Faculty of Engineering and Sustainable Development

Mobile Multiplayer Game Interaction With XMPP

Olov Nord
June 2011

Bachelor Thesis, 15 credits, C
Computer Science

Creative Computer Graphics
Examiner: Jonas Boustedt
Supervisor: Anders Jackson
Mobile Multiplayer Game Interaction with XMPP

by

Olov Nord

Faculty of Engineering and Sustainable Development
University of Gävle

S-801 76 Gävle, Sweden

Email: ncg08ond@student.hig.se

Abstract

Smartphones are becoming a part of our society, and their usage is becoming more and more widespread. Multiplayer games for these smartphones are increasingly popular and the hardware that is used often puts limits on the developer in terms of battery life and transfer rate. This paper investigates the possibility of using the Publish/Subscribe Extension of the Extensible Messaging and Presence Protocol, or XMPP, as a data transfer protocol in a multiplayer game on Android. To succeed with this, a test application is programmed and used to send position data using PubSub. The report also discusses different ways of adapting the publish/subscribe node tree to the game map. The final conclusion is that it is possible to use XMPP for gaming purposes, but comparisons to other methods will have to be made for each specific case due to the workload of interpreting XML data and the amount of data the XML needs.

Keywords

XMPP, XML, PubSub, Android, Multiplayer Game, Network
Contents

1 Introduction ................................................................................................. 1  
  1.1 Research problem .................................................................................. 1  
  1.2 Research questions ................................................................................ 2  
2 Theoretical Background ............................................................................... 2  
  2.1 XMPP .................................................................................................... 2  
     2.1.1 PubSub .......................................................................................... 4  
     2.1.2 Previous uses for XMPP ................................................................. 5  
     2.1.3 Security .......................................................................................... 6  
  2.2 Android .................................................................................................. 6  
     2.2.1 Programming ................................................................................. 6  
  2.3 3G ......................................................................................................... 7  
3 Method ....................................................................................................... 7  
  3.1 Preparation ............................................................................................ 7  
     3.1.1 Network Model 1 .......................................................................... 7  
     3.1.2 Network Model 2 .......................................................................... 8  
     3.1.3 Network Model 3 .......................................................................... 9  
     3.1.4 Summary and Selection of Network Model .................................... 10  
  3.2 Equipment ............................................................................................. 10  
     3.2.1 Development Environment .......................................................... 11  
     3.2.2 Code library ............................................................................... 11  
     3.2.3 XMPP Server ............................................................................... 11  
     3.2.4 Testing Devices ........................................................................... 11  
  3.3 Code ....................................................................................................... 11  
  3.4 Data ........................................................................................................ 13  
  3.5 Testing ................................................................................................... 13  
     3.5.1 Two devices sending GPS coordinates ........................................ 13  
     3.5.2 Two devices sending screen coordinates ....................................... 14  
     3.5.3 Four devices sending screen coordinates ....................................... 14  
4 Results ........................................................................................................ 14  
  4.1 Comparison of data amounts in theory .................................................... 14  
5 Discussion and Conclusion ......................................................................... 16  
  5.1 Experience of the Used Software Packages .......................................... 16  
     5.1.1 ASmack ....................................................................................... 16  
     5.1.2 Openfire ...................................................................................... 16  
  5.2 Can XMPP/PubSub be used? ................................................................. 16  
  5.3 Other transfer methods? ....................................................................... 17  
     5.3.1 Bandwidth Latency ....................................................................... 17  
  5.4 How can XMPP/PubSub be used? ........................................................... 18  
  5.5 When can XMPP/PubSub be used? ......................................................... 18  
  5.6 Conclusions ......................................................................................... 18  
  5.7 Limitations ............................................................................................ 19  
6 Recommended Future Research ................................................................. 19  
References ..................................................................................................... 20  
Appendix A: Definitions ............................................................................... 22  
Appendix B: Complete source code for Java ................................................. 23
1 Introduction

1.1 Research problem

Creating an application for a Smartphone puts certain restraints on the programmer; he must consider obstacles such as battery life, lower processing power and memory than on a desktop computer, as well as the sometimes slow or unstable network connection. The network connection is a limitation for any mobile application that will stream considerable amounts of data, but the specific application focused by this thesis is multiplayer games.

In multiplayer games, large amounts of data can be sent to the different players, including, but not limited to: where the players are, what they do and what their avatar wear. Usually various security measures are also needed to verify that the user playing on the server is the rightful owner of the game character. If all this data is to be sent continuously to all players, it is likely that it will be too much for a mobile device to handle.

