Using Speakeasy for Ad Hoc Peer-to-Peer Collaboration

W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, Trevor F Smith,
Dirk Balfanz, D. K. Smetters, H. Chi Wong, Shahram Izadi
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
{kedwards, mnewman, sedivy, tfsmith, balfanz, smetters, hcwong, izadi}@parc.com

ABSTRACT
Peer-to-peer systems appear promising in terms of their ability to support ad hoc, spontaneous collaboration. However, current peer-to-peer systems suffer from several deficiencies that diminish their ability to support this domain, such as inflexibility in terms of discovery protocols, network usage, and data transports. We have developed the Speakeasy framework, which addresses these issues, and supports these types of applications. We show how Speakeasy addresses the shortcomings of current peer-to-peer systems, and describe a demonstration application, called Casca, that supports ad hoc peer-to-peer collaboration by taking advantages of the mechanisms provided by Speakeasy.

Keywords
Casca, Speakeasy, peer-to-peer, ad-hoc collaboration

INTRODUCTION
Imagine the following scenario: Julius is on his way to a conference in New Orleans and has a layover in Atlanta. While waiting in the airport, he runs into his old friend and collaborator, Calpurnia, who is on her way to the same conference. They start talking about things they're working on and decide that they'd like to try to write a paper for a different conference with a deadline in a few weeks. They both have laptops with various capabilities: Calpurnia's can connect to the Internet over GPRS; Julius is out of range of his ISP, but is able to connect to the airport's 802.11b network to take advantage of certain local services. Both laptops have Bluetooth capabilities, and so are able to connect to one another.

To plan their writing, they exchange current contact information, information about the conference, notes, documents, pointers to related work, and so forth. This is information that is, variously, located on one of their two laptops, or on the network at one of their institutions, or on the public Internet.

Even though only a subset of these resources would normally be available to each participant—because of connectivity to different networks, access privileges, and so forth—they can still share the information they need in order to collaborate. For example, even though Julius does not have an Internet connection, Calpurnia does, and she is able to display web pages about related work on Julius' machine for him to see.

Both Julius and Calpurnia are also able to access and share devices around them, in addition to information. Calpurnia is able to print her related work pages using a nearby public printer by using Julius' connection to the airport network; she is also able to make her laptop's integrated webcam available to Julius, so that he can see if she is in her office during the coming weeks, as their collaboration continues.

This scenario illustrates a type of ad hoc collaboration that we would like to be able to support. Peer-to-peer (P2P) networking is a promising technology for supporting this type of application, but current P2P systems are lacking in several ways. The *Speakeasy* framework for recombinant computing [7] addresses many of the limitations of current P2P systems, and supports ad hoc collaboration.

In order to demonstrate this, we have built an application called *Casca* that is designed to support ad hoc collaboration across different networks and using a wide variety of computing resources. Casca allows users to easily create shared spaces (called *converspaces*), and use those spaces to share files and other resources, such as printers or webcams, with other users. Because of the

Figure 1: Using Casca in an impromptu collaboration

CSCW'02, *November 16–20, 2002, New Orleans, Louisiana, USA.*

extensible nature of Speakeasy, Casca is able to provide far-ranging functionality, such as display sharing, that the application was not explicitly written to support.

THE POTENTIAL OF PEER-TO-PEER COLLABORATION

Peer-to-peer (P2P) technologies hold out many advantages for supporting collaboration. Collaborative systems such as Groove [11] and WorldStreet [26] that take a peer-to-peer approach appear to be gaining adherents in the marketplace [4, 19]. Furthermore, as ad hoc networks such as Bluetooth begin to take hold, the P2P approach emerges as a good solution for off-the-Internet collaborations [16]. Although this aspect has been explored to a lesser extent, P2P systems seem especially promising in their ability to support "spontaneous" collaboration, where two people decide to have an unplanned interaction with one another.

The advantage of P2P systems comes from their potential ability to support *anywhere* collaboration. This benefit is architectural—by not requiring that all parties have mutual access to some common resource (such as a server, or a corporate intranet), P2P systems can potentially allow users to collaborate without regard to the particulars of the networks on which they happen to be. Further, these systems are easily supported by proximity-based networking technologies (such as Bluetooth), which can provide a certain natural ease of use with regard to initiating collaborations between participants.

We say "potential" in the previous paragraph because current P2P architectures have a number of limitations. First, P2P systems tend to be able to share only a fixed set of things, such as files. Systems like Napster [13] and Gnutella [8] are well-known systems that have, at their core, distributed protocols for file sharing. They cannot be easily extended to share *new* types of resources, such as nearby devices, or services that appear on the network.

Second, many of these P2P systems use a very limited form of discovery. Napster, in particular, is not a true P2P system: it requires the use of a centralized index server to allow searches over content. Only the actual exchange of data is done in a peer-to-peer way. Applications based on Gnutella, while decentralized, restrict their discovery to other instances of Gnutella-based applications only. These applications have not integrated or exploited the wide range of discovery protocols in use today, including [9, 21, 23], that allow access to external devices and services. Put another way, these applications are *only* about interconnecting with other instances of the same application, *not* about interconnecting with external resources. Even Groove, which is perhaps the most powerful example of a P2P collaborative system, is not a general-purpose tool for interacting with whatever resources happen to be at hand, and allowing them to be used collaborative; rather, it is restricted to sharing small applets that are hosted inside the Groove viewer.

Third, many of these systems assume that you will interact with all available peers, rather than letting you choose your collaborators.

