◆ Telco Meets the Web: Programming Shared-Experience Services

Robert M. Arlein, Dennis R. Dams, Richard B. Hull, John Letourneau, and Kedar S. Namjoshi

With current technology, it is relatively easy to create a new web service or a mashup combining several Web services. On the other hand, it is considerably more difficult to build services that combine telecommunications and Web aspects. Our thesis is that this is primarily due to the lack of a simple, yet expressive model for representing telco services. This paper presents a model, called the session data type (SDT), for the particular class of shared-experience services. The SDT model results in a considerable simplification of the design and the implementation of "telco+web" mashups. This paper provides several examples of such mashups and describes a prototype implementation.

Introduction

A *shared-experience* service [1] is one in which multiple participants share one or more media streams. The participants can play, contribute to, and control the shared media. The media can include video, audio in a phone call, or instant messages. The control of such services can be either through the media devices, or via a Web interface. Thus, the construction of a shared-experience service typically requires 1) dynamic addition/removal of participants to the experience, 2) multiple media, 3) reactivity to control events at end devices, and 4) a combination of Web and telecommunications interfaces. Our goal is to define a framework which simplifies the design and creation of such services. Before describing our design decisions, it is illuminating to consider several example services of this kind.

1. *Active conference call*. A typical conference call requires all participants to call into an audio bridge. An "active conference" service reverses this process: the service is linked to a calendar and sets up a bridge and calls the participants at the specified meeting time. Other features may include display of active participants' names on a Web page; the ability to switch at any point from video to audio; and the ability to set up side-conversations with other participants. (Note that we assume "video" to include sound and use "audio" for the audio-only case.)
2. *Family chat-room*. Any member of a family can call at any time into a continuously active family chat room. A child can get in touch with (call out to) a parent by pressing a particular phone key. Parents may choose to have a policy by which children are not allowed to bring their friends into a conversation unless a parent is present.
3. *Coffee-room experience*. In a coffee-room experience, people drift in and out, join an ongoing conversation, or start their own. A Web page might list

Bell Labs Technical Journal 14(3), 167–186 (2009) © 2009 Alcatel-Lucent. Published by Wiley Periodicals, Inc.
Published online in Wiley InterScience (www.interscience.wiley.com) • DOI: 10.1002/bltj.20395

Panel 1. Abbreviations, Acronyms, and Terms

AGI—Asterisk gateway interface
API—Application programming interface
CHSM—Concurrent hierarchical state machine
DS0—Digital signal level 0
FSM—Finite state machine
HTTP—Hypertext Transfer Protocol
IM—Instant message
IMS—IP Multimedia Subsystem
IP—Internet Protocol
ISDN—Integrated services digital network
NSF—National Science Foundation
OSA—Open systems architecture
PBX—Private branch exchange
POTS—Plain old telephone service
RMI—Remote method invocation
SDK—Software development kit
SDT—Session data type
SIP—Session Initiation Protocol
SM—Session manager
SMS—Short message service

ongoing conversations, perhaps featuring short snippets of ongoing conversations to spark interest.

4. *Virtual movie night*. A virtual movie night includes friends and family who watch a movie together from far-flung locations. Participants can carry on side-conversations and control the presentation with pause, rewind, and fast forward controls.

In the telecommunications world, frameworks such as Parlay*/Parlay X [5], Session Initiation Protocol (SIP) [8], and Asterisk* [2] have been defined for the purpose of developing services. While these frameworks can indeed be used to create shared-experience services, significant effort and a deep knowledge of the telecommunications network are required for their effective use.

Our main contribution is the development of a simple, focused model, which we call the *Session Data Type* (SDT for short). The SDT model has a small number of primitive concepts: *bubble*, which models a shared conversation over a single medium; *session*, which is a group of related bubbles, possibly over different media; and *event triggers*, which are used to communicate device actions back to an application. These basic primitives can be combined together in a variety of ways to achieve shared experience services. In particular, the services described above can be expressed easily and succinctly in terms of the SDT primitives.

The key motivation for our work, which underlies the design decisions made for the SDT model, is that a programmer should be presented with a simple, abstract, yet accurate view of the telecommunications network. By simplifying the authoring of the telco part of a shared-experience service, the model also simplifies the authoring of combined telco+web services. As shown later in the paper, our current implementation of the SDT provides a Hypertext Transfer Protocol (HTTP) interface to an application, resulting in a further unification at the transport layer.

The paper is organized as follows. The following section defines the SDT model and the abstract interface provided to a programmer. Next, we describe representative applications and illustrate how these applications can be designed in terms of the SDT. The subsequent section presents an abstract implementation architecture in which a server component, the session manager (SM), mediates between applications and an underlying communications network. This architecture is based on an abstract model of the network; the programming interface presented in the earlier sections is ultimately defined in terms of this model. Finally, we describe our current prototype, which was built using various open-source components; compare the SDT with other approaches; and present directions for future work.

The Session Data Type

The SDT model is defined in terms of its structural concepts, which are *sessions*, *bubbles*, and *endpoints*, and its behavioral (i.e., run-time) concepts, which consist of *state changes* and *notifications*. A shared audio experience such as a conference call is modeled as an audio bubble containing one or more participants, each represented by an endpoint. Every

participant can listen (read) or contribute (write) to an audio bubble. An application can alter the state of a conference in response to events which occur within the bubble (such as telephone pad key presses) or which arise through an alternative interface (such as choosing a Web page link). A session groups together multiple related bubbles, perhaps with different media types. In combination, these simple concepts suffice to program a rich variety of applications.

The bubble concept is not limited to audio: it can also be used for participants sharing a whiteboard or video. In addition, the SDT model does not prescribe a particular method for combining media from multiple participants in a bubble. For audio, the standard method is to mix the audio streams; for text, the standard method is to tag and interleave the individual contributions; for video, choices include tiling the individual streams or selecting a particular stream to display.

The run-time behavior for a bubble is defined in terms of a base, or standard behavior, permitting multiple extensions. Each extension defines a particular kind of bubble. The analogy here is with sub-typing: the base behavior defines the base type; all others arise as subtypes of the base.

The standard behavior considers a simple set of actions for each endpoint, shown in **Figure 1**, from the point of view of an entity interacting with an endpoint (a "mirror" to the view of the endpoint itself).

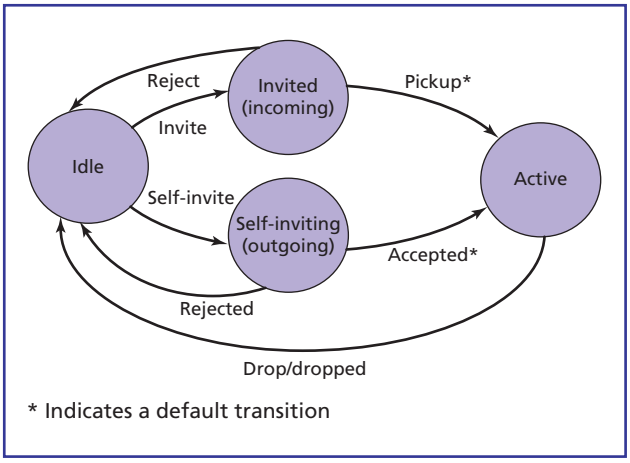

Figure 1.
Finite state machine.

An endpoint responds to an incoming "invite" message with a "pickup" or "reject"; it can "self-invite" itself, which may be "accepted" or "rejected"; and it can "drop" or be "dropped" at any time while in the active state. The pickup and accepted actions imply media connectivity.

Since this basic behavior is implemented by every bubble, a programmer is freed from having to define it explicitly and instead can focus on the logic of the application that is being built. Such an application may for example implement a Web-based service which incorporates telecommunication features into a Web page. Or, it can be more like a traditional telecommunications service that responds to actions (such as key presses) or events (such as incoming calls) which take place within the telecom network.

