Cross Platform Networking Framework to Simplify Mobile Application Development

Federico Cristina[1], Sebastián Dapoto[1], Pablo Thomas[1], Patricia Pesado[1,2]

[1] Instituto de Investigación en Informática LIDI
Universidad Nacional de La Plata

[2] Comisión de Investigaciones Científicas de la Provincia de Buenos Aires
La Plata, Argentina

{fcristina, sdapoto, pthomas, ppesado}@lidi.info.unlp.edu.ar

Abstract. The need for sharing information among mobile devices exists in many applications, and almost every data exchange between these devices involve the same requirements: a means for discovering other mobile devices in a wireless network, establishing logical connections, communicating application data, and gathering information related to the physical connection. This paper presents a cross platform open-source developer-oriented framework that acts as a support layer for host discovery, data communication among devices, and quality of service monitoring. Its purpose is to simplify the issues related to networking for mobile application developers. Currently, the framework is implemented for different platforms, such as Android, J2SE, and J2ME.

Keywords: mobile devices, host discovery, communication, QoS, networking

1 Introduction

A currently increasing trend in mobile environments is the development of applications in which several devices on a network share real time information. These applications rely on some sort of connectivity support in order to achieve the proper interaction among devices. This support can be grouped into three main categories or services: (1) *Host discovery*, a means for searching other reachable devices ready to communicate in a network, (2) *Data communication*, a service for handling the specific exchange of information between devices, and (3) *Quality of service*, a monitoring service that provides QoS related information. Since these services are application-independent, a framework has been implemented in order to support specific aids, simplifying the network-related aspects for developers. The main goal of the proposed framework is to meet these requirements. The features provided allow several types of implementations with different network configurations, such as a typical client/server architecture or a centralized/decentralized peer-to-peer solution.

Even though there are several development frameworks [1, 2] none of them propose an open source, cross platform solution with the proposed features in this paper. Some of these frameworks refer to *networking features* as simply retrieve wireless connection information, but no additional functionality is supported (e.g. PhoneGap [3], Titanium [4]). Other frameworks cover these features, but as a part of a complete solution for a specific domain like games development (e.g. Unity3D [5]). Lastly, some frameworks are proprietary paid solutions for mobile-apps development (e.g. Corona [6]).

The reason for choosing Android as the primary development target for the proposed framework is based on its widespread use and popularity [7]. However, two additional benefits should be mentioned. First, it is an open source software released under the Apache License. This allowed several non-official versions such as Android for x86, ARM, and MIPS architectures. Some examples given in the present paper were tested on these versions running in a Virtual Machine, without the need for real devices. Second, Android Java is functionally compatible with J2SE in matters of network communication. This means that the framework Application Program Interface (API) can be referenced from both types of Java projects. Given that one of the purposes of the framework is to achieve cross platform compatibility, a J2ME version was developed, allowing interoperability between the other platforms. The current implementation of the project can be found at [8] hence the description in this paper will be far from explaining the code (or code details).

The remainder of this paper is organized as follows. The next section describes the proposed framework API. Afterwards, a general overview of the framework and how applications interact with it is provided. The following section presents several applications which make use of the framework features. Finally, the results and benefits of using the framework and an outlook on future work are described.

2 API definition details

This section will present the main classes and interfaces of the framework from an application developer point of view. The highest level of the API is directly focused on application support features (e.g. initial framework configuration) and the lowest level is divided into three main parts, as shown in Fig. 1:

- *HostDiscovery*, for handling the information related to hosts that are ready to communicate to/from each device. As its name suggests, HostDiscovery services/operations include searching for hosts and/or hosts status.
- *NetworkCommunication*, for handling the specific exchange of information between applications. Basically, NetworkCommunication should include the necessary send and receive services/operations for applications.
- *QoSMonitor*, for providing the user and/or programmer the necessary information on signal quality as well as performance indexes such as available network bandwidth.

The initial aim for each part is to achieve a very simple interface for the developer, simplifying the API usage as well device programmability. As a general concept, the framework is designed to support different implementations for each of the services (Discovery, Communication, and QoS). Through an *Abstract factory* pattern [9], the developer can specify which implementation should be used in each case. The details explained in this section go beyond any implementation, covering the issues at a higher level of abstraction.