Finally, these systems typically impose restrictions on the sorts of network transports they use. For example, applications written to use Bluetooth (e.g. Palm's BlueBoard [17]) cannot take advantage of other networks.

In the next section, we describe the Speakeasy infrastructure, and how it addresses the problems with current P2P systems outlined here. In particular, Speakeasy provides an open-ended framework for the sharing of arbitrary devices, services, and information. It is adaptive in the way it uses the network: Speakeasy applications can be dynamically extended to new data exchange protocols, new discovery protocols, and even new data types and user interfaces. This ability allows Speakeasy applications to discover and use resources on different networks. Finally, it allows application writers to create flexible models of sharing, according to the demands of the application.

Following the overview of Speakeasy, we will introduce the Casca application for peer-to-peer collaboration. We will show how the mechanisms in Speakeasy enable powerful collaborative tools to be easily constructed.

We will then conclude with a discussion of our experiences with both the architecture and the application and a comparison of both with existing systems.

THE SPEAKEASY INFRASTRUCTURE FOR PEER-TO-PEER COMPUTING

We have created an infrastructure called Speakeasy [7] that addresses many of the shortcomings of current peer-to-peer architectures. It is designed to allow the construction of applications that can use new devices, services, and networks, without requiring that the application have specialized knowledge of any of these.

Speakeasy provides a platform for extensible peer-to-peer computing that can be applied to a number of domains where flexibility is paramount, including ubiquitous computing and, as discussed here, collaboration. While the system is not, in itself, architected exclusively for collaboration, we will show that its ability to overcome the limitations of peer-to-peer architectures make it a good framework for building collaborative applications.

In the Speakeasy model, any entity that can be accessed over a network is cast as a *component*. Components are discrete elements of functionality that may be interconnected with other components or used by applications. Components may represent devices such as printers, or video cameras; services such as search engines or file servers; and information such as files and images.

In the scenario at the start of this paper, each shared resource—files, URLs, and printers—would be a component. Some of these exist on the laptops of Calpurnia and Julius themselves, while others exist on a public network or the Internet at large. As new components appear on the network, Julius and Calpurnia would be able to use them without installation of drivers or software upgrades.

Applications engage in a discovery process to find components "around" them on the network. These may be

components that are running locally (that is, on the same machine as the application itself), or on nearby machines (perhaps discovered using some proximity-based networking technology such as Bluetooth), or on remote intranets or the Internet itself.

In the scenario, both Julius and Calpurnia would run a Speakeasy-aware application that allows them to discover and use any components around them, and also allows them to share these components with each other.

As will be described, Speakeasy applications use no fixed set of discovery protocols; instead, applications dynamically adopt new discovery protocols as their underlying networks change. Likewise, Speakeasy applications do not depend on fixed data exchange protocols or data types; instead, they acquire new behavior as they interact with the components around them.

In the scenario earlier, this mechanism allows Calpurnia to discover and even use a printer accessed through the Bluetooth card in Julius' laptop.

The Speakeasy Architecture

Central to the Speakeasy design is the idea that all components expose their functionality via a small number of fixed interfaces that allow them to "snap together." Speakeasy applications use these interfaces to access component functionality; because the set of interfaces is fixed, any new component can be used by any existing application that understands the interfaces.

Establishing an *a priori* set of interfaces that must be used to describe all component functionality can obviously be limiting. The approach Speakeasy takes is to use these interfaces to allow components to deliver mobile code to their clients. Mobile code is portable executable code that can be downloaded over a network to a client, where it is executed. This approach allows a component to extend the behavior of applications that use it, to adapt them to new functionality.

Of course, mobile code is not a panacea. Any application that receives mobile code must still know what to *do* with that code. Systems like Jini [24], while they use mobile code, do not dictate a common set of interfaces that all clients are expected to understand. Speakeasy uses a set of four interfaces that dictate the ways in which applications can use the mobile code that they receive:

- Data transfer: how do components exchange information with one another and with applications?
- Aggregation: how are components "grouped" for purposes of discovery and containment?
- Context: how do components reveal descriptive information about themselves, and adapt themselves to their users?
- Control: how do components allow users and applications to effect change in them?

An application that uses a component will interact with it using one or more of these interfaces, depending on what the user wishes to do. Interactions between an application and a component involve only those two entities; no third party need be involved in the exchange, and so the architecture inherently supports a peer-to-peer style of interaction. Even the discovery process, which in many systems requires access to some directory service, is done through peer-to-peer interactions.

The **data transfer interface** is used by applications that wish to send or receive data to or from a component, or send or receive data between components. A source component, rather than providing data directly, instead returns an *endpoint* object to its caller. This is a bundle of mobile code that executes entirely within the receiving component, and extends it to speak whatever data transfer protocols the source uses.

For example, suppose in the scenario that Julius discovers a Speakeasy video camera. His application need not understand a suite of video delivery protocols in order to use this device. When he begins using it, the code necessary to speak the device's protocol is transparently downloaded to his application by the camera component itself, as shown in Figure 2. This code can be arbitrarily complex, performing adaptive compression, retransmission, and so on. In certain circumstances, components can even extend applications that use them to be able to handle new data types. For example, a camera can provide code to view MPEG2 video streams to its callers on demand. Users need not worry about installing appropriate applications or plug-ins for interacting with the components they encounter.

