A Communication Virtual Machine

Yi Deng, S. Masoud Sadjadi, Peter J. Clarke, Chi Zhang,
Vagelis Hristidis, Raju Rangaswami, and Nagarajan Prabakar
School of Computing and Information Sciences
Florida International University
11200 SW 8th St., Miami FL 33199
{deng, sadjadi, clarkep, czhang, vagelis, raju, prabu}@cs.fiu.edu

Abstract

The convergence of data, voice and multimedia communication over digital networks, coupled with continuous improvement in network capacity and reliability has significantly enriched the ways we communicate. However, the stovepipe approach used to develop today's communication applications and tools results in rigid technology, limited utility, lengthy and costly development cycle, difficulty in integration, and hinders innovation. In this paper, we present a fundamentally different approach, which we call Communication Virtual Machine (CVM) to address these problems. CVM provides a user-centric, model-driven approach for conceiving, synthesizing and delivering communication solutions across application domains. We argue that CVM represents a far more effective paradigm for engineering communication solutions. The concept, architecture, modeling language, prototypical design and implementation of CVM are discussed.

Keywords: model driven, communication application, multimedia, middleware, telemedicine.

1. Introduction

Communication is the most fundamental function of business, government and society at large. In recent years, the convergence of data, voice, and multimedia over digital networks coupled with the continuous improvement in network capacity and reliability has enabled a wide range of communication applications. Examples range from general-purpose communication applications such as VoIP telephony, voice, video or multimedia conferencing to specialized applications such as disaster management and telemedicine. The pace of innovation of new communication applications will undoubtedly accelerate further as both capacity and demand increase.

When we exam this trend a little further, however, it reveals several major issues. First, today's communication tools are developed in stovepipe fashion with limited separation between application needs and logic, device types and underlying networks. The combined complexity of these aspects results in high cost and a lengthy development cycle. Second, such vertically developed systems typically have fixed functionality and interface and do not interoperate with each other (because of differences in design, architecture, API, and network/device assumptions). It is difficult to adapt the systems to fit changing user needs, the dynamics of underlying networks, and new device and network technologies [17]. Users, particularly sophisticated domain specific users, are forced to hop between tools to satisfy their communication needs. Third, the fragmented development approach poses major challenges in integration and in providing integrated communication solutions. Last but not the least, it hinders the development of new communication tools, particularly for domain specific applications (e.g., telemedicine), because of the complexity, cost, and lengthy cycle required of vertical development.

In this paper, we present a fundamentally different approach for engineering communication solutions. This approach, which we call *Communication Virtual Machine (CVM)*, represents a paradigm shift on how a communication application is conceived and delivered. We argue that the CVM approach provides the basis to effectively address the problems discussed above.

The design of CVM draws from the concepts of model-driven engineering [1,25], communication middleware [23] and middleware-based architecture [24]. However, by focusing on the communication domain only, CVM achieves high degree of automation and effectiveness, and avoids the pitfalls of many general purpose methods and techniques for model-driven engineering that are overreaching and consequently ineffective. Furthermore, the CVM approach goes far beyond the goals of communication middleware towards end-to-end communication solutions. (See Section 7 for more in depth discussion). As opposed to stovepipe development, which will inevitably leads to repetitive and incompatible designs, and costly, lengthy and error-prone developments,

Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)
0-7695-2655-1/06 $20.00 © 2006 IEEE

CVM provides a model-driven process for conceiving, structuring, synthesizing and delivering communications that are tailor-made for user or application needs. In CVM, general purpose or domain specific communication needs are specified in a model, called communication schema, independent of device types and underlying network configuration. Such a model is instantiated, negotiated, synthesized and executed, by a fully automated process, to satisfy the users' communication needs. Since even a sophisticated communication model can be built in terms of hours or days, rather than months or years needed for designing and implementing a major communication application (e.g., telemedicine) CVM provides an effective way to support user-centric on-demand communications. (As discussed later, the on-demand feature of the CVM is also reflected by its ability to change the communication model at run-time.)

This model-driven communication is supported by the CVM layered architecture (Section 3). These layers are common to and shared by different communication applications. This architecture separates and encapsulates major concerns of communication modeling, synthesis, coordination, and the actual delivery of the communication by the underlying network and devices, into self-contained compartments with clear interface and responsibility. We will show that this architectural principle of separation of concerns employed in CVM is the basis for its automation and flexibility. This is because system components and communication protocols common to different applications can be identified and shared without having to be hard coded into a stovepipe system. This will also enable the CVM architecture to be independent of the underlying networking infrastructure and communication devices.

Coupled with the CVM architecture are several major components, together forming the CVM system (Section 4): A communication modeling language providing an intuitive graphic form for users (or user organizations) to model declaratively their communication requirements; a synthesis engine responsible for negotiating and synthesizing user communication sessions; a communication engine for executing user communication logic; and network communication broker to interfacing with the underlying network infrastructure.

