Proceedings of the 9th IASTED International Conference
Software Engineering and Applications (SEA 2008)
November 16 - 18, 2008 Orlando, Florida, USA

A USER-CENTRIC COMMUNICATION MIDDLEWARE FOR CVM

Yali Wu, Andrew A. Allen, Frank Hernandez, Yingbo Wang and Peter J. Clarke
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
email: {ywu001, aalle004, fhern006, ywang002, clarkep}@cis.fiu.edu

ABSTRACT

The advances in communication frameworks, such as Skype and Google Talk facilitate the increasing needs of communication-intensive and collaborative applications. These communication frameworks also make it possible for end-users to be more involved in the development of such applications if the appropriate level of abstraction can be provided.

In this paper, we propose the design of a user-centric communication middleware (UCM) that supports raising the level of abstraction appropriate for end-users to create and realize models using the communication virtual machine (CVM) technology. The CVM technology consists of the communication modeling language (CML) and CVM, and supports the rapid conception, construction and realization of new communication services using a model-driven approach. The UCM is a layer in CVM that provides operating simplicity to the end-user by masking the underlying technology. We present the design goals of UCM, high-level architecture, a description of the runtime environment and a case study showing how the communication needs of a medical scenario is realized in the UCM.

KEY WORDS
Abstraction, Middleware, Communication Services.

1 Introduction

The recent proliferation of multimedia communication frameworks, such as Skype [1] and Google Talk [2] is drawing people's attention from all aspects of society. They provide essential communication services including video conferencing, instance messaging,IP telephony, email, and file transfer. The access to these communication services has increased the need for end-users to be more involved in developing user-centric communication-intensive applications. *User-centric* means focusing on the benefits for the user and offering operating simplicity to mask the complexity of the underlying technology [3].

Deng et al.[4], developed a new technology, communication virtual machine (CVM) that raises the level of abstraction appropriate for end-users to create and realize communication models. The CVM technology consists of the communication modeling language (CML) [5] and CVM, and supports the rapid conception, construction and realization of new communication services using a model-driven approach [6]. The User-Centric Communication Middleware(UCM) is designed as one of the layered components of CVM to enforce communication requirements as captured by CML models as well as facilitating the realization of platform independent models (PIM) in CML into platform specific models(PSM) related to particular frameworks. UCM extends traditional middleware by encapsulating communication services into a self-contained communication control script, which is executed by invoking the APIs provided by the network communication broker (NCB) in the CVM.

In this paper we describe how the UCM supports raising the level of abstraction in the CVM and enables the rapid realization of communication-intensive applications. The specific contributions of the paper are as follows:

1. The mechanisms of the control scripts and macros used by UCM to support raising the level of abstraction are described.
2. The architecture of the UCM and the runtime environment, including algorithms to generate and execute the macros, are presented.
3. A case study describing the realization of the communication services for a medical scenario is described.

In the next section we present background on communication middlewares and CVM. Section 3 describes high-level design goals of UCM. Sections 4 and 5 describe the control scripts and macros, and UCM architecture. Section 6 explains the execution environment. Section 7 presents the case study for a medical scenario. Section 8 describes the related work and we conclude in Section 9.

2 Background

In this section we describe concepts related to middleware and give an overview of the communication virtual machine (CVM).

2.1 Communication Middleware

The middleware approach has emerged as a promising solution for heterogeniety and distribution problems in the design of complex distributed systems [7, 8]. It provides

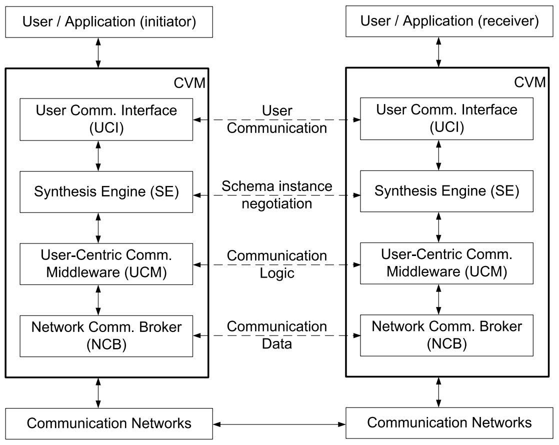
User / Application (initiator)
User / Application (receiver)
CVM
CVM
User Comm. Interface (UCI)
User Communication
Synthesis Engine (SE)
Schema instance negotiation
User-Centric Comm. Middleware (UCM)
Communication Logic
Network Comm. Broker (NCB)
Communication Data
Communication Networks
Communication Networks

Figure 1. Layered Architecture of the CVM.

standard programming interfaces and protocols to the application layer, enabling applications to interact with other applications or services transparently in the distributed environment. The distributed programming models all perform synchronization, marshalling, mapping data representation, and network communication.

