行政院國家科學委員會專題研究計畫　成果報告

子計畫二:嵌入式 JAVA 及 JINI 系統之設計與實作

計畫類別：整合型計畫
計畫編號：NSC91-2218-E-002-006-
執行期間：91 年 08 月 01 日至 92 年 07 月 31 日
執行單位：國立臺灣大學資訊工程學系暨研究所

計畫主持人：陳俊良

報告類型：完整報告

處理方式：本計畫可公開查詢

中　華　民　國 93 年 2 月 20 日

行政院國家科學委員會補助專題研究計畫成果報告

eHome：電子家庭雛型之設計與實作(3/3)

子計畫二：嵌入式 Java 及 Jini 系統之設計與實作

計畫類別：整合型計畫
計畫編號：NSC 91－2218－E－002－006－
執行期間：91 年 08 月 01 日至 92 年 07 月 31 日

計畫主持人：陳俊良
共同主持人：
計畫參與人員：

成果報告類型(依經費核定清單規定繳交)：完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
□出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、
列管計畫及下列情形者外，得立即公開查詢
□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

執行單位：國立台灣大學資訊工程學系

中　華　民　國 92 年 12 月 31 日

Abstract

HAVi, Home Audio/Video interoperability, is one of the leading home-networking middleware technologies. Services in HAVi are modeled as objects called software elements, the basic units in HAVi software framework. The design and implementation of software elements greatly affect the performance of HAVi networks. However, there is little explanation for software element design in the HAVi specification. The purpose of this report is to discuss all software element design issues that developers may encounter and to provide their solutions. This report may serve as a guide for consumer electronics manufacturers and application developers to design and implement software elements in HAVi.

摘要

HAVi，即 Home Audio/Video interoperability 的縮寫，是目前家庭網路中間體(middleware)的主流技術。HAVi 以 IEEE1394 為基礎，主要目的是將家電相互連接且能傳送影音資料。在 HAVi 中，服務透過軟體元件(software element)提供，軟體元件是 HAVi 軟體結構的基礎單位。如何設計與實作軟體元件將影響 HAVi 網路的效率。然而在 HAVi 的規格中，並沒有針對軟體元件做詳細說明。此篇論文討論設計軟體元件將會遭遇的問題，並提供解決方案，其可供消費性電子廠商及應用程式設計師作為設計與實作軟體元件的參考。

Contents

List of Figures

List of Tables

List of Examples

Chapter 1

Introduction

1.1 Overview

In the past, a traditional home's only digital device was the PC. With the introduction of various digital appliances (digital camcorder, digital STB, digital TV, web pad, video games and so on.), the customer's need for inter-device information sharing, efficient appliances, and the internet has skyrocketed. People also hope that home appliances from different vendors can communicate to each other and be operated from anywhere in the home, using whichever appliance is nearest. The need for simple, flexible, and reliable home networks is greatly increasing. In this report, the term “home networking” is defined as the intelligent communication and the mutual data transfer among various digital home appliances.

Today, there are many home networking standards, both in underlying interconnection technologies and middleware. These standards are competing to be the next mainstream technology in home networking.

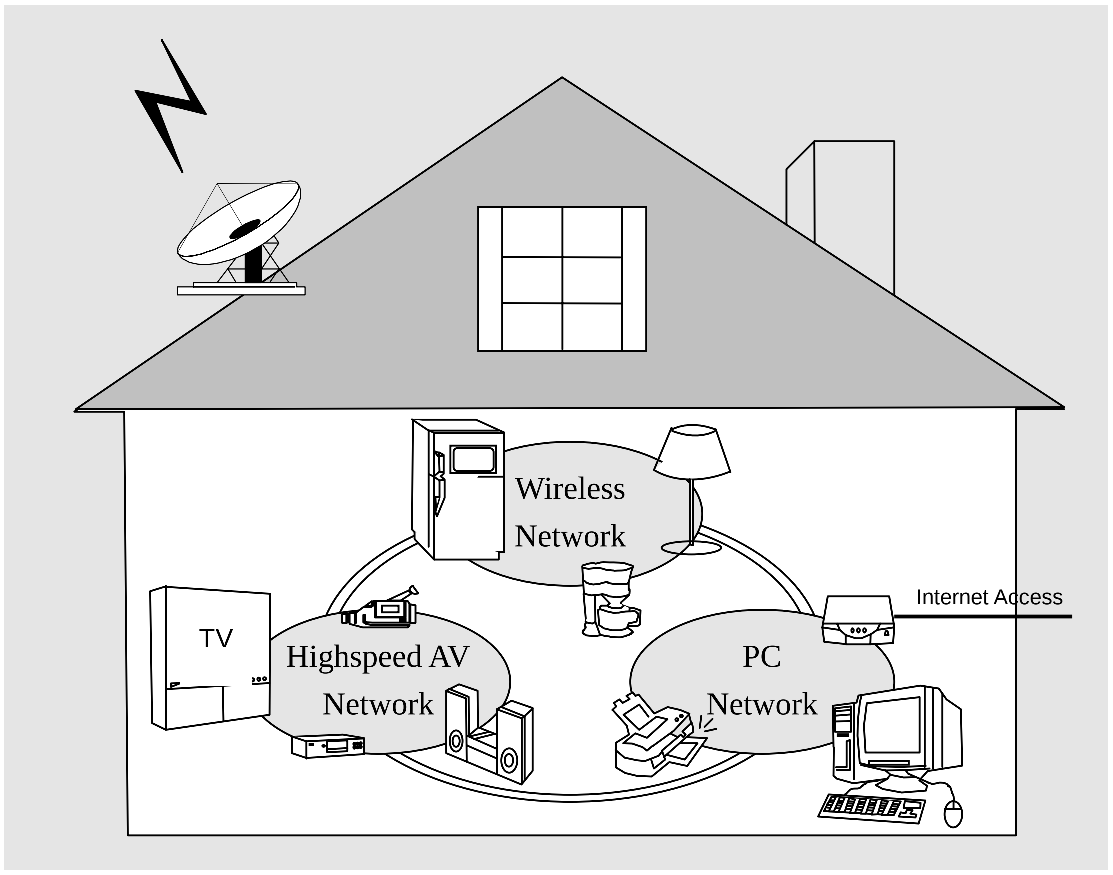


Figure 1-1: A possible future networked home--multiple technologies

1.2 Home Networking Technologies

Current home networking technologies can be divided into three categories: new wiring, existing wiring, and wireless [1].

New wiring imply that tearing down walls and floors to place wires, yet it provides several advantages such as high bandwidth, flexibility, and great security. The leading technologies include Ethernet [2] and IEEE 1394 [3]. Ethernet, the most widely used networking technology today, is transferred over cabling called CAT5. It is economical and extremely reliable but may be difficult to install and configure. IEEE 1394, also known as i.LINK or FireWire, is ideal for multimedia home networking due to its ability to provide high bandwidth (400Mbps) and isochronous data transfer.

Existing wiring system in home includes power lines and phone lines. HomePNA 2.0 [4], a phoneline home-networking technology, has 10Mbps bandwidth, and it can coexist with voice and xDSL signals on a single piece of telephone wire. HomePlug [5], CEBus, and X-10 are the leading powerline technologies. With multiple outlets in almost every room, powerlines present a cost-effective, ubiquitous, easy-to-adopt home networking solution.

Wireless home networking allows mobility and convenience that users can access from anywhere in home. Nevertheless, bandwidth and cost remain issues. Leading technologies include wireless LAN (IEEE 802.11) [6] and Bluetooth [7].

Table 1-1: Home networking technologies

Technologies	Max Speed	Media
New Wiring		
Gigabit Ethernet	1000Mbps	CAT5 UTP
IEEE 1394a	100-400Mbps	STP
Existing Wiring		
HomePNA 2.0	10Mbps	Phonelines
HomePlug 1.0	14Mbps	Powerlines
Wireless		
WLAN 802.11b	11Mbps	RF
Bluetooth 1.1	720Kbps	RF

1.3 Middleware

Home network applications cover home automation, configuration of devices, security tasks, multimedia applications, entertainment, communications, internet, and many other areas. Therefore, there is a need of middleware that provides services such as service and device discovery, co-scheduling of network resources, security, and messaging. Simply speaking, home networking middleware is a layer of software that lies between a home appliance operating system and applications. It provides Application Programming Interfaces (APIs) allowing consumer electronics manufacturers and third parties to develop applications for home networks. Leading home middleware standards include HAVi (Home Audio/Video Interoperability) [8], Jini [9], and UPnP (Universal Plug and Play) [10].

Table 1-2: Home networking middleware

Middleware	Media	Pioneered by	Comments
HAVi	IEEE 1394	Sony, Philips, and others	• Use Java [11] for highest interoperability • Focus on multimedia home networking
Jini	Any media	Sun Microsystems	• Based on the Java platform • Use Java Remote Method Invocation [12]
UPnP	Any media	Microsoft	• Leverage TCP/IP and the Web technology • Just send data over the network • Independent of OS and language

Each technology presents unique pros and cons, yet home networking is incomplete without the ability to transfer data, voice, and video together. Providing high-speed and reliable delivery of audio/video content, the combination of IEEE 1394 and HAVi show the most promise to become the standard for home networking.

1.4 HAVi

1.4.1 Overview

Home Audio/Video Interoperability, also known as HAVi, is a standard proposed by Grundig A.G., Hitachi, Ltd., Matsushita Electric Industrial Co., Ltd. (Panasonic), Royal Philips Electronics N.V., Sharp Corporation, Sony Corporation, Thomson Multimedia, and Toshiba Corporation in 1998. The main goal is to allow home appliances to communicate to each other. Moreover, HAVi has been designed to meet the particular demands of digital audio and video.

Here are some possible applications of a HAVi home network.

- People can control all home appliances via any fully HAVi-compliant device.
- A TV connects to a video telephone. When a phone call comes in, the TV can be muted and used as the display automatically. Then, people can answer the phone via the TV.
- A video camera can automatically display a picture on the TV screen when a visitor arrives; or start a recording if the same thing happens unexpectedly during the night.

IEEE 1394 has been chosen as the underlying interconnection medium. It has capacity to carry multiple digital audio and video streams simultaneously around the house, and it provides support for digital copy protection.

1.4.2 Benefits of HAVi

A HAVi compliant appliance can offer some advantages [13].

- Brand independence: Devices from different manufactures can communicate with each other in a HAVi network. For example, a Panasonic VCR can work with a Sony amplifier and be controlled by a Mitsubishi TV as long as all these devices are HAVi compliant.

