Journal of Universal Computer Science, vol. 10, no. 4 (2004), 473-498
submitted: 17/10/03, accepted: 2/2/04, appeared: 28/4/04

A Survey of Multimedia Software Engineering

Mercedes Amor
(University of Málaga, Spain
pinilla@lcc.uma.es)

Lidia Fuentes
(University of Málaga, Spain
lff@lcc.uma.es)

Mónica Pinto
(University of Málaga, Spain
pinto@lcc.uma.es)

Abstract: Developing multimedia applications entails understanding a variety of advanced technologies; in addition, multimedia programming poses a significant challenge in terms of handling a variety of hardware devices, multimedia formats or communication protocols. Therefore, breakthrough software engineering technologies should be applied to produce reference architectures able to support ever-changing requirements. In this paper, we first present the challenges that designers must face in multimedia programming, and how current frameworks address them, specially regarding the management of architectural evolution. We then show what breakthrough approaches or technologies can be used to produce more reusable, extensible and open multimedia systems. We focus on presenting the benefits of applying component-based software development and application framework technologies. We also illustrate how to componentize all multimedia functionalities and (re)use the resulting components as COTS in application frameworks. This approach helps to add multimedia capabilities to an application without requiring specific knowledge on multimedia.
Key Words: multimedia systems, components, framework
Category: D.1, D.1.5, D.2, D.2.13

1 Introduction

Since the early 90s, multimedia has been recognized as one of the most important cornerstones in the software engineering field. Recent technologies for digital multimedia as well the ever-increasing popularity of the Internet and the web justify the need to provide new multimedia applications. Some current examples include multimedia groupware, video on demand services, video conference, electronic shopping systems or entertainment systems.

The management of multimedia data implies the ability to create and include images, audio and movies into the applications. The ability to deliver multimedia information interactively is changing the way in which people communicate. It is the basis for more interactive and customizable applications since the presentation of high-quality media makes it possible to offer users more choices more naturally. Apart from capturing and presenting multimedia information, such applications need to deliver large amounts of data, which requires the use of appropriate Internet protocols. The Internet makes it possible to implement synchronous real-time services to distribute high-quality audio and video according to different interaction models.

Therefore, developing multimedia applications entails understanding and using a variety of advanced technologies, which increases the complexity of multimedia programming. The main requirements that should be taken into account to design, implement and maintain such environments are the following:

- Multimedia applications need to deal with many different hardware devices, e.g., cameras, microphones, or speakers. Due to their continuous evolution, they need to be upgraded continuously, e.g., to manage a brand-new multimedia mobile phone, DVD, USB camera, or network camera.

- These devices capture or present multimedia data in different formats, which makes format negotiation between the participants necessary. The nature of multimedia data and presentation environments will continue to evolve, e.g., low-power mobile systems. As a result, we should expect developers to cope with evolving data formats and emerging host environments.

- Distributed multimedia applications need a communication protocol to transfer data across the network, e.g., HTTP is used for on-demand access to multimedia documents and UDP is normally used for audio, whereas dynamic media require several continuous stream delivery protocols such as RTP or IP-multicast.

- Developers of distributed multimedia applications must understand the constraints imposed by a delivery protocol and the impact it can have on the interaction and presentation of information. These constraints can be bandwidth, delay jitter, or synchronization. Furthermore, such applications should be able to adapt to the continuous variations of the data delivery rate, decreasing or increasing data resolution as the network traffic changes.

- Multimedia applications need to share some information among remote users from time to time. Such applications build on share spaces that enable the collaboration between the users that are inside the same place. Designers should offer applications flexible enough to allow almost any kind of interaction among a group of users, which is not a trivial task. User access control, customization and awareness are common issues in these applications.

As a result, the development of multimedia applications entails the important issue of handling such a variety of hardware devices, multimedia formats, multimedia databases or communication protocols. Furthermore, multimedia application developers need to understand the peculiarities of the main application areas or at least be able to (re)use some existing domain specific software. Traditionally, software engineers have not spend too much time on this task. Perhaps the issue is that software engineers do not like multimedia, and multimedia engineers are not interested in applying software engineering principles. As a consequence, we think that few efforts have been spent in developing highly configurable multimedia software that really facilitates adding multimedia capabilities to an application without requiring specific multimedia knowledge. To cope with this challenge, it is crucial to use advanced software engineering technologies, i.e., the implementation cannot be seen as the only activity in the development of a multimedia system [Gonzalez, 2000]. Designers should produce reference architectures and distributed environments able to support the important architectural evolution requirements of multimedia applications. This requires using software technologies that make it possible to produce more reusable, extensible, and open multimedia systems.

In this context, there are two main technologies, namely: Component-Based Software Development (CBSD) [Brown and Wallnau, 1999][Szyperski, 2002] and Application Frameworks [Fayad and Schmidt, 1997]. They are becoming essential to reduce costs and improve reusability, flexibility and maintainability. In a component-based platform, components act as replacement units that can be composed to build extensible, reusable, and open systems. This technology allows to build an application by plugging in commercial off-the-shelf (COTS) components, developed at different times, by different people, and with different uses in mind. However, the flexibility provided by components should be complemented by a framework that provides an underlying architecture [Krieger and Adler, 1998]. A framework is a reusable, semi-complete application that can be refined to produce custom applications [Fayad and Schmidt, 1997]; thus frameworks can be considered a better approach than isolated object-oriented or component-oriented technologies.

We have enough experience in applying component and framework technologies to multimedia applications. MultiTel [Fuentes and Troya, 1999] is a component framework that is particularly well-suited to develop multimedia applications; later, we developed CoopTel [Pinto et al., 2001a], which is an aspect-component framework for collaborative virtual environments (CVEs) that combines the benefits of components and advanced separation of concerns [AOSD, 2004][Kiczales et al., 1997]. From our experience, component, aspect and framework technologies are particularly well-suited to address the strong architectural evolution requirements of multimedia applications. However, to be able to offer multimedia COTS components, we first had to deal with existing multimedia application frameworks. We used the Java Media Framework, or JMF for short [de Carmo, 1999], and Lotus Sametime [IBM, 2003], which are two of the most widely used multimedia frameworks for Java. Our first aim was to componentize them and reuse the resulting components inside MultiTel and CoopTel.

The rest of the paper is organized as follows: we present a short survey of multimedia APIs and frameworks [see Section 2]; then, we go into deeper details, and report on JMF and Sametime [see Section 3]; later, we report on how to componentize JMF and Sametime [see Section 4]; finally, we present our conclusions [see Section 5].

2 Related Work

Currently, there is a plethora of solutions for multimedia software development by means of APIs, SDKs, or application frameworks. Well-known examples include the Java Media APIs by Sun Microsystems [Sun Microsystems, 2004a], the Windows Media SDK by Microsoft [Microsoft, 2004] and the Digital Media Library by Silicon Graphics [Silicon Graphics, 1996]. Other approaches consist of software products that offer a complete solution for programming basic multimedia services as well as sophisticated multimedia services like collaborative applications, e.g., IBM Lotus Sametime [see Tab. 1].

Java Media APIs provide a unified, non-proprietary and platform-neutral solution for adding multimedia capabilities to Java applications. Low-level APIs, such as 2D, 3D, Advance Imaging, Image I/O, and Sound and Speech, support the integration of audio and video clips, animated presentations, 2D fonts, graphics, images, speech input/output, and 3D models. More advanced APIs, such as the Java Shared Data Toolkit (JSDT) in conjunction with the Java Media Framework (JMF), provide high-level support to develop collaborative and multimedia applications. JSDT can be used to create shared whiteboards, chat environments, remote presentations, shared simulations, and to distribute data in workflow applications [see Section 3.1 for an overview of JMF].

