ARCHITECTURE AND IMPLEMENTATION OF A REMOTE MANAGEMENT FRAMEWORK FOR DYNAMICALLY RECONFIGURABLE DEVICES

Rajiv Chakravorty

Cambridge Computer Laboratory,
Gates Building, University of Cambridge,
Cambridge CB3 0FD, England
Email: rajiv.chakravorty@cl.cam.ac.uk

Hans Ottevanger

Philips ASA Labs,
Building SFJ, Philips Electronics N. V.,
5600 JB Eindhoven, The Netherlands
Email: hans.ottevanger@philips.com

ABSTRACT

This paper presents Smart Box Management (SBM) – an end-to-end remote management framework for Internet enabled devices. In SBM, client devices securely communicate over the public Internet for device management specific services such as remote registration, remote configuration, dynamic updates (downloads) and device diagnostic uploads with the SBM server. The SBM client device is a *smart* box - a device that can easily adapt to dynamic software updates constituting entirely new applications, bug fixes or patches to latest updates of the existing code base. SBM uses HTTP to leverage a web-based device management infrastructure that offers several benefits: ubiquity, security, reliability and a high degree of user friendliness. In this paper we delineate the SBM framework, detail the SBM protocol and describe our experiences in the implementation of a successful SBM prototype.

1. INTRODUCTION

Growth in ubiquitous and mobile computing systems coupled with the *anytime-anywhere* phenomenon has led to an early introduction of wide variety of Internet capable devices. These devices seek Internet access for information download (e.g. WAP/GPRS phones), content download (e.g. Internet enabled *mp3* players) and/or content streaming (e.g. Internet Radio). It is expected that the volume of such devices will soon outnumber conventional PCs – the traditional way to access the Internet. However, with new devices being introduced and a range of such devices to choose from, customers are often confused as to *when* and *what* to look for that can fit to their expectations. What is seemingly obvious is that customers are now looking for - apart from various other features and functionality - how long a device can serve useful purpose. Apparently, they are looking for devices that can *adapt* to any future customisation. SBM augments exactly this idea of "future ready" devices with the concept of dynamically reconfigurable devices. It builds up a framework that helps manage dynamically updateable Internet devices.

The idea to extend software reconfigurability within embedded devices is not new; in fact, several leading CE device (e.g. DVDs, CDs) manufacturers allow internal firmware upgrades by running system CDs made specifically for this purpose. These devices make use of *hard* updates to enable download that eventually require devices to be rebooted, before the downloaded piece of code takes effect. SBM extends this idea of software updates to decouple human intervention or any reboot requirements to make possible dynamic software re-configuration, which we term here as a *soft* update.

A dynamically reconfigurable device is advantageous in many ways: (i) *Remote Bug Fixes:-* An immensely profitable proposition – while customer help desks are free to attend problems of individual customers, device vendors can avoid huge losses due to massive product recalls, an occurrence that is more than common in high volume electronic businesses, (ii) *New Features/Applications:-* Software download enables addition of new features within a device. Applications can be updates or upgrades that add some new feature or functionality within the device, and (iii) *Customization:-* Customization allow users to attune devices according to their personal preferences. New ways of customising a device are possible; for instance, downloading choice of display browsers for Interactive TVs or mobile phones, plugins for digital audio/video players etc..

SBM addresses these issues with a complete end-to-end device management solution. It is a technology to service devices through an external network connection. SBM offers services like remote activation, remote reconfiguration, dynamic software updates and diagnostics uploads to a SBM client device. We discuss several issues related to SBM further.

2. SBM END-TO-END ARCHITECTURE

The SBM architecture encompasses a number of home domains having range of Internet enabled devices. ISPs provide Internet access through conventional dial-ups or broadband connections that make use of cable/xDSL modems. An edge device in the form of a residential gateway or a set-top box provides Internet access to a home domain that includes a network address translator (NAT) and a firewall facility. We envisage use of NATs as they have outgrown in popularity - mainly due to depleting IPv4 addresses and introduction of IPv6 still seemingly distant. These NAT routers not only provide isolation for an end user home domain, but also the flexibility to switch service providers. We conjecture that all existing features of a NAT router and set-top-box will be eventually combined in some form of a sophisticated edge (gateway) device.