- Interoperability: Functions on a device within a HAVi network can be controlled from another device within the network. For example, search for an available VCR to record a TV program, with commands being given via the menu selection of another TV display.

- Legacy appliances support: HAVi supports legacy appliances, including non-IEEE1394 devices and non-HAVi devices. This plays an important role since the transition to networked devices is gradual and there could be multiple standards in a home network.

- Plug and Play: A HAVi-compliant appliance can configure itself and integrate itself into a HAVi network without user intervention. This can greatly simplify installation and setup.

- Upgradeable: HAVi devices may have only basic function, and new functionality can be automatically downloaded via the Internet. For instance, a HAVi Panasonic VCR can install the necessary application on a Sony TV in order to make two appliances interoperable.

Services in the HAVi are modeled as objects called software elements, accessible through their APIs (application programming interface). A software element is a basic unit in HAVi software framework. The design and implementation of software elements greatly affect the performance of home networking. Although the HAVi specification is available, there is little explanation about software element design.

The purpose of this project is to examine all issues that developers may encounter and to provide their solutions. The report may serve as a guide for consumer electronics manufacturers and application developers to design and implement software elements in HAVi.

1.5 Structure of Report

The report is divided into five chapters. Chapter One gives an introduction about home networking. The second chapter describes background about HAVi, and the third chapter describes the software element architecture. The fourth chapter explains our implementation. Chapter Five concludes the report.

Chapter 2

Home Audio/Video Interoperability

The HAVi Architecture specifies a set of Application Programming Interfaces (APIs) allowing consumer electronics manufacturers and third parties to develop applications for the home network. The primary goal of the HAVi Architecture is to assure that products from different vendors can interoperate, that is, can cooperate to perform application tasks.

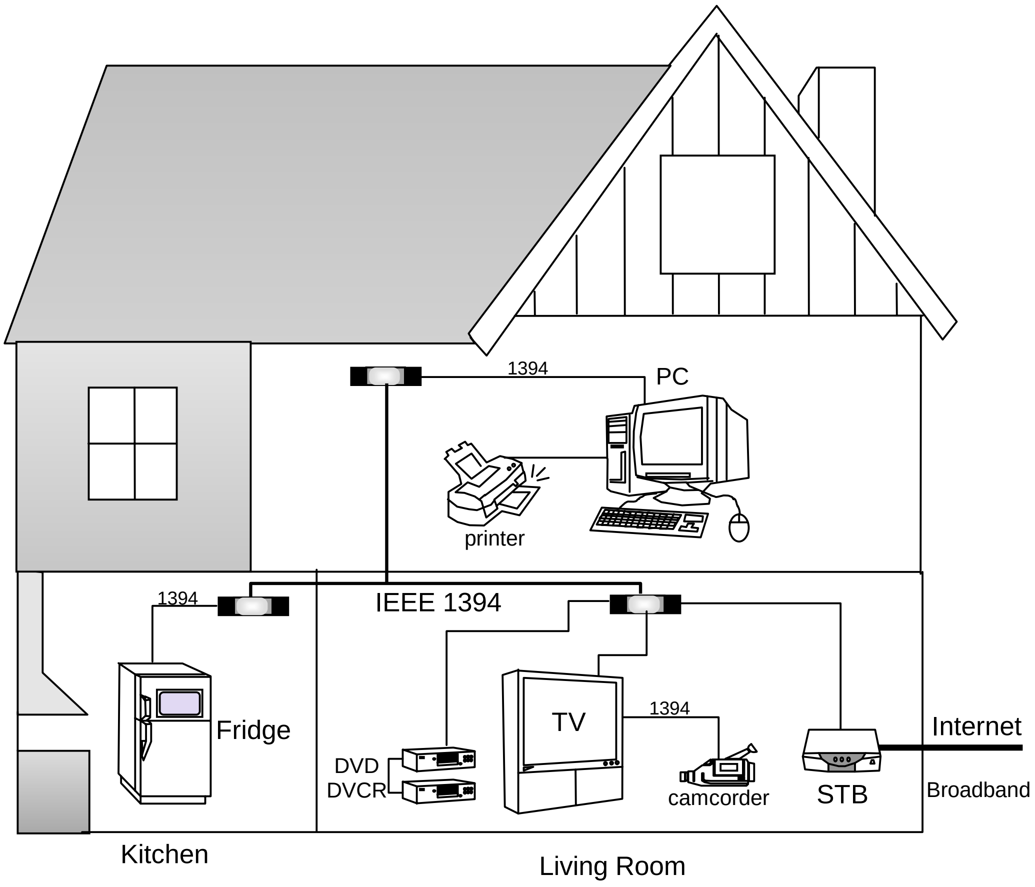


Figure 2-1: An example of a HAVi home network

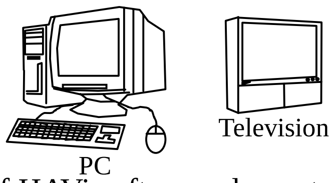
Typically, there will be several clusters of devices in the home, with one per floor or one per room. A HAVi home network is viewed as a distributed computing platform, and devices communicate in a peer-to-peer fashion, with no single master control device. An example of a HAVi home network is shown below:

2.1 Device Classification

HAVi classifies consumer electronics (CE) devices into four categories: Full AV devices (FAV), Intermediate AV devices (IAV), Base AV devices (BAV), and Legacy AV devices (LAV). HAVi-compliant devices are those in the first three categories. LAV is either a non-IEEE1394 device, or an IEEE 1394 device not supporting HAVi.

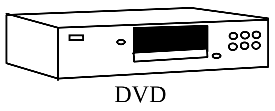
Usually FAV and IAV are controllers; BAV and LAV are controlled devices. A controller is a device that acts as a host for a controlled device. A controller is said to host a driver--Device Control Module (DCM) in terms of HAVi--for the controlled device. The control interface is exposed via the API of this DCM. This API is the only access point for applications to control the device. For instance, an intelligent television in the living room might be the controller for a VCR. An application in the TV could control the VCR via its DCM in the TV.

2.1.1 Full AV Devices



A Full AV device contains a complete set of HAVi software elements (see Figure 2-2). This device class generally has greater computing power and more resources; thus, it can support a complex software environment. The primary distinguishing feature of an FAV is the presence of a runtime environment for Java bytecode. This allows an FAV to upload bytecode from other devices and so provide enhanced capabilities for their control. Likely FAV devices would be Set Top Boxes, televisions, residential gateways, and home PCs.

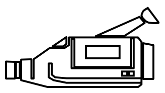
2.1.2 Intermediate AV Devices



An Intermediate AV device is generally lower in cost and has fewer resources than a FAV device. The main difference from FAVs is that IAVs have no Java runtime environment, so IAVs cannot control arbitrary devices within the home network.

Nevertheless, an IAV still can control particular devices if it has native-code DCMs for them. Possible IAV devices are DVD players and VCRs.

2.1.3 Base AV Devices


camcorder

BAV devices support IEEE 1394 but they cannot act as controllers in a HAVi home network. They can be controlled by an FAV or IAV device by providing uploadable Java bytecode (DCM) in their ROM. They do not host any software element of the HAVi Architecture. A FAV or IAV device and a BAV device communicate by the IEEE 1394 command protocol used by the BAV device. Likely BAV devices are audio players and camcorders.

Here is a possible scenario. A DVD player (BAV) contains Java bytecode that constructs a user interface for the device and allows external control of the device. When the DVD player connects to a television (FAV), the TV obtains the user interface and control code from the DVD player. An icon representing the device may then appear on the TV screen, and the device could be controlled via the TV.

2.1.4 Legacy AV Devices

LAV devices do not aware of the HAVi Architecture. The difference from BAVs is that LAVs may not support IEEE 1394 and do not provide uploadable control code. Hence, LAVs can work in a HAVi network only if an FAV or IAV recognize them and use proprietary protocols for their control. An FAV or IAV device and a LAV communicate by the legacy command protocol used by the LAV.

For example, a Sony VCD player, which does not support HAVi, connects with a Sony TV (FAV). If the TV recognizes the VCD player and has the DCM for it, users can control the VCD player via TV.

2.2 Software Element

As its name implies, a software element is the most basic unit in HAVi software architecture. In terms of object-oriented concept, a software element is a HAVi object.

Software elements provide services, which are accessible through the API. The diagram below is a possible arrangement of software elements on an FAV device.

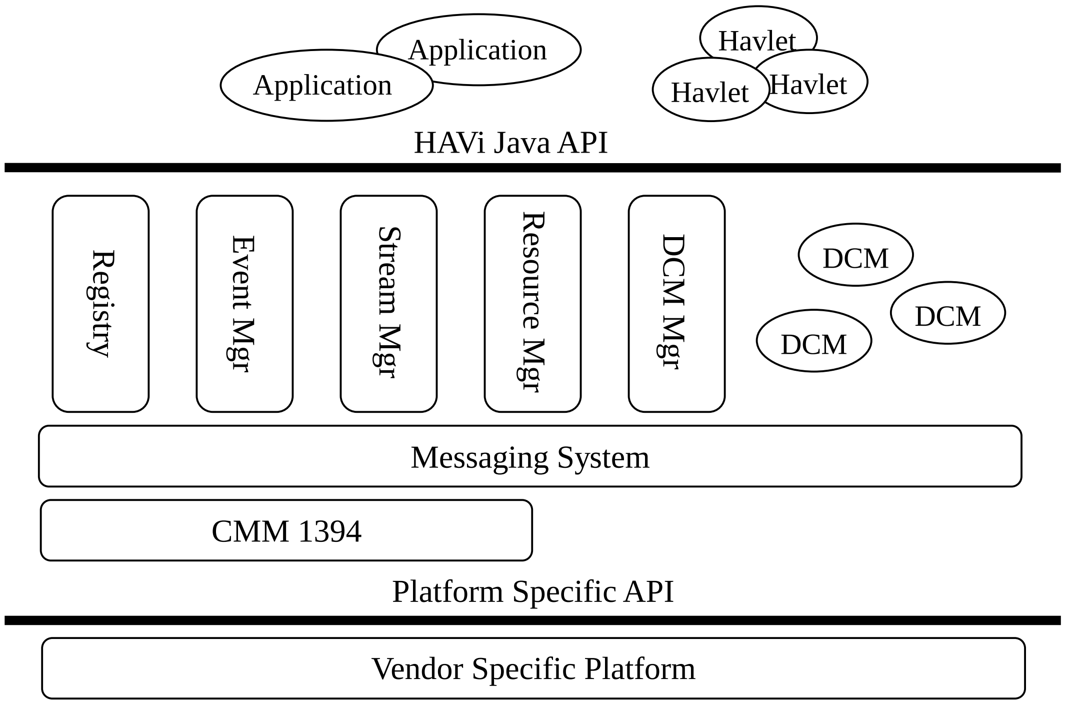


Figure 2-2: HAVi architecture diagram (FAV)

Here is the list of software elements in the HAVi Architecture and the services they provide.