Just as the data transfer interface uses mobile code to extend applications to new communications protocols and data types, the **aggregation interface** is used to provide dynamic discovery behavior. This interface returns a mobile code-based object to callers that resembles a key-value map, where the values in the map are themselves components. Callers can also iterate through the

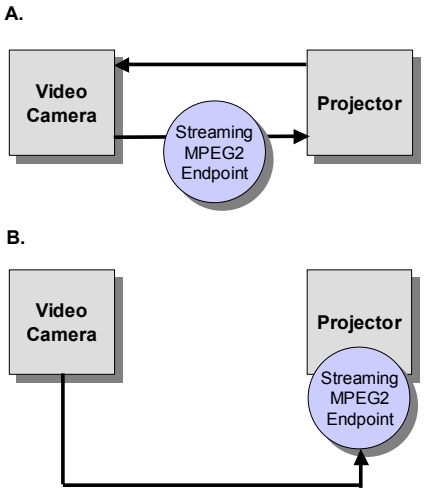

Figure 2: Data transfers in Speakeasy use source-provided endpoint code. In A, a video camera delivers a source-specific protocol endpoint to the projector. In B, the endpoint is used to fetch data using a camera-private protocol.

components in the map, search the map, and receive asynchronous notifications when the map changes.

The implementation of this map, however, is specialized by the aggregate component that returns it. As shown in Figure 3, a filesystem component can return a map implementation that provides a client-side implementation of the Network File System [22] protocol to access existing fileservers and return components representing files; a discovery component can return code that implements a network discovery protocol to find new components. Other aggregates may communicate to some back-end "bridge" service to access components on wholly different networks. This pattern of use allows an application to be written against a single interface that can provide a great deal of network flexibility, including locally executing discovery protocols, bridges, and so forth.

In the scenario, Julius runs a Bluetooth component locally that Calpurnia can access remotely. The code the component downloads to her application allows her application to communicate back to Julius' Bluetooth hardware, to access any Bluetooth device he can discover.

The **context interface** returns to a calling application an object that provides access to the metadata of a component. This may include information such as the component's location, its owner, and other descriptive information. One key aspect of the Speakeasy architecture is that, in order for any application to use a component's interfaces, that application must provide a context object describing *itself* to the component. Thus, all operations in Speakeasy are tagged with a set of descriptive information about who requested the operation. This pattern supports not only exploration and sensemaking of new network environments, but also accountability of actions.

In the scenario, both users can use the contextual information provided by the components around them to better understand their surroundings—what's on the airport network (and, hence, nearby them), what's on their work networks, what's actually on their laptops. When either of them performs an operation, identifying information about the circumstances are carried along with the operation.

Finally, the **control interface** can be used by applications to return to them a complete user interface, specialized by the component that provided it [14]. A component may potentially be able to provide multiple UIs including graphical UIs, voice UIs, and so forth. This mechanism allows users to control arbitrary aspects of a component's functionality, without requiring that their application be pre-built with control panels for every possible component.

In the scenario, Calpurnia may decide to walk through her practice talk with Julius. When she drops her PowerPoint slides on Julius' display component, the slides are displayed on his laptop and a set of controls for the slide show appear on her laptop.

Together, these four interfaces represent the shared agreement that we expect to be present in Speakeasy-aware

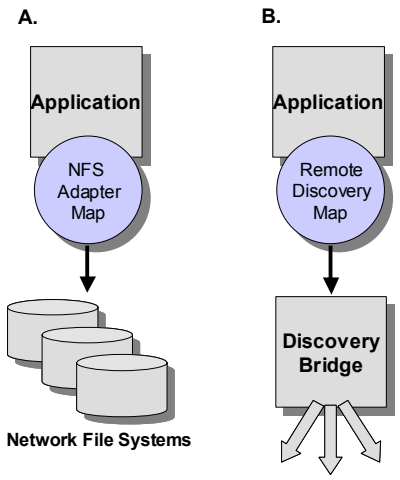

Figure 3: The two examples here show how the single aggregate interface can be applied to multiple uses via the use of component-specific objects. In each of these, a map object has been returned by a Speakeasy component to an application. In A, the map extends the application to speak the NFS protocols. In B, the map talks to an external discovery bridge.

applications and components. Components describe their functionality using one or more of these interfaces, while applications use these interfaces to extend their own behaviors in response to the needs of components.

There is, of course, an obvious drawback to the approach of dictating a small, fixed set of interfaces that programs use to interact with components: programs will have only very generic knowledge about the components with which they interact. Put another way, all Speakeasy components—whether filesystems or printers or video cameras—look largely the same to applications. In fact, it is this programmatic similarity that enables interoperability and extensibility to new components, and is *necessitated* by the Speakeasy model. If we restrict programs to being able to talk to those things they already know about, then those programs would not be able to talk to new types of things.

In such a world, the role of the infrastructure is to ensure that programs *can* talk to arbitrary components; users, on the other hand, dictate *when* and *whether* to use any particular component. The human has primacy in this model; this, in turn, places a set of requirements on the infrastructure, and on applications built using it.

Namely, the infrastructure must provide enough information to allow users to make informed decisions about the components in their environments—what they do, where they are, and so on. The context and control interfaces provide a simple form of this information: contextual information can be used by applications to organize the components around them, and the control interface provides a way for a user to directly interact with features of components not expressed using the other interfaces. Different applications will embed different

semantic models for how discovered components are grouped, shared, organized, and so on. The infrastructure must support these various uses. The second half of this paper, on the Casca application, discusses how an application can exploit the Speakeasy architecture to provide a user model compatible with the collaborative scenario described at the first of this paper.

