NetworkDCQ: A Multi-platform Networking Framework For Mobile Applications

Federico Cristina[1], Sebastián Dapoto[1], Fernando G. Tinetti[1,2], Pablo Thomas[1], Patricia Pesado[1,2]

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

[2] Comisión de Investigaciones Científicas de la Provincia de Buenos Aires - Argentina

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

Abstract. Currently, the number of mobile applications that require (wireless) connectivity is constantly increasing. 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 proposes an open source developer-oriented framework that acts as a network support layer for host discovery, data communication among devices, and quality of service characterization, which can be used for developing several types of applications and is proposed for different platforms, such as Android Java, J2SE, and J2ME.

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

1 Introduction

The middleware presented in this paper, called NetworkDCQ, is proposed bearing in mind the evolution of mobile devices as well as specific network requirements of mobile applications. The following subsections briefly explain three topics: trends in mobile devices, mobile network applications, and the initial development platform selected for (a proof-of-concept) implementation.

The remainder of this paper is organized as follows. The next section describes the proposed Application Program Interface (API). Afterwards, an architectural overview of the framework is given. The following section presents several applications which use NetworkDCQ. Finally, we describe the results and benefits of using the proposed framework and conclude with an outlook on future work.

1.1 Trends in Mobile Devices

The worldwide internet mobile traffic is expected to overtake the desktop internet traffic by 2014 [1], which means that more users will be accessing the Internet through their mobile phones than through their PCs. This phenomena has already been experienced in some countries, like China [2] or India [3, 4].

Currently, nearly 50% of recent device sales are mobile (smartphones, tablets) [5]. Mobile applications are tightly related with this trend. The increasing number of these devices in the last years has led to a revolution in terms of mobile application development and usage. Among all OS mobile systems, Android is by far the most deployed platform [4, 6], with 136 million units shipped and 75% market share in Q3 2012 [7], seconded by iOS and BlackBerry OS with 14.9% and 7.7% market share respectively. Additionally, Android has a large community of developers writing applications that extend the standard functionality of the devices. Google play has hit the 25 billion-download mark by September 2012 [8].

1.2 Mobile Network Applications

Although there is a large number of standalone mobile applications (which require no connectivity at all), 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: a) *Host discovery*, a mean for searching other reachable devices ready to communicate in a network, b) *Data communication,* a service for handling the specific exchange of information between devices, and c) *Quality of service,* a monitoring service that provides QoS related information.

Since these services are application-independent, a framework can be implemented in order to support specific aids, simplifying the network-related aspects to the developer. The main purpose of the proposed infrastructure is to meet these service's requirements. The features provided by *NetworkDCQ* 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 mobile development frameworks [9, 10], none of them proposes an open source, multi-platform solution that presents the features proposed in this paper. Some of these frameworks refer to *networking features* as simply retrieve wireless connection information, but no additional functionality is supported. Other frameworks cover these features, but as a part of a complete paid solution for mobile-apps development. The most representative examples are PhoneGap [11], Unity3D [12], Titanium [13] and Corona [14].

1.3 Development Platform

The reason for choosing Android as the primary development target for the proposed framework is based on its widespread use and popularity (as previously explained).

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 much richer than J2ME. Actually, the similarities with J2SE API (Application Programming Interface) led to the Oracle vs Google lawsuit [15]. As will be shown, this is a considerable advantage due the compatibility between both languages in matters of network communication. This means that the proposed API can be referenced from both types of Java projects. Given that one of the purposes of the framework is to achieve multi-platform compatibility, a J2ME version is also being developed, allowing interoperability between the other platforms.

2 Proposed and developed API

The main goal is having a minimal (yet useful) communication-related software infrastructure so that different mobile devices can be programmed. The focus is on the Java language since it is (by definition) cross platform. Even when currently development platforms tend to be very different, it is possible to use Java in almost all of them. While the first problem to be solved is programmability, other issues such as interoperation are left open for future release/development. This section will present the main classes and interfaces of the framework from an application developer point of view. Based on the previous analysis, and the types of interaction required among hosts, the highest level of the API is directly focused on application data communication (Application Support) 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 startup time (latency) and available network bandwidth.

