Mobile Multiplayer Real-time Games over Wireless Networks
Alf Inge Wang, Martin Jarret, and Eivind Sorteberg
Dept. of Computer and Information Science
Norwegian University of Science and Technology
alfw/jarret/sorteberg@idi.ntnu.no

ABSTRACT
This paper describes results and experiences from designing, implementing and testing a mobile multiplayer real-time game for mobile phones over mobile networks. The paper reports on network latency and bandwidth measurements from playing the game live over GPRS, EDGE, UMTS and WLAN using both the TCP and the UDP protocol. The results show that the choice of network and protocol has severe impact of what kind of network games that can be played fluently, and what types of multiplayer games that can be played over the various networks. Further, the paper investigates various approaches that can be used in mobile multiplayer network games to minimize issues related to like latency and low bandwidth. Specifically, the paper describes various approaches for how much of the game should run locally on the client and how much should run on the server. The game we designed was named BrickBlock, and was designed to reveal all kinds of implementation issues the developer has to face due to latency in mobile network multiplayer games. The goal of BrickBlock is for a player to push other players around and into traps where they loose their lives, and to pick up power-ups on the way to make the task easier. The game relies heavily on collision detection between the player and game objects, and between players. Collision detection between players and between players and game objects gets very difficult to handle in a network game with high latency. The paper describes various approaches that can be used to handle such collisions and highlights the advantages and disadvantages of the various approaches.

KEY WORDS
Multiplayer game, Mobile computing, Network performance, Game design, Latency

1 Introduction
Online games like World of Warcraft [1] have become very popular with more than 10 million paying subscribers around the world (2008). Most current AAA titles released on consoles or PCs provide some kind of online multiplayer support. This trend has also reached mobile gaming, but mostly on mobile game consoles platforms like Sony Playstation Portable and Nintendo DS. On mobile phones, there are few online multiplayer games. Some examples are Pirates of the Caribbean [2], Samurai Romanesque [3], and Tibia Micro Edition [4]. The online multiplayer games for mobile phones on the market today are either turn-based games or slow-paced games to avoid the inherent problem with the latency and low bandwidth of wireless networks. Such games can live with network response time up to 1 second without ruining the gameplay. However, real-time multiplayer games require much lower response times if they should be fun to play. In such games, it is required that the game events are distributed over the network to all involved players many times per second. Especially, in games that involve collision detection between players, it is critical that all game events are updated frequently. The update frequency of game events depends on the game genre. For instance, a fighting game requires higher update game event frequency compared to a strategy game. In addition to response time, transfer bandwidth is critical for multiplayer games. The network performance of multiplayer mobile games must also be scalable so many players can play simultaneously.

This paper describes the results from the BrickBlock project where the focus was on investigating the limitations of multiplayer real-time games played over the most commonly available wireless networks today: GPRS, EDGE, UMTS (3G), and WLAN (WiFi). WLAN has successfully been used
for multiplayer online gaming for years and was included in the investigation as a benchmark. The investigation consisted of a network performance tests performed on mobile phones running several instances of the BrickBlock game over various networks. The test was conducted in live networks and measured the response time and the transfer speed. The paper reports on the practical implications and identified challenges based on the results of the network performance tests. Further, the paper describes various approaches for handling network issues related to collision handling and providing a fluent game experience for the user. Finally, the paper describes experiences learned from gameplay sessions over various mobile networks.

The rest of the paper is organized as follows. Section 2 describes the mobile multiplayer real-time game BrickBlock. Section 3 describes how the network performance tests were run. Section 4 presents the results of the performance tests, along with a discussion of the implications of the results. Section 5 describes various approaches for collision handling that takes network latency into account. Section 6 describes approaches for getting a smooth and responsive gameplay for multiplayer games played over mobile networks. Section 7 describes the results from testing the gameplay of BrickBlock over mobile networks. Section 8 describes related work. Finally, Section 9 concludes the paper.

2. BrickBlock – a Mobile Multiplayer Network Game

The BrickBlock game concept was developed with testing real-time performance over wireless networks in mind. This section describes the BrickBlock game.

2.1 The Game Concept

In BrickBlock, each player controls his brick around a two dimensional playfield. The goal of the game is to push other players into certain areas defined as traps. When a player is pushed into a trap he dies, loses points, and after some time respawns (re-appears). The winner of the game is the player that has died least number of times within a specified time. This concept opens for tactical play, as the players most likely will have to find some players to cooperate with in order to push and block one targeted players. Further, such alliances will have to be temporary for one player to become the winner of the game. Ambitious players will most likely jump from one alliance to another several times during a game session to make sure he is always in the best position for the victory. In other words, BrickBlock is a game characterized by its anarchy, chaos, and treachery – attributes that makes it an entertaining, unpredictable and social game.

An illustration of the game is shown in Figure 1. When the game starts, the strength, size and speed of the players' bricks are equal. This will change when a player consumes one of the three kinds of power-ups provided. The Speed power-up gives the player increased speed making it easier to avoid other players and to pick up other power-ups. The Size power-up increases the size of the player's brick making it easier to pick up more power-ups and easier to push other players. This power-up has also a negative effect, as it is easier to hit traps with the larger brick. The Strength power-up increases the player's strength making it easier to push other players around.

![Figure 1: Illustration of the BrickBlock game](image-url)
2.2 The Game Architecture

The architecture of the BrickBlock game is a combination of three architectural patterns: the client-server, the layered and the model-view controller pattern as shown in Figure 2.

![Figure 2: Architectural Overview of BrickBlock](image)

The bottom layer consists of the Communication module that manages all communication between the server and the client. The same communication and message-parsing interface is used on both sides to provide a uniform communication between the server and the clients, and thus making it easier to support various message formats such as plain text and XML.

The Test module is not necessary for running the game itself, but is used to run network performance tests between the server and the client. This module also implements the communication interface, and can therefore be used as a communication module by the models layer.

The Model layer contains the information needed to represent the current state of the game. It also keeps track of the messages needed to be sent, or being received. The model part on the server side stores and manages information about the game that also is stored and managed by the clients. The server contains the whole view of the game, while the clients have a more local representation of the game.

The View layer provides the graphical user interface for the server and the clients. The view module on the server is very simple showing the players being connected and some settings. The view module on the clients displays the game running including screens for a game lobby and the game itself. The server was implemented in Java SE while the client was implemented in Java ME.