There is a special class of middleware that focuses on multimedia communications that is more suited to the work presented in this paper. This type of middleware exhibits special features needed for multimedia communication. These features include: event notifications, logging, multimedia streaming, persistence, security, fault tolerance and distributed concurrency control [9].

2.2 Communication Virtual Machine

The CVM technology [4] consists of a communication modeling language (CML) [5] and CVM. The CVM technology supports rapid conception, construction and realization of new communication services using a model-driven approach. CML supports the modeling of communication-intensive applications by domain end-users. The models created using CML are referred to as *communication schemas* and represent the configuration of the participants in the communication and the media (or media structures - *forms*) to be exchanged between the participants. Forms allow individual media to have actions and constraints that can be applied during realization.

CVM uses a layered architecture to support the realization of communication applications. Different concerns of this process are separated and encapsulated into self-contained components with well-defined interfaces and responsibility. The principal separation of concern provides the basis for automation and flexibility. The major layers of abstraction in the CVM architecture are:

- *User communication interface* (UCI), provides a language environment for users to specify their communication requirements in CML, as a declarative PIM
- *Synthesis engine* (SE), is a suite of algorithms to automatically synthesize a user communication schema instance to an executable form called a communication control script, which is an imperative PIM
- *User-centric communication middleware* (UCM), executes the communication control script to manage and coordinate the delivery of communication services independently of underlying network configurations, acting as a bridge from PIM to PSM
- *Network communication broker* (NCB), provides a network-independent API to UCM and works with the underlying network protocols to deliver the communication services [10], directly interfacing with PSM

3 Design Goals for UCM

While UCM has the core concepts of traditional middleware, its role in facilitating the development of communication intensive applications in CVM requires the middleware be tuned for a model driven approach. Specifically, UCM should aid the transformation of declative PIM into PSM. In this section, we will investigate major design goals of UCM that realize the vision of a transformative communication middleware.

Raising the level of abstraction : Middleware at its most basic is about solving heterogeneity and distribution [7]. Specifically in the context of model-driven development, automation of model realization as a software application requires a middleware framework that masks the complexity of the underlying infrastructure software (PSM models). UCM must solve these problems in the context of the model-driven approach used by CVM [4]. It therefore must provide a less complex API to SE and equally important, an API that facilitates ease in transformation from PIM to PSM. These challenges result in the first goal of UCM : the provision of higher level service abstractions that provide both rich and easy-to-use interfaces to ease the realization of communication services.

Flexible and Extensible Functionality : To render flexible and extensible functionality of the communication middleware, it is important to separate interface and implementation of communication services through a modular design. For our design to be good, we need to clearly separate the interfaces exposed to the service requester, in this case the Synthesis Engine, with the logic that implements these interfaces, which we will introduce later as *macros*. This approach ensures the ease of change and adaptation in satisfying application request.

Fault tolerance capabilities : In case of an unexpected situation, a reliable middleware framework should not 'crash' the runtime engine but rather generate exceptions. With that in mind, exception handlers are required by the UCM which would listen for any abnormal behaviors in the runtime of the service realization processes and perform exception recovery and resolution mechanisms accordingly.

Table 1. Supported Control Script Command.

Script Command	Parameter Name	Parameter Type	Purpose
createConnections	connectionID	String	Create a local session and map the session to the designated conection
sendSchema	connectionID, senderID, recipientList, control-schema, data-schema	String, String, List String, String	Send schemas across for the purpose of schema negotiation
addParticipants	connectionID, participantList	String, List	Add the negotiated participants into the established connection
enableMedia	connectionID, mediaName	String, String	Start streaming media, such as live audio
sendMedia	connectionID, mediaName, mediaURL	String, String, String	Send a nonstream mediuam such as a file
sendForm	connectionID, formID, mediumList	String, String, List	Send a form which is a complex data consisting of multiple simple medium

Network independent execution : To avoid the communication middleware being closely tied to any particular network protocols and platforms, the design of UCM should not be limited to communication platforms like Skype API, but instead generate a set of network independent method invocations that could run on multiple platforms. Therefore, a broker facility is needed between UCM and network platforms to support network independent execution. The Network Communication Broker (NCB) in the CVM architecture provides us such a facility, which dynamically selects and configures the underlying platform based on application needs and platform capabilities[11].