The aim of this thesis is to explore the possibility to implement the Extensible Messaging and Presence Protocol (XMPP) PubSub extension in an Android multiplayer ‘game’ application. XMPP is an internet standard for sending small pieces of XML, traditionally used for Instant Messaging (IM) and PubSub allows the creation of a node tree, to which the XMPP clients can subscribe. Any information sent to (published to) a node is passed down to all clients subscribed to that node. It is possible that this structure and functionality will work as a good base on which to construct the game networks to avoid the aforementioned problems. By mapping a game map to the PubSub node tree (Fig. 1.1). It is also assumed that the amount of data can be kept quite low, as the users can subscribe to the nodes that represent the game-area where they currently are. As PubSub builds on XMPP it will give all the functionality of XMPP to the game using it as a transfer protocol, solving many of the mentioned issues. The extensibility of XMPP also ensures that if additions need to be made to the protocol, this will be possible without having to recreate the entire system.

![Fig. 1.1: Mapping the game-map to the PubSub node-tree. This is essentially what will be explored in this project.](image)

If not deemed impossible or too inefficient, this project also aims to provide a working and adaptable source code for a working PubSub (game) application.
1.2 Research questions

The specific questions to be answered are:

- Can XMPP, specifically PubSub, be used to allow a game with a large number of players on a mobile device (Android in this case)?

- How can the game area be mapped onto the XMPP node tree to achieve a low amount of sent data (as to conserve battery power and the sometimes poor bandwidth)?

2 Theoretical Background

2.1 XMPP

XMPP, or ‘Extensible Messaging and Presence Protocol’ is an open standard just like HTTP, in this case meaning that it is not bound to a specific company or product. It was invented in 1998 by Jeremie Miller under the name ‘jabberd’ and was supposed to be an open source instant messaging (IM) server. Later in 1999-2000 the core of the XMPP was more or less completed and was accepted as an internet standard protocol by the IETF [1]. The development however, does not stop, and jabberd/XMPP keeps on growing with various extensions, known as XEPs.

XMPP networks build on interconnected servers, to which clients can connect. It is important to notice that the clients connect through the servers, allowing security measures to be taken by the servers. Fig. 2.1 shows a representation of two connected servers, each with three connected clients.

![Diagram of XMPP network](image)

*Fig. 2.1: A typical layout of the XMPP network. Several servers work together and the clients have many access points.*

The clients can register to these servers with specific user ids called JabberIDs (or ‘JID’ for short) and a password. The JID contains two parts, the username portion and the domain part, separated by a ‘@’, so that a user registered with the username ‘user’ at jabber.org will have the JID ‘user@jabber.org’. A JID on this form is known as a ‘bare JID’. The JID is also often complemented with a ‘resource’ that gives more specific information about the users’ current whereabouts. It is added to the end of the JID: user@jabber.org/resource. When the JID is complemented with the resource, it is
known as a ‘full JID’. The resource allows the user to be logged in from several places at the same time. By specifying resources such as /home and /work a user can be online both at home and at work simultaneously. Any messages sent to user@jabber.org will be delivered to any resources that might be related to this particular user, in this case both /work and /home. A message can also be directed to a specific resource; if sent to user@jabber.org/home it will only reach this resource, it will not be delivered to user@jabber.org/work [2, pp. 11-28].

XMPP streams XML code over a long-lived TCP connection. The XML code is sent in the form of different ‘stanzas’, that all have their own ‘type’-attributes. These stanzas can be of the types “message”, “IQ” (Info/Query), or “presence”.

Each of the stanzas have different purposes in the communication; message, or <message/>, is the basic transmission stanza, used for sending messages from one client to another. It can have types such as ‘normal’ that indicates simple text, ‘chat’ and ‘groupchat’ indicating a message being sent in a real-time between two clients or between many users in a multi user chat session, respectively, or ‘headline’ for messages to which no response is expected. There is also an ‘error’ that is returned if something went wrong with a previous message [2, pp. 11-28], [3]. An example of a message-stanza is shown in Code listing 1.

```xml
<message type="chat" to="client2@server.com" from="client1@server.com/resource">
  <body>hello client 2! </body>
</message>
```

*Code listing 1: An example of a “message” stanza.*

The presence (<presence/> ) stanza is used to send a presence information from a user to any users allowed to access this information. It contains a single message with information such as ‘online’, ‘offline’, ‘away’ or any other custom message defined by the client (Code listing 2). In order to receive presence from a client, one must first subscribe to the client by sending a request. When the client updates his presence or ‘status’, the new information is sent to all other clients who have had their subscription requests approved [2, pp. 11-28], [3].

```xml
<presence from="client1@server.com/resource">
  <show>chat</show>
  <status>I want to chat!</status>
</presence>
```

*Code listing 2: An example of a presence stanza, declaring a user available for chatting.*

The final stanza, Info/Query (<iq/> ) allows the client to request information, or to set information from different parts of the network. Instead of simply sending information such as the other two stanzas, the IQ stanza is guaranteed to get a response. It uses the following type-attributes to decide what action is being carried out: ‘get’ is used for requesting information, ‘set’ which is used for providing information, ‘result’ which is returned if the ‘set’ operation was successful or with the requested information from a ‘get’ request. IQ does also include an ‘error’ type that is returned when the ‘get’ or ‘set’ did not succeed [2, pp. 11-28]. See Code listing 3a and 3b for examples of the IQ stanza.
<iq type="get"
    to="client1@server.com"
    from="client1@server.com/resource"
    id="1001">
  <query xmlns="jabber:iq:roster"/>
</iq>

Code listing 3a: An example of an IQ stanza requesting client1’s roster, or friend list.

