ChaiTime[1]*:* A System for Rapid Creation of Portable Next-Generation Telephony Services Using Third-Party Software Components

Farooq Anjum[2], Francesco Caruso, Ravi Jain[3], Paolo Missier and Adalberto Zordan[4]
Bellcore
445 South St.
Morristown, NJ 07960

Abstract. **We present the architecture, design and experimental research prototype implementation of *ChaiTime*, an open system architecture for the rapid development of advanced next-generation telephony services that overcomes some of the limitations of the current closed PSTN architecture and service model. ChaiTime allows communication sessions to be set up over the PSTN, the Internet, or a combination of both. Services can be provided by multiple cooperating distributed service providers, some of whom may use third-party *software components* which can be "plugged in" or even dynamically downloaded from the network as needed. This allows advanced services to be deployed and delivered to users rapidly, a crucial requirement in the increasingly competitive telecommunications services marketplace.**

ChaiTime is built upon an object-oriented call model called *Java Call Control (JCC)* which we have defined as a small set of extensions to the standard Java Telephony API (JTAPI) call model that allows support for distributed providers as well as advanced services. JCC hides details of underlying call-state management, protocols and hardware from applications. In our prototype, we have designed a small set of extensions to SIP, called *Extended SIP*, for supporting advanced services. The ChaiTime prototype software is currently operational in our laboratory. We briefly describe its current implementation as well as future work to address issues such as fault tolerance.

I. INTRODUCTION

It is increasingly being recognized that the current Public Switched Telephone Network (PSTN) infrastructure has an architecture that is outmoded in several respects [1]. In this paper we present the architecture, design and prototype implementation of *ChaiTime*, a system for the rapid development of advanced next-generation telephony services that overcomes some of the limitations of the current PSTN architecture and service model. In particular, ChaiTime allows communications sessions (voice, video, multimedia, etc.) to be set up over the PSTN, the Internet, or a combination of both. More importantly, services can be provided by multiple cooperating distributed service providers, some of whom may use third-party *software components* which can be "plugged in" or even dynamically downloaded from the network as needed. This allows advanced services to be deployed and delivered to users rapidly, a crucial requirement in the increasingly competitive telecommunications services marketplace. One illustration of rapid deployment of an advanced service facilitated by ChaiTime is the following:

Example Scenario: *Dynamic Service Download*. In this scenario, a service can be activated for one-time use on a customer's terminal. If the terminal is not set up for the service, the necessary software can be downloaded and installed just prior to use. The main steps in the proposed scenario are as follows: (1) Alice, the user of terminal *A*, invites Bob, who is using terminal *B*, to a communication session where they can use a whiteboard to share information; (2) *A* issues the whiteboard session invitation to *B*; (3) Unfortunately, *B* does not have the whiteboard software installed, and it informs *A* accordingly; (4) *A* suggests a location on the Internet where the software can be obtained; (5) *B* downloads the software, negotiating one-time use payment with the software provider if the software is not public-domain; (6) *B* sets up a whiteboard session with *A* so Alice that Bob can communicate. Note that Alice and Bob have largely been spared the details of this session negotiation and dynamic service download.

Several variations of this scenario are possible. For instance, Bob may configure his terminal to consult him before paying for one-time service activation fees if the fees exceed some threshold. Alice may authorize her own terminal to download the whiteboard software to Bob's terminal. Another possible situation is as follows. A third party, Carol, is interested in having *A* and *B* establish the session. For instance, Carol may be Alice and Bob's common supervisor. In this case, Carol may take an active role in providing Bob's

---

[1] ***Chai*** *(n.; rhymes with "shy")*: In several languages, the word for tea (as opposed to coffee, Java, etc.)

[2] This work was done while the author was a summer intern at Bellcore. Current contact: farooq@glue.umd.edu

[3] Address correspondence to: Ravi Jain, Applied Research, Bellcore, 445 South St, Morristown, NJ 07960. Phone: (973) 829-3178. Fax: (973) 829-2645. Email: rjain@bellcore.com

[4] This work was done while the author was a summer intern at Bellcore. Current contact: zordan@lisa.record.unipd.it

Preprint: IEEE Conf. on Open Architectures and Network Programming (OPENARCH), New York, Mar. 26-27, 1999.

terminal with information about suitable providers that can supply the services required for the session.

Our contributions in this paper can be summarized as follows. We argue that next-generation telecommunications systems should allow rapid creation of advanced, portable services by third parties using interchangeable software components. The *ChaiTime* system represents a step toward this goal. Intelligence is assumed to be pervasive but not completely ubiquitous; if the user does not have access to a smart terminal, services can be provided, albeit with some restrictions, by means of proxies located in the network. The software architecture of ChaiTime allows services to be deployed by means of *software components* that can be developed by third party software vendors and added to the set of available services transparently, while a session is in progress.

In order to provide these facilities, ChaiTime needs a flexible and extensible *call model*, a specialized virtual machine for the development of telephony-related applications. To this end, ChaiTime itself is built upon an object-oriented call model called *Java Call Control (JCC)* which we have defined as a small set of extensions to the standard Java Telephony API (JTAPI) call model [2]. While JTAPI is generally intended for use in a single-platform environment (e.g. a PBX or a Call Center), JCC is designed for use in a distributed environment, and can thus be regarded as "Distributed JTAPI". JCC also hides the underlying communication protocols and architectures, providing a high-level object model for maintaining and managing the state of a call session. In particular, JCC allows different underlying session initiation and control protocols to be used, e.g. H.323 or SIP [3]. In our prototype we have designed a version of SIP, called *Extended SIP*, for session initiation which defines a small set of extensions to SIP for supporting advanced services. The ChaiTime prototype implementation is currently being used in our laboratory for voice, text, whiteboard, and image-exchange sessions.