4 Control Scripts and Macros

In this section, the mechanisms of control scripts and macros used by UCM are discussed to show how it raises the level of abstraction in CVM to facilitate rapid realization of communication-intensive applications.

4.1 Communication Control Script

Following the demand for a higher level of abstraction than full APIs provided by traditional middleware frameworks, the UCM accepts a communication control script that represents network-independent control logic for user-level communication sessions. This high level control script is concerned with the general primitives of establishing communication such as connection creation, adding or removing participants and media transmission. It also incorporates application specific issues of communication services, such as the enforcement of user defined policies, communication constraints (e.g., if bandwidth is low then substitute video with images) or security properties (e.g. encrypt all patient data). The high level control script facilitates model transformations in CVM in that declarative communication models could be synthesized into the script in the synthesis engine (SE), converted into the appropriate macros and then executed in the UCM.

Syntactically, a control script from SE to UCM consists of one or more script commands, each command similar to a method call in the form of *commandName(paramList)*. Each script command encapsulates a piece of self-contained functionality. Table 1 shows a partial list of script commands that are currently implemented in UCM. The four columns in Table 1 consist of the script commands, parameter names, parameter types and a short description of each command.

4.2 Macros

Macros are programming language statements that, when processed, generate a sequence of more detailed language statements. We view a macro here as the basic atomic unit for the execution of a control script, representing a mapping from a script command to a method-like template that contains the detailed execution steps for realizing a single communication service. Syntactically, we define a macro as follows: *macro (name:string, returnType:string, paramTypeList:string, paramNameList:string, script:string)* where *name* is the unique name of the macro, *returnType* is the type of the object returned by the macro, *paramTypeList* is the list of parameter types that are passed as arguments to the macro, *paramNameList* is the list of parameter names, and *script* is the source code containing the functionality of the macro. Macros encapsulate not only general application-specific communication functionalities, but also customized event and exception handling mechanisms. As new exceptions occur with newly added functionalities, corresponding exception handling macros could be added to the repository for completeness.

To support the operational semantics of the control script and increase the efficiency of the UCM it is required that the macros provide a mechanism to support synchronization. To resolve the synchronization issues with macro execution, macros can be tagged as either blocking or nonblocking. Depending on the type of macros, the script interpreter will either block that connection for a target event, or execute the macro in a non-blocking manner. Details of macro synchonrization would be explained in Section 6.

Macros provide a way of extending functionalities and adapting to new QoS requirements in a faster manner. They could be developed, maintained, and stored in the repository offline, or loaded and instantiated at runtime . An example of a macro is shown in Table 2.

5 UCM Architecture

In our design of UCM, we used the microkernal and repository architectural patterns to guide the process of subsystem decomposition. The resulting architectural diagram is shown in Figure 2. We identified several components that together form a self-contained execution platform for an

Table 2. Macro for createConnections Script.

Name	createConnections
paramNameList	connectID
paramTypeList	java.lang.String
returnType	cvm.ucm.handlers.exception.ConnectIDException
Exceptions	cvm.ucm.handlers.exception.ConnectIDException
script	`import static java.lang.String` `import static cvm.ucm.handlers.` `exception.ConnectIDException;` `if(connectID == null) {` `ConnectIDException exception = new` `ConnectIDException();` `return exception; }` `ncb.createSession(connectID);` `return null;`

incoming communication control script. In this subsection, we will explain the functionality for each component:

1. *SE-UCM Interface* - Exposing the functionality of UCM to the upper layer SE. Through this interface, application specific services could be realized by simply generating a high level control script without knowing about details of service realization process.
2. *UCM Manager* - Coordinating the activities of UCM. It dispatches the control script to the script interpreter and keeps track of the status for each uncompleted script. It is also responsible for notifying the SE of all SE events and signaling the script interpreter regarding the status update for connections.
3. *Script Interpreter* - Parsing and interpreting the control script by loading corresponding macros from the repository and handling execution runtime to realize the communication described in the control script.
4. *Repository Manager* - Facade to the repository subsystem, which store macros that define detailed execution logic for the control script, as well as other runtime information.
5. *Exception Handler* - Deciding how to act on exceptions received due to control script faults, or bad function call returns. This ensures the fault tolerance of the framework by avoiding system crashes in case of an unexpected failure.
6. *Event Handler* - Coordinating and orchestrating the events raised by NCB as well as deciding what to do in each case. Due to the different types of events, different event handling mechanisms will be applied.
7. *UCM-NCB Interface* - Interfacing with the NCB layer for the management of network sessions. This ensures the network independent execution of the control script, due to NCB's interoperability with multiple available communication frameworks.

6 UCM Execution Environment

As an execution environment within CVM, UCM is designed to perform a series of service processes that provide a framework for the execution of communication control scripts, including system initialization, macro loading and interpretation, exception and event handling and runtime media management In this section, we present details for three of the major runtime system activities.

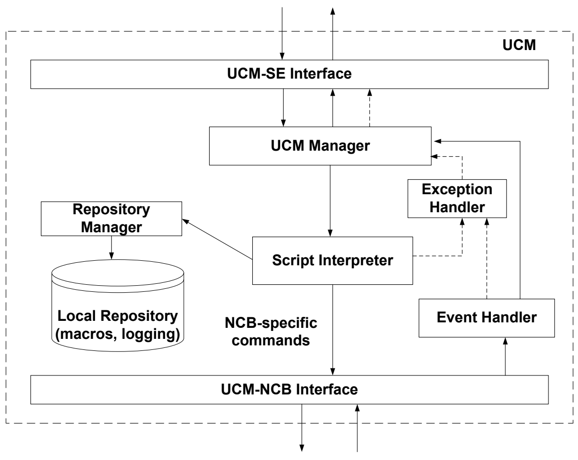

Figure 2. Block Diagram for UCM.

```
1: Build Executable Macro List
   /* Input: Control Script - consist of one or more script commands*/
2: for all script command i in Control Script do
3:    load macro i
4:    substitute values in script command i in macro i parameters
5:    append macro i to executable macro list
6:    if macro i is of type BLOCKING then
7:       append macro SLEEP_CONNECTION to executable macro list
8:       append macro HANDLE_SIGNAL to executable macro list
9:    end if
10: end for
```

Figure 3. Algorithm to Build Executable Macro List.

Dynamic loading and execution: Dynamic loading means that macros are only loaded on demand, like the way class loaders work in the Java Runtime Environment. Upon receipt of a control script, the UCM manager delegates the execution request to the script interpreter. A control script includes one or more script commands, each of these script commands is evaluated and their execution done in a sequential manner. Script interpretation can be viewed as a two phase process, the first phase involving dynamic loading of the macro and the second one involving macro execution.

The loading process, see Figure 3, parses the control script and for each script command the corresponding macro is loaded from the repository, instantiated with actual parameters, then appended to the execution macro list(EML). As discussed in Section 4.2, a macro can be tagged as blocking indicating that some event generated by the lower layers as an effect of its execution must be received before continuing the execution sequence of the executable macro list. As such two additional system macros are appended after each blocking macro in the EML, see Figure 3 step 6 to 9, to ensure that further processing of commands for the specific connection will wait while freeing the interpreter to process other connections' requests.

The `SLEEP_CONNECTION` macro is a special flag macro to put the EML for a specific connection to sleep. As Figure 4 steps 3 to 7 shows, it will increment the counter to the next macro to be executed in the EML, signals to the UCM Manager to update the status of the queue for the specific connection to "blocking" and then saves the EML to the repository. The execution of this specific EML is terminated and the interpreter is made available to other EMLs. The receipt of the event will have the interpreter load the EML from the repository and the event processed, as in steps 9 to 12 in the case of a `HANDLE_SINGAL` macro, another flag macro for handling received events. A "successful" event allows the EML to continue its execution sequence. The macro definition for `createConnections` is shown in Table 2. As seen, similar to a method invocation, the source code contained in the macro is executed with the values of the parameters replaced by actual values.

