Experiences with Building Distributed Middleware for Home Computing on Commodity Software

Tatsuo Nakajima, Shuichi Oikawa, Hiro Ishikawa, Kunitoshi Iwasaki and Midori Sugaya
Department of Information and Computer Science
Waseda University
tatsuo@dcl.info.waseda.ac.jp

Abstract

In this paper, we describe our experiences with developing two middleware infrastructures for networked home appliances on commodity software platforms to show what future distributed system designers need to take into account.

1 Introduction

High level abstraction for building advanced home computing applications should be provided to develop home appliances in an easy way. If there is no such abstraction that can be openly used by various third-parties, a few people can make advanced applications, but this does not make us happy since attractive and useful home entertainment services are usually developed by usual users. Also, in home computing environments, high level abstraction is important to support context-awareness. Context-awareness provides attractive features to applications, such as personalization and adaptation of the applications' behavior according to the current situations. However, implementing high level abstraction is not easy since it is complex. One of ways to solve the problem is to enhance the portability of programs by implementing them on commodity platforms.

In this paper, we describe our experiences with developing middleware infrastructures for networked home appliances on commodity software platforms. In our project, we have developed middleware infrastructures on commodity software platforms explicitly to structure complex software for increasing portability and for reducing development cost. We present two middleware infrastructures for building home computing environments, which have developed in our projects. The first middleware is used to build HAVi-based audio and visual home appliances. The middleware provides high level abstraction to build advanced audio and visual home computing applications. The second middleware is used to build a Web-based home computing system that integrates various home appliances by using Web-based standard protocols.

2 Software Design for Complex Software

We have developed several middleware infrastructures on a microkernel-based operating systems in the past projects[3, 4], and our experiences with developing them show that the microkernel is useful to design various high-level abstraction. However, we also found that the abstraction provided by the microkernel is too generic and usual programmers are not easy to develop software on the low level abstraction. Thus, it is more difficult to write programs on microkernel-based operating systems that provide more generic low level abstraction like L4[2] or Exokernel[1]. Therefore, we believe that it is hard to provide the low level generic abstraction and develop high level abstractions on the low level abstraction.

Another approach to solve the problem is to adopt a software platform that is widely available. The solution allows us to build complex software on various appliances in an easy way if the platform can be used on a wide range of appliances. Also, adopting widely used commodity platforms makes software highly portable.

We already have a lot of existing software on various commodity platforms, and reusing these software is important to reduce development cost of complex software. Also, there is no software platform that is suitable for developing various types of applications because we need to take into account a variety of requirements to design complex software. Therefore, it is necessary to consider tradeoffs among the requirements. This means that it is desirable to adopt several commodity software platforms simultaneously according to the characteristics of respective software components, and to use different software platforms for executing respective middleware infrastructures.

In our approach, we configure several software platforms in a layered structure. We call the structuring *multi-layered software platform*. Software in our middleware infrastructures is divided into several components. For example, one component requires to be developed rapidly, but it does not require severe timing constraints. On the other hand, it is more important to satisfy timing constraints of another component than to reduce development cost.

Our approach allows us to replace a lower level software platform according to the characteristics of hardware platforms. For example, we may like to replace lower level software platforms such as Linux to either WindowsCE, VxWorks or EPOC due to the availability of device drivers, various libraries, or these resource management capabilities. Traditionally, providing multiple software platforms assumes to adopt a generic low level platform for supporting to build various high level software platforms, but the generic low level platform does not have adopted widely. Also, in the approach, it is not

easy to use multiple commodity software platforms simultaneously to build one system. Our approach is similar to operating system emulation, which emulates an operating system interface on another operating system interface. For example, WINE emulates Microsoft Win32 API on POSIX based operating systems, and Cygwin emulates the POSIX interface on the Win32 API. The approach allows us to reuse existing applications running on different operating systems. Also, our approach is similar to a virtual machine approach. The virtual machine interface is a kind of a low level platform described above. Although the approach allows us to use many applications on different operating systems running on a virtual machine monitor, there is no explicit way to use multiple commodity software platforms for developing complex software. We believe that we need a way to compose multiple widely adopted software platforms in a system for developing a large scaled complex program.