- 1394 Communication Media Manager (CMM) – allows other software elements to perform asynchronous and isochronous communication over IEEE 1394.
- Messaging System – responsible for passing messages between software elements.
- Registry – serves as a directory service, allows any software element to locate another software element in the home network
- Event Manager – serves as an event delivery service. An event is the change in state of a software element or of the home network.
- Stream Manager – responsible for managing real-time transfer of AV and other media.

- Resource Manager – facilitates sharing of resources and scheduling of actions.
- Device Control Module (DCM) – a software element used to control a device.
- DCM Manager – responsible for installing and removing DCMs on FAV and IAV devices.
- Havlet – a HAVi Java application, offers interaction with uses via user interface

"System software element" or "system component" is defined as a software element providing basic system services. System software elements include CMM, Messaging System, Event Manager, Registry, DCM Manager, Stream Manager, and Resource Manager. Non-system software elements include Application, Havlet, and DCM.

The following table summarizes required and optional software elements for the four device classes.

Table 2-1: Software elements presented on various device classes

Software Element	FAV	IAV	BAV	LAV
Java Runtime	✓			
Application Module	[✓]	[✓]		
DDI Controller	[✓]	[✓]		
Resource Manager	✓	[✓]		
Stream Manager	✓	[✓]		
DCM Manager	✓	[✓]		
Registry	✓	✓		
Event Manager	✓	✓		
Messaging System	✓	✓		
1394 CMM	✓	✓		
SDD Data	✓	[✓]	✓	✓
DCM	✓	[✓]	✓	✓

✓ : required [✓]: optional

2.2.1 Software Element Identifier

Each software element has a software element identifier (SEID), which is an 80-bit value for identification and guaranteed to be unique. SEID are allocated by Messaging System when a software element initiates. The Messaging System of a device allocates SEIDs only for the software elements of that device. Software elements use SEIDs to be registered on the home network, and to communicate (via HAVi messages) with each other. For example, a software element wants to send HAVi messages to another software element. The SEID of the destination SE has to be specified when invoking the Messaging System API.

2.3 Messaging System

2.3.1 Description

HAVi network is message-based. All software elements communicate using messages via Messaging System, which is independent of the network and transport layers. This message passing mechanism abstracts from the details of physical location, that is, there is no distinction between a software element on the same device and one on a remote device. A Messaging System has mainly two jobs: forwards the messages from local software elements to other MS and dispatches the incoming messages from other MS to local software elements. A Messaging System is embedded in all FAV and IAV devices, but not in BAV or LAV devices. Thus, FAV (or IAV) devices communicate by HAVi messages via their Messaging System; FAV or IAV devices control BAV or LAV devices by proprietary protocols (not HAVi messages).

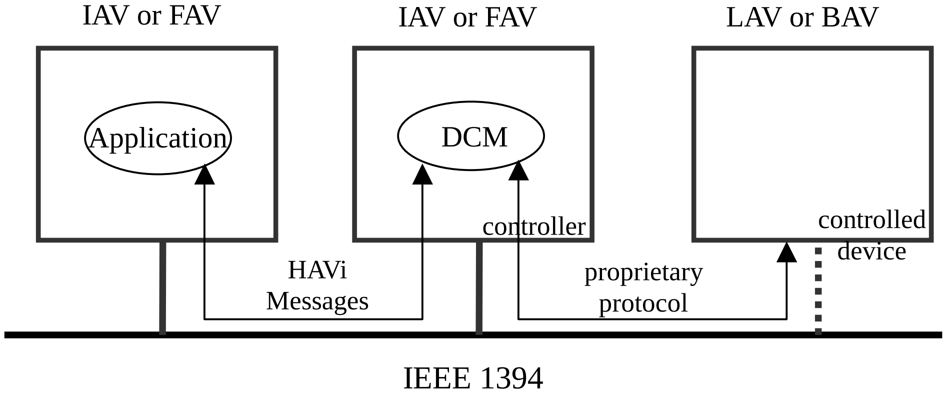


Figure 2-3: Communication among HAVi devices

To receive messages, a software element has to indicate callback function to the Messaging System. After Messaging System gets incoming messages, it will notify the destination software element via its callback function. The request will be handled in the callback function and corresponding actions will be executed. Callback function also called listener.

Here we define “callback thread”. When a Messaging System receives a message, the callback functions of the software element will be invoked. The thread that executes the callback function of the software element is called “callback thread”.

2.3.2 Message Transfer Modes

Messaging System provides two modes to send a message: simple and reliable mode. In this report, only reliable mode is discussed. In reliable mode, messages are sent synchronously or asynchronously.

The asynchronous mode is described through the following example. Suppose that there are two devices. Software element A and Messaging System 1 are on the same device, and software element B and MS 2 are on the other device. When A sends a message to B, the MS 1 sends the message to the MS 2. The MS 2 then tries to invoke the callback function of B. If the callback function successfully returns without errors, MS 2 sends an acknowledgment to MS 1. When MS 1 receives the acknowledgement, the Messaging System API invoked by A will return. The acknowledgement timeout is 30 seconds. That is, a caller software element always blocks until either the acknowledgement returns or a 30 seconds timeout occurs.

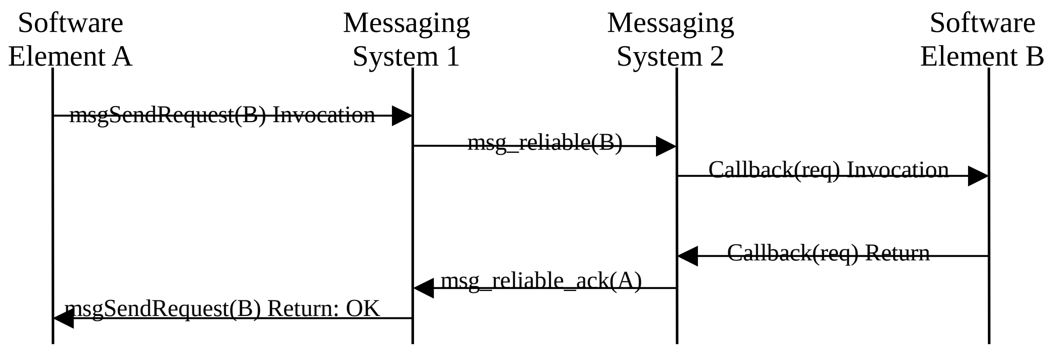


Figure 2-4: An example of asynchronous message transfer

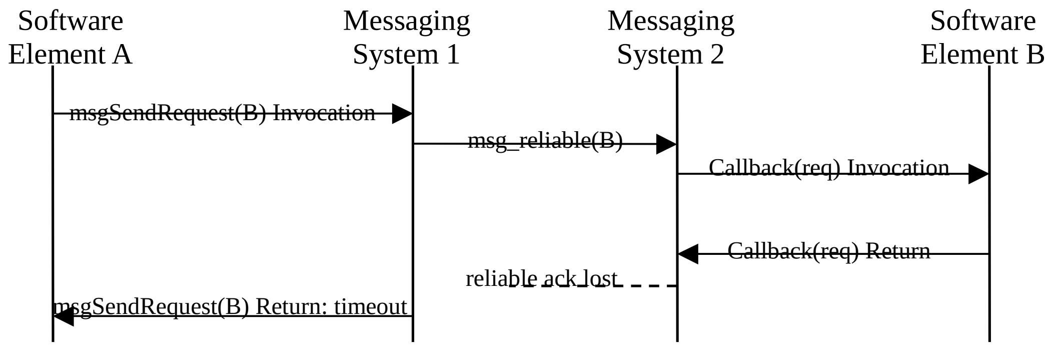


Figure 2-5: Messaging failing due to time expiration

Note that when a callback function successfully returns, it indicates the destination software element starts to process the request rather than finishes. When the response returns, all callback function installed by the caller will be invoked. Then, the caller uses the *transactionId*, an identifier is given while sending the message, to match the corresponding incoming responses in the callback function.

In synchronous mode, a caller blocks until the response is received (not the acknowledgment). A timeout parameter of the API indicates the maximum time that the caller will be blocked. As shown in the following figure, when the caller receives the acknowledgement, the caller keeps waiting until the response returns or a timeout occurs. Note that the timeout here is the parameter given in the API rather than the 30-second acknowledgement timeout in asynchronous mode.

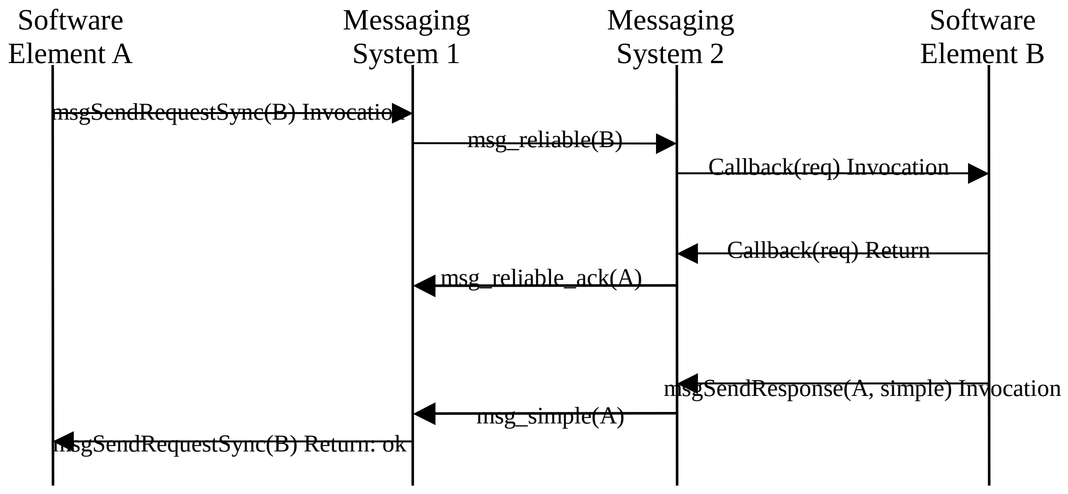


Figure 2-6: An example of synchronous message transfer

2.4 Registry

Registry serves as a software element directory, providing services for software elements to search other software elements in the network. Software elements that want to be contacted have to register with Registry. Registry maintains the SEID and the attributes for each registered software element.

