The Junction Protocol for Ad Hoc Peer-to-Peer Mobile Applications

Ben Dodson Aemon Cannon Te-Yuan Huang Monica S. Lam

Computer Science Department
Stanford University
Stanford, CA 94305
{bjdodson, aemon, huangty, lam}@stanford.edu

ABSTRACT

Junction is an application-level communication protocol and library designed for writing mobile applications for ad hoc groups without centralized application servers. We propose that applications be built using a generic *switchboard* service for communication. Each dynamic instance can designate a unique switchboard, hosted by a peer or by a third-party, for the session. Our convention of naming a session by a Junction URI, which encodes the location of the switchboard, enables a simple click-and-run user experience.

The Junction abstraction allows developers a choice of switchboard implementations: XMPP for scalability, IRC for free availability, Pocket Switchboard for mobility, and OpenFlow, a software-defined network, for performance. Invitations to sessions can be carried over NFC, Bluetooth, or QR codes. Junction is available as open source for Android, iPhone and Javascript platforms. Over ten applications in multimedia sharing, games, communication, education, and for enhancing security for online transactions have been developed in Junction.

INTRODUCTION

Smart phones enable a large-class of ad hoc multi-party mobile and social applications from games, communication, and collaboration. Yet today, there are relatively few multi-party applications available. This is reminiscent of the early days of the internet where homes started to get connected but there was relatively little content available online. The World Wide Web took off overnight with the introduction of the HTTP protocol in 1991. This paper asks if we can define an analogous standard protocol that helps make multi-party application mainstream?

Lampson lamented how computer science researchers did not come up with the web in his SOSP keynote address in 1999. This could be attributed to the lack of academic focus on *adoptability*, which is the strength of HTTP. For ad hoc mobile applications, while many of the necessary ingredients, such as discovery [20, 14] and naming [4], have been explored in prior research, numerous obstacles remain in the deployment, creation, and the use of such applications. This paper attempts to identify and eliminate these obstacles to improve adoptability.

As billions of smart phones come online, the demand for multi-party interactions will rival the number of phone calls we see today. The relatively few multi-party mobile applications available today are typically implemented with an *application server* in the cloud mediating all interactions. We postulate that having all the application-specific code run locally on endpoints can greatly reduce the barrier-to-entry and support large numbers of interactions. ISVs (independent software vendors) do not need to (1) write application server code nor (2) deploy scalable servers. Eliminating centralized application servers that can monitor all transactions has an added benefit of improved user privacy.

To facilitate communication between mobile devices, which are often not mutually addressable, each instance of a multi-party application uses a thin message routing service, known as a *switchboard*. This communication service is application-agnostic and can be provided by existing generic messaging solutions.

User Experience Illustration

Let us use the Texas Hold'em poker game to illustrate the behavior enabled by Junction. Each player's phone holds a private hand and the TV or a tablet provides a communal screen for the community cards (Figure 1). Using Junction, a user can simply start the game on his phone and tap his phone with a friend's to have him join. The friend receives a notification, asking if he wishes to join the game. The friend confirms, the phone automatically downloads the game if it is not pre-installed, the game launches and joins the session. Either player can then tap their phone to the TV's NFC reader to bring up the display. The TV displays the game progress on the large screen requiring no further human intervention.

Junction applications are distinctive in that there is no server running any application-specific code in the cloud. All Junction applications rely on a generic switchboard (or a chatroom) for communication. In this instance, the switchboard is hosted on the phone; the session is identified by its Junction URI, which in-

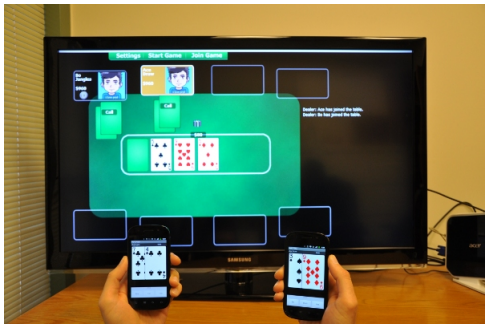


Figure 1: A game of weHold'em.

cludes the switchboard's ip address; the Junction URI is shared with other players and the communal display via NFC. NFC protocols can be used with non-NFC devices by using connection handovers via bluetooth [6]. An NFC sticker, holding the bluetooth address of the TV, can be placed on the remote. A touch of the remote launches the game on the TV.

This game can also involve remote participants who see their cards on their phones and the table on their own TV or computer. In that case, a public IRC chat room may be utilized as the switchboard for the session; the Junction URI contains the IRC chat session address; the URI can be transmitted in an SMS message or email. All the application codes remain the same and run on the end-point devices.

Ad hoc Peer-to-Peer Computing

The scope of this work is to support interactions among friends in the same vicinity specifically, but can also include remote participation where relevant. More specifically, we want to support applications with these characteristics:

- Mobile.
- Multi-party: Two or more participants.
- Ad hoc: No prior arrangements necessary, participants may come and go as they wish.
- Native applications: To take advantage of sensors on resource-constrained devices.
- Cross-platform: Participants may leverage different devices, including servers.
- Device-spanning: A single user may use different devices.

