SkeenZone: A distributed Android chat application and extensible middleware

Jonas Michel and Kyle Prete
The University of Texas at Austin
Austin, Texas, USA
{jonasrmichel, kylep}@mail.utexas.edu

ABSTRACT

Mobile computing devices are becoming ubiquitous parts of everyday life. This growth opens a development space for practical applications geared towards decentralized collaboration and coordination. The mobile development platform Android provides rich APIs for interacting with and harnessing devices' hardware components, but does not provide developers with adequate support for distributed computing.

We present *SkeenZone*, a lightweight and extensible Java middleware framework created to enable the development of distributed mobile applications, and *ChatHoc*, a simple distributed Android chat application for evaluating and demonstrating our middleware's ease of use. We describe two limitations of the Android platform that impose heavy restrictions on developers' ability to build and test applications involving ad hoc networks.

1. INTRODUCTION

The increasing ubiquity of mobile computing and sensing devices in our daily lives has engendered an entirely new development space for practical mobile applications requiring the support of accepted distributed computing concepts. For example, a distributed chat application may require chat messages be totally ordered on all cooperating users' devices; a decentralized collaborative whiteboard application may depend on users' devices obtaining mutually exclusive access to the "whiteboard" space for reading and/or writing; a distributed mobile workflow management system may desire that a notification be dispatched to a manager when certain employees' progress meet a set a requirements.

Currently, very little software support exists within the major industry-standard mobile software platforms, including Android OS, for creating applications geared towards decentralized collaboration. The Android development platform boasts robust application program interfaces (APIs) that enable easy access to and interaction with a device's hardware components (e.g., accelerometers, GPS sensors, Wi-Fi and Bluetooth cards, etc.), but it lacks software constructs to coordinate direct communication among multiple devices. This motivates the need for more readily available developer-friendly software tools for distributed applications.

In an effort to help alleviate this need, we introduce *SkeenZone*, an extensible Java middleware framework that implements Skeen's algorithm for total order [12] to enable totally-ordered message passing between devices and *ChatHoc*, a simple Android chat application to evaluate and demonstrate the middleware's capabilities. SkeenZone provides a canonical interface that we demonstrate may be easily implemented and extended to meet an application's specific requirements. Not only does the middleware guarantee a total order of messages, but allows any number of independent types of messages to be specified, such that a total ordering is only relevant within message types. ChatHoc is a simple distributed chat client implemented on the Android platform to demonstrate SkeenZone's use. The application allows a user to create chat "sessions," that each pertain to independent conversations, with independent groups of other users in a peer-to-peer (P2P) fashion over an internet connection. We elected to target the Android mobile platform because it is open-source and it runs applications written in the Java programming language, which we believe to be familiar to all types of developers.

Our intention with the SkeenZone middleware's use within Android applications is that it be fully functional on any Android device without requiring that a user obtain extra permissions (i.e., root permission) to unlock restricted functionality on their device. In other words, SkeenZone is intended for use within applications that will work on any out-of-the-box Android device. Obtaining root permission on an Android device, or "rooting" a device, is a complex procedure for an average user and certainly not encouraged (in some cases not even allowed) by device manufacturers. Rooting a device may sometimes even void a device's warranty. Therefore, we designed SkeenZone to be usable on non-rooted devices, which required some sacrifices in functionality (see Section 7).

The rest of this paper is organized as follows. Section 2 provides descriptions of related work, including design patterns, distributed computing concepts, and existing related projects. Section 3 describes the SkeenZone framework, our preliminary development prototypes, and ChatHoc application in detail. In Section 4 we explain how SkeenZone may

be extended to any type of Android or Java application. Next, Section 5 includes our testing and verification procedures. Sections 6 and 7 respectively include a discussion of technical challenges we overcame and Android-specific restrictions we found to limit the SkeenZone middleware and, more generally, applications targeting distributed applications in general. Finally, in Section 8 we conclude.

2. RELATED WORK

In this section, we discuss the major design patterns employed in the SkeenZone framework, distributed computing algorithms for total ordering of messages, the Android platform and relevant distributed mobile applications, and existing Java projects that enable the development of decentralized applications.

2.1 Design Patterns

A major goal of this project was to design an easily extensible library for distributed applications on Android. The focus of this extensibility would be in adding new types of messages and new handlers to handle received messages. The authors considered implementing a Visitor pattern[5], but rejected it because it required the *Message* types to be predetermined. Instead we implemented a version of Strategy combined with chain-of-responsibility [5], where a new *Handler* inherits the functionality of the old by calling up to the superclass's implementation if it does not handle the incoming *Message* type. This architecture will be explained in more detail in Sections 3 and 4. We also implemented a version of Model-View Controller (MVC) in which the view interacts with Android, the model handles and stores messages, and the controller interfaces between the two.

