Problems and Solutions for Mobile Multiplayer Real-time Games over Wireless Networks

Alf Inge Wang[1], Eivind Sorteberg[1], Martin Jarrett[1], and Anne Marte Hjemås[2]
[1]Dept. of Computer and Information Science
Norwegian University of Science and Technology
N-7491 Trondheim, Norway
alfw@idi.ntnu.no, sorteber@idi.ntnu.no, martinja@idi.ntnu.no,
[2]Telenor Research & Innovation
Trondheim, Norway
anne-marte.hjemas@telenor.com

***Abstract*—Most recent released PC and console games offer multiplayer online support where players can compete or cooperate in real-time. However, very few games on mobile phones provide multiplayer support other than shared high score lists. This paper describes problems game developers have to face when developing real-time multiplayer games for mobile phones. Typical problems the developer has to deal with are incoherent representation of players' positions, jagged player object movement, missing collision detection between player objects and background, and wrong or missing collision detection between player objects. The paper describes the mobile multiplayer real-time game BrickBlock, which was developed to investigate the performance issues related to multiplayer games played over the wireless networks. Further, we describe our approaches to solve network latency problems in the game, which made the game playable even over networks with high latency.**

Keywords: Mobile and Wireless Collaboration Systems, Collaboration M&S for Exercise Support and Gaming, Multiplayer real-time games

I. Introduction

Online games like World of Warcraft [3] have become very popular with more than 9 million paying subscribers around the world (2007). Many new games support some kind of online functionality, and many online multiplayer gaming. This trend has also reached mobile gaming, but mostly on mobile game consoles like Sony Playstation Portable and Nintendo DS. On mobile phones, there are few online multiplayer games. Some examples are Pirates of the Caribbean [11], Samurai Romanesque [10], and Tibia Micro Edition [7]. The online multiplayer games for mobile phones on the market today are either turn-based games or slow-paced games to avoid problem with the latency and low bandwidth of wireless networks. Such games can live with network response time up to 0.5 to 1 second without ruining the game play. However, real-time multiplayer games require much lower response time if the games should be fun to play. In such games, it is required that the movements and actions of all the players in the game are updated on all mobile devices many times per second. Especially, in games where collision detection is involved, it is critical that the player events are distributed to all clients frequently to be able to detect crashes or direct contacts between objects in the game. The update frequency of player events depends on the game genre. For instance, fighting games require high game event update frequency compared to strategy games. 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.

In the project MObile and Social gameS (MOSS), the goal is to explore the opportunities of developing mobile multiplayer real-time games for mobile phones. In the first phase of this project we wanted to assess the limitations of existing wireless networks when used to play multiplayer real-time games. The wireless networks we consider are the mobile networks available in Norway: GPRS, EDGE, UMTS (3G), and WLAN (Wifi). From experiences on mobile consoles we know that WLAN works well for real-time multiplayer games, so we wanted to include WLAN in our tests as a benchmark. As only high-end mobile phones provide support for WLAN, other wireless networks are considered to be more important for multiplayer gaming on mobile phones.

This paper describes the mobile multiplayer real-time game BrickBlock, which was developed to investigate the performance issues related to multiplayer games played over the wireless networks. BrickBlock is a game where players must collaborate and compete to win. The paper describes problems and solutions relevant for visual real-time collaborative systems with mobile clients. It investigates how much of the coordination of the system should be managed by the server and how much by the clients. Further, we describe our approaches to solve network latency problems in the BrickBlock game, which made the game playable even over networks with high latency.

The rest of the paper is organised as follows. Section II describes the mobile multiplayer real-time game BrickBlock, Section III describes issues related to Collisions, Section IV describes how collisions can be handled, Section V discusses game control issues, Section VI describes experiences from

running the game, Section VII describes related work, and finally Section VIII concludes the paper.