Contributions

This paper makes the following contributions:

Proposal of the Junction protocol for decentralized, ad hoc multi-party applications. Anyone can participate in a multi-party session provided he knows the session's unique Junction URI, which identifies not where the code is run, but where the switchboard is located. The Junction framework, whose architecture is illustrated in Figure 2, has the following advantages:

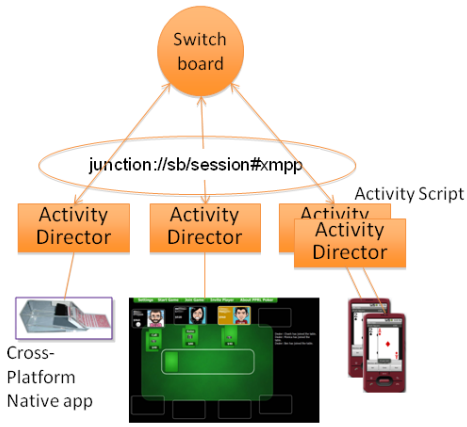


Figure 2: Junction application architecture

- No centralized application server. This reduces the burden of ISVs to create and deploy scalable server codes and eliminates the invasion of privacy due to central monitoring.
- Ease of use. An *activity director* which can launch any application with a single click.
- Ease of programming made possible by three important abstractions. The *invitation abstraction* removes the complexity of coding different methods including NFC, QR codes, and Bluetooth. The *switchboard abstraction* enables applications to enjoy a large variety of messaging implementations, each tailored for different usages. They include XMPP–a popular general-purpose multi-user chat protocol, IRC–a chat protocol freely available in the cloud, a lightweight mobile switchboard implemented directly on top of TCP, Bluetooth for close range communication, as well as OpenFlow, a software-defined network infrastructure to provide high performance. Finally, the *Props* abstraction helps programmers maintain shared state across devices.

Open-source implementation. The Junction framework is fully implemented for Android, web, and iOS (iPhone/iPad) platforms and made publicly available.

Extensive application experience. We have developed over ten applications for multimedia sharing, games, communication, and for enhancing security for online transactions. In particular, a research group in education used Junction to study social learning in the third world. We showed that many of the social applications run well across a variety of switchboards, so the choice can be based on the locality of participants, scalability, deployment ease, and privacy. We also showed that Junction can be used to bridge the semantic gap between applications and the network, so real-time data applications can be written at a high-level while leveraging low-level network optimizations.

THE JUNCTION PROTOCOL

In the following, we describe the Junction URI naming convention, the activity script which defines the application codes for the activity, and finally the session management and communication protocol.

Junction URIs

A Junction URI specifies how to access a switchboard of an activity. It has the following form:

```
junction://[host]/
  [session]#[transport]?role=[role]
```

- host: Address of the device hosting switchboard. It may be a public IP address, a local IP address, or Bluetooth mac address.
- session: The unique session ID for the activity's switchboard.
- transport: The transport used in that switchboard. Examples are XMPP, Bluetooth etc.
- role: As explained below, an activity can have multiple roles. For example, the weTube application for remote-controlled YouTube videos has two roles: the "player" and the "display". A participant is invited to take on a particular role.

A URI for a weTube session hosted on the public XMPP switchboard at sb.openjunction.org looks like:

```
junction://
  sb.openjunction.org/un1q#xmpp?role=player
```

Activity Scripts

The activity script defines the roles of the activity and the specification of each role. A role can be run on a variety of platforms and implemented in different languages, with download instructions listed in this activity script. For example, a role may be available both natively for the iPhone and Android OS, as well as for the web.

The activity script is a set of key-value pairs, structured as a JSON object, with the following keys:

- ad: A unique identifier for the activity.
- friendlyName: A user-friendly name.
- roles: A mapping of role names to specifications for those roles. Each role specification contains a "platforms" object, mapping the name of a platform to details about the code available on that platform, including a download location. The fields of a platform specification are defined uniquely for each platform. For example, the "package" field allows an application to be found and launched on an Android device, whereas a "protocol" is required for launching an application on iOS.

Session Management

We now describe the Junction protocol for creating multi-party activity sessions, as depicted in Figure 3. In this illustration, the "session initiator" (SI) runs a multi-party activity on some switchboard. She invites a "participant" (P) to join. The participant has an activity director preinstalled and may or may not have the application in question.

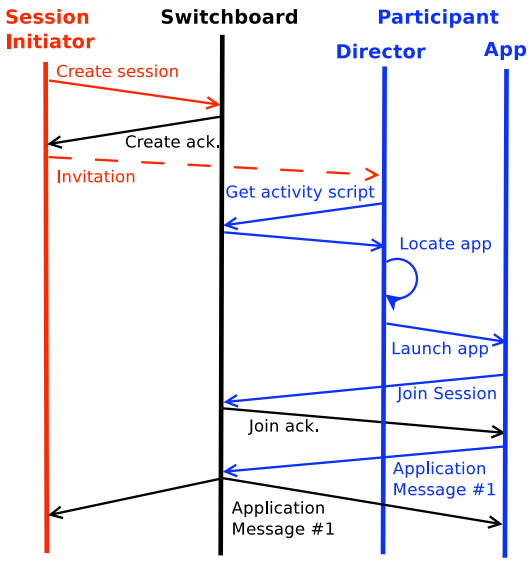


Figure 3: Session Management Protocol.