All SBM clients and server hold a general notion of services maintained by SBM. These services have to be quite independent, allowing different parties to run different sets of services. SBM can offer number of services to its client devices; in this article, however, the focus will be on providing the basic set of services like

Rajiv is currently a Phd student at Cambridge University. This work was done when he was working with Philips ASA Labs., Eindhoven, The Netherlands.

remote activation, remote configuration, dynamic updates (downloads) and device diagnostic uploads.

2.1. Remote Registration (Activation) of the Device

Activation is a process by which a user (through his device) can establish an account with its service provider. As a virgin box the device for the first time connects to a server and using its unique device identifier logs all user related data (user name, language, country, area-code etc.), which is then passed on to an account server. Internally, the account server creates a customer record and returns an allocated account code as a response to the applicable connection data. Finally, it also returns service specific information that is stored as persistent data in some form of non-volatile memory within the device.

2.2. Remote Configuration Service

Remote configuration service is typically used to manipulate settings within the SBM client boxes. This type of service can be useful for: (1) the consumer (e.g. during relocation), (2) the configuration manager (e.g. configuration data changed), (3) a service engineer (e.g. some regularly occurring problem, attributed to an error in configuration data) and (4) a software engineer (e.g. a bug is fixed that necessitates new values for service parameters).

2.3. Remote Software Updates (Download)

Software download service is central to device reconfiguration and to remote device management. Possibilities include: (1) *Updates* that improves the quality and/or reliability of the device, (b) *Upgrades* or extensions that transform an existing CE device into a new device with novel functionalities, (c) *Dynamic data*, usually the content that is downloaded by devices from the service providers.

2.4. Upload Service (Diagnostics)

Remote diagnostic service deals with any problems within SBM client devices and also of the network to which they are connected. The approach to deal with remote diagnostics can be two pronged - either reactive or proactive. In the reactive approach, for example a situation wherein a customer calls the helpdesk with a particular problem and the diagnostics is carried out ad-hoc. A proactive treatment can involve a service engineer, who collates diagnostic reports uploaded by all the client devices on regular basis and analyses the data collected for any problems within a device.

2.5. SBM Protocol

SBM uses HTTP [1] as a protocol of choice for enabling device management specific services. We chose HTTP as it provides the desired reliability and security for data upload and download functionality in implementing SBM specific services. HTTP offers easy service extensibility and can also be tailored for use in thin HTTP compliant devices. Session layer security using SSL or TLS with HTTP is provably the easiest option for security. Another benefit using HTTP is the possibility of a web-based management infrastructure that eases large scale management of vast variety of Internet devices. Such features cannot be easily offered by other device management protocols or paradigms. We further elaborate on this in section 5.

SBM uses HTTP's POST method and appends its own *parameters* to distinguish it from those of the HTTP header. Since no standard parameter in HTTP/1.1 is mandatory in a generic way, services can pick amongst standard parameters and enforce their presence to create a specific protocol action between the client and server. Parameter presence in SBM is not mandatory, which means services cannot fail in its absence. Instead, it should generate an error message and report it back to the client device. The order and multiplicity of any SBM parameter also does not preclude the action of the protocol.

2.6. Device Registration (Activation) Protocol

The activation service assumes that a device is Internet enabled, has an IP address and is ready to send data into the network. The device connects to the SBM server using the *activationStart* service URL, which is preset within the device. If the device implements session layer security mechanisms like SSL or TLS, it goes through a series of protocol handshakes for server-client authentication and establishes an encrypted communication channel with the server. The device then starts by sending an *ActivationStart* message to the server (figure 1(a)). This request includes unique device ID (Product ID or Client ID) of the device. No further steps are invoked if the server uses a response value (SBM *ResponseValue* parameter) of "stop" in the response message. Server can also postpone further steps using a response value of "tryLater:delay" (for e.g. *ResponseValue=tryLater:60*) to postpone the client request by in the response message by *delay*(60) seconds. Usually, servers typically respond with "continue", client prepares and posts the next *ActivationSelect* message to the server that includes all user level details (e.g. UserName, Area Code, Phone Number etc.). This message might optionally include certain parameters along with their values requested by the SBM server in its response to the first request. When the device registers for the first time, server allocates an account code to the new device. After getting a valid response, the client prepares the final request *ActivationProcess* and posts it to the server. If the server is satisfied, it responds with all service and subscription related information to the client, which is then stored in the persistent memory of the device.