Runtime Media Management: During the process of media transmission, certain media management information needs to be maintained beyond the script runtime for the purpose of handling media requests in a controlled manner. This includes the location of the media, parties that are authorized to share it, and possibly other constraints on the media. This runtime media information could be saved in a media mapping table stored in the repository. At runtime, when a macro like `sendForm` finishes execution, this information is saved in the medium table, and possibly retrieved later to handle a remote media request. In this way, runtime media delivery are controlled and enforced in UCM.

Exception handling : To provide a flexible way of catching unexpected situations as well as predictable errors, we design the exception handler in such a way that besides the general exception handling, customized exception resolution mechanisms could be added offline and loaded at runtime. Macro developers could define new exception handling mechanisms and store them as macros in the repository. As a runtime exception occurs, the exception handler will handle it accordingly if it is a recognizable exception, otherwise it would delegate it to the manager to load appropriate macro for exception handling, as in the case of a customized exception.

```
1: Execute Macro List
   /*Input: Executable Macro List(EML) - consist of one or more macros
   current_macro - macro to be executed
   i.ConID - Connection Identifier for control script request */
2: for all macro i in EML do
3:   if macro i is a SLEEP_CONNECTION macro then
4:     set current_macro to macro i+1 in EML
5:     signal UCM Manager to block new scripts from macro i.ConID
6:     save EML to repository
7:     terminate execution of EML
8:   else if macro i is a HANDLE_SIGNAL macro then
9:     if event type SUCCESSFUL then
10:      set current_macro to macro i+1 in EML
11:    else
12:      generate an exception for the event
13:    end if
14:  else
15:    run macro i // e.g Table 1
16:    set current_macro to macro i+1 in EML
17:  end if
18: end for
```

Figure 4. Algorithm to Execute Executable Macro List.

7 Case Study

In this section, we describe a case study in the healthcare domain to show the role of UCM in the development of user-centric communication applications. We will first introduce a communication scenario involving a post surgery conference, and then walk through the process of realizing it in UCM. Future work in extending the macro facility and self testing of control scripts are also discussed.

7.1 Communication Scenario

Scenario : Patient John Demo has been referred by Dr. Sanchez (a heart specialist) for heart surgery. Dr. Burke performs the surgery on John. After surgery, Dr. Burke contacts the referring doctor, Dr. Sanchez, and the attending physician, Dr. Monteiro, to let them know that the surgery went well, and to share several aspects of John's medical record with them, including the post-surgery echocardiogram (echo) and images of the patient' heart captured during the surgery and a text summary of the patient record. Dr. Burke also needs to outline the post surgery care for John that should be followed by Dr. Monteiro later. Given the echocardiagram and image are large, they are only sent to the doctors when they request them.

7.2 Realizing Scenario in UCM

The realization of this healthcare scenario starts from capturing the communication needs of this application using CML and synthesizing the CML instance into a communication control script that UCM accepts and executes. For the purpose of post-surgery conference, the following scripts are generated on Dr. Burke's side:

```
1. createConnections("c1");
2. sendSchema("c1","Dr.Burke","Dr.Sanchez,
   Dr.Monteiro", controlSchema,dataSchema);
3. addParticipants("c1","Dr.Sanchez,
   Dr.Monteiro");
4. enableMedia("c1", "LiveAudio");
5. sendForm("c1","patientRecord866",
   (("summary",".../sum866.doc","sendNow"),
   ("echo",".../echo866.gif","sendOnDmd"),
   ("scan",".../scan866.mpg","sendOnDmd")));
6. sendMedia ( "c1", "postSurgeryCare",
   "www.cs.fiu.edu/john_postcare.doc");
```

We currently have a preliminary prototype of UCM, which could demonstrate the realization of communication services through UCM. We use MS Access as the local repository, with the macros stored in tables. However, we haven't yet performed a complete evaluation in terms of the performance benefits of the UCM approach. Future evaluation involves the comparison of script execution time of hard-coded macros versus dynamically loaded macros; the comparison of the efforts of changing the execution logic of

CML models directly in SE versus changing related macros without touching system code. Meanwhile, the research aspects of UCM leave open several issues such as:

Extension of macros: Currently, the functionality of macros are limited to general communication services. However, the extensible framework allows us to incorporate customized communication requirements for specific communication applications. For example, future macros could capture communication constraints like role-based data transmission like doctors could only send patient discharge form to the nurse, as well as security and privacy properties based on HIPAA in the healthcare domain. Moreover, communication workflow such as referring doctors need to send the patient discharge form to the nurse for validation before sending to the discharge physician, could also be captured and enforced by the macros in the UCM.