1. We begin with a user who wishes to start an activity. The SI navigates to an application on her cell phone and launches it.
2. Using the Junction library, the SI generates a unique session identifier for the activity session. The SI creates the activity session by sending a "create" request to the switchboard with the identifier, along with the activity script.
3. The session has a unique URI that represents it. The SI shares this URI with a participant P using some out-of-band method.
4. P accepts the invitation to join by opening it with his activity director.
5. The director contacts the switchboard to determine the activity details, contained in the activity script.
6. Knowing the user's platform, the director locates the appropriate code needed to join the activity. If it is available locally, it is launched; otherwise, the script contains details of where it can be downloaded.
7. The director launches P's local application to join the session. It passes the invitation URI as an argument.
8. P's application joins the session, and the activity commences.

Invitations

A Junction URI can be shared in any way a web URL can be shared today, including email, instant message, and embedding them on web pages. However for many ad hoc face-to-face applications such as sharing business cards using mobile phones, these methods are awkward. For example, Bump allows two phones to find each other by shaking them together [3]; it relies on a central server to match accelerometer readings sent from the devices.

Transport	Latency	Max. Distance	Users per Session
Bluetooth Pocket Switchboard	5-20ms	Room-scale (10m)	Spec: 7 [25] In practice: 3 or 4
TCP/IP Pocket Switchboard	∼10ms	Room/Building-scale (10-100m)	∼100 with switchboard on phone
IRC	∼300ms in WAN with server throttling	Internet-scale	1000s on public servers
XMPP	∼200ms in WAN	Internet-scale	1000s per server supports federation
Software-Defined Network (OpenFlow)	∼10ms in LAN ∼200ms in WAN	Campus-scale	100,000 (depends on switch's flow table size)

Figure 4: A comparison of various switchboard implementations.

To reduce surveillance opportunities and improve scalability, it is desirable that invitations be transmitted locally whenever possible. As smart phones all come with cameras, QR codes can be used to share URI sessions between devices like TV or web pages with mobile phones. They can also be printed out and displayed, say, at an event. Invitations can also be broadcast using a Bluetooth radio beacon [4]; this method is suitable for inviting many participants in the vicinity at once, perhaps in a meeting or classroom environment.

As NFC becomes available on more smartphones, it will become an ideal method for sharing invitations. An invitation can be made available over NFC without requiring blocking UI or extra button presses from the user. Simply touch two devices to share an invitation.

To accept an invitation, we need to pass the Junction URI representing the session to the director. This can be easily achieved on systems that support protocol handlers, such as Android and iOS, by having the director register itself as a process that handles Junction URIs.

What if a device does not already have a director? To bootstrap, one approach is to embed the Junction URI in a web URL that points to a server hosting a director web application. Clicking such a URL would bring up a web page that recommends downloading the director code. The downloaded director would register itself as a listener of the particular web URL, so subsequent clicks of the same URL would be intercepted by the director, which would launch the desired application.

MULTI-PARTY COMMUNICATION

Besides session management, Junction also defines a high-level chatroom-like communication protocol. The protocol includes sending messages to a particular actor (by id) or to a set of actors claiming a role (say, a poker "dealer" or "player"). The high-level communication primitives provide a valuable abstraction, saving the developers' time and providing a choice of switchboard implementations. It is also possible to develop using one protocol, which may be more supportive of debugging, and deploy on another. A high-level comparison of the different switchboards is shown in Figure 4.

XMPP: Federated Multi-User Chat

It is attractive to implement Junction switchboards on top of XMPP because of its widespread deployment, extensibility, and support for federation. XMPP has support for native socket connectivity as well as HTTP connectivity using BOSH [18], thus supporting both phones and web browsers. By running entirely within a Multi-User Chat room [22], Junction can be run on existing infrastructures. If the XMPP server is publicly named, then players can participate remotely; encryption can be used to keep the communication confidential. The XMPP server can be embedded in access points in an institution or be available in home servers like game consoles or multi-media centers for reduced latency of local interactions and better scalability and privacy.

IRC: Leveraging Public Services

Freely available IRC services can also be used as implementations of Junction switchboards. Users can connect to many IRC servers without an existing username and password. Public servers typically add software delays to messages to avoid flooding and general network misuse. Thus, IRC is useful only for applications that have relatively little traffic and are tolerant of delays.

Pocket Switchboard: No Servers

It is sometimes desirable to run the switchboard on the phone itself, while leveraging the phones ability to communicate via Bluetooth or WiFi hot spots hosted on the phone. We found this especially true for rural areas– standing up a local XMPP server was a significant obstacle in using Junction for social learning in Africa.

We have developed a lightweight Pocket Switchboard for the phone that can support a small number of users. Using Bluetooth radio beaconing for invitations, ad hoc mobile applications can be run on just mobile devices, without the need of any infrastructure. We have created two variants, one that runs over TCP/IP sockets and another over Bluetooth. Bluetooth pairing is required only between the peers with the switchboard service, but not with each other. The number of peers that can connect over Bluetooth is limited to 7 [25]. Our experience shows that using the Motorola Droid as a switchboard, we can connect a maximum of three phones (one acting as the switchboard), while a Nexus One can support four.

OpenFlow: Integration with Networks

Some multi-party applications have high communication bandwidth requirements, examples include voice and video teleconference calls. With the emergence of the new software-defined networks [16, 12], we can implement the Junction switchboard directly in the networking infrastructure itself and communicate the application requirement with the underlying network. For example, the Junction library can transparently substitute message routing through a switchboard with a direct network multicast implementation [29].