Sessions, Bubbles, Parties, and Endpoints

A bubble models the notion of a communication medium that is shared by a number of *parties*. A party is a person (*participant*) with a device, e.g., a telephone, or, if the communication medium is video, a device that can capture and display video such as a laptop with built-in camera. A device, as commonly understood, can have multiple *endpoints* (e.g., a laptop is a device with multiple ports). More precisely, a participant in a bubble is represented by an endpoint. A *session* is used to group bubbles.

An example is depicted in **Figure 2**. Bubble 1 is an audio bubble, and the parties within it use various types of devices. At the same time, within bubble 2, Charles and Deborah are sending each other short message service (SMS) messages. While Charles uses his desktop computer for the audio connection, he uses his cell phone for texting. Therefore, while in bubble 1 Charles appears as <Charles, desktop-pc>, in bubble 2 his presence is indicated by <Charles, cellphone>. Since these two bubbles are related by being part of the same conversation, they are part of a single session. In general, a session reflects the fact that the bubbles it contains are related to each other, but the nature of this relation is not constrained by the SDT—it is defined by the application.

Different sessions can exist concurrently. As an example, suppose that the participants in the conference described by session 1 in Figure 2 decide to split

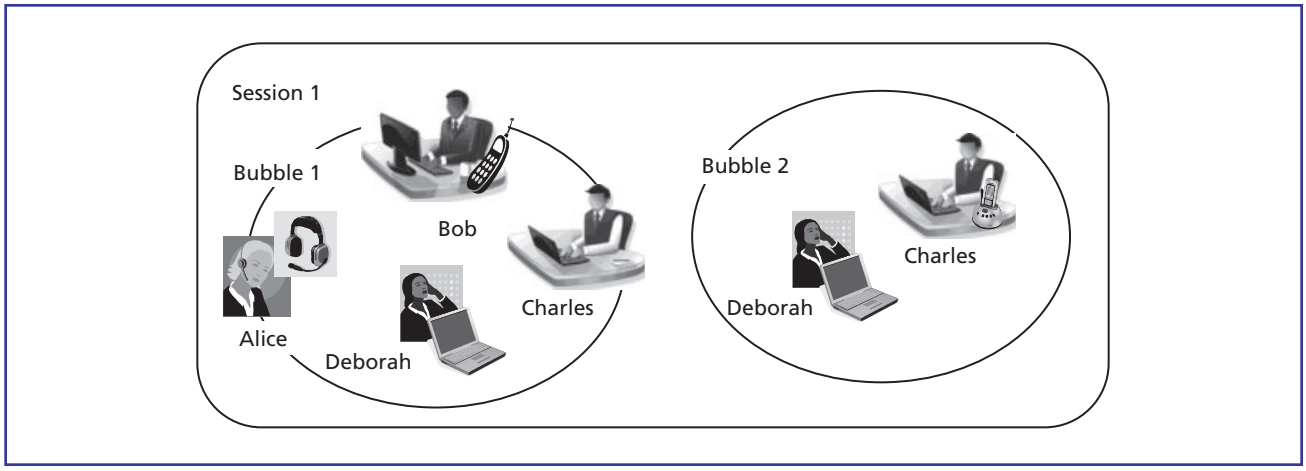

Figure 2.
Sessions and bubbles.

up into two separate groups: Alice and Bob on one hand, and Charles and Deborah on the other. This can be modeled by creating a new session with a new audio bubble, and by moving Charles and Deborah from bubble 1 into the new bubble.

Panel 2 details the most important structural operations of the SDT model. These are illustrated by code examples from our current Java* application programming interface (API). Variable declarations are not repeated across examples.

State Machines, Events, and Notifications

Once a session/bubble structure is created, there is a basic behavior, as explained previously. For instance, adding a party, "Joe," to an audio bubble results in a call being placed to Joe's phone; if he picks up, the default behavior is to add Joe's endpoint to the bubble.

Behavior is specified (as in language/automata theory) by sets of allowed sequences of *events* which may arise (such as "pickup" or "drop"). It is convenient to represent the allowed behavior by a finite state machine (FSM) over events. While state machines can be arbitrarily complex in general, those that define participant behavior are rather simple. Figure 1 shows a sample state machine for an endpoint. This state machine represents both the set of allowed events at each state (e.g., a pickup in the active state is disallowed) and indicates the default action whenever a nondeterministic choice is indicated. For example, in the "idle" state, the default action on a "pickup" is to move to "active."

A key behavioral concept for the SDT is a simple, yet flexible mechanism to alter this default behavior by means of *triggers*. A trigger is a pair consisting of a *condition* and a *handler*. A trigger is defined by an application and is activated when the associated condition becomes true. Conditions are specified through associated code. Examples of triggers can be seen in the code provided in **Appendix A**.

Formally, a condition is a conjunction of *predicates*. Each predicate is either an event predicate (such as "drop(b,Joe)," which is true when Joe disconnects from bubble b by event "drop") or a state predicate (such as "active(b,Joe)," which is true if Joe is in state active in bubble b). A well-formed condition has exactly one event predicate. Thus, a well-formed condition is true ("fires") when its associated event occurs and the state predicates in the condition are also true. A condition is evaluated within the scope of a session, a bubble, or a single party in a bubble, as determined by the application when it defines the trigger.

Whenever a condition fires, the application executes the corresponding *handler*, which is code that contains the intended response. For instance, if the condition "pickup(b,Joe) && invited(b,Joe)" fires,

Panel 2. Structural Operations of the SDT Model

Creating and removing sessions and bubbles.

```
AudioSession s;
AudioBubble b, b2;
s = new AudioSession(this.getSessionManager()); // this is a
  SessionManagerClient
b = s.getBubble(); // retrieve the pre-allocated bubble that comes with
  session s
b2 = s.addBubble(); // create another bubble in s
s.drop(); // clean up s and its bubbles
```

Adding and removing parties from bubbles, moving a party from one bubble to another

```
PartyInBubble pb;
PartyURI caller;
pb = b.addParty(caller,Role.ReadWrite);
pb.moveTo(b2);
pb.drop();
```

Merging bubbles within a session

```
b.merge(b2); // merge b2 into b
```

Manipulating endpoint connectivity

(each endpoint can be in "read-only," "read-write," or "write-only" mode)

```
pb.setRole(Role.Read);
```

Connecting one bubble to another

The source is viewed as an endpoint by the target bubble.

Additional media operations

Depending on the media type and source, additional media operations such as "fast forward" might be available on a bubble

Queries to find the set of parties currently in a bubble, and to inquire about parties' state and media connectivity

```
java.util.Set<PartyInBubble> ps;
ps = b.getParties();
```

the handler can test (with application-specific data) whether Joe should be allowed to enter bubble b, and, if not, prevent him from doing so by activating the non-default action (reject) from state invited to state idle. In this example, the trigger mechanism is used to enforce an access control policy.

The SDT model requires that a handler (i.e., the application response to a trigger) should appear to execute *atomically* with respect to other processing in the SDT. In our current implementation, this is ensured by suspending processing for *incoming* events for the affected session while a handler is executed. The handler code is free to interact with the SDT model to create new bubbles, for example, or to generate *outgoing* events, such as invitations to join a bubble. Depending on the application structure, handlers may execute asynchronously with the rest of the application code; in such a situation, the application programmer is responsible for proper synchronization of accesses to data structures shared between a handler and the rest of the application.

Predicates in conditions may contain *free variables*. For instance, "drop(b,x)", where x is a variable, is an event predicate that is true whenever *any* endpoint

in bubble b generates the "drop" event; the particular endpoint for which this is true which is then assigned to x (e.g., "*x*= Joe") and this value is reported to the application.