3. Testing Real-Time Network Gaming in Live Wireless Networks

One main goal in the investigation of the performance of real-time games in wireless networks was to find the actual performance of such games in a real network environment. This ruled out the choice of using computer simulations to calculate the network performance. We chose to measure the performance using a real and network-demanding multiplayer real-time game consisting of a server running on a standard PC and mobile clients running on two Sony Ericsson K750i, Sony Ericsson K800i and Nokia N73 mobile phones. The performance tests were run several times at various times of the day and on various days to capture normal data traffic variations.
The two most important measurements for network performance for real-time games are *response time* and *transfer speed*. The former determines the expected lag of updating game changes across to the clients. The latter determines how much data can be sent between the client and server, and limits the number of players that can play the game simultaneously.

### 3.1 Response Time

The *response time test* measures the Round-Trip Time (RTT) – the time a small packet uses from the server to a client and back. The test module generates packets of only 4 bytes containing an id and a separator character. The number of packets generated depends on the number of intervals and the number of packets sent in each interval. These packets are sent with a delay of a fixed mount of milliseconds, which increases with each interval. The test calculates the time values, extracting the highest and lowest times, and calculates the average for the remaining of several runs. These values can be used to find the optimal interval with the best performance (lowest RTT).

Figure 3 shows how the RTT test was performed by sending a packet in intervals. With more packets per interval, the time between transmissions is not increased before all packets in that interval have been sent. The data packet is represented as a rectangle with a length \( l \). The time the packet uses from the server to the client and back is denoted as \( t_i \), where \( i \) the represents the number of the packet. Finally, the transmission intervals are denoted as multiples of \( \Delta I \).

![Figure 3: Measurement of Response Time](image)

The purpose of the response time test is to find the transmission interval that gives the shortest RTT resulting in low latency enabling smooth gameplay. However, every time the transmission interval is increased, the total time to send a packet is also increased with the same amount of time. The transmission interval with the lowest RTT may therefore not necessarily be the optimal transmission interval.

### 3.2 Transfer Speed

The *transfer speed test* measures the transfer time (transfer speed) of transmissions of different packet sizes from the server to a client and back. The test uses the same interval setup as the response time test (see Section 3.1). The test transmits a data packet of an initial size and the data packet size is
increased with specified amount bytes for each transmission. The packet consists of an id, a separator character, an end-of-message character, and a number of 'x' characters to fill up the rest of the packet to the desired size. The delay between each packet was defined according to the optimal interval found in the response time test (see Section 3.1). The packet size is increased after the completion of a transmission from the server to the client and back.

Figure 4 shows how the transfer speed test was performed. The packet to be sent is illustrated by the rectangle and the size of the packet is the initial length $l_0$, and the increment in size $\Delta l$. The time the packet uses from the server and back is denoted as $t_i$ where $i$ represent the number of the packet, and the interval between transmissions is denoted $\Delta I$.

![Figure 4: Measurement of Transfer Speed](image)

The purpose of the transfer speed was to compare the actual transfer speed of different mobile network technologies, and the two transfer protocols TCP and UDP. The transmission time was measured from when a packet was sent from server to client, and back. This test also found the highest and lowest times, and calculated the average for the rest. This average was used to calculate the average transfer speed by dividing the packet size with the transfer time.

4. Results of the Performance Tests

This section presents the results of the network performance tests described in previous section.

4.1 Results Of The Response Speed Test

For real-time multiplayer games like BrickBlock, the response time has high significance, as the game requires small data packets to be sent frequently. Thus, the response time values for the wireless network technology measured in this test indicated the suitability of the network for running real-time multiplayer games.

Figure 5 shows the measured response time for the wireless network technologies GPRS, EDGE, UMTS, and WLAN, and transport protocols TCP and UDP. Figure 6 shows the measured response times including the pause interval.
The charts in Figure 5 and 6 show that UDP performs much better than TCP on all networks. Further, the transmission interval with the lowest response time is between 150 and 200 milliseconds, depending on the mobile network technology used. WLAN has the lowest response time, followed by UMTS, EDGE, and GPRS as expected according to the theoretical specifications.

Figure 5 shows that there is a big difference in response time for GPRS for the two network protocols UDP and TCP. This difference is less prominent for the other networks. The difference between using the two protocols over GPRS is around 1 second in average. Further, UDP gives better response times with EDGE and GPRS than running TCP over UMTS for transmission intervals larger than 75 ms. Hence, UDP clearly is the transport protocol that provides the best performance and is the only protocol that can be used for mobile real-time games running over GPRS, EDGE or UMTS. However for WLAN, TCP gives low enough response times for playing over the network.

From both Figure 5 and 6, WLAN clearly outperforms the other network technologies. The WLAN response times always stays below 0.2 seconds for all transmission intervals, and with both transport protocols. The lowest response times measured for the other network technologies are 0.217 seconds for UMTS with UDP, 0.291 seconds for EDGE with UDP, and 0.445 seconds for GPRS with UDP. The UMTS response time is always less than that of EDGE using UDP. With TCP, the distance between the two is significantly less and the EDGE response time is even lower than the UMTS response time for some intervals. GPRS with UDP has close to identical performance as EDGE between 100 ms and 150 ms. However, for the other send intervals, EDGE is closer to UMTS. With TCP, GPRS never has a response time below 1.5 seconds, which can even make turn-based multiplayer games unresponsive.

In addition to find the optimal transmission intervals for lowest response time, we also ran several tests to look at the variation for transmission intervals between 100 ms to 250 ms. The results showed that transmission intervals around 250 ms had significant less variation than for shorter intervals. For
UDP, the standard deviation was 80.92 ms for GPRS, 27.59 ms for EDGE, 5.12 ms for UMTS, and 1.19 ms for WLAN. When the TCP protocol was used, the standard deviation increased at least threefold.

Figure 6: Measured response time including transmission interval

Figure 6 shows the response times including the transmission interval, i.e. the total response time from the previous packet is sent from the server to the server receives the return message from the client. This indicates the range of transmission intervals that will provide the lowest total response time. The longer the pause interval is, the longer the total response time will be. Figure 6 shows that pause interval values between 50 and 250 ms will provide the best total response time. Table I shows the minimum, maximum and average response times including transmission intervals. The number in Table I shows that the optimal game update intervals over the network are just above 0.5 seconds for GPRS and EDGE, 0.288 seconds for UMTS and 0.088 seconds for WLAN. If the TCP protocol is used, the optimal update intervals are 1.58 seconds for GPRS, 0.82 seconds for EDGE, 0.58 seconds for UMTS and 0.11 seconds for WLAN.