Our paper does not present a systematic way how to structure multi-layered software platform, but we provide our experiences with building complex software by using multi-layered software platform. We believe that the experiences are useful to develop a systematic methodology to build complex software by using multi-layered software platform in the future. Also, we think that the experiences are useful to develop a new structuring technique for future distributed systems. It is necessary to develop a more systematic structuring technique to build complex middleware infrastructures that can be customized for each application and hardware platform.

3 Two Middleware Infrastructures for Home Computing

In this section, we describe two middleware infrastructures for home computing, that have been developed in our projects. In [10] and [11], we show details about these middleware infrastructures. The paper shows overviews of these infrastructures and how they work.

3.1 Distributed Middleware for Networked Audio and Visual Home Appliances

3.1.1 Overview

The following issues are considered when designing the middleware.

- We like to gain experiences with building complex middleware on commodity software.

- We like to evaluate the effectiveness of high level abstraction.

In the first issue, our system has been developed on Linux and Java. Especially, our system evaluates to use Java for embedded systems. In the second issue, we have developed several applications on HAVi to evaluate the effectiveness of the high level abstraction[10].

The most important component in our home computing system is HAVi that is a distributed middleware infrastructure for audio and visual home appliances. HAVi provides standard high level API to control various A/V appliances. The reason to choose HAVi as a core component in our distributed middleware for audio and visual home appliances is that HAVi is documented very well, and some commercial products will be available in the near future. Therefore, it is possible to use the products in our future experiments. The HAVi component is written in Java, and the Java virtual machine for executing HAVi runs on the Linux operating system. The second component is a continuous media management component processes audio and video streams. Currently, our system can handle MPEG2 and DV streams, and emulates several appliances such as DV cameras and digital televisions. The component needs to process continuous media in a timely manner. Thus, the component has been implemented in the C language, and runs on Linux directly.

3.1.2 How the System Works?

In this section, we present a sample scenario to build a home appliance using our prototype system. The scenario has actually been implemented using our prototype and give us a lot of experiences with building distributed home computing environments.

The scenario emulates a HAVi-based digital TV appliance. A Linux process to emulate a digital TV receives a MPEG2 stream from an IEEE 1394 device driver, and displays the MPEG2 stream on the screen. The sender program that simulates a digital TV tuner receives the MPEG2 stream from a MPEG encoder, and transmits it to the IEEE 1394 network. Currently, the MPEG2 encoder is connected to an analog TV to simulate a digital TV tuner, and the analog TV is controlled by a remote controller connected to a computer via a serial line. A command received from the serial line is transmitted to the analog TV through infrared.

The receiver process contains three threads. The first thread initializes and controls the program. When the thread is started, it sends a DCM containing a Java bytecode to a HAVi device. In our system, the HAVi device is also a Linux based PC system. When the HAVi device receives the bytecode, a HAVi application on the HAVi device shows graphical user interface to control the emulated digital TV appliance. If a user enters a command through the graphical user interface, the HAVi application invokes a method of the DCM representing a tuner. Then, the HAVi device executes the downloaded bytecode to send a command to the emulated digital TV appliance, and the first thread will receive the command.

After the second thread in the emulated digital TV appliance receives packets containing MPEG2 video stream from an IEEE 1394 network, it constructs a video frame, and inserts the frame in a buffer. The third thread retrieves the frame from the buffer, then the video frame is delivered to a MPEG2 decoder according to the timestamp recorded in the video frame. Finally, the decoder delivers the frame to a NTSC line connected to the analog TV display.

3.2 Web-based Home Appliance Middleware

3.2.1 Overview

This section presents our Web-based home computing system. The middleware infrastructure supports a set of

high level protocols to integrate a variety of home appliances. Web-based means that users can access home appliances from standard Web browsers via the HTTP protocol. Therefore, the access to home appliances becomes uniform and easy due to the Web-based operations. Also, our system connects appliances that support various protocols for home computing such as Jini and HAVi by converting the protocols to the HTTP protocol[6].

The following goals are taken into account to design our system.

- Our system should be simple. Especially, it should be avoided to make appliances too complex.

- Our system should be integrated with various services on the Internet.

- Our system should support context-awareness.