In our prototype implementation, trigger conditions are optimized by having the SDT evaluate them rather than the application. For this reason, defining a trigger may be viewed as subscribing to a notification mechanism; when the trigger condition fires, the subscribed application receives a notification, possibly including bindings of free variables from the condition. The precise timing of the notification is unspecified.

The final behavioral concept is the idea that an application may be prompted from within the telco network. The prompt contains the identity of the endpoint(s) involved in generating the prompt. In the SDT view, these endpoints are considered to be part of a fresh bubble. A programmer has to design a handler for such incoming prompts. This handler is unconstrained by the SDT model, but a typical action is to move one or more of the endpoints into an application bubble. This concept is useful for applications that are either initiated or controlled by calls made inside the network, as opposed to those where application control is managed via a Web interface. An example is the speed-conference application described in more detail in the next section.

Applications

In this section, we describe how applications can be built from the SDT primitives. This includes some of the potential applications discussed in the introduction, along with others which we have implemented. These examples are representative in that they combine aspects of the SDT with Web servers and database access. The descriptions are necessarily brief and focus on the use of the SDT primitives over supporting code, such as auxiliary databases or table entries. Appendix A contains a listing of the actual code required to program the speed-conferencing application described below.

A common thread in these designs is that bookkeeping and access control is performed primarily by the non-SDT part of the application, while the SDT model is used to set up and control the telco entities. This arrangement results in considerable flexibility and ease of implementation. "Control" actions, which result in state changes both to the SDT and the non-SDT parts, originate either from the SDT (for instance, key-presses originating from the network via the SDT), or from outside it (for instance, clicking on a Web link). Changes to the SDT state (for instance, a participant accepting or dropping from a call) are communicated to the application through the trigger mechanism. This is illustrated in **Figure 3**.

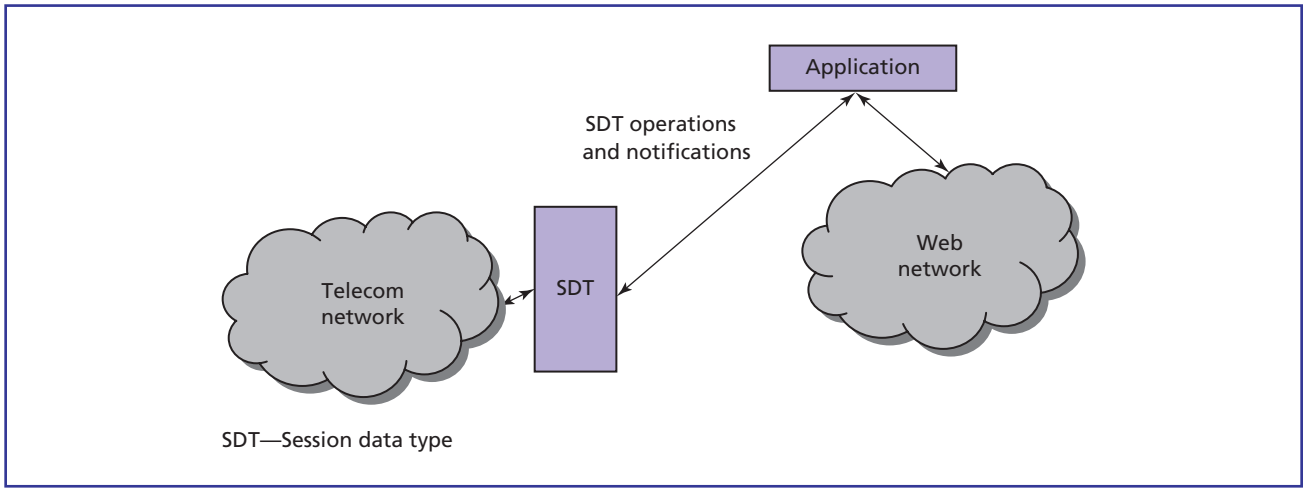

Figure 3.
SDT and the Web.

Click-to-Call

Click-to-call is a particularly simple application. Users select a group of phone numbers from a Web page and set up a conference call. This application can be modeled as a session with a single bubble; the call is set up by issuing invitations to the phone numbers in the selection.

Active Conference Call

The active conference call mentioned in the introduction is modeled as a single session with a single bubble. The initial set of participants is invited into the bubble at the time specified in the meeting calendar. The list of active participants is displayed on a Web page. This page is kept up-to-date by handling notifications sent to the application whenever an invited participant accepts (condition "pickup(b,x)") or when a current participant drops out (condition "drop(b,x)").

Family Chat-Room

The family chat-room concept is modeled as a single session with a single bubble. The application is triggered by prompts from the network generated by family members calling a pre-set phone number. The identity of the caller is included in the prompt; if it is a family member, the application moves the caller from the bubble associated with the prompt to the chat-room bubble and requests notification of key presses. By recognizing particular sequences of key presses as phone numbers, the application can enable friends and other family members to enter the bubble. Access policies, such as one where children cannot invite others unless a parent is present, can be enforced by querying the list of participants in the family bubble or by responding to notifications about add and drop events.

Coffee-room. The coffee-room experience is related to the family chat-room experience, where the "family" is now a dynamic group of people. Here, we expect multiple conversations to be active simultaneously, which can be represented as a session containing multiple bubbles. Conversations may be tagged as "public," where a recording of, say, the previous two minutes is posted on a group Web page. Either through links on this Web page or through key-presses the coffee-room application can be instructed to add someone to a conversation, to start a new one, or to switch from one conversation to another.

Virtual Movie Night

A virtual movie night application attempts to replicate the experience of viewing a movie in a room with friends and family who may actually be in far-flung locations. It can be realized as a session with a video bubble, which carries the movie, and at least one audio bubble, where a conversation takes place. The client software that displays the movie on each participant's viewing device might have the capability to send control commands (such as pause or fast forward), which are interpreted and implemented within the movie bubble by the application. This control feature is analogous to the use of key-presses in the previous examples. Policies regarding who can exercise control can be set up through the application, as in the family portal example.

Web-Based Configuration of Speed-Conferencing Keys

Through a Web page, phone keys can be configured to add/drop parties in an ongoing multi-party call. In the sample configuration shown in **Figure 4**, key 1 is configured to add (or drop, when pressed a second time) the user's home phone, and key 2 to add/drop an office phone. Since the conference bridge resides in the network, this application allows uses that go beyond conferencing capabilities implemented on the phone itself. A sample scenario might play out as follows: Suppose a user is talking on his cell phone when he arrives home and would like to continue the conversation on his landline home phone. To do so, he would press key 1 on his cell phone and pick up his home phone (which will ring); now both phones are part of the bubble. The cell phone can then be dropped, and the conversation can continue on the home phone. Notice that the transfer occurs without disruption to the conversation.