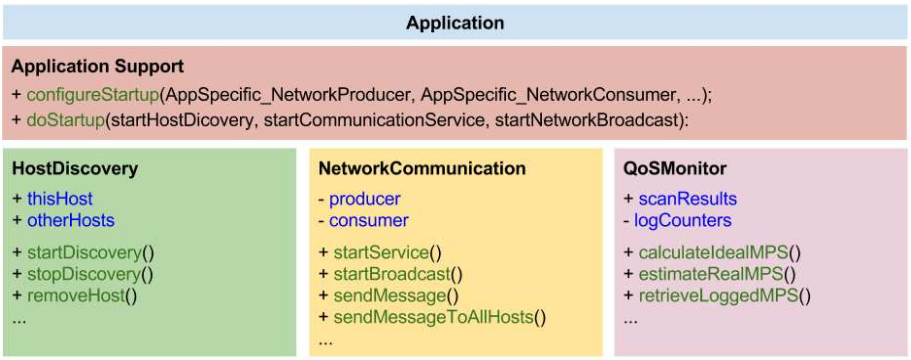

Fig. 1. API Main Components

2.1 Application Data, Producer, Consumer

Generally, the framework will require a data producer, a data consumer, and the data itself to be transferred among hosts. The three will be instances of user-developed classes which extend/implement a specific class/interface. Based on *Inversion of Control* [10, 11], these instances will be passed to the framework as arguments. Specific methods of the instances will be called from the framework in order to generate new data, process incoming data, handle a new host in the network, etc.

The base class for the application-level data is the abstract class *NetworkApplicationData*. This class will be the superclass for any information to be sent/received through the *NetworkCommunication* services. Subclasses must augment the initial data structure as needed.

The producer class is in charge of generating the updated local information to be sent to the other hosts. This class must implement the *NetworkApplicationDataProducer* interface. This interface only requires one method to be implemented, which returns an instance of a subclass of *NetworkApplicationData* with the actual data. This method will be called periodically if the periodic Broadcast feature from the *NetworkCommunication* service is active. If this feature is not desired, then there is no real need for a Producer class to be implemented. However, it is advisable to centralize the creation of data in a specific class.

The consumer handles every type of incoming information, mainly related to application data from other hosts as well as notifications of arrivals and departures of hosts to/from the network. Every time a new message arrives, the framework will invoke a specific method so that the application can act accordingly. A subclass of

NetworkApplicationData object is received as a parameter, containing the actual data. When the *HostDiscovery* service identifies some network change related to hosts, the corresponding method will be called. This allows applications to behave in a specific way under these events (e.g. a host joining or leaving the network).

2.2 Host Discovery

As mentioned above, this service is responsible for searching new hosts in the network as well as exchange host status periodically. The status of a host is simply an online/offline flag in order to know if the host is ready to receive information at a certain moment. The discovery service will make the framework to look/listen for/to new hosts, calling a specific method each time a host joins or leaves the network. The service can be stopped at any time, and this implies neither sending local status nor receiving other hosts status anymore.

The periodicity a host sends its status can be set depending on the application requirements. Support for deactivating discovery as well as the periodicity value are necessary features for the programmer in order to have control on energy and communication overhead/usage. The current list of hosts which are part of the network can be accessed so that at any time, the application would be able to search for specific hosts available and the total number of hosts with which could exchange information.

2.3 Network Communication

Network communication services allow hosts to exchange application-level data in different ways, depending on the specific needs of the application being developed. Client/server, broadcast, and Producer/Consumer communication models are available for the applications. Once started, the service waits for incoming connections from other hosts. An established connection will be used for sending and receiving the application-level data. When a message is received, a *Consumer* will be able to process the incoming information.

Sending a message simply implies specifying the target host and the data to be sent (using *NetworkApplicationData*, as mentioned above). Additionally, a host might need to send information to every online host in the network. The service can be stopped if it is not needed anymore, and this will close all currently established connections.

Sending data to all hosts periodically is also supported. In this case, the framework will require the updated local information in each sending. A *Producer* will have to generate this information. This feature is useful in cases when a constant exchange of data among hosts is needed at regular intervals, for instance in a network game. The application-level periodic data broadcast can be stopped at any time. The periodicity a host sends data can be set depending on the application requirements.

