# NetworkDCQ: Multiplatform Network Support for Discovery, Communication, and QoS in Mobile Devices

**Fernando G. Tinetti**$^{(1)(2)}$, **Federico Cristina**$^{(1)}$, **Sebastián Dapoto**$^{(1)}$

$^1$III-LIDI – Fac. de Informática – Universidad Nacional de La Plata (UNLP)
50 y 120 – 2do. Piso – La Plata – Argentina

$^2$Comisión de Investigaciones Científicas de la Prov. de Bs. As. – La Plata – Argentina

Technical Report UNLP-PD-2012-2
December 2012

{fernando, fcristina, sdapoto}@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 mean of discovering other mobile devices in a wireless network, establishing logical connections, communicating application data, and gathering information related to the physical connection. This work 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 features, which can be used for developing several types of applications and is proposed for different platforms, such as Android Java, J2SE, and J2ME in the near future.*

## 1. Introduction

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

### 1.1. Evolution of Mobile Devices

The worldwide internet mobile traffic is expected to overtake the desktop internet traffic by 2014 [Morgan Stanley 2009], 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 [China Internet Network 2012] or India [StatCounter Global Stats 2012] [Meeker 2012].

Currently, nearly 50% of recent device sales are mobile (smartphones, tablets) [Asymco 2012]. 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 [Gartner, Inc. 2012] [Meeker 2012], with 136 million units shipped and 75% market share in Q3 2012 [IDC 2012], 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 [Google, Inc. 2012].

### 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 network 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:

- Host discovery: A mean for searching other reachable devices ready to communicate in a network/exchange application data.
- Data communication: A service for handling the specific exchange of information between devices.
- Quality of service: A monitoring service that provides QoS related information.

Given that 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 fulfill these service's requirements.

The features provided by the framework proposed in this work allow several types of implementations with different network configurations, such as a typical client/server architecture or a centralized/decentralized peer-to-peer solution.

### 1.3. Development Platform

The main 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 [Reuters 2012]. 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 will also be developed, ideally allowing interoperability between the other platforms.

## 2. 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 inter-operation are left open for future releases/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 API is directly focused on application data communication (Application Support) and the lowest level API is divided into three main parts: *HostDiscovery*, *NetworkCommunication*, and *QoSMonitor*, 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.*

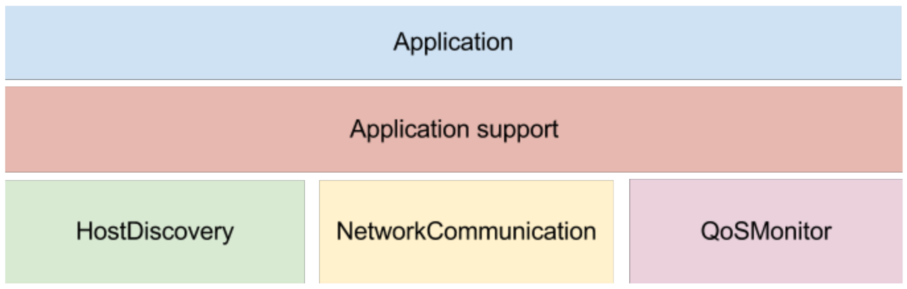

**Figure 1. Main Structure of the Framework**

The main goal 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). A *factory* then allows the user to choose which implementation should be created in each case. The details explained in this section go beyond any implementation, covering the issues at a high level of abstraction.

## 2.1. Application Data, Producer and Consumer

In general, 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 [Martin 1996] [Fowler 2004], these instances will be passed to the framework. 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.

### 2.1.1. Application Data

The base class for the application-level data is the abstract class *NetworkApplicationData.* This abstract class will be the superclass for any information to be sent/received through the *NetworkCommunication* services:

```
public abstract class NetworkApplicationData implements
  Serializable
```

Currently, the only information contained in this class is a reference to the source host (the one that originates the message),

```
protected Host sourceHost = null;
```

Subclasses must augment the data structure as needed, and any data type/object can be used as long as it implements the Serializable interface.

### 2.1.2. Producer

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

```
public interface NetworkApplicationDataProducer
```

This interface only requires one method to be implemented, which returns an instance of a subclass of *NetworkApplicationData* with the actual data

```
public NetworkApplicationData produceNetworkApplicationData();
```

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 (BROADCAST_LOCAL_STATUS_INTERVAL_MS). 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, calling to the *produceNetworkApplicationData* method will have to be done manually from some application-level class when required.