<iq type="result"
    to="client1@server.com/resource"
    from="client1@server.com"
    id="1001">
  <query xmlns="jabber:iq:roster">
    <item jid="client2@server.com"/>
    <item jid="client3@server.com"/>
  </query>
</iq>

Code listing 3b: The possible result of the previous example requesting client1’s roster. Note how the message id is identical to the sent message.

The ‘roster’ is sometimes called a ‘contact list’. It contains the information and presence of the clients that a user has approved presence subscriptions with. It is basically a list notifying which users are allowed to access a client’s information (a list containing approved users such as this are also sometimes referred to as a ‘whitelist’). The users in a roster can be grouped accordingly to the roster-owners preferences. To subscribe to a client, a presence stanza is sent containing the type attribute ‘subscribe’, which is a request that needs to be approved by the client and answered by a presence stanza containing either the type ‘subscribed’ if approved, or ‘unsubscribed’ if denied [2, pp. 31-44].

During the whole active session, the TCP stream is open, sending and receiving these XML stanzas between the server and client. Only when the session ends, the TCP connection drops.

2.1.1 PubSub

Pubsub or Publish/Subscribe is an extension to XMPP defined in XEP-0060 [4]. The main idea is that instead of having a user ask the server for new information, the server sends out the information when it has been updated.

The PubSub functionality builds on a node tree on the server; each node can either be a node containing other nodes or a leaf node at the end of a branch. Users can then subscribe to information from these nodes. The users can also publish to the nodes; both to the collection nodes and the leaf nodes. Any information published to a node is sent on to the subscribers and leaves of that node [2, pp. 95-122]. For an example of such a node tree, see Fig. 2a in section 3.

Depending on the server and node settings, these nodes support authorization of the users or groups from a specified roster or whitelist to tell whether the user is allowed to subscribe or publish to the node.

Most PubSub operations such as publishing, subscribing, unsubscribing, changing node settings etc. are done by sending IQ stanzas, containing a <pubsub/> element that hold varying content depending on what action is taken. If an object is published to a node, the server sends out a message stanza to each of the subscribers containing the published information (Code listing 4a and 4b) [2, pp. 95-122].
<iq type="set" to="pubsub.server.com" from="client1@server.com" id="1002">
<pubsub xmlns="http://jabber.org/protocol/pubsub">
<publish node="nodename">
<item>
<entry xmlns="http://www.w3.org/2005/Atom">
<title>This is a message title</title>
<summary>This is a summary of what is included in our message.</summary>
<link rel="alternate" type="text/html" href="http://server.org/anewmessage"/>
<id>tag:server.com,2011:entry-101</id>
<published>2011-06-01T10:00:02Z</published>
</entry>
</item>
</publish>
</pubsub>
</iq>

Code listing 4a: Example IQ stanza for publishing a sample ‘Atom’-notification, the PubSub element contains a publish tag, as that is what we are doing. If the client instead wanted to subscribe to the node he should use the subscribe-tag. As this is an IQ stanza, the server must reply to it, in this case with a result type IQ if the publication was successful, or an error type IQ if the publication failed.

<message to="client2@server.com" from="pubsub.server.com">
<body>This is a message title: This is a summary of what is included in our message.</body>
<event xmlns="http://jabber.org/protocol/pubsub#event">
<items node="nodename" id="a8f54e">
<item id="b56e4">
<entry xmlns="http://www.w3.org/2005/Atom">
<title>This is a message title</title>
<summary>This is a summary of what is included in our message.</summary>
<link rel="alternate" type="text/html" href="http://server.org/anewmessage"/>
<id>tag:server.com,2011:entry-101</id>
<published>2011-06-01T10:00:02Z</published>
</entry>
</item>
</items>
</event>
</message>

Code listing 4b: The message a client subscribed to the node ‘nodename’ would receive from the previous example publication. Note how the message is addressed from the server and not the publishing user.

2.1.2 Previous uses for XMPP

During its long history, XMPP usage has spread beyond IM. With a large number of extensions made allowing different uses, XMPP has been implemented for applications ranging from voice chat [2, pp. 123-136] to alarm systems for apartments [5].
Relating to the project at hand, there have also been previous multiplayer games built on XMPP [6][7], but these games are only for a small limited number of players in a very limited game world (both allow two players, performing actions on the area visible on the screen). They also seem to build on the IM parts of the protocol, with the chatting being a large part of the game. This differentiates from this project, aiming to create a model supporting a larger world with a great number of players using the PubSub function as an underlying layer for data transfer.

2.1.3 Security

It has been mentioned that XMPP allows security measures. In fact, XMPP supports both SASL authentication and TLS encryption. When starting a new XML stream, depending on the server settings, the client can specify if it wants to encrypt the stream using TLS. After that the client must authenticate itself by sending its password and username over SASL. The server can naturally be set so that the different security measures are not only optional, but also required to establish a stream [1], [2, pp. 165-196].