Table 1: Response Time including Transmission Intervals for Various Wireless Networks

<table>
<thead>
<tr>
<th>Network</th>
<th>UDP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPRS</td>
<td>EDGE</td>
</tr>
<tr>
<td>Min</td>
<td>0.545s</td>
<td>0.510s</td>
</tr>
<tr>
<td>Max</td>
<td>0.971s</td>
<td>0.751s</td>
</tr>
<tr>
<td>Average</td>
<td>0.779s</td>
<td>0.614s</td>
</tr>
</tbody>
</table>

4.2 Results Of The Transfer Speed Test
The transfer speed measures how much data the network is able to transport per second. For a network offering a high transfer speed, larger data packets can be transmitted without loss of performance. This
may have significant impact of how much data can be sent between the clients and the server without any lag.

Figure 7 shows the results of the transfer speed tests (short transfer time is best). The test results show the measured transfer time for each packet size instead of the transfer speed. This is because the speed can easily be calculated from the transfer time, and the transfer time can be more directly related to the kind of games we are evaluating. The figure shows the time a specified network technology applying the UDP or TCP protocol uses to send a packet of a specified size from the server to the client and back. The figure shows that when using UDP there is less variation in transfer time compared to TCP.

![Figure 7: Measured transfer time](image)

When using UDP as the transport protocol, the size of the packet does not matter up to a certain size. In Figure 7, the transfer time using UDP is almost constant for all networks. WLAN has the lowest transfer time by far; whereas the UMTS transfer time is around 200-250 ms longer. The EDGE transfer time is additional 100 ms longer, and the transfer time using GPRS is yet another 150 ms. The limit for a satisfactory transfer time depends on the transmission interval chosen and the amount of data sent to the client. With UDP, the four different mobile network technologies all have transfer times around 500ms or less. This transfer time is measured from the server to the client and back, so the time from the server to the client can be expected to be half the measured transfer time. The most common used packet sizes for games like BrickBlock do not affect the response time when UDP is used.

However when using TCP, the packet size affects the transfer time more, except for WLAN where the transfer time is barely longer than with UDP. The transfer time with TCP vary more over the different packet sizes. With some packet sizes, EDGE is better than UMTS, but in average UMTS is better. GPRS with TCP has the most fluctuating transfer time. It also has the longest transfer time with all packet sizes. The average of all the packet sizes is more than one second longer on GPRS with TCP than the second worst network technology's average transfer time with TCP (EDGE).
The upper part of Table II shows the minimum, maximum and average transfer times for when varying the packets from 40 to 760 bits. The two rows at the bottom of Table II show for what packet sizes we have the lowest transfer time (min size) and highest transfer time (max size).

Table II: Transfer Time for various Wireless Networks (40-760 bits)

<table>
<thead>
<tr>
<th>Protocol</th>
<th>UDP</th>
<th>TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>GPRS</td>
<td>EDGE</td>
</tr>
<tr>
<td>Min</td>
<td>0.467s</td>
<td>0.349s</td>
</tr>
<tr>
<td>Max</td>
<td>0.531s</td>
<td>0.410s</td>
</tr>
<tr>
<td>Average</td>
<td>0.492s</td>
<td>0.366s</td>
</tr>
<tr>
<td>Min size</td>
<td>520 bits</td>
<td>440 bits</td>
</tr>
<tr>
<td>Max size</td>
<td>200 bits</td>
<td>760 bits</td>
</tr>
</tbody>
</table>

4.3 Large Data Packets
The results in previous section shows that the transfer time does not vary much with typical sizes of data packets used in games like BrickBlock (see Section 4.2). In an additional test, we wanted to find the upper limit for package sizes (beyond 760 bits). This test was only run over UMTS using the UDP protocol, as UMTS has the most suitable characteristics for mobile network games apart from WLAN. Our test showed that the transfer time is relatively stable between 200 ms and 300 ms independent of the packet size up to 11 616 bits. For package larger than 11616 bits, all packages are lost (practical upper limit for package size using UDP and UMTS).

4.4 Practical Impact Of Network Performance On Gameplay
As expected from the specifications of the four networks considered in this paper, WLAN has the shortest response time and fastest transfer speed, followed by UMTS, EDGE, and GPRS. Since WLAN is not widely available on mobile phones in Europe, whereas UMTS is, UMTS is currently the most promising network technology suited for multiplayer real-time games that can run on most new phones. The transmission interval has significant effect on the response time. The response time decreases with increased transmission intervals, but this will of course also increase the total response time. In practice, multiplayer games run over GPRS and EDGE using the UDP protocol can expect game updates every 0.5-0.6 seconds. This means that only turn-based or slow-paced games like strategy games, role-playing games or board games can be played over these two networks. If UMTS and UDP are used, game events can be updated about every 0.3 seconds (3 updates per second). Thus real-time games without fast direct interaction between players can be played over this network. This result also means that mobile multiplayer real-time games should include code to compensate for the network latency to provide a smoother gameplay. Multiplayer games over WLAN and UDP can expect game updates every 0.09 seconds (11 updates per seconds), which is sufficient for most real-time games. For wireless networks where the user will be charged for every byte transferred, the costs of playing the game will increase with more frequent game updates and the number of players. As an example, a two-player game of BrickBlock over UMTS costs in Norway about 15 cents/min. A four-player game costs 17 cents/min. We predict that an eight-player game will cost about the double (30 cents/min) of a two-player game. New pricing policies for mobile online games must be in place to make such games popular.

The transport protocols TCP and UDP perform very differently on the various mobile networks. UDP gives better results on mobile network technologies with lower performance specifications than TCP does on technologies with better specifications. The transfer time tests indicate that the packet size has no influence on the transfer time when using UDP as the transport protocol. Because of this, the
package size does not affect the transfer time as long as the size is less than 11,616 bits and UDP is used. This means that for all the networks but WLAN, UDP must be used for real-time games. However, this means that game software must cope with possible loss of packages, which is handled by TCP but not in UDP.