2.2 Total Ordering of Multicast Messages

The principal theoretical foundation of this project is Skeen's algorithm for total ordering for multicast messages between distributed processes [12], which is an optimization of Lamport's algorithm for total ordering [9]. A total order requires that for all messages x and y and all processes P and Q, if x is received at P before y, then y is not received before x at Q [6]. The concept of total ordering of messages lends itself nicely to a distributed chat client; ideally, we would like each user to receive messages in the exact same order. A total ordering of message ensures this requirement. In a system of N processes, Lamport's algorithm requires $3(N-1)$ messages per broadcast message. Skeen's algorithm offers an optimization in the case of multicast messages. To send a multicast message to G processes, a subset of the N total system processes, Skeen's algorithm only requires $3(G-1)$ messages. Our chat client application enables users to create chat any number of chat "sessions" (see Section 3) with small groups of other users. Chat messages sent in a particular session are only visible to the other users in that session and hence only need to be ordered with respect to the other messages in that session (i.e., they are multicast messages). Therefore, we choose to implement Skeen's algorithm over Lamport's because of it's complexity advantage in the particular case of the use of multicast messages.

2.3 Android Platform

Our implementation targets applications built on the Android platform for mobile devices [3]. Android is a widely used open-source software stack for mobile devices that was introduced in 2003. The Android platform is based on the Linux kernel and includes an operating system, middleware, and a verbose API [2]. We chose to target the Android platform because of its open-source doctrine, its wide use in today's mobile application community, and because it supports applications written in Java.

Several Android applications that support P2P fashion network interaction already exist on the Android market or are available on the internet [4, 10, 11, 1]. The Android SDK [2] even comes shipped with a Bluetooth chat application that enables P2P chat between two devices. However, none of these applications enable true scalable (multi-user) P2P network interaction; they rely on a cellular service provider or domain name system for routing and peer discovery. Futhermore, none provide extensible software interfaces; their design is extremely ad hoc and not easily made portable. These limitations are primarily a result of current restrictions built in to the Android platform (see Section 7). Our implementation is both extensible and designed at the boundary of Android's current P2P capabilities to enable multi-user P2P interaction in a distributed fashion.

2.4 Java Projects for Distributed Applications

Distributed mobile applications for practical use are not a new concept. Several major Java software projects that focus on distributed applications already exist. The Java Agent DEvelopment Framework (JADE) is a versatile middleware that is aimed at simplifying the creation and development of multi-agent systems on a varitety of hardware platforms [8]. JADE provides software constructs (an API) and tools to develop applications where mobile agents are communcating in a distrubted P2P fashion. The JXTA platform is an open-source Sun Microsystems project that provides a set of protocols enabling any connected device on a network to communicate in a P2P manner [7]. JXTA provides functionality for resource advertisement on a network independent of the underlying network implementation and topology, P2P communication, and dynamic P2P collaboration (group formation). We decided to avoid attempting to use these resources and create our implementation from scratch using point-to-point TCP sockets for three reasons. First, these projects are large and come with a high memory overhead; our implementation would likely only be using a small subset of the provided functionality. Second, and closely related, our primary objective with this project is to create an extensible software platform; it would not bode well for portability if our framework depended on other extraneous libraries that may become unsupported. Finally, the functionality provided by these resources would likely hide or interfere with that of our implementation.

We employ the use of JmDNS [13], a Java implementation of multi-cast DNS, to advertise and dynamically discover other devices running our application on a local area network (LAN). JmDNS enables a network service to register itself on a local network using a known identifier such that devices may listen for, identify, and spontaneously connect to one another. The JmDNS project is still under active development, but provides optional, but very useful, functionality for our implementation.