II. BrickBlock - A Mobile Multiplayer Game

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

A. 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, looses points, and respawns. The winner of the game is the player that has died least number of times within a limited time. This concept opens for tactical play, as the players most likely will have to find other players to cooperate with in order to push and block other 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 they are always in the best position for the victory. In other words, BrickBlock is a game characterised by its anarchy, chaos, and treachery - attributes that makes it an entertaining, unpredictable and social game.

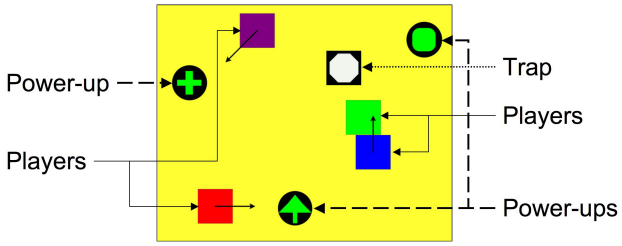

Fig. 1. Illustration of the BrickBlock game

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 power-ups. 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, but it also makes it easier to hit traps. The **Strength power-up** increases the player's strength making it easier to push other players around.

B. 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.

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

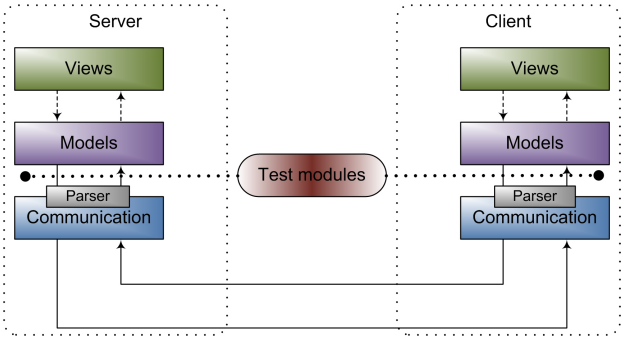

Fig. 2. Architectural overview

between the server and the clients to support various message formats such as plain text and XML.

The *Test module* is not necessary to run 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.

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 has a more local representation of the game.

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

C. Real-time Game Challenges

The BrickBlock game concept was developed to create a game with very high real-time requirements for a game played over a wireless network. The aim was to create a game that would reveal gameplay issues related to network lag and low network bandwidth:

- 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 (see Figure 1) hits the walls limiting the play area.
- 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 the game over a GPRS wireless network with long latencies (0.6 secs), 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.
- In the first version of the game, the wall detection was performed on the server to minimise the load of the mobile device. 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). This problem cannot simply be solved by doing the collision detection locally as all the players involved have to be taken into account. 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 or the bricks ended up on top of each other. The latter should not be an allowed state of the game.

To avoid these problems, all the mobile clients must be updated over the wireless network frequently so they all have about the same representation of what is going on in the game. How frequently the data must be exchanged between the server and the clients depends on how much the players are moving the bricks around, and the screen framerate of the game. Ideally the data exchange update rate should be equal to or above the framerate of the screen. The screen framerate of a game depends on the kind of game and the kinds of movement on the screen. If objects move very fast on the screen, the framerate should be higher. A game like BrickBlock does not require a framerate above 10 frames per second, which means that the data should ideally be exchanged more than 10 times per second.

To investigate whether mobile multiplayer real-time games can perform well over existing wireless networks we performed some network performance tests. The results from these tests showed that the expected latency for game updates using the GPRS or EDGE is 0.5-0.6 secs, 0.3 secs for UMTS and 0.09 secs for WLAN [15]. The rest of this paper will describe issues and solutions to minimise the effect of the network latency on the gameplay.

III. Issues related to Collisions

Collisions are one of the most important aspects in nearly all computer games with moving objects. For shooter games, collision detection is needed to detect when the players shoot each other or run into each other. For classic games like Tetris, collision detection is needed to stop the bricks in the correct position. Even in games where collisions do not have immediately visible effects, like simple driving games, collision detection is needed for example to detect when the driving surface changes, i.e. when the car goes off track and onto grass.