Whisper Conversations: Linking the Phone With Email

A group of students from Columbia University has implemented an application which combines several of the features described earlier and adds a new one: issuing invitations for a conference call via e-mail or instant messaging (IM). The application, which is programmed as a Web server, creates a Web

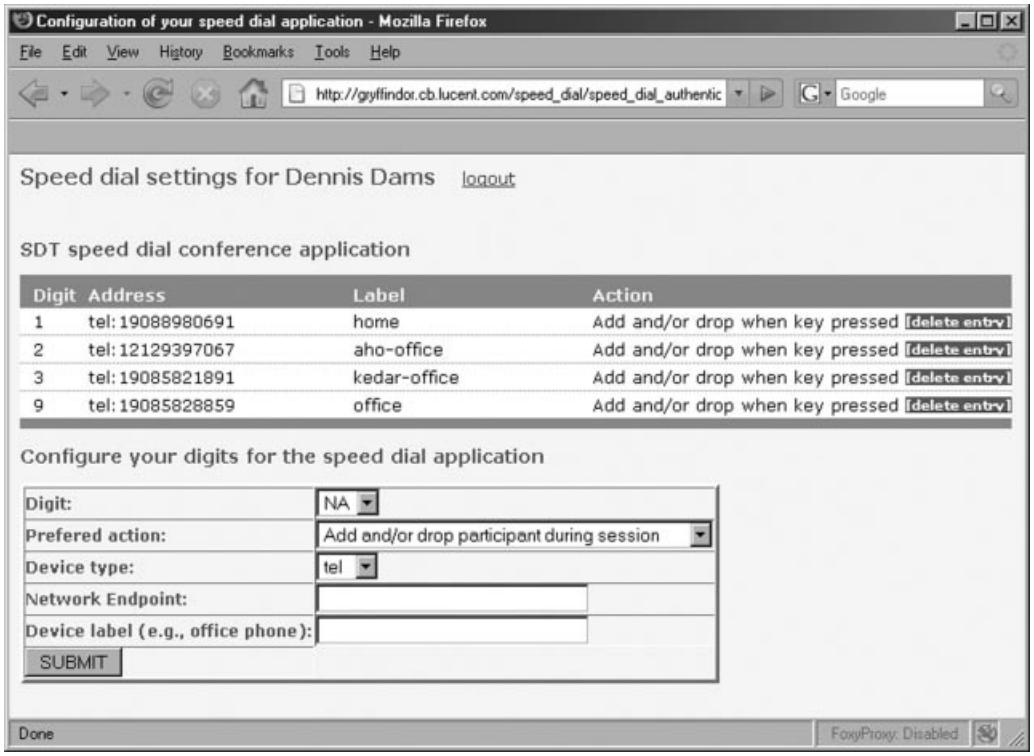

Figure 4.
SpeedConf Web page.

page for each user, with a list of contacts, and active and suspended conversations. The Web page displays the list of participants for each active conversation. An invitation from, say, Alice to Bob is sent to Bob's Web page if he is currently logged in, or to Bob's e-mail address if he is not. This is accomplished by creating a unique, time-limited key sent as a Web link. In either case, acceptance results in Bob's phone's being invited to the bubble.

This invitation mechanism can be used to create a "whisper" conversation between Alice and Bob, which proceeds in parallel with a larger conversation in which both are members. A second bubble is created for the whisper conversation—connected as read-only into the main bubble—and Alice and Bob are moved (after Bob accepts Alice's e-mail invite, and in a manner invisible to the larger conversation) from the larger bubble. A key-press notification can be used to toggle a participant in and out of a whisper conversation, in a manner similar to that described in the previous examples.

These descriptions, while necessarily brief, show that rather elaborate conversation settings, with nontrivial control policies, can be created using the simple session/bubble/trigger primitives. The current implementation defines a Java API whose methods correspond directly to the primitives of the SDT model. Thus, each of the actions described above, such as inviting a participant, moving a participant from one bubble to another, creating a notification trigger, and constructing a handler, can be programmed easily, often with a single line of code. In fact, for the speed-conferencing and whisper session applications, the majority of the code handles nontelco aspects such as database access and the Web interface.

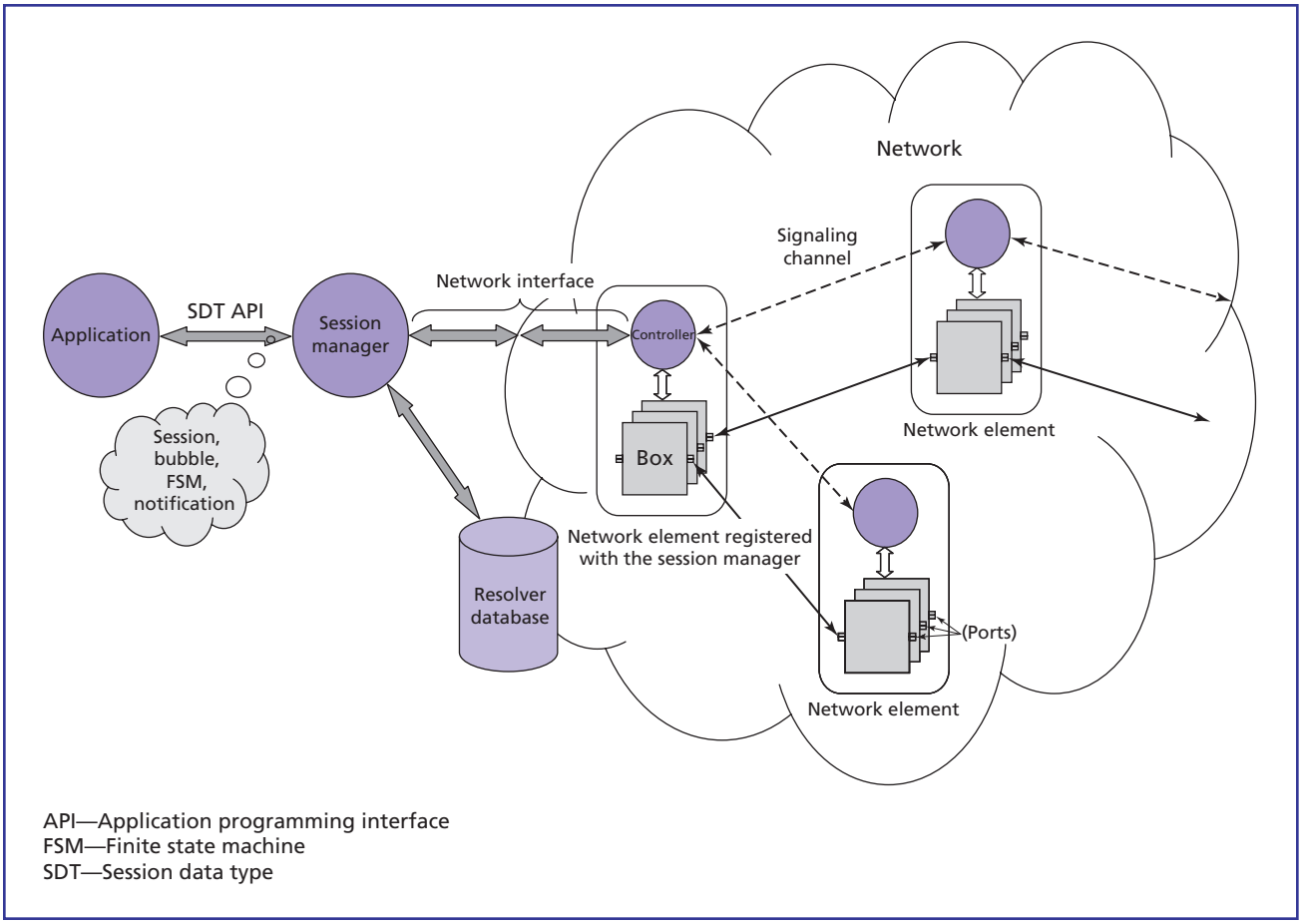

Figure 5.
SDT architecture.

Architectural Design

In this section, we describe in abstract terms an implementation of the SDT model. This implementation provides a programming interface for the creation and management of telecommunication sessions that involve multiple participants and multiple communication media, as described previously. It presents the programmer with a collection of concepts like bubbles, parties, and state machines; manipulating these entities causes the network state to change. In order to define these effects more precisely, we need to consider the context of the implementation: the telecommunication network. It suffices to consider an abstraction of the network that presents its relevant features and leaves out details that are of no concern to the SDT implementation. This abstract network is defined using a small number of relatively simple concepts; **Figure 5** provides an overview. The network and the session manager are described in more detail below. A third component shown in the figure, the resolver database, stores and manages information about participants (people) and their communication devices, names, numbers, preferences, rights/permissions, and billing criteria. This database is outside the research focus of our work.