A prototypical design of the CVM is implemented and fully functional (Section 5 and 6). In addition to supporting general purpose communication functions (e.g., multimedia conferencing), we have worked with physicians and technical staff at the Miami Children's Hospital (MCH) and the Teges Corporation, which supplies MCH with its patient medical information system called *i-Rounds*©. We have conducted case studies of using the CVM to support communication between doctors involved in cardiovascular operations based on scenarios and criteria formulated by the doctors. We have demonstrated that and it took less than a day to discuss, formulate and structure the telemedicine scenarios into CVM communication schemas; and it took two graduate students in a week to integrate the CVM system with the i-Rounds patient information systems, which transforms a general purpose CVM into a communication platform capable of supporting a variety of telemedicine communications with structured exchange of patient information.

2. A Motivating Example

Let us consider the following scenario: *Eric is a general practitioner who is examining one of his patients. He observes an unusual symptom and decides to call Mary, who is a specialist. During their conversation, Mary calls John, who is a researcher working in a medical laboratory, and asks him to join the call. This turns the two-way call into a conference call. Eric then decides to share parts of his patient's record with Mary and John and show them some related images. This turns the voice conference into a multimedia telemedicine application.*

Clearly, carrying out this scenario is possible with today's technology. For instance, Eric would first place a phone call to reach Mary. Next, assuming Mary's phone has conferencing capability, she switches to a conference call to include John in a three-way conversation. Otherwise, they have to use a conferencing application such as Yahoo! Messenger. Eric would then use a separate custom developed telemedicine application for sharing the patient's record with Mary and John. In case either Mary or John does not have access to such a custom application, Eric may need to send the images via email or a file sharing application. In general, although such scenarios can be accommodated with today's technology, the users would either have to jump between different tools (e.g., phone, email, file-sharing, and messenger application), or to rely on custom-developed applications, which are typically expensive and rigidly designed.

In the following sections, we show how this scenario, as well as any other similar scenarios, can be satisfied on-demand and with ease using CVM.

3. Communication Virtual Machine

To better understand CVM, let's first look to the data management area. Figure 1 captures the parallelism between the CVM paradigm and the one for managing heterogeneous data sources. As the use of computers and databases increases, we reach to a

point where data is dispersed to a large number of sources with different data models (e.g., relational, XML, text, and so on), data schemas and querying interfaces (e.g., SQL, XQuery, and keyword search). However, applications require access to multiple of these data sources and it became clear that developing specialized access mechanisms for each data source is cumbersome and inflexible since, if a legacy database is replaced by a newer one, the application has to be changed. Hence, the logical data abstraction paradigm was proposed, to hide the specifics of the underlying sources and export a uniform interface to the applications for querying data. The right part of Figure 1 shows a popular mediator architecture [7], where XQuery (See www.w3.org/XML/Query/) is used as the common data extraction language.

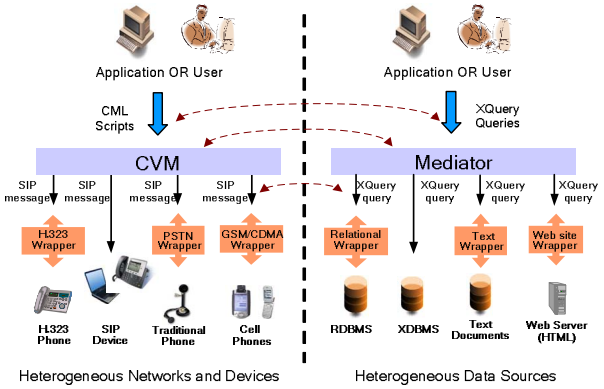


Figure 1: Analogy between CVM and Data Mediator.

In the communication domain there is a similar need to hide the underlying device and network infrastructure and provide a unified communication abstraction layer. The CVM plays the role of the mediator and it handles the execution of communication requests specified in CML (which corresponds to XQuery. The wrappers on the left and right side of Figure 1 play the role of abstracting the network/device and data specifics, respectively. Finally, the SIP messages (See www.iptel.org) play the same role as break-down XQuery queries. Notice that the arrows between the left and right parts depict the correspondences between the two paradigms.

In this section, we present a CVM architectural model for achieving the vision discussed earlier. There are four major tasks need to be performed to serve the users' communication needs:

(1) Conceive and describe the users' communication requirements. In the case of a multimedia conferencing, it is to specify who the participants of the conference are and what kind of media or data are to be exchanged. In the case of the telemedicine application, it also includes the policy that governs who can access which part(s) of the patient's medical record.
(2) Transform the user communication requirements into a sequence of commands or actions, which when executed will control the flow of user communication as dictated by the requirements.
(3) Provide a platform or environment in which the said sequence of commands can be executed to regulate the flow of communication.
(4) Deliver the media or data among the communicating parties through a communication network or networks.

Today, these tasks are typically hard coded in a communication system or tool, which predefines the way that user will use the system. Such a stovepipe design is the root cause of the problems discussed in Section 1. At the heart of CVM is a layered architecture, which provides a clean separation and compartmentalization of these major concerns in the spirit of [4], as illustrated in Figure 2. The CVM architecture divides the major communication tasks into four major levels of abstraction, which correspond to the four key components of CVM:

(1) ***user communication interface (UCI)***, which provides a language environment for users to specify their communication requirements in the form of a *user communication schema or schema instance*[1].
(2) ***synthesis engine (SE)***, which is a suite of algorithms to automatically synthesize a user communication schema instance to an executable form called *communication control script*;
(3) ***user-centric communication middleware (UCM)***, which executes the communication control script to manage and coordinate the delivery of communication services to users, independent of the underlying network configuration; and
(4) ***network communication broker (NCB)***, which provides a network-independent API to UCM and works with the underlying network protocols to deliver the communication services.

