Managing Ad Hoc Networks of Smartphones

Tiancheng Zhuang, Paul Baskett, and Yi Shang

***Abstract*—Wireless ad hoc networks of mobile devices enables users to communicate with each other in an infrastrucre-less environment. This paper presents the design and implementation a middleware on mobile Android devices for managing wireless multi-hop ad hoc networks. The middleware is implemented in the user space over Android OS and requires no kernel modification. It uses the Ad hoc On-demand Distance Vector (AODV) protocol for application layer routing and the Reliable User Datagram Protocol (RUDP) for reliable data delivery. It is implemented in two ways, one using traditional layered networking structure and the other based on software-defined networking principles. The implementations are portable across different versions of Android and provide easy-to-use interfaces for Android applications. To evaluate our middleware implementations, a text chat and a file sharing application were implemented based on the middleware and experimental results show good performance, up to 5Mb/s and 4Mb/s throughput in single-hop and two-hop communications, respectively.**

***Index Terms*—Ad hoc wireless networks, networking middleware, software-defined networking, smartphones.**

I. Introduction

Nowadays, modern smartphones are rapidly gaining popularity. The powerful processing and wireless networking capabilities of these smartphones enable users to communicate and share information much easily than ever before. In forming mobile ad hoc networks (MANETs), mobile devices including smartphones can connect to each other wirelessly without existing network infrastructure. This extends the usage of smartphones and also makes information sharing more convenient at lower cost.

Although mobile ad hoc networking has been actively researched for many years, implementations on smartphones are rare. Android and iOS supports single-hop wireless networking, but not multi-hop. Most existing MANET middleware designs require modifying the OS's networking stack in order to achieve multi-hop communication. Kernel modification hinders portability across different hardware and kernel versions [1].

In this paper, we present the design and implementation of a middleware for WiFi-based ad hoc multi-hop networking on Android devices. Compared to Bluetooth, WiFi offers a much higher transmission bandwidth over longer distance. The middleware is implemented in user space to make it more portable on different Android devices. It enables user applications to discover nearby devices and connect to remote applications through a simple wireless networking interface. As part of the middleware, the Ad hoc On-demand Distance Vector (AODV) protocol [2] is implemented for application layer routing and the Reliable User Datagram Protocol (RUDP) [3] for reliable data delivery. Both unicast and broadcast data services are provided. The middleware also supports transmission control for establishing reliable end-to-end connections. It is implemented in two ways, one using traditional layered networking structure and the other based on software-defined networking principles.

The rest of the paper is organized as follows. In Section II, we describe related work. In Section III, specifics of the middleware design using traditional layered networking structure are presented and discussed. In Section IV, implementation details using traditional layered networking structure are presented. In Section V, the implementation based on software-defined networking principles is presented. Section VI shows experimental results. Finally, Section VII concludes the paper.

Manuscript received April 1, 2013; revised June 1, 2013.

Tiancheng Zhuang, Paul Baskett, and Yi Shang are with the Department of Computer Science, University of Missouri, Columbia, MO 65211 USA (e-mail: harrycn1987@gmail.com, pkbkbc@mail.missouri.edu, and shangy@missouri.edu).

II. Related Work

Although many implementations of ad hoc multi-hop networking have been developed on PC operating systems such as Windows and Linux, very few are available for smartphones.

BEDnet [4] is a Bluetooth-based mobile ad hoc networking implementation in Java. It uses a simple scatternet formation algorithm and DSDV routing protocol. BEDnet has a poor data transfer speed due to limitations of J2ME. It can only be used by a single application and does not work on Android and iOS devices.

As the successor of BEDnet, Beddernet [5] is a Bluetooth-based mobile ad hoc networking software developed for Android OS. It is built on top of Bluetooth's RFCOMM protocol and also implements DSDV routing protocol. Compared to BEDnet, Beddernet has better communication throughput and supports both unicast and multicast.

Although WiFi is widely used for communication on mobile devices and offers faster speed and longer distance communication than Bluetooth, development of ad hoc networking via WiFi on smartphones is rare. WiFi (IEEE 802.11) standard specifies two operating modes for wireless communication: infrastructure mode and ad hoc mode (also called infrastructureless networking mode). In ad hoc mode, each device can directly communicate with another device and the management of the ad hoc network is done through collaboration between the nodes in the network. Only single-hop communication is supported, so if devices A and B are not within the transmission range of each other, they cannot talk to each other. Multi-hop communication in ad hoc mode is not supported due to the lack of routing functions.