Our system can realize the goals to adopt the Web-based protocol. The most important requirement for designing our Web-based home computing system is that an appliance containing a Web server should not be modified. Because commercial appliances do not allow us to modify their Web servers. Also, developing a new plug and play protocol is not easy since it is hard to standardize the new protocol. However, automatic configuration that can be found in Jini or HAVi must be supported for building home computing environments because usual people does not know how to configure home networks.

Our home computing system provides a Jini like directory server, and it makes it possible to integrate Web-based home appliances. In our current implementation, a directory server is implemented in Java because there are many exisiting class libraries to develop the directory server easily. On the other hand, Web-based home appliances are written in the C language since accessing special devices are easier from C programs, and we assume that complex functionalities should be implemented in the directory server. However, our system does not assume the extension of the Web server to integrate commercial products that contains Web servers in our system. Therefore, our directory server finds all connected Web-based appliances proactively. This means that the server contacts a DHCP server and retrieves IP addresses leased for all appliances connected to a network. Then, the directory server transmits HTTP GET requests to all appliances whose IP addresses are retrieved by the DHCP server. If a file describing service specification provided by each appliance is returned, the information is stored in the database on the directory server that is used for routing HTTP requests received from clients to target appliances. Also, the information can be used to offer advanced services such as context-awareness.

3.2.2 How the System Works?
In this section, we describe a sample scenario that shows how our system works. In the scenario, a user has a PDA device for controlling appliances. These appliances contain Web servers, and a directory server collects information about the currently available appliances. In the current example, the database converts a function name to a house code and a device code of a X10 module connected to an appliance indicated by the function name.

When a PDA device opens a Web browser in our home, the first HTTP GET request is snooped by a gateway Web server running on a router. The server returns a HTML page containing a list of appliances that are currently available in our house. The page is automatically generated by the directory server using information collected from Web-based appliances. In our example, we assume that the page contains a link to http://A.B.C.D//?func=TV/!power=on/ that means to turn a TV appliance on in our house, where A.B.C.D is an IP address of the directory server.

Let us assume that a user sends a command to turn on the TV appliance that is connected to a X10 module by clicking the above URL. The browser of the PDA device transmits a HTTP GET command with http://A.B.C.D/?func=TV/!power=on/. The URL is translated to http://A.B.C.E/?house=a&device=4/!power=on/ on the directory server, and forwards the URL to the Web server of the TV appliance, where A.B.C.E is an IP address of the TV appliance. This means that a user accesses the television of our home without knowing the IP address or the host name of the directory server since the address is automatically searched by the system. Also, "http://A.B.C.D/?func=TV/" is translated to "http://A.B.C.E/". This means that the user needs not to know the actual IP address or the host name of our television. Finally, when the Web server of the TV appliance receives the URL, the server turns the power switch of the TV appliance on by sending a X10 command to turn on a X10 module to be connected to the TV appliance.

4 Experiences with Building Middleware Infrastructures

4.1 Experiences with Linux

This section describes several experiences to use Linux as an operating system for building home appliances. Linux provides a lot of useful functionalities such as networking, file systems, window systems, and web browser to build a variety of future information appliances. Therefore, it is easy to create advanced applications using these features. Also, there are several extensions to support real-time resource management. Especially, Linux/RT[9], that is used in our project, provides a resource reservation capability that is useful to make continuous media processing predictable. These features are desirable to build future networked home appliances.

Also, a problem occurs when we choose Linux/RT as embedded Linux. Linux/RT extends standard Linux API by inserting dynamic loadable kernel modules that provides real-time functionalities such as real-time synchronization, fine-grained clocks and resource reservation capabilities. The interface is not standardized in the Linux community so that software that accesses the interface should be modified when the software is ported on other Linux kernels. The fact decreases the value to use commodity software significantly.

The unpredictable worst case response time of Linux is not appropriate for processing continuous media such as audio and video because the response time prevents continuous media from being processed at correct time. The problem is caused due to the priority inversion in the Linux kernel, and we found that non preemptability in the kernel is the most serious factor of the priority inversion. However, a simple extension to make the kernel preemptable solves the problem[7], and Linux makes it easier to build audio and visual home appliances than traditional real-time operating systems.