Microsoft offers a similar set of SDKs. One of them is the Windows Media Technology, a set components that are suited to build multimedia applications in Windows. Some of their capabilities include: (i) the Player Component, which provides a programming interface for rendering a variety of network streaming and non-streaming multimedia formats; (ii) the Format Component, which can be used to read, write and edit files in Windows Media Format, both for multimedia playing and content creation; (iii) the Encoder Component, which supports streaming delivery, together with the automation of the local and the remote encoding process; and (iv) the Services Component, which is used to configure, manage, and administer Windows Media Servers. Another important Microsoft SDK is DirectShow, which offers an architecture to capture and playback multimedia streams using Windows Driver Model devices in a variety of formats, e.g., MPEG, AVI, MP3, WAV. Sample applications include DVD players, video editing applications, format converters, MP3 players, and digital video capture applications. Since DirectShow is based on the Component Object Model (COM) [Box, 1998], multimedia programmers can develop their own COM components or (re)use components provided by DirectShow.

Microsoft provides a set of products for adding collaborative features to applications. The Collaboration Data Objects technology is appropriate for building messaging applications. NetMeeting supports standardized data, audio and video conferencing, Windows application-sharing and shared whiteboards. Furthermore, Microsoft Windows Messenger includes an API to register and run new peer-to-peer applications for Messenger clients. The main drawback of these SDKs is that the resulting components can run only in Windows, which reduced drastically their (re)use in other platforms. In

Java Media APIs	- *2D, 3D, Advanced imaging and image I/O APIs* for graphic, image and visual environment creation and manipulation
	- *Sound and speech APIs* for audio playback, capture, mixing, MIDI sequencing, MIDI synthesis and speech recognition
	- *Shared data toolkit* to add collaboration features to applets and applications
	- The *Java Media Framework* for ...
	  - Media files and live stream capture and presentation
	  - Media conversion
	  - Video and audio conference using RTP
	  - Multimedia data management reference architecture
	  - Cross-platform technology

Windows Multimedia Technologies	- *Windows Media SDK* and *Windows DirectShow SDK* to capture, edit and play back Windows media-based content and multimedia streams - *DirectX, OpenGL, WIA and GDI APIs* to add graphics, video, 3D animation and sound to Windows applications - *Windows messaging and collaboration* technologies to support collaborative capabilities
Digital Media Library	- Audio and video files and live stream creation - Presentation of audio and video - Encoding of audio and video

Sametime Toolkit	- Sametime Java Toolkit for ...
	  - Login, basic awareness and messaging
	  - Video and audio conference using IP
	  - Chats, application sharing and shared whiteboard
	- Sametime Community Server Toolkit for adding new real-time services to Sametime

Table 1: Related Work in Multimedia Development Approaches

addition, multimedia programmers are supposed to have broad technical skills on low-level multimedia and Windows programming. Nevertheless, these Microsoft products cover the most important issues of multimedia programming.

The Digital Media Library encompasses a set of APIs for data format description, live audio and video input/output with built-in conversion capabilities, and some facilities to work with multimedia files. Silicon Graphics also supplies end-user desktop media tools for capturing, editing, recording, playing, compressing, and converting audio data and images. From the point of view of code reuse and modularization, using

Digital Media implies programming using the rather old-fashioned C language.

IBM Lotus Sametime consists of a family of web-based collaboration products that provide real-time meeting services with awareness, screen-sharing capabilities, and IP audio and video conferencing. Sametime can be used to develop custom multimedia applications since it includes two kinds of application development toolkits, namely: the client and the community toolkits. Client toolkits enrich client applications with collaborative and real-time capabilities; the community toolkit allows writing and including new Java server applications into the Sametime server. Although Lotus claims that these toolkits cover nearly all software development platforms, i.e., COM, Java, and C++, Sametime APIs are not prepared to be (re)used inside component platforms as stand-alone components [see Section 3.2 for an overview of Sametime].

These technologies provide a similar set of toolkits or APIs for multimedia programming. To provide a deeper insight into the steps involved in the development of multimedia applications, henceforth we focus on two of them, namely: JMF and the Sametime Java Client Toolkit. They both use Java and attempt to provide powerful abstractions and mechanisms to ease the development of multimedia services. However, JMF is a low-level API that allows programmers to capture and configure hardware devices, encode and transmit media streams, and so on, whereas the Sametime Java client toolkit can be considered a high-level API that provides a set of components that use the Sametime server functionality and hides low-level details.

3 Application Frameworks for Multimedia Software Development

Usually, multimedia services must support heterogeneous users since applications must be deployed across different software platforms. Therefore, using Java helps avoid platform dependencies. JMF and Sametime are two examples of object-oriented frameworks written in Java that can be used to develop portable multimedia applications. In this section we present an overview that focuses on how easy to use they are and what multimedia requirements they support. We also present a comparison regarding the extensibility and adaptability of each framework.

3.1 Overview of JMF

The Java Media Framework API (JMF) is an application programming interface for incorporating audio, video and other time-based media into Java applications and applets [Sun Microsystems, 2004b]. JMF was Sun's answer to the demands for multimedia application frameworks in Java. It raised important expectations from the beginning, and has been used extensively. It is more than a simple API since it offers an application framework that provides a frame architecture to model basic multimedia functionality. Therefore, incorporating multimedia processing into Java applications should be easy with JMF, but, unfortunately, the complexity of multimedia software development is similar to other approaches for non-experts. We think that this is due to two reasons:

1. It deals with specific multimedia management only, and it is not appropriate to construct complete multimedia applications, e.g., e-learning environments. JMF supports the capture and transmission of multimedia data between participants only; tasks such as user authentication, media negotiation, or service scheduling need to be programmed from scratch.

2. It covers all the issues related to the capture, transmission and presentation of multimedia data at a low level of detail. Although providing a complete framework permits a more accurate development, complexity increases enormously. For instance, JMF developers do not work with objects representing real world devices like cameras, microphones or speakers, as we would expect; instead, they have to deal with objects such as data sources, players and processors, which need to be combined to capture and reproduce multimedia data.

Therefore, the core abstractions in the JMF programming model are data sources, players and processors, namely: a data source encapsulates the media streams captured for any kind of input device supported by JMF, e.g., video/audio files, cameras, microphones; the player reproduces media streams by using the appropriate output devices, e.g., speakers, video panels; processors allow to process multimedia data before it is rendered, e.g., format conversion, enconding, decoding. [Fig. 1] shows the relationships among these elements. First, a datasource object gets the data stored in a video/audio file [label 1]. Next, a processor allows to demultiplex the stream to separate the audio and video tracks in the file (demultiplexer plug-in), to add effects (effect plug-in), to encode it (codec plug-in), and to mix the audio and video tracks again (multiplexer plug-in) [label 2]. The processor can reproduce the stream or either generate a new datasource that will be the input for another processor or player [label 3] . Finally, this media stream can be either reproduced by a local player or can be sent through the network to a remote player [label 4].

In the second case, JMF offers the RTP API for the distribution of real-time streams using the Real-Time Transport Protocol (RTP) [Schulzrinne, 1996]. RTP is a transport protocol designed to be used by applications with real-time requirements. It provides information about the format and the time of the data to be transferred, and allows the correct reconstruction of data at the target. Furthermore, it offers control information about the quality of the current transmission and the participants engaged in a session by means of the RTCP protocol (Real-Time Transport Control Protocol). The rest of this section describes JMF interfaces and how to build a multimedia service.