The initial aim for each part is to achieve a very simple interface for the user, 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 [16], the user 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.

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 [17, 18], 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. Currently, the only information contained in this class is a reference to the source host (the one that originates the message). Subclasses must augment the data structure as needed, and any data type/object can be used as long as it implements the *Serializable* interface.

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 the method *produceNetworkApplicationData* 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. The period is given by the user in milliseconds, also provided by the API. If this feature is not desired, then there is no real need for this class to be implemented. However, it is advisable to centralize the creation of data in a specific class. In this case, calls to *produceNetworkApplicationData* method will have to be done manually from some application-level class when required.

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 the *newData* method so that the application can act accordingly. A *NetworkApplicationData* object is received as a parameter, containing the actual data. The consumer will have to cast this object to the corresponding application-level data type. 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 in this cases. Thus *newHost* or *byeHost* methods will be called when there is a new host in the network or when a host leaves the network respectively.

2.2 Host Discovery

As mentioned above, this service is responsible of searching for 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 can be started simply by invoking the *startDiscovery* method. This will make the framework to look/listen for/to new hosts, calling a specific method each time a host joins or leaves the network. When the service is not needed anymore, the *stopDiscovery* method can be invoked. 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. Making available *stopDiscovery* as well as the periodicity value to the programmer is necessary 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 through the *otherHosts* collection 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 (provided by *NetworkCommunication*) 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 to the applications. In order to establish an application-level communication with other hosts, the *startService* method must be started. Once started, the service waits for incoming connections from other hosts. A host can establish a connection to another host through the *connectToServerHost* method. 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), through the *sendMessage* method. Additionally, a host might need to send information to every online host in the network calling the *sendMessageToAllHosts* method. When the service is not needed anymore, the *stopService* should be called. This will close all currently established connections.

Also, the framework is able to handle sending data to all hosts periodically. In this case, NetworkDCQ will require in each sending the updated local information. A *Producer* will have to generate this information. This feature is available by calling the *startBroadcast* method and 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 by simply invoking *stopBroadcast* method. The periodicity a host sends data can be set depending on the application requirements.

2.4 QoS Monitor

A useful set of services is currently being defined, 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 startup and 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 wi-fi 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. From a

higher level of abstraction, a method such as *calculateMPS* for an estimation of the number of application-data messages per second would be able to be exchanged, and it would aggregate some low level information, along with the specific application data to be communicated periodically. Although an initial API is proposed, this service is currently under development and unavailable to user applications.

3 NetworkDCQ proposed architecture

This section will discuss in detail the implementation aspects of the proposed architecture. As mentioned before, the framework supports different implementations for each low level service. Currently, an *UDPDiscovery* and *TCPCommunication* was developed for *HostDiscovery* and *NetworkCommunication* services respectively, and *QoSMonitor* is under development. Fig. 1 shows the most relevant details on each layer, which will be explained in the following subsections (excepting QoSMonitor).