In BrickBlock, the need for collision detection and handling is obvious. Without collision detection, pushing other players is impossible, and nothing will happen if the players move across a trap. This section discusses the different situations where collision calculations are needed, and whether these calculations are better handled on the server or on each client.

A collision in BrickBlock occurs when a part of a player's brick touches another object or a wall. This can happen when a player moves his own brick into the other object or wall, or when he is pushed. There are four main causes for collisions: collision with walls, power-up objects, traps, or other players. In the following, each of these cases is discussed, and we propose some approaches for detecting the collisions.

A. 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 himself, or when another player pushes him to this position. The first case is quite simple to detect, as this only requires checking if the next move causes the brick to end in an illegal position. If so, the move is disallowed. Since this is such an easy case of collision detection, self-caused wall collisions should definitely be handled locally on each client. Figure 3 illustrates this situation.

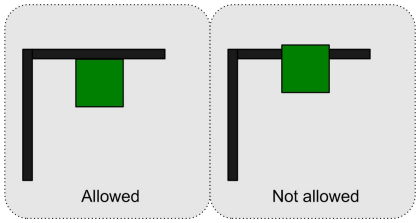

Fig. 3. Self-caused wall collision

The other case of wall collision is a little more complex, as this involves interaction with another player. Several possible solutions are possible for this situation. Firstly, a similar approach as mentioned above can be used: If pushing a player results in that the other player is placed in an illegal position, the move is disallowed. However, the problem with this solution is that the approximated position of the pushed player is not necessarily completely correct, because of the latency of information transmissions. 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 solution 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 for such events 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, the only client visibly affected by this action is the player that pushed, 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 4.

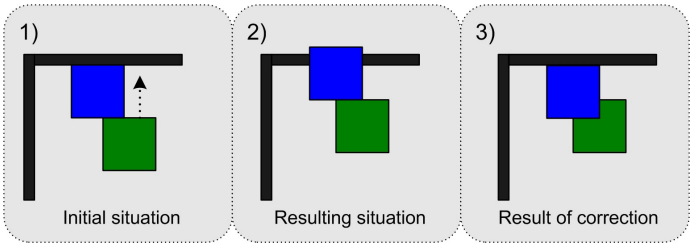

Fig. 4. Externally caused wall collision

Unfortunately, as the figure shows, both the solutions involving correction when a collision is detected is likely to result in players being placed on top of each other when the pushed player is returned to a legal position. Thus, the mentioned drawback of the first solution 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 4 are detected and stopped before they are executed.

B. 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. Like wall collisions, there are several solutions for this kind of collision detection, all of which have both advantages and disadvantages.

The simplest solution for detecting power-up collisions is to utilize the sprite collision available in the *javax.lcdui.microedition.Sprite* class in MIDP 2.0. This can only be performed by the local client, or it can be performed by all the clients each time a player moves. However, performing a collision detection every time a player position is received on all clients requires quite an amount of processing. This is not desirable for a game designed for mobile phones, and should be avoided when possible. Furthermore, since information from one client takes some time to be distributed to the others, the player may see the power-up on their local screen for some time after another player has picked it up. This may lead to several players picking up the same power-up object.

Another possibility is comparing the player's position with the position of the power-up objects on the server whenever a position update is received. A server is normally far more powerful in terms of resources than a client, but this solution leads to visible delay for the players, as the collision with the power-up will not be registered before a little after the actual collision. The game should offer feedback to the player when colliding with a game object, such as vibration or flashing lights. If the collision detection is performed on the server, this feedback is likely to appear too late.