A Model of the Network

We model the network as consisting of *network elements* and *links*. A network element may correspond to a single physical device, e.g., a telephone or a network switch that provides voice conferencing, but the mapping need not be one-to-one. Inside every

network element, we distinguish the traffic plane from the signaling plane. Media traffic enters and leaves the element via links. A *box* inside the element serves as a connector for several links and acts as a "mixer" that outputs some combination of its *in-ports* on each *out-port*. A single element has, in general, a collection of boxes available. Allocation of these boxes, connecting links to them, manipulation of the particular kind of mixing performed by a box (which may include temporarily "muting" a port), and freeing boxes upon completion are operations that are carried out by the element's control unit, called the *controller*. Operations on boxes and links include mechanisms to:

- Allocate and release a box.
- Request and release a link with certain properties (e.g., quality, bandwidth, or encryption level).
- Connect and disconnect a link to/from a port on a box.
- Set and retrieve box and port properties, e.g., in order to "mute" a connection.

The controller receives and sends signaling messages, which travel over *signaling channels* that are separate from the traffic links. Operations performed on boxes and links are the result of some request on a channel coming into the controller from another element or from an entity outside the network (such as the session manager). The exchange of signaling messages between the elements takes place according to a *protocol* that defines the intended meaning (effect) of individual messages, as well as the order in which they are exchanged. The SDT design is protocol-independent in the sense that it does not presuppose any particular protocols and can be instantiated to handle any protocol. Note that here we use the term *protocol* to refer to both the general notion of an "exchange of messages" and particular schemes like SIP.

The establishment of a link between one of an element's boxes and (a box of) some other element may involve a protocol that is dependent on the particular box, e.g., on its media type, or even on an individual port of the box. Thus, the element's controller may implement several protocols, each of them associated with a box or port.

Network groundings. The abstract network has been conceived so as to provide a simplified, yet precise view of various possible concrete networks. We refer to each such concrete network as a *grounding* of the abstracted network (Figure 5); we also use this terminology when referring to individual network components (elements, boxes, or links). One or more network elements register with the session manager upon its initialization. The session manager interfaces with the network grounding by controlling and receiving notifications from the registered network elements.

For example, in our current implementation, a single, registered network element is grounded to an instance of the Asterisk softPBX, through which our session manager connects to the network. Thus, this Asterisk instance provides the grounding for the registered network element. Since Asterisk handles SIP as well as public switched telephone network (PSTN) calls, the "network" as depicted in Figure 5 grounds to a network that encompasses both SIP and PSTN capable devices. Asterisk can create so-called bridges that act as boxes for voice conference calls. It is connected to the PSTN by a gateway that connects to a T1 trunk. This trunk carries both the voice traffic and the signaling messages, using in-band signaling.

Alternatives. Alternative groundings also can be considered, e.g., using an IP Multimedia Subsystem (IMS)-based product such as the Alcatel-Lucent 5420 Voice Call Continuity server, or H.323 technology [4], to interface with the network. In order to keep the session manager as independent as possible from the particular grounding chosen, a network interface mediates between the session manager and the registered network elements. This interface provides operations to control and receive notifications from the elements, such as notifications of the progress of calls at the elements. Porting the SDT implementation to a new grounding requires the implementation of a network interface for the grounding.

Session Manager

Network access is provided by having the SDT control one or more network elements through the session manager. The network elements under its control are said to be *registered* with the SM. The registered elements are the only ones that the SM directly interacts with. Any effect that the SDT may

exert on other elements occurs as a result of message exchange (signaling) that is initiated, at the request of the SM, by a controller of a registered element. Messages from the SM to a registered controller represent requests to establish, manipulate, and drop connections to other network elements. Messages from the controller to the SM are responses to such SM-initiated requests, or they represent notifications about events that originate in the network.

As an example, consider the creation of an audio bubble with two participants. The session manager directs a registered element to create a box that will act as the "communication bridge." It then requests the controller of the registered element (the *registered controller*) to establish links from this bridge to both of the participants' devices. These devices are also viewed as network elements. Thus, the registered controller engages in a message exchange with the device controllers. In a simple instance, the notification from the registered controller back to the SM might be just success or failure, depending on whether the links have been successfully connected. In a more elaborate instance, there might be intermediate notifications informing the SM, for each device, that it has been successfully alerted, that a pickup has occurred, or, alternatively, that a timeout occurred after alerting. After the call setup succeeds, in-call notifications may notify the SM about the occurrence of key presses or hang-up events initiated at the devices.

Of particular interest to the SM is certain information about the states of protocols that are implemented in the controllers, such as the stage of call setup within a controller (e.g., on-hook, alerted, or in-call, in the case of a telephone), or, while in-call, whether any in-call events such as key presses have occurred. This information can be represented conveniently by a finite state machine. Therefore, as part of the SM, a collection of FSMs is maintained, each of which provides an abstract view of (or "mirrors") the controller of some network element that is relevant to the SM. The triggers (condition/handler pairs) of an application are in fact evaluated against the FSMs. The SM has the responsibility to keep the FSMs in sync, as far as possible, with the actual controllers that they represent.

Prototype SDT Implementation

This section describes a concrete prototype, built to the specifications described in the previous section. **Figure 6** illustrates the SDT environment from the application developer's point of view. For most applications, the SDT interface is just one of the many used by the application developer. For example, the applications described in the Applications section use a Web interface and database access, in addition to the SDT.

The internal implementation of the SDT (in its current form) is described in abstract terms in **Figure 7**. The SDT has two interfaces: one to the application and one to the underlying network. The application interface is responsible for acting on API requests and for delivering notifications for triggered predicates. The network interface is responsible for receiving events from the network (such as a phone "pickup"), for media connections (such as allocating a media bridge), and for connectivity with endpoints (such as sending an invite to a phone). Our implementation uses several open source products. The various components are capable of executing in a distributed environment and are interconnected via an Internet Protocol (IP) network.

The application interface is structured as a set of processes, one for each session. The session process, in turn, contains the state for each bubble and maintains the state machine for each endpoint in the bubble. The state machine is programmed using the concurrent hierarchical state machine (CHSM) package [3]. Predicates and notifications are handled through a rule-based execution environment called Drools [6]. Each predicate is compiled into the guard of a Drools rule; the action invoked once the guard is enabled is to notify the application. This, of course, requires a tight coupling between the state machines and the Drools rule engine, to make the engine aware of each state change. As a call is being set up, terminated, and during mid-call events, the state machine engine allows for an orderly progression between states, while invoking appropriate actions with each step. Notifications to the application can be either asynchronous (notify-only) or synchronized (notify-wait). In the latter case, network events destined for the specific session are queued until the application handler signals completion.