The big difference in performance between UDP and TCP is related to the way the two protocols treat retransmissions. TCP was designed for wired, reliable networks where packet loss is due to congestion in the network. In wireless networks, most packet loss is due to link failure [5]. Even if different TCP implementations perform different over wireless networks, they all perform poorly compared to UDP. Another problem that must be solved when using the TCP protocol over wireless network is to handle variations in delay and rate. This problem can partly be solved by introducing acknowledge buffers that absorb the channel variations and using TCP-aware scheduling and buffer sharing algorithms [6]. Further, TCP traffic over wireless networks can be improved by choosing the optimal package size and managing retransmissions in alternative ways [7]. However, mobile real-time games require frequent updates with minimal latency without retransmissions, thus TCP is not a good solution.

The transfer time test showed that packages up to 11Kbits could be used without affecting the transfer time. This should be sufficient for most mobile multiplayer games. The amount of data should be kept to a minimum in any case to avoid network congestion issues and to minimize cost.

4.5 Challenges For Mobile Real-time Multiplayer Games

The goal of the BrickBlock project was to design a game that revealed issues related to network lag and low network bandwidth shown clearly in the gameplay:

- It is critical that the positions of the players (the bricks) are correctly reproduced on all the players' screens, as how bricks are positioned on the play area is critical to the gameplay.
- It is critical to detect when a brick (the player’s object) hits the walls limiting the play area and collision with other objects such as power-ups and traps.
- It is critical to detect when two or more bricks collide to correctly move the bricks according to the involved physical forces.

From preliminary tests running first version of the game over a GPRS wireless network with long latencies, we noticed a number of problems:

- The position of the same brick was different on different players' screens. As such, the players did not have one coherent representation of game world. When watching several mobile screens at the same time, it did not look like the players played the same game, as the network lag could not cope with the movement of the involved players.
- The collision detection with walls was not always working, as it was performed on the server to minimize the load of the mobile device [8]. Unfortunately, in some cases the server did not discover when a brick hit the wall in time, and the brick would float outside the play area (unstable state of the game).
- The most noticeable problem was inaccurate detection of collisions between players (bricks). In some cases, players could simply run over other players without any collision detection at all. In other cases, bricks were pushed around when it looked like they did not collide on one of players' screens or the bricks ended up on top of each other (illegal state).

The following two sections will look further into issues related to network lag and low bandwidth in BrickBlock and suggest approaches to overcome these problems.
5. Collision Handling

Collision management [9] is one of the most important aspects in most computer games since it usually controls how the user can move around on screen, score points, avoid enemies, shoot enemies etc. In BrickBlock, collision handling is very central to the gameplay, as the gameplay is all about pushing other players and interacting with game objects. A collision in BrickBlock occurs when the player’s brick touches another player, a wall, a trap or a power-up. The main challenge with the collision handling in BrickBlock is to ensure that all players experience the same collisions and that the game model is consistent. For such games as BrickBlock, the developer must decide whether the collision handling should be performed all locally on the client, all centralized on the server or a hybrid of these two approaches. With the network lag in mind, the developer must find the optimal solution where the server ensures a consistent representation of the game while the local mobile client ensures that the interaction between game elements locally looks and feels right to the user.

5.1 Collision With Walls

Wall collisions occur when a player's brick moves to a position where it is partly or completely located outside the game board. This happens either when the player tries to move to this position, or when another player pushes him to this position.

The first case is quite simple to handle and detect locally on the client, as the game simply have to check that the player’s next move is allowed or not as shown in Figure 8.

![Figure 8. Self-caused Wall Collisions](image)

The second case of wall collision is more complex, as it involves interaction between two players. This problem can be solved in different ways. One approach is not to allow a player to move if it pushes another player to be placed in an illegal position. The problem with this solution is that the approximated position of the pushed player is not necessarily completely correct due to network latency. A move towards a wall may therefore be incorrectly disallowed, because a player that is not actually there is detected to be standing in the way. Another approach is allowing such a move, and only checking the local player’s position against the wall. In this case, a player may actually be pushed outside the wall. Such situations need to be detected and corrected either by the client of the pushed player or the server. In both cases, the simplest solution is transmitting a new position for the pushed player so that he is placed back in a valid position. If this is done by the server, only the pushing player will be visibly affected, as the pushed player will be moved to another position shortly after the push occurred. If it is done by the pushed player, all clients will be visibly affected, as they first receive a notification that a player has been pushed, and a corrected position shortly after. But the advantage of this last solution is that this detection is already mostly done by the calculation of self-caused wall collisions. A three-step illustration for this situation is shown in Figure 9.
Unfortunately, as the figure shows, both the solutions likely result in players being placed on top of each other after the correction. Thus, the mentioned drawback of the first approach can be accepted and let wall collisions be detected by the pushing player. The result of this solution is that step 2 and 3 of Figure 3 are detected and stopped before they are executed.

5.2 Collision With Power-Up Objects
Another type of collision detection is collisions with power-up objects. When such an event occurs, the power-up needs to be removed from the game board, and the player's attributes need to be updated for all participants.

The decision is whether the power-up collision should be performed locally on the client or on the server. A local detection of power-up collision can simply be implemented using the Java MIDP built-in sprite collision. A server-based detection of power-up collision can be implemented comparing the position of the player with the position of the power-up. The advantage of local detection is that the player can get an immediate response when the power-up has been picked up (using sound, graphics or vibration) and immediate change of the brick’s attributes. The disadvantage of this approach is that multiple players can pick up the same power-up, since there will be a latency all clients knows that a power-up has been picked up. The advantage of server detection is that only one player can pick up the power-up (a consistent game model), but the player will not get an immediate response when he has picked up the power-up. Our solution is a hybrid using local collection detection and server check for consistency and removal of power-up from all players. When a power-up collision is detected, player X will get an immediate response in a form of a vibration or a sound. A message is sent to the server to check if another player has picked up the power-up. If not, a message is broadcasted to all players that player X has changed attributes and the power-up must be removed.

5.3 Collision With Traps
A trap collision occurs when a player collides with the trap object on the game board. This will usually happen when the player is pushed by other player(s) into the trap, or self caused accidents. Both these cases are similar to the power-up collisions discussed in the previous section, and are best handled by using Java MIDP's built-in support for collision detection.