2.7. Configuration Protocol

Using configuration service, existing parameters of a service can be modified or new service specific parameters added. Parameters that can be manipulated are assumed to be present in some sort of persistent memory within the device. The protocol starts with a *queryService* message; every time the box establishes a physical connection with the network and at regular intervals (e.g. every 24 hrs). If the server responds with the action "settings" (actions=settings), it acts as an indication to the client that it can now proceed with the next steps, i.e. *confirmConfig* and *writeConfig*. The *confirmConfig* step allows the server to either request read some parameter value within the client, or to directly specify the parameter that it wants to modify (along with its value). In *writeConfig* step, the client updates the server with the existing value of requested parameters. If the server after reading the parameter value decides to change it, it can do so in its response to *writeConfig*. Figure 1(b) shows a three step process. A two-step process allows only a *write* cycle with the device, while the use of the third step permits a *read-modify-write* cycle with the client device.

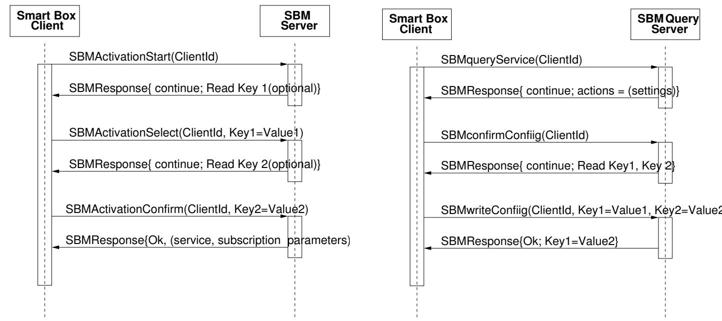

Fig. 1. (a) Activation Protocol and (b) Configuration Protocol

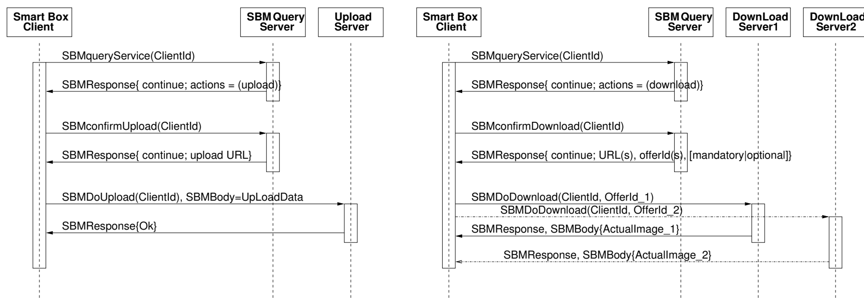

Fig. 2. (a) Upload Protocol and (b) Download Protocol

2.8. Upload Protocol

The upload protocol offers a diagnostic service that enables upload of device diagnostic information to the service provider. Its starts as usual with a *queryService* message request posted to the SBM server. If the server is interested to read diagnostic reports from the device, it can respond with an action "upload" (see figure 2(a)) The client issues the next request *confirmUpload* as an indication to the server that it has diagnostic information to upload. The server responds to the client along with the resource identifier of the upload server to which it can upload the information. The client then posts a *doUpload* message with device diagnostic data appended. The server responds with success once all the data is successfully uploaded.

2.9. Download Protocol

Download feature can enable data transfer from servers to the clients. Typical samples for download include software modules (updates or upgrades), new applications (e.g. micro-browser) or content (e.g. mp3 audio or jpeg images). As shown in figure 2(b), when the action to the response to *queryService* message from the SBM server is "download", the client prepares and sends a *confirmDownload* message request to the server. The server replies with a list of offers (along with their offer ID's) for download by the client device. The response also consists of the resource identifier's of the download servers corresponding to each offer available to the client. While some offers will be mandatory (e.g. bug fixes, module updates etc.), others will be optional and might involve some payment. The user is informed only when offers are optional and necessitates user consent. The subsequent request *doDownload* message is sent to the respective download server for each offer accepted along with their offer ID's. A response to each *doDownoad* request message from the download servers will have the packaged downloaded data appended as the body of the response message.