Security

An important, but usually neglected component of P2P applications is security. In Speakeasy, the goal is not only to make it simple for a user to make components available to others, but also to provide means for that user to control who or what has access to a component.

As part of Speakeasy, we envision providing a security solution that 1) operates in a flexible and generic enough fashion to seamlessly work in the dynamic Speakeasy environment; 2) makes it easy for developers using Speakeasy to embed application-layer security semantics into their application; and 3) makes it intuitive for end-users to communicate their access control decisions.

We have begun building into Speakeasy an underlying security layer that achieves these goals. This security layer operates according to a few basic principles. First, all communication between components should be encrypted and authenticated. Second, it should not rely on the existence of any shared infrastructure. Third, it is the device "offering" a service that must make the final decision about who can access that service. Furthermore, that decision must be made on the machine actually running the service—it cannot be delegated to mobile code running on the requestor's machine. Fourth, while it is the service provider that makes the final access control decisions, the authentication must be mutual—the client must have assurance that it is indeed talking to the service it intended to talk to. Finally, while the nuts and bolts of encrypting communication can be handled relatively generically by the lower layers of the Speakeasy architecture, the decisions of who should be able to access what are determined by the semantics of the application. Therefore the security layer will call up to an application-provided access manager (or in its absence, a system-wide default) to get answers to access control decisions.

As we alluded to before, giving application developers a security infrastructure to use only solves half the problem; end-users should be able to specify their access control decisions in a simple and intuitive way. We achieve this usability goal by extracting final access control decisions from users' (not necessarily security-minded) actions, rather than in a separate, explicit (and usually skipped) security step. The user takes an action that reflects their intention, and the security plumbing adjusts to match. For instance, we allow the user to indicate that they want their PDA to communicate with a particular laptop by using IR (thus avoiding having to type in names or addresses). Under the covers, the system uses that brief exchange to distribute authentication data (cryptographic keys) sufficient to let those two devices encrypt and authenticate all of their future communications with each other [1]. More concretely, when two users meet, they arrange for their devices to talk directly to one another, while making sure that no other device is eavesdropping on (or manipulating) their communication. "Talking directly to one another" may be achieved by beaming a short message between them using IR (or by touching them, or by having them play to each other a short audio sequence). Using a two-stage protocol, the devices first exchange a small amount of data across this preferentially trusted channel, and then use that data to authenticate their further connection across the network. If the users do not meet face-to-face, they can carry out a version of this approach via Email, or by having the user type in information about their desired endpoint. These variants require higher degrees of user interaction, with somewhat lower degrees of assurance.

Platform Experiences

The Speakeasy framework has been implemented atop the Java platform, and uses Java bytecodes as its mechanism for mobile code, and Java Remote Method Invocation [25] system for its code loading, distributed garbage collection, and so forth.

To date, we have built over two dozen components, which explore a range of different behaviors. All of these components are interoperable using the basic facilities afforded by Speakeasy, and can be used by virtually any Speakeasy-aware application with no modifications to the application. These components include a whiteboard capture system; a digital camera; a filesystem that can provide access to any Windows NT or Unix directory tree to other Speakeasy components; an instant messaging gateway compatible with AOL and Yahoo!; discovery components for Jini, Bluetooth, and IRDA; a bridge that allows interaction with Cooltown devices; printers; projectors; microphones, speakers; video cameras; a component that can render a Powerpoint presentation as a slide show, and provide controls for that slide show; a screen capture component that can redirect a computer's standard monitor display to another component (such as a projector); and a number of others. We have also implemented a number of applications, including a browser, a description and evaluation of which are described in [15].

Together, the architectural features of Speakeasy address several shortcomings of current P2P systems. Speakeasy applications are not limited in the set of things they can use. As new components appear, existing applications should be able to use them seamlessly. Furthermore, applications are not restricted to discovering only files, or other instances of the same application. Speakeasy's discovery mechanisms provide open-ended support for discovering new types of things on new networks.

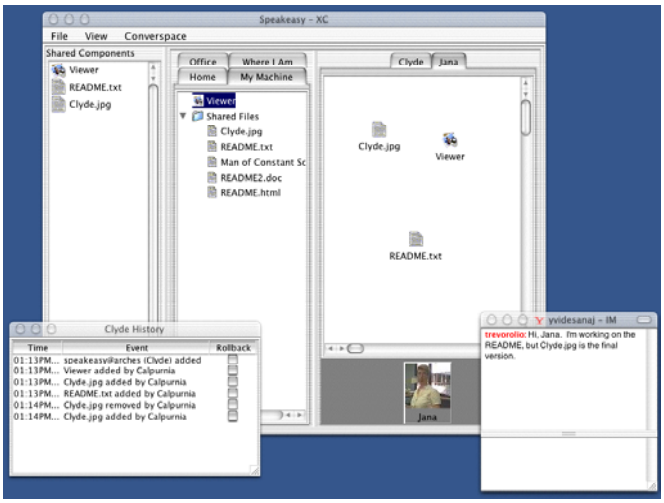

Figure 4: This Casca application. The current user is participating in a converspace with Jana, and is sharing two files and a viewer with her.