The problem with several players colliding with the trap at the same time does not apply to trap collisions as with power-up objects since several players can die at the same time. However, when a player dies, he must respawn (re-appear) in an unoccupied part of the game board. This involves checking all players’ positions and finding a space available. If several players die at the same time, the game must make sure that none of the players respawn at the same place. Thus, we use the same approach for dealing with power-up collision for traps. When a player dies, his phone flashes and vibrates and get immobilized. A notification is sent to the server, which will update scores, give an available respawn position, and broadcast the update to all clients.
5.4 Collision With Other Players

Like collisions with power-up objects and traps, collisions with other players are quite simple to detect using the built-in Sprite collision. The main difference is that players move around all the time. As mentioned previously, it is impossible all the time to have completely correct information about all players’ position due to network latency. This makes player collisions harder to detect correctly and even more difficult to handle.

The simplest case is when one player is standing still while another is pushing. This is just like collision detection of game objects by updating the position of the pushed player by new positions sent by the pushing player. But when both players move at the same time, the situation is more complex. The following three situations might occur as shown in Figure 10.

![Figure 10. Player Collisions with Simultaneous Movement](image)

The left image of each case shows a possible representation of the player positions, whereas the right image shows the actual positions of the players. The three situations illustrated in the figure can arise when: 1) an existing collision is not detected because both players have moved into the same area, but the position of at least one player has not yet been received, 2) a non-existing collision is detected because both players who were in the same area have moved away, but the position of at least one player has not yet been received, and 3) an existing collision is detected, but it is not completely correct since the position of at least one player has not yet been received.

The first case may result in two players occupying the same board position for a short period of time, until the new position has been received and the collision is detected. The second case is the exact opposite of the first, and may cause problems. The consequence can be that a player is pushed even though he has actually managed to get away from the pushing player. If this happens close to the trap, the player may unintentionally die. However, like in the first case, the correction will come quickly enough to cause any critical damage. For the third case, there is no consequence for how the players experience the game. Whether this collision occurs at the edge of or at the centre of the brick, the result is the same that the strongest brick moves the other in the strongest player’s movement direction.

In BrickBlock, the server holds the most accurate picture of the total game state, while each client knows best its own state. We have chosen to manage player collisions locally to avoid unresponsive gameplay in collisions introduced by server-based collision detection. This means that the collision is not always accurate, but it is very responsive for the user.

5.5 Handling Player Collisions

Correct actions must be taken when collisions between players are detected. In BrickBlock, the results of such collisions are change of speed and movement direction for at least one of the players. First, the strength ratio between the players involved is calculated. If one of the players is stronger than the other, the strongest player will be able to push the other in the strongest player’s movement direction. How much the player can be pushed depends on the strength ratio between the players, as well as the movement speed of the strongest player. Example, if the strongest player is 50% stronger than the
weakest and the speed of the strongest player is 2, the weakest player will be pushed with a speed of: 0.5x2 = 1.

Collision handling like collision detection can be handled on the server with more processing power or on the client with shorter response time. The server has a more accurate game model than the involved clients and can calculate relatively accurate positions for the pushing and the pushed player. However, both players will be able to move forward a short time until the server receives the collision notification and transmits the new positions. Thus, the players will experience that their bricks will be moved backwards seemingly without reason. This situation is illustrated in Figure 11. The figure shows a step-by-step procedure of how calculations will be performed and messages transmitted when the server is responsible for handling player collisions. Step 5 is performed in parallel on the local and remote client. According to the figure, the redrawing of positions is carried out in step 5 on the local client. Step 2 and 4 are transmission between server and client that with a slow network can issue visible and noticeable latency for the players.

Another approach to ease this problem is to let the pushing player have responsibility for calculating the results of the collision. Figure 12 shows a step-by-step illustration of this approach. Here, the redrawing of the players occurs already in step 3. Furthermore, no message transmission is necessary before the game board is updated. This approach will give a far more responsive game experience from the pushing player’s point of view.

Calculating the new position of the pushed player and transmitting this position may lead to the problem of unnatural position corrections. However, an improvement can be achieved by letting the pushing player transmit a movement vector instead of a static position. With this solution, the pushed player will not be reset to a previous state, but rather corrected with an amount corresponding to the
strength ratio between the players and the speed of the pushing player. The procedure for detecting and handling player collisions on a client can then be carried out as described in Listing 1.

**Listing 1. Procedure for Handling Player Collisions**

<table>
<thead>
<tr>
<th>Search for collisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a collision is detected</td>
</tr>
<tr>
<td>If I am stronger than the collidee</td>
</tr>
<tr>
<td>Calculate the resulting movement vector for both players (will be equal)</td>
</tr>
<tr>
<td>Transmit the collidee’s movement vector to the server</td>
</tr>
<tr>
<td>Calculate my new position</td>
</tr>
<tr>
<td>Transmit my new position to the server</td>
</tr>
</tbody>
</table>

The movement vector solution could also be used on the server, and will reduce the problem of position corrections. However, the problem with game responsiveness still remains. When a player collides with another player, he expects one of the players to be forced by the other player. With server-side collision handling, there will be a noticeable delay before this happens.

Due to the latency in the network, a player may experience to be pushed without contact between the players displayed on his phone. Also, there may be situations where a push should occur, but does not. This is equal to the situations illustrated in Figure 10. With client-side collision handling, these situations will occur more often and with larger deviations than when performed on the server. However, such situations will be less an issue for the player than an unresponsive gameplay due to server-side handling. As a consequence, collision handling should be performed on each client when a collision is detected.

### 6. Handling of Smooth Gameplay

It is very important for any game that the gameplay and the visualization are smooth, especially in a multiplayer mobile game like BrickBlock. If the gameplay and/or the graphics stutter, it will ruin the enjoyment of the game. This means that all on-screen actions must be updated as frequently as possible on all clients, and all player-actions must immediately trigger game actions. In addition to network specific solutions, clever calculation algorithms and smart programming can be used to camouflage the network latency. This section describes some approaches we considered to ensure smooth game experience in the BrickBlock game.

#### 6.1 Movement Prediction

*Movement prediction* is an approach to overcome warping of game objects that move around due to loss of data packets or low bandwidth. *Warping* means that moving objects seems to jump from one location to another [10]. Fun gameplay with a good game flow requires accurate representation of player objects on all clients. To avoid warping and to accurately represent the player position at any time, the game client predicts the other players’ movement, based on previous movement. This leads to position changes happening earlier than by using the positions received from the server. Thus, the player movement will be perceived as more fluent.