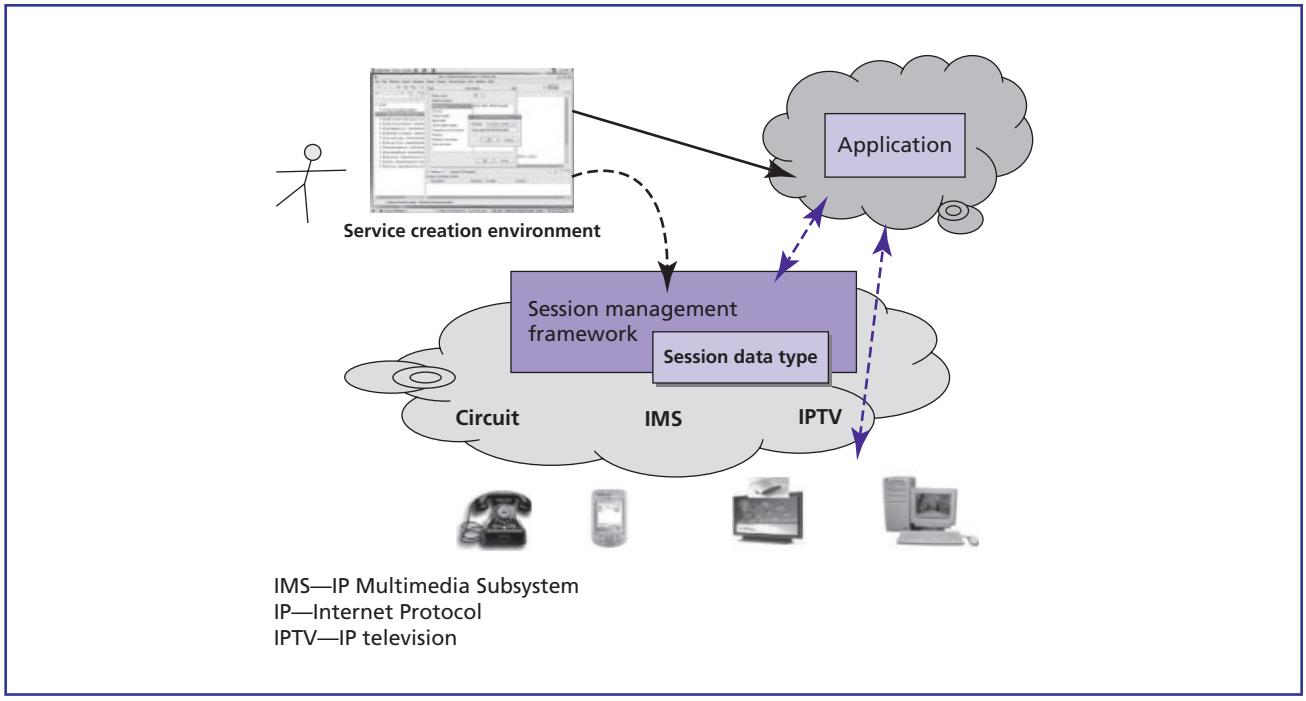

Figure 6.
Deployment architecture.

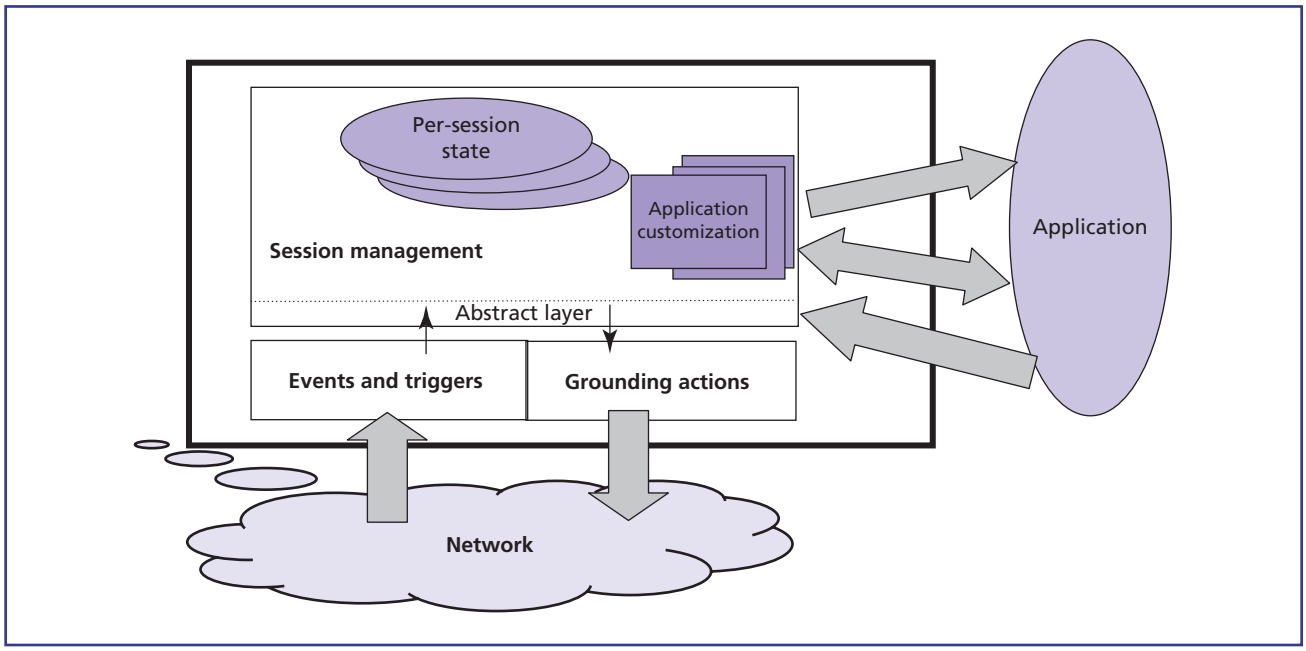

Figure 7.
Framework architecture.

On the network interface, we use an Asterisk PBX [2], connected to a Cisco* gateway. Asterisk provides two important capabilities: media (voice-only) bridges can be allocated using Asterisk (bridging is implemented in software), and calls can be made and received through Asterisk via the gateway. The Asterisk layer interface hides the details of the gateway configuration and access.

The Asterisk module is responsible for grounding the framework to this specific type of communication system. The Asterisk dial plan within the PBX is fairly simple and straightforward, but its details are tightly coupled with the Asterisk module. The module uses Asterisk APIs to react to network events, as well as to manage actions used to perform call set up and control.

Comparison to Related Approaches

We compare the SDT application interface to the interfaces provided by Parlay [5], SIP Lite [9], and Asterisk [2]. In this comparison, we look at the amount of knowledge required to develop a simple application (the steepness of the learning curve), and with Parlay and Asterisk we look for difficulties in building apps that are easy to build with SDT. Because of the complexity of those interfaces, we do not expect SDT to match their expressiveness.

Parlay

Several other efforts exist that aim at facilitating the integration of telecommunication functionality into (Web) programming. A recent example is British Telecom's Web21C development kit, which is being merged with the facilities offered by the Ribbit* software development kit (SDK) [7]. Together, this software provides APIs for different kinds of telecommunication functions such as setting up two-way and conference (multimedia) calls, messaging, authentication, and phone book services. A richer set of functions can be accessed by the Parlay X Web service API, which also includes third-party call control, billing, and presence. Even more power is enabled by the Parlay open systems architecture (OSA) API, from which Parlay X was derived. The general rule here is that more power comes with more complexity and a steeper learning curve. Indeed, these difficulties, as perceived by Web service developers, have driven the simplification of APIs starting from Parlay (or TINA, which came before it), via Parlay X and Web21C, to Ribbit. Each of these can be seen as derived from the previous by removing complexity in the API, e.g., by limiting the number of method parameters or by using fixed values for suppressed arguments.

Compared to Parlay X, the SDT offers a much more integrated view of multimedia communication. In the Parlay X API, there are different "islands" for two-way calling, multi-party calls, SMS, and multimedia messaging, each with its own set of methods. Illustrative of this is that with Parlay X, it is not possible to convert a two-way call into a conference call without dropping the connections and setting them up anew.

Another key difference is that the SDT framework is based on explicit finite state machine code that describes the behavior of endpoints involved in communication sessions, and applications have access to states and events at run-time.