Table 2-2: Services provided by Registry

API	Description
RegisterElement	Add a software element in the Registry
UnregisterElement	Remove a software element in the Registry
RetrieveAttributes	Read the attributes of the given software element
GetElement	Get software elements that satisfy the query parameter
MultipleGetElement	Get software elements that satisfy the query parameter

2.5 Java and HAVi

HAVi specifies a Java programming environment for applications and DCMs. The use of Java assures that applications and DCMs will run on any FAV device, the only device class that offers runtime environment for Java bytecode. This is an important feature because it allows third parties can also develop HAVi applications. Besides, it assures HAVi is compatible with future home appliances because applications and DCMs are upgradeable.

It is also possible to write HAVi applications in other languages, such as C or C++. They are native applications, which execute on a device-specific platform and therefore execute only on a particular HAVi device. Native applications can only be written and supplied by the vendor of the HAVi device or by someone who has specific knowledge of its platform. But anyone can write HAVi Java applications for FAVs, not just the manufacturer of FAVs.

A portable HAVi application can run on any FAV device from any manufacture, but it must be written in Java and confine itself to HAVi Java APIs, the set of Java class packages specified by HAVi.

2.5.1 The HAVi Java APIs

Software elements offer their services to other software elements via APIs. HAVi specifies APIs in IDL (Interface Definition Language) [14]. IDL is programming-language independent; it indicates the input and output parameters of interfaces, but not their implementation. HAVi provides a Java binding of IDL form APIs. Example 2-1 shows the IDL form of the *`GetElement`* service of a Registry software element and the Java representation is shown in Example 2-2.

Example 2-1: IDL form of an API

```
Status Registry::GetElement(in SimpleQuery query,
                            out sequence<SEID> seidList)
```

Example 2-2: Java binding for the API

```
package org.havi.system;
public class RegistryClient extends HaviClient {
    void getElement(IntHolder transactionId, SimpleQuery query);
    void getElementSync(int timeout, SimpleQuery query,
    SEIDSeqHolder seidList);
}
```

In the IDL form, the service has a *`SimpleQuery`* input parameter and a SEID array output parameter. In the Java representation, *`RegistryClient`* is a Java class whose methods correspond to the services of Registry software elements. Note that each service is mapped to two Java methods: asynchronous version and synchronous version. Asynchronous one has the same name, and the synchronous one has the name suffixed with *`"Sync"`*. Synchronous service methods have a timeout parameter. An exception will be thrown if the Messaging System does not receive a response before the specified timeout.

The set of packages defined by HAVi for developing in Java is called the HAVi Java APIs [15]. A portable HAVi application can only use the packages appearing in the HAVi Java APIs.

The HAVi Java APIs consists of the following packages:

- org.havi.constants
- org.havi.types
- org.havi.system
- org.havi.fcm.*
- org.havi.iec61883
- org.havi.ui
- org.havi.ui.event

In addition, the following packages can be used in a portable HAVi application:

- java.lang
- java.util
- java.util.zip
- a subset of java.io
- java.net.URL and java.net.MalformedURLException
- a subset of java.awt

2.5.2 The SoftwareElement Class

The *'SoftwareElement'* class is a class in HAVi Java APIs. In this report, "Software element" refers to the concept of HAVi object. Italic "*SoftwareElement*" refers to "an instance of *SoftwareElement* class".

For non-system software elements, like applications or DCMs, they create software elements representing themselves by constructing *SoftwareElement* objects. A new SEID will be created in the constructor. Applications use *SoftwareElement* objects to send HAVi messages to other software elements and use service of system components. In concept, a HAVi application is a software element; In Java implementation, a HAVi application owns a *SoftwareElement* object. A HAVi Java application example is shown in Example 2-5.

As for system software elements, their design is implementation dependent. They may implement in Java and use the *SoftwareElement* class the way as an application does, or they may implement in any way as long as they provide the services conforming to the HAVi specification.

The following is the definition for the `SoftwareElement` class.

Example 2-3: The SoftwareElement class

```
public class SoftwareElement {
    public SoftwareElement(HaviListener hl);
    public SoftwareElement();
    public final void close();
    public final boolean msgIsTrusted(SEID seid);
    public final SEID msgGetSystemSeid(SEID seid, int softwareElementType);
    public final void msgWatchOn(SEID destSeid);
    public final void msgWatchOff(SEID destSeid);
    public final void msgSendSimple(byte protocol,
    SEID[] destSeidList,
    HaviByteArrayOutputStream buffer);
    public final void msgSendReliable(byte protocol,
                                      SEID destSeid,
                                      HaviByteArrayOutputStream buffer);
    public final void msgSendRequest(SEID destSeid,
                                     OperationCode opCode,
                                     HaviByteArrayOutputStream buffer,
                                     IntHolder transactionId);
    public final void msgSendResponse(SEID destSeid,
                                      OperationCode opCode,
                                      int transferMode,
                                      Status returnCode,
                                      HaviByteArrayOutputStream buffer,
                                      int transactionId);
    public final void msgSendRequestSync(SEID destSeid,
                                         OperationCode opCode,
                                         int timeout,
                                         HaviByteArrayOutputStream bufferIn,
                                         HaviByteArrayInputStream bufferOut,
                                         StatusHolder returnCode);
    public final SEID getSeid();
    public final void addHaviListener(HaviListener hl);
    public final void addHaviListener(HaviListener hl, SEID targetSeid);
    public final void removeHaviListener(HaviListener hl);
}
```

```
public SoftwareElement (HaviListener hl);
```

The `hl` parameter in the constructor is an instance of a class that extends *`HaviListener`*. A *`HaviListener`* object is installed for a software element either via the *`SoftwareElement`* constructor or via the `addHaviListener()` method. HAVi listeners determine how a software element handles incoming messages. Received messages are delivered to all *`HaviListener`* objects installed for the associated *`SoftwareElement`* object.

```
public final void addHaviListener(HaviListener hl);
```

This method adds a *`HaviListener`* object to the list of listeners maintained by this *`SoftwareElement`* object. When a message arrives for this *`SoftwareElement`*, the *`SoftwareElement`* will call each of the listener's *`receiveMsg()`* method, in no particular order.

```
public final void msgSendRequestSync(...);
public final void msgSendRequest(...);
public final void msgSendResponse(...);
```

A software element can use these three methods to send HAVi messages to other software elements.

2.5.3 The HaviListener Class

An object should extend *`HaviListener`* class and implement the abstract *`receiveMsg()`* method if it hopes to listen to incoming messages and responses to a specific *`SoftwareElement`*.

The HAVi Java APIs define *`HaviListener`* as:

```
public abstract class HaviListner {
    public abstract boolean receiveMsg(
        Boolean haveReplied,
        byte protocolType,
        SEID sourceId,
        SEID destId,
        Status state,
        HaviByteArrayInputStream payload);
}
```

receiveMsg() returns true only if it receives an request that it can handle. The *haveReplied* parameter shows whether any other listener has replied to the incoming message. The *protocolType* parameter indicates whether the message uses the HAVi RMI (Remote Method Invocation) protocol or some private (application-specific) protocol. The state parameter shows error condition. *sourceId* indicates the SEID of the sender and *destId* indicates the SEID of the receiver. *payload* is the message payload from which request parameters can be retrieved.

2.5.4 The HaviClient Class

The *HaviClient* class is extended by all classes that allow an application to access the services of system components. For instance, *RegistryClient* extends *HaviClient* and provides access to Registry services (see Example 2-4). The first parameter in the constructor *RegistryClient()* is a *SoftwareElement* object that will be a client of Registry services; this is typically a software element created by an application. The second parameter identifies the GUID of the device.

In most cases, software elements use the system components in the same device rather than remote ones. '*RegistryLocalClient*', the local client class of the Registry, provides the service of the local Registry running on the same device with the client

Example 2-4: The HaviClient class and the RegistryClient class

```
public abstract class HaviClient {
    HaviClient(SoftwareElement se, SEID destSeid);
}
public class RegistryClient extends HaviClient {
    RegistryClient(SoftwareElement se, GUID destGUID);
    // other Registry APIs
    void getElement(IntHolder transactionId, SimpleQuery query);
    ...
}
public class RegistryLocalClient extends RegistryClient {
    RegistryLocalClient(SoftwareElement se);
}
```

2.5.5 An Example

In Example 2-5, a software element will be created to represent this application, which offers a "Hello World" Service.

In line 16, a software element is initiated in the constructor and "*`this`*", a HaviListener, is passed as a parameter. In lines 31, *`receiveMsg()`* handles incoming messages to the software element. The HAVi message payload contains three fields: operation code, control flag, and transaction identifier. These fields are retrieved in lines 42-44. Operation code specifies the service being requested. Control flags indicate whether the incoming HAVi message is a request or response. Transaction identifier is an integer allowing the destination software element to match response with requests. In lines 49, it does a switch on the operation code of the requested service. If it is a "Hello World Service" request, a "Hello World" string will be printed and a response will be sent via the *`msgSendResponse()`* API of the software element (*`mySe).`*

Example 2-5: A Simple HAVi Java application

```
1:  package lab441.havi.demo.thesis;
2:  import ntu.havi.constants.*;
3:  import ntu.havi.system.*;
4:  import ntu.havi.types.*;
5:
6:  public class SimpleApplication extends HaviListener {
    private SoftwareElement mySe = null;
    // operation code for “Hello World Service”
    private final byte HelloWorldId = (byte)0x80;
    private OperationCode opCode = null;
    private byte controlFlags = 0;
    private int transactionId = 0;
    private Status returnCode = null;

    public SimpleApplication() throws Exception {
        mySe = new SoftwareElement(this);
        // register this application
        RegistryLocalClient registry = new
                                       RegistryLocalClient(mySe);
        Attribute[] att = new Attribute[0];
        HaviByteArrayOutputStream hbaos = new
                HaviByteArrayOutputStream();
        hbaos.reset();
        hbaos.writeHaviString("Hello World");
        att[0] = new Attribute(
                ConstAttributeName.ATT_DEVICE_MANUF, hbaos);
        registry.registerElementSync(3000, mySe.getSeid(),
                                     att);
    }

    public boolean receiveMsg(
            boolean haveReplied,
            byte protocolType,
            SEID sourceId,
            SEID destId,
            Status state,
            HaviByteArrayInputStream payload) {
          if(haveReplied)
            return false; // another listener has replied
        // retrieve operation code from the HAVi message header
        try {
            opCode = new OperationCode(payload);
            controlFlags = payload.readByte();
            transactionId = payload.readInt();
            if((controlFlags & 0x01) == 1) {
                // incoming message is a response, not a resquest
                return false; // ignore
            }
            switch(opCode.getOperationId()) {
                case HelloWorldId:
                    System.out.println("Hello World");
                    returnCode = new Status(opCode.getApiCode(),
                            ConstGeneralErrorCode.SUCCESS);
                    mySe.msgSendResponse(sourceId, //?
                            opCode,
                            ConstTransferMode.SIMPLE,
                            returnCode,
                            null,
                            transactionId);
                return true;
            default: return false;  // unknown operation id
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    } // end of receiveMsg()
} // end of class MyHAViApplication
```

Chapter 3

Design Methodology of Software Element

3.1 Pre-assumption of Messaging System

It is critical that software elements carefully use the service of Messaging System and Messaging System efficiently dispatches messages to software elements. The design of software element is strongly related to that of Messaging System (MS), especially how MS dispatch messages and invoke callback functions.