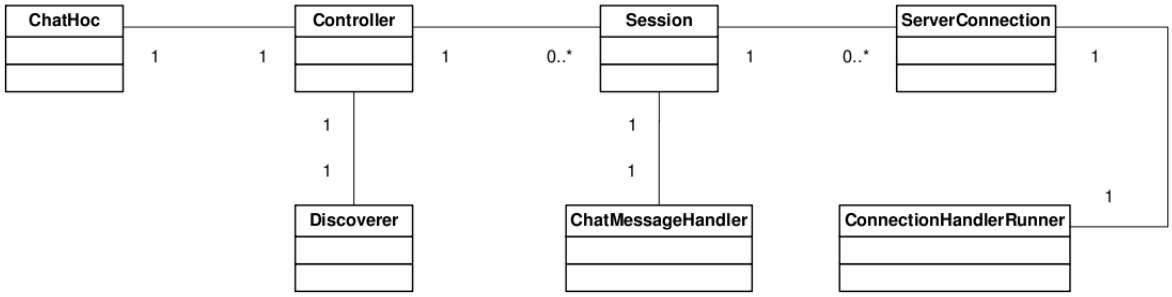


Figure 1: The major classes involved in Skeenzone's MVC

One of our prototypes was built on one of the former projects of the second author: Javassonne.[1] He and his teammates built a distributed tile game in Java in which each player's machine acted as both client and server, using heavy-weight Remote Method Invocation (RMI) messages to control each other. One of the coolest features of Javassonne was its lobby. When a player joined the network, he advertised using JmDNS that he was available for a game. Using the connection advertised in the lobby, another user could connect to the game and begin playing using RMI. We envisioned using this system to enable distributed applications on Android.

3. IMPLEMENTATION

In this section we describe the SkeenZone framework we built to support distributed applications, some prototype precursors to our final product, and the Android frontend chat application built on SkeenZone.

3.1 SkeenZone Framework

Major Classes The following code elements are shown in Figure 1. The *Controller* provides methods to display information from SkeenZone, including members of a *Session* and the user's IP address. Its other responsibility involves setting up a *ServerSocket* to listen for new connections, setting up a *ServerConnection* to handle the connection I/O and remote clock values, and assigning this *ServerConnection* to a *Session*. The *Controller* holds a set of references to the *Sessions* it has created, and handles destroying a *Session* when the user leaves it. In addition, the *Controller* sets up the JmDNS *Discoverer* module.

A *Session* contains everything involved in a chat session. It holds a *ServerConnection* for each connected chat client and a *MessageHandler* instance to read and reply to messages. The *Session* also records and updates the local clock for this *Session*. When its *Controller* destroys it, the *Session* destroys each *ServerConnection* it holds.

A *ServerConnection* holds onto the *Socket* for this connection, the largest known clock value of the remote server, and handles sending *Messages* along the input and output streams associated with this *Socket*. A *ServerConnection* is tightly coupled with a *ConnectionHandlerRunner*, which is in charge of listening for *Messages* written to the *ServerConnection* streams, sending them to the *MessageHandler* for this *Session*, and handling timeouts by sending a *HeartBeatMessage* or announcing death after two consecutive failures. When a *ServerConnection* is destroyed, the *ConnectionHandlerRunner* in charge of it realizes the *ServerConnection* has been destroyed when the next timeout occurs and allows its thread to die.

Total Ordering Algorithm Next we describe the sequence involved in sending and receiving *ChatMessages*. The reader can follow along in the sequence diagram in Figure 2. First, the *Session* receives a String from the UI to be sent to members of the *Session*. The *Session* forwards this String to the *MessageHandler*, which encapsulates it in a *ChatMessage* and asks *Session* to broadcast it. The *Session* informs the *MessageHandler* who will receive the *Message* and the *MessageHandler*, which will be remembered. The *ChatMessage* eventually arrives at the second *ConnectionHandlerRunner*, when it's sent to the second *MessageHandler* to be handled. It constructs a *ProposedDeliveryTimeMessage* response and forwards it to the second *Session* to be sent back to the first *ConnectionHandlerRunner*. When this *Message* is received, it's sent to the first *MessageHandler*. When the first *MessageHandler* receives one such *Message* from all receivers of its original *ChatMessage*, it sends out a *FinalDeliveryTimeMessage* to everyone in the *Session* including itself. When the *FinalDeliveryTimeMessage* is received, the original *ChatMessage* is marked as deliverable. Finally, deliverable messages with the lowest timestamps in our queue of *ChatMessages* are delivered to the UI by the *Session*.

Message and MessageHandler are the two class hierarchies intended to be subtyped by users of our framework. A description of these components and how to extend them is located in Section 4.