3.1.1 Device Capture

The first step in any multimedia application is to capture multimedia devices. JMF provides resource managers to deal with the registration and capture of real-time multimedia devices. The JMF resource manager is implemented by a `CaptureDeviceManager`

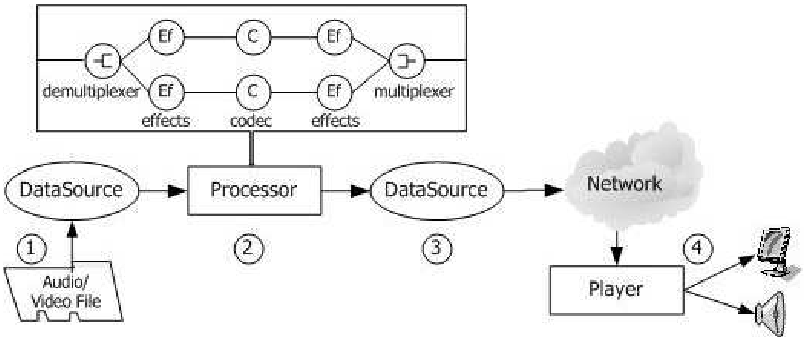

Figure 1: JMF Presentation and Processing

component that offers methods to register new devices (`addDevice()`) and to provide information to capture them (`getDevice()` and `getDeviceList()`) [see Fig. 2].

The use of the JMF resource manager has the advantage that all the devices installed in the user machine that are supported by JMF are directly available to JMF programmers. JMF supports many different devices, specially in Windows. In addition, JMF provides the JMFRegistry, which is a stand-alone Java application to register new devices, data sources, or plug-ins. This registry is crucial to cope with the continuous evolution of hardware devices. However, the JMF resource manager also entails some drawbacks regarding the information a software developer needs to know to capture a device. Using the `getDeviceList(format)` method in the `CaptureDeviceManager` object, developers need to specify exactly the format of the device to be captured. This may lead programmers to cumbersome errors, e.g., if there is a camera installed, but a format that is not supported by that camera is requested, the application will not capture it. In addition, JMF allows to capture a device by using its unique device identifier (`getDevice(devName)`), but this solution is platform-dependent because device identifiers change from one system to another, which reduces the reusability of applications in different platforms. For instance, `vfw://0` may refer to a camera in Windows, whereas the same device is referred to as `sunvideo://0` in Solaris.

Anyway, interaction with the JMF resource manager is not sufficient to have access to the multimedia data captured by a device. This is due to the fact that JMF does not provide an atomic method to capture a device. Instead, applications must invoke several methods that create a set of chained components, which increases the complexity of the capture process and requires more expertise. [Fig. 2] shows the JMF device capture process in detail. Once one knows the format of the device to be captured, the first step is to use the `CaptureDeviceManager` object to get the list of devices with that format. By invoking the `getLocator()` method on one of the `CaptureDeviceManager` structure elements returned by the JMF resource manager, we obtain a `MediaLocator` that allows to create a `DataSource` object from which multimedia data is read.

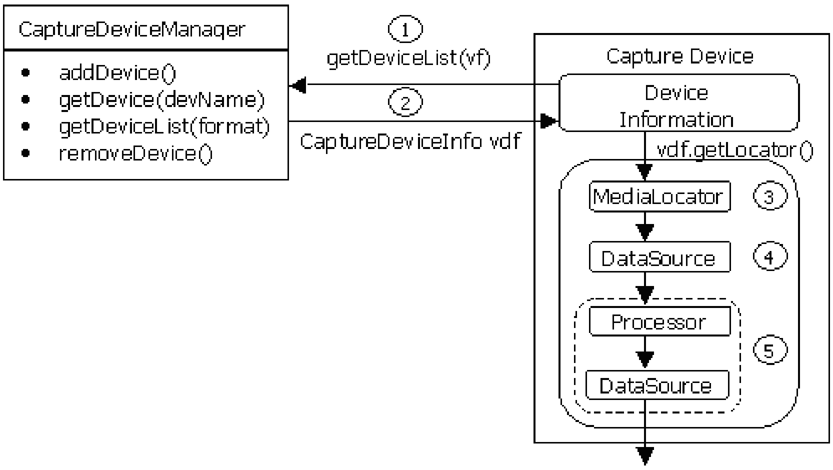

Figure 2: JMF device capture process

Optionally, if the media format has to be processed, a `Processor` and a new `DataSource` need to be created. This code gives an idea of how vast the user's knowledge of multimedia programming has to be, and in particular regarding JMF API classes, to be able to capture a device, which should be a simple task.

3.1.2 Format Configuration and Negotiation

If we need to change the format of the data that is being captured from a device, we need to add a `Processor` component that makes the new coding. JMF supports an extensive list of audio and video formats, e.g., AIFF, GSM, MPEG Layer II Audio, Sun Audio, Wave, and MPEG-1 Video. With respect to the media that can be transmitted through RTP, some examples include G.711, GSM mono, G.723, MPEG Layer I, II and mono DVI for audio and JPEG, H.261, H.263 and MPEG-I for video. Anyway, it is not realistic to expect that all the participants that may join a multimedia application will transmit data in the same or a directly compatible format. Each participant may have devices from different manufacturers or may have different requirements regarding the available bandwidth or even user preferences. Thus, data format negotiation is a significant issue in multimedia programming.

Although JMF allows to add new codecs as plug-ins, its architecture does not address how the different participants can negotiate data formats. If we assume that RTP is the communication protocol used, then every participant can know the transmission format of any other participant or can change it if a problem is detected. But how can participants negotiate the format? How do they know whether other participants will be able to reproduce the data they are sending? JMF says nothing about this, so the problem remains unsolved.

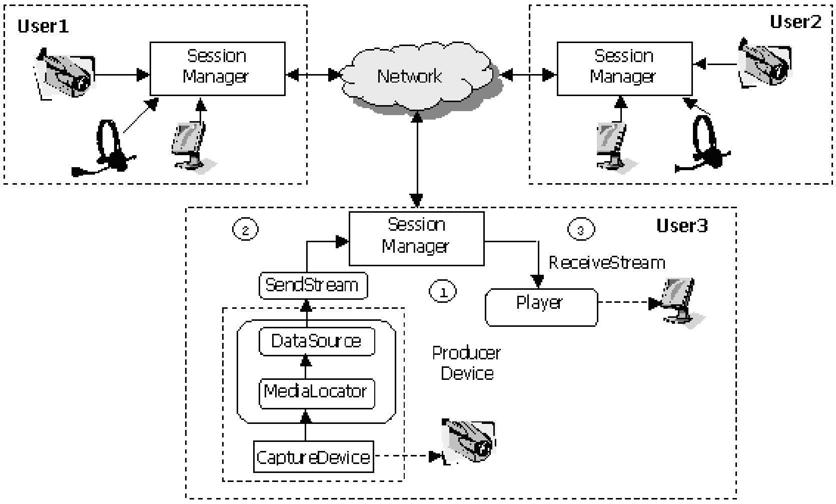

Figure 3: JMF architecture for multimedia services

3.1.3 RTP Media Transmission

The software architecture for multimedia data delivering proposed by JMF is based on the RTP protocol, whose definition is provided by the JMF RTP API. This API enables you to open RTP sessions for sending and receiving RTP streams that are then reproduced using a JMF player. In addition, the session can be monitored using the stream statistics that are sent through the RTCP protocol. [Fig. 3] shows the architecture for RTP multimedia services. It is shared by all kinds of multimedia services transmitting real-time media data, such as video conference and video on demand services. The main steps required to create an RTP session and initiate the transmission of RTP streams are outlined. After capturing the participating devices, the first step is to initiate the transmission by means of a `SessionManager` [label 1]. This class is a local representation of an RTP session and performs media transmission operations for each participant in the service. It maintains information about the participants and controls the input and output data streams. Each participant sends and receives RTP data packets and RTCP control packets through this component. Then, a `SessionManager` creates a stream called `SendStream` and connects it to the device output `DataSource` (label 2 in [Fig. 3]). At the target site, the service must wait for an event that provides information about input data from a new participant. When this occurs the `SessionManager` creates a stream called `ReceiveStream` and a `Player` component for data presentation [label 3].