Hence, there appears to be a choice between saving the client for these calculations and ensuring that only one player can pick up a power-up, or introducing a lag that can be avoided. However, the two solutions can be combined into a solution that both ensures only one player picking up a power-up, as well as immediate feedback to the player. In this solution, the collision detection is performed locally on the client, as in the first solution, and the phone flashes and/or vibrates if a collision is detected. However, instead of immediately increasing the player's attributes, a notification that the player has collided with the power-up is transmitted to the server. The server then checks if the power-up has been picked up by any other players. If not, the server notifies all connected players that player X has picked up a power-up object, and has increased one of his attributes. All clients must then remove the power-up object from the game board once they receive the notification.

This solution still contains the problem of introducing more collision calculation on the client. However, in this case, the extra processing is worth the cost, because of the increased immediateness of the game. Thus, the detection of power-up collisions should be performed locally on the player's client when he moves. When a power-up collision is detected, a notification is sent to the server, and if the pick up is approved, the notification is forwarded to all connected players.

C. Collision With Trap

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, but it can also happen if the player is unlucky and moves himself into the trap. Both of these cases are similar to the power-up collisions discussed in the previous section, and is best handled by using the built-in support for collision detection in MIDP 2.0. Trap collisions and power-up collisions are therefore detected equally and at the same time on the local client.

The problem with several players colliding with the trap at (close to) the same time does not apply to trap collisions as with power-up objects. There is no rule against several players dying at the same time. However, when a player dies, he needs to be moved to an unoccupied corner on the game board. This involves traversing the player list and comparing the players' positions to the possible new position of the player. In itself, this operation is much like the collision detection already performed on the client. However, if several players die at the same time, all of these players need to be moved to an available corner. Because of the network latency, the new positions may be generated, and the players moved to the corner, before the other player's new positions are received. Hence, two or more players may be placed in the same corner if the respawning position is generated on the clients.

Due to this problem, collisions with traps are handled in the exact same way as collisions with power-up objects. If a player collides with the trap, his phone flashes and/or vibrates, and a collision notification is sent to the server. The server then generates the player's respawning position, as well as the player's new score, and transmits this information to all players.

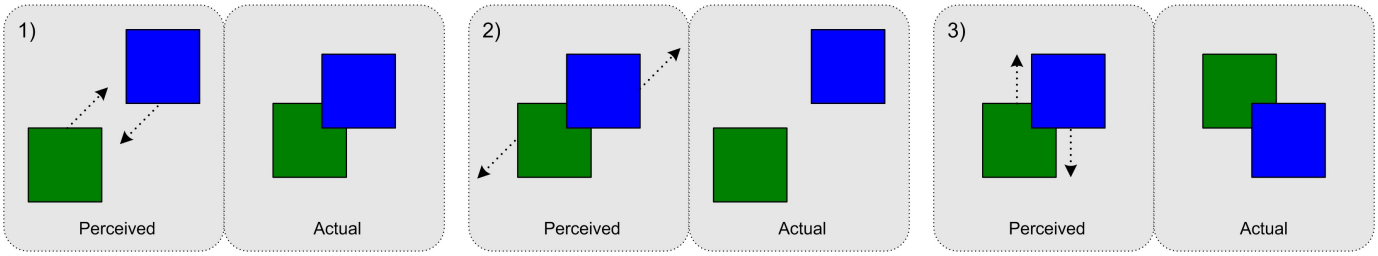

Fig. 5. Player collisions with simultaneous movement

D. Collision With Other Players

Like collisions with power-up objects and traps, collisions with other players are quite simple to detect using *Sprite* objects. However, power-up objects and traps have constant positions and do not continuously move around on the game board like players do. As mentioned previously, it is impossible to have a completely correct overview of exactly where all the players in the game are at all times. This makes player collisions harder to detect correctly than collisions with other game objects, and even more difficult to handle.

The simplest case of collision detection and handling between two players is when one of the players is standing still while the other is pushing. In this case, the collision detection is similar to game objects. The position of the pushed player can then simply be updated by letting the pushing player send a message that says that the player has been pushed to a new position.