The movement prediction is performed by calculating a vector based on the two latest known positions of the object. These positions are stored on the client and are replaced whenever a new position is received. When a new position is received from the server, the new position and the previous position are used to calculate a the movement vector by subtracting the old coordinates from the new and multiplying the result with the player object's speed property. This movement vector is then used to move the player object while waiting for the next position update from the server. This procedure for movement prediction is further described in Listing 2.
Listing 2. Procedure for predicting movement

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Last received position update from player A: ((x_0, y_0))</td>
</tr>
<tr>
<td>2</td>
<td>Receive position update for player A: ((x_1, y_1))</td>
</tr>
<tr>
<td>3</td>
<td>Movement (X = x_1 - x_0)</td>
</tr>
<tr>
<td>4</td>
<td>Movement (Y = y_1 - y_0)</td>
</tr>
<tr>
<td>5</td>
<td>If Movement (X) = 0</td>
</tr>
<tr>
<td>6</td>
<td>Movement (X = \frac{Movement \ X}{\text{Math.abs}(Movement \ X)} ) // Operate with 0's or 1's</td>
</tr>
<tr>
<td>7</td>
<td>If Movement (Y) = 0</td>
</tr>
<tr>
<td>8</td>
<td>Movement (Y = \frac{Movement \ Y}{\text{Math.abs}(Movement \ Y)})</td>
</tr>
<tr>
<td>9</td>
<td>Each time the game board is redrawn</td>
</tr>
<tr>
<td>10</td>
<td>A's position = A's position + (A's speed) [Movement \ X, Movement \ Y]</td>
</tr>
</tbody>
</table>

However, this approach has its drawbacks if a player suddenly changes direction as illustrated in Figure 13.

![Figure 13. Warping in movement prediction](image)

The figure illustrates a situation where the player has performed a 90° turn. The maximum deviance will be in a 180° turn. A calculation of the prediction error \(e\), when the player has moved a distance of \(d\) after performing a \(\alpha\) degree turn using the cosine formula \(c^2 = a^2 + b^2 - 2ab \times \cos \ AB\) is shown in Listing 3.

Listing 3. Formula for prediction error

\[
\begin{align*}
\text{e}^2 &= d^2 + d^2 - 2 \times d \times d \times \cos \alpha \\
\text{e} &= \sqrt{(2d^2 - 2d^2 \cos \alpha)} \\
\text{e} &= d \sqrt{(2 - 2 \cos \alpha)} \\
\end{align*}
\]

(Formula applied to Figure 13

Table III shows warp distances for some selected angles using the formula in Listing 3. The values in the table are based on the worst-case scenario where the player turns immediately after a position update has been sent. Statistically we can expect the average error to be half of the values in the table.

<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(0^\circ)</th>
<th>(45^\circ)</th>
<th>(90^\circ)</th>
<th>(135^\circ)</th>
<th>(180^\circ)</th>
<th>(225^\circ)</th>
<th>(270^\circ)</th>
<th>(315^\circ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E)</td>
<td>0</td>
<td>(d\sqrt{(2-\sqrt{2})})</td>
<td>(d\sqrt{2})</td>
<td>(d\sqrt{(2+\sqrt{2})})</td>
<td>2(d)</td>
<td>(d\sqrt{(2+\sqrt{2})})</td>
<td>(d\sqrt{2})</td>
<td>(d\sqrt{(2-\sqrt{2})})</td>
</tr>
</tbody>
</table>
In addition to warping of the player on the screen, the usage of movement prediction can also cause problems for calculating collisions with other players, game objects or with walls. Another negative effect of using movement prediction is that it demands some processing power on the mobile client. If many players are playing the game, the movement prediction might strain the mobile phone’s CPU and slow down the game.

However, the positive effects of using movement prediction are far greater than the negative effects. Instead of players being moved around at the screen in discrete steps, movement prediction makes the players move smoothly in most situations and makes multiplayer games on mobile devices much more enjoyable.

### 6.2 Masking Delay

A clever approach for masking network delay and solving the problem of warping introduced with movement prediction is to use animations for major changes of directions. This means that if the player moves in the same direction, the player can maintain speed and movement prediction is used. If the player decides to change the direction, the player’s game object uses an animation to change direction before it can start to move in the new direction. This animated delay gives time to get the new movement vectors right and avoid the problem of warping the player around on the screen. The main negative effect is that it will be much slower for the player to change direction of the game object. Such an approach is used in the game “Pirates of the Caribbean” [2]. When the ship in the game changes direction, the turning of the ship is animated on the client, and the position of the ship is not actually changed until a position update is received.

A similar approach can also be used in collisions to hide network latency. For example, instead of an implementation where the results of a collision will be an immediately change of position, an explosion can be animated on screen while the client waits for the update from the server.

A third approach of masking network latency is the use of interpolation of the players’ movement [10]. This means that turning of game objects will result in smooth movement instead of discrete jumps around on the screen. By using interpolation, the player’s movement will be computed by finding the intermediate positions between current and the new position. The intermediate positions are computed locally on the client and are adjusted as the according to the updates from the server. The negative effect of using interpolation is that it removes some of the responsiveness of the player’s movement and the player’s object might continue to move after the player has stopped waiting for an update from the server.

A forth approach is simply to reduce the speed of the player objects on the screen. This might seams like a useless approach, but it can be used if it fits the type of gameplay wanted. To justify slow movement in the game, the graphics and the physics of the game should justify the approach. E.g., if the player controls a snail instead of a racing car, slow movement of game objects is more naturally. Another example is that the player controls large and heavy objects that are likely to be slow to move. In this way, the game developer can make most out of the restrictions of the mobile network.

In the BrickBlock game, it was hard to utilize the approaches described in this section. Animation to hide the change of direction or interpolation to make the change of direction smoother were not useful approaches for masking the network issues as the gameplay is all about sudden movements. The approach we could use was to slow down the movement of the player on the screen to reduce warping problems when played at slow networks.

### 6.3 Message Broadcast

During gameplay, all clients will continuously send updates to the server about their local state. To keep the clients up-to-date, the server needs to forward these updates to all clients as soon as possible. The simplest approach is to make the server forward the client messages as soon they are received. However, this approach may cause a congestion of messages and an increasing queue of messages to
Another and improved approach is bundle several client messages into one message with updates that will reduce the number of messages sent considerably. The server will then broadcast the bundled message regularly to all clients at the same time ensuring a more consistent view of the game by all players. This approach requires a bit more processing on the server, but will not cause any significant delays compared to just forwarding the client updates as they come in. The main disadvantage with this approach is that it is crucial that the bundled message is not lost, as it contains much more information. This approach was chosen for the BrickBlock game.