The paper is organized as follows. In Section II we present the background and motivation for this work in more detail. In Section III we discuss the main entities in the ChaiTime architecture from a services point of view, and how services are deployed and implemented. The JCC call model and the use of the SIP protocol are described briefly in Section 0, using dynamic service download as an example. We also describe how to provide a different type of advanced service, namely the exchange of electronic business cards between users in the midst of a call, which requires multiple parallel sessions to be set up between users. In Section V we provide some more details on the implementation, using the Unified Modeling Language (UML) [4], to describe the associations and interactions among the objects in the system. Section VI briefly discusses the design as well as directions for future work, and in Section VII we end with brief concluding remarks.

II. Background and motivation

The Public Switched Telephone Network (PSTN) implicitly assumes that the *computational resources*, the *service creation process*, and the underlying *business model* for telecommunications services are centralized. In terms of computational resources, the current PSTN assumes that the resources available at the user terminals, or "Customer Premises Equipment" (CPE), are extremely limited, so that all the intelligence required to provide services is centralized in network switches, databases, and operations support systems. In terms of the *service creation process*, the Advanced Intelligent Network (AIN) architecture represented an important advance when it was introduced in the PSTN in that it separated service development from switching, allowing service logic to be developed more quickly and placed in specialized network databases (e.g. the Intelligent Service Control Point, or ISCP) while switches could be optimized for speed and efficiency. However, it still assumed that services were created on specialized platforms, using specialized languages and programming models, by specialized personnel employed by a telecommunications service provider. Hand in hand with this assumption was that of a *business model* in which a centralized service provider, one per geographical region, controlled the available communications resources and provided service.

Clearly, advances in hardware and software technology, the rise of the Internet, and market deregulation have combined to make the assumptions underlying the PSTN invalid. One of the likely scenarios for next-generation telephony is one that blends features from the old PSTN world with those from the growing domain of IP-based data networks. Many differences exist, of course, between the two networks, but we are concerned specifically with differences at the service level. The PSTN architecture is *network-centric* and assumes that the service logic, or "intelligence", resides within the network itself, normally in the form of programmable network elements and of centralized systems, such as the ISCP, which implement user-level services. IP-based networks, on the other hand, are *application-centric*, in that the network passively provides connectivity among hosts, and is largely unaware of the applications that run on those hosts.

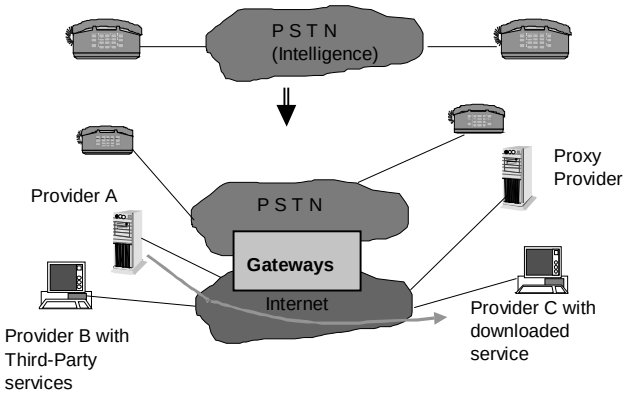


Figure 1 Evolution of the telecommunication system

Currently, market forces are increasingly calling for these network models to converge, with consequences on the way user services are defined, developed and deployed. In a broad sense, the two approaches are complementary and are already converging: the PSTN, designed to provide primarily basic telephony, has been evolving to support new user-level, "intelligent" services. The Internet, traditionally used for client-server and peer-to-peer data applications, is increasingly being used for telephony and telecommunications services. One of the expected effects of this convergence is that that service logic will now be spread throughout the network. This means that rapid development and deployment of telephony services becomes possible, and that network providers and the service providers no longer need to coincide.

Figure 1 illustrates the next-generation telephony scenario we assume. The PSTN and the Internet will continue to coexist, and will interface with each other through gateways. On the resulting combined network, various types of terminal equipment will exist, ranging from "black" phones to personal workstations. A user will be able to enjoy a variety of advanced services, some of which are offered by multiple cooperating providers that may or may not also be PSTN providers or Internet Service Providers (ISPs). In some cases, users or the service providers themselves will adopt software developed by third-party Independent Software Vendors (ISVs), and from time-to-time providers may download services from other providers on behalf of their users. A user who is using a black phone will be able to obtain some services by means of *proxies* connected to the network, where the proxies themselves may be offered by service providers. For instance, while black phones may be unable to support video-based services, some of the more advanced audio services can still be available to them through the proxy.

In this context, we have developed a software architecture and system, called ChaiTime, that allows for the rapid creation of advanced, portable telephony applications by third party providers.

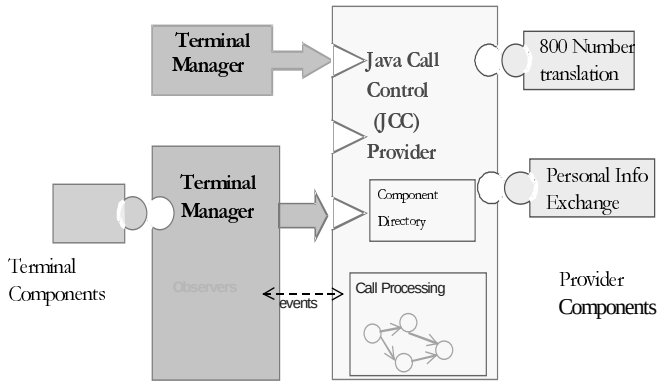


Figure 2 ChaiTime service architecture

We give an overview of the ChaiTime system in the following sections.

III. ChaiTime service architecture

In this section we describe how services are deployed and activated in the ChaiTime architecture; in the following section we will discuss the Java Call Control (JCC) call model used to implement service deployment and activation.