Our implementation uses the OpenFlow technology [16]. The multicast session is identified by a multicast IP address—messages sent to that address are broadcast to participants. Once a session is set up in a switchboard, the switchboard asks the OpenFlow controller to set up the multicast session and hands the multicast address to the Junction platform running on the participating end nodes. The switchboard updates the OpenFlow controller with the latest list of IP addresses of the participants. The controller then installs the multicast route between the participants.

SHARED STATE MANAGEMENT

When an application runs on end-point devices, application writers must cope with the difficulty in implementing shared state in a distributed manner. Distributed state consistency is a well-studied problem [24, 1, 19]. We noticed that for some applications (the shared whiteboard or the shared playlist, for example) the entire behavior can be described in terms of operations applied to a replicated state. To simplify this style of programming in Junction we extended the platform with the notion of a Prop, a managed data structure that is replicated and kept consistent between all peers. Our design is informed by the state replication strategy of Croquet [23].

A Prop is a developer-defined data structure with a set of operations to be applied to the data structure. It provides several fundamental properties useful for ad hoc multi-party applications.

1. Participants can join and drop out at different times. A participant can acquire the current state of the data structure upon joining; all this is done without designating a particular participant as the ultimate source.

2. Sequential consistency. Every participant *eventually* sees the same global order for everybody's operations.

3. Predictive operations. For the sake of responsiveness, it is desirable for users to get immediate feedback, even though the state may need to be adjusted later. For example, when a user submits a song, he should see the song on the list right away and on his phone as an acknowledgment.

At runtime, an instance of the Prop is created at each peer. The Prop instances conduct a synchronization protocol using the standard Junction messaging layer. These messages are namespaced so as to be invisible to higher level application code. The base Prop behavior is defined in terms of an abstract state and abstract operations. The application developer extends the class Prop and defines i) the form of the state (this can be arbitrary), ii) the operations that may be applied to the state (again, arbitrary), iii) a routine to create a deep copy of a given state.

In Croquet, all operations are broadcast through a thin, central router with no application logic. Sequential consistency is guaranteed because the global order of operations is enforced by the router and every peer applies them in the same order. Props borrows this same model, where the router is replaced with a Junction switchboard. We also borrow Croquet's state transfer concept. When a new user joins the activity, their instance of the Prop listens to the protocol and determines if it is out of date. If so, it requests an update from the other instances. The state (copied using the routine mentioned above) is streamed from the instance that responds with the most advanced state within a short window of time. This is an optimistic strategy: there is no authoritative state due to the potential for drop out.

We extended the croquet scheme to include support for prediction to improve responsiveness. Predictive operations are applied locally, speculatively, to a *copy* of the last-known consistent state before being sent to the switchboard. If an update from the switchboard invalidates a prediction (the update from the server arrives before the confirmation of a prediction), we rollback to the last-known consistent state and re-apply all operations in the correct order.

Props interfaces with application code via change events. When a new state is streamed from a peer or a new operation is applied, the Junction system will send the application code a "state-change" event. Upon receiving such a notice, the program may need to refresh the screen or update application-level private state to reflect the update. This is a seemingly naive strategy, as application code must be written to account for sudden and unexpected state changes. However, we've found in practice that the states evolve incrementally, and the event-driven design encourages good separation between the model and view portions of a program.

EVALUATION

We have a full implementation of the Junction framework described above and performed an extensive evaluation of the system. We have written over ten useful applications to understand the ease of use and ease of development of the system. The system is mature enough for a research group in education to use Junction to study how social interactions can enhance learning in rural areas. To understand the characteristics of different switchboards, we implemented them and measured them with micro-benchmarks. We created a real-time game to stress the switchboards and a teleconference

application to test the OpenFlow implementation.

Implementation of the Junction Platform

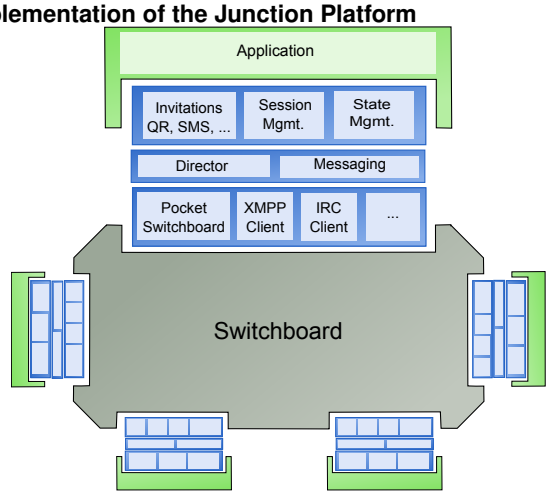


Figure 5: The Junction client library architecture.

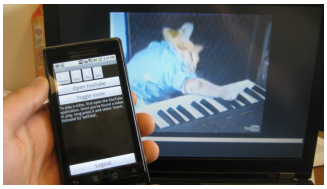
We have developed a prototype of the Junction infrastructure for the desktop, web, Android, and iOS platforms. The client libraries are composed of the same architecture, depicted in Figure 5. Each library has a pluggable set of supported switchboard implementations that provide connectivity to peers. On the mobile clients, we bundle the Pocket Switchboard's server implementation as well as client library, and include the libraries for XMPP and IRC connectivity. The application developer interacts with an abstraction layer on top of these implementations with a common set of functionality. The library exposes the ability to create and join a session, send messages (either directly to peers or to the entire session), and propagate lifecycle events such as "session joined" and "session created" to the application layer. An application may also use Props to help manage shared, distributed state.