THE CASCA APPLICATION

Casca is an application, built on Speakeasy, designed to support ad hoc collaboration across different networks and using a wide variety of computing resources. It allows users to easily create shared spaces (called *converspaces*), and use those spaces to share files, devices, and services with other users. The main application window, shown in Figure 4, is divided into three regions. The rightmost is a tabbed pane of the user's current converspaces. Each converspace shows all components that have been shared by any members. The converspace's members are shown below. The center region shows all discoverable components, organized into categories including those on the local machine, at the office, at home, and in my current location. The leftmost region provides an always-visible list of all components that the user is sharing in any converspace.

Leveraging Speakeasy

Casca takes advantage of Speakeasy's mechanisms to use multiple discovery protocols and adapt to changes in available network connections and transports. Thus a user on, say, a laptop can access components running locally on the laptop, components running on an intranet to which the laptop has access (a work network, for example), "public" components that may exist in the environment, and so on.

Components, once discovered, can be used with other components or resources present on the laptop, or be shared with other users. Since components are the principal unit of sharing, Casca can be regarded as a highly "generic" application—it provides a framework and user interface for component discovery and interaction. The rest of the application's functionality is largely determined by the set of components that are available for use. As new components are discovered (or installed locally), they can extend the application to accommodate new protocols, data types, and user interfaces, using the Speakeasy mechanisms described earlier. Likewise, these components can also be readily shared with other Casca users.

For example, if a user has a video camera installed on his or her laptop, that user can share it with any other party, without that party having to install drivers or update software. It is by making the *component* the principal unit of sharing that Casca supports richer opportunities for collaboration than "simple" information sharing—the particular set of components that are shared enable the collaborative functionality. While Casca has no notion of cameras, microphones, or displays, videoconferencing capability can be added to the collaboration simply by having those components available.

Sharing Resources through Converspaces

A converspace represents an ongoing collaboration among a set of users. It contains the set of components accessible to anyone who is a "member" of the converspace. Users can be involved in multiple converspaces at the same time. Casca allows a user to easily create a new converspace to represent a new collaboration, and to add new members to existing converspaces, all of whom can then add components to it.

To revisit the scenario at the first of the paper, Calpurnia and Julius both run Casca on their laptops; when they first encounter each other, they create a new converspace to share their ongoing work. They then add components representing files and directories to the converspace. They might also add components like the "viewer", which allows other converspace members to show information on one's local display.

The converspace imposes a unified mechanism for access control and discovery on the "base" mechanisms provided directly through Speakeasy. Users can add *any* discoverable component to a converspace, including locally available components as well as those available through a variety of discovery mechanisms.

Local components—that is, components running on the same machine as Casca—are only available through *explicit* sharing. That is, they are *never* discoverable nor usable by any user, except those who are members of a converspace in which that component is shared.

Sharing a component via a converspace thus amounts to a limited form of "publishing" that component—it becomes visible and usable to the other members of the converspace, and only to those users. In essence, converspace sharing provides a separate discovery channel, independent of whatever other discovery channels may be in use.

Through the converspace mechanism, users can even provide access to components that might otherwise be undiscoverable by others. A user who shares a component need not worry about whether others can access it, since the act of sharing *makes* the component accessible to others.

The advantage of this model is that it does not require tedious access control lists, or a priori specification of access rights, or understanding of the differences of

discovery versus use, or complete knowledge of what resources might be discoverable by others. Not only is the use of such tools at a user level confusing, but it is also prone to error: users make mistakes, forget what resources are shared, and so on. There is precisely one concept in the Casca sharing model: if a component is in a converspace, it can be used by the other members of that converspace, and its access control automatically adapts as members come and go. The changing of access rights is incidental to the act of sharing [20].

Using the Speakeasy Security Model to Implement Converspaces

Under the hood, as individuals create and use a converspace, a number of security-related steps are taking place.

First, the system relies on the approach of using preferentially trusted channels to bootstrap secure interactions among users who share no a priori trust information with each other (perhaps using IR, as described previously). Thus, when Julius and Calpurnia meet, Julius simply "points out" Calpurnia so his application can find her. In the process of indicating that he wants to talk to Calpurnia (using IR, contact or another appropriate medium to "point"), Julius' and Calpurnia's devices exchange public key information and IP or Bluetooth addresses.

Now that they can securely identify each other's public keys, Julius and Calpurnia can now set up a secure connection to create a new converspace. One of them (let's say Julius) creates the space on their machine, and a "root credential" for it. The root credential is a self-signed certificate tagged with its connection to the space. That credential is stored with the space. Julius then creates a certificate for Calpurnia, designating her membership in this particular space through information contained in the certificate. These two certificates (Julius' self-signed certificate and the certificate he signed for Calpurnia) form the credentials Calpurnia needs to prove her membership in the converspace. Calpurnia stores these certificates with her representation of the space. Most importantly, as almost all communications in Casca applications occur over Java RMI (which is TCP-based), these certificates are sufficient to secure further interactions using SSL/TLS.

Note that none of this certificate exchange is visible to the users. Julius simply indicates (e.g. using IR) that Calpurnia is the person he means to add to a converspace, and all the rest happens implicitly and automatically. Also note that the same sort of exchange would be used by either Julius or Calpurnia to add yet another individual to the converspace. Depending on the security requirements of a converspace, either a subset of members (those that have some sort of special privilege) or all members can add other individuals to the space. In either case, Julius will be the only root certification authority, even though those allowed to add new members effectively work as lower-level certification authorities. The important characteristic provided by this architecture is independence: if Julius adds a third member to the group, Pluribus, then Pluribus and Calpurnia will be able to authenticate each other as group members without any intervention from Julius. The same is true between two other members invited into the group by Pluribus and Calpurnia, and so on.