3. PROTOTYPING SBM

The SBM client hardware for the system used an Algorithmics P4032 board with R5 MIPS (RM5231 from QED) running VxWorks OS at 133Mhz. SBM Server used Stronghold 2.4.2 running on solaris 2.7, which is a commercially supported version of Apache 1.3.11. Servlet support was enabled using JServ 1.1, JDK 1.2.104 and JSDK 2.2, coupled with mySQL as SQL server that had a JDBC type 4 interface and compatible with JDBC 3.0. Client software modules were coded in C++, excluding IPD and the dynamic loader that were done in 'C'. Native TCP/IP stack available from VxWorks was used in the client. The client leased a local static IP address and used a native DNS resolver library from VxWorks.

3.1. SBM Server Side Components

A SBM server has two kinds of users - SBM clients and actors. Both clients as well as actors can connect to the SBM server simultaneously. Figure 3(a) shows the SBM server side components. SBM server side consists of a set of servlets, a SQL server and one or more backend servers. It has two different and independent interfaces: an actor-server interface that handles actor and server

communication and a client-server interface to handle messages exchanged between the SBM clients and the server.

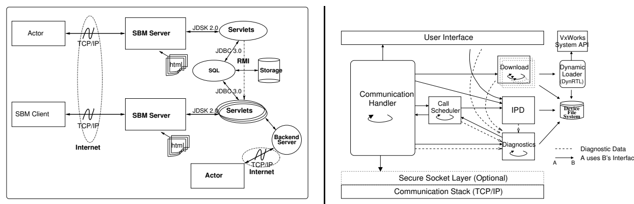

Fig. 3. (a) SBM Server Components and (b) SBM Client Modules

3.2. SBM Client Design

Modularity was most important goal while designing the SBM client. Figure 3(b) show the client side modules. We briefly describe the modules used in a client SBM:

Identifiable Persistent Data(IPD):- The IPD manages, stores and provides access to the data (service specific parameters) along with their values in some kind of persistent memory of the device, e.g. a flash or NVRAM. After a virgin device goes through its first activation, it retrieves service related information and stores it within the IPD. Using this state, an end-user can resurrect a device that had panicked due to an update failure.

Call Scheduling Block:- The call scheduling block processes service calls. It accepts service calls and schedules them on first come first serve or some pre-defined priority basis. It maintains a timer to query the server (using *queryService* message) from time-to-time and also to reschedule any service calls that was postponed by the server.

Communication Handler:- It enables communication between client and server, by execution of service-calls, according to the SBM client-server protocol. Here, SBM service specific messages are assembled and sent to the server using traditional socket level API. It runs an independent task and communicates with various other modules (call scheduling, Download and IPD) through specific IPC mechanisms.

Download Module:- The download component unpacks modules, stores them in a file-system and causes the dynamic loader to integrate the software module(s) in the client's software stack. It ensures that the data which is downloaded finds its appropriate place within the device. For this, it uses functions exported by the loader for dynamically loading and unloading modules to and from the memory.

Diagnostics Module:- The diagnostics module periodically reports performance related data about the device to the server. It collects diagnostics information from other modules within the device and depending upon the severity of the bug detected, diagnostic data can be immediately uploaded to the SBM server.

4. SBM COMPONENTS AND TOOLS

To enable dynamic downloads, modules in SBM client should function very similar to a COM [5] object. This imposes certain requirements in a SBM client device: (a) a *modular* code organization (b) a *dynamic loader* that can enables run-time binding including dynamic loading and unloading of modules in a client device, (c) external to SBM, a compile and link time *wrapper tool* that can proxy patch every software module for use by the dynamic loader in the device and finally, (d) an externally available *package builder* software to package data (software modules) intended for distribution among the clients. The following sections discuss this in detail.