DOI: 10.7763/IJIET.2013.V3.332

Many methods have been proposed to implement routing protocols to support multi-hop ad hoc networking on PC OS such as Windows and Linux. A common approach is the in-kernel implementation, in which a routing module in the kernel is added to modify IP layer management. The in-kernel module handles route discovery and route maintenance by adding some additional processing to all IP packets that are going through the IP module. Once a route is found, a new route entry is created in the OS's kernel routing table to achieve packet forwarding. Because most of the routing service is done by the routing module in the IP layer, the biggest advantages of this type of implementation is that it minimizes the cost for copying packet between kernel and user space, and usually has the best performance in terms of speed and efficiency.

Split kernel-user implementation is an approach to reduce kernel modifications and make the system more portable. Usually a routing daemon is implemented in the user space and a packet sniffing module is implemented in the kernel space. The approach requires the use of packet sniffing tools such as Netfilter and Snoop to intercept all the incoming and outgoing IP packets in the kernel space [6]. These tools are usually implemented as a kernel module to identify many events that may trigger a routing action. The routing daemon maintains and updates the kernel routing table through IPC calls between kernel space and user space.

When the kernel of an OS cannot be easily modified, like the case of Android, a user-space implementation is an alternative to support ad hoc multi-hop networking. The major drawback of this approach is the reduction of communication efficiency. ORWAR on Android by Davide Anzaldi is an example of this approach on Android OS [7]. It is a standalone Android application and does not provide any API for third party applications to use it. Ad hoc on Android aims to implement an ad hoc networking library that can be included in Android applications [8]. The implementation uses reactive routing and the library cannot be used by multiple applications at the same time. Both of these two systems are UDP based and offer no data flow control, so they may be used for simple message exchange and are not sufficient to support applications such as file sharing and interactive real time gaming.

Similarly, our multi-hop ad hoc networking middleware on Android is implemented in the user space. It utilizes WiFi communication and provides a richer set of functionalities than existing implementations. Its unique features include data flow control, service discovery, and multiple applications support.

III. New Middleware Design Using Traditional Layered Networking Structure

Even though the Android OS runs on top of a Linux kernel, existing implementations of MANET routing protocols on Linux can't be ported to Android OS. This is because Android runs on its own version of a well protected Linux kernel that user access is very limited. Tools that are typically used to intercept IP packet and modify system routing table such as Netfilter, Snoop and Netlink are not available on Android. In addition, any improper modification to Android kernel could easily crash the whole system.

Our middleware is designed to run as a standard Android Service in its own process in user space. The service can be installed and uninstalled as a normal Android application. To avoid modifying existing networking stack of Android, the middleware is built on top of existing transport protocol APIs provided by Android and data packet routing is performed in the application layer. In order to share the networking resources among multiple user applications, inter-process communication needs to be implemented between user applications and the middleware. The implementation is designed to meet the following requirements:

- Enable an Android device to construct and join an ad hoc network.
- Reliable user-space data forwarding and route maintenance in mobile environment.
- Support multiple applications and provide easy-to-use application interfaces.
- Portable to different versions of Android systems and different devices.

As shown in Fig. 1, the middleware consists of three layers: Data Link Layer, Route and Service Discovery Layer, and End-to-End Transport Layer.

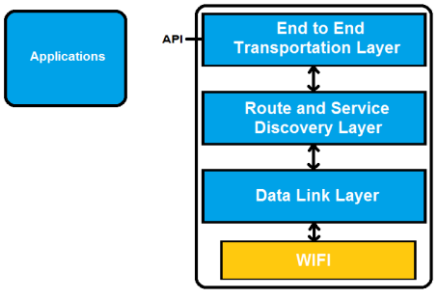


Fig. 1. The proposed middleware architecture.

A. Data Link Layer

The data link layer handles data transmission between directly connected devices and hides low-level implementation from the upper level. In this layer, we use UDP (User Datagram Protocol) in direct communication between devices. All control and data packets are encapsulated into UDP packet to sent and received through JAVA UDP Socket. We choose UDP instead of TCP for two reasons:

- UDP supports both broadcast and unicast. Since each node has no knowledge of its neighbor in the beginning, broadcast is necessary for discovering neighbors and updating routes.
- TCP is connection-oriented and goes through end-to-end handshaking before communication. This process is time consuming and requires multiple threads to maintain the connections. Compared to TCP, UDP is much simpler and faster.