According to HAVi specification, while a software element is performing its callback function, it does not block other software elements or the underlying Messaging System. Besides, during the time that the callback function blocks, the software element may not be able to process other incoming messages

3.2 Design Issues

The basis of a HAVi network is that requests are sent from software element to software element, actions are taken, and corresponding responses are returned. How a software element handles these requests, actions, and responses is largely up to the developers. The design of software elements strongly affects the efficiency of HAVi system.

We encountered some design issues about software elements while we designed and implemented our HAVi stack. We concluded them into four issues.

3.2.1 Blocking Time

When a software element sends an asynchronous request, it will block until the acknowledgement returns or a timeout occurs. The software element that receives a callback invocation has to immediately return from the callback. However, the callback function may take long to complete, and such a design may result in poor performance. For example, when a Registry gets a query, the Registry has to forward the query to every other Registry in the network and collects the responses. It may take much time, and the software element that sends the query may be blocked for a long time.

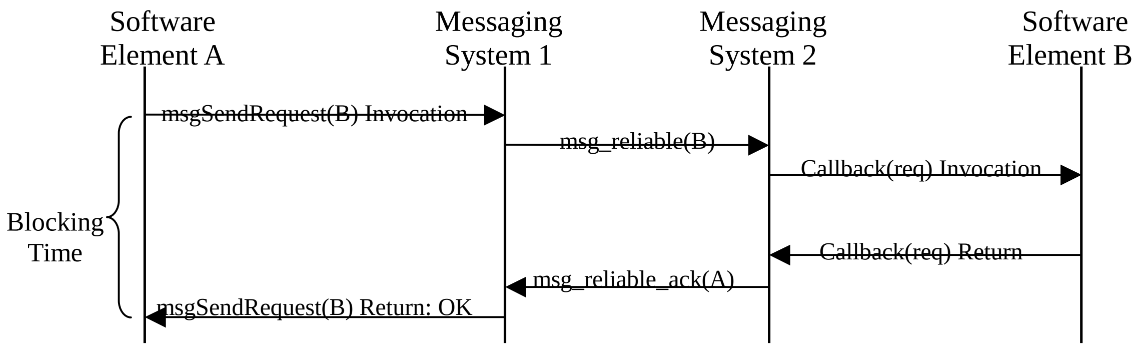


Figure 3-1: Blocking time

3.2.2 Multiple Requests

When a software element receives a message, it will trigger various actions. If the actions are implemented in a way that blocks the software element that receives new messages, no new message will be processed until the previous action completes. In the case that a software element receives multiple requests at the same time, many of them may end with timeouts. For example, when a Registry gets a query, it has to forward a query to every other Registry in the network. Suppose that a Registry gets several queries at the same time and it handle the queries one by one. While the first query is being processed, all other queries will be queued. It is inefficient and may cause many timeouts.

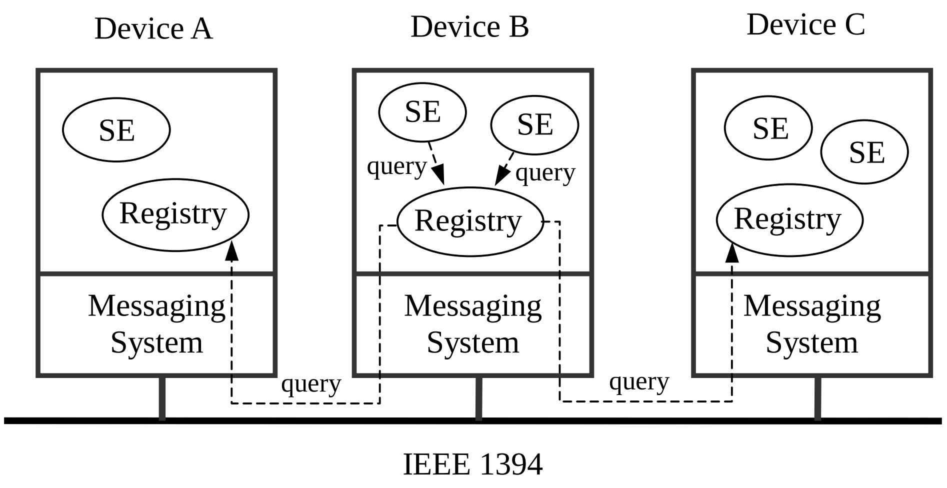


Figure 3-2: A Registry forwards the query to every other Registry

3.2.3 Synchronization

Suppose that a software element is implemented in a way that it handles one request at a time. If a callback function includes a request to another software element, a deadlock may occur. For example, when two Registries simultaneously query each other, but cannot receive the responses, a deadlock may occur because they cannot complete their actions until the queries from other Registries are answered.

For example, in the case that a Registry is implemented in such a way that it uses synchronous request within its callback function to forward GetElement() requests to other registries. When two applications on two different devices, A1 and A2, happen to query their Registries, R1 and R2, at approximately the same time, a deadlock may occur. R1 is busy handling the request of A1 and waits for the response to the GetElement() forwarded to R2. During this period, R1 cannot process incoming messages. Furthermore, R2 is busy handling the request of A2 and waits for the response to the GetElement() forwarded to R1. Also, during this period, R2 cannot process incoming messages. This deadlock situation will result in timeouts of the synchronous call in both R1 and R2.

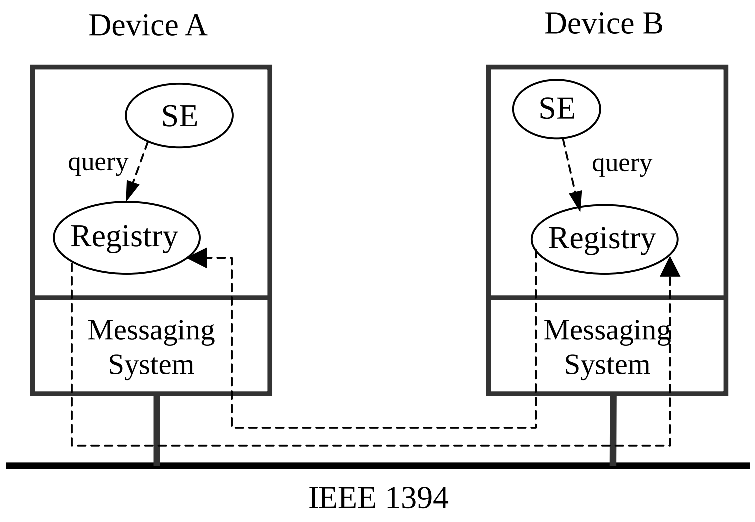


Figure 3-3: Two Registries simultaneously query each other

Suppose that a software element can handle multiple requests at the same time. The software element is handling two requests simultaneously. If the two requests may change the same data in the software element, the data should be protected by synchronization mechanism. For example, a Registry is processing two requests, RegisterElement() and UnregiserElement(), at the same time. Since RegisterElement() will add an entry to the Registry database and UnregisterElemen() will delete an entry, an error may occur if there is no synchronization mechanism.

There is a more complicated situation. When a software element receives a new request, it should stop the former requests and process only the new request. For instance, a network reset event is typically generated when the network topology changes or a device is activated or deactivated. Whenever a DCM Manager receives a network reset event, it should start the leader selection protocol, which may take a long time to complete. If the DCM Manager receives another network reset event, the leader selection protocol should be restarted, that is, the new callback thread should notify the thread executing protocol to stop.

3.2.4 Multicasting

Multicasting is a delivery of information to multiple destinations simultaneously. In this report, multicasting refers to sending HAVi messages to multiple software elements simultaneously.

A software element may have to send messages to more than one software elements and wait for responses. The number of destination software elements could be large. For example, GetElement() method of Registry is used to get a list of software element identifiers that satisfy the query given through the parameter. When a Registry receives a query from a local software element, the Registry has to forward the query to all other Registries in the network and collect the responses. In the Figure 3-4, it shows that a Registry receives a request from a software element, and then forwards the query to all other Registries in the network.

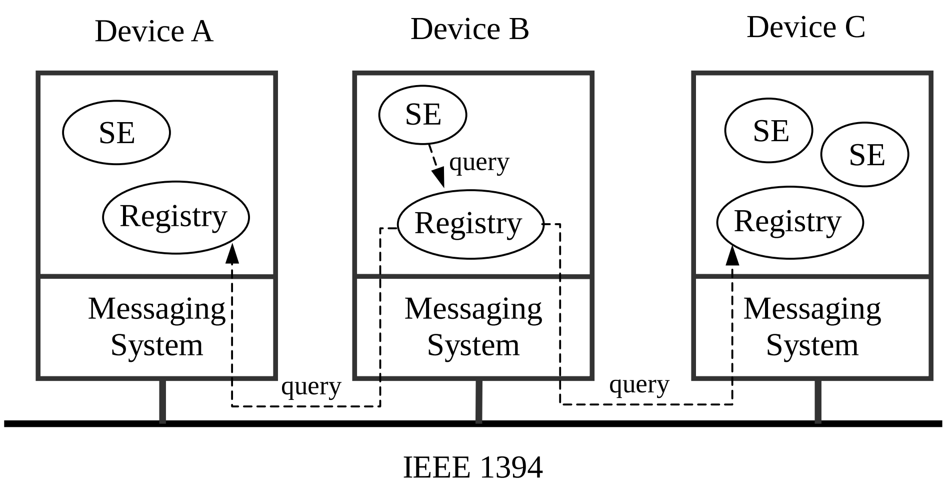


Figure 3-4: A Registry forwards the query to all other Registries

It is common for a software element to send messages to various software elements at the same time. It seems easy, but if we study this problem in depth, every solution brings more problems.

