The topic of web services and their possible uses has raised quite a buzz (and lot of misunderstanding) in the last year. So first we need to address two immediate questions:

1. What is a web service?
2. Are they of use to smart devices such as automated factory machines, medical diagnostic equipment or hundreds of other embedded devices (many of which are just PCs in disguise these days)?

A web service is nothing more than an exchange of information using a standardized format and transport protocol. In all cases the standardized format for the exchange is XML. The transport protocol could be any of the higher-level Internet protocols but has generally used the “web” protocol, HTTP. We can go further and set additional standards for the information interchange, e.g. SOAP, but we do not have to.

The usefulness of web services to smart devices probably depends on your perspective. Most, if not all, the uses for web services could already be accomplished using proprietary messaging over a TCP/IP socket. What a web service brings to the table is a group of third party components that reduce our development time and allow us to create scalable and maintainable servers. At its very simplest we can easily embed a “read channel” web service into almost any device with IP access that provides a TCP/IP socket interface (see <http://www.the-solution-llc.com/webservice.zip>). In this manner we can provide our customers with the data they want without having to spend the time to write the code that presents it.

We can take the web service concept a lot further though. This paper will look at just such an application, the automation of updating a device’s software components.

UPDATE SERVICE

The update service provides the means for a smart device to automatically or on user command, update its software components. This is one of three services (the others being remote control/diagnostics and synchronous/asynchronous trouble notification) that need to be in place to make a device capable of lights out operation. For the device manufacture it presents an even more interesting prospect, mainly the ability to generate a revenue stream with the service. The service has been implemented as part of the open source PC on the Factory Floor framework (see <http://www.the-solution-llc.com/proposal.pdf> and <http://www.the-solution-llc.com/release0-2.zip>). More detail on the design and implementation is provided in the following sections.

Overview

The service has a number of use cases to support. This includes not only the update sequence itself but also the administration of the service etc. The only one we will explore is the core device update case. This can be summarized as:

1. Machine connects with update web service
2. Sends identity, authentication and current configuration
3. Gets update list
4. Gets update files
5. Installs updated files

The flow of the operation is proved in Figure 1:

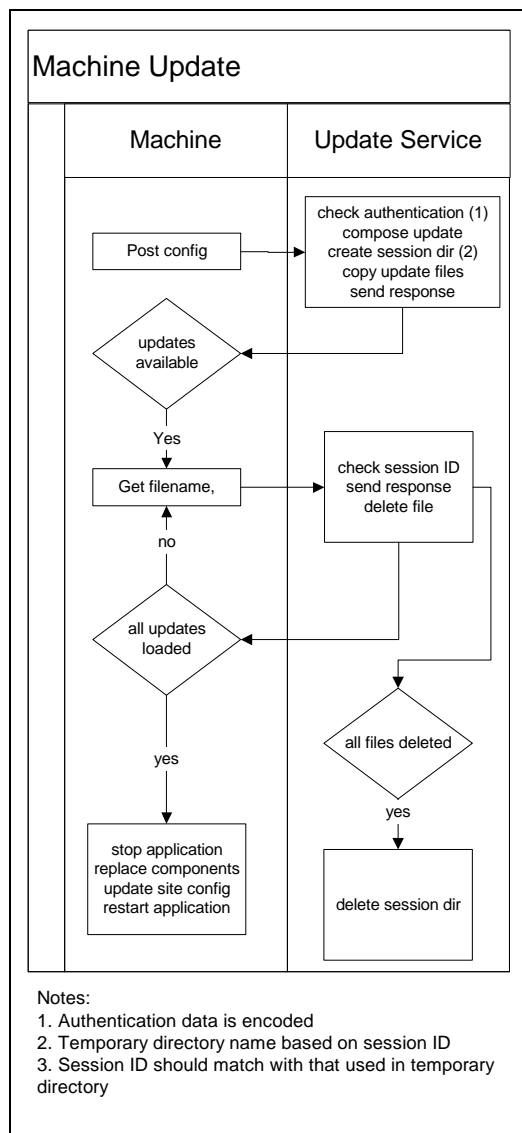


Figure 1, Machine Update Sequence

The ability to control access, determine what components require updating or replacement etc. is made possible with a server side database. A minimal prototype in entity – relationship form is provided in Figure 2:

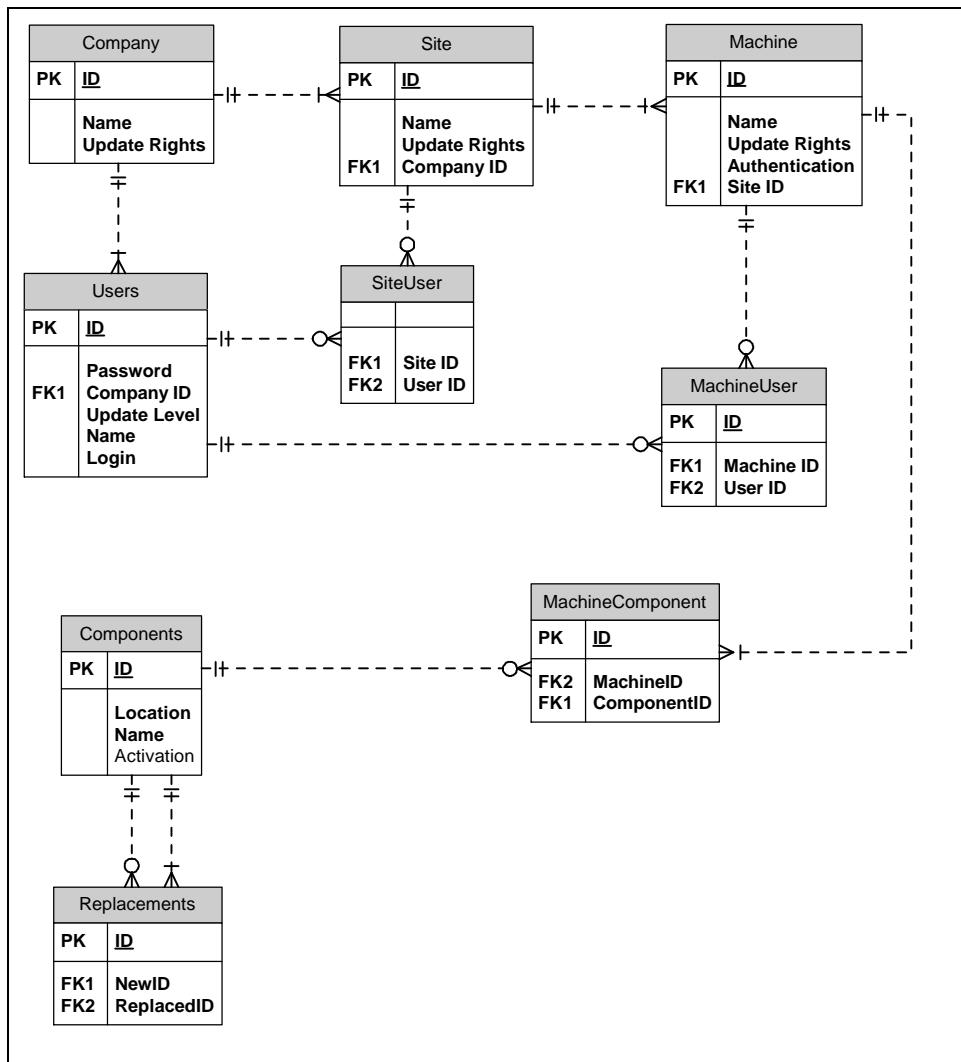


Figure 2, Update Database

The key points are:

1. Combining its company, site and machine names uniquely identifies a machine.
2. A machine can authenticate itself using its authentication string, extending it in a service specific manner and properly encoded.
3. A machine can have update rights (i.e. the machine’s owner has purchased a service subscription) or not.
4. A software component can be uniquely identified by its name and date – time stamp.
5. The replacement of a component is possible by checking for replacements or a newer version (the component found in “Location” has a more recent date – time stamp then the component on the machine).