4.2 Experiences with Java

The first experience when building our prototype system is that the current Java does not provide a mechanism to efficiently communicate between Java programs and C programs running on Linux. There is two ways to solve the problems. The first way is to add a new primitive using the Java native interface, but the approach decreases the portability of software. The second way is to use the socket interface. The approach is portable, but is inefficient due to the overhead caused by the Internet protocol. We require more high level abstraction to support communication between C programs running on Linux and Java programs.

The second experience concerns the behavior of the Java virtual machine. The implementation of threads and synchronization is different on each virtual machine. This means that the behavior of each virtual machine is different. If the correctness of a program depends on the behavior, the portability of the program cannot be ensured. The behavior is also quite changed on each operating system because the behavior of each operating system differs according to the implementation. Especially, differences in scheduling behavior and the cost to create threads make the behavior of Java programs unpredictable. Therefore, it is not easy to develop Java programs that are executed efficiently on a variety of platforms although a lot of people claim that Java offers "Write Once, Run Everywhere". If a Java program needs to deal with timing critical processing, the situation becomes more serious. We believe that a commodity software platform should specify its behavior in a rigorous way.

Currently, the real-time specification for Java has been developed and the specification provides a variety of memory management schemes. Especially, scoped memory allows us not to use garbage collection to reclaim unused memory, and physical memory access allows us to write device drivers in Java. Real-Time Java can support various levels of abstraction. Thus, it seems that we can adopt a single software platform for building various software. However, although Real-Time Java provides a low level abstraction for memory management, high level abstraction is still provided by other services such as file and network services, and the abstraction may not be appropriate to build complex embedded software. Therefore, Real-Time Java will not replace all programs written in the C or C++ language, but we think that Real-Time Java is useful to reduce the development cost of complex software. Because Java hides memory management issues from programmers, but they may need to know the cur-

rent memory utilization to build stable software, and Real-Time Java provides a way to control physical resources allocated to respective applications.

4.3 Middleware for Home Computing

In our project, HAVi based middleware has been developed first, and we have developed several applications on Jini and UPnP for evaluating home computing middleware. Our experiences show that HAVi is too complex to be installed in home appliances. Thus, the cost of an appliance is increased by adopting HAVi. Also, it is difficult to use Jini for build home appliances due to the current implementation of Jini. It is necessary to completely reimplement the current implementation of Jini to be used for building embedded systems. Also it is necessary to define a standard protocol between Jini-based devices for satisfying the interoperability among appliances. Thus, we believe that Jini is not suitable to be used for embedded systems in the near future.

We think that it is important to integrate with Web services to develop attractive applications, but HAVi and Jini need different API to access Web services and to control home appliances. UPnP is promising in the near future because each appliance needs to implement a small size of programs to support UPnP. Also, UPnP adopts Web-based protocols to control appliances. Thus, it is easy to be integrated with Web services. However, the naming scheme provided by UPnP for identifying appliances is very weak.

4.4 High Level Abstraction

We have implemented several home computing applications on our implementation of HAVi. One of these applications is a remote control application that is integrated with a Web server. It enables the application to convert commands from the embedded Web server to commands to access the HAVi API. This means that it is easy to build an application to control home appliances from a variety of control devices embedding Web browsers. We believe that the key factor to realize fantastic home computing environments is to provide high level API that enables us to build a variety of home applications easily.

The high level API provided by HAVi allows us to build advanced home applications that are personalized for each person. For example, it is easy to build a program customizing graphical user interface according to the preference of a user. The program identifies a user, and changes the layout of of graphical user interface according to the user. However, it is not easy to identify the current user who likes to use an appliance. Also, it is not easy to represent context information to customize the behavior of applications. For example, it is not simple to generate graphical user interface according to the currently available appliances according to the preference of a user because the look and feel of an automatically generated GUI is usually shabby for home appliances.

The high level API enables us to compose several appliances, but it is not realistic to create an application for respective compositions. Therefore, it is important to build an application to compose appliances from a composition specification. The specification should be declarative, and

it should be possible to compose several specifications to reuse existing specifications. Our experience shows that the current high level API provided by HAVi does not have enough power to represent the composition. Especially, we believe that the naming scheme for identifying functions contained in appliances and a variety of services is a key issue to develop a useful composition specification.

On the other hand, we found that our naming scheme is powerful to develop home computing applications. Our naming scheme can support from a low level naming such as supporting X10 device naming to a high level naming such as supporting context-aware naming. We believe that our naming scheme is flexible enough to satisfy the requirements such as composing appliances in a spontaneous way.

