2006年 8月

碩士學位論文

DESIGN of SIP BASED VIDEOCONFERENCING APPLICATION BASED ON UBIQUITOUS MULTIMEDIA FRAMEWORK

朝 鮮 大 學 校 大 學 院

情 報 通 信 工 學 科

DAS BABITA

DESIGN of SIP BASED VIDEOCONFERENCING APPLICATION BASED ON UBIQUITOUS MULTIMEDIA FRAMEWORK

*2006*年 *8* 月 *25* 日

朝 鮮 大 學 校 大 學 院

情 報 通 信 工 學 科

DAS BABITA

DESIGN of SIP BASED VIDEOCONFERENCING APPLICATION BASED ON UBIQUITOUS MULTIMEDIA FRAMEWORK

指 導 教 授　韓　承　朝

이 論文을 工學碩士學位申請 論文으로 提出함.

2006 年　*4* 月　　日

朝 鮮 大 學 校　大 學 院

情 報 通 信 工 學 科

DAS BABITA

*DAS BABITA*의 **碩士學位 論文**을 **認准**함

委員長　朝鮮大學校　敎授　朴　鍾　安　印

委　員　朝鮮大學校　敎授　韓　承　朝　印

委　員　朝鮮大學校　敎授　邊　宰　榮　印

2006 年　*5* 月　　日

朝鮮大學校　大學院

C O N T E N T S

ABSTRACT

DESIGN of SIP BASED VIDEOCONFERENCING APPLICATION BASED ON UBIQUITOUS MULTIMEDIA FRAMEWORK

Das, Babita

Advisor : Prof. Han, Seung-Jo, Ph.D.

Department of Information & Communications,

Graduate School of Chosun University

유비쿼터스 멀티미디어 애플리케이션 개발의 널리 보급에 있어 가장 큰 도전은 디바이스간의 이질성이다. 이번 논문은 통합된 구조에서 디자인과 수행을 말한다. 또 미디어의 시각화를 다루기 위한 인터페이스 그리고 독립적인 디바이스 애플리케이션의 구축을 위한 관리에 대해 제안하고 있다. 하부구조는 기본적 독립 디바이스 신호 조정 구성에 접근 하기위해 cross-platfrom API을 지원하도록 디자인 되어져있고 오디오-시각적 성분에 의존하고 있는 디바이스를 위한 회귀 기능을 제공하도록 디자인 되어졌다. 이 하부구조는 상위 개발자가 ‘JNI(Java Native Interface) wrapper'를 통해 자바 멀티미디어 애플리케이션을 디자인할 수 있게 하여준다. 그리고 계속해서 C 나 C++로 개발된 음성&비디오 핸들링을 위한 미리 편집된 신호 스택의 라이브러리, RTP 스택, 그리고 미디어 처리 구성요소를 지원한다.

이러한 구조에 근거하여 최종적으로 이 논문은 SIP가 PDA나 PC의 작동을 위해 오디오-비디오 다룸의 기능을 가지고 있는 비디오 컨퍼런스 애플리케이션에

어떻게 기초하고 있는지의 개관을 설명하고 있다. 이 구조는 개발자에게 중요한 시간을 절약시킬 것이다.

List of Figures

List of Tables

I. Introduction

A. Overview

Research in Ubiquitous Computing has arrived at a crossroad: A point of convergence where a technology proliferated environment meets with the ability of people to interact with, and make use of, the possibilities that this technology creates. Advances in the various fields of technology allow us to create artifacts and environments that provide computing and communication resources. Nowadays use of mobile devices has exponentially increased, there will be no surprise in the near future that a single user may own multiple such mobile devices or information appliances with different sizes, shapes, functionalities, and capabilities. More and more computing platforms and devices are developed and coming out to our everyday life. The old problem of adapting applications to multiple platforms has become even more important with the large diversity of ubiquitous computing that has emerged within the last couple of years. Problems occur when the applications become collaborative, especially when the users are using heterogeneous devices. These devices have different capabilities in processors, memory, networking, screen sizes, input methods, and software libraries. We also expect that future users are likely to own many types of devices. We believe that there is a need for an application framework that can both assist developers to build multi-platform applications that can run on heterogeneous devices in an effortless manner.

SIP based videoconferencing application is developed using the proposed framework and it automatically selects the most appropriate adaptation strategy at the component level for a target platform. The SIP [1] is a general-purpose communication protocol supporting interactive session established across the Internet. It defines a complete process mechanism for establishing a distant communication session, which is independent of the underlying transport protocol and without dependency on the type of session to be established and also defines how to establish, maintain and terminate Internet sessions including multimedia conferences. It supports personal mobility by discovering users and locating devices, as well as the negotiation among session participants with different capabilities to determine an agreed communication session and supports various multi-party conferencing models, ranging from mixing in end systems to multicast conferences. It seems to be the preferred standard, which provides us with capabilities to architect applications over ubiquitous platforms.

The main objective of this work is to build a time efficient technique, which requires less effort to port on any device. The entire framework components are implemented using C/C++ [2] and are compiled as a library after wrapping inside a Java Native Interface (JNI) wrapper. JNI [3] is a native programming interface that allows Java programmers to integrate native code C/C++ into their Java applications. This paper introduces the Framework elements, explains the features and capabilities and delineates state-of-the-art design and implementation of SIP based videoconferencing application.

B. Motivation

A framework is proposed, which is capable of running multimedia applications on heterogeneous computing and communication devices with different hardware and software capabilities. Heterogeneous devices are desktop computers, notebooks, cell phones, or other emerging mobile devices and information appliances. Currently different applications are needed to be developed for running on different devices. This is an improvement over device-specific applications development. However, application developers have to develop different applications that perform the same function on different types of devices. This is a waste of developers' development time. Using this framework, application developers can develop one application that can run on different types of devices with effortless manner.

The focus of my work is to address the ubiquitous applications development problem, which takes too much effort for authors to learn different device-specific languages and tools, and then to implement and maintain a large number of device-independent applications at design time. In order to solve this problem, a technique is proposed, which assists developers to build device-independent multi-platform applications at design time using less effort. SIP based videoconferencing application [4] is developed, which is based on splitting of an application into components, and it automatically selects the most appropriate adaptation strategy at the component level for a target platform. The framework of this multimedia application will be split into signal-control and audio-visual components. By splitting this application into these two components multi-plat form application for heterogeneous devices can be built.