3.2 Preliminary Prototypes

3.2.1 Skeenssonne

For our first prototype, we extracted the lobby feature of Javassonne. We attempted to distill the connection-handling capabilities so we could focus on integration with Android and implementation of distributed algorithms. However, we were met with some unusual challenges. First, Javassonne had not been tested on Linux (which is also the basis for Android). When we attempted to run the distilled

[1] `http://code.google.com/p/javassonne/`

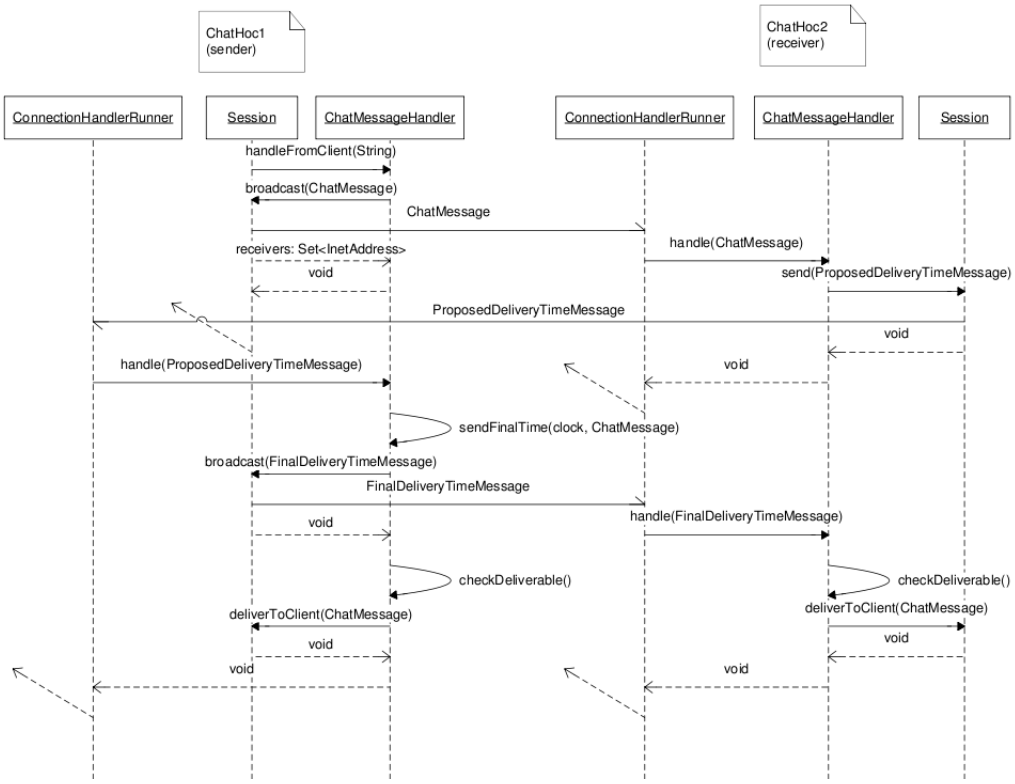


Figure 2: This sequence diagram displays the order of events involved in sending and receiving a *ChatMessage*

Skeenssonne, we encountered the Linux *InetAddress* bug (see Section 6.1. Even after the fix, the RMI service still advertised on the loopback address. We attempted to find and fix the bug in the Spring source code (Spring was the RMI framework used by Javassonne[2]), but decided it would be simpler to throw away RMI and start over with a simpler prototype, using only the LAN advertisement from the Javassonne lobby encapsulated in the JmDNS library.

3.2.2 JmdnsNet

To verify the functionality of JmDNS [13], we implemented a simple Java prototype called *JmdnsNet* that operates from a console window. JmDNS enables service advertisement, discovery, and resolution on a LAN. JmdnsNet simply performs these actions and alerts the user when they occur. For example, when a user starts JmdnsNet (assuming they are connected to a LAN), it starts a JmDNS service and advertises it on the LAN. When other matching services (JmDNS services advertising on the LAN with the same identifier) are discoverd, the user is alerted. JmdnsNet then attempts to resolve a connection to any discovered matching service and announces when a JmDNS connection is made or broken.

3.2.3 SkeenConsole

Our final prototype was a Java console application that could correctly handle a connection to one other instance of the application. This prototype contained the complete model for our final product, including our total ordering algorithm implemented in *ChatMessageHandler*. The components have been improved since the protoype, and a description of the improved architecture can be found in Section 3.1. We used this prototype to assess the feasibility of the idea before implementing it for Android.

[2]http://www.springsource.org/

3.3 ChatHoc

To evaluate the portability and extensibility of our software platform we created a distributed chat application on the Android platform called *ChatHoc* requiring a total ordering of messages between users. Therefore, our software framework fit this type of application very naturally. Below are the notable features of the chat application.