Table 3-1: Situation when a software element sends multiple HAVi messages

Software Element	API	Description
DCM Manager	DMinitilization DMInquiry DMCommand	Query other DCM Managers
	DMInitialInquiry	Notify other DCM Managers
Event Manager(EM)	PostEvent ForwardEvent	Forward event to other EMs
Registry	GetElement MultipleGetElement	Query other Registries
StreamManager	GetGlobalConnectionMap	Query other StreamManagers
ResourceManager	Reserve	Reserve a number of FCMs.
	ScheduleAction	Bandwidth checking protocol
	GetScheduledConnections	Get a list of connections
Any application		

The simplest solution is that the software element sends the synchronous messages in turn. At first, the SE synchronously sends the message to the first target, and the software element will block until it receives the response. Then, it sends the message to next destination SE. After the software element send the messages to all destination SEs, it finishes. Apparently, this method takes too much time and it is inefficient.

Another solution is to use additional threads. The software element creates several worker threads to send synchronous requests. The term “worker threads” here refers to the threads created to help the original thread to do some jobs. Every worker thread sends a message to a destination software element. However, this may not work because if the SE waits the response in the callback function, the SE may not be able to process other incoming messages. Also, a small number of threads are acceptable for simple situations, but multiple threads may be costly, and it may be difficult to predict the maximum number of threads needed. How the worker threads communicate with each other is another problem.

The third solution is to send requests asynchronously. The software element sends asynchronous messages to different destination software elements in turn. The SE does not block while sending. Instead, the callback function will be invoked automatically. Note that the software element shall not wait responses in the callback function since it may not handle other messages. When making an asynchronous request, a software element must store the transaction ID and possible other information about the original request in a sort of table. Normally the entries are removed every time a matching response is received. In the case that a matching response is not received, the corresponding request information will not be automatically removed from the table, causing a potential memory leak. Worse yet, if the SE never receives the response, it may not be able to complete its action, causing parts of the HAVi system to freeze. There are many reasons that may cause responses not to be received. One reason may be that the destination software element, or destination device, is removed during a request. In these cases a timeout will occur, completing the request.

To avoid these problems, software elements that implement asynchronous requests should also implement a timeout mechanism. The timeout mechanism would ensure that actions are completed, and that request data does not build up in tables. Unfortunately, the timeout may not occur until after a long delay, resulting in very long response times.

Overall, software element architecture should satisfy the requirements in the above issues and has an efficient and simple design.

3.3 Design for Blocking Time

A calling software element sending asynchronous message is blocked until the callback function of the target software element returns. To avoid blocking, the callback function should create a thread to handle the message and then return as soon as possible.

Example 3-1: Creating a thread to handle messages in the callback function

```
class Application extends HaviListener {
    MessageHandler msgHandler = null;
    public boolean receiveMsg(boolean haveReplied,
                              byte protocolType,
                              SEID sourceId,
                              SEID destId,
                              Status state,
                            HaviByteArrayInputStream payload) {
       // create another thread to handle the message
        // pass the parameters to the message handler
        msgHandler = new MessageHandler(haveReplied,
                    protocolType, sourceId, destId, state, payload);
        msgHandler.start(); // start the thread
        return true; // the callback thread can return quickly
    }
}
class MessageHandler extends Thread {
    void run() {
        // handle the message
    }
}
```

In order to reduce the blocking time more, a software element should activate a watch on the target before sending a request. Messaging System provides an msgWatchOn() API for a software element to be notified if the target is removed. Another option would be to register for GoneDevices and GoneSoftwareElement events. When being notified the target disappears, a software element can respond more quickly.

3.4 Design for Multiple Requests

In order to handle multiple requests at a time, multiple threads are necessary. Adopting the solution in 3.3 , software elements can handle multiple requests at the same time. However, initiating a new thread every time is time-consuming. If request for the software element are time-critical, the software elements could implement thread pooling to reduce response time.

Example 3-2: A software element with three threads

```
class Application extends HaviListener {
    // this software element has three threads
    Worker[] workers = new Worker[3];
    public boolean receiveMsg(...) {
```

```
5:          if (any of the three threads is available) {
6:              // resume the available thread to handle the message
7:          }
8:          else { // new another thread to do the job
9:              msgHandler = new Worker(...);
10:             msgHandler.start();
11:         }
12:         return true;
13:     }
14: }
15: class Worker extends Thread {
16:     ...
17: }
18:
```

This software element owns three threads. In line 5, if any of the three threads is available, just resume it to process the message. Or else another thread is created to handle the message because the callback function should return as soon as possible (line 9).

3.5 Design for Synchronization

Adopting two solutions above, software elements can process multiple requests at the same time. The deadlock condition mentioned in 3.2.3 is unlikely to happen since the callback function always returns in a short time. However, different worker threads may manipulate variables of a software element simultaneously, which may introduce errors. To avoid this, we can use Java synchronized data structure such as Vector and Hashtable [16], or use Java build-in synchronization mechanism (synchronized method modifier and synchronized statement block).

There is another issue: while a worker thread is processing a request, a new incoming request may have to interrupt it. Java built-in language synchronization can be used in this situation. When a Thread object's interrupt() method is invoked, a *`InterruptedException`* will be thrown. If a worker thread can be carefully implemented, it can be interrupted just by invoking its interrupt() method.

Example 3-3: An example of interrupting a thread

```
class Application extends HaviListener {
    MessageHandler wt;
    public boolean receiveMsg(...) {
        if(under necessary condition) {
            wt.interrupt(); // interrupt the previous thread
        }
        return true;
    }
}
class MessageHandler extends Thread {
    void run() {
        try {
            // process the incoming message
            if(Thread.interrupted()) // check if interrupted
                throw new InterruptedException();
            // continue to process the message
        } catch(InterruptedException e) {
            return;
        }
    }
}
```

3.6 Design for Multicasting

In this section, a model is proposed to send messages to multiple software elements efficiently. This model may be implemented in synchronous or asynchronous messaging. Besides, an API, *`msgSendMultipleRequest()`*, is added to the *`SoftwareElement`* class for developers.

3.6.1 Synchronous Multicasting

In the synchronous message mode, a caller will block until a response is received. Since sending synchronous messages in turn is not efficient, the only way is to use additional worker threads to send synchronous requests.

In the beginning, the callback thread creates another thread to handle the message so that the callback thread can return immediately. Then, the message handler thread creates worker threads to send messages: one thread for one target software element. While the worker threads are sending synchronous messages, the message handler thread checks if all responses are returned unless a timeout condition occur.

Example 3-4: The design in synchronous multicasting

```
public class Application extends HaviListener {
    public void multicasting(...) {
        for(int i = 0; i < num_of_target; i++) {
            new WorkerThread(ith request, ...).start();
        }
        while(not timeout && not all acknowledgement received) {
            sleep(1000);
        }
    }
    class Worker extends Thread {
        void run() {
            // send the request
            mySe.msgSendRequestSync(...);
        }
    }
}
```

However, multiple threads could carry lots of resource overhead. The number of destination software elements is unpredictable; it could be just one or a large number like 50. Each thread requires memory resources and processor resources. Besides, there is also work involved in starting a thread. Thread pooling can be considered.

3.6.2 Asynchronous Multicasting

In this design, only one thread is required. At first, the message handler thread sends asynchronous messages to all targets in turn. New entries, including the transaction ID and other information about the request, are stored in a table. The thread waits all the responses return until a timeout occurs. Even if no responses return (e.g. a network failure), the thread can continue the work, avoiding the problem that the part of the software elements freezes. When the thread is waiting, the callback function may receive responses and store them in the table. The thread stops waiting if all the responses return or a timeout occurs, and then the entries in the table are removed, so there will be no memory leak.

Example 3-5: Using only one thread to send messages to all target software elements

```
class Application extends HaviListener {
    Hashtable table;
    public receiveMsg(...) {
        // store the responses in the table
    }
    void multicasting(...) {
        for(int i = 0; i < num_of_target; i++) {
            mySe.msgSendRequest(...);
        }
        while(not timeout && not all responses have received) {
            // keep waiting
        }
    }
}
```

3.6.3 Comparison

In network programming, I/O can be classified as synchronous I/O or asynchronous I/O. "*`msgSendRequestSync()`*" acts as synchronous network I/O; the caller blocks until the response returns. "*`msgSendRequest()`*" acts as asynchronous network I/O; the caller does not block and wait for the response. In network programming, asynchronous network I/O is generally more efficient than synchronous network I/O. However, for the multicasting issue in HAVi, we think the synchronous design is better for two reasons.

First, the synchronous design in 3.6.1 is faster than the asynchronous design in 3.6.2 . In asynchronous mode, the caller has to wait until the acknowledgement returns, which may take some time. Since the caller sends asynchronous messages in turn, the delay could accumulate to a long time. In synchronous mode, every thread is responsible for one target and they are waiting concurrently. As long as the thread pooling is used to eliminate the thread initialization time, the synchronous design is faster than the asynchronous design.

Second, the synchronous design is straightforward and easier to implement. The most convenient way for application developers is to implement the multicasting in a single method. In the asynchronous design, the responses will be returned in the callback function, so the multicasting cannot be completed in a single method. And the developers have to implement a request/response table in the application to pass the responses between the threads. This makes the implementation more complicated. As for the synchronous design, all the implementation can be put in a single method. The threads can be provided by the HAVi system. All the application developers have to do is invoke the method.

3.6.4 A New API: msgSendMultipleRequest()

Since many software elements need to send messages to multiple targets, the design should be implemented in a single method of the SoftwareElement class. The function prototype is similar to msgSendRequestSync(), making it easier to use.

Example 3-6: The function prototype of the msgSendMultipleRequest() API

```
public class SoftwareElement {
public void msgSendMultipleRequest(
    SEID[] destSeid,
    OperationCode[] opCode,
    int timeout,
    HaviByteArrayOutputStream[] bufferIn,
    HaviByteArrayInputStream[] bufferOut,
    StatusHolder[] returnCode);
}
```

Chapter 4

Implementation

4.1 Environment

4.1.1 HAVi stack

We implemented a HAVi stack. For platform independence, an abstraction layer was implemented to deal with Linux specific libraries and Java Native Interface [17]. The abstraction layer was written in C and Java. The rest, the system components and the applications, were all written in Java. Thus, our HAVi stack could be easily ported to other platform with a new abstraction layer. Since all system components were written in Java, this HAVi stack is aimed at FAV devices because IAV have no Java runtime environment.