Although everybody agrees in that RTP is an appropriate protocol for multimedia data transmission, it is not the only option available. Unfortunately, JMF lacks sufficient flexibility to use other communication protocols for transporting multimedia data.

JMF does not offer a common interface modelling the connections between the participant of a service, and therefore the networking part of JMF cannot be extended with new communication protocols that may appear in the future, nor can designers choose among protocols that offer, for instance, different quality of services. Consequently, services that require the use of proprietary protocols or new ones, for instance, have to implement them from scratch..

3.2 Overview of Sametime

Lotus Sametime is a family of collaboration products that provides a set of collaborative tools and services such as real-time awareness, screen-sharing capabilities, and live audio and video inside a community of users. Its architecture includes a server and a set of client applications that enable real-time meetings over an intranet or the Internet. An important contribution of Sametime is that it provides a set of APIs that allows multimedia programmers to use Sametime services in in-house applications. We focus on the Sametime Java Client API [IBM, 2004], which provides a collection of Java-based interfaces that are used for accessing all the functions and services provided by Sametime. [Fig. 4] shows the services provided by the Sametime Java API and how they are organized, namely:

Sametime Community Services: These include sending text and/or binary messages between users during a chat session (Instant Messaging Service or IM), or just handling one-way messages (Announcement Service). The Awareness Service allows to know the status of objects such as users, groups, and servers. The Community Service is the core service used to log in and out from a Sametime server. Furthermore, the Community Service provides the ability to handle messages from the administrator to change your status and privacy in the community.

Meeting Services: They provide collaborative services like application sharing, shared whiteboard, and IP audio and video. The Application Sharing Service (`AppShare`) allows a user to share a local application with remote users. In addition, remote users are able to view and/or control the shared application. This service is used to include on-line editing, remote control, software demonstrations, and presentations. The Whiteboard Service allows a set of users to view and annotate contents in a shared whiteboard by using a basic set of tools for drawing lines, boxes, circles, and text. The Streamed Media Interactive Service provides two-way communication via audio and video. This service encompasses all the functionality needed to participate and control two-way or multiple-way IP audio and video conference. The Streamed Broadcast Media Service allows to receive audio, video, and data content produced by an interactive media client. This service allows to view and listen to the information only and uses multicast whenever possible, i.e., it is not interactive, which makes it more scalable.

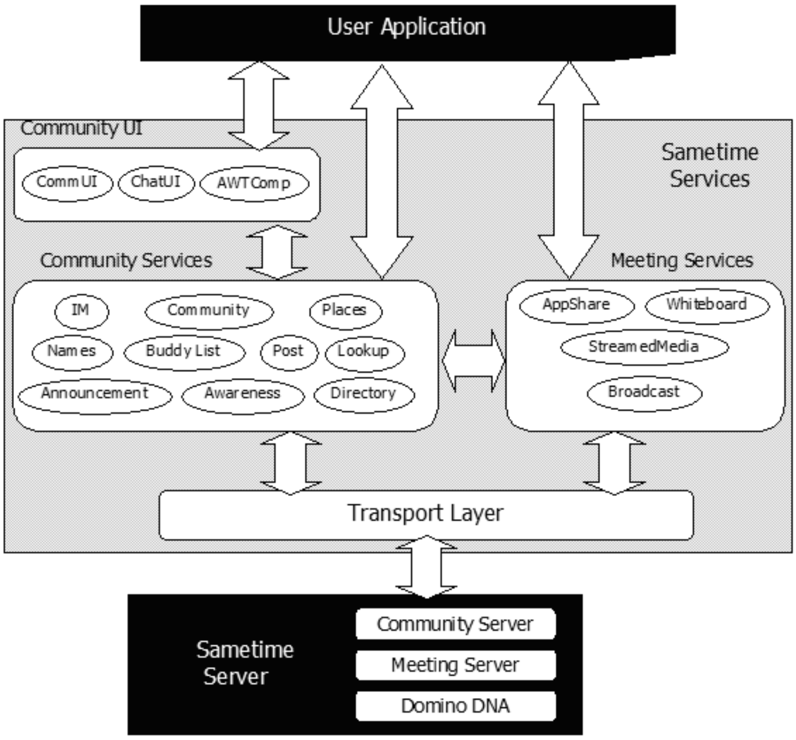

Figure 4: The Sametime Java Client and Server Architecture

Community UI Components: They provide a standard user interface for the Community Services functionality, e.g., user awareness lists or chat windows. These components allow a non-expert developer to listen to and display events from Community Services components without having to program a new user interface. The problem is that it cannot be modified; thus if we want a different one we have to program it from scratch, including the handling of the proper input events.

When developing a Sametime-based application we have to take into account that the core abstractions are services, which are defined as Java interfaces and are implemented by a class. Service invocation in Sametime is always asynchronous, i.e., event-based. Sametime applications use the standard delegation-based Java event model, and register themselves as listeners for service events. The Sametime API also provides classes and attributes for representing connected users, groups of users and sessions in the Sametime server. An important feature is that it provides virtual places to group users who wish to meet and collaborate. A Sametime virtual place is an abstraction of a meeting place which users enter to collaborate. Therefore, collaborative activities provided by Meeting services, such as application sharing or live audio and video, must happen in Sametime virtual places. This architecture unifies the Meeting and Community Services in Sametime, since it binds the activities provided by the Meeting Services with a virtual place provided by the Places Service. Thus, the Places service allows creating places and adding meeting activities to them.

The main advantage of using Sametime is that it is a high-level programming API with which it is easy to develop multimedia applications. The API undertakes all low-level details of capturing devices and transmitting media streams in a transparent way. Developers just need to learn how to use the appropriate services. It is thus well-suited for non-expert developers in multimedia programming. However, the API requires skills in Java programming for managing events and Java AWT components. One of the current problems of Sametime-derived applications is that the graphical interfaces provided by Sametime services are based on the rather old Java AWT package.

Although the high-level abstractions that Sametime provides help keep developers away from specific media details, they prevent to control the devices, the formats and the transmission process. Sametime simplifies capturing multimedia devices by providing atomic methods, but it does not allow to configure them or to add new devices. Furthermore, the media formats supported are limited to both the G.711 and the G.723 audio formats and the H.261 and H.263 formats for video transmission using RTP. The API does not provide any service for changing or negotiating media formats. Depending on the bandwidth requirements, the audio format is changed by accessing the server management tool, and the resulting configuration is maintained for all transmissions.

Once the corresponding devices are captured and audio and video streams are initiated, the developer is released from synchronizing audio and video streams, which is delegated to the Sametime server. In Sametime, the input video stream is always from the same user who is transmitting on the incoming audio stream, which means that the video player displays the user who is currently speaking and nobody else. Again, the developer's task is simplified, but it is limited to receiving from one source at a time. For instance, it is not possible to display a user who is not speaking or to multiplex several audio and video streams, which is a major drawback.

3.3 Comparing JMF and the Sametime Java Client Toolkit

In this section, we present a brief comparison between JMF and Sametime services regarding how extensible, interoperable, easy to evolve and usable they are. We hope that the results of this evaluation may guide programmers to choose the most suitable API in each case.

Extensibility: Can new devices be added? Can new data formats be supported by adding new codecs? Can new communication protocols for real-time media transmission be added? JMF can be extended so as to accommodate new devices, data sources and plug-ins such as codecs, effects, multiplexers and demultiplexers; however, Sametime offers a predefined set of services that cannot be extended. None of them allow to use communication protocols other than RTP. Therefore, designers must choose between high-level multimedia programming with reduced configuration possibilities, i.e., Sametime, or tedious low-level multimedia programming with adequate configuration options, i.e., JMF.