4.1. Dynamic Loader (DynRTL) Module

The dynamic loader (DynRTL) module offers run-time support for resolving inter-module function references (binding) as well as for dynamically loading and unloading of modules. DynRTL module makes extensive use of the target's system symbol table. The symbol table contains externally accessible functions in the system and its associated memory addresses. DynRTL is responsible for relocation, registration of entry points for existing and any new modules with the system symbol table and if needed, its removal as well. It is used by the wrapper layer of wrapped modules, the download module within SBM subsystem and part of the boot procedure (in VxWorks) that loads and starts the application software.

Figure 4(b) shows the functions exported by the DynRTL Module. The entry point ("front door") to DynRTL is the function *DnLd_Main*, which is used by the download module in SBM client to transfer downloaded data (essentially bag of bytes) to the dynamic loader. The *DnLd_Main* acts as an integration module that combines certain functionality borrowed from the download module. Operations that can be specified: (a) Loading, unloading and *soft* replacement of modules, (b) Installing, deleting and replacing files in the local file system (data files as well as application software), (c) Creating or deleting directories in the local file system and (d) Identifying modules that must be loaded and started when system is booted.

The rest of the exported ("back-door") functions in dynRTL are used by other modules used in SBM, and also by the *DnLd_Main* function and any application software used in the client device. Figure 4(a) shows two software modules - dynamic binding with "Module1.out" and dynamic update of "Module2.out".

4.2. Wrapping Modules for dynamic binding

A wrapper tool generates wrapper layer to be externally compiled and wrapped for each module. It has functions that implement dynamic binding, as well as functions to enable detach mechanism for "soft" replacement. It intercepts unresolved references to external functions and invokes dynamic binding, announces a module that is going to be replaced and provides necessary functions that can enable those modules to be detached from the server (retreating) module. To intercept external function references, SBM makes use of proxy functions. The proxy function makes use of a *jump* instruction to redirect to a *Bind* function that eventually resolves the reference.

All proxy functions transfer control to the *Bind* function. The *Bind* function consists of code that is common to all proxies. The machine code for *Bind* is patched to the wrapper layer using the wrapper tool. Its purpose is to call the function *dyl_Bind* that is a part of the Dynamic Loader (DynRTL) module, insert the address obtained into the proxy and then re-execute the proxy function. The function *dyl_Bind* finally resolves the call.

The *moduleName* argument in *dyn_Bind* function is used in the administration of client-server module relationship. It is needed when a module (server) retreats and so must ask all other modules (clients) that are using it to detach. The use of a *can_retreat* function is vital for modules that are *active*. Active modules run

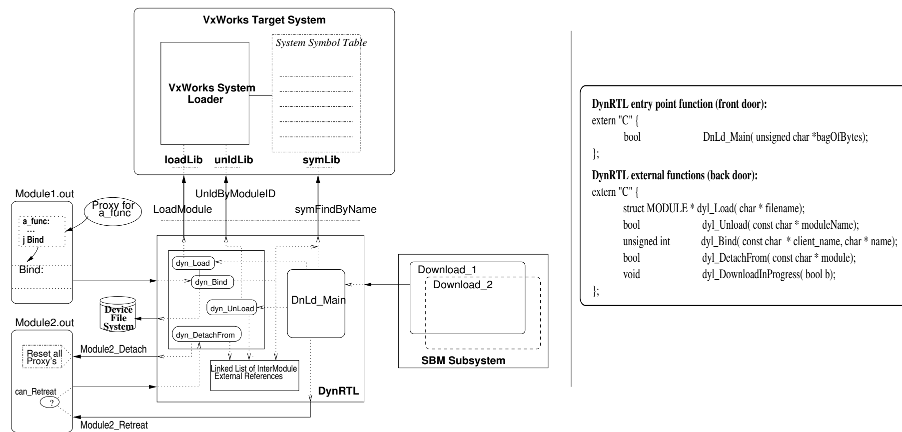

Fig. 4. (a) Dynamic Binding and Update in DynRTL (b) DynRTL Interface

independent tasks and are often difficult to replace. The difficulty in updating an active module is related to its timing model. Poorly timed models can lead to race conditions during state transformation. An active module to be replaced should first get an opportunity to transfer state information to its successor. Traditionally, this has been possible by using user provided timing constraints [7] (see p.g. 122-123). In SBM, we have made use of a *can_retreat* function, implemented in all the active modules within the device. The use of this function ensures that an active module first deactivates itself, before being finally removed. The function investigates whether the internal state of the module allows a retreat, waits until ongoing function calls have returned and saves pertinent state information in persistent memory to be picked up later by the replaced module. While *can_retreat* can be externally patched into 3rd party or legacy software, it remains an open question if *can_retreat* can make effective use of the internal state information of the software being patched.