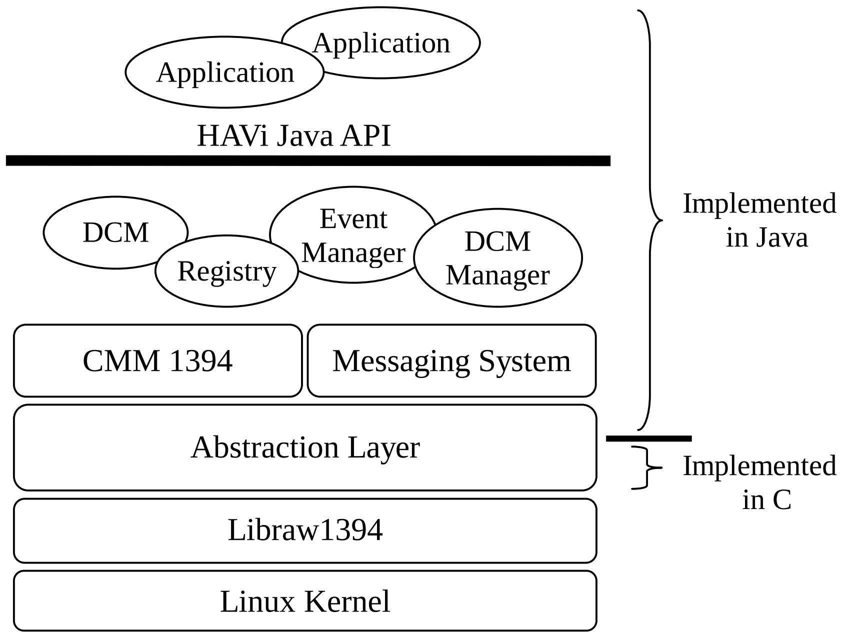


Figure 4-1: Our HAVi stack

We implemented all system components except Stream Manager and Resource Manager. The following is the description of our HAVi stack. The abstraction Layer is platform dependent, providing IEEE 1394 service for MS and CMM, using Libraw1394 [18], a Linux C library, to access IEEE 1394 bus. Our CMM, MS, Registry, and Event Manager were mostly implemented as the HAVi specification defines. As for DCM Manager, only the DCM installation service was implemented. We implemented a DCM for the DV and implemented an application to control the DV.

4.1.2 Demonstration

Two personal computers and a DV (Digital Video Camcorder) were used to build a HAVi home network. One computer was simulated as a HAVi set-top box (FAV); the other simulated a HAVi TV. The DV represented a LAV. The computers were running on Linux and equipped with IEEE 1394 FireWire cards.

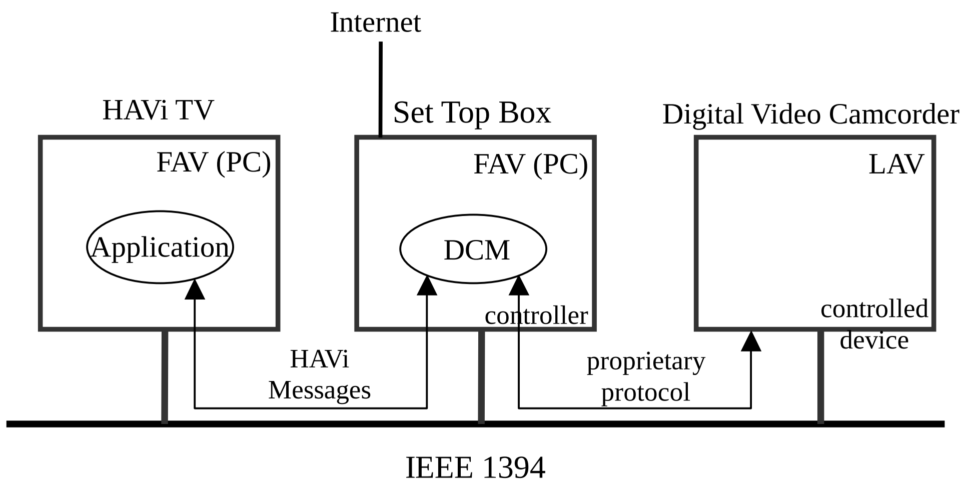


Figure 4-2: A HAVi network composed of two computers and a DV

Our HAVi stack was installed in both computers. The demonstration worked as the following. When the DV was plugged into the network, the set-top box detected it and downloaded the DCM for the DV. Then, the HAVi TV could control the DV via the DCM. The images were transferred isochronously from the DV and showed on the HAVi TV.

The DV acted as a LAV device, so no DCM was embedded in it. We assumed the set-top box and the DV were from the same device maker. When the DV was plugged into the network, the set-top box would receive a NEW_DEVICE event, recognize the device, and download the DCM from the Internet. The DCM Manager installed the DCM and then the DCM registered itself in the Registry. The set-top box communicated with the DV via Libavc1394 [19], a Linux C library for the AV/C (Audio/Video Control) Digital Interface Command Set [20]. The application in the HAVi TV queried the Registry for DV and obtained the SEID of the DCM. Users could control the DV via the UI of the application. Functions included play, stop, forward, and reverse.

4.1.3 Messaging System

The Messaging System was implemented in Thread-Per-Message with thread pooling. The pool had five threads, meaning five callback functions could be invoked at once. We did not choose Thread-Per-SoftwareElement because it may create too many threads. And the threads may just wait there and do nothing if no requests received.

4.2 Implementation of msgSendMultipleRequest()

The implementation of *`msgSendMultipleRequest()`* method is shown in Example 4-1. *`msgSendMultipleRequest()`* is used to send HAVi messages to multiple software elements, and this method can be added to *`SoftwareElement`* class. A *`Worker`* class is used to send to a message.

Example 4-1: Implementation of msgSendMultipleRequest()

```
public class SoftwareElement extends HaviListener {
     public void msgSendMultipleRequest(
               SEID[] destSeid,
               OperationCode[] opCode,
               int timeout,
            HaviByteArrayOutputStream[] bufferIn,
            HaviByteArrayInputStream[] bufferOut,
            StatusHolder[] returnCode) {
        int n = destSeid.length;
        for(int i = 0; i < n; i++) {
            new Worker(mySe, destSeid[i], opCode[i], timeout,
            bufferIn[i], bufferOut[i], returnCode[i]).start();
        }
        long startTime = System.currentTimeMillis();
        boolean allReceivedFlag = false;
        while((!allReceivedFlag) &&
              ((System.currentTimeMillis()-startTime) < timeout)) {
            allReceivedFlag = true;
            // to determine if having received all responses
            for(int i = 0; i < n; i++) {
                if(returnCode[i].getValue() == null) {
                    allReceivedFlag = false;
                    break;
                }
            } // for
            if(!allReceivedFlag) {
                try {
                    Thread.currentThread().sleep(1000);
                } catch(InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    } // end of method
    class Worker extends Thread {
        private SoftwareElement mySe;
        private SEID destSeid;
        private OperationCode opCode;
        private int timeout;
        private HaviByteArrayOutputStream bufferIn;
        private HaviByteArrayInputStream bufferOut;
        private StatusHolder returnCode;
        Worker(SoftwareElement mySe,
44:                   SEID destSeid,
45:                   OperationCode opCode,
46:                   int timeout,
47:                   HaviByteArrayOutputStream bufferIn,
48:                   HaviByteArrayInputStream bufferOut,
49:                   StatusHolder returnCode) {
50:                 this.mySe = mySe;
51:                 this.destSeid = destSeid;
52:                 this.opCode = opCode;
53:                 this.timeout = timeout;
54:                 this.bufferIn = bufferIn;
55:                 this.bufferOut = bufferOut;
56:                 this.returnCode = returnCode;
57:             }
58:             public void run() {
59:                 try {
60:                     mySe.msgSendRequestSync(destSeid, opCode, timeout,
61:                                             bufferIn, bufferOut, returnCode);
62:                 } catch(Exception e) {
63:                     e.printStackTrace();
64:                 }
65:             }
66:         }
67:     }
```

The parameters (lines 3-8) are similar to those in *`msgSendRequestSync()`* except they are arrays. Worker threads are created or obtained to send the messages (line 11). The while loop (lines 16-17) checks if a timeout occurs or all responses have been received. If not, the current thread sleeps for 1000ms (line 28). The *`Worker`* class (line 35) is used to send a message using *`msgSendRequestSync()`* (line 60).

4.3 An Example: A HAVi Application

The Application class, shown in Example 4-2, implements the software element architecture proposed in Chapter 3. The application has two services: to print a string "Hello World" in the console or to perform addition. If the application receives a request, a worker thread is obtained to handle the request and the callback function can return quickly. The worker thread will handle the request and send the response.

This application has the following advantages. First, the callback function always returns quickly, so the caller software element does not block. Second, the application can handle multiple requests simultaneously. Third, there is no deadlock. Fourth, the interrupt() method in the ThreadPool class can interrupt all worker threads gracefully. That is, the request being processed can be cancelled.