4.5 Distribution Transparency

The issue discussed in this section is how to manage distribution. Home appliances connected by networks store a variety of information that should be consistent whenever some failures occur or the configuration of networks is changed. Also, to build stable distributed systems, it is important to define rigorous semantics for all operations in systems.

The following two issues are usually ignored by the current researches on home computing. The first issue is how to deal with failures of appliances. There is a lot of work to maintain consistency in distributed systems such as transactions and process groups, and these concepts are useful to build distributed home appliances. For example, a digital TV appliance may invoke a transaction for on-line shopping. The database to monitor the behavior of users should keep the consistency even if an appliance is turned off while modifying the database. We believe that transactions are also a useful concept in home computing environments, and it is an important to develop an appropriate transaction model for home computing environments. Also, fault tolerance is an important topic in home computing environments since it is not desirable to fail to record a broadcast program due to the crash of an appliance.

We believe that receovery-oriented techniques[8] are a promising approach to make home computing reliable because the techniques are light-weighted. However, both middleware infrastructures and application programs should be taken into account quick recovery while designing them. We think that it is desirable to develop a component framework that supports recovery-oriented computing directly.

It is important to reconsider each popular abstraction that is used currently because home computing environments require to take into account different tradeoffs from traditional distributed systems.

We need to reconsider what low level properties can be hidden from application programmers in distributed home computing environments because implementing complete distribution transparency is impossible[12].

4.6 Interface vs. Protocol

HAVi offers standard programming interface to develop home computing applications. The interface enables us

to build portable applications that can run on various vendor's appliances. However, each appliance likes to extend the interface to add extra values to be appealed to users. Also, the implementation of middleware infrastructures usually becomes large and complex due to the standard interface that likes to satisfy various requirements. Our experiences show that standard interface-based approaches such as CORBA and DCOM are not appropriate for home computing. Since respective home appliances are independently upgraded or developed in a rapid way. We believe that programming interface should be customized for respective underlying platforms. In this case, protocols among appliances are standardized, and the protocol should be flexible and simple. Our Web-based home computing infrastructure chooses HTTP-based protocols. Each vendor can choose his own implementation and programming interface in our approach. We believe that the approach is more suitable than traditional distributed objects for rapidly changing environments like home computing environments. However, we need to investigate the topic in the near future since portability is an important topic in these environments.

4.7 Multi-Layered Software Platform

The first experience is that it is difficult to understand the behavior of each software platform. Our knowledge about respective platforms is qualitative. Therefore, we may find that our assumption is wrong after developing programs. We believe that it is important to define the assumption of each software platform explicitly in a rigorous way. Therefore, it is possible to predict the behavior of each software platform, and a programmer can select a suitable software platform correctly.

The second experience is that there is no standard way to communicate between programs running on different software platforms. For example, there is no standard API to communicate between a Java program and a C program running on Linux. We believe that it is desirable to develop a component framework that provides a mechanism to communicate among programs running on different software platforms. If the component framework is developed, a programmer does not take into account which platform needs to be adopted to develop his component since the framework can hide its implementation.

Our current approach that supports multi-layered software platform is ad-hoc. We also have an experience to design the Linux on the ITRON operating system[1]. In the system, one part of a program, which requires a fast response time or to use existing ITRON based software, is executed on ITRON, but another part, which requires to be developed rapidly or to use existing Linux based software, runs on Linux. The approach enables us to balance between development cost and fast response time, but implementing Linux on ITRON requires many ad-hoc engineering efforts.

---

[1]ITRON is a small operating system, that is widely used for various embedded systems in Japan. Japan Embedded Linux Consortium is working to define the Linux on ITRON specification. It makes us to reuse existing software on both ITRON and Linux, and to build advanced embedded systems rapidly.

4.8 Porting Issues in Embedded Systems

In embedded systems, underlying platforms provide a variety of advanced features. The features are different on respective platforms, and it is impossible to standardize these features since the standardization may hide some useful details of respective platforms. Also, it is important to support Quality of Service(QOS) parameters to accommodate a variety of resource constraints. However, a general QOS specification is not suitable for embedded systems since the specification will be too complex and big. Therefore, it is desirable to provide a specialized QOS specification for each platform. The QOS specification should take into account the predictability of underlying software such as scheduling behavior and worst case execution time.