When each user chooses to publish a component to other members of a group, they use their group credential for that group to secure the server end of requests for that component (thus group members, rather than components, are certified and carry group credentials). Each component that a user publishes uses standard Java mechanisms to indicate that a client that wishes to use it must do so using SSL/TLS. The client that connects to that component does so using the group credential for the converspace in which the object is located. It expects that the server, or service publisher, that it talks to on the other end will also have a credential indicating that it is a member of the same group.

Although this access control logic is application-specific—it "knows" about converspaces and the components inside them, the way that it is implemented is actually generic. We constrain all Speakeasy components in general to require the use of SSL/TLS connections. On both the client and the server, the components at the endpoints of the connection know that they must query an application level security manager to find out whether they should accept the connection (for the server) or should trust the server (for the client). The Speakeasy framework supplies several default security managers that could be used by a minimal application: e.g. one that allows all connections (no security), and one that only allows connections back to the same machine. The developer of an application such as Casca would install an application security manager appropriate to their application semantics (in practical terms, this is usually one fairly short class). In the case of Casca, the security manager knows about the existence of converspaces, and that members of a converspace can access components that have been published to that space.

Flexibility and its Discontents

A major aspect of Speakeasy's design philosophy was carried forward into the design of the sharing model. The emphasis on placing users in control of what is shared with whom reflects Speakeasy's core belief that users are the ultimate arbiters of interoperability among devices and services. There are, of course, a number of implications of this approach, foremost among them is that we permit users to make bad decisions. Users can, for example, allow access to components that are perhaps better left unshared. In particular, users can permit access to a component that exists on a private network, and which the other members of a converspace would otherwise not be able to even discover, much less use.

Users can also add new parties to a converspace without the approval of others. We believe this is a requirement for settings in which network connectivity is not a given, or for which we do not want to require all users to be constantly connected to one another. This design choice, however,

allows a users to make ill-informed (or even malicious) decisions about granting others access to a converspace.

Our belief is that these are acceptable—and even necessary—trade offs. In general, we believe that the benefits of permitting loosely-structured, informal collaboration outweigh the potential costs of bad decisions about sharing. There are a number of important requirements that result from this stance however: limited sharing, accountability, recoverability, intelligibility, and awareness of others and their actions.

Limited Sharing

First, sharing must be limited in scope to exactly the set of users defined by the user sharing the resource. As described above, in Casca this is made possible by the simplified sharing model that dictates that *only* components that have been explicitly placed in a converspace are shared with only the members of that converspace, and that there are *no* hidden effects that would provide access to anyone else. In fact, beyond components on my machine not being usable by others outside of a converspace, they are not even *discoverable* by others outside of a converspace.

Accountability and Recoverability

Second, social protocols work best when there is accountability [3]. Users need to be aware of who has done what in order for social protocols to be able to play out. To this end, we have built in a number of features to support accountability in the application, as well as recoverability from undesired user actions. Casca provides accountability by recording and making visible all actions that affect the composition of a converspace (components added or removed, users added or removed), as well as accesses to components that are carried out through the converspace. Figure 4 shows an the history window in the lower left hand corner. This visualization provides not only a tool for accountability, but also simple recall.

The history also provides a convenient user interface for recovering from simple forms of sharing mistakes. The history retains references to all the components that have ever been in the converspace. From the history window, they can easily be added back to the converspace, if they have been inadvertently deleted. So, for example, if another user accidentally removes a component that you were using with a third member, you can add it back quickly.

Intelligibility

Finally, intelligibility of sharing is essential for understanding. Even if no sharing happens except through explicit actions, users can still forget about what they've left shared, especially if they are engaged in a number of long-term collaborations, or a large number of collaborations These problems have been identified previously in the literature [2, 5]. Constant intelligibility of the state of sharing allows users to be aware of what's being shared, and to whom.

To prevent some of these problems with "invisible sharing," we strived to provide intelligibility about the state of sharing. Casca provides an always-visible display of what components the user has directly shared to others (see the left-most panel in Figure 4). A user can quickly ascertain whether any components are being shared, and what groups of people they are shared with. This panel also provides controls for globally changing the sharing settings for particular components.

Awareness and Communication

The benefits of providing *awareness* of others in a collaborative system have been well documented [6]. Casca provides a number of features for awareness of the state of others, and the state of the converspace itself.

Most importantly, each instance of Casca detects when the other members of a converspace are online. The system presents a graphical image of the membership of a converspace, and graphically indicates which are current online and which are offline. The system uses a simple heartbeat protocol to determine status; there is a short window during which transitions from online to offline and vice versa may go undetected.

The history feature, described earlier, allows users to be aware of any changes that have happened in a converspace since they were last connected, or of any changes that others made while they were offline. The history also records data transfers among components in the converspace. Since most operations in Speakeasy involve such transfers, for example viewing or copying a file, many operations can be recorded in this way.

Finally, Casca provides a number of tools to facilitate direct user-to-user communication. By clicking on the graphical representation of any converspace member, a user can access a menu that allows him or her to email the member, or send an instant message.

Implementing Distributed Converspaces

The state of a converspace is distributed among the members of that converspace. When components or users are added or removed, the changes are propagated immediately to all online members. Otherwise, each member tries periodically to connect to each offline member, and upon reconnection, a simple pair-wise synchronization protocol is used to update the state of each instance of the converspace.