The ChaiTime architecture consists of two main software entities: *Endpoints* and *Providers*. An Endpoint has a single unique logical address (which may be a phone number or an Internet address) in the system, and consists of one or more user terminals, a Terminal Manager, or TM, and possibly a proxy server. A user's terminal may be a black phone, a wireless handset, a laptop, or a workstation. The TM is software that either runs on the user's terminal or on a proxy server in the network. For ease of exposition in this paper we will assume that an Endpoint consists of a TM running on a single user terminal, although other configurations are possible, and refer to Endpoints and TMs interchangeably.

The ChaiTime architecture is based on a network of distributed, interconnected Providers (e.g. B and C in Figure 1). While Endpoints have a single logical address, Providers manage disjoint sets of such addresses, called domains, and provide network connectivity and session management for the corresponding Endpoints. Figure 2 provides an overview of the relationship between Endpoints, Providers, and plug-in service\s. The JCC Provider manages the call processing logic, defined as a state machine, among peer TMs. TMs whose logical addresses belong to the Provider's domain are shown attached to the Provider.

Services can be hosted by Providers or directly by the TMs, or both. Like services traditionally provided in the PSTN, a service can be permanently installed and constantly active at a Provider. For instance, an 800-number translation service, which maps 800-numbers to a physical access line number (or an Internet address), can be offered by a Provider for its domain. The number translation can take place in a transparent way during a normal session setup when the caller application

requests a connection to a peer identified by the 800 number. (The actual 800-number database could be installed at the Provider, or the Provider could itself access a database offered by a service provider via the network.) Alternatively, a service can be provided by the user's TM. For instance, a speed-dialing service, in which important or frequently-dialed numbers are mapped to single digits (represented as hardware or software buttons on the user's terminal) could be provided by the TM. Whether a particular service is provided by a TM or a Provider depends upon several factors, one consideration being that a service that is likely to be shared by many TMs (e.g. 800-number translation) may be better located at the Provider, while services that are very specialized or user-specific may be better located at a TM.

In ChaiTime, service and even and activation can take place as part of the session setup. An instance of a service is downloaded on demand, and activated only for the duration of a session, as in the whiteboard service of the dynamic service download scenario. The whiteboard application can be a short-lived service obtained for the duration of that session. Its lifetime and the Provider designated to supply the service can be specified during session setup.

As shown in Figure 2, services are generally software components that plug-in to the TM or the Provider. Two types of dynamic resources provide access to services. The first consists of Terminal Components that can be attached to a TM statically, or dynamically during call setup, and typically are user-specific or specialized services. The second type of resource, Provider Components, implement services that are normally shared amongst TMs in the domain. A Terminal Component is implemented as a software component (e.g. a JavaBean [5]) that is analogous to (or, depending upon the service, could in fact actually be) a Web applet [6]. A Provider Component is implemented as a software component (e.g. a JavaBean or a "CORBA Bean" [6, 7]) that is analogous to a Web servlet [8] (one simplistic definition of servlets is that servlets are to Web servers what applets are to Web browsers). An example of a Terminal Component is a JavaBean that provides a speed-dialing service. An example of a Provider Component is a JavaBean that maintains a cache of 800-number translations, and if a dialed number is not available in the cache, contacts a database offered by an 800-number service provider to obtain the translation.

The motivation for implementing services as software components is that, in principle, different third-party software vendors can provide competing implementations of the same service. For instance, a Provider may remove one vendor's implementation of 800-number translation, and replace it with another vendor's implementation that offers better functionality (e.g. is more up-to-date), performance (e.g. better caching or search algorithms) or cost. Current software component technology like JavaBeans, and Enterprise JavaBeans [6, 9], offers support for discovering the attributes of a software component at run-time by means of standardized naming conventions, and, given further standardization of the service interface itself, allowing components to be "plugged-in" for use dynamically.

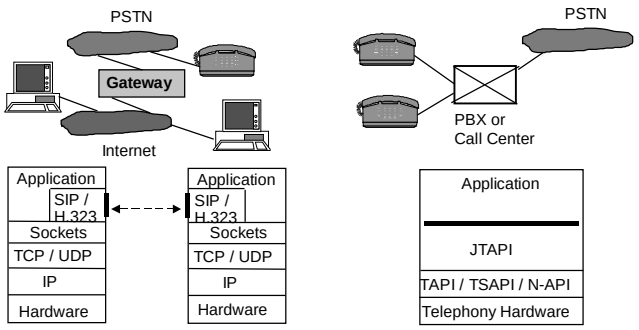


Figure 3 Internet Telephony (IT) vs. Computer-Telephony Integration (CTI) approaches

The issue of managing software components in ChaiTime is similar to that of managing applets and servlets in a Web-based client-server environment. In general, ChaiTime services are registered as software components in a Component Directory. In the case of Terminal Components, the current ChaiTime implementation only allows for services that set up communications sessions with different types of media. Thus, services are represented by media types, which are described using the familiar notion of MIME types. For instance, suppose that during call setup the caller requests the use of a whiteboard. A specific MIME type, say "media/whiteboard", is used for the request. On the called party's side, if a Terminal Component is associated with the requested type, then the call can be immediately accepted and the component can be activated. Otherwise, an attempt will be made to obtain a resource that can be associated with that MIME type, using dynamic service download.

Provider Components are registered in a Component Directory and can be activated either by a TM or implicitly by the provider. The 800 number translation service is an example of the latter. A request for a connection to an 800 number triggers the service invocation on the provider. Once the service invocation returns with the translated destination number, the provider completes the connection. Notice that there is nothing preventing a provider implementation from exposing the service to the TM for a direct invocation. We omit further details of the use and maintenance of the Component Directory due to lack of space.

IV. CHAITIME CALL MODEL

The call model used to implement ChaiTime is called Java Call Control (JCC). By hiding the details of the underlying communication protocols and architectures, JCC provides a suitable abstraction of the telephony network (the combined PSTN/IP network), and offers a high-level object model for maintaining and managing the state of a call session. Specifically, JCC builds upon complementary ideas developed by the Computer Telephony Integration (CTI) and Internet Telephony (IT) communities.