4.3. Wrapper Generator Tool and Package Builder

The wrapper tool is a multi-function tool controlled by command line options. The tool generates a wrapper layer from: (a) a file "wrapper.s" containing MIPS assembly code for the proxy functions corresponding to each unresolved external function symbols, the *Bind* function and a proxy table data structure and, (b) a 'C++' source file "wrapper.cpp" containing functions (*Detach* and *Retreat*) used during dynamic replacement of the module.

The wrapper tool operates at various steps during the wrapping process of an object module. The underlying technology with this tool is its ability to read and interpret the ELF (Executable and Linking format) files. The obvious choice to select ELF is its support towards the target platform, VxWorks. The centre of the program is a class file *Elf32File* that can read, modify and write (parts of) ELF files. The class reads the section headers, the symbol table and the associated string table. The only thing it can modify is the attributes of symbols. The file layout is not changed by this operation, and only the modified symbol table is written back in its original place. When generating the wrapper for an object module, the program also generates a text file with hard-coded names of all the unresolved external function symbols for that module. This is later used to convert attributes of proxy functions from global to local.

A *PackageBuilder* software was also developed to package data intended for download by the client devices. It consists of a user interface that provides options for packaging data. The general structure of a package consists of two-level hierarchy that assist in sequential as well as simultaneous replacement of software modules in a device. At the highest level, groups are specified, which are processed sequentially. Each group consists of one or more entries that are conceptually processed simultaneously. An entry specifies the operation to be performed (for e.g. remove, force retreat, download etc.), the location of the new module within the device file system and details of how to effectuate the downloaded data (e.g. reboot, load immediately, load at boot etc.).

5. SBM PROTOCOL ALTERNATIVES

In this section we evaluate a number of other possible protocol alternatives for remote device management. SNMP (Simple Network Management Protocol), SIP (Session Initiation Protocol), serial consoles, telnet or any other proprietary network protocol can be used as opposed to a HTTP based approach. SNMP offers a standardized way for access and alert (event notification) methods to devices. Abstract Syntax Notation (ASN) standard ensures interoperability with different vendors and the use of UDP makes it a lightweight protocol. However, network management (NMS) platforms using SNMP are particularly costly, and not always user friendly. SNMPv1 offers little security (use of plain text community names) and the fact that it uses UDP makes it an unreliable transport option for device upload and download functionality. Also, devices making use of user interfaces (UIs) will require extra developmental efforts on part of a vendor that targets a specific platform/OS. While efforts will be expended to create and maintain such UIs, its integration into disparate devices may entail a host of other problems.

SIP extends interoperability, mobility, security and scalability essential for device-to-device communication paradigm. However, SBM and SIP are mutually exclusive models; each address their own problem space. While SBM is a client-server device management framework; SIP, which advocates use of UDP is particularly suited for query and asynchronous event notification schemes used in device-to-device control. A possible extension in SIP might enable support for SBM in future.

Serial consoles provide cheap, easy to implement and a fairly secure remote access method. They are good for local access, however, they cannot be easily used to manage devices located across enterprise networks. In case of Telnet, network access is possible by re-directing the console through a Telnet driven session. Again, such simplistic schemes fail to offer the kind of advantages that HTTP can offer for remote device management.

6. SECURITY ISSUES IN SBM

While security for client devices is crucial, server authentication is imperative since an intruder can impersonate and pretend to act as a server and usurp control of the device. In the jargon of security, this is commonly known as "man-in-middle" attack. The implications of such attacks can range from exposition of device related information to disruptive code injection or wrongful device control. SBM client devices and service protocol messages can also be tampered with to violate integrity. As a priori remedy for all such attacks, SBM targets a security policy model that is multi-level. Device and server authentication using session-level digital certificates and combinations of other forms of authentication (e.g. HTTP digest) can be used. In addition, SBM recommends encryption of all messages and data exchanged between the clients and the server. This is possible by making use of security protocols such as SSL/TLS that provides proper client-server authentication and encryption. Additional network level security available in a trusted network (e.g. private network using IP security protocol) can provide implicit security between the SBM clients and the server.