The synchronization protocol supports offline work without the burden of more formalistic protocols, such as locking the converspace, or transactions that cannot be committed until the user is online, or the imposition of versioning (and a UI to resolve inconsistent versions), and without requiring access to some centralized server. This approach allows users to work individually or in small groups separate from other converspace members, and still be able to make changes to the converspace.

The intent with the sync protocol is simply to make the propagation of state as simple as possible from the user perspective; not to ensure rigorous global orderings of updates. Synchronization conflicts—such as two users adding the same component, or one offline member

removing a component while another adds it—are dealt with in a very simplistic way. The update with the latest timestamp "wins." Casca provides only a user interface to undo any undesired synchronization changes.

Histories are also synchronized. When one application instance carries out an action that results in a history record being generated, it creates the record and propagates it to all currently connected instances. Changes are propagated to offline members when they are reconnected. The history is not used to merge changes from multiple converspace instances. Instead, it is purely for human use to support accountability. History records are synchronized, but there is no attempt to resolve conflicts in the history.

DISCUSSION

The Casca application we have just described is intended to support flexible peer-to-peer collaboration. It depends critically on Speakeasy to provide many of its features. Most importantly, Speakeasy makes possible a runtime extension of functionality that would be difficult or impossible to accomplish otherwise. Typically, dramatic changes in functionality can only be provided by rewriting an application, which forces users to upgrade. This is made more problematic with a collaborative application, where each party has to have a compatible version of the application in order to work together. Speakeasy allows the application—and by extension, its users—to bring any resources in their environments to bear on their work.

Furthermore, we believe that Speakeasy made the task of developing Casca substantially easier than it would have been without it. We built a highly functional version of the core application from scratch in a little over a month with four developers. This task was made easier since Speakeasy abstracts away many of the unpleasant aspects of dealing with distributed components on the network—aspects such as discovering and connecting these components. Also the components we had already built were able to be integrated into Casca with no extra work, so the ability to share things like files, displays, printers, and video cameras was obtained "for free" because these already existed as Speakeasy components.

The Speakeasy framework provides these facilities to application writers, but at a cost. Namely, applications have only generic knowledge about a potentially open-ended set of components, discovery protocols, and so forth, that may appear in the environment. As we have said, we believe that this cost imposes a number of requirements on application writers, such as intelligibility, accountability, recoverability, and awareness. While these would be desirable features in any collaborative system, they are made more pressing by the additional flexibility and openness encouraged by Speakeasy.

Casca addresses these requirements in two ways. First, it supports sensemaking in complex environments. Users can organize the components around them by locale, owner, and other intrinsic aspects of components. The system preserves complete histories of the interactions in a converspace, to support accountability of users and their actions, and to allow a user to see how a space evolved to a particular state. The tenet of "always visible" sharing also supports this use.

Second, and more importantly, is the use of the converspace notion to unify access control, discoverability, and salience. From the user's perspective, a component is added to a converspace merely because it is salient to collaboration. The system uses this act to establish access control and component visibility accordingly—the model "collapses" many security operations down to a single user-visible concept. Most important to us was that the security machinery provide "real" access control, based on strong cryptographic techniques, but without "getting in the way" of what users want to do. Users never see a key, for example, or have to understand the security model in terms of certificates or access rights. Further, the model supports flexibility in sharing: if I add a user to a converspace, they *immediately* have access to the resources in it. There is no use of a central authority, or a requirement that all members be online in order for access to be granted. In other words, the security model is flexible from a user perspective, and is also compatible with the peer-to-peer model espoused in this paper.

The Speakeasy infrastructure supports the ability of applications to layer such semantics atop the core infrastructure. The contextual metadata interfaces, for example, allow applications to impose their own structure and organization on components that are discovered, based on proximity, spatial location, virtual location, or any other property that components use to describe themselves. The use of context objects to "tag" operations allows actions to be traceable and supports accountability through disclosure of identity and action. The Speakeasy security model allows applications to impose new access control and sharing semantics that—while they are defined by the application—are managed, enforced, and vetted by the components that provide service.

Our future work on these systems falls largely into two categories. First, we plan to continue refining the core functionality in the Speakeasy platform. The security infrastructure, in particular, is fairly new and we need more experience with it to ensure that it is compatible with other, divergent application security semantics. Second, we plan to continue investigating the user experience issues of genericity that are present (and, we believe, will necessarily be present) in richly networked environments. The Casca application is one such exploration into the domain of collaborative systems. We plan to continue this work, and also explore the issues of genericity in new domains.

RELATED WORK

We have described the Speakeasy framework and how it dovetails nicely with the requirements of spontaneous, peer-to-peer collaboration. Other frameworks and toolkits have attempted to address various aspects of the peer-to-peer collaboration space.

The Groove framework [11] provides mechanisms for easily initiating collaborations that can persist indefinitely, support for communicating and sharing across firewalls, and some degree of extensibility. However, they require all parties be able to contact a central server in order to initiate collaboration (the other parties' Groove ID's need to be looked up in a directory), so they cannot support off-the-Internet setup. Also, Groove allows collaboration spaces to be extended to incorporate any of a set of pre-established "tools" provided by Groove (e.g. whiteboard, voice chat, file space, co-browsing) or any third-party tools that are developed to conform to the Groove interface. In contrast to Speakeasy components, however, these tools are intended to be much more high-level and structured, and so do not offer extensibility at as fine a grain. Furthermore, Groove is unable to take advantage of "off machine" devices and resources—Groove is only able to talk to other instances of itself.