The implementation of framework of this application is done using two portable languages C/C++ as a back end and Java as a front end environment. C/C++ needs less effort and short time to write codes as they already have built-in functions, which makes easier for authors to develop applications. Once the code will be written in C/C++, it will be embedded in Java and will run on Java platform. For this I chose Java Native Interface, which allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

C. Thesis Organization

The thesis is structured in the following way. In chapter 2 the terms of Ubiquitous Computing is assessed. Then the concept of programming method is introduced.

Chapter 3 follows the basic approach that concentrates on issues relevant for supporting the design issues of framework of ubiquitous multimedia application. An application model and platform is introduced which has natural distribution properties built into the architecture.

In chapter 4, issues relevant for supporting the implementation of application and a new method to ease design and implementation is introduced. A number of prototypical implementations, following the approach, are presented. Libraries and templates for the design of hardware, communication, and software are provided.

The thread performance of this work is shown in chapter 5, where a simple simulation is written in both languages and the execution time is measured.

In chapter 6 summarizes the contributions made in the thesis, but also critically assesses the shortcomings and limitations detected in the course of the research. Furthermore new issues that have been surfacing while working on the thesis is addressed in the future work section of the chapter.

II. BACKGROUND

A. Ubiquitous Computing

When Mark Weiser coined the phrase "ubiquitous computing" in 1988 he envisioned computers embedded in walls, in tabletops, and in everyday objects. In ubiquitous computing, a person might interact with hundreds of computers at a time, each invisibly embedded in the environment and wirelessly communicating with each other [Weiser,93]. Weiser introduced the area of ubiquitous computing (ubicomp) and put forth a vision of people and environments augmented with computational resources that provide information and services when and where desired [Weiser, 91]. For the past decade, ubicomp researchers have attempted this augmentation with the implicit goal of assisting everyday life and not overwhelming it. Weiser's vision described a proliferation of devices at varying scales, ranging in size from hand-held "inch-scale" personal devices to "yard-scale" shared devices. This proliferation of devices has indeed occurred, with commonly used devices such as hand-held personal digital assistants (PDAs), digital tablets, laptops, and wall-sized electronic whiteboards. The development and deployment of necessary infrastructure to support continuous mobile computation is arriving.

Ubiquitous computing assumes there will be large numbers of 'invisible' small computers embedded into the environment and interacting with mobile users. Users will experience this world through a wide variety of devices, some they will wear (e.g. medical monitoring systems), some they will carry (e.g. personal communicators that integrate mobile phones and PDAs), and some that are implanted in the vehicles they use (e.g. car information systems). This heterogeneous collection of devices will interact with intelligent sensors and actuators embedded in our homes, offices, transportation systems to form a mobile ubiquitous computing environment which aids normal activities related to work, education, entertainment or healthcare. There is a need for wireless communication to support mobile interaction but the environment will also provide access to wired backbone networks connected to the internet.

Although these intelligent communicators will be far more sophisticated than current mobile phones, they will always have limited storage, processing, display capabilities and battery power compared to fixed PCs. There is thus a need to adapt information and applications so that they are compatible with the limited capabilities of the devices but also to provide information or adapt services that are relevant to the current context of the user. Sensors in the environment, possibly in collaboration with personal devices, would determine user's current activity - driving a car, walking down a street, in the cinema, in a meeting, running for a bus, about to watch television. The ubiquitous computing environment would thus support users in common day-to-day activities by adjusting lights, switching on the television for favorite programmes, record the programme when unable to watch it, monitor health and alert emergency services in case of problems, warn drivers about potential component failures in their car etc.

Approaches in Computer Science in the last 50 years can be related to the quantitative relationship between computers and humans. At the very beginning many people shared a single computer, then the idea that each user has a single computer significantly changed the way people used computer systems. In the last decade this changed further into a many-to-one relationship, where one user has many computers, or at least devices with processing capabilities available to, and surrounding a single user. This recently started era is referred to as Ubiquitous Computing; however, Ubiquitous Computing raises many issues beyond the quantitative relationship between computer and user [Weiser, 91], [Weiser, 96]

A major challenge in Ubiquitous Computing is physical integration and embedding of computing and communication technology into environments and artifacts. Such developments lead to ‘augmented artifacts’, raising issues beyond the physical integration. Embedding technology into everyday artifacts also inevitably implies embedding the “computer” into tasks done by the user. This leads to new research challenges and further questions.

B. Programming Model

Embedded devices, like PDA or mobile phones have known a great development in the last few years. In the same time, their users required better performances and richer applications. Today, such devices are mainly used to manage calendars or emails, but adding multimedia data and high speed wireless networks would allow new applications to appear: games, music and video content playing.

Making them work in an efficient way is still a hot research topic. As embedded devices have limited resources (processor, memory, energy), a lot of work remains in several areas: hardware, operating systems and applications. An execution environment for embedded architecture should:

- Make the concurrent execution of several applications efficient and robust;
- Minimize the energy consumption of the devices;
- Allow the execution of multimedia applications, with real-time constraints;
- Allow applications to be downloaded dynamically;
- Provide an open architecture, as portable as possible.

Meeting such requirements, a Java environment appears to be the best solution. First, it allows the dynamic load of new applications. Then, Java bytecodes are not architecture specific: applications can be ported to many platforms without effort. Java also offers a great stability and security because its memory is automatically managed. If the environment has a garbage collector, no pointers are used and no direct memory accesses are possible. This allows a regular user to download and run any Java program, while being almost sure it will not mess-up the whole memory. Finally, Java appears to be a popular language and an ever growing number of applications are developed, particularly for embedded devices. Nevertheless, the main problem with the use of Java is its execution performance. A virtual machine must always be present, introducing more operations and thus leading to a performance loss and bigger resources consumption. Thus, even if more and more mobile devices are “Java Compliant” and can execute a program written in Java, this does not mean that they can do it efficiently. Indeed, today there are no efficient couples of hardware and software solutions. A lot of enhancements can still be made, in term of performances, energy consumption and platform flexibility.

The implementation of framework of our application is done using two portable languages C/C++ as a back end and Java as a front end environment. C/C++ needs less effort and short time to write codes as they already have built-in functions, which makes easier for authors to develop applications. Once the code will be written in C/C++, it will be embedded in Java and will run on Java platform. For this we chose Java Native Interface, which allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

C. Why Java Native Interface?

While we can write applications entirely in Java, there are situations where Java alone does not meet the needs of your application. Programmers use the JNI to write Java native methods to handle those situations when an application cannot be written entirely in Java. The Java Native Interface (JNI) is a native programming interface. It allows Java code that runs inside a Java Virtual

Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly. The most important benefit of the JNI is that it imposes no restrictions on the implementation of the underlying Java VM. Therefore, Java VM vendors can add the support for the JNI without affecting other parts of the VM. Programmers can write one version of native application or library and expect it to work with all Java VMs supporting the JNI.