2.2 Android

Android is an operative system (OS) owned by Google Inc. used for mobile phones (smartphones) and tablets. The devices on which Android are run usually supports many hardware features such as WiFi, 3G, GPRS, GPS, touchscreens, accelerometers, etc.

Vital parts of the Android system are the applications commonly referred to as ‘apps’. These are (usually smaller) programs that can be run under the OS.

2.2.1 Programming

Developing an application for Android is commonly done using Java, with specific libraries from Google to get control of the phone’s functions. The Java code is then not compiled to Java bytecode, but to a special instruction set for Androids own Dalvik virtual machine.

The applications commonly contain the following parts [8]:

- An ‘Activity’ class that handles the creation of the window, as well as what views it will contain. It can also handle events such as pausing or resuming the application or pressing buttons. The activity handles the creation of ‘View’ objects that handles what is drawn on the screen. There are specialized views for drawing special things such as text or images, but the programmer can also extend them to create custom views.

- A ‘Service’ is the part of the application which can handle long lasting processes. It does not have a user interface, but simply runs in the background. They are used for actions such as playing background music when another application is active.

- A ‘Content provider’ that can read and write data on the phone storage. Using a content provider, different applications can access files from the phone if permitted.

- A ‘Broadcast receiver’ that receives notifications from the system about major events, such as the screen turning off or the device running low on battery.

- An ‘Android manifest’ in XML, with information to the device about what activities are included, what permissions these applications need (such as internet usage) and other general information about the application.
2.3 3G

3G is the third generation networks for mobile data transmission. It has been mentioned in the specifications that the minimum transmission rate for a stationary or walking user is 2 Mbit/s [9].

In mobile network services such as 3G there is apart from the transmission rate also a delay, or latency that influences the experienced performance. This latency depends highly on the service provider, and the connection quality.

3 Method

3.1 Preparation

Before any of the actual code is written, three different approaches to the network layout are sketched up. These models are here introduced in their own sub-sections.

3.1.1 Network Model 1

The first and most straightforward approach can be seen in Fig. 3.1a. The ‘Areas’ refer to PubSub nodes for larger game areas, such as a whole game level, where there can be no interaction between users in different areas. In Fig. 3.2b an example game map of a city, a field outside the city and a small cave has been marked to illustrate how this model could look.

This approach is simple, but might cause some problems considering the amount of network traffic. If a level is very large, the node associated with it will possibly have a large amount of players publishing their information to it, as well as having to receive information about all other players, even if these players are too far away on the game map for the information to be of any use.

Fig. 3.1a: Network layout with a PubSub node for each game area (level). The users are all subscribed to their current level’s node, and publish their own information to it.
Fig. 3.1b: A fake game map depicting a possible way to divide the game into areas which can be mapped to the different nodes. In this example it is assumed that the city, the field and the cave are treated as different ‘levels’, meaning that a player cannot exist in one of these areas and see into the other. The only way to reach another area is to load new level data.

3.1.2 Network Model 2

The second approach tries to minimize the data size by introducing smaller regions, ‘sub areas’ (Fig. 3.2a, Fig. 3.2b). Each of these smaller areas cover only a small predefined part of the whole level each, meaning fewer players publishing to the nodes and thus lowering the strain on the users’ network connections. The player would have to subscribe to at least one of these sub area nodes, but any time he is closer to the edge of the area representing a node, he or she would also have to subscribe to the neighbouring node as well to get all the surrounding data for a seamless experience.

Fig. 3.2a: Sub areas are introduced. These smaller areas each cover a small part of the main area.
Fig 3.2b: The larger areas have now been divided into smaller sub-areas. This means that the user in the house that is sub area 2 will not receive information from the users on the street outside. On the field outside it might be necessary to either make larger areas, or to subscribe to information from more than one area, as one should be able to see what happens in the surrounding areas.

3.1.3 Network Model 3

The third approach, shown in Fig. 3.3a and Fig. 3.3b, is more difficult to implement, but might give the best performance in terms of data traffic. It uses more dynamic nodes, where each player has a node for himself that he publishes to, and is subscribed to all players’ nodes that are in his vicinity. The problems arise when the users move. A user has to be notified that someone has moved close to him and that he should subscribe to this new node.

Fig. 3.3a: Each user has its own node to publish to, and is subscribed to neighbouring users’ node.
Fig. 3.3b: Each user has a node that other users in range can subscribe to. In this example user 1 is subscribed to user 2 and vice versa. User 3 is farther away and does not receive any data at all.

3.1.4 Summary and Selection of Network Model

Summarizing these methods, the second is merely an extension to the first, providing more nodes, and thus fewer messages in each node. However, it does come with the cost that users sometimes have to subscribe to several nodes at once, and the subscriptions have to change over time. The topology of the third tree differs from the first two in that the users subscribe to all nodes owned by the users surrounding them, and can only publish to the node they own. This method could be extended by combining it with the second model, so that every user sends infrequent updates to a collection node for the area in which they are. All users in this node would receive this update, and subscribe to the users private node if they are close enough. In the private node, updates are published much more frequent and with more detailed information than just position, such as player actions (walking, standing, sitting etc).