- A ChatHoc user may create any number of chat *Sessions* (See Figure 3(b)), each which may be shared between any number of other users (a single default *Session* is created when the application starts). The SkeenZone middleware guarantees a total order of chat messages between all users' devices within each *Session*. When a *Session* is created, it is automatically assigned a unique identifier by the creator's device.
- To begin chatting, any user in a *Session* may invite other users to that session by their IP address (See Figure 3(d)). A chat invitation message includes the session identifier and the IP addresses of all other users in the *Session*.
- When a user receives a chat invitation, they may accept or reject it. If the invitation is accepted, the invi-

tee's device attempts to establish a connection to each of the IP addresses included in the invitation message.

- Users in a chat session may share chat messages, which are exclusive to that *Session*. While in a *Session*, a user may also view a list of all other users and their IP addresses currently in that *Session* (See Figure 3(c)).

- A user may wish to leave a *Session*, at which point their device ends all connections associated with that *Session* and "forgets" the *Session* reference. Similarly, users on the other end of this terminated connection likewise remove their knowledge of the leaving user from that *Session*.

- ChatHoc also enables a user to switch between currently active chat *Sessions* and at any time view a list of currently available local users and their IP addresses (dynamically discovered by JmDNS).

ChatHoc is a lightweight Android application (about 600 Kb) that consists of a single Android activity [2] (about 450 lines) and some standard Android user interface components (icons, layout templates, etc.). The currently selected chat session's chat history is always displayed in the background. A user interacts with and navigates through the application via menu buttons. The application's activity file contains the definitions of behavior for each of the menu buttons (e.g., what dialog window to display and what SkeenZone functionality to trigger) and an Android `Handler` [2] for asynchronous communication with the SkeenZone middleware.

ChatHoc plugs into the SkeenZone middleware via the SkeenZone *Controller*. The *Controller* is the single point of entry into the middleware. User interface actions performed in ChatHoc are communicated either directly to SkeenZone via the *Controller* for synchronous actions (e.g., starting a chat session, viewing the list of users in a chat session, and viewing a list of locally available users) or by using an Android `Handler` to receive asynchronous feedback from the *Controller* (e.g., inviting another user to a chat session and waiting for the acknowledgement that they accepted and joined the session).

4. EXTENSIONS

This section is intended to describe in detail the classes available in SkeenZone and describe how to extend the functionality to other applications. The two class hierarchies of interest are *Message* and *MessageHandler*. The abstract *Message* class declares implementation of *Serializable* (to send a *Message* across object streams) and *Comparable* (so messages can be sorted in a *PriorityQueue* or other similar data structure). It stores the clock value and address of the sender, provides default implementations for `equals()`, `hashCode()`, and `compareTo()`, and declares a header for the method to construct automatic replies to a given message type. The last is the most interesting feature; for example, a *HeartBeatMessage* constructs an *AckMessage* as the default reply. This reduces the work a *MessageHandler* implementation has to declare.

The base *MessageHandler* defines some default behavior for handling messages and declares two abstract methods to be implemented by subclasses. The first, `announceDead()`, performs whatever action is necessary when a connection is dropped. In our chat implementation, *ChatMessageHandler*, this methods deletes from our pending message queue all undeliverable messages from the dead server as well as stops the handler from waiting for any messages from the dead server. The second, `handleFromClient()`, acts on input from the user. In *ChatMessageHandler*'s case, we encapsulate the *String* from the user in a *ChatMessage* and broadcast it to everyone in the *Session* and begin waiting for a *ProposedDeliveryTimeMessage* from everyone to whom it was sent.

In addition to extending the above classes for your new implementation, you will have to write a new front-end view and a *Controller* to describe the interraction between the view and the model. We will provide descriptions of two extensions we think follow naturally for Android.

(1) We describe a distributed paint program. Following the example of *ChatMessage*, we define a *PaintMessage* that encapsulates a brush and location instead of a string. A *PaintMessageHandler* is created following the example of *ChatMessageHandler*. Finally, one need only define a user interface and connect it to the *Controller*, perhaps with minor modifications to *Controller*.