The following examples illustrate when we may need to use Java native methods:

- The standard Java class library may not support the platform-dependent features needed by the application.
- You may already have a library written in another language, and wish to make it accessible to Java code through the JNI.
- You may want to implement a small portion of time-critical code in a lower-level language such as assembly.

By programming through the JNI, we can use native methods to:

- create, inspect, and update Java objects (including arrays and strings)
- call Java methods
- catch and throw exceptions
- load classes and obtain class information
- perform runtime type checking

We can also use the JNI with the Invocation API to enable an arbitrary native application to load and access the Java VM. This allows programmers to easily make their existing applications Java-enabled without having to link with the VM source code.

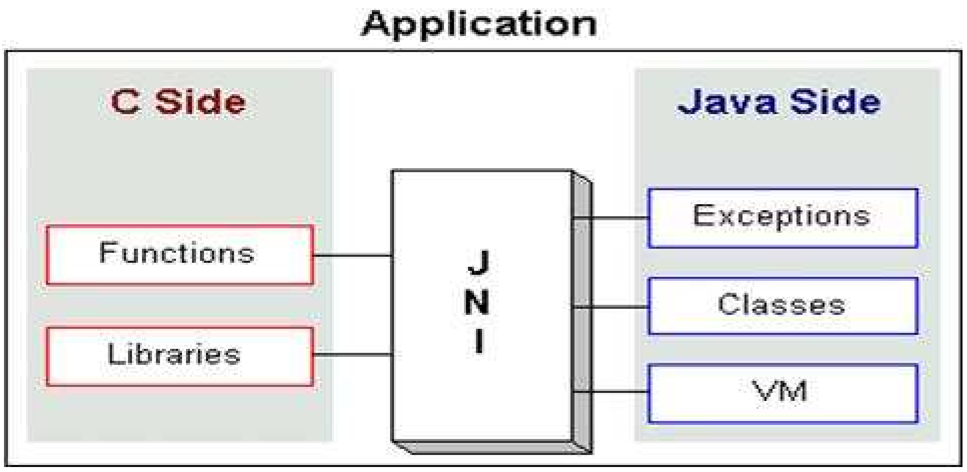

Fig.1. Java Native Interface Block Diagram

The JNI specification is formed following a series of discussions among Java Soft and the licenses on the design of a standard native method interface. In particular, all major Java VM vendors played an active role and made extensive contributions to the JNI design.

Using Java to interoperate with natively compiled code usually removes the portability benefits Java brings to the table. However, there are cases when doing so is acceptable, even required, such as when interfacing with legacy libraries, interfacing with hardware or the operating system, or even just improving performance. Writing to the JNI standard does at least guarantee that the native code will work with any JVM implementation.

3. DESIGN ISSUES

The framework is a set of media and control management interfaces and utilities that may be used in multimedia communications applications. The user develops custom applications using the framework application-programming interface (API) [7] to control and manage the system software components. The framework includes:

1. Java Native Interface
2. Component Manager
3. Media processing component
4. Packet processing component
5. Control and signaling component

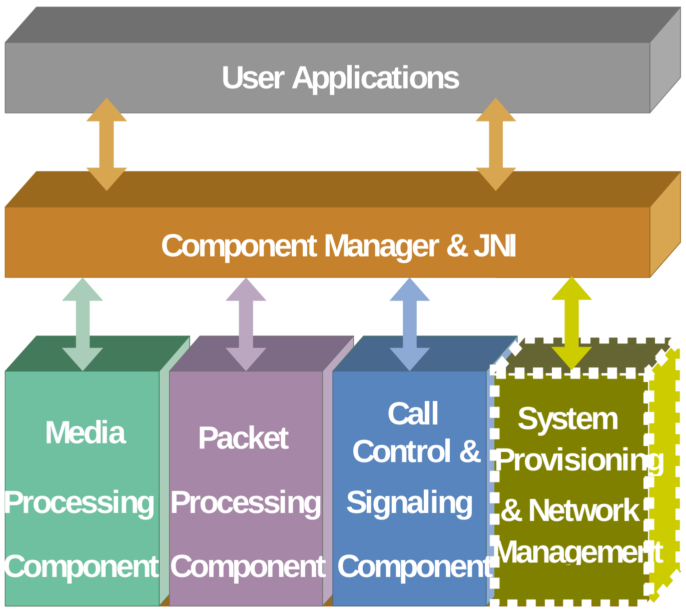
User Applications
Component Manager & JNI
Media Processing Component
Packet Processing Component
Call Control & Signaling Component
System Provisioning & Network Management

Fig.2. Proposed Framework Architecture

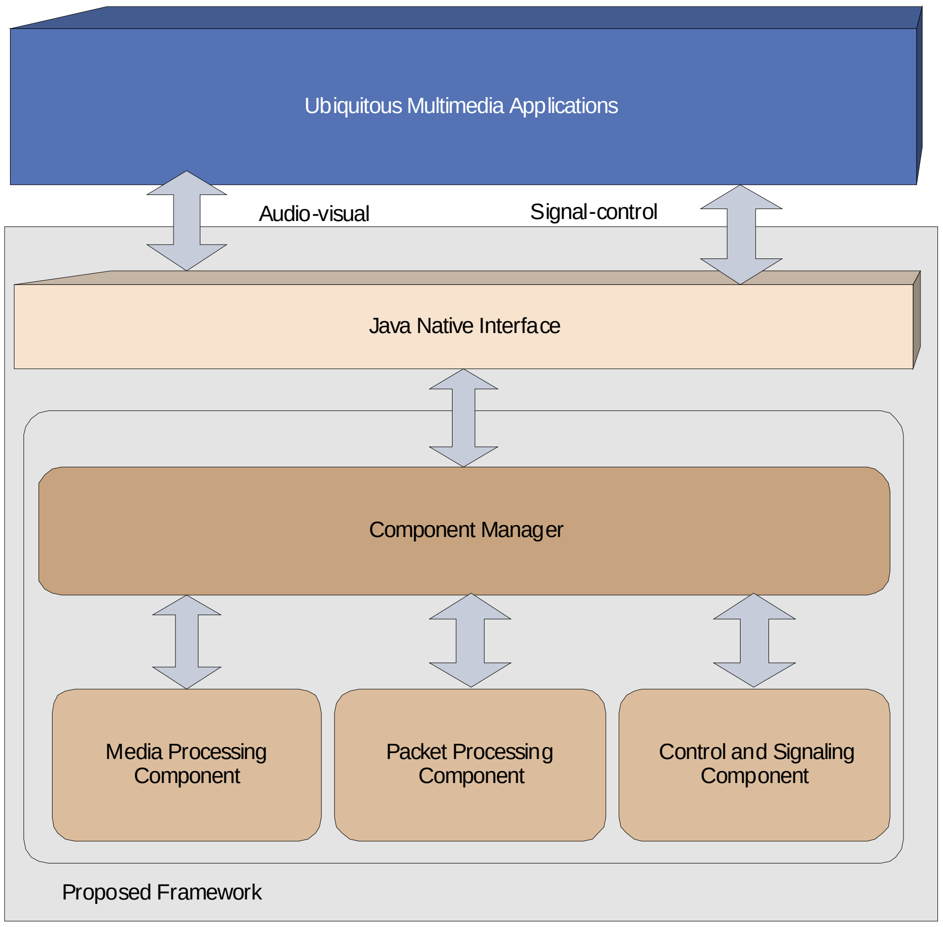
Ubiquitous Multimedia Applications
Audio-visual
Signal-control
Java Native Interface
Component Manager
Media Processing Component
Packet Processing Component
Control and Signaling Component
Proposed Framework