The CTI approach is oriented towards developing portable software for applications such as call centers, PBXs, etc. For example, Sun's Java Telephony API (JTAPI) [2] provides applications with a standard call model for maintaining call state, and hides the hardware API or other API (e.g. TAPI, TSAPI, etc.) from the application (see the right side of Figure 3). In contrast, the IT approach (left side of Figure 3) is oriented towards developing protocols (e.g. SIP, H.323) that allow interoperability and communication between software running on user terminals or gateways.

JTAPI itself, while offering a convenient abstraction for thinking about next generation telephony and networking, is somewhat limited. In particular, JTAPI seems to be oriented towards providing support for developing applications in two types of scenarios: (1) where applications run on a single platform (e.g. a PBX); and (2) where applications run on a platform that is "horizontally partitioned", i.e., the higher layers of software (the application and the JTAPI layer) communicate via Java Remote Method Invocation (RMI) with the lower layers (e.g. TAPI and the hardware) over a network. Also, in JTAPI a Provider is assumed to be in control of all the legs of a call (they all hang off the same Call object managed by the Provider). While this assumption may add to the convenience of managing a centralized call center, it is not realistic in the broader setting of Internet Telephony.

JCC is a "Distributed JTAPI" that provides the communication support for multiple Providers to coordinate and interact in order to provide advanced services. For example, the JTAPI model assumes that the Provider object does not change throughout the lifetime of a Call. It is not clear how the JTAPI model could be used if the user wants to have different providers for different portions of the network traversed by a single call or different legs of a multi-party call. JCC overcomes these limitations.

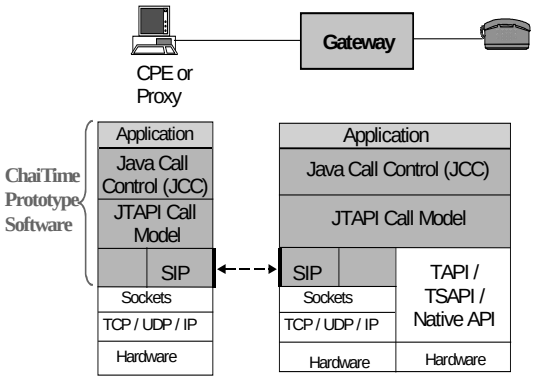


Figure 4 ChaiTime prototype software layers

The main goals of JCC are to: (1) provide for greater flexibility in service creation e.g. support dynamic service download, one-time service activation, and multiple parallel sessions between tow users; (2) facilitate rapid development of applications by providing higher-layer abstractions (rather than simply sockets or JTAPI); and (3) provide application portability across various communication protocols, e.g. SIP or H.323, and across platforms, using Java.

In our prototype implementation, JCC offers a high-level

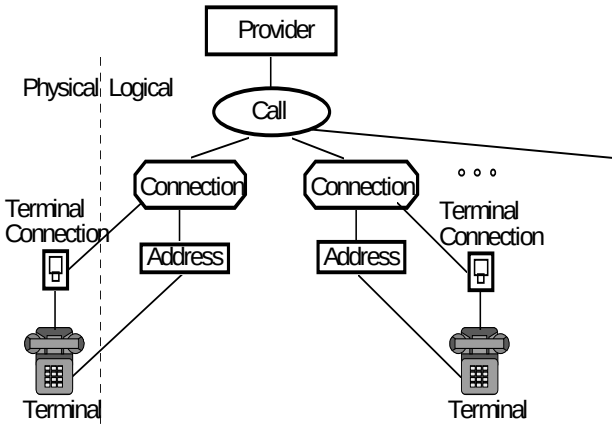


Figure 5 Objects in the JTAPI core call model

API to applications (see Figure 4), thus freeing them from the tasks of call-state maintenance, and although for reasons of simplicity and ease of implementation we use SIP as the underlying session control protocol, the applications are independent of our choice. Observe that JCC also essentially provides a means to integrate the IT and CTI paradigms. For instance, as illustrated in Figure 4, JCC can be implemented at a smart user terminal or proxy, where it can be used for advanced Internet Telephony applications. In addition, it could be implemented at a gateway that interfaces between the PSTN and the Internet, using telephony hardware and APIs f\or access to the PSTN and IP-based protocols and data communications hardware for access to the Internet.

In the rest of this section, we define the JCC model in more detail. We first briefly describe the JTAPI core model and then introduce the JCC extensions.

A. JTAPI Core Model

The Java Telephony API (JTAPI), published by JavaSoft [2], is a portable, object-oriented interface for Java-based computer-telephony applications. The API defines a core call model to support basic call setup, and a number of extensions, mostly designed to model call center features, multi-party

conference calls, call routing, etc. The core model consists of a few telephony classes and their relationships, as shown in Figure 5. Each object in the figure corresponds to a physical or a logical entity in the telephone world. The Provider is an abstraction of a telephony service provider. A Provider class manages Call objects, representing calls at various stage of progress.

Terminal objects represent the physical endpoint of a call, while Address objects are logical endpoints. Notice that each Address can be associated with multiple Terminals and vice versa, reflecting the standard configuration for a call center. Connections model the logical relationship between a Call and an Address. For a multi-party call, one Connection object is associated to each leg of the call. Finally, a Terminal Connection represents the relationship between one Connection and one physical Terminal.

The state of a telephone call is maintained by finite state machines associated with Call, Connection and Terminal Connection objects (e.g., when a call is answered by the called party, the originating Connection moves to the CONNECTED state). The complete definition of the state machines is part of the published JTAPI specifications.