Ability to self-test UCM : Another interesting point with the macro facility is its self-testing ability. Macros that test the UCM could be developed, therefore providing us with a systematic approach to test these components during initialization and during adaptation. With the extension of control scripts as well as frequent changes of the network service provided, manual testing of UCM would be more tedious and error-prone, and this makes self-testing a rewarding feature worthy of future research.

8 Related Work

Domain specific middleware tailored for multimedia communication services are extensively researched in the community, for providing flexible, customized communication functionality. [12] proposed an end-system communication middleware called Da CaPo++ for multimedia applications. While it has similar purpose to our work in terms of a high-level API and efficient runtime support, it targets supporting applications in terms of flexible protocol selection and QoS support, not as a communication middleware handling user centric sessions or facilitating model driven development of commmunication services, as UCM does in the context of CVM.

In [13], Narnia is proposed as a middleware that helps programmers build distributed applications. It uses a collection of programming abstractions to support the creation of communication services and the Narnia virtual machine as an event based run-time environment. Narnia is similar to our work but more focused on supporting application servers that must respond to high server loads, with efficient event addressing schemes while UCM is more concerned with enforcing the user's communication requirements specified in CML.

9 Conclusion

In this paper we presented the design of a user-centric communication middleware (UCM) to support the rapid realization of communication services in the communication virtual machine (CVM). The notion of control scripts, macros and the supporting UCM architecture are presented to show how it raises the level of abstraction than traditional middleware in realizing application-specific communication services. The execution environment provided by UCM is also elaborated in a detailed fashion. Future work involves evaluation of UCM, the extension of macros as well as the self-testing ability of these macros.

Acknowledgement

This work was supported in part by the National Science Foundation under grant HRD-0317692.

References

[1] Skype Limited. Skype developer zone, Feb. 2007. `https://developer.skype.com/`.

[2] Google. Google talk, September 2007. `http://www.google.com/talk/`.

[3] Philippe Lasserre and Dennis Kan. User-centric interactions beyond communications. Alcatel Telecommunications Review, 2005. `http://alcaesd-f.nl.francenet.fr/docs/1/S0503-UCBB_interactions-EN.pdf`.

[4] Yi Deng, S. Masoud Sadjadi, Peter J. Clarke, Chi Zhang, Vagelis Hristidis, Raju Rangaswami, and Nagarajan Prabakar. A communication virtual machine. In *Proceeding of COMPSAC 06*, pages 521–531. IEEE Computer Society, 2006.

[5] Peter J. Clarke, Vagelis Hristidis, Yingbo Wang, Nagarajan Prabakar, and Yi Deng. A declarative approach for specifying user-centric communication. In *Proceeding of CTS 2006*, pages 89 – 98. IEEE.

[6] Object Management Group. Omg model driven architecture, Oct. 2008. `http://www.omg.org/mda/`.

[7] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996.

[8] Wolfgang Emmerich, Mikio Aoyama, and Joe Sventek. The impact of research on middleware technology. *SIGOPS Oper. Syst. Rev.*, 41(1):89–112, 2007.

[9] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, 2002.

[10] Chi Zhang, S. Masoud Sadjadi, Weixiang Sun, Raju Rangaswami, and Yi Deng. A user-centric network communication broker for multimedia collaborative computing. In *Proceedings of CollaborateCom 2006*, pages 28–32. IEEE Computer Society, November 2006.

[11] Andrew A. Allen, Sean Leslie, Yali Wu, Peter J. Clarke, and Ricardo Tirado. Self-configuring user-centric communication services. In *Third International Conference on Systems (icons 2008)*, pages 253–259. IEEE, April 2008.

[12] Burkhard Stiller, Christina Class, Marcel Waldvogel, Germano Caronni, and Daniel Bauer. A flexible middleware for multimedia communication: Design, implementation, and experience. *IEEE Journal on Selected Areas in Communications*, 17(9):1614–1631, September 1999.

[13] Mauricio Cortes and J. Robert Ensor. Narnia: A virtual machine for multimedia communication services. In *MSE '02: Proceedings of the Fourth IEEE International Symposium on Multimedia Software Engineering*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.