7. Experiences
The BrickBlock game was tested through gameplay sessions over various mobile networks. The playability tests showed that the gameplay was very smooth on WLAN and that the game was fully playable over UMTS using UDP with minor latency issues. The game was even playable over GPRS and EDGE networks and that our measure for limiting latency issues mostly works well. However, there were some issues that will be covered in this section.

In BrickBlock, we use a simple movement prediction where each player simply keeps moving in the same direction until a new position update is received. A new movement vector then is calculated, and the player is moved along this updated movement vector. Most of the time, this movement prediction works satisfactory and helps the game run smoothly. However, there are a couple of situations where the movement prediction algorithm does not work.

7.1. Warping
As long as the player moves in straight lines most of the time, and does not constantly change direction, our movement prediction algorithm works very well. This is of course not always the case and may lead to need for correction of other player's positions on the local client. In the worst-case scenario, the warp distance can be as much as 2d (d is the distance moved). If this happens very often, the players will jump around on the game board every time new position updates are received, and trying to hit and push other players will be close to impossible.

This problem has two possible solutions. The simplest of these is minimizing the size of d. Since d is the distance the player moves between position updates, it can be reduced by simply sending position updates more often, or reducing the speed of the player. It is important to find the right balance between these two, and to allow some warping as a compromise.

7.2. Detecting stopped players
Another problem related to our movement prediction occurs sometimes when a player stops moving. To avoid sending unnecessary position updates when the player is standing still, the client sends two equal positions when the player stops, and then waits for the player to start moving again before sending new position updates resulting in a 0-vector. This method works very well most of the time.

However, since we use UDP protocol for performance purposes, data packets are sometimes lost. If this happens with one (or both) of the equal position updates, the player's movement vector will never be detected to be 0 and the player will not stop moving. To make things worse, since position updates are not transmitted while the player is standing still, no new position updates are received by any of the other clients. Because of the movement prediction algorithm, the stopped player will therefore keep moving in the same direction on the other player's screen. Eventually, he will disappear through one of the walls. If one of the equal positions is lost on the way from the stopped player to the server, all connected clients will experience this. If the packet is lost on the way from the server to a client, only the receiver of the packets will be affected.

The optimal solution to this problem would be implementing a mechanism for safe transmission of critical messages. This is implemented in TCP, but not in UDP. However, due to performance issues,
TCP cannot be used. Another, but much less elegant solution is to continue to send position updates when the player has stopped. In this way, a few lost position updates is not that critical, since a position update will correct the player's position soon enough. Although this solves the problem, sending more information than necessary is not desirable.

For BrickBlock, this problem is not critical. Firstly, packet losses are rare. Secondly, BrickBlock is not designed for static play, as the players have to move around all the time.

7.3. Pushing Other Players
The main goal of BrickBlock is to push other players into a trap. It has been challenging to implement the pushing of other players work in satisfactory over a mobile network. In the current implementation, this is done by calculating the strength ratio between colliding players, and moving the weakest player in the strongest player's movement direction according to this ratio. However, because of the position updates that are continuously transmitted, sometimes the force vector and the players' positions are out of sync. It can result in situations where the brick of one player gradually will be placed under the brick of another. This problem increases when one player has picked up a speed power-up.

The main approaches to reduce this problem are to increase the transmission frequency and forbid position updates from pushed players. The consequence of the latter is that the pushed player could not move away from the stronger player.

8. Related Work
This section describes research in the area of network issues and gaming. As there is little work on real-time mobile multiplayer games, papers on real-time online multiplayer games for wired network are also presented.

Busse et. al describe experiences from running a ported online game over GPRS and UMTS from an quality of service perspective [11]. Busse et. al set the maximum acceptable end-to-end delay for their ported game to be between 100ms and 200ms. The packet size of the original game was 44 bytes including the UDP header. This packet contained game states and events. As the wireless network provider used Network Address Translation (NAT), TCP had to be used instead of UDP. The game setup consisted two-player game where one client ran on a PocketPC PDA and one client ran on the game server. The game server sends game states 20 times per second. The result from this test shows that for GPRS the average response time is about 1 second, whereas for UMTS the average response time is about 285ms. The authors conclude that the game is unplayable over both GPRS and UMTS. The results found in this paper are not very different from the results we found. However, we believe that Busse et. al concluded that real-time games are unplayable over wireless networks are due to that TCP was used as transfer protocol and that the game was not implemented to handle latency of wireless networks. Our implementation of BrickBlock compensate for a less frequent update of game states than traditional wired online.

Fritsch et. al present how Content Addressable Network (CAN) can be used to support mobile games [12]. CAN was designed to improve broadcasting over the Internet and was tailored to support games by mapping an n-dimensional virtual area to the set of mobile nodes. This approach was used to provide a peer-to-peer architecture for the location-aware game Conquer the Tag running over a WLAN ad-hoc network. The main benefit from using CAN is that fewer packages are required compared to simple broadcasting.

Fitzek et. al propose an analytical framework for investigating impact of mobility on connectivity and stability of multi-hop networks using multi-player games [13]. The paper describes how the Quake II game engine can be used to simulate various mobility issues in multi-hop networks in order to
improve communication performance. E.g., the virtual world can model scenarios where the only network available is hotspots and how this affects the coverage and network performance when the users move around.

Beigbeder et. al investigate the effect of loss and latency on user performance in the online first-person shooter game Unreal Tournament (UT) 2003 running on PCs over wired network [14]. The paper describes how introduction of loss and latency in the network affects the players when they performed simple movement (following a trail), complex movement (navigate an obstacle course), precision shooting and a death-match. From studying UT 2003 servers, they found that maximum loss rate was about 3% and maximum latency was about 140ms. Although the introduction of loss and latency affected the players' performance to some degree, the difference was not statistically significant. E.g., the performance of precision shooting decreased for latencies above 100ms, and for latency over 300ms the performance decreased 50%. For movement, neither latency nor loss had a noticeable impact on the player's abilities. In full game tests, packet loss did not impact the player performance, but increased latency caused a decreasing trend for the player performance although not significant.

Beigbeder et. al also noticed that all the packages sent while playing UT 2003 was less than 400 bytes in size. Finally, the players were able to notice sluggishness in gameplay for latencies as low as 75ms and found the game less enjoyable at latencies over 100ms.