As the timeframe for this project is quite narrow, only one of these approaches can be tested properly. The first model is the base for all of the three discussed methods, with the users subscribing to a single node per level. In order for any of the other methods to work, this is the approach that must first be implemented and working. Thus it is the chosen network model for this project.

3.2 Equipment

In order to start working, a number of tools and programs are necessary. This section will provide insight in these tools and how to acquire them. It is important to know that the final result is made for an Android-device and therefore will be programmed in java. Many of the choices are made because of these presumptions.

What follows is a list of equipment used for this project. Each of these points has been given their own subsection below.

- A software development environment.
- A code library for XMPP.
- An XMPP server.
- Testing devices.
3.2.1 Development Environment

To program efficiently, an IDE (integrated development environment) is used, in this case Eclipse. Eclipse was chosen as it supports the official Android development tools (ADT) [10].

The ADT consists of a number of tools helping with the development of Android applications. It has functions for creating emulated devices on a computer, as well as code libraries needed to access the system and carrying out device operations. ¹

3.2.2 Code library

As all code will not be written entirely from scratch, the XMPP client code-library ‘aSmack’ will be used, which is an android-adaption of the ‘Smack’ library. ²

3.2.3 XMPP Server

In order to send any data there needs to be a server to handle all the data traffic. For this, Openfire (Version 3.7.0) was selected as it is made by the same developers as Smack, and therefore work well together with it. ³

The server was run with standard settings, except it was set to listen to 0.0.0.0 instead of 127.0.0.1.

3.2.4 Testing Devices

The following devices have been used to test the code:

1. Android 2.2 on emulator on a desktop computer.
2. Android 2.2 on HTC desire
3. Android 2.4 on Google Nexus One
4. Android 3.0 on Nook Color
5. Android 2.2 on Samsung Galaxy S

3.3 Code

This section provides the pseudo-code necessary for recreating the test application used to prove the possibility of using PubSub as the main data transfer method in a game. The completed application can listen for changes in the player’s location, and publish any updates to a PubSub node as a custom XML payload containing the player’s username and the new position as x and y coordinates.

As Android is the target platform, there are two code parts; one for the activity that handles the connection and one for the view that draws on the screen. Both the activity and the view inherit a number of methods. These are not described in the following pseudo-code (Code listing 5), as they are not essential for the general functionality of this program.

---

¹ Eclipse can be found at http://www.eclipse.org [26 may 2011]
   The ADT can be found at http://developer.android.com/sdk/eclipse-adt.html [31 may 2011]

² Smack can be found at http://www.igniterealtime.org/projects/smack/ [26 may 2011]
   aSmack at http://code.google.com/p/asmack/ [26 may 2011]

³ Openfire can be found at http://www.igniterealtime.org/projects/openfire/ [26 may 2011]
Activity:
When the activity is created:
Function onCreate():

    Set the VIEW to be used
    Create a USER and PASSWORD
    Get a ProviderManager
    Add Providers to the ProviderManager
    Create a ConnectionConfiguration specifying a SERVER
    Connect to the SERVER using the ConnectionConfiguration
    Login on the SERVER using the USER and PASSWORD;
    Set your presence;
    Create a PubSubManager for the connection;
    BEGIN
        Receive the wanted NODE
        EXCEPTION
        WHEN NODE not found
            Create a ConfigureForm
            Add node settings to the ConfigureForm
            Create the NODE using the ConfigureForm
        END
    Add a NodeListener to the NODE
    Subscribe to the NODE
    Create a GameLocationListener for the location;
When the activity is destroyed:
Function onDestroy():

    Create a new payload with coordinates as 0
    Publish the payload to NODE
    Unsubscribe from NODE
    Remove the GameLocationListener;

    nodelistener implements ItemEventListener
    Function handlePublishedItems (ITEMS):

        DATA = ITEMS to payload to xml
        Parse xml for Sender, x, y
        IF x AND y is 0 THEN
            Remove id from MAP
        ELSE
            Put Sender, x, y in MAP

    GameLocationListener
    Function onLocationChanged (LOCATION):

        Get coordinate data
        Create a new payload with coordinates and the current user id
        Publish the payload to the NODE;
View:
When the view is created:
Constructor:
    Create a new SurfaceThread to do the drawing
    Get the MAP from the main Activity
SurfaceThread
Function DoDraw(CANVAS):
    Get an ITERATOR for all the Keys in MAP
    Clear the screen
    WHILE ITERATOR has next
        Get the next Key from ITERATOR
        Get the position corresponding to the Key
        Recalculate the position relative to the user and
        scale it to the screen
        Draw a Point at the recalculated position
    ENDWHILE

Code listing 5: The entire pseudo-code for the working test application.