But when both players move at the same time, the situation is more complex because of the network latency. This may result in three different situations. Figure 5 illustrates these situations for collisions between two players, but the same is true if three or more players collide. 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.
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 cases* 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. However, this is not very problematic, as the only time this happens is when the players touch very briefly, and does not try to push each other. The *second case* is the exact opposite of the first, and may in some situations be more problematic. The consequence of this case 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 occur fast enough that we do not judge this latency to be a critical issue. For the *third case*, there is no consequence for how the players experience the game. A collision is a collision, and whether this collision occurs at the edge of or at the centre of the brick, the result is the same. A collision has occurred, and the strongest brick moves the other in the strongest players's movement direction.

As previously mentioned, the server contains the most accurate approximation of the game state in sum, but each client contains the most accurate representation of its own state. This means that a collision that is detected on the server is more likely to be correct than one detected on the client. On the other hand, this solution introduces a visible latency to the game. The player will see that he collides with another player, but the effect of this collision will not register until the server has received the player's new position, detected the collision, and returned a collision notification. In other words, the player will experience that he is moving a bit over the other player before the collision registers and the bricks start pushing each other. Because of this, and since the consequences of a little inaccurate collision detection are not too critical, BrickBlock lets each client be responsible for detecting collisions with other players.

IV. Handling Player Collisions

When collisions between players are detected, these detections have to be handled so that the correct actions are taken. In BrickBlock, the results of such collisions are change of speed and movement direction for at least one of the players. For such events, several factors need to be calculated. First, the strength ratio between the players involved in the collision needs to be calculated. If one of the players is stronger than the other, the strongest player will be able to push the other in the strongest players'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. 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 $(0.5 \times 2 =) 1$. Since the players are pushing each other, the contact will be maintained, and the strongest player will also move with a speed of 1.

Like the other elements discussed in previous section, collision handling may also be handled both server and client