Interoperability: Both JMF and Sametime support cross-platform deployment since they are Java-based solutions, but some limitations exist. The implementation of JMF 2.1.1 supports a performance pack for both Solaris, Linux and Window platforms; in other cases, JMF provides a cross-platform version, which does not implement any multimedia device. This makes it impossible for applications running on these platforms to capture and transmit multimedia data if software developers do not provide their own device implementations. Sametime is even more restrictive: although Community services such as Awareness or Instant Messaging can be invoked from any platform, Meeting services, which support multimedia applications, can only be run on Windows. Even though the API is written mostly in Java, there are two small native modules that connect with the operating system to capture live audio and video, which are provided for Windows only. In addition, Sametime can deal with only a small set of manufacturer cameras and sound cards.

Ease to evolution: We define this property as the ability of an API to be extended and evolve without affecting applications developed with previous releases. The interfaces defined in the specification of JMF have remained stable since 1999. Since then, Sun has been upgrading the API implementation, including support for new devices and JMF plug-ins. From our experience using Sametime, we can conclude that the applications that were developed using Sametime 1.x are totally incompatible with Sametime 2.x. The evolution between 2.x and 3.x releases was smoother. Anyway, an application using version 2.x may not work properly with Sametime 3.x. We are not sure what is going to happen with future releases.

Usability: The usability of a framework does not depend on its complexity only, but also on the documentation provided. Even though JMF is a low-level complex API, it provides enough documentation and working examples to make JMF one of the most widely used multimedia APIs. However, its complexity makes extending it a dreadful task. Contrarily, Sametime is a high-level API that hides low-level details and makes developing a multimedia application easier, but it is poorly documented.

Last, but not least, we should mention that JMF is a freeware solution while the Sametime Java API requires purchasing the Lotus Sametime family of products.

4 JMF, Sametime and Component-Based Software Development

On account of the previous sections, the reader might well have realized that developing a multimedia service is not easy. In addition to the implicit complexity of distributed applications, developers must learn how to manipulate multimedia data, configure hardware devices, or even connect user hardware and software devices according to different interaction patterns. Neither JMF nor Sametime are component-based technologies. It is not possible to add a multimedia service from any commercial multimedia product to our application as an external plug-in. This means that basic multimedia services are not available to be (re)used as black-box components. As a consequence, software developers have to deal with low-level details of the API. We consider that the main goal of multimedia software engineering is to produce highly (re)usable multimedia components ready to be used in a certain application. Developers of multimedia software should be able to search and (re)use such components in the component marketplace. Component-based software development may encourage the (re)use of multimedia software components even by non-experts in this field.

In this section, we report on our experience in the development of component-based multimedia and collaborative applications by using JMF and Sametime to implement simple and advanced multimedia services as stand-alone components. These components were designed for the MultiTel and CoopTel component frameworks.

4.1 Combining JMF and the MultiTEL Framework

MultiTEL (Multimedia TELecommunication services) is a complete development solution that applies component and framework technologies to multimedia and collaborative applications [Fuentes and Troya, 2001]. It defines both a compositional model implemented as a composition platform, and an application framework. The former provides separation of concerns between computation and coordination by encapsulating them into two different kinds of entities. The compositional units of the model are components that encapsulate computation and connectors [see the circles in Fig. 5] that encapsulate the coordination protocols specified by state transition systems. A component is able to receive messages and react to them by performing some internal computation that usually terminate by throwing an event that is handled by a connector [see the dotted lines in Fig. 5]. Connectors interact with components by sending messages. By applying the separation of concerns among computation and coordination, MultiTEL increases the (re)use of both kind of entities in different contexts.

The connection between components[1] and connectors is established in MultiTEL at run time, which provides a powerful mechanism for late binding among them. This

[1] In this context, we do not distinguish between components and component instances, so we use the term component to mean a set of classes assembled to be deployed together and executed as a single software unit.

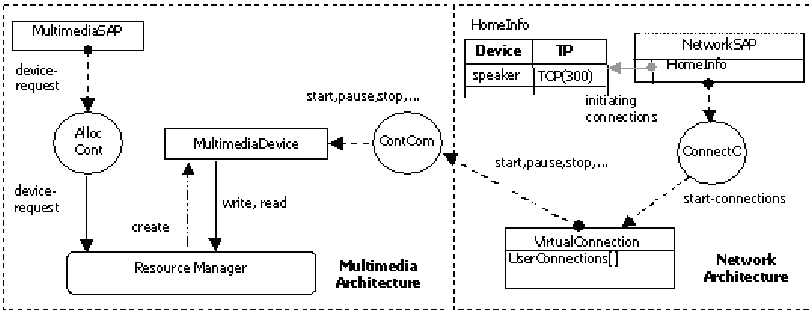

Figure 5: MultiTEL architecture for multimedia services

is very useful for implementing multimedia applications in which components evolve over time.

MultiTEL components and connectors are organized in three subsystems: the service subsystem that models the multimedia service logic; the multimedia subsystem that captures multimedia devices and establishes the connection among them; and the network subsystem that encapsulates the communication protocol and performs the transmission of data between participants. In this section, we focus on the multimedia and network subsystems.

The architecture of the multimedia and network subsystems contains the main components and connectors necessary to capture and transmit multimedia data [see Fig. 5]. `MultimediaDevice` components are atomically created by means of a `ResourceManager` as they are requested by an `AllocCont` connector that knows which devices have to be captured for each user in the service. In addition, for each multimedia device, the multimedia subsystem creates a `ContCom` connector to control the device. Once devices are captured, user connections are established through the `ConnectC` connector according to the service logic defined by the service architecture. After this, the system is ready to initiate the transmission and reception of multimedia data. During transmission, real-time media streams are captured from producer devices and are distributed to be reproduced by target devices. In MultiTEL, the communication protocol used to transmit the data is encapsulated in a `VirtualConnection` component. By changing the implementation of this component it is possible to adapt the communication protocol, e.g., RTP, UDP over IP multicast, without an impact on the other components and connectors in the application.

Once MultiTEL is described, we can illustrate how to use JMF to implement its components. It is not difficult to realize that it can be useful to implement both the `MultimediaDevice` and the `VirtualConnection` components [see Fig. 3 and 5].

4.1.1 JMF-based `MultimediaDevice` Implementation

In MultiTEL, the addition of a new device implies the implementation of a `Multi-mediaDevice` component. This component has to implement the `ProducerDevice` or the `ReceiverDevice` interfaces depending on whether we need to capture data from devices or render information. As an example, we report on how to implement a component called `JMFCamera` that allows to capture information from a new camera device. This component extends the MultiTEL `Component` base class and implements the `ProducerDevice` interface [see Fig. 6]. The main methods in this interface are the `setFormat()` and the `getFormat()` methods for configuring and consulting the format of the data to be captured, the `initialize()` and the `release()` methods for preparing the camera to transmit or to release it, the `startTransmission()` and the `stopTransmission()` methods to initiate and stop the capture of data, and, the `read()` method for reading video frames from the camera.

To use JMF to implement the `ProducerDevice` interface we encapsulate all the JMF objects needed to capture multimedia data into the `JMFCamera` component [see Fig. 6]. Note that we hide the complexity of JMF since it is encapsulated into the MultiTEL framework. Thus, software developers who use it just need to know a few things, for example, that a multimedia device has to be initialized before using it, that the format of the multimedia data captured can be changed or that multimedia streams can be read. They need not care of low-level and complex concepts such as data sources, processors, or media locators.