### 2.1.3. Consumer

The consumer is in charge of handling 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.

```
public interface NetworkApplicationDataConsumer
```

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 receiver (consumer) will have to cast this object to the corresponding type.

```
public void newData(NetworkApplicationData receivedData);
```

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

```
public void newHost(Host aHost);
public void byeHost(Host aHost);
```

will be called when there is a new host in the network or when a host leaves the network respectively.

## 2.2. HostDiscovery

As mentioned above, this service is in charge 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.

```
public abstract boolean startDiscovery();
```

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.

```
public abstract void stopDiscovery();
```

The periodicity a host sends its status can be set simply by changing the corresponding value, depending on the application requirements:

```
public static int DISCOVERY_INTERVAL_MS = 1000;
```

Making available *stopDiscovery* as well as the periodicity value to the programmer is necessary in order to have control on energy and communication overhead/channel usage. The current list of hosts which are part of the network can be accessed through the otherHosts collection

```
public static Map<String, Host> otherHosts;
```

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. NetworkCommunication

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.

```
public abstract boolean startService();
```

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.

```
public abstract  boolean connectToServerHost(Host target);
```

Sending a message simply implies specifying the target host and the data to be sent (using *NetworkApplicationData*, as mentioned above).

```
public abstract void sendMessage(Host targetHost,
  NetworkApplicationData data);
```

Additionally, a host might need to send information to every online host in the network.

```
public abstract void sendMessageToAllHosts(
  NetworkApplicationData data);
```

When the service is not needed anymore, the *stopService* should be called. This will close all currently established connections.

```
public abstract boolean stopService();
```

Also, the framework is able to handle sending data to all hosts periodically. In this case, NetworkDCQ will require in each send the updated local information. 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.

```
public abstract boolean startBroadcast();
```

The application-level periodic data broadcast can be stopped by simply invoking *stop-Broadcast* method.

```
public abstract boolean stopBroadcast();
```

The periodicity a host sends data can be set simply by changing the corresponding value, depending on the application requirements:

```
public static int BROADCAST_LOCAL_STATUS_INTERVAL_MS = 30;
```

### 2.4. QoS Monitor

A useful minimal 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, message startup time, 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 as shown in Eq. (1).

$$t(n) = \alpha + \beta * n \tag{1}$$

where $\alpha$ is the network startup time and $\beta$ is the inverse of the network bandwidth. 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

```
public abstract int 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 such as the previous time message model, $t(n)$, along with 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. Architecture

This section will discuss in detail the (proof-of-concept) implementation aspects of the proposed architecture. As mentioned before, the framework supports different implementations for each low level service (*HostDiscovery*, *NetworkCommunication*, and *QoSMonitor*). Currently, an *UDPDiscovery* and *TCPCommunication* was developed

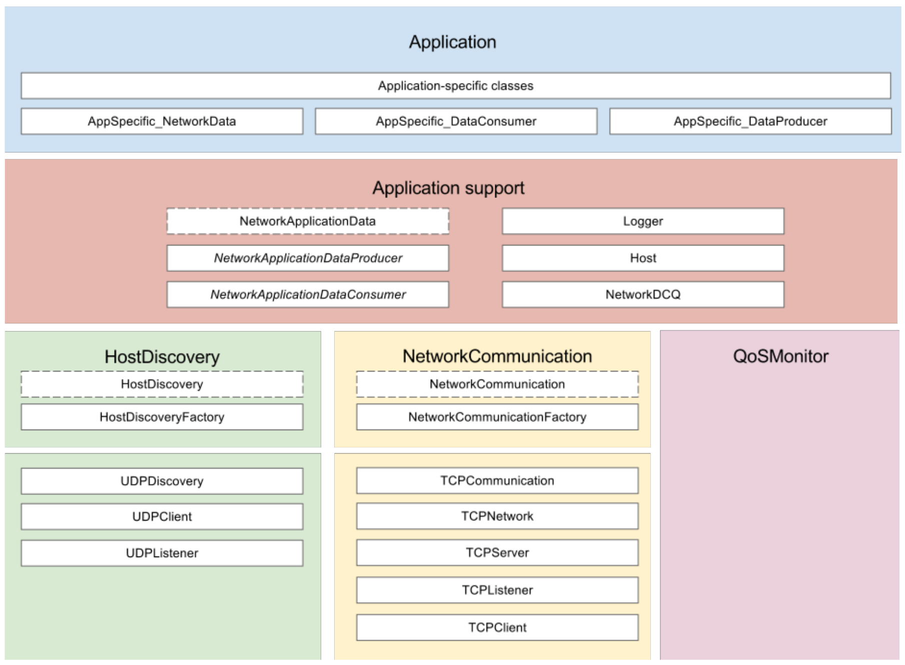

**Figure 2. Detailed Architecture of the Framework**

for *HostDiscovery* and *NetworkCommunication* services respectively, and *QoSMonitor* is under development. Fig. 2 shows the most relevant details on each layer, which will be explained in the following subsections (with the exception of *QoSMonitor*). In Fig. 2, abstract classes are identified with dotted lines, and interfaces are those in italic font. The current implementation of the NetworkDCQ project can be found at [NetworkDCQ Group 2012c] 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.