B. Route and Service Discovery Layer

All nodes in a multi-hop ad hoc network play a role of routers, determining the path to forward a data packet toward its final destination. A distributed routing protocol needs to be adopted by every node in the network. Routing protocols for MANET should be able to detect link breakage and maintain the networking when nodes moves or exits. Existing routing protocols can be divided into two main categories:

proactive and on-demand. Proactive routing protocols, such as DSDV [9] and OLSR [10], maintain a routing table by continuously broadcasting topology discovery messages. The advantage is that routes are available at any time and the disadvantage is the high routing overhead. On-demand routing protocols, such as AODV and DSR [11], discover routes on an "on demand" basis. Route discovery messages are only flooded in the network when there is a need to reach an unknown destination. They have less routing overhead, but incurs a delay from the route discovery process.

The route and service discovery layer is responsible for forwarding data packets and discovering routes and services. Each device maintains a routing table for the next hop. To achieve multi-hop data transmission, additional route information is injected into the header of each data packet. Data packets are held in user space until a route is established to avoid the problem of having packet cross the boundary between user space and kernel space before route request is even detected, as stated in [6].

In order to establish routes, AODV routing protocol is used due to its simplicity and good performance [12]. When a source node needs to discover a new route, it generates and floods a Route Request (RREQ) message containing the destination's IP address through the network. Each intermediate node receiving the RREQ packet updates its reverse route to the source if the sequence number is higher than its cached one. When the RREQ reaches the destination node or a node has a fresh route to the destination by comparing the destination sequence numbers, A Route Reply (RREP) message will be generated and sent back to the originator through the reverse route. The originator uses the information contained in RREP to update its routing table and start data transmission. Each device periodically broadcasts HELLO message to its neighbors to maintain links. If a link breakage is detected, a Route Error (RREQ) message will be sent to the source node.

Service discovery can also support nodes running the same applications to connect to each other. For example, multiple smartphones running the same peer-to-peer photo sharing application could find and connect to each other to transmit photos between them. For such applications, communication targets are not specified and may be changing over time. In the service discovery mechanism, each node maintains a service table to keep track of current services being provided by running applications. Each service entry contains a Service ID and a Service Content that are specified by the application. The Service ID is used as a unique identifier for each service and Service Content contains all required information for connecting to a remote application. When an application issues a service request to find all available communication targets, a service request message containing the requested Service ID is flooded in the network. When a node receiving the service request message has the service available on one of its running application, it unicasts a service reply message back to the source with the requested Service ID and its Service Content. To make this service discovery process more efficient, both service request message and service reply message contains route information (source address, destination address, hop counts and sequence number) to be used by each intermediate node for updating its routing table. So a communication path between the source and target can also be built during this service discovery process.

C. End-to-End Transport Layer

The transport layer is the layer that third party user applications interact with the middleware and provides interfaces for end-to-end communication between applications. Both connectionless and connection-oriented transport services are provided in this layer. Because application data are sent using UDP at the data link layer, where packet delivery is not guaranteed, reliable UDP protocol (RUDP) [3], [13] is used in this layer to achieve reliable unicast over UDP. RUDP extends UDP by adding several additional features such as acknowledgment of received packets, windowing and flow control, retransmission of lost packets, and over buffering. It follows the general principles of TCP's retransmission mechanism and flow control, but does not have any congestion control. A RUDP connection would fail if packet retransmission exceeds certain limit. At that time, the routing layer will be notified of this invalid route and a discovery request for a new route will be issued.

The interfaces provided to application at this layer are simple, easy to use, and similar to network socket API to minimize changes to existing TCP and UDP based applications when using our middleware.

IV. Implementation Using Traditional Layered Networking Structure

The middleware was implemented using the Android Java SDK and tested on Google Nexus One smartphones running Android 2.3 operating system. In order for a smartphone to join an ad hoc network, the phone needs to be rooted so that its WiFi adapter can be accessed and configured to work in ad hoc mode. The middleware is implemented as an Android Service [14] and runs its own process in the background. It can be installed and uninstalled as a normal Android application and requires no OS configuration. Third party user applications communicate with the middleware via Android's AIDL (Android Interface Definition Language) [15], which is used to define the programming interface for inter-process communication (IPC). This feature of Android allows a single instance of our service to communicate with multiple user applications. Our RUDP implementation is based on the Simple Reliable UDP open source project [16] initiated by Adrian Granodos and Marco Carvalho in 2011.

The implementation of our multi-hop ad hoc networking middleware in user space has the following benefits:

- Easier development and testing in the user space than in the Android's Linux kernel.
- A clean network layering model and a more reliable system without interference with the Linux kernel.
- Better portability by having the application runs on top of JVM.