[Fig. 6] also shows part of the implementation of the `JMFCamera` component. Before using the camera it has to be initialized, which implies instantiating a set of classes [see Fig. 2]. The implementation of the `initialize()` method encapsulates the steps needed to capture a JMF device [see lines 1-8 in Fig. 6]. If the video format is known [see line 1], the JMF resource manager is then consulted to obtain the list of devices that support this format [see line 2]. Once the first such device is got [see line 3], a media locator is created [see line 4] to feed a data source [see line 5]. Then, a processor is created [see line 6] to process the data that the device captures, which is the input to a new data source [see line 7]. Finally, this data source instance is connected to the source described by the media locator [see line 8].

Following the MultiTEL compositional model, the `initialize()` method throws an `initialized()` event to notify that the device was initialized when it finishes. This event is captured by the `ContCom` connector [see Fig. 5], which changes its state according its coordination protocol. Since this connector controls how the device works, it waits in the new state until an event initiating the transmission is caught. On reception of this event, the connector sends the `startTransmission()` message to the `JMFCamera` component. Method `startTransmission()` [see Fig. 6] starts the transmission [see line 11] of the first stream of data captured [see line 10]. This method throws a `readyToSnd()` event [see line 12] that is caught by the `ContCom` connector. The data is captured from the device when the `ContCom` connector sends the `read()`

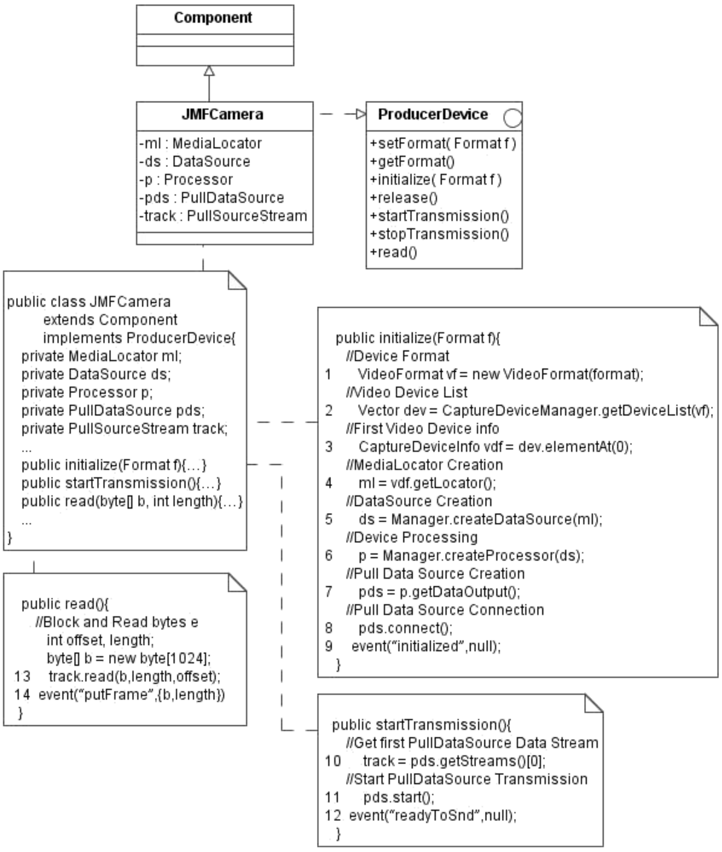

Figure 6: Implementing a JMF-based MultiTEL Camera

message to the component. This method blocks and reads an array of bytes from the data stream selected [see line 13]. Finally, the method throws the `putFrame(byte[])` event with the array of bytes as a parameter [see line 14]. Note that the MultiTEL framework specifies that media frames are captured as an array of bytes. This is the reason why we use the JMF `PullDataSource` class [see Fig. 6], which is a subclass of the `DataSource` class that gives us the required behavior by blocking and reading data from the device streams as byte array.

Once implemented, the new device has to be registered with the MultiTEL `ResourceManager`. The resource manager hides the device platform dependencies by providing a unique identifier and a common interface for each device type. [Fig. 7] shows how the `AllocCont` connector, which is part of the MultiTEL multimedia subsystem, asks the resource manager to capture devices by means of their unique identifiers ("camera" and "speaker"). Thus, multimedia applications need not know the actual

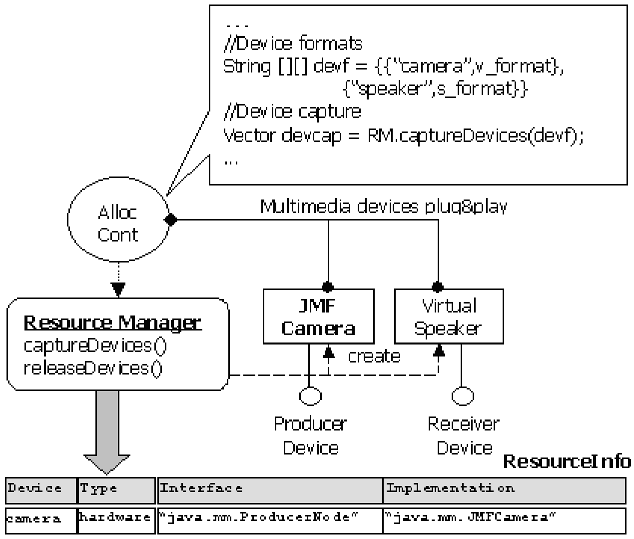

Figure 7: MultiTEL Resource Manager

implementation class; it only has to invoke the methods of the common interface implemented by the device component. Only the local resource manager knows the implementation class, which is stored in the `ResourceInfo` structure. Thus, the components that model the devices join the service at run time by means of a plug&play mechanism.

4.1.2 JMF/RTP-based `VirtualConnection` Implementation

It is not difficult to realize that the MultiTEL `VirtualConnection` component is similar to the `SessionManager` class in JMF [see Figs. 3 and 5]. The `VirtualConnection` component is local to each participant and encapsulates all of the connections that use the same communication protocol, similar to the JMF `SessionManager` class. The only difference is that the latter is specific to the RTP protocol, whereas the former is general enough to encapsulate any communication protocol. Anyhow, encapsulating the RTP protocol as a `VirtualConnection` component is not easy because the classes to capture streams and the classes to transmit it over RTP are highly coupled. That is to say, a `DataSource` that represents a device stream is an input parameter to the `createSendStream()` method in the `SessionManager` class, cf. the following snippet, which is part of the implementation of a MultiTEL `VirtualConnection` component to encapsulate the JMF `SessionManager` class:

```
// Create RTP Session
SessionManager rtpsm = new RTPSessionMgr();
...
// Create SendStream
```

```
SendStrem st = rtpsm.createSendStream(pds, 0);
```

where `pds` is an instance of the `PullDataSource` object created by a `JMFCamera` [see Fig. 6]. Since JMF does not offer a way to convert an array of bytes to a `DataSource` object, we need to adapt MultiTEL to use the JMF RTP API to implement the `VirtualConnection` component. Anyway, due to the separation of concerns between computation and coordination encouraged by the MultiTEL compositional model, changes required are located in a few components and connectors, namely:

- The implementation of the `JMFCamera` component needs to be changed so that frames are not represented as an array of bytes, but as a `DataSource` object. We emphasize that the `ProducerDevice` needs not to be changed, but the implementation of the component. The reason is that the frames that are captured are provided by throwing an event instead of returning them with the `read()` method. Thus, software developers just need to change the implementation of the `read()` method [see Fig. 6] to throw an event that has a parameter of type `DataSource` or `Object` instead of an array of bytes.

- The interface of the `ContCom` connector needs to be changed to be able to catch the new kind of events. This is easy since the coordination protocol that the connector encapsulates remains the same; only the type of the parameter of an event changes.

- Furthermore, the interface of the `VirtualConnection` component needs to be changed to receive a `DataSource` instance. This is needed because this component encapsulates an instance of the JMF `SessionManager` object that needs a DataSource to create the send stream.