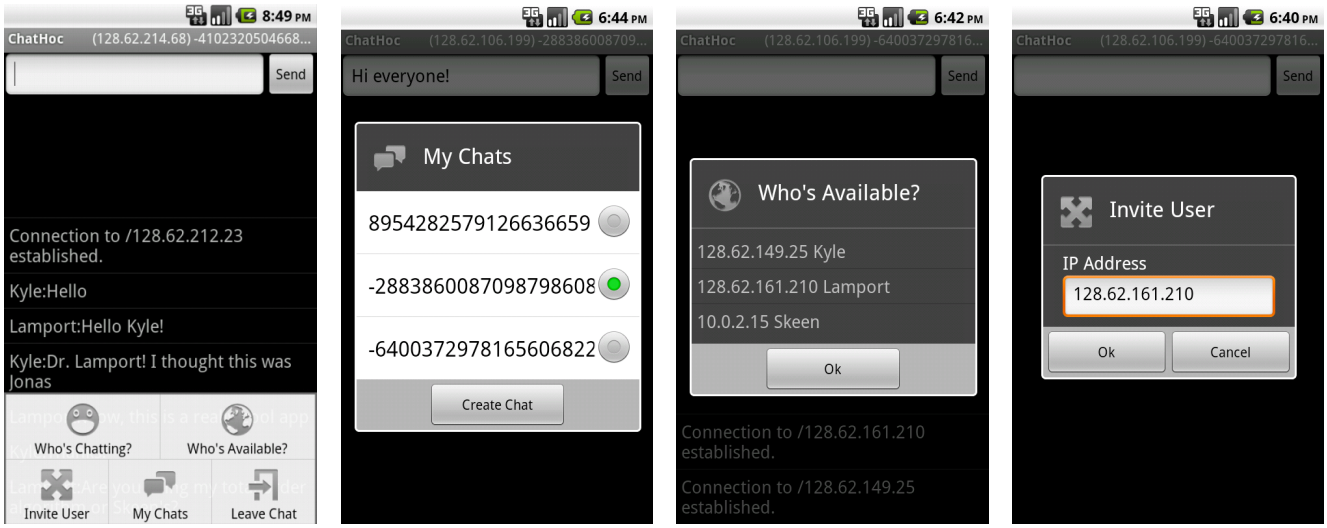
(2) A turn-based distributed game requires the identification of one person as the one whose turn it is, or in other words, elect the "leader." The easiest way to do this is for the inviter to be the leader. The leader needs to determine a turn order and broadcast this to all players. When a new player is added, the leader adds this player to the turn order and broadcasts the new list. Therefore, we need a *TurnOrderMessage*. When the leader's turn is up, it passes the leader token to the next in the turn order with a *TokenMessage*. If the leader dies without passing on the token, the other nodes will realize it when he fails to respond to their *HeartBeatMessages*. The next in the turn order will assume leadership and everyone will agree on him as the new leader. In addition to a *MessageHandler* for these new message types, user interface elements need to be written.

5. TESTING AND VERIFICATION

We incrementally verified the operation of the SkeenZone middleware's operation and the ChatHoc Android application independently.

5.1 SkeenZone Testing

To test the correctness of our implemntation of Skeen's algorithm within the SkeenZone middleware we used the *SkeenConsole* prototype described in Section 3.2.3. We first visually verified the total ordering of manually sent messages between two user consoles on separate machines. Obviously, this did not allow us to capture boundary cases and was more intended for feasibility purposes. Next, we verified the total ordering of messages between four machines. Because SkeenZone relies on the fact that TCP sockets listen for incomming connections on a globally known common port we could only instantiate one *SkeenConsole* per IP address (i.e., per machine). We created an automatic message generator program to produce random traffic between running instances of the *SkeenConsole* prototype. After connect-



(a) Session user interface (b) Available chats (c) Available chatters (d) Inviting a user

Figure 3: The ChatHoc user interface

ing four machines using SkeenZone, we generated 100 messages at each machine and visually verified the total ordering of messages between the four *SkeenConsole* instances. Given these correct results between four machines, we assume that our implementation of Skeen's algorithm extends to any number of connected machines.

5.2 ChatHoc Testing

Our incremental test procedure for the ChatHoc Android application followed a similar spirit. First, we verified that two users could successfully create and leave sessions, invite each other to separate sessions, send messages, and see one another within sessions and on the network. For this step, we installed ChatHoc on an emulated device and on a physical mobile device (a Motorola Droid), connecting them both to the university's secure Wi-Fi internet. The Android device emulator's network interface is isolated from that of the development machine's. We used a port forwarding scheme to expose the emulator's interface to the development machine's public network interface and enable it to connect to outside devices. However, our port forwarding scheme masks the IP identity of an emulator prohibiting an emulated device from successfully recognizing and receiving incoming chat invitation requests. Therefore, an emulated device can only play the role of an "initiator", inviting other users to chat sessions. This limitation is discussed in more detail in Section 7.