We found the Android Platform's system of Intents [9] very helpful for building programs that provide services to other apps. The director application uses Intents as a way to return p2p configuration details to a Junction application running in a separate process. This is in contrast to the iOS platform, where interprocess communication is limited to applications registering handlers for URIs opened by other programs.

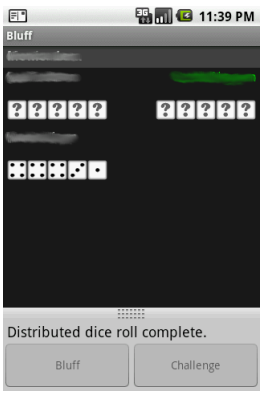
We found connectivity to be a challenge for Javascript. Since Javascript cannot open standard socket connections to remote servers, we need to tunnel socket-based protocols like XMPP over HTTP [18]. Tunneling libraries carry a heavy performance overhead. We plan to use WebSockets for connectivity in the future, which should be comparable in efficiency to a standard socket connection. Where possible, we use CORS [27] to avoid cross-domain request restrictions.

Applications

We have found that Junction is useful for a variety of applications, from collaborative multimedia sharing, to games, to improving security by using the phone as a second factor, and for communications. A selection of 9 applications built using Junction platform is summarized in Figure 6.

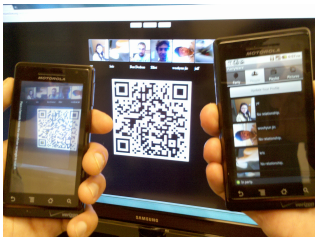


(a) weTube



(b) weBluff

(c) PocketTanks



(d) weMeet

Figure 7: Screenshots of Junction Applications

PocketSchool

In addition to the applications we have developed internally, Junction has also been used by a research group in the Stanford School of Education to create PocketSchool. PocketSchool is an educational game that runs amongst several cell phones (Figure 8.) The activity consists of a facilitator and several students. Each student reads a short story and responds to quiz questions at the end. The first student to complete the quiz successfully wins. Another version of the game requires the students to work together to comprehend the story being told.

PocketSchool has been piloted both at nearby schools and rural communities in the third world, and is discussed further in [13]. By taking advantage of Junction's Pocket Switchboard, PocketSchool can be used in rural areas without networking infrastructure.

Application	Description	Lines of Code	Dev. Time
Multimedia: Quick and fun collaboration			
weTube	Use phones to share YouTube videos on a communal display, with support for voting up favorites (Figure 7(a)).	Display: 450 lines html, css, js Android remote: 1500 lines Java iOS remote: 900 lines Objective-C	2 days
wePix	Share photos either recently taken or in the phone's existing library with a group and display them on a communal display.	Display: 450 lines of html, css, js Android controller: 1400 lines of Java iOS controller: 1500 lines Objective-C	2 days
Games: Interactive Multiplayer Social and Real-Time			
weBluff	A "common hand" version of the Liar's Dice game; each phone shows the player's view of the dices, 5 for each person. 2–8 players. (Figure 7(b))	Player: 1500 lines of Android Java	3 days
weHold'Em	A Texas Hold'Em variant using Android devices to display private cards and a web browser to display communal cards. 2–8 players. (Figure 1)	Player: 800 lines of Android Java Table: 750 lines of html, css, js Dealer: 1700 lines of server Java	30 days
PocketTanks	A mobile real-time tank commander game; users drive around to shoot each other. 2–5 players (Figure 7(c))	Player: 2000 lines of Android Java	2 days
Security: Using Phones as a Second Factor			
Snap2Pass	Visually authenticating into a web page using a mobile phone as the authentication token.	Provider: 1600 lines of server Java Web: 120 lines of html, css, js Phone: 400 lines of Android Java	6 days
Communication: From Introductions to Teleconference			
weMeet	A real-time mobile application for exchanging profiles with a group (Figure 7(d)).	Android: 1600 lines of Java iOS: 1200 lines Objective-C	2 days
weScribble	A cross-platform collaborative whiteboard	Android: 1400 lines of Java Web: 400 lines html, css, js iOS: 700 lines Objective-C	3 days total
weTalk	Real-time voice with optimized network traffic.	Android: 850 lines of Java	7 days

Figure 6: A selection of applications we have built using Junction

Application Development Experience

The applications illustrate how various features can be combined in different ways to achieve the desired effects. Except for weBluff, PocketTanks, and weTalk, all the applications run across different platforms. The weTube, wePix, weHold'Em applications can use a large screen display for communal display of information; Snap2Pass interacts with the web applications on the PC. WeTube, wePix, weMeet, and weScribble run on both the Android and iOS mobile platforms. In addition, WeScribble is available on the iPad and as a web application runnable on any PC, since it is interesting in a non-mobile setting as well.

weMeet, weTunes, wePix, and PocketTanks make use of Props to manage application state; weBluff, weHold'em, PocketSchool, and Snap2Pay/Snap2Pass is implemented directly with simple message passing. Snap2Pass and Snap2Pay require an explicit call to the QR code scanner, while the remaining activities can rely on the activity director for session management. This variety suggests that the features are general and useful for many different applications. Junction is not designed for low-latency, high-bandwidth interactions. To understand its limitation, PocketTanks and weTalk were written to evaluate the latency and bandwidth supported by Junction.