The key information exchange is made possible with two XML documents. The first is an inventory of what exists on a machine. A typical example follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<config>
  <company>company 1</company>
  <site>site 1</site>
  <machine>machine 1</machine>
  <date_time>2002-05-07T12:24:05</date_time>
  <authentication>60263</authentication>
  <no_components>10</no_components>
  <components>
    <filename>listener.htm</filename>
    <version>2001-11-15T16:15:58</version>
    <filename>BuildLog.htm</filename>
    <version>2002-05-07T16:23:02</version>
    <filename>test.txt</filename>
    <version>2002-02-06T18:55:28</version>
    <filename>english.txt</filename>
    <version>2002-02-06T18:24:20</version>
    <filename>hmi.dll</filename>
    <version>2002-02-11T21:11:52</version>
    <filename>webserver.exe</filename>
    <version>2002-02-11T16:34:46</version>
    <filename>remotestub.exe</filename>
    <version>2002-02-11T16:38:02</version>
    <filename>machine.exe</filename>
    <version>2002-02-13T19:46:46</version>
    <filename>hmi.exe</filename>
    <version>2002-03-11T15:37:36</version>
    <filename>UpdateClient.exe</filename>
    <version>2002-05-07T16:23:00</version>
  </components>
</config>
```

The server, if satisfied with the client's authenticity, provides a response that lists any updates. A typical example follows:

```
<?xml version="1.0" encoding="utf-8"?>
<update >
  <path>appl/logs/CED6E346C32107841026BDBF344651A8</path>
  <components>
    <filename>listener.htm</filename>
    <replaces>listener.htm</replaces>
    <filename>webserver.exe</filename>
    <replaces>webserver.exe</replaces>
    <filename>hmi.exe</filename>
    <replaces>hmi.exe</replaces>
  </components>
  <no_components>3</no_components>
</update>
```

When the client machine gets an update list, it uses the path and filename(s) to know what to get and where to get it. The HTTP protocol provides us a transport to move the XML documents and to get any updated files. The exchange of XML documents is done with a POST. This looks like:

```
POST /appl//config.xml HTTP/1.1
Host: updateservice
Content-type: text/xml
Connection: keep-alive
Content-length: 1032

Config.xml file
```

The server's response to the POST provides a list of update if any or an error message. Errors occur if the authentication in the request fails or the identified machine does not have update rights. A successful response looks like:

```
HTTP/1.1 200 OK
Content-type: text/xml
Connection: keep-alive
Content-length: 530

Update.xml file
```

[NOTE: This exchange is SOAP like in general construct but the SOAP standard was not used since it introduced unnecessary complexities. It can be best understood as a document exchange sequence rather than a remote procedure call.]

When an update list file includes updates, the client device is responsible for obtaining each of the updates. This is nothing more than a HTTP GET transaction. This looks like:

```
GET path/filename HTTP/1.1
Host: updateservice
Cookie: JSESSIONID=xxxxxxxxxxxxxx
Connection: keep-alive