3.4 Data
The data that is sent between the devices can be different depending on the application’s needs. For this project, two different kinds of position data have been studied, both using the ‘geoloc’ extension of XMPP [11]. The first datatype used was simple latitude and longitude coordinates received using the device’s built in GPS. This data was then mapped onto the screen to display points representing the users.

The second kind of data was screen coordinates, received when any of the users dragged a finger over the device’s touch-screen. The data was then sent and displayed on all the users’ screens in the location where the sending user had been touching his/her screen. For this, no special transformation of the coordinates was needed as both screens were to show the same pattern of datapoints.

The reason for working with positions (or coordinates) is that this is one of the basic attributes needed for a good multiplayer game. A game may work fine without actions such as sitting or standing, or opening doors, but it is likely that the players will want to know the locations of the other players in most kinds of multiplayer games.

3.5 Testing
Testing the data transfer has been done in three different stages, each given a sub section in the following. The network model used was the first one described in the preparation section as this is the base for both of the others.

3.5.1 Two devices sending GPS coordinates
The first test was done with two devices (HTC desire and Nexus One) receiving GPS coordinates that then was published to a node. The GPS listener was set to receive new data at least every 2 seconds, or when the user had moved more than a meter. Upon receiving these coordinates from the server, they were mapped to the device screen so that the tester’s (player’s) coordinates were always centred.

For this test two stationary ‘dummy’ points were used. These points were not transferred over the network, but predefined in the program to simply help the player compare himself to a stationary point.
3.5.2 Two devices sending screen coordinates

The second test was simpler than the first, in that all testers saw the same things on their screens. When a player touched his/her screen, the screen coordinates were sent to the node. Upon receiving published coordinates from the server call-back a dot was drawn where the screen was previously touched by the publishing user.

3.5.3 Four devices sending screen coordinates

The last test did not significantly differ in any aspect from the second except that it involved four devices instead of two. In both of these ‘painting’-tests the drawn dots were colour coded depending on what user ID was included in the received payload.

4 Results

This section covers the results from the tests described in the methods section.

In all the tests the data transmission worked well. However, it was noticeable that even with as few as two or three users the program became slow when sending data more frequently.

In the tests sending GPS coordinates, a certain shakiness of the points were noticed by the users, as the coordinate accuracy seemed quite low. This was especially noticeable as the dummy points with fixed coordinates were moving too, in relation to the player (meaning that the player coordinates was changing). This could be fixed by introducing an average (possibly weighted) of the recent positions. It was also very hard to discern whether the positions were correct or not, as the program did not provide any sense of direction, only relative distances.

The final tests using screen coordinates worked as expected, replicating the pressed points on all the screens. It was in this test that the transmission became quite slow, as experienced by the users, due to the large amounts of data sent when dragging a finger across the screen, as this method continuously sends data when the screen is touched, compared to the GPS method, that only sends data every 2 seconds.

It was also noticeable that the drawn points were not always the same. It is likely that this depended on the drawing method rather than data loss in the transfer. As the drawing was made continuously from another (low priority) thread, it is possible that this thread did not manage to draw all existing points before they had been replaced by new incoming data.

4.1 Comparison of data amounts in theory

Looking at the first tests sending GPS coordinates; one can calculate the minimum amount of data being transferred per second. Assuming that the sent message includes a custom payload with three tags, the complete (sent) message might look like the following example (Code listing 6):
<iq id="messageID" to="pubsub.0.0.0" type="set">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="pubsub/someNode">
      <item>
        <namespace xmlns="customnamespacehere">
          <x>0.85432</x>
          <y>0.453212</y>
          <player>playerName</player>
        </namespace>
      </item>
    </publish>
  </pubsub>
</iq>

Code listing 6: An example PubSub message that contains some player information in a game.

This sum up to 276 characters, including whitespace. If the XML is encoded using UTF-8, this allows a minimum of one byte per character [12], meaning that the total size must be more than or equal to 2208 bits (this number will of course vary depending on what data is included in the message).

Assuming that the user is connected over a 3G-connection (2Mbit/sec), as well as a similar size of the received messages as the sent ones, almost 900 messages can be sent each second if the network could be used to its full extent and no other factors limited the performance.

As new data was received from the GPS every other second, and then immediately sent to the server, this gives an estimated average of 1104 bits/second (sent per user), allowing a theoretical maximum of 1800 users per node.

There is no data that can show how often the screen-coordinates were sampled in the second and third test, but assuming 15 samples per second gives a data rate of slightly above 33kbit/second, and a definite maximum of 60 users per node.

Calculating the possible cost of sending this data as a UDP packet containing three float values (x, y and a unique user id) instead of XML over XMPP gives the following result: each float value is made up of 32 bits [13], giving a total of 96 bits per message, which is much lower than the 2208 bits of the XML code. One will however have to take into consideration that with XMPP the user only sends one message to the server and receives one message for each user. Using the UDP packets, one can either have a specialized server working much like XMPP to lower the amounts of messages, or use a method where each user sends messages to every other user without the need of a server.