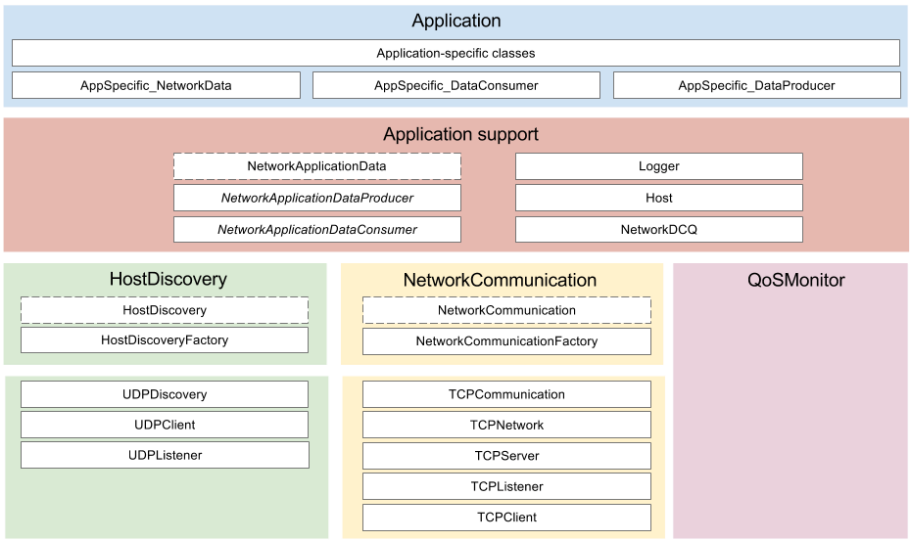

Fig. 1. Detailed Architecture of the Framework

In Fig. 1, abstract classes are identified with dotted lines, and interfaces are those in italic font. The current implementation of the project can be found at [19] hence the description in this section will be far from explaining the code (or code details), which can be downloaded, used, etc. Section 4 will explain in detail (via specific examples) the step-by-step guide in order to configure and use every feature of the framework.

3.1 Application support

This layer involves additional classes which are referenced along several parts of the

framework. For instance, *NetworkApplicationDataConsumer* is related with Discovery and Communication services. Host instances exist in Discovery, but they are also used in Communication. A special class in this layer is *NetworkDCQ*, which is explained in detail in the next subsection.

3.1.1 NetworkDCQ

This class is the framework main entry point, and has two main static methods. Method *configureStartup* allows the developer to specify the Producer and Consumer instances. Method *doStartup* is the one in charge of starting each service or feature (discovery, communication, broadcast), since they can be started independently. It is expected that *configureStartup* is called before any usage of the framework and method *doStartup* identifies the point from which the application would start using every framework service (discovering hosts, establishing communication/s, etc.).

3.2 UDPDiscovery

UDPDiscovery is the implementation of *HostDiscovery*, extending its abstract class. As such, it implements *startDiscovery* and *stopDiscovery* methods. When the discovery service is started, the *UDPDiscovery* spawns two threads: *UDPListener* and *UDPClient* as shown in Fig. 2a. The former first joins the network group via a *MulticastSocket*, and then waits for incoming host status updates from other hosts. The latter periodically sends multicast packets with its local host status.

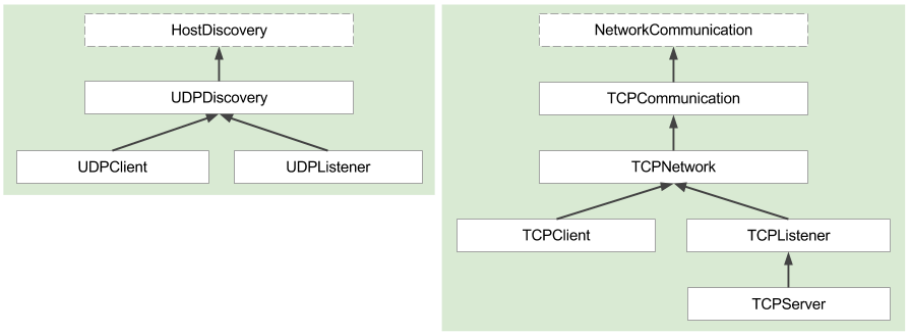

Fig. 2. a) *UDPDiscovery* Hierarchy, b) *TCPCommunication* Hierarchy

UDPDiscovery has an additional responsibility, which is to check for hosts that leave the network without giving the proper signal. This is achieved by a connection timeout validation, i.e. by checking - for each remote host - the timestamp of the last received status update. If the lapse of time exceeds a predefined threshold, then the host is removed from *otherHosts* list and *byeHost* method is invoked. This validation is executed periodically.

3.3 TCPCommunication

TCPCommunication is the implementation of *NetworkCommunication*, extending its abstract class. This service will spawn several threads, depending on the framework configuration. The following is a brief explanation of the methods discussed above and taking into account the details shown in Fig. 2b.

Method *startService* will spawn a *TCPListener*, in charge of listening for new TCP connections from other hosts. For each new connection, this class will spawn a new *TCPServer* thread, which is in charge of receiving *NetworkApplicationData* objects from a specific host.

Method *startBroadcast* will spawn a *TCPCommunication* thread, which will periodically send a *NetworkApplicationData* object (relying on the configured *NetworkApplicationDataProducer* that generates the data), using the *sendMessageToAllHosts* method. This last method simply iterates the *HostDiscovery.otherHosts* collection, and calls *sendMessage* method in each case.