2.4 QoS Monitor

A useful set of QoS features were developed so that each application will be able to decide if it is possible to run under the current network bandwidth, signal strength, etc. At the lowest level of abstraction, an application should be able to ask for the current available bandwidth, so that it will be possible to model the time required to send a message of n data items. Also, some of these performance indexes would depend on wifi signal strength, so it would be useful to provide the application with the current signal strength as well as some previous values so that the tendency would be able to be estimated [12, 13, 14].

From a higher level of abstraction, a method such as *calculateMPS* is desirable for an estimation of the number of application-data messages per second would be able to be exchanged. The key aspect of this feature is to obtain an almost real value based on the network current status at an application-level, as opposed to the estimations provided by the Android API which ignores upper software layers which implies an overhead in communications.

In order to validate this feature, a testing application was developed which calculates the maximum amount of messages per second that a host can exchange in the current network. Fig. 2 shows the results of a series of tests using both alternatives with a fixed size application-data message, considering changes in the signal strength based on the distance of the device to the wireless access point. As expected, the amount of messages per second decreases as the signal strength also decreases. In average, the Android API returns too optimistic estimated values compared to the real ones obtained with the implemented solution.

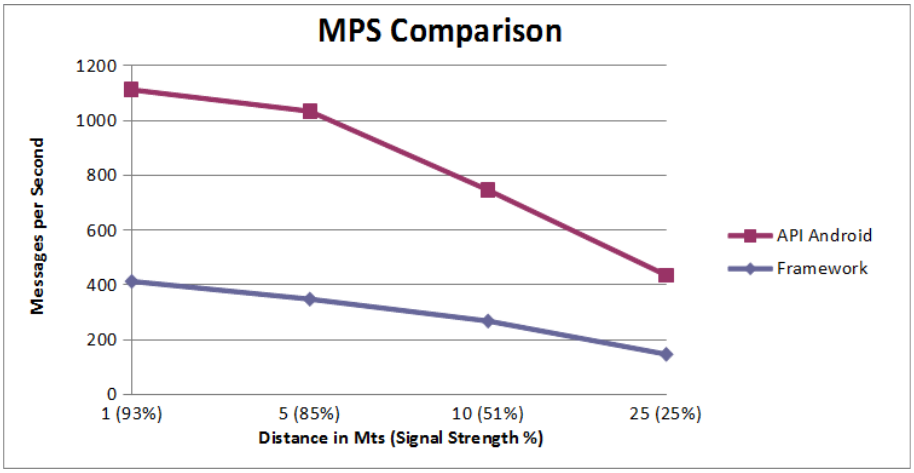

Fig. 2. QoS *calculateMPS* vs Android API bandwith funcion, depending on Signal Strength. Values are normalized to *Messages Per Second* for a correct visual interpretation.

3 Framework interaction

This section will discuss in detail the interaction aspects of the proposed architecture. In order to understand how applications interact with the framework, a simplified example involving the main components of each part will be shown.

3.1 Framework configuration

Before starting any service, the framework requires a *Producer* (a subclass of *NetworkApplicationDataProducer*), a *Consumer* (a subclass of *NetworkApplicationDataConsumer*) and throughout the execution, the information to be exchanged (a subclass of *NetworkApplicationData*).

Fig. 3 shows an example where an application implements the following three main components:

- *AppSpecific_DataProducer*, implementing the *produceNetworkApplicationData()* method.
- *AppSpecific_DataConsumer*, implementing methods such as *newData()*, *newHost()*, *byeHost()*.
- *AppSpecific_NetworkData*, where the default information to be exchanged is augmented with member *aValue*.