This layered division of responsibility is reminiscent of the OSI layered stack model for network communication [10]. Each layer has a specific role in the stack and communicates logically with the peer-layer at a remote site during communication sessions. Each layer builds on the upper layers in the stack to finally realize the user communication schema.

UCI is responsible for providing users with means to define and manage their communication schema, which describes the role of communicating parties and

[1] A schema is a generic model of communication; an instance is an instantiation of the schema for a particular communication session. We will use the terms interchangeably until Section 4, where communication modeling is discussed.

the communication logic (e.g., participants, flow of data or information). For this purpose, a communication modeling language is needed [8]. Such a language (see Section 5) should be intuitive enough to support on-the-fly communication modeling without requiring knowledge of underlying networks and yet rich enough to describe a variety of communication tasks. The language design poses many interesting research issues in its own right. In addition, it is responsible for maintaining consistency between the views of participants, and for serving as the runtime interface for users to manage their sessions.

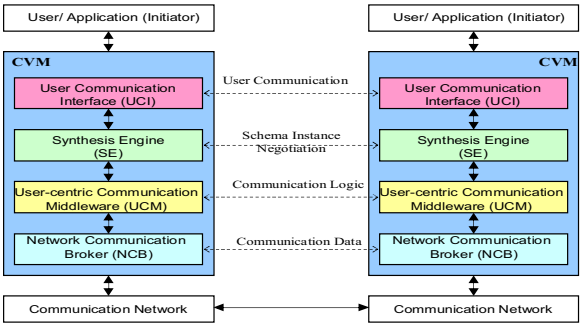


Figure 2: Layered CVM architecture.

SE performs two major tasks. The first is schema negotiation among participants of communication to ensure that all parties agree to a consistent schema. Second, SE automatically transforms the schema to an executable communication control script. This script represents the network-independent control logic for user-level communication session specified in the schema. A basic requirement for SE is that the synthesis process must be fully automated. SE uses a repository of pre-defined components for common as well as domain-specific communication functions. SE puts together the communication control script by combining pre-defined components (e.g., communication session establishment or transmission of text message) based on the user-defined schema. Consequently, the capability of a schema synthesizer can be improved incrementally as more "middleware" components are developed. The design of automated and efficient synthesis techniques and the middleware components represents another class of interesting research issues.

UCM is the execution engine for communication control scripts. Based on the communication logic defined in the script, UCM invokes the common services provided by the NCB layer to perform tasks including: (1) session creation, (2) adding a participant to the session, (3) adding a media to the session, (4) transmitting media, and (5) adjusting media QoS. UCM is also responsible for updating the user communication schema resulted from runtime changes. These changes (received in the form of signals from the NCB layer) may include: (1) session invitation, (2) receive media, (3) end media transmission, and (4) connection failed. Furthermore, UCM is responsible for providing a safe state transition between the running and updated communication control scripts. For example, when a session participant changes the communication schema by switching from a person-to-person call to a multi-way conference, SE will generate a new communication control script that reflects the change. Once the new communication control script is deployed to UCM, it should transfer the state of the old control script to the new one seamlessly and safely [29].

Table 1: Summary of high-level tasks of CVM layers.

CVM Layer	Tasks
User Communication Interface	1. Create/modify the communication schema instance based on user input. 2. Check the correctness and validity of the user communication schema. 3. Maintain consistency between participants' instances.
Synthesis Engine	1. Ensure the consistency of user communication schema through schema negotiation. 2. Perform schema synthesis to obtain the communication control script. 3. Deploy the script to the user-centric communication middleware.
User-centric Communication Middleware	1. Execute the communication control script. 2. Update the user communication schema based on changes made by other participants. 3. Perform a safe state transition from an older schema to an updated one.
Network Communication Broker	1. Provide a high-level communication API, which is independent of the platform. 2. Utilize and coordinate the available, low-level network and hardware services. 3. Provide self-management in response to dynamics of the underlying infrastructure.

NCB is responsible for providing a uniform API of high-level and network-independent communication services to diverse communication applications [31], in such a way to shield user applications from the underlying network/device heterogeneity and dynamics. It utilizes and coordinates networking functions (e.g., signaling, encoding & decoding, and transmitting & receiving) provided by the underlying networks, systems, and libraries. Given the variety and complexity of network configurations, it must exhibit a self-managing behavior that can respond to dynamics of the underlying device and network infrastructure. The concept of NCB offers a novel approach to simplify application development and interoperation,

and introduces many important research issues including self-management, dynamic configuration, definition of application independent communication API, and software framework for hiding network heterogeneity.

These layers collectively fulfill the promise of CVM – that of generating communication applications that are reconfigurable, adaptive, and flexible based only on a high-level description of communication requirements. A summary of the high-level responsibilities assigned to each of these layers is presented in Table 1.