Furthermore, the complete information and management of sessions can be handled by SDTs, whereas when using Parlay/Parlay X, the application has to explicitly manage different subsessions of the rich session as individual Parlay/Parlay X sessions. The SDT subscription language is richer than that of Parlay X, because it is based on FSMs and can include conjunctions of atomic predicates.

While it remains to be seen how our SDT API will be received by Web developers, one advantage it has is that there is a relatively simple model underlying the API. This can be illustrated by drawing a parallel to a programming library for, say, a stack. Apart from a set of method declarations, such an API comes with a model of what a stack is. Otherwise, operations like "pushing" and "popping" cannot be understood by the programmer. Since the API is supposed to allow for different implementations of stacks, e.g., by arrays or by linked lists, explaining what a stack is should be done in abstract terms that do not refer to the concepts (e.g., arrays) used in a particular implementation. The box-link-controller model that underlies our SDT fulfils a similar role as the abstract stack description. In contrast, a look at the specification documents that accompany Parlay X reveals that the meaning of

a method in Parlay X can only be understood in terms of the underlying Parlay implementation. In other words, Parlay X is a collection of convenience macros that may shorten the programming of services, but not the learning curve that is required.

SIP

A pair of SIP endpoints can exchange messages for the purpose of negotiating media streams between the pair which compose a session. Additional messages can be exchanged to modify or tear down the session. A SIP request and one or more responses compose a transaction. Each transaction occurs in a dialog. The messages involved in setting up and tearing down a session compose a dialog. Dialogs and transactions are modeled by a state machine, with messages effecting state transitions. Even using the high-level SIP Lite API [9], a SIP application writer must be aware of these state machines.

SIP services such as third-party call control or a conference call are built up from component sessions. Moreover, the standard supports multiple ways of creating these services. So, for example, a conference call may be built from multiple sessions from the conference endpoints to a central endpoint that has been customized to mix the component media streams. A full mesh conference call can be built from sessions between each pair of participant endpoints. In this case, software at the endpoints must mix the media streams.

It should be clear from the discussion that writing a SIP application requires a lot of knowledge. The information from well over 200 pages of SIP standards is required to write basic SIP applications. Additional lengthy standards are required to write conferencing or other types of applications.

Asterisk

The Asterisk PBX software is well known and broadly used by many small companies. It provides a means by which the business can specify the topology of their network and perform many basic call processing capabilities. Asterisk also provides a means to interact with external software, the goal being to foster the development of more sophisticated call processing applications. The PBX supports connections to various types of phone lines (digital signal level 0 [DS0], integrated services digital network [ISDN], and plain old telephone service [POTS]) and provides SIP registrar and proxy services over an IP network.

The basic call processing capabilities are specified in a proprietary language in what is termed a *dial plan*. These capabilities include development of interactive voice response navigation, as well as conference calls, voice mail box control, and call queuing. The API to external software is split into two categories: a management interface and an Asterisk Gateway Interface (AGI). Both are proprietary. These APIs allow a developer to perform various low level manipulations of the call flow by directing the PBX to take certain actions or by directing the caller to various features available though the dial plan. Developers need to know which capabilities are coded into the dial plan in order to make use of them. They must also be aware of the channels being used to carry the user's voice and other state information presented on the interfaces. In addition, there are also various commands, variables, and processing logic within the language of the dial plan itself which make coding for any non-trivial, well-behaved application a fairly sophisticated task.

The language of the dial plan is reminiscent of Basic. Each of the network's extensions requires a series of steps that specify what must transpire in order to process a call to that extension. This may include adding a person to a conference bridge or performing other treatments. Invocation of application logic on the AGI interface is one such step. Each of these Asterisk sub-modules has its own set of options and behaviors that make for a rich programming environment for those who choose to learn the intricacies of dial plan programming.

One particular aspect of coding that can be non-trivial is the ability to deal with mid-call user input, typically taking the form of a key press on the phone. This level of sophistication is necessary for applications that provide the ability for changes to the configuration of ongoing communication among multiple parties. In the case of Asterisk, such events require special coding within the dial plan. Programmers must make use of special switches for the conference call

module directing it to exit upon detecting such a mid-call event. The dial plan must also collect the input that is keyed in and send it via the AGI API to the application process. The dial plan logic is then suspended awaiting a return from that AGI call. It is possible that the next step in the dial plan is not executed, as the application has the ability to direct the dial plan to resume execution at some other location. The application then assumes more responsibility and must manipulate the channel of the user who hit the key so as to send him or her back to the conference bridge, if that is what the intent of the scenario dictates. Otherwise it is free to take any other action as outlined by its service logic.

Conclusions

We have designed a framework for building "shared experience services." These services can be built as a combination of a simple Web service and the SDT primitives. The SDT implementation appears to an application much like that of a Web service: the application interacts with it through HTTP or remote method invocation (RMI) connections to a session manager. The session manager handles the telecommunication aspects via its grounding in the network. We have built a single grounding based on Asterisk. While the SDT model necessarily limits access to a grounding, the SDT primitives are rich enough to build the wide range of applications described in the paper. In the sections of the paper that compare our framework to others, we show that it can be significantly easier to build services with our framework. Finally, the experience we have gained with our prototype implementation has allowed us to improve both the SDT and the grounding interfaces.

It is anticipated that future work will proceed along several dimensions. First, the design of the SDT model necessarily strikes a balance between expressiveness (full access to a grounding) and simplicity. We continue to look for ways of expanding expressiveness without compromising the simplicity of the SDT model. Secondly, our current implementation with Asterisk is based on audio media. We are experimenting with other media, such as video and text messaging. A third dimension is to experiment with different grounding layers such as SIP to validate the soundness of our grounding API. Another dimension is the scalability and performance of an implementation. Our focus so far has been on the model itself, and the implementation is a prototype built for experimentation with the concepts in the model.

Appendix A. Speed Conferencing Application Implementation Code.