```
public static boolean configureStartup(
  NetworkApplicationDataConsumer consumer,
  NetworkApplicationDataProducer producer)
```

- Method *doStartup* is the one in charge of starting each service or feature (discovery, communication, broadcast), since they can be started independently.

```
public static boolean doStartup(boolean startHostDicovery,
  boolean startCommunicationService, boolean
  startNetworkBroadcast)
```

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 other hosts, being discovered, 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. 3. 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. *UDPDiscovery* has an additional responsibility,

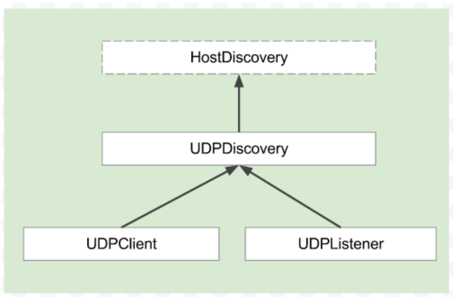

**Figure 3.** *UDPDiscovery* **Hierarchy**

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 defined in

```
HostDiscovery.DISCOVERY_TIMEOUT_LIMIT_MS
```

then the host is removed from HostDiscovery.otherHosts and

```
NetworkApplicationDataConsumer.byeHost();
```

is invoked. This validation is executed periodically, defined by

```
HostDiscovery.DISCOVERY_TIMEOUT_CHECK_INTERVAL_MS
```

## 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. 4:

- 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.

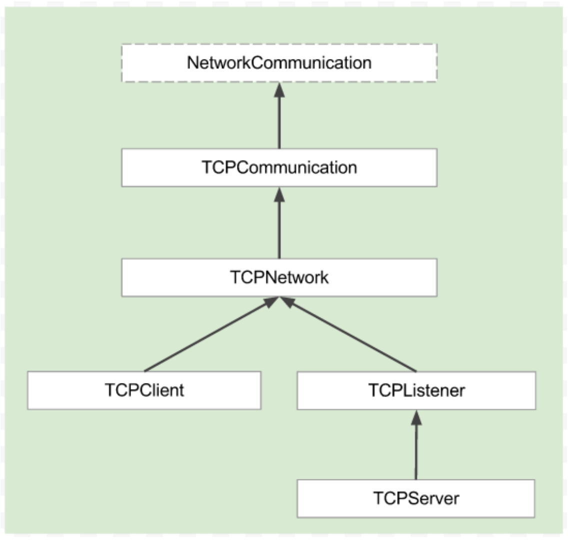

**Figure 4.** *TCPCommunication* **Hierarchy**

## 4. Examples

In this section two 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. In both cases, sample code will be given in order to highlight the most relevant details related to networking. The complete code of both examples, Asteroids and Tic-Tac-Toe, can be found at [NetworkDCQ Group 2012b] and [NetworkDCQ Group 2012e] respectively. Also, these projects are *completely* built on top of the the NetworkDCQ project [NetworkDCQ Group 2012c], i.e. there is no access to other Host Discovery and Communication services beyond those provided by the NetworkDCQ framework.

### 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. 5. 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 [NetworkDCQ Group 2012a], 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.

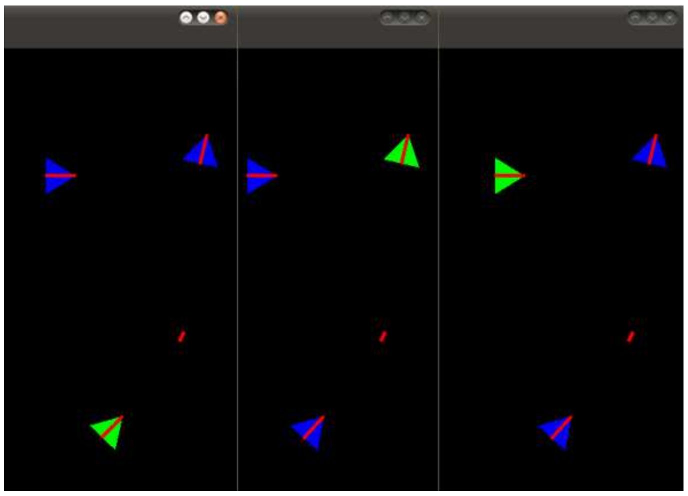

**Figure 5. Asteroids running on Three Android x86 v2.2 Virtual Machines**

### 4.1.1. Application Data Structure

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:

```
public class AsteroidsNetworkApplicationData extends NetworkApplicationData {
  protected Point2Df position       = new Point2Df(100, 100);
  protected float heading           = 0;
  protected boolean[] shotActive    = new boolean[StarShip.AMMO_COUNT];
  protected Point2Df[] shotPosition = new Point2Df[StarShip.AMMO_COUNT];
  protected float[] shotHeading     = new float[StarShip.AMMO_COUNT];
}
```

### 4.1.2. Producer

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:

```
public NetworkApplicationData produceNetworkApplicationData() {
  if (instance == null)
```

```
      instance = new AsteroidsNetworkApplicationData();
    instance.setData(HostDiscovery.thisHost, Globals.starShip);
    return instance;
  }
```

The *setData* method simply updates all the members of *AsteroidsNetworkApplicationData* with its current values according to the model changes.

### 4.1.3. Consumer

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.):

```
  public synchronized void newData(NetworkApplicationData receivedData) {
    remoteShipInfo = (AsteroidsNetworkApplicationData)receivedData;
    ...
  }
```

### 4.1.4. Configuration and Startup

The last step simply requires setting the corresponding instances of Producer and Consumer of the framework, and starting the services:

```
  NetworkDCQ.configureStartup(new AsteroidsNetworkConsumer(),
                              new AsteroidsNetworkProducer());
  NetworkDCQ.doStartup(true, true, true);
```

## 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, the game must control turns.
- There is no need for a continuous sending of information, specific events (a player marking a space in the board) trigger communications.

Fig. 6 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 Sam-

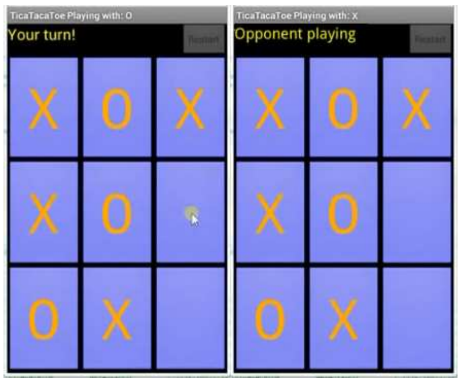

**Figure 6. Tic-Tac-Toe running on two Samsung Galaxy SII mobile devices with Android 4.0.3**

sung Galaxy can be found at [NetworkDCQ Group 2012d]. 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.

### 4.2.1. Application Data Structure

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. All of this information is represented with

```
public class Data extends NetworkApplicationData {
  public static final int ACTION_WHO_STARTS = 0;
  public static final int ACTION_SET_CELL   = 1;
  public static final int ACTION_RESTART    = 2;

  public int action   = -1;
  public int position = -1;
}
```

### 4.2.2. Producer and Consumer

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 Data type is needed in order to retrieve the members in the instance (action and position if needed):

```
public synchronized void newData(NetworkApplicationData receivedData) {
  Data data = (Data)receivedData;
  ...
}
```

### 4.2.3. Action Events

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. As an example, the following code is executed when the user clicks in one of the nine cells:

```
protected void sendButton(Button aButton, int number) {
  ...
  Data data = new Data();
  data.action = Data.ACTION_SET_CELL;
  data.position = number;
  NetworkDCQ.getCommunication().sendMessage(
     HostDiscovery.otherHosts.getValueList().get(0), data);
  ...
}
```

Notice how 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*.

### 4.2.4. Configuration and Startup

The final step is starting the required services. In this case, null will be passed to the framework as the Producer and the broadcast service will not be started:

```
    NetworkDCQ.configureStartup(new Consumer(this), null);
    NetworkDCQ.doStartup(true, true, false);
```

## 5. Conclusions and Further Work

The paper presented an integral solution for handling network-related issues in the development of applications running in mobile environments. The proposed framework was designed to support different implementations for each service, gaining flexibility, and versatility. As shown in the examples presented the previous section, the proposed API and reference implementation is actually useful for several types of applications, network requirements, and configurations. Using Android as a general development platform allowed an immediate integration with J2SE desktop applications. Some of the future lines of work include:

- As explained previously, the QoS service is still in development. Completing this feature is a short-term objective.
- Currently, a port to J2ME is being developed. The first aim is to achieve J2ME to J2ME communication, and then intercommunication with the Android platform and J2SE.
- Implementing the complete set of features of the framework for iOS, Windows Mobile, and BlackBerry 10 OSs are mid to long-term objectives.

## References

Asymco (2012). The Rise and Fall of Personal Computing. Can be found at http://www.asymco.com/2012/01/17/the-rise-and-fall-of-personal-computing/.

China Internet Network (2012). China Internet Development Statistics Report. China Internet Network Information Center, can be found at http://www.cnnic.cn/research/bgxz/tjbg/201207/P020120719489935146937.pdf.

Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection Pattern. Can be found at http://martinfowler.com/articles/injection.html.

Gartner, Inc. (2012). Smartphone Sales Increased 47 Percent. November 2012 Press Release. Can be found at http://www.gartner.com/it/page.jsp?id=2237315.

Google, Inc. (2012). Google Play Hits 25 Billion Downloads. Google Official Blog. Can be found at http://officialandroid.blogspot.com.ar/2012/09/google-play-hits-25-billion-downloads.html.

IDC (2012). Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter, November 2012 Press Release. Can be found at https://www.idc.com/getdoc.jsp?containerId=prUS23771812.

Martin, R. C. (1996). The Dependency Inversion Principle, Robert C. Martin, C++ Report. Can be found at http://www.objectmentor.com/resources/articles/dip.pdf.

Meeker, M. (2012). D10 Conference. Internet Trends. Kleiner Perkins Caufield Byers, can be found at http://www.kpcb.com/insights/2012-internet-trends.

Morgan Stanley (2009). *The Mobile Internet Report*. Morgan Stanley Research, 1st edition. A digital presentation with highlights of the report can be found at http://www.mobitechintl.com/pdf/MS_MI_Report_Setup_12.09.pdf.

NetworkDCQ Group (2012a). Asteroids for Android Example Video on YouTube. Can be found at http://www.youtube.com/watch?v=HiRTk8daqi4.

NetworkDCQ Group (2012b). Asteroids for Android Project Hosted in Google Code. Can be found at http://code.google.com/p/asteroidsa/.

NetworkDCQ Group (2012c). NetworkDCQ for Android Project Hosted in Google Code. Can be found at https://code.google.com/p/networkdcq/.

NetworkDCQ Group (2012d). Tic-Tac-Toe for Android example video on YouTube. http://www.youtube.com/watch?v=mrf01putSec.

NetworkDCQ Group (2012e). Tic-Tac-Toe for Android Project Hosted in Google Code. Can be found at http://code.google.com/p/ticatacatoe/.

Reuters (2012). Oracle sues Google over Android. Can be found at http://www.reuters.com/article/2010/08/13/us-google-oracle-android-lawsuit-idUKTRE67B5G720100813.

StatCounter Global Stats (2012). Mobile vs Desktop Internet Traffic Report from Oct 2011 to Oct 2012. Can be found at http://gs.statcounter.com/#mobile_vs_desktop-IN-monthly-201110-201210.