Its performance will not be as fast as a kernel implementation because all packets need to be passed from the kernel space to the user space and processed in user space programs. In addition, Android system runs on top of the Java virtual machine, which doesn't manage the memory as efficient as native code such as C and C++.

A. Application Interface

Instead of directly exposing AIDL interfaces to user applications, we implemented two java libraries for user applications to use for data transmission. The MultihopUDPSocket library is implemented to enable third party applications to broadcast and unicast UDP datagram to destinations through multi-hop paths. The ReliableMultihopUDPSocket library is implemented to enable user to set up end to end connections and provide byte streaming services. The libraries are responsible for setting up AIDL connections to the middleware and passing data to the middleware's transport layer through AIDL interface. In addition, the following AIDL interfaces are implemented for applications to use in service discovery.

- RegisterService(int serviceID, Document doc): When an application wants to make a new service available, this interface is called to add the new service to its service table.
- DiscoverService(int serviceID, int timeout): This interface is called by an application to discover available services in the network. A temporary buffer is created for caching received service replies. After a specified timeout, the cached replies are converted to XML document and returned back to the application.
- UnregisterService(int serviceID): When an application wants to make a service unavailable, this interface is called to remove the specified service entry from service table.

B. Middleware Configuration

The middleware must be turned on and configured before any third party user application can use it. A configuration program with a simple GUI is provided for users to configure the middleware. In Fig. 2, the left image shows the main screen of the configuration program. Users can manually configure networking settings includes ESSID, channel and IP address. Note that in order for devices to join the same ad hoc network, they must use the same ESSID and channel. The IP address of each node is within the 192.168.2/24 prefix and is currently manually assigned by users. By clicking on the "Start Middleware" button, the ad hoc service will be started and available for other applications to use.

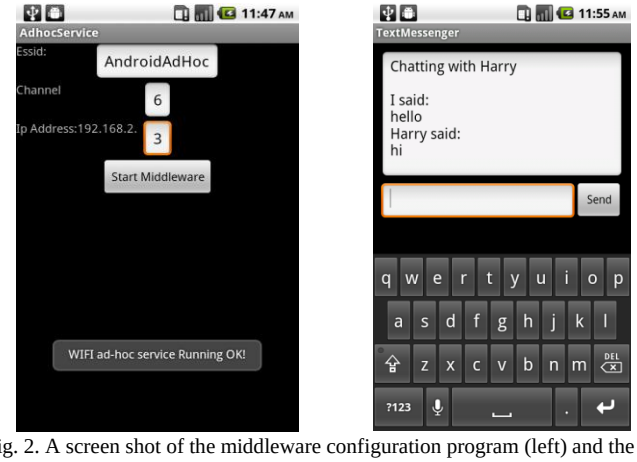


Fig. 2. A screen shot of the middleware configuration program (left) and the text chat demo app (right).

C. Chat and File Sharing Demo Applications

A text chat app and a file sharing app were developed on top of our middleware to verify the correctness of our implementation and evaluate its performance. The applications allow users to chat and share files between smartphones. In Fig. 2, the right image shows a screen shot of a chat session.

V. Implementation Based on Software-Defined Networking (SDN) Principles

Software defined networking (SDN), a new approach for the next generation Internet, is proposed to make networks more adaptable and easily configurable to meet the changing demands and operating environments [17], [18]. Open standards, such as OpenFlow, transforms networking architecture and turns individual network elements into programmable entities [19], [20]. By decoupling the control and data planes, network intelligence and state are logically centralized and the underlying network infrastructure is abstracted from the applications. As a result, highly adaptive, flexible, and scalable networks can be built.

In SDN, a network is separated into three distinct components: communication layer, network operating system (NOS), and control program. Similarly, our implementation has three layers: a) an ad hoc networking layer based on AODV, b) an NOS layer that maintains a global map of the network, manages sub-networks for each application, and supports dynamic change of routing protocols, and c) a control program that controls forwarding rules, routing tables, and routing protocol on the fly. The system enables virtualization of a physical ad hoc network through virtual network slices, which provides a convenient way to implement application-specific routing rules and resource management.

A. Ad Hoc Networking Layer

While traditional SDN uses network routers with proprietary routing protocols, our middleware relies on ad hoc networking services written in the application layer on each node. In the smartphones prototype, AODV over Wi-Fi is used as the underlining networking service. Each node tracks its neighbors through the use of periodic Hello packets. When a route is desired, a route request (RREQ) is broadcast throughout the network. When a route to the destination is found, a route reply (RREP) is sent back to the initiating node, updating routes to both source and destination for intermediate nodes. When a node loses a neighbor a route error (RERR) is sent to nodes that rely on the node's connection with the lost neighbor for routes. Android Interface Definition Language (AIDL) is used for inter-process communication between the three layers.