For those devices that implement session layer security mechanism, it is possible that these devices might decide to bypass the proxy and establish a direct secure connection with the SBM server. Alternatively, secure tunnelling through the edge proxy can also be enabled for such devices.

7. RELATED WORK

Limited papers can be cited in the area remote device management. An article by Sharon et. al. [2] elaborates on *class 3* devices that are smart; devices that interpret user behaviour and enable implicit device reconfiguration to enhance user experience. Schmidt et. al. [4] discuss approaches to build *smart* devices and clearly relate *context awareness* to device smartness. SBM offers no specific interpretation to device smartness (as *context awareness* is typically device dependent), however, like a class 3 device as in [2], a SBM client device is smart enough to enable dynamic software reconfiguration. Research in related area, but not directed towards Internet devices, have focussed extensively on dynamic software updates [7] and code instrumentation techniques such as *Detours* [8]. SBM achieves dynamic reconfigurability using a function interception technique very similar to the one found in *Detours*. But unlike *Detours* that intercepts and patches proxy code to external functions during execution time (in memory), patching in SBM is possible using a special wrapper tool that wraps proxy patches to external function references in software modules within a device.

8. SBM AND OSGI

The Open Services Gateway Initiative (OSGi) [6] is an important step in the direction of remote management of Internet devices. The goals in OSGi are: (i) building a common platform for development of network applications, (ii) creating a bridge between home network and the Internet and, (iii) incorporating bridges among home devices that have heterogeneous physical-layer and control-protocol standards.

In many ways, both SBM and OGSi synergize the concepts of remote device management. OSGi (Release 2) seeks to create an open specification for network delivery of managed services to local networks and devices. It offers an execution environment for electronically downloadable services. In OSGi terminology, services are known as *bundles* that executes in a Java runtime environment by using an OSGi framework registry. Using such a registry, bundles can also be made to inter-operate.

While SBM offers dynamic downloads using native code in a device, OSGi benefits using Java to offer a generic platform for all applications. In many other ways, SBM and OSGi share similar goals. Hence implementation experiences from SBM can be offered to augment the essential goals and objectives in OSGi.

9. SUMMARY AND CONCLUSION

The idea to remotely manage dynamically updateable devices is fairly new that offers completely new opportunities, options and challenges. An SBM client device can self-activate, (re)configure and dynamically update or customize its internal software stack with new standards, features and/or applications. The novelty in SBM is that it extends dynamic reconfigurability applied to native code within an embedded client device. Presence of a reconfigurable software stack makes it possible for the device to have a dynamic update of any software module (e.g. a new release) that are eventually bug-free.

SBM promises an enabling technology; however, a detailed evaluation of the requirements for a range of such Internet capable devices would still be required. SBM is a only a step forward towards realization of such efforts that will be involved in the management of the Internet-networked devices.

10. REFERENCES

[1] R. Fielding et. al., "Hypertext Transfer Protocol-HTTP/1.1", *IETF RFC 2616*, June 1999.

[2] Sharon Eisner Gillett et. al., "Do Appliances Threaten Internet Innovation?", *IEEE Communications Magazine*, pp.46-51, October 2001.

[3] Janne Riihijarvi, et. al., "Providing Network Connectivity for Small Appliances: A Functionally Mimnimized Embedded Web Server", *IEEE Communications Magazine*, pp.74-79, October 2001.

[4] Albrecht Schmidt and Kristof Van Laerhoven, "How to build smart appliances?" *IEEE Personal Communications Magazine*, pp. 66-71, August 2001.

[5] Microsoft COM, http://www.microsoft.com/com/

[6] The Open Services Gateway Initiative, http://www.osgi.org/

[7] Micheal Hicks, *Dynamic Software Updating*, PhD Thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[8] G. Hunt and D. Brubascher, "Detours: Binary Interception of Win32 Functions", In *Proc. of the 3rd USENIX Windows NT Symposium*, pp. 135-143. Seattle, WA, July, 1999