B. Call Model extensions

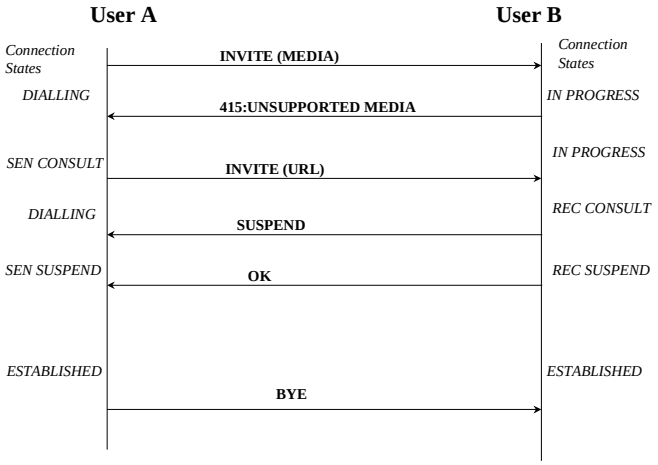
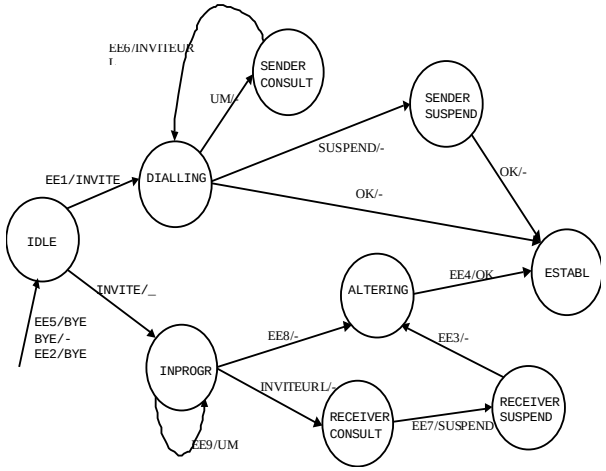


Figure 6 Message flow for dynamic service download

A Terminal Manager in ChaiTime is a user application that corresponds to a single JTAPI Address, plus all the Terminals that map to that Address. Thus, a TM is addressed using a JTAPI Address, and a Terminal becomes one of the resources available to the TM. Each Terminal Connection models the state of that resource.

The state machine for the JTAPI Connection object has been extended in order to support media negotiation during call setup. The number of API calls available to the calling parties for conducting the negotiation has likewise been extended (a complete description of those is omitted for reasons



KEY: External Events (EE):
EE1: Caller initiates call attempt. EE2: Time Out
EE3: Callee concludes action (e.g. download)
EE4: Callee accepts call. EE5: User hangs up
EE6: Suggest action (e.g. URL for download) to callee
EE7: Callee authorization. EE8: Callee supports media

Figure 7 State Machine for the extended Call Model

of space). Figure 7 llustrates the new state machine. The new states introduced are *Sender_Consult*, *Sender_Suspend*, *Receiver_Consult*, and *Receiver_Suspend*. All state transitions occur in response to events that are generated by the parties involved. One instance of the state machine is instantiated for each connection (leg) of a call. The sender and receiver states are used by the caller's and receiver's connections, respectively.

1. Dynamic service download

We illustrate the extended Connection state machine as well as our extensions to SIP using the dynamic service download example (Figure 6).

The caller initially sends an INVITE message to a party specifying the use of a particular media type. If the callee does not have a resource corresponding to that media, it can respond with an UNSUPPORTED_MEDIA message, which moves the caller's connection state to *Sender_consult*. In this state, the caller's TM has a chance to provide a suggestion to the callee, indicating a service provider, reachable through a URL, which may be able to supply the resource. If the caller accepts the suggestion, it initiates a negotiation with the provider. Since this operation takes place during call setup but is in fact conducted with a party not involved in the call, the connection for the callee moves to the *Receiver_Consult* state. In this state, the human user at the callee end of the call may be consulted, e.g. to authorize payment for or download of the service. The caller is notified of the suspension through a SUSPEND event, and its connection is moved to

Sender_Suspend. When the callee has terminated the acquisition of the resource, it notifies the caller. In case of an OK message, the caller then resumes its processing by moving to the standard (JTAPI) *established* state.

Notice that, at any point of time, either party can unilaterally decide to disconnect the call and return to the IDLE state. Also, various timeouts, not indicated in the figure, are set up in order to provide an upper bound on the negotiation time.

The remaining standard state transitions are defined in the JTAPI documentation [2].

2. Support for multiple concurrent sessions

A second example of the use of JCC for programming complex user-level sessions, together with media negotiation, is the support for concurrent sessions between the same TMs. Consider the example scenario of *Business Card Exchange.* Suppose that, during a session (after call setup), two users want to exchange their business cards from within that session. This is, incidentally, a common problem when business is conducted over the phone and a buyer is repeatedly requested to supply her name, phone number, address etc. (Notice that a business card information format is in fact being standardized [10], and software for performing this function in limited domains, e.g. between pairs of PalmPilot hand-held personal computers, is available.) Suppose also that such a card exchange service is in fact available to the users through a third party.

A straightforward way to accommodate this scenario in ChaiTime is to consider business cards as a new media type. Supporting mid-call negotiation to exchange this media, however, would further complicate the call model. The alternative we use is to start a new call, in which the new media is included during setup, and then manage the logical relationship between the old and the new call. This alternative provides an opportunity to introduce and experiment with dependent relationships between calls. It also simplifies the design of the Provider for the support of complex interactions, and offers more flexibility to the users. The same Address and Terminals share a number of active calls. Each call connection with the endpoint is represented by a different Connection and Terminal Connection object.

A higher-level session between TMs consists of all the calls shared by those TMs. Each call can be handled independently, or an explicit parent-child relationship can be enforced among them. In the case of the business card exchange, for instance, the TMs could specify whether terminating the main session would imply the termination of all the further calls that were setup during that session. Managing related calls is also useful for billing. For instance, calls that belong to the same high-level session can be billed together, possibly at a discount.