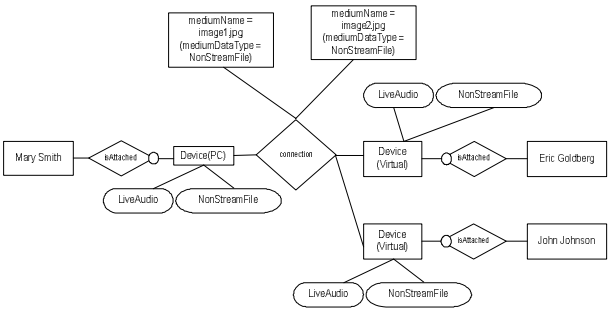
4. Communication Modeling Language

We present an intuitive communication modeling language (*CML*) for modeling user communication requirements. We developed two equivalent variants of CML: the XML-based (X-CML) and the graphical (G-CML), which are documented formally and in detail in [8]. The former is the version that CVM understands and processes, while the later is the user-friendly graphical form. G-XML is analogous to the E-R diagram [5] in the database domain. In [8], we show how these two variants can be automatically converted to each other. Figure 3 shows the (instantiated) X-CML and G-CML representations for the scenario of Section 2.

A *connection* in CML is a user session, and is defined as a communication among a group of participants, where exchanged data is by default broadcasted to all participants. In addition to the local side, a connection contains a set of media (mediaAttached) currently transferred in the connections (user communication session) and a set of remote participants (remote). Both local and remote participants are associated with a communication device (e.g., PC, cell phone), which is associated by a set of capabilities (deviceCapability).

Notice that the specific characteristics of a device, such as its type (e.g., PC or cell phone) or its connection type to the network (e.g., IP or cellular), are not defined nor required. The reason is that CML operates on an abstraction of the underlying network and devices as mentioned above. We assume there is a single virtual device per person, which has the union of the capabilities of all physical devices attached to the user.

A medium is a data piece or data stream, like a Word document or a live video feed respectively. A medium has a type which is one of the predefined types supported by the system, a mediumURL that contains the location of the medium (a file location for a data piece or a port for a data stream), a suggestedApplication which defines the application that can be used to view or process a medium (e.g., Powerpoint for ppt files), and an action which defines a default action that is performed on a medium. Actions "send" and "doNotSend" mean transfer automatically the medium or wait for the user to choose respectively, while "startApplication" orders the system to open the suggested application of the medium once transferred.