Extending our testing of ChatHoc to multiple devices presented a challenge; for reasons discussed in Section 7 we could only use one emulated device in a network of connected ChatHoc user devices. Therefore we had to perform our multiple device test using physical mobile devices. We verified that ChatHoc worked correctly with three physical mobile devices (two HTC Incredibles and a Motorola Droid) and one emulated device connected over the university's secure Wi-Fi internet. Given our verification of the application's performance with four devices, we assume that ChatHoc will work correctly for any number of connected devices.

6. CHALLENGES

Following are descriptions of the challenges we overcame.

6.1 Local Host

In Java, `InetAddress.getLocalHost()` is intended to return the externally visible address of the local host. This call in Linux returns `127.0.0.1`, the loopback address of localhost. Because Android is built on a version of Linux, it suffers from this same problem. Because we need the address so those we are in contact with can reply, we needed a workaround for this problem. The solution involves scanning all connected network interfaces and reading their addresses until we find one that is not a loobback address.

6.2 Session Naming

To set up sessions among multiple clients, we needed a way for an invite to add the new user to everyone's session instead of just locally on the inviter's session. The best way for this to work was to assign each *Session* a unique *SessionIdentifier*. Initially, we hoped to allow users to assign their own *SessionIdentifiers*, however, there was no way to globally enforce uniqueness. Therefore, we generate a random `long` when a new *Session* is created and assign it that `long` as a *SessionIdentifier*. This reduces the chance of collision to a negligible probability, but does hinder user interaction by providing an human-unintelligible identifier for each *Session*. Using these unique identifiers, the *SessionIdentifier* is sent as part of the *HandshakeMessage* when a new connection is made. The receiver adds the sender to this *Session* or creates the *Session* if it does not yet exist locally.

6.3 UI Thread

The view of chat messages a client has received is backed by an ArrayAdapter datasource stored in the Session. When we deliver a new message, we append it to the ArrayAdapter and the corresponding UI element is automatically updated. However, Android does not allow modification of UI elements in any thread other than the UI thread. Therefore,

when the Controller needs to deliver a message to the view, it has to use a Bundle to encapsulate the chat message and the SessionIdentifier of the receiving Session and send them to the Android UI Thread. The UI defines a Handler for these UI messages that delivers the chat message into the correct Session.

7. ANDROID PLATFORM EXPERIENCES

During the course of this project we ran into two limitations of the Android platform that severely restricted the functionality of our implementation or the extent of our testing. First, Android does not permit devices to discover or connect to ad hoc networks without rooting the device. This restriction limited our ability to implement a direct P2P Android application. Second, the Android emulator runs behind a virtual router service that hides it from the the host machine's network interfaces. This detail forced us to use port forwarding to join the emulator's network interface with that of the host machine, which masked the emulator's IP identity and prevented "outside" devices from correctly identifying an emulated Android device.

7.1 Ad Hoc Network Restrictions

Android does not, by default, enable a device to discover, create, or join ad hoc networks.[3] Even if available, ad hoc networks are not even displayed to a user under the list of available networks. A handful of known hacks,[4] patches,[5] and applications[6] exist to circumvent this restriction. However, all known methods of enabling the use of ad hoc networks require the user obtain or already have root permission on their device. This is rarely a simple process (e.g., one that an average user would feel comfortable performing) and often not desirable by most device manufacturers.

Our goal with this project was to create a framework that would be fully functional on a standard Android device without root permissions enabled. Therefore, we were forced to rule out an ad hoc P2P implementation and instead use provided Wi-Fi networks. This is not a limitation of our software platform, but a limitation of our implementation within the current Android OS. In theory, our SkeenZone platform would function exactly the same using P2P connections over ad hoc networks without any changes.

7.2 Emulator Networking

The Android SDK includes an emulator that enables a device to be simulated and debugged locally on a development machine (e.g. a PC). Using the Android emulator is advantageous because (1) it's much faster to deploy and debug on a local machine versus a tethered USB mobile device and (2) the Android SDK includes all released versions of the Android OS, enabling an application to be tested on a variety of target platforms.

However, the emulator's networking capabilities are severely limiting. Android's emulator runs behind a virtual router service that isolates it from the development machine's network interfaces and settings [2]. To allow an emulator to explicitly communicate (e.g., with manual TCP connections) via a development machine's network with emulators either on the same machine or on other machines, a port forwarding scheme must be used to bridge the two network interfaces. We accomplished this using a two-hop port forwarding method. Using a single default port,

(1) we forwarded network communication from the emulator to the development machine's *localhost* using the Android Debug Bridge (ADB) `forward` command then

(2) we forwarded from the development machine's *localhost* to its public network interface using *rinetd*, a TCP redirection software tool.