Therefore, there are a number of situations in which JMF is not flexible enough to be completely and transparently encapsulated into COTS components, and some of its features need to be made visible. This results in an important drawback, since the (re)use of these components in different contexts decreases since they are JMF-dependent. Furthermore, after creating an RTP `SendStream` object from a data source, MultiTEL loses control over the transmission of data, which relies exclusively on the JMF classes. These problems arise because the interaction model defined by the JMF RTP API is incompatible with the interaction model of components and connectors in MultiTEL.

Despite these limitations, the integration of JMF with the multimedia and network architectures of MultiTEL results highly appropriate and entails important benefits for both MultiTEL and JMF, namely: MultiTEL benefits from using the RTP protocol, which needs not be implemented from scratch; JMF benefits from the services offered by the MultiTEL service architecture, which allows to implement a complete multimedia service, not just to capture and transmit multimedia data; furthermore, software developers can use JMF without the inconvenience of learning such a complex API since it has been componentized into the MultiTEL framework.

4.2 Combining Sametime and the CoopTEL Framework

CoopTEL [Pinto et al., 2001b] is a component-aspect middleware platform that derives from MultiTEL and separates other cross-cutting properties, in addition to the coordination aspect. CoopTEL implements a component-aspect model in which components and aspects are first-order entities that can be weaved at run time. Components encapsulate the functional behavior, and aspects model extra-functional features present along multiple components. Aspects can change or evolve independently from components. Components communicate by sending messages and/or throwing events, and aspects are evaluated during component communication. Another important feature is that the architecture of the applications is separated from the constituent components, which makes both components and aspects independent from each other and autonomous. Therefore, the information about connections between aspects and components is not hard-wired, but resides on a middleware layer. It also stores the data shared among several components, which is defined as application specific properties. Properties help define truly independent software components and aspects, putting any kind of data dependency as a shared property.

CoopTEL also defines an application framework for developing collaborative virtual environments (CVE) for the web. CVEs are distributed applications that integrate local and remote resources, i.e, users, documents, collaborative applications, in a shared space. The shared space of a CVE is divided into room components, which are containers of other CVE resources, e.g., documents or applications, and help organize the shared environment hierarchically, which provides a reference for user navigation. Each room component maintains a state that is retrieved by a persistence aspect when the component is instantiated, which occurs when a user enters the room. The application framework also provides several awareness components that supply information about which users are connected to the application and their location, i.e., in which room, or status, e.g., "idle", or "do not disturb". This information is very useful in a collaborative environment because we can easily locate or interact with other users.

The application framework defines an open-ended list of shared components, e.g., a clock, a database, or a software note. The components that model collaborative tools such as the audio or video conference component, a shared whiteboard component or a windows application sharing component were implemented using the Sametime Java toolkit. Next, we report on our integration of the Sametime API in CoopTEL.

4.3 Sametime-based Component Implementation

In CoopTEL, the addition of Sametime collaborative components implied wrapping different Sametime stand-alone applications as CoopTEL components. Our goal was to include the resulting CoopTEL components in CVE applications as resources of a room. We hope this can serve as a practical example of how to address componentization of coarse-grained multimedia frameworks.

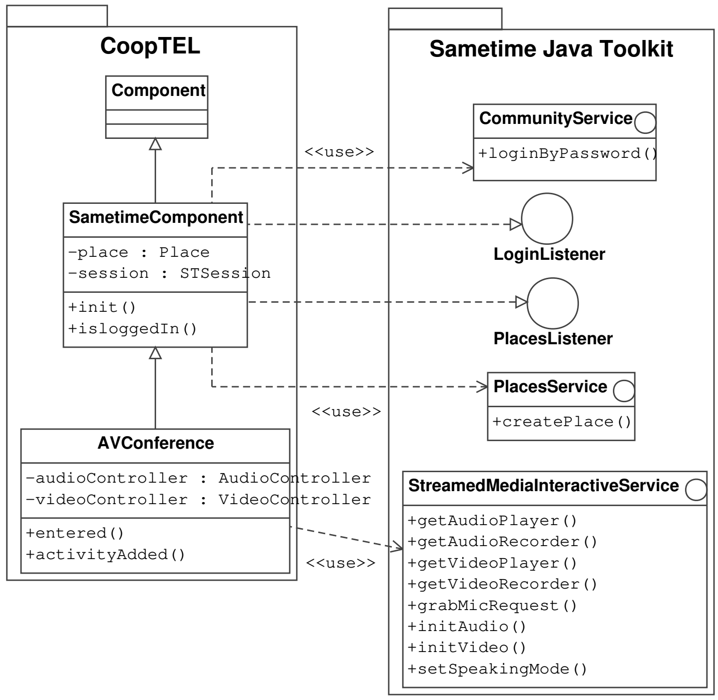

Figure 8: Implementation of a Sametime-based Component for Audio and Video Conference

During the design and implementation of these components, we realized that there are several steps regarding the Sametime server connection, initialization and service invocations that must be included by any Sametime-based application. This common functionality was finally included in a class called `SametimeComponent` that inherits from the `Component` class, which encapsulates the basic functionality that any CoopTEL component can offer [see Fig. 8]. The `SametimeComponent` constructor is in charge of creating a session in the Sametime server, which is encapsulated in a `STSession` object, and loading the components that allow using Sametime services. Since there can be more than one Sametime-based component running at a time and they have to share the same `STSession` object to access Sametime services, this object is stored in the platform as an application property. The first Sametime component that is created when the multimedia service is launched opens a session in the Sametime server and loads the Sametime service components. Immediately after this component stores the session object, successive Sametime-based components will get this object. Sametime component can log into a Sametime community and use the Community Service interface (`CommunityService`) by means of this session [see Fig. 8]. A client can log in by providing a user name and a password or anonymously; anyhow, user authentication cannot be considered part of the `SametimeComponent` functionality since it is modeled as an aspect. Thus, we use the Aspect Factory service provided by CoopTEL to wrap the user login as an aspect. After a user introduces his or her login and password, the `Login` aspect stores both strings as an application property; later the `SametimeComponent` component will retrieve this information from the platform. This way, we can configure a multimedia application to have authenticated access or not by simply adding or removing the `Login` aspect. Furthermore, both the `Login` aspect and the component to which it is applied can be (re)used separately since there are no dependencies between them. Other aspects that can be applied to this component are the awareness aspect that informs about new users, the access control list aspect that allows the owner of a collaborative application to define who can join it, the access control aspect that verifies if a user has permission to access an application, and the persistence aspect that stores and retrieves the state of a collaborative application.

To receive login events from the login service, the `SametimeComponent` class is registered as a login listener and implements the `LoginListener` interface. Since collaborative applications require the use of meeting services, this component creates and enters a virtual place in which collaborative activities such as shared whiteboard or live audio and video conference can be performed (it uses the `PlacesService` interface [see Fig. 8]). Again, the component is registered as a listener to receive events thrown by the `Places` service after being notified about a successful place creation.

When a CoopTEL component is created in the context of a room, it throws an event to notify remote instances of the same room that a new application has been started. The room component displays a notification component to inform the users in that room that a new collaborative application is running and inviting them to collaborate. For Sametime-based components, this event contains the identifier of the Sametime virtual place in which collaborative activities were added. If a user accepts the invitation to join this Sametime application, he or she will enter the room and will be able to participate in joint activities.

By encapsulating the common functionality of Sametime-based applications into a class we avoid rewriting it for every different Sametime-based collaborative component. Thus, to develop a new collaborative component we have to implement a class that inherits the SametimeComponent class and overrides the `entered()` method to add different collaborative activities to that place.