With the PubSub/XMPP style network, assuming that users receive their own messages, the total amount of messages can be calculated by the following equation:

\[ M = U + 1 \]  \hspace{1cm} (1)

Where M is the total number of messages a user has sent or received after each user has sent one message and U is the number of users. This is because the user receives one message from each user (including himself) and sends one message.
The amount of messages handled with the UDP method however is calculated differently:

\[ M = 2^*(U-1) \]  

(2)

As each user will have to send packets to every other user except himself and also receive packets from every other user except himself.

5 Discussion and Conclusion

5.1 Experience of the Used Software Packages

5.1.1 ASmack

Smack is a very good library for XMPP services; it does however not work perfectly with Android. This is why aSmack has been independently developed, providing patches and fixes for Smack to work with Android. For the most part this works great, but there is very little documentation for aSmack, and because of the patches made, the Smack documentation is not always exact. One example is that Smack can extract XML data into a Java bean, but this does not work in aSmack as the ‘JavaBeans’ package is missing a lot of classes in Android.

One can also note that even though the library comes with a built in debugging tool (which works with Androids ‘logcat’), it does not work perfectly. Often, no information shows up in the log at all, and when it did, longer log entries were regularly cut off midway.

There is also very little access to the actual XML code being transferred, as it is only possible to see the Smack commands and classes. The XML is sent on a deeper layer, out of user control.

5.1.2 Openfire

Using the Openfire server software worked great. It was quick and easy to set up, and the more freely built node system was perfect for this project. However, certain aspects could definitely need improvements.

For a project such as this, more debugging options had been a great addition to the server. It was not possible to get insights into what was really sent through the server was nonexistent (combined with the fact that almost no XML is viewable from the client side in Smack either was really troublesome at times). An XML console displaying sent messages and allowing the server to send messages would have been a great addition.

Openfire does support PubSub, but the server status/settings interface displayed nothing about this. For future projects one should definitely look into possibilities of a server that can display more detailed debugging information in this field, possibly a graphical display of the current node tree or a quick way to see which users are subscribed to which nodes from the server?

5.2 Can XMPP/PubSub be used?

The results show that XMPP and PubSub is an option for sending data in games. However, during the early stages of testing another server software was used (ejabberd, http://www.ejabberd.im/ [26 may 2011]) with which a successful connection never was established. This was most likely due to a faulty node setup, as the node hierarchy has specific requirements in ejabberd [14], and these were later found not to have been followed promptly, possibly generating errors.
Building your application network on XMPP gives you all the features that come with XMPP, such as scalability and security [2, pp. xi-xviii]. As there is no restraint on what kind of data you send in the XML, it is also very extendable, allowing you to add new data without having to remake a large portion of the code. It is in addition possible to use the chat-functions to send text messages between players (there are specific chat functions that come with the Smack/aSmack libraries [15]).

The method of sending real-time data with XMPP comes with a few problems as well, the most notable being that all data is sent as XML pieces. While stable and easy to decide what data to send by including or excluding different XML tags, this means that all data will have to be parsed by an XML parser when received. Unfortunately this can be slower than needed, especially with large amounts of received data from many players.

5.3 Other transfer methods?

Looking at the subjects mentioned in section 4.1 one will also have to consider the usage of other methods of data transfer, especially when the amount of active users will be very low. One can extend equations 1 and 2 with a constant representing the amount of bits needed for one message:

\[ D_1 = B \times M = B \times (U+1) \]  

Here \( D \) is the total amount of data handled (received and sent) by the device and \( B \) is a constant representing the data amount each message contains. This equation is for the server model where each user sends one message and receives one message for each user.

The same addition can be made for equation 2:

\[ D_2 = B \times 2 \times (U - 1) \]  

And then by comparing \( D_1 \) and \( D_2 \) it is possible to get an estimate of the most effective transmission model (data wise).

If using a custom transfer method built on UDP packets (or TCP), it is also important to realize that many of the built in functions of XMPP are not included and the programmer will have to add support for all needed features himself. This will increase the data amount (from identification, authorization, encryption etc.), as well as increase the work amount needed to be carried out for the programmer.

5.3.1 Bandwidth Latency

It is important to understand that the bits/second is only a measurement of how much data can be transferred, and not how fast the transfer is. In wireless networks there is often a delay, or latency, as briefly mentioned earlier in this paper. This latency depends much on the network provider, but also on the phone hardware and signal strength. It is theoretically possible for a phone with low latency to send large amounts of data faster than a phone sending small amounts of data with high latency. Consequently, even if a phone has a high bandwidth the client can still experience choppiness or ‘lag’ if the network latency is too high. Thus it is important to calculate an expected latency as well as the packet size to get a good estimate of the transfer speed and program performance.
5.4 How can XMPP/PubSub be used?