Example 4-2: Application.java--an application that implements the design methodology

```
import java.util.*;
import ntu.havi.constants.*;
import ntu.havi.system.*;
import ntu.havi.types.*;

interface ConstApplication {
    final short apiCode = (short)0x8000;
    final byte HelloWorldId = (byte)0x80; // print "Hello World"
    final byte AdditionId = (byte)0x81; // perform addition
}

public class Application extends HaviListener {
    private SoftwareElement mySe;
    private ThreadPool pool;

    public Application () throws Exception {
        mySe = new SoftwareElement(this);
        // create a thread pool with three threads
        pool = new ThreadPool(3);
        // register this application
        RegistryLocalClient registry = new
                                      RegistryLocalClient(mySe);
        Attribute[] att = new Attribute[0];
        HaviByteArrayOutputStream hbaos = new
                HaviByteArrayOutputStream();
        hbaos.reset();
        hbaos.writeHaviString("Hello World");
        att[0] = new Attribute(
                ConstAttributeName.ATT_DEVICE_MANUF, hbaos);
        registry.registerElementSync(3000, mySe.getSeid(), att);
    }
    public void receiveMsg(byte protocolType,
                          SEID sourceId,
                          SEID destId,
                          Status state,
                          HaviByteArrayInputStream payload) {
        boolean haveReplied = false; //

        OperationCode opCode;
        byte controlFlags;
        int transactionId;
        if(haveReplied) {
            return false;
        }
        if(state.getErrorCode() != ConstGeneralErrorCode.SUCCESS) {
            // Messaging System problem, ignore
            return false;
        }
        if(protocolType != ConstProtocolType.HAVI_RMI) {
            // incoming message is not a HAVi RMI service
            return false;
        }
        try {
            opCode = new OperationCode(payload);
            controlFlags = payload.readByte();
            transactionId = payload.readInt();
            if((controlFlags & 0x01) == 1) {
                // incoming message is a response, ignore
                return false;
            }
            if(opCode.getApiCode() != ConstApplication.apiCode) {
                // unknown API
                return false;
            }
            switch(opCode.getOperationId()) {
                case ConstApplication.HelloWorldId:
                case ConstApplication.AdditionId:
                    // obtain a worker thread to handle the message
                    // pass the parameters to the message handler
                    Worker worker = pool.getWorker();
                    worker.start(sourceId, payload, opCode,
                              transactionId);
                    return true;
                default: return false; // unknow operation id
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
        return true; // the callback thread can return quickly
    }

    class ThreadPool {
        private Vector idleWorkers;
        private Vector busyWorkers;
        ThreadPool(int numWorkers) {
            idleWorkers = new Vector();
            busyWorkers = new Vector();
            for(int i = 0; i < numWorkers; i++) {
                idleWorkers.add(new Worker(idleWorkers, busyWorkers,
                true));
            }
        }
        Worker getWorker() {
            synchronized(idleWorkers) {
                if(idleWorkers.size() > 0) {
                    Worker w = (Worker)idleWorkers.remove(0);
                    busyWorkers.add(w);
                    return w;
                }
            }
            Worker w = new Worker(null, busyWorkers, false);
            busyWorkers.add(w);
            return w;
        }
        void interrupt() {
            synchronized(busyWorkers) {
                for(int i = 0; i < busyWorkers.size(); i++)
                    ((Worker)busyWorkers.get(i)).interrupt();
            }
        }
    } // class ThreadPool

    class Worker {
        private Vector idleWorkers;
        private Vector busyWorkers;
        private Thread internalThread;
        private volatile boolean noStopRequested;

        private SEID sourceId;
        private HaviByteArrayInputStream payload;
        private OperationCode opCode;
        private int transactionId;

        Worker(Vector idleWorkers, Vector busyWorkers,
                boolean noStopRequested) {
            this.idleWorkers = idleWorkers;
            this.busyWorkers = busyWorkers;
            this.noStopRequested = noStopRequested;
            Runnable r = new Runnable() {
                public void run() {
                    try {
                        this.run();
                    } catch(Exception e) {
                        e.printStackTrace();
                    }
                }
            };
            internalThread = new Thread(r);
            internalThread.start();
        }
        void start(SEID sourceId,
                 HaviByteArrayInputStream payload,
                 OperationCode opCode,
                 int transactionId) {
            this.sourceId = sourceId;
            this.payload = payload;
            this.opCode = opCode;
            this.transactionId = transactionId;
            this.notify();
        }
        private void run() {
            do {
                try {
                    if(idleWorkers != null)
                        idleWorkers.add(this);
                    wait();
                    handleRequest();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    busyWorkers.remove(this);
                }
            } while(noStopRequested);
        }
        // interrupt the thread
        void interrupt() {
            internalThread.interrupt();
        }
        // interrupt the thread and request it to stop
        void stop() {
            noStopRequested = false;
            internalThread.interrupt();
        }
        private void handleRequest() {
            try {
                if(opCode.getOperationId() ==
                   ConstApplication.HelloWorldId) {

                    System.out.println("Hello World");
                    HaviByteArrayOutputStream buffer =
                            new HaviByteArrayOutputStream();
                    Status returnCode = new Status(opCode.getApiCode(),
                                    ConstGeneralErrorCode.SUCCESS);
                    // send the response
                    mySe.msgSendResponse(sourceId, opCode,
                            ConstTransferMode.SIMPLE, returnCode,
                            buffer, transactionId);
                  return true;
                } else if(opCode.getOperationId() ==
                  ConstApplication.AdditionId) {
                    // read two integers from the payload
                    int a = payload.readInt();
                    int b = payload.readInt();
                    HaviByteArrayOutputStream buffer =
                            new HaviByteArrayOutputStream();
                    // perform the addition and
                    // store the result in the buffer
                    buffer.writeInt(a+b);
                    Status returnCode = new Status(opCode.getApiCode(),
                                    ConstGeneralErrorCode.SUCCESS);
                    // send the response
                    mySe.msgSendResponse(sourceId, opCode,
                            ConstTransferMode.RELIABLE, returnCode,
                            buffer, transactionId);
                }
            } catch(InterruptedException e) {
                Thread.currentThread().interrupt();
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    } // class Worker
} // class Application
```

This program has an ConstApplication interface, an Application class, a ThreadPool class, and a Worker class.

Interface `ConstApplication` defines the API code for the appilcation and the operation codes for two different services (lines 6-10).

In the constructor of `Application` class, a `SoftwareElement` object is created to represent this application (line 17); A thread pool with three threads is created (line 19); And the application is registered in Registry (lines 21-30). The request is handled in `msgReceive()`. At first, the message is checked if it is a request for the application (lines 41-63). Then, a worker thread will be obtained (line 69) and started (line 70).

The `ThreadPool` class has two member variables (lines 82-83), `idleWorkers` and `busyWorkers`. `IdleWorkers` is a Vector containing idle worker theads; `BusyWorkers` is a Vector containing busy worker threads, the threads handling requests. In the constructor, worker threads are created and put into the pool (line 88). The getWorker() method first checks if there is a idle worker thread in idleWorkers (line 94). If yes, the idle thread is returned. If not, a new worker thread is created and returned (lines 94-102). Note that the additional worker threads will not added into the pool. The number of worker threads in the pool is fixed. The `interrupt()` method is used to interrupt all the busy worker threads. The previous requests will be cancelled and the threads will be back in the wait state.

As for the `Worker` class, the boolean variable `noStopRequested` (line 116) stands for whether there is a stop request. In the constructor, a thread is created and started (lines 137-138). This thread will execute the run() method (line 129) immediately. Then, the worker thread executes in a while loop in the run() method all its life time (lines 151-162). Mostly the worker thread is in a wait state (line 155). If the worker thread is notified by `start()` (line 148), it returns from `wait()` and executes `handlRequest()` method. Before a worker enters wait state, it adds itself into the idleWorkers and makes itself available (line 154); after a worker thread finishes handling a request, it remove itself from the busyWorkers (line 160). In the `handleRequest()`, two integers read from the payload are added and written into the buffer. Then, the result is sent back by `msgSendResponse`().

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this report, four design issues about software elements are discussed and the solutions are provided. The first design issue is blocking time. Every software element should adopt the design in 3.3 so that the HAVi network can be efficient. The second is multiple requests. The software elements that may receive lots requests at the same time should adopt the design in 3.4 . The third is synchronization. All software elements should adopt the design in 3.5 to avoid deadlock. The fourth design issue is multicasting. The software elements need to multicast can use the design in 3.6

The main contribution of this report is to serve as a design guide for developers. We encountered those design issues while we designed and implemented our HAVi stack. We believe that the design methodology can help developers design and implement software elements that are efficient and flexible.

Besides, the design in 3.6 could be useful especially for those systems providing asynchronous messaging via callback functions. A HAVi network is a message-based distributed system. In other distributed system, there may also be a need to send messages to multiple destinations.

How Messaging System dispatches messages affects the design of software elements. In order to guarantee the portability of DCM and HAVi application, we suggested that the next version of HAVi specification should indicate that whether the callback function of a software element would be invoked concurrently. We also suggested that the callback function of a software element should not be invoked by Messaging System concurrently. Thus, software elements can process the messages in sequences. If software elements want to handle the requests concurrently, worker threads have to be created and the software elements should deal with the synchronization problems.

5.2 Future Work

Although our programs have been tested in a simulated HAVi network composed by three computers, they have not tested with real HAVi-compliant devices. In the future, our HAVi stack and applications should be tested in a real HAVi home network. There may be more devices; the network topology may be more complicated; the distance among them may be longer. Then, the actual performance of our implementation can be measured.

References

[1] Frederic Feit et al, The Home Network Revisited: Which LAN Technologies Will Bring the Network Home?, http://www.itpapers.com, November, 1999.

[2] Nick Pidgeon, How Ethernet Works, http://computer.howstuffworks.com/ethernet.htm.

[3] IEEE Std 1394-1995, Standard for a High Performance Serial Bus.

[4] The Home Phoneline Networking Alliance, http://www.homepna.org/.

[5] The HomePlug Powerline Alliance, http://www.homeplug.org/.

[6] IEEE P802.11, The Working Group for Wireless LANs, http://grouper.ieee.org/groups/802/11/.

[7] The Official Bluetooth® Wireless Info Site, http://www.bluetooth.com/.

[8] The HAVi Specification Version 1.1, http://www.havi.org/.

[9] The Jini Community, http://www.jini.org/.

[10] The UPnP^{TM} Forum, http://upnp.org/.

[11] The Source for Java Technology, http://java.sun.com/.

[12] Java Remote Method Invocation, http://java.sun.com/products/jdk/rmi/index.html.

[13] The HAVi Organization, http://www.havi.org/.

[14] Object Management Group, IDL to Java Language Mapping, v1.2, http://www.omg.org, August, 2000.

[15] HAVi Java API 1.1, http://www.havi.org/, May, 2001.

[16] Hyde, Paul. Java Thread Programming. St., Indianapolis, Indiana:Sams Publishing, 200

[17] Sun Microsystems, Inc, Archive: Java Platform 1.1 API and Documentation, http://java.sun.com/products/jdk/1.1/docs/api/packages.html.

[18] Sun Microsystems, Inc, Java Native Interface Guides, http://java.sun.com/j2se/1.4.1/docs/guide/jni/.

[19] Andreas Bombe, Technical Documents of libraw1394, http://www.linux1394.org/doc/libraw1394/book1.html. 2001

[20] Project: GNU/Linux 1394 AV/C Library, http://sourceforge.net/projects/libavc1394.

[21] 1394 Trade Association, “AV/C Digital Interface Command Set General Specification, Version 2.0.1”, http://www.1394TA.org/, 1998

Publication

Conference Paper:

Kuo-Wei Hsu, Chuen-Liang Chen, Wu-Cheng Li, and Ting-Ying Yu, "A Message Delivery Mechanism for HAVi Network", Internet and Multimedia Systems and Applications (IMSA), 2003