V. OBJECT MODEL AND IMPLEMENTATION

In this section, we provide some details of the existing prototype implementation of ChaiTime.

We describe the implementation using the Unified Modeling Language (UML) [4], a graphical language that is rapidly

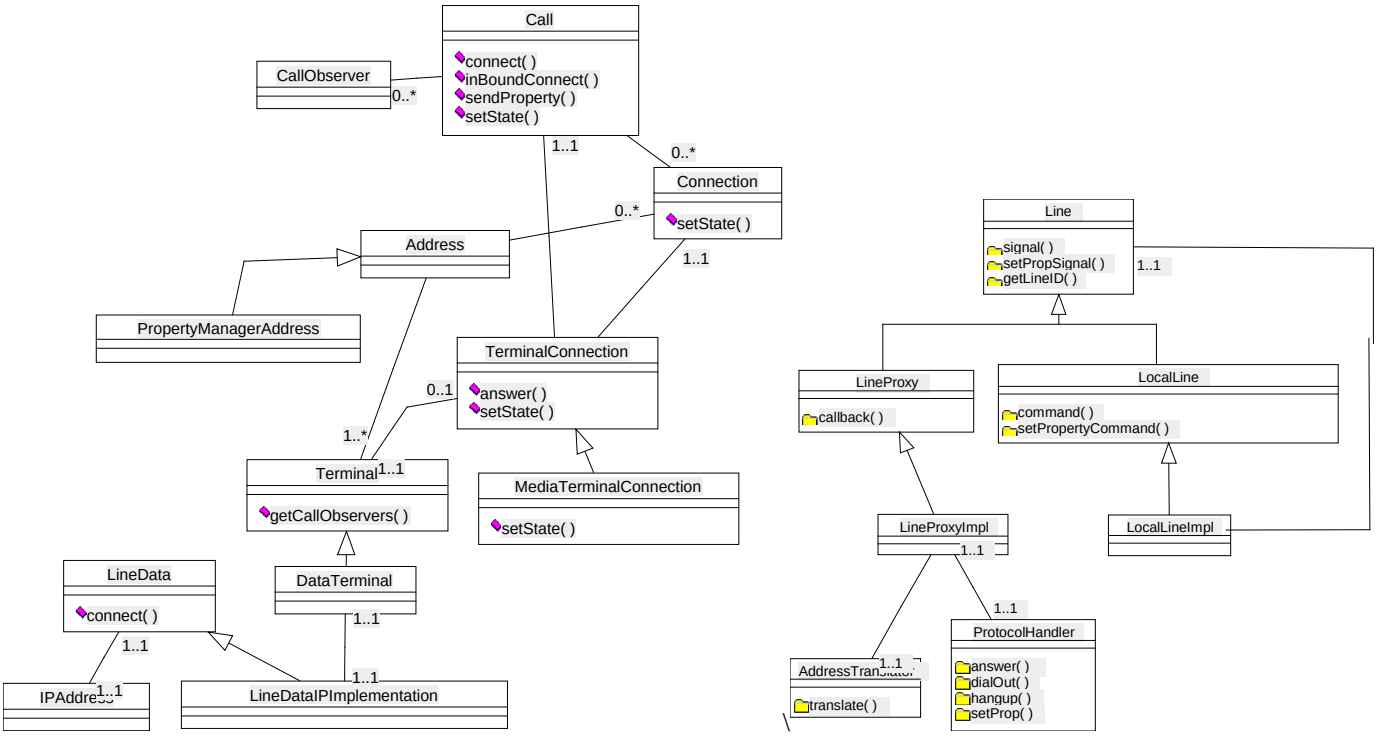


Figure 8 Top-level and line-level object model for ChaiTime

gaining acceptance as a standard for modeling and designing object-oriented systems. The UML model described here was prepared using the Rational Rose Case tool [11].

A UML object model as shown in Figure 8 describes the static view of a system in terms of classes and their associations. In the figure, the boxes denote classes and the undirected lines denote associations between them. Some of the methods available for the classes may be specified in the class box. An association can be labeled with its multiplicity at either end of the line indicating, for example, that a Call may have zero or more Connections (labeled as 0..*) but has a one-to-one association with a Terminal Connection (shown as 1..1). A specialization association, indicated by the triangle symbol, specifies that the class(es) at the bottom inherit the behavior of the single class at the top and specialize it in some way, typically by overriding some of the methods definitions or by defining new methods (the new methods are usually listed in the class box.). For instance, a 'DataTerminal' is a specialization of a generic 'Terminal'. In this figure, it should be easy to identify the classes in the JTAPI call model described earlier, and in fact each JTAPI class is represented by one class in the model[1]. The purpose of the Call Observer class is to receive and handle events emitted by a Call object. In the JCC event model, these events usually represent transitions in the state of the call. Examples of events include a new incoming call, a hang up, call answer or reject, etc. By subscribing to these events, Call Observers can monitor the progress of a call. When they are defined as part of the TerminalManager, they are used to inform the user on the state of the call and to ensure a consistent behavior for the TM.

JCC specialization's to JTAPI include the PropertyManagerAddress, which is just an address with the additional capability to query a remote peer for the value of its properties (e.g., a property could be the IP address of one of its attached services, for instance a Talk listener). Similarly, the DataTerminal class is a specialization of the JTAPI Terminal class that supports the notion of a simple data line (a different specialization could be for Voice Terminal).

In Figure 9 we present part of a UML *sequence diagram* that shows the steps involved during a *remote call*, i.e., when a call is placed from a terminal on one Provider to a terminal on a different Provider, normally situated on a different machine. These steps in general subsume those involved in a *local call* between terminals supported by the same Provider. In our case, a remote call proceeds over an IP network.