-----
HTTP/1.1 200 OK
Content-type: xxxx
Content-length: yyy

Filename file
```

Client Side Implementation

The initial client side implementation is a stand alone Windows console application using “generic” C++, C run time library support, generic TCP/IP socket routines (except where Windows specific is absolutely necessary) and the open source Xerces C++ XML parser. Because of this it can be easily moved to other environments of interest including Linux, VxWorks, QNX etc.

It connects to the remainder of the application system using the standard mailbox/post office that is the hub of the PC on the Factory Floor framework. This provides the means to stop/start parts of the system as necessary to perform the actual updating of executables. The top-level sequence of the client is captured in Figure 1. There are a couple of key implementation points that deserve comment:

1. The client treats the entire update sequence as a transaction. Either all the updates are made or none are. Therefore the update files are temporarily stored until all the updates are obtained etc.
2. Both the XML files are verified using XML Schema held locally.
3. The Windows implementation required the development of less than 700 new lines of C++.

Server Side Implementation

The server side can be implemented in a number of different ways. The desire is to provide a server that is capable of meeting long term commercial requirements including the ability to scale and maintain. So “proprietary” development schemes (such as the custom client) were dropped. There are at least two possible platforms for meeting this requirement, J2EE and .NET. Of the two J2EE is the more mature, the more widely used and can be constructed using very low cost components. The initial server side implementation is a J2EE web application. The primary components used are:

1. Java 1.4.0
2. Tomcat 4.0.1
3. MySql 1.7.5
4. DBConnectionBroker 1.0.13
5. MM – MySql JDBC driver 1.7.5

Each of the components except item 1 is open source and free. The Java SDK for many environments (including Windows and Linux) is available from Sun for free. So the total cost for putting a server together is \$0. The overall design is summarized in Figure 3.

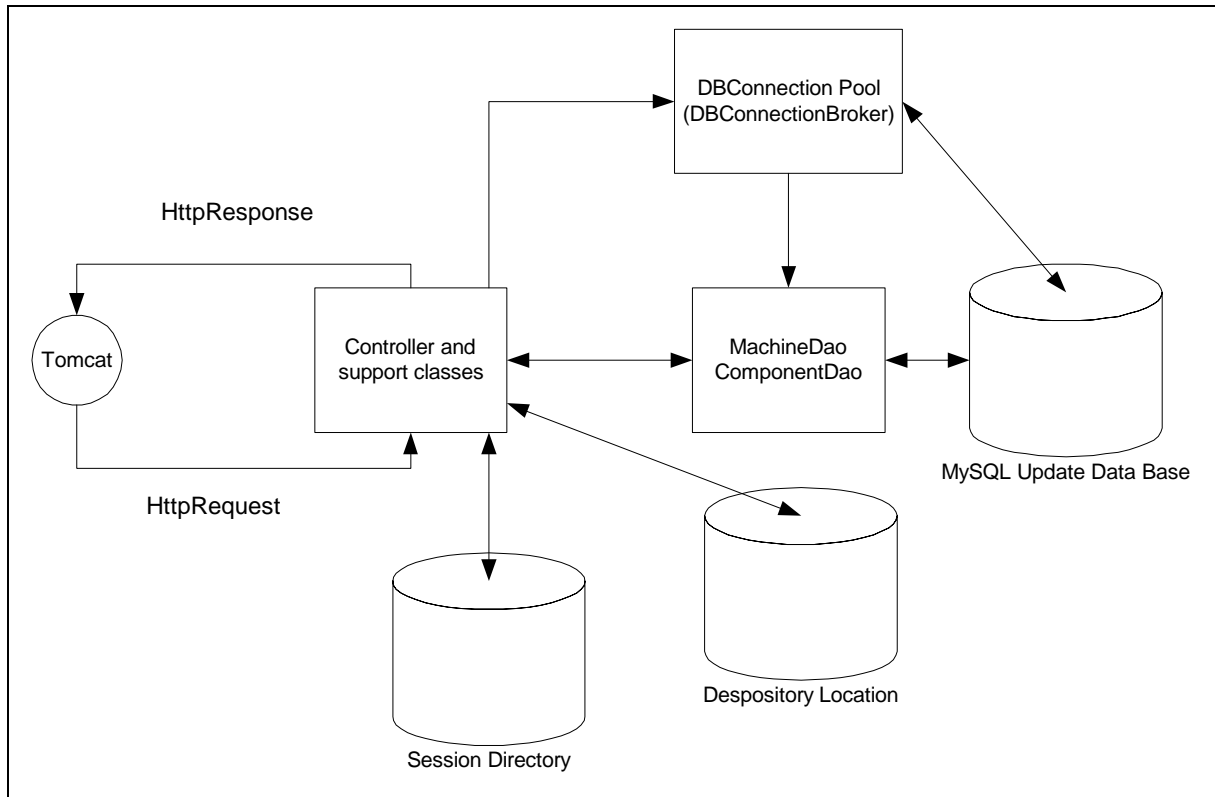


Figure 3, Server Side Design

There are some of key implementation points that deserve comment:

1. Both the XML files are verified using XML Schema held locally.
2. The depository location(s) is used to maintain the “distribution set” of currently released software.
3. The web application uses a temporary session specific directory. It copies the update file set from the depository location(s) to this directory. After each file is obtained the file is deleted from the directory. When the sequence is completed (i.e. there are no files left in the directory) the directory is deleted.
4. The web application will automatically remove a session’s directory and its contents if the time between client requests exceeds a dynamically set threshold.
5. The update sequence is logged to facilitate detection of systematic problems and illegal entry attempts, provide customer billing etc.
6. A small amount of functionality was implemented using JNI.
7. The complete implementation required less then 500 lines of Java.

CONCLUSIONS

The concept of a web service, especially if we look at it as a document exchange, provides a powerful tool for addressing many of the interactions that smart devices have with the external world. Using readily available support for implementing web services, we can deploy our machine update service with a very small development effort (creating less then 1200 lines of code) that requires no software capital investment. In short, we can do it fast and cheap.