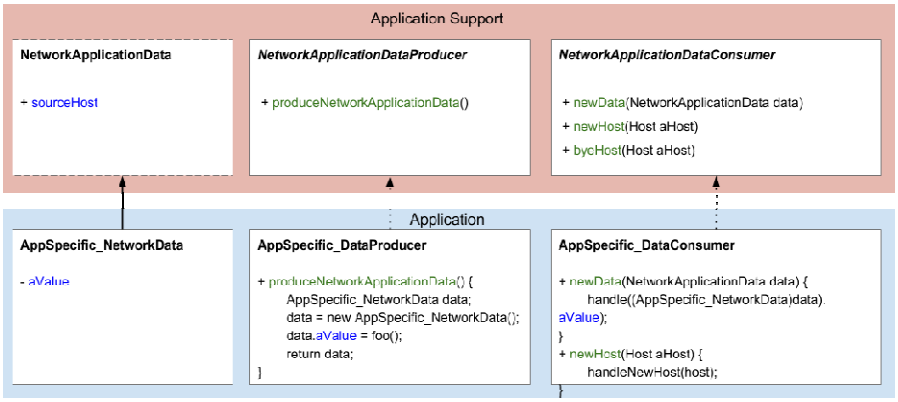

Fig. 3. Framework (top row) and application (bottom row) main interaction components.

3.2 Framework interaction sequence

Fig 4. presents a sequence diagram for a typical scenario. The first stage represents a new host joining the network. *Discovery* service will detect it and inform *AppSpecific_DataConsumer* about this event so that it can work accordingly. In this case, the application decides to establish a connection to this new host, and simply involves calling a *Connection* service method. Afterwards, *Communication* service will periodically ask *Producer* for new information to be sent to other host. The only

task for *Producer* is simply return an updated instance of *AppSpecific_NetworkData*. This information will be handled by the framework, and once it reaches the destination host, *Communication* will notify its consumer that new data has arrived (through the *newData()* method). At some point, a host may leave the network and *Discovery* will inform this situation to the *Consumer*.

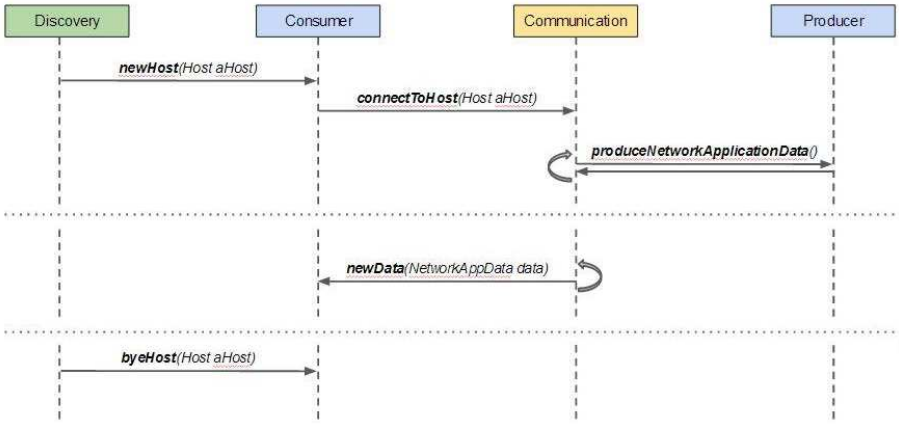

Fig. 4. Sequence and interaction diagram. *Discovery* and *Communication* are components of the framework. *Consumer* and *Producer* are application-level subclasses of *NetworkApplicationDataConsumer* and *NetworkApplicationDataProducer*.

4 Examples

This section will present real examples in which the network requirements for each application differs considerably, among other factors. The first one is a competitive multiplayer Asteroids-like game (referred to as Asteroids, from now on). The second one is a two players Tic-Tac-Toe game, both currently running in Android. The third example is a simple chat application implemented both in Android and J2ME in order to show cross platform communication. The fourth example is a Client/Server Wi-Fi remote control running on Android for an image display server running on J2SE (from now on, WiFiRemote), in order to show heterogeneous application interaction from a platform point of view.

These projects are *completely* built on top of the framework project [8], i.e. there is no access to other services beyond those provided by the framework. The complete code of the first two examples can be found at [15] and [16] respectively. For the third example, the J2ME version of the chat application is built on top of the J2ME version of the framework project [17].