Fig.2.1 Proposed Framework Architecture

The architecture of the framework is outlined in the following sections and the components are described in the sections that follow.

A. Java Native Interface (JNI)

The Java Native Interface [3] is a powerful framework for seamless integration between Java and other programming languages (called “native languages” in the JNI terminology). The JNI is useful when existing libraries need to be integrated into Java code, or when portions of the code are implemented in other languages for improved performance. A common case of using the JNI is when a system architect wants to benefit from both worlds; implementing communication protocols in Java and computationally expensive algorithmic parts in C++ (the latter are usually compiled into a dynamic library, which is then invoked from the Java code). The JNI renders native applications with much of the functionality of Java, allowing them to call Java methods, access and modify Java variables, manipulate Java exceptions, ensure thread-safety through Java thread synchronization mechanisms, and ultimately to directly invoke the Java Virtual Machine.

The Java Native Interface is extremely flexible, allowing Java methods to invoke native methods and vice versa, as well as allowing native functions to manipulate Java objects. However, this flexibility comes at the expense of extra effort for the native language programmer, who has to explicitly specify how to connect to various Java objects.

The main objective of using JNI is to allow the developers to build a device-independent application. This layer provides interface to Java programmers to make calls to native code and use our C/C++ based designed components as shown in above figure1. The most important benefit of the JNI is that it imposes no restrictions on the implementation of the underlying Java Virtual Machine (VM). Therefore, Java VM vendors can add the support for the JNI without affecting other parts of the VM. Programmers can write one version of native application or library and expect it to work with all Java VMs supporting the JNI.

B. Component Manager

The Component Manager as the name suggests manages all the components in the Framework. It instantiates the various components to be added in the system as directed by the application developer, provides an interface to the application to interact with system components, activates and deactivates the components and allows the runtime addition and removal of the components from the system.

C. Media Processing Component

Media-processing, such as signal processing, 2D- and 3D-graphics rendering, and image and audio compression and decompression are the dominant workloads in many systems today. Media applications demand large amounts of absolute performance and high performance densities (performance per unit area and per unit power). Therefore, media processing applications often use special purpose fixed-function hardware.

The content creation system of multimedia streaming services may have one or more media sources (e.g., a camera and a microphone). In order to compose a multimedia clip consisting of different media types, the raw data captured from the sources are edited. It should be noted that multimedia content could also be synthetically created without a natural media source. In order to facilitate attractive multimedia retrieval service over commonly available transport channels such as low-bit-rate modem connections, the media clips are also compressed in the editing phase before they are handed to a server. Typically, several clients can access the server over a determined network. Then the client decompresses and plays the clip. In the playback phase, the client utilizes one or more output devices, most often the screen and the loudspeaker of the client. By streaming, a media server opens a connection to the client terminal and begins to stream the media to the client at approximately the playout rate. During media receiving, the client plays the media with a small delay or no delay at all. This technique not only frees up precious terminal memory, but also allows for media to be sent live to clients as the media event happens.

D. Packet Processing Component

Realtime transport protocol (RTP) [8] is an IP-based protocol providing support for the transport of real-time data such as video and audio streams. The services provided by RTP include time reconstruction, loss detection, security and content identification. RTP is primarily designed for multicast of real-time data, but it can be also used in unicast. It can be used for one-way transport such as video-on-demand as well as interactive services such as Internet telephony. RTP provides the basic functionality needed for carrying real-time data over packet networks. It does not offer mechanisms for reliable data delivery or protocol-specific flow and congestion controls such as the ones offered by TCP. RTP relies on other protocol layers, capable of managing and controlling network resources, to provide on-time delivery and the framing service. The services provided by RTP include payload-type identification, sequence numbering, time stamping and delivery monitoring. RTP typically runs on top of user datagram protocol (UDP).

RTP has been standardized by IETF (Internet Engineering Task Force) as RFC (1889). It provides end-to-end network delivery services for real-time media. This is achieved primarily through the RTP [5] packet header, which is appended to the data as it descends through the protocol stack. This header has a variety of fields which the sender or receiver can use to manipulate the data stream. For example, a 32-bit timestamp field can be used to discard time-delayed packets at the receiver, or synchronizes two incoming streams. This additional information is necessary for time based media to be presented correctly and serves to illustrate why RTP is necessary on top of UDP. While RTP does not provide any mechanism to ensure timely delivery or provide other QoS guarantees, it is augmented by a control protocol, called Real-time Transport Control Protocol (RTCP), which monitors the quality of the data distribution. This protocol can also be used to provide information about the source, such as its geographical location, as well as providing general signaling functions such as notifying the receiver that the end of the data stream has been reached.

E. Control and Management Signaling Component

Session Initiation Protocol (SIP) seems to be the preferred standard, which provides us with capabilities to architect applications over ubiquitous platforms. This paper introduces the Framework elements, explains the features and capabilities and delineates state-of-the-art design and implementation of SIP based video conferencing system.

The deployment of SIP (Session Initiation Protocol) in enterprise networks gives users significantly greater flexibility to use and control commonly used communication technologies. An interesting feature of SIP is the ability to separate the control and media portions of a connection. In fact, they may take place at different devices. Thus SIP allows a level of indirection and late binding of the media device. This is a powerful concept and has the potential to do for communications what pointers did for programming by separating data from addresses and virtual memory did for large programs by creating virtual addresses to overcome limits of real addresses.