The usage of PubSub for the data transfer seems to be a versatile and stable approach, allowing a large number of players as long as appropriate optimizations to the node-tree are made to keep the amount of sent data to a minimum. The main bottleneck appears to be the processing of the received messages. Depending on how the receiving parser is built, one will either have to keep the general data size or the amount of messages low. If the parser has to iterate over every character in the received XML code, the amount of characters (data) will have to be kept to a minimum. If instead the parser is build more efficient and the length of the XML is less relevant, it is enough to keep the amount of messages received small to assure performance and speed. This is consistent with the conclusion of J Kangasharju et al. [16] who states that “First, it is imperative to decrease the sizes of messages, since wireless communication is very costly, both in terms of power and (currently) money. Second, in some scenarios XML processing can be the bottleneck of the application.”

5.5 When can XMPP/PubSub be used?

Drawing conclusions from this, one can safely say that XMPP is an option for sending different kinds of data in games. Especially in the cases where one expects a large number of players but not overly frequent transmissions. A fast paced game publishing position coordinates 30 times/second might not get the performance required for a good game experience compared to a slower text-based adventure game, sending user actions maybe once every other second (which might not be a feasible goal anyways, due to the expected latency of the mobile network). This is because of the aforementioned parsing of the XML code. When choosing whether to use XMPP or not, the programmer will carefully have to weigh the different factors such as how often and much data will be sent, how efficiently can this data be extracted (parsed) and how many players will this data be sent to, as well as other surrounding factors depending on the game application.

5.6 Conclusions

Relating to the first question this research was meant to answer;

- Can XMPP, specifically PubSub, be used to allow a game with a large number of players on a mobile device (Android in this case)?

The answer is yes, as the test application has successfully transferred XML data concerning possible game attributes between up to four users, and there does not seem to be anything preventing more users from joining, apart from a possible limit on the data/message amounts.

As for the second question;

- How can the game area be mapped onto the XMPP node tree to achieve a low amount of sent data (as to conserve battery power and the sometimes poor bandwidth)?

Three different models have been discussed in section 2.1, and the most basic of these have been used for the test application.
5.7 Limitations

Because of the relatively short timespan, certain limitations have been put on this paper. Especially as a large part of the time has been spent learning about XMPP, smack, and how to get it to work on Android. With more time available, one could have created a more advanced and game-like testing application, allowing the collection of quantitative data. It could also have been a possibly to test the application with more users, and all the different node layouts discussed in the method chapter. The author has chosen not to extend the project, as better results might come to if specialized research projects building on top of the code and knowledge gained from this project. As of now all follow-up have been recommended as future research fields in the next section.

6 Recommended Future Research

Working with this project has both given many answers as well as questions. Several new and interesting research subjects have become available, including the following recommendations:

- Comparison of efficiency between XMPP and other data transfer methods in a multiplayer game environment.

- Implementing and trying out different game to tree mappings as discussed in the method section of this paper, to optimize the data flow.

- Efficient interpreting of XML streams for use in a real-time environment.

- Implementing custom XMPP extension and for usage in a multiplayer game and a provider for Smack to handle the extension.
References


[6] Tankaar


documentation/messaging.html (visited 31-05-2011)

[16] J. Kangasharju et al., ”XML messaging for mobile devices: From requirements to
   Available: http://dx.doi.org/10.1016/j.comnet.2007.06.008
Appendix A: Definitions

Android: An operating system for mobile devices such as smartphones and tablets.

Application: In this study, application refers to a smaller program developed for the Android system.

Client: The computer or device connecting to or sending/receiving data through a server over a network. In this paper, the word ‘user’ is also frequently used, meaning a person using a client to communicate over a server.

IM – Instant Messaging: Near real-time text communication between two computers, done by using clients to send data over a network, usually through a server. Classically used for online chatting.

Parsing: Analyzing a text in such a matter that the program can understand and extract certain elements. In this report the parsing mentioned refers to analyzing XML code to give access to the tags and values of these tags according to the aforementioned XML standard.

Player or Tester: The user (see client) trying out the test application developed for this project.

PubSub – Publish / Subscribe: A way of sending or receiving data through XMPP that builds on nodes, which a user can subscribe to, and then receive any information that others might publish to it.

Server: The computer or device that handles the data sent to or received from the connecting clients.

XML – Extensible Markup Language: A set of rules describing the data that can be stored in an XML document, and how to interpret this data. http://www.w3.org/TR/REC-xml/ [31 may 2011]

XMPP – Extensible Messaging and Presence Protocol: An internet protocol for sending XML data over a network at near real-time. It was originally designed for IM, but has over the years been adapted for many different kinds of applications. http://xmpp.org/about-xmpp/ [31 may 2011]
Appendix B: Complete source code for Java

The complete and working source code is attached on DiVA (http://diva-portal.org) as a zip-archive (SourceCode.zip) containing the following:

- AndroidManifest.xml
- login.xml and main.xml (these are the layout xml files for the application)
- ItemProviderPatched.java (a fix for the aSmack ItemProvider class)
- SmackSurfaceView.java (this is the custom view class)
- SmackTesting.java (containing the main activity as well as the custom NodeListener and GameLocationListener classes)
- StartupActivity (a simple activity allowing the user to select one of four possible users)