Other toolkits have addressed either the peer-to-peer aspects or the collaboration aspects, but not both. For example, JXTA [10] is a toolkit for building peer-to-peer applications of all stripes—not just collaborative applications but also file sharing, instant messaging, and so forth. It shares with Speakeasy an emphasis on network independence and interoperability, but does not directly support runtime extensibility. At the other end of the spectrum, collaboration toolkits like GroupKit [18] have focused on supporting tightly-coupled collaborative applications, such as collaborative document editors and real-time conferencing applications, but have not concentrated as much on supporting spontaneous collaboration or runtime extensibility.

DACIA [12], like Speakeasy, uses mobile code to create collaborative systems that adapt changes in resources and networks. Whereas Speakeasy is concerned with discovery and use of new resources by applications, DACIA is focused on dynamically relocating the functional units of a groupware application to adapt to changes in the environment. We believe that these systems target different application needs.

CONCLUSION

We have shown how the Speakeasy system addresses the shortcomings of current peer-to-peer systems, by providing flexibility in terms of discovery protocols, network usage, and data transports. This has been demonstrated through our development of Casca, an application that supports ad hoc peer-to-peer collaboration by taking advantages of the mechanisms provided by Speakeasy.

REFERENCES

1. Balfanz, D., Smetters, D.K., Stewart, P. and Wong, H.C., Talking To Strangers: Authentication in Ad-Hoc Wireless Networks. *Proceedings of Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA, USA, 2002).
2. Bellotti, V. Design for Privacy in Multimedia Computing and Communications Environments. in Rotenberg, M. ed. *Technology and Privacy: The New Landscape*, MIT Press, Cambridge, MA, 1997, 62-98.
3. Bellotti, V. and Edwards, W.K. Intelligibility and Accountability: Human Considerations in Context Aware Systems. *Jour. of Human Computer Interaction*, *16:2-4*. 2001.
4. Breidenbach, S. Peer-to-peer potential *Network World*, 2001.
5. Cohen, J., Out to lunch: Further adventures monitoring background activity. *Proceedings of ICAD 94, International Conference on Auditory Display*, (Santa Fe, NM, USA, 1994), Santa Fe Institute, 15-20.
6. Dourish, P. and Bellotti, V., Awareness and Coordination in Shared Workspaces. *Proceedings of ACM Conference on Computer-Supported Cooperative Work (CSCW)*, (Toronto, Canada., 1992), ACM Press, 107-114.
7. Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F. and Izadi, S., Challenge: Recombinant Computing and the Speakeasy Approach. *Proceedings of The Eighth ACM International Conference on Mobile Computing and Networking (Mobicom 2002)*, (Atlanta, GA USA, 2002).
8. Gnutella. http://www.Gnutella.com, 2002.
9. Goland, Y.Y., Cai, T., Leach, P., Gu, Y. and Albright, S. *Simple Service Discovery Protocol/1.0: Operating Without an Arbiter.* Internet Engineering Task Force Internet Draft. 1999.
10. Gong, L. *Project JXTA: A Technology Overview.* Sun Microsystems, Inc. April 25, 2001.
11. Groove Networks Inc. http://www.Groove.com, 2002.
12. Litiu, R. and Prakash, A., Developing Adaptive Groupware Applications Using a Mobile Component Framework. *Proceedings of Computer Supported Cooperative Work (CSCW)*, (Phil., PA, USA, 2000), ACM Press, 107-116.
13. Napster. http://www.Napster.com, 2002.
14. Newman, M.W., Izadi, S., Edwards, W.K., Smith, T.F. and Sedivy, J.Z., User Interfaces When and Where They are Needed: An Infrastructure for Recombinant Computing. *Proceedings of Symposium on User Interface Software and Technology (UIST)*, (Paris, France, 2002), ACM.
15. Newman, M.W., Sedivy, J.Z., Edwards, W.K., Smith, T.F., Marcelo, K., Neuwirth, C.M., Hong, J.I. and Izadi, S., Designing for Serendipity: Supporting End-User Configuration of Ubiquitous Computing Environments. *Proceedings of Designing Interactive Systems (DIS)*, (London, UK, 2002), 147-156.
16. Palm, I. *Bluetooth: Connecting Palm Powered(r) Handhelds.* October 22, 2001.
17. Palm Inc. Palm Bluetooth Card, 2002.
18. Roseman, M. and Greenberg, S. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer Human Interaction*, *3* (1). 1996. 66-106.
19. Scannell, E. Wall Street firms embrace WorldStreet's p-to-p offerings *InfoWorld*, 2001.
20. Smetters, D.K. and Grinter, R.E., Moving from the Design of Usable Security Technology to the Design of Useful Secure Applications. *Proceedings of New Security Paradigms Workshop*, (Norfolk, VA, 2002), ACM.
21. Sun Microsystems. *Jini Discovery and Join Specification.* January, 1999.
22. Sun Microsystems. *Network File System Protocol Specification (RFC 1049)*. 1989.
23. Universal Description Discovery and Integration Consortium. *UDDI Technical Whitepaper.* September 6, 2000.
24. Waldo, J. The Jini Architecture for Network-centric Computing *Communications of the ACM*, 42:7, 1999, 76-82.
25. Wollrath, A., Riggs, R. and Waldo, J. A Distributed Object Model for the Java System. *USENIX Computing Sys.*, *9*. 1996.
26. WorldStreet. http://wwWorldStreet.com, 2002.