This scheme is the only working method we were able to identify that enables access an emulator's network interface from an outside connection.

SkeenZone's TCP connections use a default port for listening and rely on the fact that each device will be identifiable by its IP address. These assumptions enable a connection to be made to a device if its IP address is known. Unfortunately, when forwarding through a development machine's *localhost*, the public interface becomes invisible to the emulator. Therefore, an emulated device has no notion of it's IP identity as visible from the outside world (e.g., the internet or LAN). Because of this, other devices may not explicitly connect to an emulated device using SkeenZone (the emulated device will not recognize *inbound* chat invitations addressed to its development machine as meant for itself). However, SkeenZone includes a setting that allows an emulated device's IP address to be hard-coded enabling it to make *outbound* connections with other physical devices.

Ideally, we would have liked to test the ChatHoc application with numerous devices. However, because emulated devices running the ChatHoc application will not recognize chat invitations sent to their development machine's IP address, our testing was limited only to P2P networks of physical devices using a maximum of one emulated device that could only act as an "initiator". We conclude that a more direct means of briding an emulator and development machine's network interfaces is much needed to encourage networking-focused Android application development.

8. CONCLUSIONS AND FUTURE WORK

Mobile computing devices and sensors are have become commonplace in our daily lives. The increasing ubiquity of these devices has opened a new development space for practical applications geared towards decentralized collaboration and coordination. Android, a major mobile development platform, provides rich APIs for interacting with and harnessing devices' hardware components. Currently, however, it does not provide developers with support for distributed computing interaction and constructs.

In an effort to alleviate this void, in this paper we first introduced SkeenZone, a lightweight and extensible Java middleware created to enable the development of distributed mo-

[3] `http://code.google.com/p/android/issues/detail?id=82`
[4] `http://hydtechblog.com/2009/09/14/how-to-connect-to-ad-hoc-networks-using-tmobile-g1-android/`
[5] `http://www.xda-developers.com/android/android-ad-hoc-wireless-network-support/`
[6] `http://code.google.com/p/adhoc-on-android/`

bile applications. Second, we presented ChatHoc, a simple distributed Android chat application as a means of evaluating and demonstrating our middleware's ease of use. SkeenZone uses model-view-controller to separate the display of chat messages from their storage and handling. The framework utilizes a combination of strategy and chain-of- responsibility to handle incoming messages and compose replies. These two pieces can be easily extended to handle new types of messages and new ways of displaying the data contained within them.

We encountered and overcame some challenges in our project, including ignoring loopback addresses returned by `InetAddress.getLocalHost()` by reading local addresses from the network interfaces themselves, naming sessions uniquely using randomly generated ids, and the modifying Android user interface elements from worker threads via asynchronous messages.

During our implementation of the SkeenZone middleware in our ChatHoc Android chat application, we encountered two limitations of the Android platform that impose heavy restrictions on developers' ability to create and test direct P2P applications using ad hoc networks. First and foremost, Android does not currently permit ad hoc network discovery and use by default. This means that P2P networking must be performed over regular Wi-Fi networks instead of by establishing direct Wi-Fi connections to other devices. Second, the Android emulator is extremely useful for testing applications locally on a development machine using a variety of versions of the Android OS. However, the emulator's network interface is isolated from that of the development machine, hindering its utility in testing networking-oriented applications. If Android wishes to encourage developers to create true P2P networking applications, both of these issues will need to be addressed in the near future.

9. REFERENCES

[1] Distributed systems course project: Let's chat. September 2009.

[2] Android. Android developer resources, April 2011. http://developer.android.com/.

[3] Android. Android website, April 2011. http://www.android.com/.

[4] J. Filbert. Developing a multi-purpose chat application for mobile distributed systems on android platform. Technical report, Helsinki Metropolia University of Applied Sciences, March 2010.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Paterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[6] V. Garg. *Elements of Distributed Computing*. Wiley, 2002.

[7] Sun Microsystems Inc. Jxse: The java implementation of the jxta protocols, March 2010. http://jxse.kenai.com/.

[8] Telecom Italia Lab. Java agent development framework (jade), July 2010. http://jade.tilab.com/.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] AndroidChat Project. Androidchat, May 2010. http://code.google.com/p/androidchat.

[11] PeerDroid Project. peerdroid: Jxta peers running on android platform, April 2011. http://code.google.com/p/peerdroid/.

[12] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3):219–228, 1983.

[13] A. vanHoff. Jmdns, December 2002. http://jmdns.sourceforge.net/.