Feng et. al describe the results of an analysis of network traffic of the online game Counter-Strike [15]. Their analysis shows that network traffic from fast-paced action games are very different from e.g. traffic from web browsing characterized by small, highly periodic UDP packets. UDP is used for most online games, as it is needed to send packets at an interval shorter than it would take to retransmit lost packets. The dominant source of the network traffic is real-time action and coordinate information sent back and forth between clients and server. In addition, game states are broadcasted to all clients at fixed intervals. Feng et. al found that a typical game session each client will consume about 40kb/s. The game state is sent to all clients every 50 ms, where as the packet size varies from 40 to 300 bytes.

Chen et. al investigate the network traffic of a Massive Multi-player Online Role Playing Game (MMORPG) called ShenZhou Online [16]. The results shows that MMORPG have similar network characteristics as FPS that they both generate small packets and require low bandwidth. MMORPGs require less bandwidth than FPS due to less real-time gameplay. The more distinctive network characteristics of MMORPGs are the strong periodicity where game updates are accumulated and sent at fixed time intervals, the temporal locality in the game traffic due to chain-reaction of actions, the irregularity if traffic due to diversity of user behavior, and the self-similarity of the aggregate traffic due to the heavy tailed activity/idle activities of individual players. Typical packet size for this game for overall game traffic is 84 bytes in average. The TPC protocol is used for this game and 38% of the packets are pure TCP acknowledgements. The average bandwidth required for this game is 7Kbps, which is much lower than FPS games. Further, the observed inter-packet times are typically 333ms (1/3s) for fighting actions and 167ms (1/6s) for movement actions. This means it would normally take twice the inter-packet times to distribute actions from one player to all other players.

Dick et. al analyze how network latency and jitter affects the performance and perception in multiplayer online games (wired) [17]. This paper presents a survey where players state their subjective perceptions for how network latency and jitter affects the performance and gameplay for twelve different games representing four different game genres: first person shooter, real-time strategy game, sport game and car racing simulation. The result of this survey shows the player's perception of the magnitude of latency that is accepted for an unimpaired game is about the same for all game genres: 80.7ms in average. The perception of how much network latency that can be tolerated before it ruins the gameplay is up to 150ms with an average of 118ms. For the football (European) game FIFA, 100ms was the maximum latency tolerated. It was a bit surprising that the players did not perceive that
first person shooters would require lower latency than real-time strategy games which is characterized with a slower gameplay.

Sheldon et. al describes results from a controlled experiment to investigate the effect of latency on user performance in the real-time strategy game Warcraft III [18]. The experiment considered three different type of activities in the game: (1) building - players gather resources, construct defenses and recruit units, (2) exploring - players send units out to determine geographical layout and location of other players' unit, and (3) combat - players engage their units with other units in battle. The results of the experiment show that there is no significant effect on the performance of the players when the latency is increased (from 0 to 3500 ms). However, for exploring there is some correlation between explore time and latency (0.63). The results from analyses from playing the full game showed that the round-trip times was about 100 ms for the game width less than 0.1% data loss. The average data rate was about 4.0Kbps, where as typical packet sizes where 46 or 49 bytes (including headers). Analysis of users playing the game showed that the users could compensate for latencies up to 500ms. For latencies above 800ms, the game appeared erratic which degraded the game experience.

Cacciaguerra and D’Angelo presents an approach for mimicking the mobile player in case of network failure until the network is working again to enhance the overall playability of network games [19]. This is not very relevant to games like BrickBlock, as the game sessions are short and very intense.

Approaches for managing network issues in online games can be found in [20], [21] and [22].

9. Conclusion
In this paper we have presented results from performance tests running a real-time multiplayer game over various wireless networks. The results show that UMTS can be used for real-time games when the UDP protocol is used and when the game software compensates for the network latency. For most wired online games, the gameplay gets affected when the latencies are higher than 150ms. For mobile online games played over UMTS, twice the latency must be tolerated. This means that the game must be designed to limit the effects of the latency. For collision handling, it is important that local collision management is used to detect that a collision has occurred to give the player an immediate response. If the collision involves other players, the player’s movement should be drawn before the position and game state is sent to the server. The server will look at the various players positions reported and broadcast a consistent state to all players. If a collision involves common game objects such as traps or power-ups, the same approach is used.

Other approaches to compensate for mobile network latency are movement prediction, movement interpolation, the use of animation when changing direction of the player’s object, and slowing down the movement speed of the player. If these approaches are used in a clever way and the design of the game has taken these approaches into account, it is possible to develop interesting and fun real-time multiplayer games that can be played over mobile networks.

Acknowledgement
We would also like to thank Richard Taylor at the Institute for Software Research (ISR) at University of California, Irvine (UCI) for providing a stimulating research environment and for hosting a visiting researcher from Norway.

References
[2] mDisney-Studios, "Pirates of Caribbean Multiplayer Game," Web:


Comparison of TCP Protocols over Mobile Ad Hoc Wireless Networks," in Proceedings of the
Electronics, Robotics and Automotive Mechanics Conference - Volume 02: IEEE Computer


over Wireless Networks," in Proceedings of the 17th International Conference on Distributed

fifteenth annual ACM symposium on Principles of distributed computing Philadelphia,


networked mobile gaming," in Proceedings of 3rd ACM SIGCOMM workshop on Network

[12] T. Fritsch, H. Ritter, and J. Schiller, "CAN mobile gaming be improved?," in Proceedings of
5th ACM SIGCOMM workshop on Network and system support for games Singapore: ACM,
2006.

[13] F. Fitzek, L. Badia, M. Zorzi, G. Schulte, P. Seeling, and T. Henderson, "Mobility and
of the 2nd workshop on Network and system support for games Redwood City, California:

loss and latency on user performance in unreal tournament 2003," in Proceedings of 3rd ACM
SIGCOMM workshop on Network and system support for games Portland, Oregon, USA: ACM,
2004.


[16] K.-T. Chen, P. Huang, C.-Y. Huang, and C.-L. Lei, "Game traffic analysis: an MMORPG
perspective," in Proceedings of the international workshop on Network and operating systems

perception in multiplayer games," in Proceedings of 4th ACM SIGCOMM workshop on Network

performance in Warcraft III," in Proceedings of the 2nd workshop on Network and system