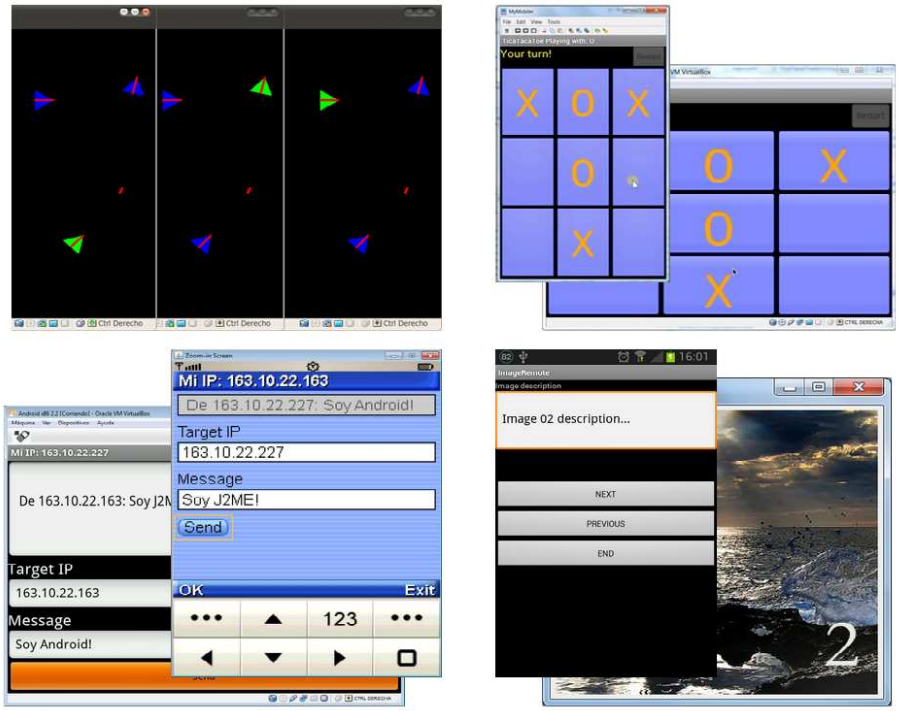

Fig. 5. Asteroids running on three Android x86 v2.2 virtual machines (top left), b) Tic-Tac-Toe running on two Samsung Galaxy SII mobile devices with Android 4.0.3 (top right), c) Chat application running on Android x86 and J2ME Emulator (bottom left), d) WiFiRemote, server running on desktop J2SE and client running on Android 4.1.2 (bottom right).

4.1 Asteroids

Multiplayer Asteroids is a very simple game, in which a ship (controlled by a user) must destroy enemy ships firing laser shots. Every ship corresponds to a user in a host (e.g. mobile device, tablet) in the network, as shown in 5a. The local ship will be rendered in green and remote ships will be rendered in blue. An example video of the game can be found at [18], where it is also shown that the entire example is run on virtual machines with Android.

Although very basic, the application is representative in terms of CPU and network usage of a class of game applications: the game must continuously update its local model, share local information among all hosts, receive and update remote hosts information, and render the corresponding graphics. Considering an update rate equivalent to 30 frames per second, the network consumption is considerably high and grows proportionally to the number of players. Furthermore, the game uses the *Periodic Broadcast* feature from the *Communication* service.

The data defined to be sent/received through the network includes ship position and heading, as well as shots position and heading that the ship shoots when the user triggers the fire action.

4.2 Tic-Tac-Toe

Tic-Tac-Toe has been selected as a representative example of a completely different type of application, compared to the Asteroids game, since Tic-Tac-Toe is a two-players game, turn-based and there is no need for a continuous sending of information, specific events (players taking turns) trigger communications.

Fig. 5b shows a running example of the game on two Samsung Galaxy devices with Android 4.0.3, and an example video of the game running on a virtual machine and a Samsung Galaxy can be found at [19]. While the Tic-Tac-Toe game imposes a very different usage of the network during the game (turns, non-periodic messages, etc.) as compared to the Asteroids game, other service requirement such as those related to host Discovery remain the same.

The data structure for this application is very simple: an action value representing the possible states of the game: a) resolve who will start the game, b) set a cell with an X or an O - in this case a position value is also needed, or c) restart the game. Since there is no need for a periodic update of local host information, no *Producer* has to be implemented.

4.3 Cross platform chat application

A simple chat application has been selected in order to show cross platform networking capability, requiring only the communication features. By simply specifying an IP address and a message, the chat-app sends the corresponding text to the target host, the which shows its content on the display. Fig. 5c shows the achieved interaction among two virtual devices, one running the application on Android, and the other running on J2ME.