Unlike the static object model, a UML sequence diagram captures the dynamics of a particular interaction among objects in a system, as it unfolds in time. The columns represent named class instances, labeled in the boxes along the top of the diagram according to their classes. The arrows represent method invocations, with their input arguments, and their return values (in many cases only the critical parameters of interest are shown). Each arrow is labeled with a number, indicating its place in the sequence. Time proceeds downwards along the page. Note that only the invocation of a method by one object on another is explicitly shown; the re-

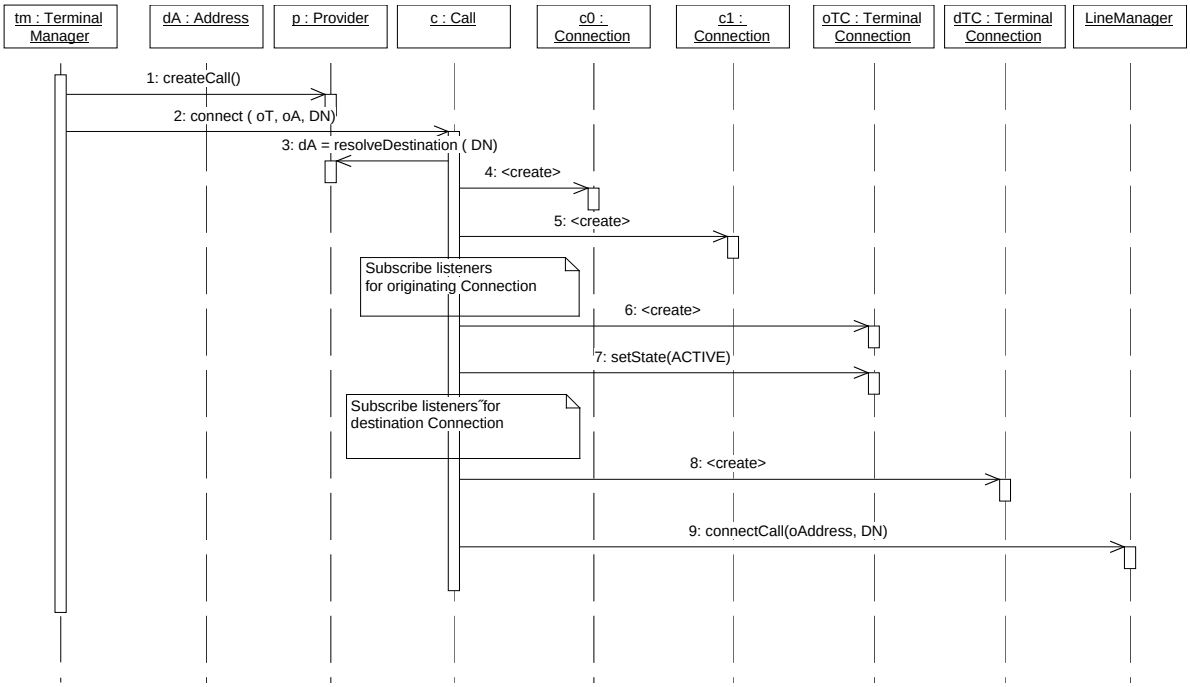


Figure 9 Sequence diagram for remote call setup

[1] Some of the classes are omitted from the figure for clarity.

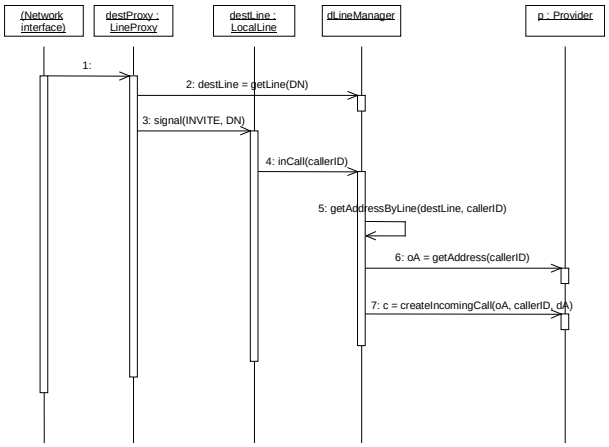


Figure 10 Remote call - receiving side

turn is left implicit. The boxes with a dog-eared corner are simply notes explaining or providing clarifying comments. For further details on UML see [4].

For the sequence diagram in Figure 9, the call begins when the user enters the destination phone number and presses the "Call" button on the ChaiTime client GUI. The Terminal Manager, or TM, object residing at the client device issues a *createCall()* method invocation to the Provider object, which creates a new Call object and returns a reference to it. The TM then contacts the Call object to set up the call, supplying the originating terminal id *oT*, the originating address *oA*, and the destination number *DN*. The Call object obtains the destination address *dA* from the Provider. At this point, the *dA* is still a reference to a logical address, i.e., to a JTAPI Address object. It is the task of the lower JCC layers to handle it appropriately. If the call is local, the actual destination Address object exists in the process space of the same Provider, and it can be contacted directly. In the case of a remote call, the Address is translated by the LineManager during the connectCall() into the actual IP address and port of the destination terminal (description of the directory lookup process involved is outside the scope of the present paper.). These differences are encapsulated in the LineManager and are transparent to the high-level Call object. The Connection objects representing the originating and terminating legs of the call are then created, and the CallObservers defined for the call are subscribed to the Call's and Connections' events. Note that the terminating Connection *c1* in this case is simply a proxy for the Connection object that will be created at the destination machine once the call setup signaling messages arrive at the destination. The use of proxies on both sides of a remote connection shields the higher layers from the details of remote signaling, by exposing a uniform behavior regardless of the type of the call. In a similar vein, when the Call object creates the Terminal Connection objects *oTC* and *dTC*, the latter is really a proxy for the corresponding remote object. Once the call model is setup, the Call object contacts the Line Manager to actually establish the connection through the signaling protocol.