(a) G-CML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<userSchema>
<connection connectionID = "connection1">
    <device deviceID = "001" isVirtual = "false">
                <deviceCapability>LiveAudio</deviceCapability>
                <deviceCapability>NonStreamFile</deviceCapability>
    </device>
    <device deviceID = "002" isVirtual = "true">
                <deviceCapability>LiveAudio</deviceCapability>
                <deviceCapability>NonStreamFile</deviceCapability>
    </device>
    <device deviceID = "003" isVirtual = "true">
                <deviceCapability>LiveAudio</deviceCapability>
                <deviceCapability>NonStreamFile</deviceCapability>
    </device>
</connection>
<person personName = "Mary Smith" personID = "007" personRole = ""/>
<person personName = "Eric Goldberg" personID = "011" personRole = ""/>
<person personName = "John Johnson" personID = "012" personRole = ""/>
<isAttached personID = "007" deviceID = "001"></isAttached>
<isAttached personID = "011" deviceID = "002"></isAttached>
<isAttached personID = "012" deviceID = "003"></isAttached>
<data connectionID="connection2">
    <medium mediumDataType = "NonStreamFile" mediumName = "Heart_Scan.jpg"
                mediumURL = "http://fiu.edu/Heart_Scan.jpg"/>
    <medium mediumDataType = "NonStreamFile" mediumName = "X_Ray1.jpg"
                mediumURL = "http://fiu.edu/X_Ray1.jpg"/>
</data>
</userSchema>
```

(b) X-CML

Figure 3: CML example for our scenario.

Finally, composite data are represented using forms in CML, which are nested structures that contain media as well as user-defined attributes (e.g., media with common suggestedApplication or action settings can be grouped together in a form). For example, it is common in medical scenarios to require transferring complex medical data consisting of multiple simple media (e.g., a page in a patient medical record).

5. A Prototypical Design of CVM

We present a prototypical design of CVM, which closely follows the CVM architecture. The notation of Figure 4 (adopted by [12]) is used to describe the interfaces between the prototype components, which

are summarized in Table 2. Notice that each layer uses (resp. handles) what the lower level provides (resp. signals).

UCI Component. The architecture design of UCI, which is detailed in [8], consists of four major components: (1) the *communication modeling environment* – provides the user with an environment to develop communication schemas and instances in G-CML which are then automatically transformed to X-CML, (2) the *schema transformation environment* – transforms an X-CML instance into a synthesis-ready X-CML instance or stores the X-CML model in the repository, (3) the *repository* – stores artifacts (*e.g.*, grammar rules and CML schemas) to support the creation of CML schemas/instances, and (4) the *UCI-to-synthesis engine interface* – provides a conduit for interaction with the synthesis engine.

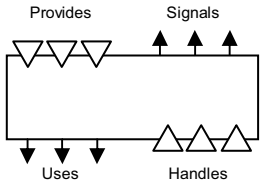


Figure 4: Generic layer architecture [12].

UCI provides the interface functions invoke(SchemaInst), invoke(SchemaData) and store(SchemaInst), shown in Table 2, to the User/Application (initiator) (Figure 2), where SchemaInst and SchemaData are a communication schema instance and the data to be sent, respectively, and are both represented in X-CML. The functions return an acknowledgement or error message in the form of a message (msg). For example, in the scenario describe in Section 2 Mary builds the communication instance in a client application and submits the X-CML shown in Figure 3(b) via the invoke functions to UCI.

Furthermore, UCI checks the syntactic and semantic correctness of a communication schema instance by building an abstract syntax tree of the X-CML representation and traversing the tree to check type compatibility of the media types and the values of the fields of the attributes. Some schema instances require that their abstract syntax trees be annotated with meta-data associated with a communication schema (a template for a class of communication schema instances). This meta-data is defined using the communication modeling environment and stored in the repository.

UCI also stores the current state of the schema instance. The state is required during communication with the synthesis engine and is updated based on the signals from the synthesis engine (see Table 2). For example, if there is a problem in transmitting the images of the patient record (scenario from Section 2) to Mary (e.g., bandwidth of the connection is too small) the UCI signals the User/Application (initiator) of the problem. This status update allows Eric, the sender of the image, to send text describing the images.

Table 2. Interface between CVM components.

CVM Layer	*Provides*	*signals*
UCI	msg invoke(SchemaInst) msg invoke(SchemaData) msg store(SchemaInst)	Altered Instance (after negotiation by SE) medium transmission status - Exceptions
SE	msg invoke(SchemaInst) msg invoke(SchemaData)	notifyMediaStatus notifyParticipantStatus notifyException notifySIStatus
UCM	executeScript(XML)	notifyMediaStatus notifyParticipantStatus notifyException

NCB	createSession(sessionID)
	addParty(userID);
	addMedia(mediaURI),
	applyPolicy(xmlString);
		notifySessionStatus
		notifySessionInvitation
		notifyNetworkFailure

SE Component. The role of the synthesis engine is to convert the user schema in CML representation to an executable communication control script. It is invoked by UCI via its provides interface functions, invoke(SchemaInst) and invoke(SchemaData). SE performs the following tasks in sequence.

First, SE determines if the invocation from the UCI requires a negotiation with the other session participants. If yes, it establishes an initial communication session with the other participants and performs *schema negotiation* to obtain an agreed schema in CML notation. SE then carries out the actual *synthesis process*. It parses the CML representation to obtain the logical components, which are the relationships along with associated media. For each logical component of the schema instance, the SE appends the appropriate script invocation (detailed below) to the communication control script. Finally, it dispatches the communication control script to the UCM layer.

In CVM, each participant of a communication session has a local copy of their schema, which maybe changed at runtime. Any change to the schema will result in an update to the schemas of all participants. If two users in a session are simultaneously altering their schemas, concurrency problems arise. The SE component uses a modified 3-phase handshake protocol [26] for schema synchronization.

As an example, the script for the scenario of Section 2 created by Mary's local synthesizer is as follows:

```
createSession("ID");
addParty("ID", "ericgoldberg");
addParty("ID", "johnjohnson");
```