The code below implements the speed conferencing application discussed in this paper. Some code, such as accessing the database with the users' speed settings, has been omitted. Comments between ellipses indicate all such places. All exception and error handling code also has been omitted.

```
package speedConf;

import sdt.mediator.SessionManager;

import sdt.mediator.PartyInBubble;
// ... more imports ...
// SpeedConf is the main class. It creates a SessionManagerClient object (see
// below) to deal with network interaction, and provides functionality for
// database access and interpretation of key presses.

public class SpeedConf {

// ... static variable declarations ...

  public static void main(String[] args) throws Exception {
    try {
```

```
        // Create a new SDT client, then connect it to the Session
        // Manager using the config. settings specified in
        // speedConf/SpeedConf_config:
        scc = new SpeedConfSdtClient();
        SessionManager.connect("speedConf/SpeedConf_config", scc);
        // ... connect to database, then wait for termination ...
    } finally {
        SessionManager.disconnect();
    }
  }
  protected static void keypress(String k) {
        //...database lookup of network endpoint (nep – e.g. a phone no.)
        // and action to be performed (act)...
        // nepStatus records for each endpoint, the corresponding
        // PartyInBubble value,
        // which is a pair that contains the party and the bubble.
        PartyInBubble pb =  nepStatus.get(nep);
        if ((act.equals("add")||act.equals("add_drop")) && pb ==  null) {
            pb =  scc.addToConf(nep); // add nep to conference
            nepStatus.put(nep, pb); // record new status
            return;
        };
        if (act.equals("add")&&pb!= null)||(act.equals("drop")&&pb == null){
            return;
        };
        if ((act.equals("drop")||act.equals("add_drop")) && pb != null) {
        scc.dropFromConf(pb); // drop nep from conference
          nepStatus.remove(nep); // record new status
            return;
        };
  }
  // ... methods for setting up and accessing database ...

// Class SpeedConfSdtClient provides the functionality to interact with the
// telco network.
class SpeedConfSdtClient extends SessionManagerClient {
        private AudioSession s;
        private AudioBubble b;
        private PartyInBubble initiator;
        private void createConf(PartyURI caller) throws SDTException {
            s =  new AudioSession(this.getSessionManager());
            b =  s.getBubble(); // a newly created session, like s, always
            // comes with a pre-allocated bubble, which is retrieved here
            initiator =  b.addParty(caller,Role.ReadWrite);
        }
```

```
        private void cleanupConf() {
                s.drop();
        }
        // Method register() is called by the Session Manager when an incoming
        // call into a registered (i.e., managed by the SM) network element
        // occurs.
        public void register(PartyURI caller) throws SDTException {
               createConf(caller); // create a new conference for each in-call
               // Create and add a rule that listens for key presses ("attention
               // events") by the initiator of the conference. The value of the
               // key pressed will be bound to rule variable x. The handler for
               // this trigger is defined below.
               Expr
cond=initiator.mkEvent(AudioBubble.SM.Events.attention("x"));
                              s.addTrigger(cond,new
                          attentionHandler("x"),TriggerKind.NotifyWait);
               // Create and add a rule that listens for a drop by the initiator
               Expr cond2 = initiator.mkEvent(AudioBubble.SM.Events.drop());
               s.addTrigger(cond2, new dropHandler(), TriggerKind.NotifyWait);
        }
        private class attentionHandler implements IHandler {
               public Response run(TriggerBindings tb) {
                      String attn 5  tb.lookup_var("x");
                      SpeedConf.keypress(attn);
                      return Response.Continue;
               }
        }
        private class dropHandler implements IHandler {
                public Response run(TriggerBindings tb) {
                      SpeedConf.nepStatus.clear();
                      cleanupConf(); // cleanup session
                      return Response.Continue;
               }
        }
        public PartyInBubble addToConf(String pid) {
               PartyURI p;
               p = new PartyID("", pid); // name of party not relevant, using ""
               PartyInBubble pb =  b.addParty(p, Role.ReadWrite);
               return pb;
        }
        public void dropFromConf(PartyInBubble pb) {
               pb.drop();
      }
}
```

Acknowledgements

We would like to thank Al Aho for many illuminating discussions regarding the SDT model and for suggesting the SDT programming project for his class at Columbia University. Thanks go also to McClain Braswell, Mohamed ElTahan, Ji Fang, Michelle Feng, and Joseph Kaptur, the students who participated in the class project in spring 2008, and to those who continued work on SDT applications during the fall. We are grateful to Rob Dinoff, Frans Panken, and Mans van Tellingen for their considerable help in putting together the prototype implementation. Richard Hull, in addition, is grateful for partial support by National Science Foundation (NSF) grants IIS-0415195, CNS-0613998, and IIS-0812578 for research described in this paper.

***Trademarks**

Asterisk and Digium are registered trademarks of Digium, Inc.
Cisco is a registered trademark of Cisco Systems, Inc.
Drools is a registered trademark of Red Hat, Inc.
Java is a trademark of Sun Microsystems Inc.
Parlay is a registered trademark of Information Builders Inc.
PLoP is a registered trademark of The Hillside Group.
Ribbit is a trademark of Ribbit Corporation DBA Duality, Inc.

References

[1] A.V. Aho, G. Bruns, D. Dams, R. Hull, J. Letourneau, K. Namjoshi, F. Panken, and H. van Tellingen, "Session Data Types: An Abstraction Layer for Shared-Experience Communications in Converged Applications," Proc. 11th Internat. Conf. on Intelligence in Service Delivery Networks (ICIN '07) (Bordeaux, Fr., 2007).
[2] Digium, "Asterisk," <http://www.asterisk.org/>.
[3] P. J. Lucas, An Object-Oriented Language System for Implementing Concurrent, Hierarchical, Finite State Machines, M.S. Thesis, University of Illinois at Urbana-Champaign, 1993.
[4] Packetizer, "H.323 Standards," <http://www.packetizer.com/ipmc/h323/standards.html>.
[5] Parlay Group, <http://www.parlay.org/>.
[6] Red Hat, JBoss, "Drools," <http://www.jboss.org/drools/>.
[7] Ribbit Corporation, "Ribbit SDK," <http://www.ribbit.com/>.
[8] J. Rosenberg and H. Schulzrinne, "Session Initiation Protocol (SIP): Locating SIP Servers," IETF RFC 3263, June 2002, <http://www.ietf.org/rfc/rfc3263.txt>.
[9] Sun Microsystems, "JSR 125, JAIN SIP Lite," Java Spec. Request, <http://jcp.org/en/jsr/detail?id=125>.

(Manuscript approved June 2009)

ROBERT M. ARLEIN is a researcher in the Services Infrastructure Research Domain at Alcatel-Lucent Bell Labs in Murray Hill, New Jersey. He received a B.S. degree in mathematics from the University of Wisconsin, Madison; an A.M degree in mathematics from the University of Michigan, Ann Arbor; and an M.S. degree in computer science from New York University. He is currently working on infrastructures for converged services.

DENNIS R. DAMS is a member of the Computing and Software Principles Research Department at Bell Labs in Murray Hill, New Jersey. He holds Ph.D. and M.S. degrees in computing science from Eindhoven University of Technology in the Netherlands. Prior to joining Bell Labs, he held a position as assistant professor at Eindhoven University. He has made research contributions to the field of program analysis and verification. Other interests include telecommunication and, more broadly, the relation between technology and interpersonal communication.

RICHARD B. HULL has broad research interests in the areas of data and information management, workflow and business processes, and Web and converged services. He is co-author of the book Foundations of Databases (Addison-Wesley); has published over 100 research articles in journals, conferences, and books; and holds six U.S. patents. Dr. Hull worked at Bell Labs Research, a division of Alcatel-Lucent, between 1996 and 2008, eventually taking the role of director of Computing and Software Principles Research. While at Bell Labs, in addition to pursuing research on semantic Web services, converged services, personalization, and data management, he was instrumental in developing and transferring new technologies into Alcatel-Lucent's product line, including the Vortex™ policy engine and the Datagrid data integration tool. Before joining Bell Labs, he served on the faculty of Computer Science at the

University of Southern California and was a frequent visitor at INRIA in France. His research has been supported in part by grants from NSF, DARPA, and AT&T. He was named a Bell Labs Fellow in 2005 and an ACM Fellow in 2007. He moved to the IBM T. J. Watson Research Center as a Research Manager in 2008.

JOHN LETOURNEAU is a distinguished member of technical staff with the Computing and Software Principals Research Area at Alcatel-Lucent Bell Labs in Murray Hill, New Jersey. He received his B.S. in computer science from Worcester Polytechnic Institute in Massachusetts and joined Bell Labs shortly afterward, participating in the One Year On Campus program and receiving his M.S. in computer science from the University of Southern California the following year. His current interests include finding ways to make a developer's job easier and more fulfilling without compromising quality and speed. Mr. Letourneau's career includes consulting as well as software developer positions on significant telecommunication products. He has conducted project architecture reviews and consulted on performance and reliability engineering and measurement for projects throughout Alcatel-Lucent. He is also active in the Pattern Languages of Programs, is a member of The Hillside Group, and is a participant at PLoP conferences. He has co-authored two previous Bell Labs Technical Journal papers on performance engineering.*

KEDAR S. NAMJOSHI is a member of the Computing and Software Principles Research Department at Alcatel-Lucent Bell Labs in Murray Hill, New Jersey. He holds Ph.D. and M.S. degrees from the University of Texas at Austin, and a B.Tech degree from the Indian Institute of Technology (IIT), Madras, all in computing sciences. His research interests span many topics in program analysis and verification. ♦