The weHold'em and weBluff applications bring out an important complication in the implementation of distributed multi-party games, where there are no trusted third-parties that executes part of the application logic. How do we ensure that no cheating takes place on the end user devices? Our implementation of the weBluff game uses an adoption of commitment-based cryptographic techniques to implement verifiably fair dice rolls [2]. The details of the algorithm are beyond the scope of this paper. Similarly we can use distributed card shuffling techniques for weHold'em [17].

Figure 6 shows the number of lines of code written for each platform and the amount of time taken. Notably, the weHold'Em poker game was developed alongside the original Junction development, and so it took much longer to complete. Similarly, weTalk was developed in tandem with the associated OpenFlow support. We note that Junction programs are succinct and can be developed quickly, thus illustrating how Junction makes it easy to write peer-to-peer activities.

Application Usability Experience

We conducted a small user study with 3 groups of 4, 5, and 7 college students. Users were invited to use the weTunes and wePix applications in a casual environment and were then surveyed about their experience. Details of the trial are beyond the scope of this paper, due to space constraints. Suffice to say that users in general liked the applications. The response to the weTunes Social Playlist app was particularly positive. Comments such as "love it" and "easy to get a playlist with songs everyone likes" were common. The applications performed reliably, and we did not receive any complaints about latency. Note that for these trials, sessions were hosted at our dedicated XMPP switchboard at open-

Figure 8: The PocketSchool educational game.

junction.org.

Micro-Benchmarking of Switchboards

The Junction abstraction provides applications a choice of switchboards. Our first performance analysis is to characterize the different switchboards with micro-benchmarks to measure the message round-trip time (RTT). The round-trip time includes the time a message propagates through the Junction API on the client machine, across the network, back to the client machine and back through the Junction software layer. The average RTTs and the standard errors in the following figures are taken over 100 messages. The local switchboards used in the following experiment were hosted on a Linux PC with an AMD P920 1.6G Quad-Core processor and 4G memory. Switchboards hosted locally are connected to their participants on an isolated LAN.

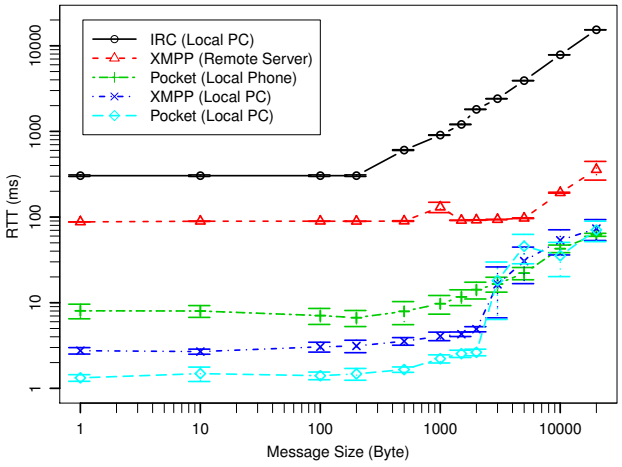


Figure 9: Message size vs. RTT (1 participant)

The first experiment measures the RTT of a single participant sending messages of different sizes (Figure 9). The results show that all the local switchboards, except for IRC, can handle messages up to 1KBytes with a RTT of less than 10 ms for 1 participant. The lightweight Pocket Switchboard is faster than XMPP (OpenFire) on a PC because it is customized for Junction; this difference is especially important for mobile devices. The comparison between running the Pocket

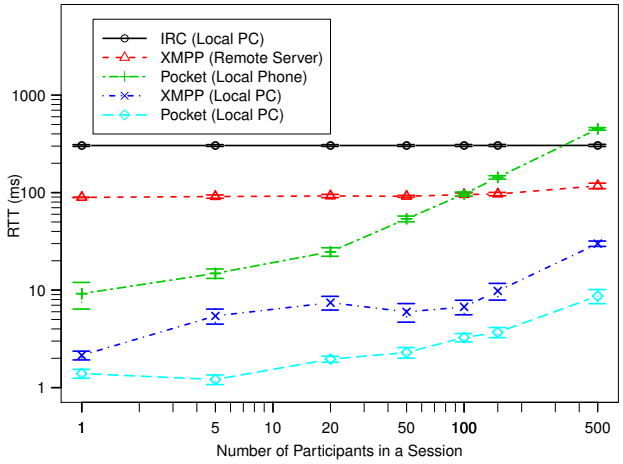


Figure 10: Number of participants vs. RTT (10-byte message size

Switchboard on the phone (a Nexus One) and a PC illustrates the performance difference of the two devices. Going remote to an externally hosted XMPP server (open.junction.org hosted at Amazon S3) results in a RTT close to 100 ms. This illustrates how using a local service, even on a mobile device, can deliver better latency.

IRC (InspireIRCD) has a RTT of about 300 ms even when hosted locally. IRC servers are intended to be used as chat servers with human clients and as such contain flood-protection features designed to prevent users from spamming a channel. These restrictions limit the speed that a message may propagate through the server. Furthermore, the slowdown of messages beyond 512 bytes, as shown in Figure 9, is due to a 512 byte size limit imposed by IRC. Our Junction library breaks long messages into 512-byte ones for IRC. Note that despite the poor comparison with other switchboards, IRC is still suitable for social applications that can tolerate high latencies. For example, we found the weScribble application usable even when connected to a public IRC server hosted at chat.freenode.net.