```
addMedia("ID", "audio", "");
addMedia("ID", "nonStreamingFile", <URL>);
addMedia("ID", "nonStreamingFile", <URL>);
```

where the <URL>'s are replaced by their resolved values for the actual Heart_Scan.jpg and X_Ray1.jpg file locations. The above script is delivered to the UCM using the executeScript() of UCM "provides" interface (as presented in Table 2).

SE also delivers four types of notifications (See Table 2) to the UCI layer. The notifyMediaStatus and notifyParticipantStatus signals notify the UCI about media delivery and participant connectivity. The notifySIStatus signal notifies the UCI about changes to the schema instance as a result of external changes due to other participants such as addition of new participant to an existing session or a change in capabilities of an existing participant, etc. Finally, the notifyException signals the UCI about exceptions such as lost network connection.

UCM Component. UCM is responsible for executing the communication control script and for maintaining the states of user level communication (as opposed to network level one). These states may include communication logs, data exchanged, and so on. In other words, UCM manages user communication sessions.SE passes the control script to UCM through the UCM API executeScript. Although UCM encapsulates some of the raw signals coming from NCB, it will process the signals and report the change of session status to SE, in order to make necessary modifications to the schema. We note that safe state transition capability in user-centric communication middleware is not yet available in this version of the prototype.

NCB Component. NCB's job is to manage network sessions. (It should be clear that each user session may result in many network sessions.) Each participant of a session can multicast to all the other participants. The NCB API (detailed in [31]) to UCM is both application- and network-independent, through which high-level communication tasks can be specified. A new session is created by invoking the createSession call provided by NCB, with a session ID, which maintains a unique association between each user and network sessions. NCB provides addParticipant, and addMedia services to UCM to dynamically add participants and media types in user sessions. The NCB interface allows application to customize NCB behavior under specific network and system conditions, based on user or application preference. The interface, applyPolicy, takes as input an XML string which describes the policy for self-management. The NCB callback interface presented in Table 2 allows it to signal the status of the network, the status of the existing sessions, and a session invitation from a remote user (i.e. the new session will be created after the local user agrees to join the session).

NCB translates a high-level communication task into a series of operations that control the underlying networking facilities. It encapsulates and abstracts the heterogeneity of the network protocols and their interfaces. The NCB core further includes modules such as Session Management, Participant Management, Media Management, and QoS and Self-Management. The current prototype implementation utilizes the JAIN SIP and the JMF library, and supports SIP and RTP as underlying networking protocols.

6. Prototype Implementation

A CVM prototype has been implemented using the following technologies. The (Web-based) user interface has been deployed with the Opera 8.5, a voice-enabled browser. This prototype enables creation, modification, and use of communication schema instance using voice commands. The technologies used at the browser side are HTML, Javascript for dynamic effects and the program logic, and XHTML+Voice[2] for voice conversation. Part of the Javascript code uses AJAX[3] technology (Asynchronous JavaScript and XML) to make web requests and responses in the background, without having to refresh the web pages. The rest of the CVM layers are implemented in Java, deployed on each node, and exposed to the browser GUI as a local Web server. JAIN SIP and Java Media Framework (JMF) are used for control and data communications, respectively. Finally, we used SER (SIP Express Router) server for registration and presence and Asterisk for connection to PSTN and audio mixing.

Figure 5 shows three screenshots of our prototype GUI. Figures 5 (a) and (b) show the prototype being used as a standalone Web- based application, while Figure (c) shows the prototype being used in combination with the iRounds system[4]. Following the scenario of Section 2, Mary loads the Telemedicine communication schema from the schema repository, and selects the two participants (Eric and John) from her Address Book (the result is shown in Figure 5 (a)). The media used in the connection are selected from the Media Library (represented by icons on the top right of Figure 5 (b)), and the two JPG files ("Heart_Scan.jpg" and "X_Ray1.jpg"), are dragged into the Connection Box by Eric (the result is shown in Figure 5 (b)).

We have tested our prototype implementation with several other case studies both in general purpose applications such as multimedia conferencing and in

[2] http://www.voicexml.org/specs/multimodal/x+v/12/
[3] http://java.sun.com/developer/technicalArticles/J2EE/AJAX/
[4] *i-Rounds*© is an integrated clinical information system developed by Teges® and currently being used in Miami Children's Hospital.

domain specific applications such as Telemedicine and Disaster Management. Our Telemedicine scenarios have been provided by our partners at Miami Children Hospital.

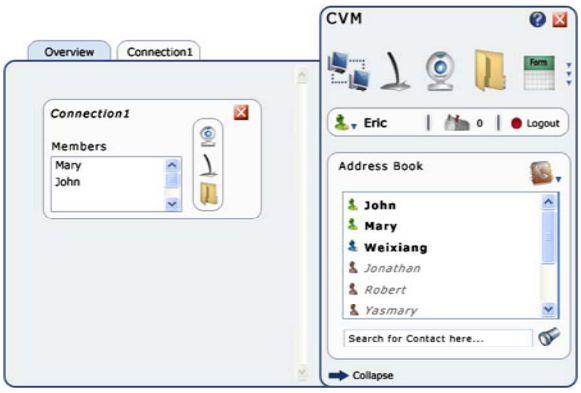
7. Related Work

There is a plethora of related research that addresses the individual processes and artifacts used in the various components of the CVM. However, not much has been published on how such components can be combined to provide flexible, user-centric and on-demand communication solutions.

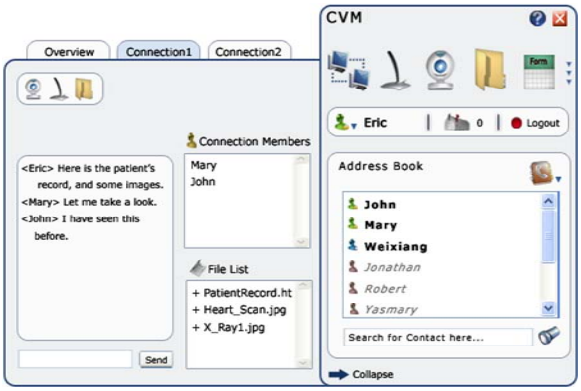