B. Network Operating System (NOS)

The NOS sends/receives application packets and maintains a network map and network slicing. It scans the network periodically to generate a network map. The node initiating the request sends a map request packet to all single-hop peers, including packet data unit type, a randomly generated integer id, the node address of original requester, a blacklist of already visited nodes, and a global view table. The first node labels itself as the original requester, adds itself to the blacklist, and adds a series of pairs to the global view table. Each pair contains the original node itself and one of its single-hop peers. When a peer receives the packet, it adds

pairs representing its single-hop peers, blacklists itself, and continues forwarding to its peers that are not already blacklisted. The original node collects responses from other nodes to build an adjacency list, representing the whole network.

A logical network refers to our implementation of network slicing. Each logical network running over the physical ad hoc network is distinguished primarily by its logical network tag. Each user application could run in its own logical network, making it easy to implement application specific routing rules and better manage network resources across applications. Our logical network implementation involves the use of a logical network manager in the NOS, which maintains a set of logical network objects. All application data sent through the network must be done via a logical network object, which currently defines a network tag and a set of node members.

C. Control Program

A simple control program is implemented in our system, which communicates with NOS using an INetworkOS AIDL interface. The control program first initializes and configures the ad hoc network. Its network management portion allows the user to view the network map and to update it by changing routing rules. It can display a node's forwarding table, where routing rules can be added or removed.

Fig. 3 and Fig. 4 are representative screen shots. Fig. 3 shows the launch of the SDNAN system via the control program and Fig. 4 shows a view of the forwarding table of a phone, while receiving a route tracing packet.

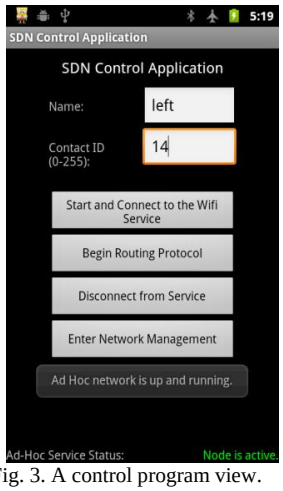


Fig. 3. A control program view.

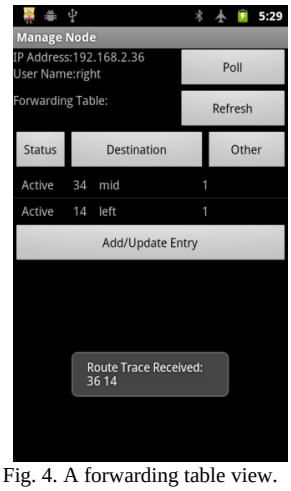


Fig. 4. A forwarding table view.

D. User Application Interface

The interface to user applications is implemented in two ways, using Android Intent system [21] and AIDL, respectively. The intent-based system requires third-party developers to implement an Android BroadcastReceiver, which listens for intents broadcast by the Android operating system. Data sent in an Android intent is provided as a key-value pair, so application developers need to know the keys used for the various Intents used by the NOS.

In the AIDL-based implementation, a small library was developed to simplify the use of AIDL for third-party developers. The library sets up an AIDL connection to the NOS and allows the application to view and communicate with other nodes in the network running the same application.

VI. Experimental Results

In this section, we present the performance results of our middleware implementations in real environments. The experiments were carried out with forced multi-hop setup in a room, as shown in Fig. 5. By programmatically forcing node A and node C to drop each other's packet, we were able to achieve a two-hop networking topology. Each smartphone was placed a few meters away from each other, e.g. 5 meters, and performance data was collected for both single-hop and two-hop communications. Each reported result is an average of 15 trials.

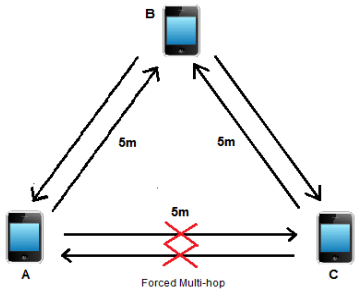


Fig. 5. Experiment set-up of the forced two hop communication.