The biggest problem in this case is the serialization-deserialization issue. Each platform implements (if it does) a specific serialization method, which can or cannot be compatible with the other platforms. In order to solve this problem, the framework defines a *NetworkSerializable* interface, containing the definition for the *networkSerialize* and *networkDeserialize* methods. Applications must contain a class which implements this interface in a consistent way on each platform. At run time, the framework then delegates the serialization-deserialization work to these classes.

4.4 WiFiRemote

WiFiRemote is a Client/Server implementation using different application platforms. It consist of a server application that displays images running on J2SE, and an

Android application that controls the slideshow (like a remote control) on the cliente side, as shown in Fig. 5d.
Thus, a user can then control the images being displayed, for instance selecting the previous or next image. For each image displayed, the server also sends to the client the image metadata, which will be displayed in the Android device.
In this case, the required data structure is quite simple: an action code that goes from the client to the server and the details of the image that returns to the client.
This is an example of how the framework is also useful in applications in which implementation logic differs in each host.

5 Conclusions and future work

This paper presented the advances achieved in the implementation of a framework designed for easily handling network-related issues in the development of mobile applications, called *NetworkDCQ* [20].

The framework covers a wide range of features such as host discovery, data communication and broadcasting, and QoS monitoring. It is designed to support different implementations for each of these services, gaining flexibility, and versatility. Its main goal is to fill a gap in the mobile development frameworks area, where currently there is no open source, cross platform solution with the features explained in this paper.

The proposed API and reference implementation is actually useful for several types of applications, network requirements, and configurations. The examples shown cover applications with a wide variety of network-related requirements like continuous data broadcasting, event driven communication, and heterogeneous platforms. These examples evidence the considerably small amount of effort needed in the development of applications with networking capability, thanks to the features included in the framework.

Although the framework is fully functional for Android and J2SE, currently there is no available version for iOS. Completing this task is a short-term objective. Implementing the complete set of features for Windows Mobile, and BlackBerry 10 are mid to long-term objectives.

References

1. Markus Falk, Mobile Frameworks Comparison Chart, http://www.markus-falk.com/mobile-frameworks-comparison-chart/.
2. Digital Possibilities, Mobile Development Frameworks Overview, http://digital-possibilities.com/mobile-development-frameworks-overview/.
3. PhoneGap, http://phonegap.com/.
4. Titanium, http://www.appcelerator.com/platform/titanium-platform/.
5. Unity3D, http://unity3d.com/.
6. Corona, http://www.coronalabs.com/products/corona-sdk/.

7. StatCounter, Top Mobile & Table Operating Systems from April 2013 to April 2014, http://gs.statcounter.com/#mobile+tablet-os-ww-monthly-201304-201404.
8. NetworkDCQ for Android Project, https://code.google.com/p/networkdcq/.
9. Gamma E., Helm R. , Johnson R. , Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, 1994.
10. Martin, R. C., The Dependency Inversion Principle, 1996, http://www.objectmentor.com/resources/articles/dip.pdf.
11. Fowler, M., Inversion of Control Containers and the Dependency Injection Pattern, http://martinfowler.com/articles/injection.html.
12. Joel Gonçalves, Luis Lino Ferreira, A Framework for QoS-Aware Service-based Mobile Systems, 2010, in press.
13. Rabia Ali, Dr. Fareeha Zafar, Bandwidth Estimation in Mobile Ad-hoc Network (MANET), 2011, in press.
14. R.Sivaraman, V.R.Sarma Dhulipala, L.Sowbhagya, B.Vishnu Prabha, Comparative Analysis of QoS Metrics in Mobile Ad Hoc Network Environment, 2009, in press.
15. Asteroids for Android Project, http://code.google.com/p/asteroidsa/.
16. Tic-Tac-Toe for Android Project, http://code.google.com/p/ticatacatoe/.
17. NetworkDCQ for J2ME Project, https://code.google.com/p/networkdcq-j2me/.
18. Asteroids for Android Example Video, http://www.youtube.com/watch?v=HiRTk8daqi4.
19. Tic-Tac-Toe for Android example video, http://www.youtube.com/watch?v=mrf01putSec.
20. Cristina F:, Dapoto S., Tinetti F., Encinas D., Thomas P, Pesado P., NetworkDCQ: A Multi-platform Networking Framework For Mobile Applications, 2013, in press.