There are a number of off-the-shelf communication applications such as Yahoo! Messenger and MSN Messenger. We are also aware of several companies' efforts to integrate various tools into comprehensive communication solutions. The development approach that these products are based on dictates that none of them possesses the flexibility, on-demand, and user-centric communication solutions addressed in this paper. For example, it would be a tall order to adapt any of these tools to a comprehensive telemedicine application.

Model-Driven Engineering The CVM approach shares some common traits with the concept of model-driven engineering [1,3,13]. In contrast to general-purpose model-driven development, automatic generation of communication services is feasible in CVM for two reasons. First, CVM is restricted to the scope of communication services and does not bear the complexity of generating general-purpose applications. The complexity of communication logic can be carefully regulated through the design of the schema modeling language. Second, CVM utilizes communication middleware components (*e.g.,* those of ACE [28]) and server-side architectures (*e.g.*, [2]) as building blocks to generate communication applications. Such existing components encapsulate procedures, patterns, and algorithms governing basic communication services (*e.g.,* session establishment of person-to-person voice call, transmission of an image file, and real-time video streaming), which are well understood. The role of CVM is limited to the identification and composition of such components [19].

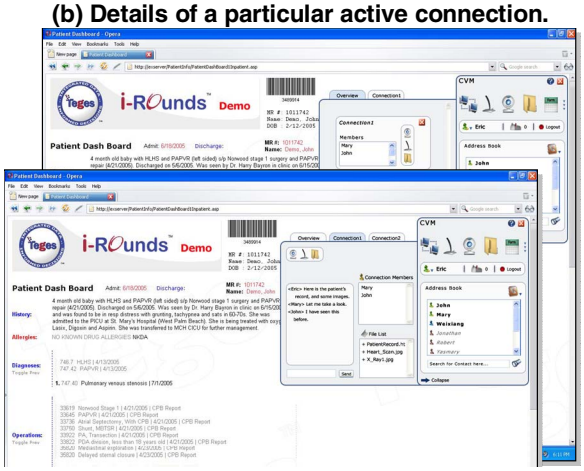
More specifically, Heckel and Voigt [13] describe how models in UML are transformed into BPEL4WS using the concept of pair grammars. We use a similar approach in the UCI but our modeling language G-CML is far more restrictive than UML and hence far more manageable and its synthesis can be automated. The implementation of the visual model in the UCI is based on the work by Costagliola et al. [6]. Costagliola et al. provide a framework that allows the user to define a visual language, create graphical models, validate these models and convert the models into strings of another language. The work in [3] generates code from models using tool suites for specific application domains that were developed using a generic modeling environment. In our work, a generic SE generates control scripts from a CML description of communication logic, with restricted utility to the communication domain.



(a) Overview of active communications.



(b) Details of a particular active connection.



(c) Integration of CVM with the iRounds system.

Figure 5: Screenshots of CVM prototype.

Communication Middleware. There has been extensive work on communication middleware. Our work used many of the principles presented by Schmidt [23], including using patterns and frameworks to alleviate complexity associated with a growing range of multimedia data types, traffic patterns, and end-to-end QoS requirements. Schmidt explored common pitfalls of developing communication software, including limitations of low-level native OSs and APIs and the limitations of higher-level middleware. The UCM and NCB components of CVM are designed exactly to avoid these pitfalls.

The existing protocol stacks may not be always suitable to take advantage of advanced transmission technologies and high-speed networks. Geppert and Rößler [11] discussed how communication architectures could be made more flexible by automatically configuring communication subsystems based on a specification of desired target service. In the NCB we use a similar approach.

Stiller et al. [27] described the Da CaPo++ system as an end-system middleware for multimedia applications adaptable to the application needs. The authors claimed that Da CaPo++ automatically configures suitable communication protocols, provides an efficient runtime support, and offers an easy to use object-oriented API, which shares some common traits with the low layers of UCM components.

UCM design also leverages the concept of adaptive and reflective middleware, such as ACE and Ensemble, to provide self-management using only a high-level guideline. ACE [28] is a real-time C++ framework that wraps OS services and provides a variety of communication-related patterns. Ensemble [21] is a groupware communication toolkit, which enables insertion of detectors in protocol graph. These detectors can trigger dynamic adaptation by distributing a new protocol-graph specification to all involved participants using a reconfiguration protocol.

JAIN SIP [15] is a standardized Java interface to SIP. Java Media Framework [16] is a library for audio and video communication. The low-level APIs of these communication libraries are still significantly complex to use, and far less usable than the user-centric session of UCM. The Java Telephony API is a high-level API for traditional telephony applications. They do not support next-generation multimedia communication applications with sophisticated business logic.

Reference [30] discusses open software architectures for IP-based voice communication. Parlay [20] is an API for rapid creation of telecommunication services. 14ERG project [14] provides core network architecture for integrated communications. These frameworks mostly address the server-side architecture and service creations. The server-side architecture has different concerns than the client-side middleware, which is the focus of UCM. In contrast to traditional telephone networks, in IP networks, end-hosts are capable of sophisticated communication logic.

Finally, the CVM principle of separating policy from mechanism has been popular in the operating systems community for several decades [18].

8. Conclusion

We have presented CVM for on demand declaring, synthesizing and delivering communications services. We have discussed its architecture, as well as supporting modeling language, components, algorithms, interfaces, and prototypical implementation.

We discussed how CVM allows users to rapidly build and execute communication schemas to provide communication solutions across different application domains. It would be a misconception, however, to assume that an end-user needs to know modeling before they can use the CVM. For most end-users (e.g., a doctor), the modeling aspect will be hidden, because the schemas they use will be packaged as predefined services by their service providers or their organizations (e.g., a hospital).