TCPCommunication has a pool of *TCPClient* objects (the ones in charge of writing data through a socket), one for each host. Method *connectToServerHost* instantiates a new *TCPClient* when invoked and will keep it in the pool for later use. Every time a message is sent to a host, *TCPCommunication* first retrieves the corresponding connection with that host, avoiding having to reconnect continuously.

4 Examples

In this section three different examples will be discussed, in which the network requirements for each application differs considerably. The first one is a competitive multiplayer Asteroids-like game (referred to as Asteriods, from now on) and 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 multi-platform communication.

In each case, sample code will be given in order to highlight the most relevant details related to networking. The complete code of the first two examples can be found at [20] and [21] respectively. Also, these projects are *completely* built on top of the NetworkDCQ project [19], i.e. there is no access to other Host Discovery and Communication services beyond those provided by the NetworkDCQ framework. For the third example, the J2ME version of the chat application is built on top of the J2ME version of the NetworkDCQ project [22].

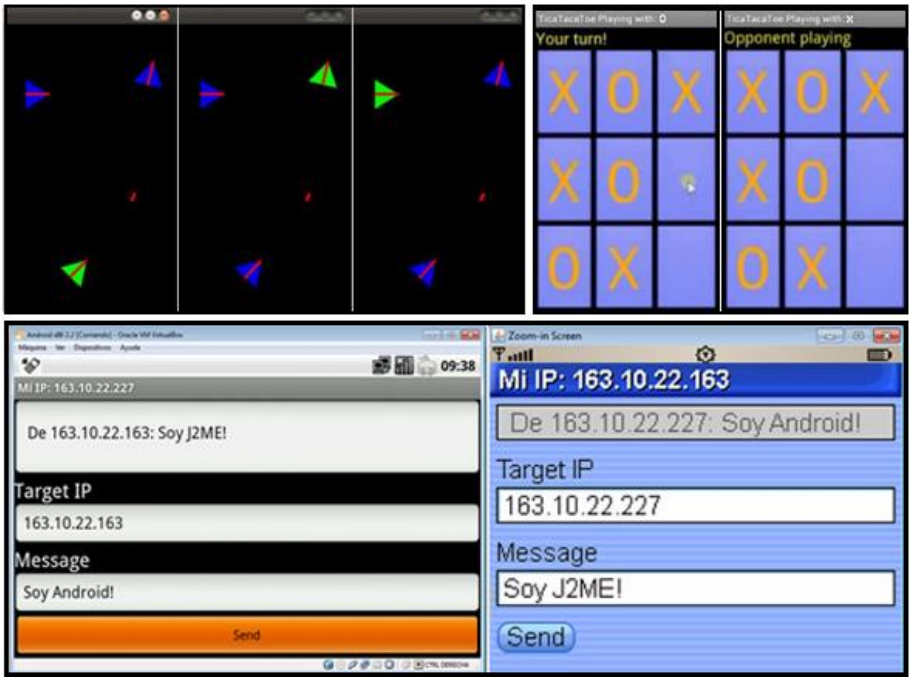

Fig. 3. a) Asteroids running on three Android x86 v2.2 virtual machines, b) Tic-Tac-Toe running on two Samsung Galaxy SII mobile devices with Android 4.0.3, c) Chat application running on Android x86 (left) and J2ME Emulator (right).

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 (i.e. mobile device, tablet, etc.) in the network, as shown in Fig. 3a. 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 [23], 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. The producer has a unique and reusable *AsteroidsNetworkApplicationData* instance (in order to avoid continuous Garbage Collector calls), which is filled in new data every time is needed with its current

values according to the model changes. The Consumer is the place where remote ships information is updated with the received data. A cast to *AsteroidsNetworkApplicationData* is needed in order to retrieve the members in the instance (ship heading, position, etc.). The last step simply requires setting the corresponding application-level instances of *Producer* and *Consumer* of the framework, and starting the Discovery, Communication and Broadcast services.