Since the ability to exercise control places very few requirements on end devices, it is possible to make wearable devices with a TCP/IP stack part of the SIP infrastructure. In this paper we argue that wearable computers are ideally suited for setting up sessions because of their availability and ability to get the user's attention in varied situations. Wearable computers also contain information that can be used to personalize and improve the user experience. Similarly stationary devices are better suited to play the media because they have fewer constraints in terms of power, size, networking, etc. The above balance is unlikely to change because it is based on human behavior patterns and basic physics.

IV. IMPLEMENTATION

A. Implementation of SIP Video Conferencing Application

The general architecture of the Framework deployed in SIP video conferencing application is shown in fig 2. It consists of a pre-compiled library containing SIP signaling stack, RTP stack and media processing components for voice and video handling. The framework is designed to allow the configuration of the existing components and a seamless integration of new components in the framework. The design ensures that no changes are required in media management component when the application is mapped to different hardware architectures and operating systems.

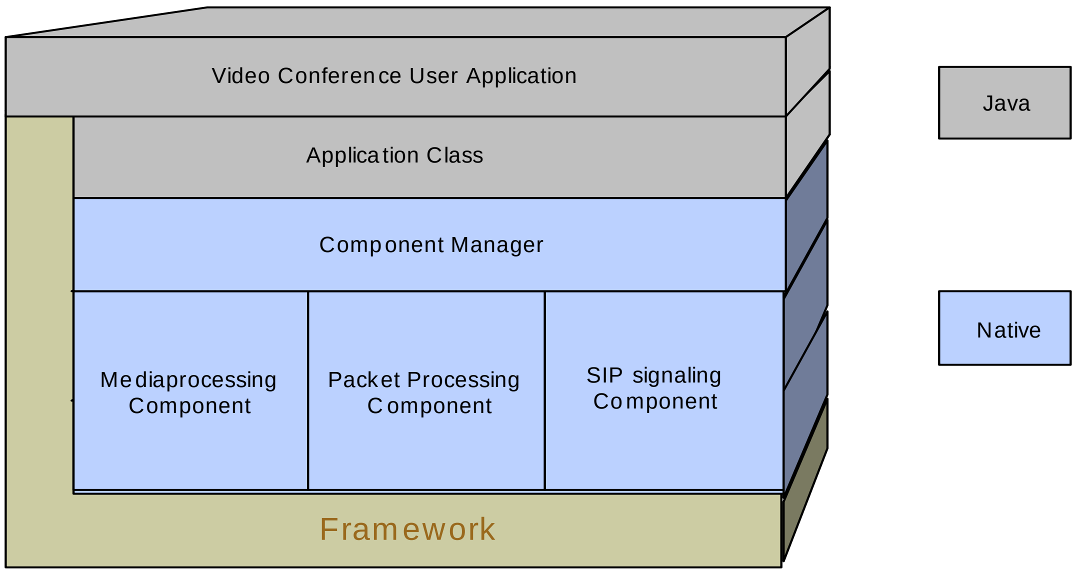
Video Conference User Application
Application Class
Component Manager
Mediaprocessing Component
Packet Processing Component
SIP signaling Component
Framework
Java
Native

Fig.3. Layered architecture of SIP videoconference application

It consists of a number of components and a manager that registers and manages them and aims at providing the necessary details to develop a SIP based Videoconferencing application. The User Application is implemented by extending the application class provided by the framework. It provides the user with a communication interface to the underlying library through an event handler. The underlying components can send events/ messages to this Java application class by calling certain call back functions. The base class also provides default functionality for handling these status and error notifications. The user application is developed as an extension of this class and the developer can over write the default functionality. The user application communicates with the system components through the component manager using JNI.

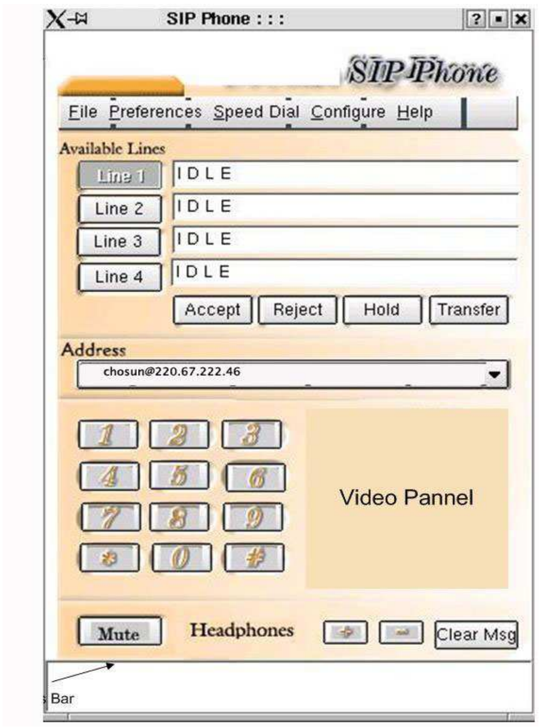

Fig.4.SIP Videoconferencing Application

This section describes the API provided by library. The API [1] is divided into subsections that deal with management of components (like addition, deletion, configuration etc.), call control management and media control interfaces.

B. Component Management Interface

The Component Manager functions as a registry of existing components. It also provides the capability of configuring all underlying components under the command of the User Application. It also provides the interface for communicating to all the components in the system included as a compiled image or added at run time. The derived User Application class has access to all the API functions of the Component Manager supported in native interface. Component Manager has a list of all the components in the system corresponding to their IDs. These IDs are assigned to the components at the time of their addition (compile/run time). Any further reference to an added component in the system is therefore through this identifier.

Each component implements this function to complete any initialization tasks like thread creation etc. This is done by the application by calling the ActivateComponents function of the Component Manager which in turn calls the StartComponent() function of the components present in the system. The Component Manager passes the functions to the corresponding component only if it is in the active state.

InstantiateComponents( )
ActivateComponents( )
ActivateComponents(final int componentId)
DeactivateComponents( )
DeactivateComponents(final int componentId)
GetComponentStatus(final int componentId,param pStatus)
GetComponentConfiguration(final int componentId,Param pConfiguration)
SetComponentConfiguration(final int componentId,Param pConfiguration)

Table.1. Component management API

C. Call control and Management Signaling Interface

Call Control and Management Protocols fall into this category. Examples are SIP, H323, MGCP etc. This component has a listener thread that receives incoming signals from the Transport, parses the messages according to the specific protocol and then intimates the application of the action that need to be taken. The signaling gateway comprises of a number of Signaling Terminations. An incoming new request is first directed to the application that decides the termination to route this request to.