A. Route Discovery Time of the Traditional Layered Networking Implementation

The route discovery time was measured using phone A as the source and phone C as the destination. The route discovery time is the time taken for phone A to generate a RREQ until a route is established upon receiving a RREP. Note that the number of hops both RREQ and RREP need to go through depends on whether phone B has a valid route to C or not. If phone B has a valid route to C, then it is a one hop route discovery process. Otherwise it is a two hop route discovery process. The results for both one hop route discovery and two hop route discovery are shown in Table I. Due to the fact that the two-hop discovery process has higher delay, the average round-trip delay time (RDT) is increased by 48% from the single-hop discovery.

TABLE I: Experimental Results of Average Route Discovery Times of Different Settings

	Average RDT	Standard Deviation
One hop	62.30 ms	26.84 ms
Two hops	92.21 ms	32.96 ms

B. Throughput of the Traditional Implementation

Using the chat app and the file sharing demo app, we evaluated throughput performance of the middleware. In this experiment, data were sent to destinations through RUDP socket. Since RUDP does not specify any congestion control mechanism, packets are sent at a certain fixed sending rate.

Fig. 6 through 8 shows the average throughput and standard deviations when packets were sent at different rates in the middleware.

The results show that initially throughput increases as sending rate increases. When the sending rate is above a certain value, packet loss will increase and throughput goes down as sending rate increases. In one-hop communications, the maximum throughput is 0.61Mbps, 2.54Mbps, 4.33Mbps, and 5.1Mbps for packet size 1KB, 5KB, 10KB, and 20KB,

respectively. Similarly, in two-hop communications, the maximum throughput is 0.47Mbps, 1.99Mbps, 2.82Mbps, and 3.91Mbps for packet size 1KB, 5KB, 10KB, and 20KB, respectively. When node A sends data through a two-hop path, each data packet is stored and forwarded in the intermediate node B, which adds a delay to the delivery of packet. This delay includes the propagation delay, transmission delay, queuing delay and processing delay. The delay has more significant impact for larger sending rates. When the intermediate node B cannot keep up, the sender will retransmit unacknowledged packets more often, leading to lower throughput.

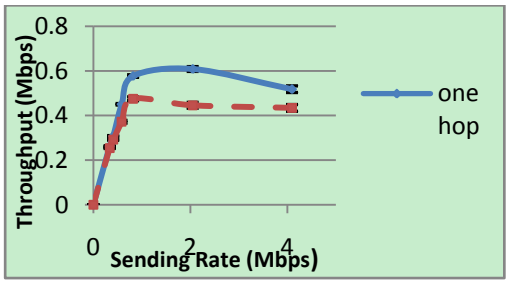


Fig. 6. Communication throughput at different sending rate, for packet size 1KB.

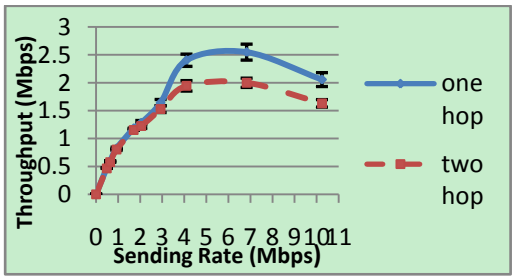


Fig. 7. Communication throughput at different sending rate, for packet size 5KB.

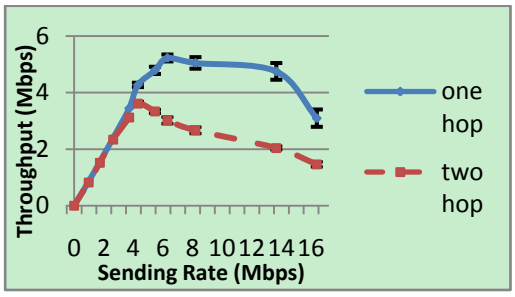


Fig. 8. Communication throughput at different sending rate, for packet size 20KB.

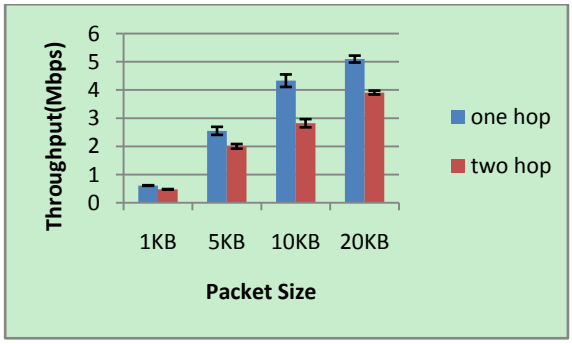


Fig. 9. Average throughput and standard deviation for different packet sizes and number of hops in the 5-meter experiments.