The LineManager sets up a pair of Line objects, representing the two legs of the call, on which the actual signaling is carried out. The handler for the signaling protocol, in this case SIP, is entirely encapsulated in the Lines. In fact, in the spirit of abstraction, from the LineManager's perspective it is irrelevant whether the lines are local or remote, and which signaling protocol they use. Externally, a Line object for a remote destination behaves the same as for a local one. Internally, it is really implemented as a proxy, and the SIP signals to its remote peer are sent over sockets.

Figure 10 illustrates the corresponding low-level sequence on the terminating side. A Proxy for the sending Line receives the initial INVITE signal from its peer through the network interface, obtains a reference to a local Line that is responsible for the indicated DN, and then signals that Line. At this point the LineManager is responsible for translating the Line into a JTAPI Address object that can be notified, with the help of the Provider, of the incoming call.

VI. Discussion and Further Work

In this section we briefly discuss various aspects of the ChaiTime implementation and features.

Use of SIP. We use SIP [3] as the common signaling protocol among distributed Providers. Compared to other proposed protocols, such as H.323, SIP offers the advantages of specification and implementation simplicity, extensibility, and neutrality with respect both to the transport protocol and to address format. The protocol is text-based, much in the spirit of HTTP, and is designed to be part of a larger suite of session management protocols for the internet, that includes for instance the Service Location Protocol (SLP) as well as the Session Definition Protocol (SDP). Its simplicity allowed us to quickly implement the slight modifications we needed in order to support negotiation during call setup. Namely, we used SIP's features to define specific messages for suspending and resuming a session, for suggesting service providers' URLs, and for informing a party of the unavailability of a resource.

Deployment model. The ChaiTime architecture can be deployed in various ways to provide for scalability and, poten\tially, for fault tolerance. The two basic deployment scenarios can be shown using the configurations depicted for IT and CTI in Figure 3. In the distributed deployment (left side of the figure), each Provider is responsible for a partition of the global address space, and Providers communicate using SIP as their common signaling protocol. In the centralized, "PBX-like" configuration, one single Provider manages all the connections, as well as the interface with the PSTN. Our laboratory prototype runs in both configurations at present. Clearly, other intermediate configurations are possible, rang-

ing from applications that run on top of a dedicated Provider, to one centralized call center with only one Provider. Notice that Providers and endpoints can themselves be connected remotely, or they can be co-located on a host.

Scalability. Thanks to the distributed nature of JCC, ChaiTime can scale reasonably well with the size of the address space, i.e., ultimately, with the number of endpoints in the network. In fact, while one Provider manages an entire partition over the global address space, the partitioning itself is quite arbitrary. This makes it possible to introduce new Providers and to reassign addresses to Providers as needed in order to balance the load across all of them. In addition, by detaching the TMs from its Provider, one Provider space can scale with the number of managed TMs. Address resolution currently remains a centralized operation that can limit scalability. In ChaiTime, address resolution is done simply by mapping a logical address to the IP addresses and port number of the Provider responsible for that address. The mapping table is managed by a Translator server that is, in principle, distinct from all of the Providers. A call is setup by contacting the involved Providers through the IP address returned by the Translator. Each Provider will then internally route the call to the appropriate TM, by setting up the call model as described earlier. Several solutions are available for scaling the translation step to a large number of addresses, such as hierarchical resolution and local caching of the translation tables. We will investigate these issues in further work.

Another clear benefit of Provider distribution is its potential for fault tolerance. We can take advantage of the fact that each Call object contains the full call model, i.e., it models all the legs of a call, to provide replication across Providers for every single call. In the case of a single Provider (i.e., of a "local call"), a detached hot-standby Provider can be used for replication. ChaiTime does not at present support fault tolerance, but we plan to address this issue in further work.

VII. Concluding remarks

The ChaiTime system is in the process of ongoing development. A prototype implementation is currently operational in our laboratory and is being used for voice, text, whiteboard, and image-exchange sessions, as well as for advanced services such as dynamic service download.

We are continuing to experiment with the use of third-party JavaBeans software components both in the TM and the Provider. Our current experience suggests that while Java and JavaBeans provide a level of portability and name standardization, an additional level of standardization is required to allow seamless plug in of software components at run time. The current ChaiTime implementation overcomes this limitation by using a Component Launcher in addition to the Component Directory which manages the details for launching components. To some extent, this represents an industry-wide issue that needs to be addressed via standardization. We are also currently involved in instrumenting and configuring ChaiTime for performance measurements and for evaluating its scalability.

Acknowledgments. We thank the anonymous referees for their helpful comments.

References

[1] A. Lazar, "Programming telecommunication networks," *IEEE Network*, *11*, 5, 8-18, Sep./Oct. 1997.

[2] Margulies, E. (ed.). *Understanding Java Telephony*. Flatiron Publishing, 1997.

[3] Handley, M., H. Schulzrinne and E. Schooler, "SIP: Session Initiation Protocol", *IETF Draft*, draft-ietf-mmusic-sip-04.txt, Nov. 11, 1997.

[4] C. Larman, *Applying UML and Patterns*, Prentice-Hall, 1998.

[5] R. Englander. *Understanding Java Beans.* O'Reilly, 1997.

[6] R. Orfali and D. Harkey. *Client/server programming with JAVA and CORBA (2nd ed.).* John Wiley & Sons, 1998.

[7] J. Nelson, "CorbaBeans Proof-of-Concept." http://www.DistributedObjects.com/

[8] Sun Microsystems. *Servlet Tutorial*. Available from http://www.java.sun.com/products/jdk/1.2/cast-out/servlet/servlet_tutorial.html

[9] A. Thomas, "Enterprise JavaBeans: Server component model for Java," *Patricia Seybold Group White Paper*, Dec. 1997. Available from http://java.sun.com/products/ejb/white_paper.html

[10] Internet Mail Consortium, "vCard: The Electronic Business Card Version 2.1", versit Consortium White Paper, Jan. 1, 1997.

[11] Rational Software Corp., *Rational Rose/C++ software version 4.0.3*, 1996.