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 Asteriods 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. 3b 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 [24]. While the Tic-Tac-Toe game impose 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. The Consumer is the place where each remote action is replicated locally (e.g.: the other player placed an X in cell 7). A cast to an application-level data type is needed in order to retrieve the members in the instance (action and position if needed). As explained previously, the sending of information is not performed periodically. The application sends a message to the other host each time an action event occurs (e.g. when the user clicks in one of the nine cells).

The application access the *Communication* service through the static method *NetworkDCQ.getCommunication* in order to use the *sendMessage* method. The other host is retrieved by accessing the *HostDiscovery* static member *otherHosts*. The final step is starting the required services. In this case, the Producer and the Broadcast service will not be started.

4.3 Multi-platform chat application

A simple chat application has been selected in order to show multi-platform networking capability, requiring only the NetworkDCQ 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. 3c 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, NetworkDCQ 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, NetworkDCQ then delegates the serialization-deserialization work to these classes.

5 Conclusions and Further Work

The paper presented a framework for handling network-related issues in the development of applications running on mobile devices, such as host discovery, data communication and broadcasting; 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, multi-platform solution with the features proposed in this paper.

The proposed API and reference implementation is actually useful for several types of applications, network requirements, and configurations. The examples shown in the previous section cover applications with a wide variety of network-related requirements like continuous data broadcasting and event driven communication.

Using Android as a general development platform allowed an immediate integration with J2SE applications. Additionally, specific interaction problems with other platforms where solved by defining the corresponding interfaces and development methodologies, allowing communication with platforms such as J2ME.

As explained previously, the QoS service is still in development. Completing this feature is a short-term objective. Implementing the complete set of features for iOS, Windows Mobile, and BlackBerry 10 are mid to long-term objectives.

References

1. Morgan Stanley. The Mobile Internet Report, 1st edition. (2009)
2. China Internet Network Information. China Internet Development Statistics Report. (2012).
3. Mobile vs Desktop Internet Traffic Report from Oct 2011 to Oct 2012. http://gs.statcounter.com/#mobile vs desktop-IN-monthly-201110-201210.
4. Meeker, M. D10 Conference. Internet Trends. (2012), http://www.kpcb.com/insights/2012-internet-trends.
5. Asymco. The Rise and Fall of Personal Computing (2012), http://www.asymco.com/2012/01/17/the-rise-and-fall-of-personal-computing/.
6. Gartner, Inc. Nov.2012 Press Release, http://www.gartner.com/it/page.jsp?id=2237315.
7. IDC. Nov.2012 Press Release, https://www.idc.com/getdoc.jsp?containerId=prUS23771812.
8. Google, Inc. Google Official Blog (2012), http://officialandroid.blogspot.com.ar/2012/09/google-play-hits-25-billion-downloads.html
9. Markus Falk. Mobile Frameworks Comparison Chart, http://www.markus-falk.com/mobile-frameworks-comparison-chart/
10. Digital Possibilities. Mobile Development Frameworks Overview

http://digital-possibilities.com/mobile-development-frameworks-overview/
11. PhoneGap, http://phonegap.com/
12. Unity3D, http://unity3d.com/
13. Titanium, http://www.appcelerator.com/platform/titanium-platform/
14. Corona, http://www.coronalabs.com/products/corona-sdk/
15. Reuters. Oracle sues Google over Android (2012), http://www.reuters.com/article/2010/08/13/us-google-oracle-android-lawsuit-idUKTRE67B5G720100813
16. Gamma, E. Design Patterns: Elements of Reusable Object-Oriented Software (1994).
17. Martin, R. C. The Dependency Inversion Principle. (1996), http://www.objectmentor.com/resources/articles/dip.pdf
18. Fowler, M. Inversion of Control Containers and the Dependency Injection Pattern, http://martinfowler.com/articles/injection.html
19. NetworkDCQ for Android Project, https://code.google.com/p/networkdcq/
20. Asteroids for Android Project, http://code.google.com/p/asteroidsa/
21. Tic-Tac-Toe for Android Project, http://code.google.com/p/ticatacatoe/
22. NetworkDCQ for J2ME Project, https://code.google.com/p/networkdcq-j2me/
23. Asteroids for Android Example Video, http://www.youtube.com/watch?v=HiRTk8daqi4
24. Tic-Tac-Toe for Android example video, http://www.youtube.com/watch?v=mrf01putSec