Several classes of issues including security and performance at different layers of CVM are not addressed in this paper. A number of useful features can also be added. Robust and effective solutions to these issues require further study, which represent exciting and interesting research topics. We argue, however, CVM represents a new paradigm for structuring and delivering communication solutions and services, which are far more effective than the current ways of development. In fact, the unique architectural traits of CVM allow new components and features to be seamlessly added as they become available. As such, CVM can serve as a communication service framework, which can be built upon and incrementally improved by the collective wisdom of the research community.

Acknowledgements: This work was supported in part by the National Science Foundation under grant HRD-0317692. We thank Eric Johnson, Eduardo Monteiro, Weixiang Sun, Eric Sanchez, Yingbo Wang, Robert Redway, Farid Hosseini, Yasmary Hernandez, Jonathan Corrales, and Onyeka Ezenwoye for their participation in CVM prototype implementation.

9. References

[1] Jorn Bettin, "Model-driven software development: An emerging paradigm for industrialised software asset development", Tech report, SoftMetaWare, June 2004.

[2] Gregory W. Bond, Eric Cheung, K. Hal Purdy, Pamela Zave, and J. Christopher Ramming, "An open architecture

for next-generation telecommunication services", ACM Transactions on Internet Technology IV(1) pp:83-123, February 2004.

[3] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema "Developing Applications Using Model-Driven Design Environments", IEEE Computer, pages 33 – 40, February 2006.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, "Pattern-Oriented Software Architecture: A System of Patterns", Wiley, 1998.

[5] P. P. Chen, "The entity-relationship model: Toward a unified view of data", ACM Trans. Database Syst. 1, 1, 9–36, 1976.

[6] Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese.A framework for modeling and implementing visual notations with applications to software engineering. ACM Transac- tions on Software Engineering and Methodology (TOSEM), 13(4):431–487, 2004.

[7] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Unman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In Proceedings of IPSJ Conference, Tokyo, Japan, October 1994.

[8] Peter J. Clarke, Vagelis Hristidis, Yingbo Wang, Nagarajan Prabakar and Yi Deng. A Declarative Approach for Specifying User-Centric Communication. Symposium on Collaborative Technologies and Systems (CTS), 2006.

[9] Department of Health. Health Insurance Portability and Accountability Act (HIPPA) ttp://dchealth.dc.gov/hipaa/ hipaaoverview.shtm (June 2005).

[10] John D. Day and Hubert Zimmermann, "The OSI Reference Model", Conformance testing methodologies and architectures for OSI protocols, IEEE Computer Society Press pp:38-44, 1995.

[11] B. Geppert and F. Rößler, "Automatic Configuration of Communication Subsystems – A Survey", Tech. Report SFB 501 Report 17/96, University of Kaiserslautern, Germany, 1996.

[12] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. System Architecture Directions for Networked Sensors. In Architectural Support for Programming Languages and Operating Systems, pp. 93-104, 2000.

[13] Reiko Heckel and Hendrik Voigt. Model-based development of executable business processes for web services. In Lectures on Concurrency and Petri Nets: Advances in Petri Nets, volume 3098, pages 559–584. Springer, June 2004.

[14] The 14ERG Project, University of California, Berkeley. http://14erg.cs.berkeley.edu/

[15] JAIN SIP: https://jain-sip.dev.java.net/, 2006.

[16] Java Media Framework API, http://java.sun.com/products/java-media/jmf/, 2006.

[17] David Krebs. "The Mobile Software Stack for Voice, Data, and Converged Handheld Devices", Mobile and Wireless Practice Venture Development Corporation, April 2005.

[18] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/Mechanism Separation in Hydra", In Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP '75), pages 132–140, University of Texas at Austin, November 1975.

[19] Philip K. McKinley, Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. "Composing adaptive software", IEEE Computer, pages 56-64, July 2004.

[20] The Parlay Group. http://www.parlay.org/specs/library/index.asp, 2006.

[21] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using Ensemble," Software Practice and Experience, vol. 28, p. 963979, August 1998.

[22] R. L. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems", Communications of the ACM, 1978.

[23] D. C. Schmidt. "Applying Patterns and Frameworks to Develop Object-Oriented Communication Software", volume 1 of Handbook of Programming Languages. MacMillan Computer Publishing, 1997.

[24] D. C. Schmidt. Middleware for real-time and embedded systems. Communications of the ACM, 45(6), June 2002.

[25] D. C. Schmidt, "Model-Driven Engineering", IEEE Computer, February 2006, 25-31.

[26] Dale Skeen: Non-blocking Commit Protocols. Pages: 133-142, SIGMOD 1981.

[27] Burkhard Stiller and Christina Class and Marcel Waldvogel and Germano Caronni and Daniel Bauer, "A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience", IEEE Journal on Selected Areas in Communications, vol. 17 no. 9 Sept. 1999, pages 1580 – 1598.

[28] D. C. Schmidt and S. D. Huston, C++ Network Programming: Mastering Complexity Using ACE and Patterns. Addison-Wesley Longman, 2002.

[29] N. Venkatasubramanian, "Safe 'Composability' of Middleware Services", Comm. ACM, June 2002.

[30] Pamela Zave, Healfdene H. Goguen, and Thomas M. Smith, "Component coordination: A telecommunication case study", Computer Networks 45(5):645-664, August 2004.

[31] Chi Zhang, S. Masoud Sadjadi, Weixiang Sun, Raju Rangaswami, and Yi Deng. User-centric communication middleware. Technical Report FIU-SCIS-2005-11-01, Florida International University, November 2005.