The second experiment shows the RTT of messages sent by one party to different numbers of participants (Figure 10). We see that both IRC and remote XMPP deliver the same performance for up to 500 participants, since the overhead in these cases is dominated by factors other than server performance. All other implementations slow down as more participants connect, since Junction requires messages to be routed to all the parties. Nonetheless, even the Pocket Switchboard on a mobile device can serve 100 users with the same latency as an XMPP server on a WAN to 100 users. From the figure, we see that a local server on a PC has no problems scaling to 500 users.

Applications with High Bit/Packet rate

We use weTalk to evaluate the ability of Junction to hand over heavy-duty communications, such as

voice/video streaming, to the underlying OpenFlow network for better performance. We show that weTalk outperforms Skype, the state-of-the-art peer-to-peer multimedia application, for conference calls of three or more parties due to the ability to multicast inside the network.

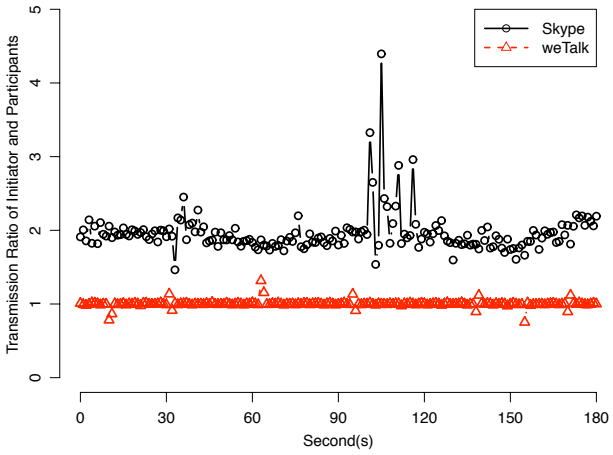


Figure 11: The transmission ratio between the initiator and the participants

For Skype to support conferencing, the call initiator serves as the relay for other parties. The transmission bitrate of the host increases linearly with the number of participants. This is not good news for the initiator, since it will consume a lot of bandwidth and power, which are both scarce resources especially on mobile devices. Figure 11 compares the ratio of transmission bitrate between the call initiator and the other participants. It shows that the call initiator on Junction/OpenFlow transmits no more data than others, while the initiator in Skype needs to transmit twice as much data as others in a three-way conference call.

Junction/OpenFlow delegates the traffic multicasting to the underlying network and no device needs to act as a relay for others. This is particularly important in pure phone-to-phone settings because of the limited battery life and bandwidth available to a phone. Furthermore, by eliminating the relay node, the round-trip delay between a pair of phones decreased by half in our experiment setup. The decrease in delay can improve the user experience, especially for multimedia activities.

Putting it All Together

As shown in Figure 12, most of the social applications have little requirement on bandwidth and latency and can be run on all but the OpenFlow switchboard. For example, the Pocket Switchboard is good for local interactions, XMPP for large-scale deployments, and public IRC service to support remote participation without requiring servers. PocketSchool, designed for third-world education, is best run completely on mobile devices using the Pocket Switchboard. PocketTanks, a mobile real-time game, will not run well on public IRC because

Application	Pocket	XMPP	IRC	OpenFlow
weMeet, wePix, weTube, weBluff, weHold'em, weScribble	✓	✓	✓	
PocketSchool	✓			
PocketTanks	✓	✓		
Snap2Pass, Snap2Pay		✓		
weTalk				✓

Figure 12: Suitability of switchboards.

of the anti-spamming measures built into the protocol. Snap2Pass and Snap2Pay are business applications used in web pages to enhance security with mobile phones, so XMPP is appropriate. Finally, weTalk can be optimized by implementing the switchboard on OpenFlow.

RELATED WORK

There are many existing systems for enabling multi-party communication. UPnP and DLNA [26, 5] are well-established standards for connecting services, typically on a LAN. They focus on the development of fixed networked services, such as printers and media servers and consumers, as compared to our focus on complete applications running across devices.

Systems such as the iRoom project demonstrate the utility of bringing devices together for new forms of collaboration [8]. Want, Lyons et al. extend this idea and explore composable computing in depth [28, 14], creating a framework for developing collaborative systems. Their Composition Framework handles device discovery and messaging across devices using an array of wireless technologies. Devices run services and capabilities that can be composed to accomplish various tasks. In contrast, our primary focus is on helping developers create complete multi-party applications across several devices.

With Obje, Edwards et al. explore how devices can participate in recombinant computing [7]. Here, devices (discovered over a LAN) expose a set of supported services that can interact with other applications on the network. The work focuses on ad-hoc interoperability between devices and services that were not originally designed to work together. Again, our focus is to provide a platform with which cohesive applications can be spread across multiple devices. We assume that all components of the application are being developed in concert, and streamline their development.

Pierce et al. use XMPP to exchange information across multiple devices of a user and provide functionality that spans personal devices [21]. They attach a list of personal devices to centrally managed user accounts. Whenever a personal device joins the network, it can be discovered by other devices that belong to the same user. Junction focuses on ad hoc activities, as compared to managing services across a user's devices.

Pering et al. uses RFID tags to configure the devices for

co-operative tasks [20]. To perform a particular task, a device should scan the coresponding RFID tag, which stores the configuration for the specific task. By scanning in the RFID readings, mobile devices can understand what application they should launch and where they should go meetup with other devices. Similarly, researchers have explored using physical interactions to allow quick device associations. They make use of the sensors on devices and start the pairing process by bumping them [11, 3], shaking them [15], and by touching them together [10]. These interactions lend themselves well to the Junction protocol, which further defines how the devices can run a joint activity and how simplifies how developers can write them. Bump™ [3] also provides an API for cross-device messaging, but requires cloud-based infrastructure to support the task.