AddSignalingTermination (int terminationId, Param pConfig)
RemoveSignalingTermination (int terminationId)
ApplyEventToSignaling (int terminationId, Enum event,Param lParam)
DisconnectSignalingTermination (int terminationId)
SetCofiguration (Param pConfig)
GetConfiguration(Param pConfig)
GetStatus(Param pStatus)

Table.2. Call control and Management Signaling API

D. Media Stream Processing Interface

The media processing components are Voice Controller and Video Controller. Voice Controller handles the different Audio Codecs on the system. It has a record of all the audio codecs that are present in the system along with their parameters like the input stream size on which they operate, the output stream size that they generate, state sizes if any is required etc. It also keeps a record of the different channels that have been created by the application and their current state (e.g. active or suspended). The I/O data buffers on which the tasks operate are provided by the application.

LocateVoiceMedia(Enum enumVoice,java.lang.String filename);
getVoiceConrtol(Enum voiceData,Param VoicObj);
setVoiceControl(Enum voiceData,Param VoiceObj);

Table.3. Voice stream processing API

Video Controller handles the different Video Codecs on the system. It has a record of all the video codecs that are present in the system along with their parameters like the input stream size on which they operate, the output stream size that they generate, state sizes if any is required etc. It also keeps a record of the different channels that have been created by the application and their current state (e.g. active or suspended).

LocateVideoMedia(Enum enumVideo,java.lang.String filename)
GetMediaStream(java.lang.string [ ] videostream,param videoObj)
getVideoControl(Enum videoEnum,param Videoparam)
SetVideoControl(Enum videoEnum,param Videoparam)

Table.4. Video stream processing API

E. Event Handling and Call Processing

The real power of the framework lies in its extendibility. This section talks of the extended functionality of the user Application such as Handling of events that are detected in the Application as well as configuration of most of the components through SNMP.

The Application class pvroides a function HandleEvents, which is called whenever a component wants to request some information or signal an event. This is a virtual function and can be overloaded by the derived class. A simple implementation is shown below:

```
errCode HandleEvents(
  final int  sourceId,
  final Enum code,
  final Param paramObj)
{
switch (code)
  {
    default:
      super.HandleEvents(sourceId, code, paramObj);
}
}
```

The event code allows the user to write custom handlers. If there is no event handler attached with a specific code, the default case calls the HandleEvents method of the base class. Following diagram shows the flow of in SIP bases video conferencing application which has been developed on top the above-described

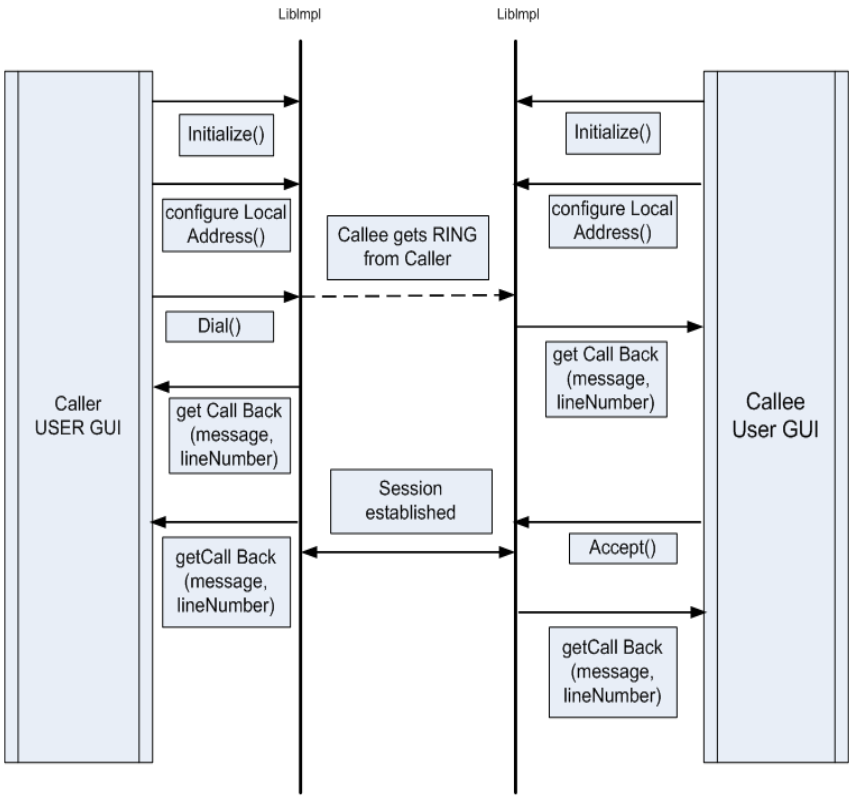
LibImpl
LibImpl
Initialize()
Initialize()
configure Local Address()
Callee gets RING from Caller
configure Local Address()
Dial()
get Call Back (message, lineNumber)
Caller USER GUI
get Call Back (message, lineNumber)
Callee User GUI
Session established
getCall Back (message, lineNumber)
Accept()
getCall Back (message, lineNumber)

Fig.5. Call flow in SIP based videoconferencing

This provides the default functionality for a particular event code. The following code provides examples of some event handlers, which a derived class should handle:

The constructor operations in the previous section should have their counterparts in the destructor. The following code shows a sample code for the destructor.

```
ShutDown ( )
{//The signaling terminations are unregsitered and removed from the //signaling
gateways. Anyother memory allocated and threads created in //the sample
application are destroyed here.
  HashMap termsigMap = new hashMap ( );
  Iterator iter = set.iterator( );
 while (iter.hasNext( ))
  {
    INT32 termId,compId;
    termId=termsigMap.get('termId');
    termConfig=term,sigMap.get('config');
    compId=config.compManagerConfig.termSigMap.get('gatewayId');
    pCompManager.RemoveSignalingTermination(compId,termId,termConfig);
  }//All the components in the stack are then stopped and deactivated.
PCompManager.DeactivateComponents( );
PCompManager.RemoveComponents( );
}
```

Sample Application then deactivates all the components present in the system. This is being done here by calling Component Manager's function deactivatecomponents( ) which in turn calls the StopComponent( ) function of all the components its handling. In the implementation of these functions, the functions deallocate any memory that was previously allocated, stop and terminate any threads that were created at start up and do any other resource release that was used by them. Remove component then deletes their instances from the system.

V. PERFORMANCE

To compare the performance of the proposed framework on following different platforms and different devices with different capabilities a simple simulation was written in both languages Java and C++ versions of the library in order to show the thread performance of these different platforms. The table below shows the different platform used and the device capabilities on which these platforms are deployed.