For the sake of illustration, we show how we implemented an audio/video conference inside a Sametime-based CoopTEL component. This component uses the Sametime `StreamedMediaInteractive` Service to access live audio and video inside a virtual place to provide audio and video conference to CVE rooms. This component adds audio and video activities to a Sametime virtual place, and uses the `StreamedMediaInteractive` interface to capture devices and retrieve audio and video streams controller objects. These controllers are used to get the graphical component that allows to reproduce and control the audio and video streams, e.g., pause or resume it, and to modify some device parameters, e.g., audio volume. Each controller object controls a player or a recorder device. A player device refers to a device used to reproduce remote multimedia data locally, e.g., a local sound card that plays audio into the local speakers; a recorder device refers to a device that can be used to capture local multimedia data, e.g., a local sound card from which local microphone input is gathered. The local recorder and player devices for audio and video streams are controlled by means of the `AudioController` and `VideoController` interfaces, which can be accessed through the `StreamedMediaInteractive` interface by invoking one of the following methods: `GetAudioRecorder()`, `GetAudioPlayer()`, `GetVideoRecorder()`, or `GetVideoPlayer()` [see Fig. 8]. Since the `StreamedMediaInteractive` service handles the explicit opening and closing of the audio and video device, we are only responsible for initializing the audio and video activities in a virtual place.

Once we get controllers for players and recorders for audio and video streams, we use the `initAudio()` and `initVideo()` methods provided by the `StreamedMediaInteractive` interface to initiate the transmission of streams. When the application finishes, it is needed to stop the audio and video devices by calling the appropriate method of the controllers. During the video conference transmission, the audio and video players reproduce the streams of a participant at a time. In Sametime, the multimedia streams that are being played locally are determined by the interaction mode selected, namely: in the fully interactive mode, any user with a microphone can speak at any time, whereas, in the conducted mode, a participant must request a turn before speaking. The `StreamedMediaInteractive` interface is also used to switch between these modes by the `setSpeakingMode()` method [see Fig. 8].

After implementing the `AVConference` component, we can include it as part of the application architecture of a CVE application. In addition, we can add some properties by means of aspects. Aspect evaluation is not hard-wired inside components; it is attached to component communication, creation and finalization. Apart from the `Login` aspect, we can define other extra-functional properties as aspects. For example, we could apply access control to component creation. We can model it as an aspect that encapsulates an access control list to control if a user has permission to start a collaborative application, or to join one that has been initiated. By using CoopTEL, we can add or remove this access control aspect to any component without changing it.

5 Conclusions

In this article, we have introduced a breakthrough approach to improve the development of multimedia services. Multimedia programming poses important challenges to developers. A multimedia programmer is supposed to have broad technical skills on low-level multimedia issues like capturing devices, negotiating formats or real-time protocols, which makes developing these services from scratch quite a difficult task. Furthermore, multimedia services have to deal with the continuous evolution of hardware devices and presentation environments, so multimedia APIs must provide extension mechanisms to upgrade final applications smoothly. Furthermore, the society demands highly configurable services able to adapt dynamically to user preferences or run-time environments. Unfortunately, as we have shown in this paper, current multimedia software does not address all these issues adequately.

We have presented the functionality and programming styles of several multimedia APIs and application frameworks. We have focused on describing and comparing JMF and Lotus Sametime, two successful object-oriented multimedia frameworks. After concluding that none of them addresses adequately all the challenges posed by multimedia software development, such as extensibility, interoperability, evolvability and usability, we proposed to apply component and framework technologies to cope with these "ilities". We have shown that it is possible to apply component-based software engineering and application framework technologies to the development of multimedia services, which improves their design. We believe that compositional framework technology is a good approach to construct more open, reusable, and extensible multimedia services easily. We have shown how to componentize the multimedia functionality that JMF and Sametime application frameworks provide. Thus, we have made it possible to (re)use the resulting components as COTS components inside two in-house multimedia frameworks called MultiTEL and CoopTEL. With this approach, multimedia services development can be undertaken from a higher abstraction level, without having to be an expert in multimedia low-level programming. Since multimedia devices and even collaborative applications can be added to the application as plug-ins, developers are able to modify or add new components without changing the others. Furthermore, since some properties are usually spread throughout different multimedia components, we model them as software aspects, according to aspect-oriented software development principles. Examples of these properties are coordination protocols, user authentication and access control. Since aspects can evolve independently of the components they affect, aspect- and component-based multimedia software development makes it possible to improve the modularity and the management of derived applications.

Acknowledgment

The work reported in this article was supported by the Spanish Ministry of Science and Technology under grant TIC2002-04309-C02-02.

References

[AOSD, 2004] AOSD (2004). *Aspect-Oriented Software Development Web Site*. `http://www.aosd.net`.

[Box, 1998] Box, D. (1998). *Essential COM*. Addison-Wesley.

[Brown and Wallnau, 1999] Brown, A. W. and Wallnau, K. (1999). The current state of CBSE. *IEEE Software*, 15(5):37–46.

[de Carmo, 1999] de Carmo, L. (1999). *Core Java Media Framework API*. Prentice Hall.

[Fayad and Schmidt, 1997] Fayad, M. E. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38.

[Fuentes and Troya, 1999] Fuentes, L. and Troya, J. M. (1999). A Java Framework for Web-based Multimedia and Collaborative Applications. *IEEE Internet Computing*, 3(2):55–64.

[Fuentes and Troya, 2001] Fuentes, L. and Troya, J. M. (2001). Coordinating Distributed Components on the Web: an Integrated Development Environment. *Software-Practice and Experience*, 31(3):209–233.

[Gonzalez, 2000] Gonzalez, R. (2000). Disciplining Multimedia. *IEEE Multimedia*, 7(3):72–78.

[IBM, 2003] IBM (2003). *IBM Lotus Instant Messaging and Web Conferencing (Sametime)*. `http://www.lotus.com/products/lotussametime.nsf/wdocs/homepage`.

[IBM, 2004] IBM (2004). *Sametime 3.1 Java Toolkit. Developer's Guide*. `http://www.lotus.com/sametimedevelopers`.

[Kiczales et al., 1997] Kiczales, G. et al. (1997). Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag.

[Krieger and Adler, 1998] Krieger, D. and Adler, R. M. (1998). The emergence of distributed component platforms. *The Computer Journal*, 41(3):43–53.

[Microsoft, 2004] Microsoft (2004). *Microsoft Windows Media Web Page*. `http://www.microsoft.com/windows/windowsmedia`.

[Pinto et al., 2001a] Pinto, M., Amor, M., Fuentes, L., and Troya, J. M. (2001a). Collaborative Virtual Environment Development: An Aspect-Oriented Approach. In *International Workshop on Distributed Dynamic Multiservice Architectures (DDMA'01)*, pages 97–102, Arizona. IEEE Computer Society Press.

[Pinto et al., 2001b] Pinto, M., Amor, M., Fuentes, L., and Troya, J. M. (2001b). Supporting Heterogeneous Users in Collaborative Virtual Enviroments Using AOP. In Batini, C., Giunchiglia, F., Giorgini, P., and Mecella, M., editors, *Proceedings of the Ninth International Conference on Cooperative Information Systems (CoopIS)*, volume 2172 of *Lecture Notes in Computer Science*, pages 226 – 238, Trento, Italy. Springer-Verlag.

[Schulzrinne, 1996] Schulzrinne, H. (1996). *RTP: A transport protocol for real-time applications*. RFC 1889.

[Silicon Graphics, 1996] Silicon Graphics (1996). *Digital Media Programming Guide*.

[Sun Microsystems, 2004a] Sun Microsystems (2004a). *Java Media API Web Page*. `http://java.sun.com/products/java-media`.

[Sun Microsystems, 2004b] Sun Microsystems (2004b). *The Java Media Framework Guide*. `http://java.sun.com/products/java-media/jmf`.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, second edition.