Fig. 9 summarizes the throughput performances for different packet sizes. The results show that larger packet sizes lead to higher throughput, with 20KB the best for both single-hop and 2-hop communications. The reason is that per packet processing overhead is quite high in our user-space middleware implementation and sending data in many small packets is less efficient. By increasing packet size from 1KB to 5KB, the average throughput was improved by 316%. However, when increasing packet size from 10KB to 20KB, the throughput was only improved by 18%. The diminishing return is because increasing packet size also causes more packet fragmentation overhead at the IP layer.

C. Effect of Weaker WiFi Links on Throughput of the Traditional Implementation

The following experiment evaluated throughput of the middle implementation over weaker WiFi links. In an outdoor parking lot, three phones were still placed in a triangle shape, but the distance between each phone was increased to 15 meters and 30 meters, respectively. The results are shown in Fig. 10 and Fig. 11.

The experimental results show that with weaker phone-to-phone WiFi links, the throughput goes down, especially for larger packet sizes. Throughput variations are small when packet size is 1KB and are larger for larger packet sizes. The average throughput is the highest when packet size is 5KB for both the 15-meter and 30-meter settings. In the 30-meter experiment, we failed to get throughput data for 10KB and 20KB packet sizes for the two-hop case. This is because larger packets have a higher chance of packet corruption and dropped packets and the RUDP connection fails after the number of retransmission of a packet exceeds the maximum limit.

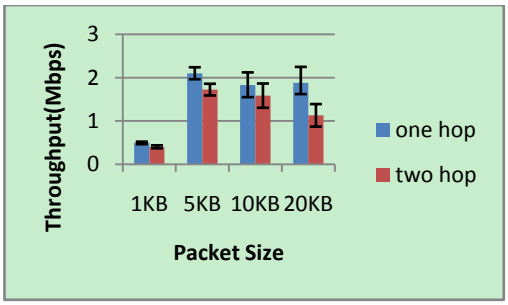


Fig. 10. Average throughput and standard deviation against packet size and number of hops in the 15-meter experiments.

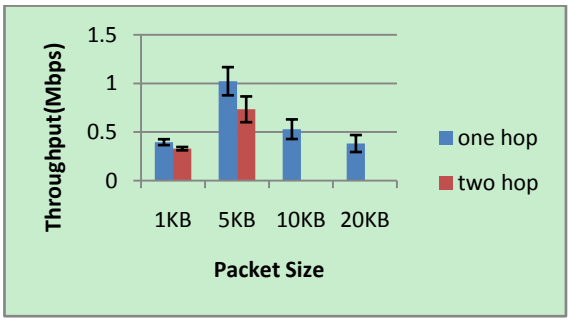


Fig. 11. Average throughput and standard deviation against packet size and number of hops in the 30-meter experiments.

D. Speed of the SDN Implementation

For a 3-node network setup with one phone sending 100 text messages to another phone through one intermediate phone, three implementations were tested: Intents-based SDN implementation, AIDL-based SDN implementation,

and the traditional layered networking implementation. The three implementations have significant speed differences. The traditional, Intents-based, and AIDL-based implementations averaged 733 ms, 2600 ms, and 1600 ms, respectively.

VII. Summary

This paper presents two implementations of an ad hoc multi-hop networking middleware over WiFi on Android devices. The middleware is implemented in the user space and is portable across different versions of Android. Experimental results show that the middleware achieves around 5Mbps transmission bandwidth in single-hop and around 4Mbps bandwidth in 2-hop communications with strong WiFi links, and is sufficient to support some common applications such as text messaging, photo sharing, and multiplayer games.

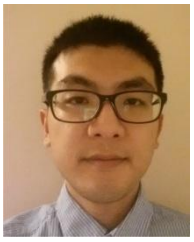
The benefit of the implementation based on traditional networking structures is its efficiency, about twice as fast as the SDN implementation. The major benefit of the SDN implementation is in its clean interfaces between its three layers and the user applications, which makes it much easier to improve the functionality and performance of each component and to develop user applications on ad hoc networks. The text messaging app takes less than 20 lines of code to use the AIDL interface library and less than 50 lines to use the intent interface, whereas implementing similar functionality in the traditional implementation takes hundreds lines of code.

Acknowledgement

This work is supported in part by NSF Grant CNS-1004606.