Platform	Average Execution time over 5 runs
RTLinux on intel	9341 ms
Windows on Intel	1538 ms
Windows CE (mobile)PDA	12910 ms
Solaris Stand-alone Java	11214 ms
Solaris C++	1568 ms

Table.5. Simulation Execution times

A. Simulation

Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The operating system kernel allows programmers to manipulate threads via the system call interface. Some implementations are called kernel threads, whereas lightweight processes is a specific type of kernel threads that share the same states and informations. Each simulation object runs in its own thread, so the performance of the underlying threading system is important for large simulations. The Java runtime system uses the underlying operating system for thread support or its own software emulation if the OS does not support threads.

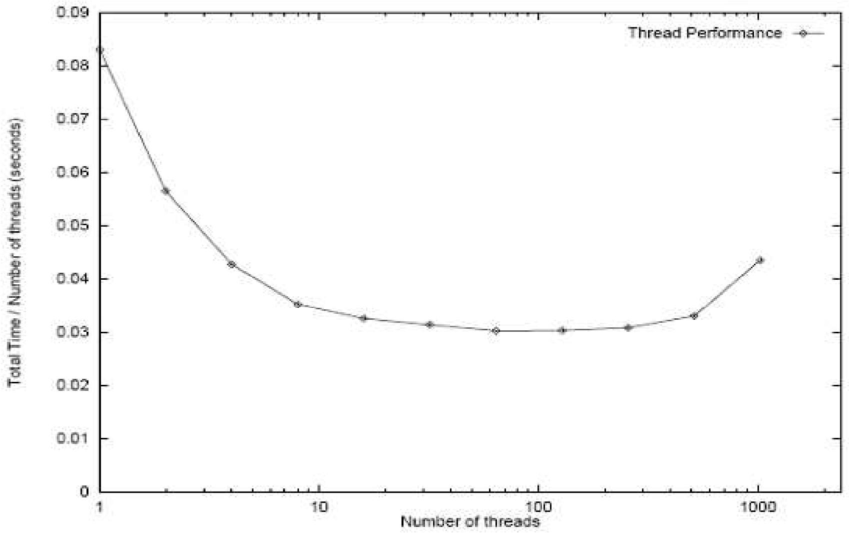

Fig.6. Thread Performance

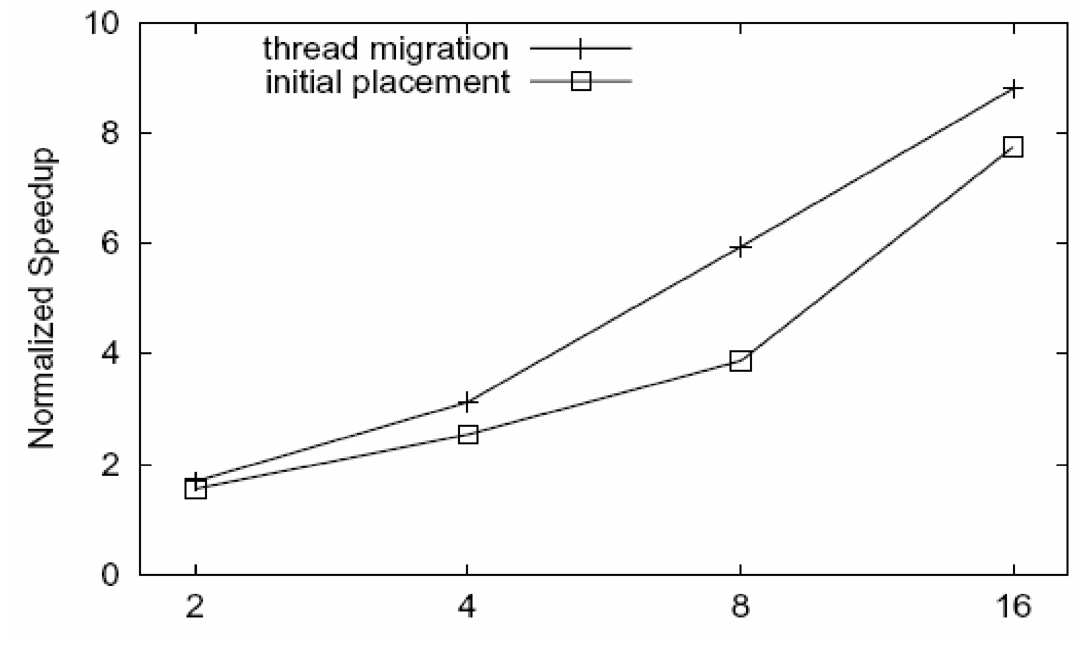

Fig.7. Multithreaded Java application simulation

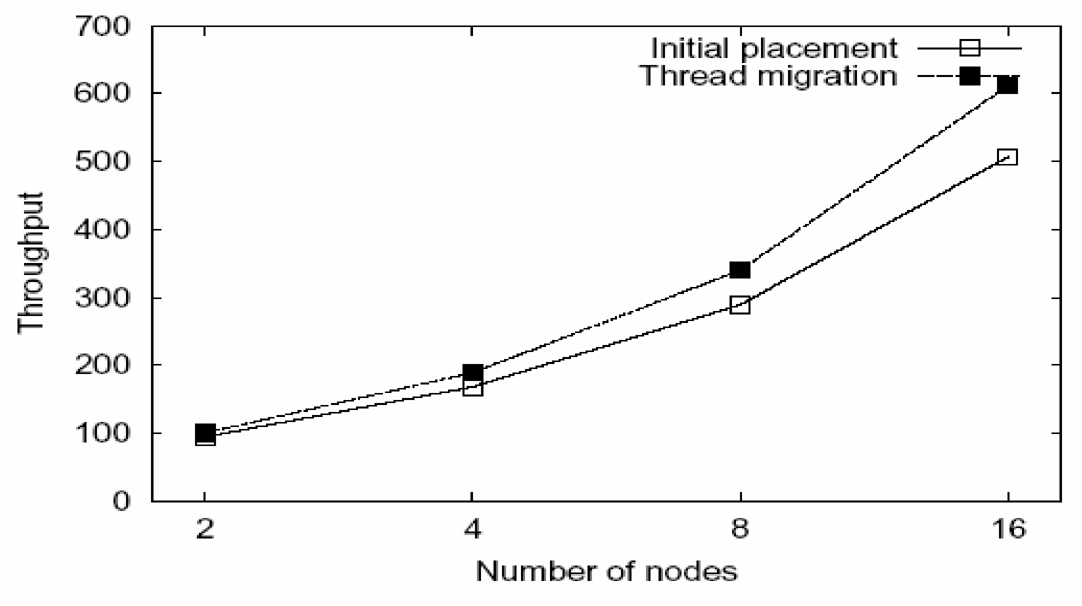

Fig.8. Thread performance simulation

VI. CONCLUSION

We have developed and implemented a framework, which is based on two components, signal-control component and audio-visual component. Splitting of multimedia application into these two components provides an edge to build multi-plat form application for heterogeneous devices. This framework enables advanced developers to design multimedia applications in Java through a Java Native Interface (JNI) wrapper and seamlessly extends interface to support pre-compiled library of signaling stack, RTP stack and media processing components for voice and video handling developed in C and C++. The framework is designed to allow the configuration of the existing components and a seamless integration of new components in the framework. The design ensures that no changes are required in media management component when the application is mapped to different hardware architectures and operating systems. Based on this scheme, we developed SIP based videoconferencing application that uses callbacks to manage audio-visual for running on PC as well as on PDA. This framework can save significant time for developers.