However, the approach makes the portability of software bad since it is necessary to rewrite programs when they are ported on other platforms. We need to develop a new methodology to develop a portable program that can be adapted to respective platforms.

In the future, our platform should be enhanced by using "System on Chip" solutions. The special hardware requires us to be accessed via special purpose interface to exploit the maximum power of the hardware. However, future systems will be structured in a layered fashion, and every layer should provide special interface for special hardwares. Also, context-awareness and resource-awareness need to provide API that should be customized for respective platforms. For example, Java virtual machines need to provide API for control their garbage collection algorithms, but the API should be different according to the algorithms. The above discussion introduces the importance to provide ad-hoc API extension for respective implementation of a platform. We need to investigate a methodology that does not degrade the portability of software even when the ad-hoc extension is added for exploiting underlying hardware and system platforms.

We believe that we need a new modularization technique to add non functional properties such as predictability, reliability, and security or to extend functional properties in a systematic way to existing software. There are many new proposals to increase the modularity of software. We like to investigate these modularization techniques for increasing the portability of middleware for home computing.

4.9 Portability and Middleware Implementation

A standard middleware component provides a common application programming interface that enables us to build portable applications. There are many open specifications for implementing commodity software. Although each product has a different implementation, an application using a common interface runs on any implementations. However, we have seen several problems in our experiences. Since Java virtual machines that have different implementation behave in different ways, the application's behavior is also changed. For example, the implementation of scheduling and synchronization has a great impact on the behavior of applications.

The issue is important to develop future environments based on commodity software. For example, if each implementation raises a different exception for the same internal fault, it is difficult to implement a portable application. In another example, if a location system returns a different granularity(precision and accuracy) of locations information, it is hard to implement portable location-aware applications. Also, if respective implementations provide different levels of security, reliability, and predictability it is impossible to build portable software. The observation shows the importance to specify the assumption of each implementation explicitly, and the specification of the behavior in an abstract way.

5 Conclusion

In this paper, we have presented several experiences with building two middleware infrastructures for home computing, that are developed in our projects. We believe that the experiences are important to build future universal software substrate for home/ubiquitous computing[5].

References

[1] D.R.Engler, M.F. Kaashoek, J.O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management", In Proceedings of the 15th Symposium on Operating System Principles, 1995.

[2] J.Liedtke, "μ-Kernel Construction", In Proceedings of 15th Symposium on Operating System Principles, 1995.

[3] T.Nakajima, T.Kitayama, H.Tokuda, "Experiments with Real-Time Servers in Real-Time Mach", USENIX 3rd Mach Symposium, 1993.

[4] T.Nakajima, H.Tezuka, A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach, ACM Multimedia'94, 1994.

[5] T. Nakajima, "Towards Universal Software Substrate for Distributed Embedded Systems", In Proceedings of the International Workshop on Real-Time Dependable Systems, 2001.

[6] T.Nakajima, D.Ueno, I.Satoh, H.Aizu, "A Virtual Overlay Network for Integrating Home Appliances", In the Proceedings of the 2nd International Symposium on Applications and the Internet, 2002.

[7] T. Nakajima, S. Ochiai, and K. Iwasaki, "Making Linux Predictable", In Proceedings of the International Workshop on Internet Appliances on Linux, 2002.

[8] D.A.Peterson, et. al., "Recovery Oriented Computing(ROC): Motivation, Techniques, and Case Studies", UC Berkeley, Technical Report UCB/CSD-02-1175, 2002.

[9] Timesys, "Linux/RT", http://www.timesys.com.

[10] K.Soejima, M.Matsuda, T.Iino, T.Hayashi, and T.Nakajima, "Building Audio and Visual Home Applications on Commodity Software", IEEE Transactions on Consumer Electronics, Vol.47, No.3, 2001.

[11] D.Ueno, T.Nakajima, I.Satoh, "Web-Based Middleware for Home Entertainments", Asian 2002. 2002.

[12] J.Waldo, G.Wyant, A.Wollrath and S.Kendall, "A Note on Distributed Computing", Technical Report, Sun Microsystems Labolatories, Inc., 1994.