CONCLUSION

Today there are relatively few multi-party applications in the appstore of mobile devices, and they predominantly use the server-client architecture. Because peer-to-peer applications are easier to deploy and more scalable, we believe that making them easier to write and use can help propel multi-party applications into the mainstream. Peer-to-peer applications also have better privacy characteristics.

We isolated many of the common burdens of creating such applications, including peer discovery, application discovery, code management, runtime connectivity, and showed that they can be simplified with the Junction protocol and framework. The key idea is that anyone can participate in a multi-party session provided he knows the session's unique Junction URI, which identifies not where the code is run, but where and how the switchboard is located.

Our experimental results show that many compelling applications in multimedia sharing, communication, games and even education can be supported with this model. A small user study suggests that the single-click launch of applications seems suitable for ad hoc social applications. The applications, while relatively easier to write, can be highly optimized for different scenarios by using the variety of switchboards that are included in the framework. We have released our Android, iOS, and web libraries as open source projects, available on openjunction.org.

REFERENCES

1. P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond, 2001.
2. M. Blum. Coin flipping by telephone - a protocol for solving impossible problems. In *COMPCON*, pages 133–137, 1982.
3. Bump technologies. `bumptechnologies.com`.
4. N. Davies, A. Friday, P. Newman, S. Rutlidge, and O. Storz. Using bluetooth device names to support interaction in smart environments. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, MobiSys '09, pages 151–164, New York, NY, USA, 2009. ACM.
5. DLNA. `dlna.org`.
6. B. Dodson. Social applications between phones and a TV using NFC, 2011. `mobisocial.stanford.edu/news/2011/02/social-applications-between-phones-and-a-tv-using-nfc/`.
7. W. K. Edwards, M. W. Newman, J. Z. Sedivy, and T. F. Smith. Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. *ACM Trans. Comput.-Hum. Interact.*, 16(1):1–44, 2009.
8. A. Fox, B. Johanson, P. Hanrahan, and T. Winograd. Integrating information appliances into an interactive workspace, 2000.
9. I. Google, 2010. `developer.android.com/reference/android/content/Intent.html`.
10. R. Hardy and E. Rukzio. Touch & Interact: Touch-based Interaction of Mobile Phones with Displays. In *MobileHCI*, 2008.
11. K. Hinckley. Synchronous Gestures for Multiple Persons and Computers. In *UIST*, 2003.
12. Junos SDK - network software development community - juniper networks. `http://www.juniper.net/us/en/products-services/nos/junos/junos-sdk/`.
13. P. Kim and A. Goyal. "PSILAN" - PocketSchool Interactive Learning Adhoc Network. `ldt.stanford.edu/~educ39109/POMI/PSILAN`.
14. K. Lyons, R. Want, D. Munday, J. He, S. Sud, B. Rosario, and T. Pering. Context-aware composition. In *Proceedings of the 10th workshop on Mobile Computing Systems and Applications*, HotMobile '09, pages 12:1–12:6, New York, NY, USA, 2009. ACM.
15. R. Mayrhofer and H. Gellersen. Shake well before use: Intuitive and secure pairing of mobile devices. *IEEE Transactions on Mobile Computing*, 8:792–806, 2009.
16. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, April 2008.
17. C. A. Neff. Verifiable mixing (shuffling) of elgamal pairs. Technical report, In proceedings of PET '03, LNCS series, 2004.
18. I. Paterson, D. Smith, P. Saint-Andre, and J. Moffitt. XEP-0124: Bidrectional-streams Over Synchronous HTTP (BOSH), 2004. `xmpp.org/extensions/xep-0124.html`.
19. J. D. Pellegrino and C. Dovrolis. Bandwidth requirement and state consistency in three multiplayer game architectures. In *Proceedings of the 2nd workshop on Network and system support for games*, NetGames '03, pages 52–59, New York, NY, USA, 2003. ACM.
20. T. Pering, R. Ballagas, and R. Want. Spontaneous marriages of mobile devices and interactive spaces. *Commun. ACM*, 48(9):53–59, 2005.
21. J. S. Pierce and J. Nichols. An infrastructure for extending applications' user experiences across multiple personal devices. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 101–110, New York, NY, USA, 2008. ACM.
22. P. Saint-Andre. XEP-0045: Multi-User Chat, 2008. `xmpp.org/extensions/xep-0045.html`.
23. D. Smith, A. Kay, A. Raab, and D. Reed. Croquet - a collaboration system architecture. In *Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings. First Conference on*, pages 2 – 9, jan. 2003.
24. C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

25. A. Umar. *Mobile Computing and Wireless Communications*. NGE Solutions, 2004.

26. UPnP. `upnp.org`.

27. A. van Kesteren, 2010. `w3.org/TR/cors`.

28. R. Want, T. Pering, S. Sud, and B. Rosario. Dynamic composable computing. In *Proceedings of the 9th workshop on Mobile computing systems and applications*, HotMobile '08, pages 17–21, New York, NY, USA, 2008. ACM.

29. K.-K. Yap, T.-Y. Huang, B. Dodson, M. S. Lam, and N. McKeown. Towards Software Friendly Networks. In *ACM APSys2010*, August 2010.