Usually with each prototype finished, each system evaluated, and each paper published a number of new issues that pose interesting challenges appear. In future this framework can be used for migration system that facilitates transparent migration from one server to another during their life time. To assist further development of the augmented scenario-based design process, we need to take it through the rigors of developing an actual working system prototype. This will provide first hand experience of its adequacy as an aid to the design process. The evaluation challenge of ubiquitous computing systems still persists, mainly because the effect of actual use situations is very difficult to recreate in lab settings.

REFERENCES

[1] S Berger, A Acharya, and C Narayanaswami, “Unleashing the Power of Wearable Devices in a SIP Infrastructure”, 3rd IEEE International Conference on Pervasive Computing and Communications.

[2] Stroustrup B, "The C++ Programming Language", 3rd edition, Addison Wesley, 1990.

[3] Sheng Liang, “The Java Native Interface: Programmer's Guide and Specification”

[4] S Zhou and J Cheung, “A Simple Platform Independent Video/Voice over IP Application”, Software Engineering and Applications- SEA, 2004.

[5] Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “RTP: A transport protocol for real-time applications”, IETF RFC 1889, January 1996.

[6] J Bates, D Halls, and J Bacon, "A migration framework for ubiquitous computing applied in mobile applications", Mobile Networks and Applications, 1996.

[7] Douglas Kramer, Bill Joy, and David Spenhoff, “The Java Platform”-A White Paper. JavaSoft White Paper, ftp://ftp.javasoft.com/docs/JavaPlatform.ps, May 1996.

[8] C Wong, H Chu, and M Katagiri, "A single authoring technique for building device Independent presentations", W3C Workshop on Device Independent Authoring, 2002.

[9] Albrecht Schmidt, “Ubiquitous Computing - Computing in Context: a thesis on Context-Awareness, Context Aware Computing, and Ubiquitous Computing”, November, 2002.

[10]Arnaud Guiton and Michel Banâtre, “An experimental study of Java objects behavior for mobile architectures”, January, 2005.

[11] Michael David Pinkerton, “Ubiquitous Computing: Extending access to mobile data”, Masters Thesis, Georgia Institute of Technology, June, 1997.

[12] Gregory D. Abowd and Elizabeth D. Mynatt, “Charting Past, Present, and

Future Research in Ubiquitous Computing”, ACM Transactions on Computer-Human Interaction, Vol. 7, No. 1, March 2000.
[13] Frank Siegemund, “Cooperating Smart Everyday Objects - Exploiting. Heterogeneity and Pervasiveness in Smart Environments”, Phd. Thesis, ETH Zurich, 2004.
[14] Song, H., Chu, H., Kurakake, and S. Browser, "Session Preservation and Migration". In Poster Session of WWW 2002, Hawai, USA. 7-11. May, 2002.
[15] YI CUI, KLARA NAHRSTEDT, and DONGYAN XU, “Seamless User-Level Handoff in Ubiquitous Multimedia Service Delivery”. Multimedia Tools and Applications, Springer Netherlands, February 2004.
[16] Candy Wong, Hao-hua Chu, and Masaji Katagiri, “W3C Workshop on Device Independent Authoring Techniques”. DoCoMo Communications Laboratories USA, Inc. August 14, 2002
[17] Manfred Glesner, Thomas Hollstein, Tudor Murgan, “System Design Challenges in Ubiquitous Computing Environments”, IEEE, 2004.
[18] Gregory D. Abowd. “Software Design Issues for Ubiquitous Computing”, College of Computing &. Graphics, Visualization and Usability Center.
[19] Arup Acharya, Stefan Berger, and Chandra Narayanaswami, “Unleashing the Power of Wearable Devices in a SIP Infrastructure”, Proceedings of the 3rd IEEE Int’l Conf. on Pervasive Computing and Communications (PerCom 2005).
[20] CS Perkins and J. Crowcroft, “Notes on the use of RTP for shared workspace applications”, ACM Computer Communication Review, Volume 30, Number 2, April 2000.
[21] Weiser, M, “The Computer for the 21st Century”, Scientific American, 265(3):94-104, September 1991. [Weiser, 91].
[22] Weiser, M, “Some Computer Science Issues in Ubiquitous Computing”, Communications of the ACM, 36(7):75-84, 1993. [Weiser, 93].
[23] Weiser, M, "Ubiquiotus Computing Homepage",
http://www.ubiq.com/hypertext/weiser/UbiHome.html, March 1996. [Weiser, 96].

[24] Tatsuo Nakajima, Kaori Fujinami, Eiji Tokunaga, and Hiroo Ishikawa, "Middleware Design Issues For Ubiquiotus Computing",Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia, 2004

[25] Eila Niemela and Juhani Latvakoski, "Survey of requirements and solutions for ubiquitous software", Proceedings MUM '04 Citation.

[26] Christoph Endres, Andreas Butz, and Asa MacWilliams, "A survey of software infrastructures and frameworks for ubiquitous computing", Mobile Information Systems, 2005.

[27] Zhexuan Song, Ryusuke Masuoka, Jonathan Agre, and Yannis Labrou, "Task computing for ubiquitous multimedia services", International conference on Mobile and ubiquitous multimedia, 2004.

[28] A Ranganathan, Robert E. McGrath, Roy H. Campbell, and M. Dennis Mickunas, "Ontologies in a Pervasive Computing Environment". Proceedings of the Workshop on Ontologies and Distributed , 2003.

[29] Stefan Berger, Henning Schulzrinne, Stylianos Sidiroglou, and Xiaotao Wu, "Ubiquitous computing using SIP", Proceedings of the 13th international workshop on Network, 2003.

[30] Gregory D. Abowd, "Ubiquitous Computing Research Themes and Open Issues from an Applications Perspective".