References

[1] P. Garc á-López, R. Tinedo, and J. Alsina, "Moving routing protocols to the user space in MANET middleware," *Journal of Network and Computer Applications*, September 2010.
[2] C. Perkins, E. Belding-Royer, and S. Das. (July 2003). Ad hoc On-Demand Distance Vector (AODV) Routing. [Online]. Available: http://www.ietf.org/rfc/rfc3561.txt
[3] T. Bova and T. Krivoruchka. (February 1999). Reliable UDP Protocols. [Online]. Available: http://tools.ietf.org/id/draft-ietf-sigtran-reliable-udp-00.txt
[4] M. Nielsen, A. J. Glenstrup, F. Skytte, and A. Gunason, "Real-world Bluetooth MANET Java Middleware," *Technical report TR-2009-120*, IT-University of Copenhagen, 2008.
[5] R. S. Gohs, "Beddernet – Bluetoth Scatternet Framework For Mobile Devices," M.S. thesis, IT University of Copenhagen, 2010.
[6] P. Gupta and R. K. Tuteja, "Design Strategies for AODV Implementation in Linux," *International Journal of Advanced Computer Science and Applications*, vol. 1, no. 6, 2010.
[7] D. Anzaldi, "ORWAR: a delay-tolerant protocol implemented on the Android platform," M.S. thesis, Linkpoings University, 2012.
[8] R. K. Jradi and L. S. Reedtz. (August 2010). Adhoc on Android. [Online]. Available: http://code.google.com/p/ad-hoc-on-android/
[9] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers," in *Proc. ACM SIGCOMM'94*, October 1994, pp. 234-244.
[10] T. Clausen and P. Jacquet, "Optimized link state routing protocol (OLSR)," *IETF Network Working Group RFC 3626*, Oct. 2003.
[11] D. B. Johnson, D. A. Maltz, and Y. Hu. (July 2004). Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR). [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt
[12] K. Lego, P. K. Singh, and D. Sutradhar, "Comparative Study of Ad hoc Routing Protocol AODV, DSR and DSDV in Mobile Ad hoc NETwork," *Indian Journal of Computer Science and Engineering*, vol. 1, no. 4, pp. 364-371, December 2010.
[13] T. L. Kuthethoor *et al*., "Reliable User Datagram Protocol for airborne network," in *Proc. Military Communications Conference*, 2009, pp. 1-6.
[14] Android Service documentation. [Online]. Available: http://developer.android.com/guide/components/services.html.
[15] Android Interface Definition Language (AIDL). [Online]. Available: http://developer.android.com/guide/components/aidl.html
[16] A. Granodos and M. Carvalho. (November 2011). Simple Reliable UDP. [Online]. Avalable: http://sourceforge.net/projects/rudp/
[17] A. Feldmann, "Internet clean-slate design: what and why?" *ACM SIGCOMM Computer Communications Review (CCR)*, vol. 37, no. 3, pp. 59-64, July 2007.
[18] Open Networking Foundation. (April 2012). Software-Defined Networking: The New Norm for Networks. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf.
[19] Scott Schenker. (2011). An attempt to motivate and clarify Software-Defined Networking (SDN). *Ericsson Research*. [Online]. Available: http://www.youtube.com/watch?v=WVs7Pc99S7w&feature=player_embedded#!
[20] The OpenFlow Switch Consortium. (2011). OpenFlow Switch Specification Version 1.1.0. [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf
[21] Android Intents and Intent Filters. [Online]. Available: http://developer.android.com/guide/components/intents-filters.html



Tiancheng Zhuang is presently working as software engineer in Cerner Corporation, Kansas city, Missouri. He received his M.S. degree in computer science from University of Missouri in 2012 and B.E. degree from Hohai University, China in 2009. His area of specialization includes computer networks and mobile computing.

Paul Baskett received his M.S. degree in computer science from the University of Missouri, Columbia in 2012 and his B.S. in computer science from the University of Missouri in 2011. He currently works for MidwayUSA as an Application Developer. His research interests are in mobile computing and wireless sensor networks.

Yi Shang is a professor and director of Graduate Studies in the Computer Science Department, University of Missouri, Columbia, Missouri. He received his Ph.D. degree in computer science from University of Illinois at Urbana-Champaign in 1997, M.S. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 1991, and B.S. degree from the University of Science and Technology of China, Hefei, in 1988. He has published over 140 refereed papers in the areas of nonlinear optimization, wireless sensor networks, mobile computing, intelligent systems, and bioinformatics, and received funding from NSF, NIH, Army, DARPA, Microsoft, and Raytheon. He is a lifetime member of ACM and senior member of IEEE.