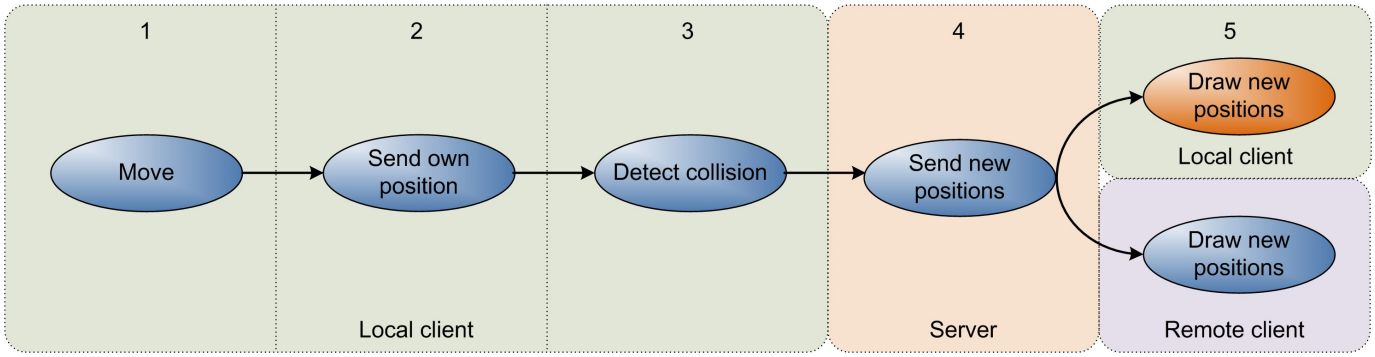

Fig. 6. Server-side collision handling

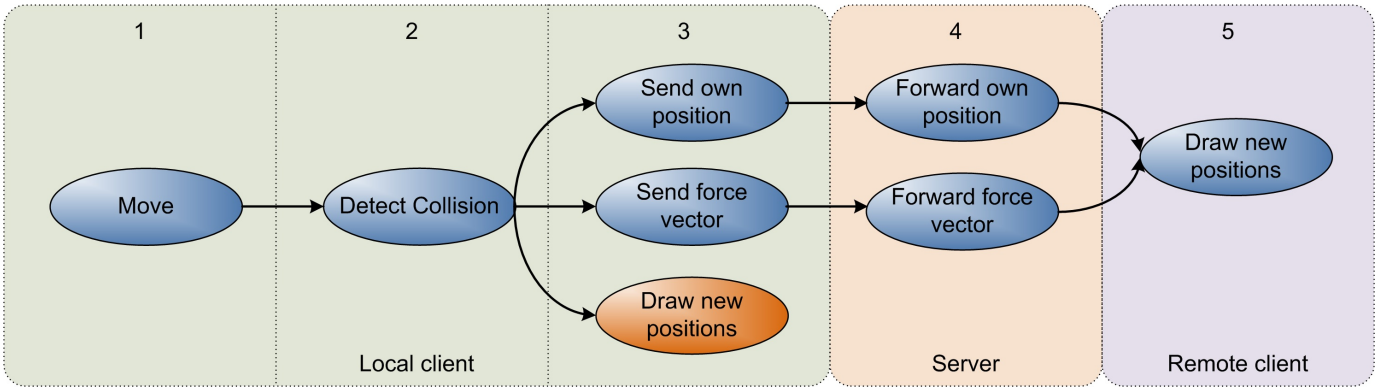

Fig. 7. Client-side collision handling

side. While the server has the advantage of plentiful processing powers, performing calculations on the client often leads to a more responsive game from the player's point of view.

In the case of player collisions and force movements, collision handling on the server profits from its more accurate world model compared to the pushing player, when it comes to calculating the new position of the pushed player. When the server is notified that a collision has occurred, it is able to calculate the new positions of both the pushing and the pushed player with relatively accurate values. However, the problem of visible delay on the involved clients once again arises. Both players will be able to move forward for a short time while the server is waiting for the collision notification, and when the server transmits the new positions, the players will experience that they are moved backwards seemingly without reason. This situation is illustrated in Figure 6. 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. Where several boxes are placed over each other, the actions are performed in parallel. As the figure shows, the redrawing of positions happens first in step 5 on the local client. Two of these steps consist of transmission between server and client, and with a slow network, it is easy to understand that this solution involves significant delay for the players.

The other solution is letting the pushing player have responsibility for calculating the results of the collision. A step-by-step illustration of this solution is shown in Figure 7. Here, we see that the redrawing of the players happen already in step 3. Furthermore, no message transmission is necessary before the game board is updated. This will lead to a far more responsive game from the player's point of view.

As mentioned previously, it is likely to be a deviance between the other player's real position and its perceived location on the local client. Calculating the new position of the pushed player and transmitting this position may lead to the same problem with seemingly 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 Figure 8.

The movement vector solution could also be used at the server, and will reduce the problem of position corrections. However, the problem with responsivity 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

```
Search for collisions
If a collision is detected
  If I am stronger than the collidee
    Calculate the resulting movement vector for both players (will be equal)
    Transmit the collidee's movement vector to the server
    Calculate my new position
    Transmit my new position to the server
```

Fig. 8. Procedure for handling player collisions

his phone. Also, there may be situations where a push should occur, but does not. This is equal to the situations illustrated in Figure 5. 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 a smaller source of irritation than the delay associated with server-side handling. As a consequence, collision handling is performed on each client when a collision is detected.

V. Game Control Issues

In addition to collisions, running the game itself requires a number of calculations that must be performed throughout the game session. The state of the game is constantly changing, and events occur both because of player interaction and because of the game's inherent behaviour. This section presents and evaluates the most significant of these events.

A. Power-ups

Power-up objects in BrickBlock are generated with random intervals. To keep track of these intervals, the power-ups should be generated by one of the devices in the network. In a peer-to-peer version of BrickBlock, the game initiator was appointed as game master, and had the responsibility for power-up generation. However, using the same approach in a client-server network with relatively high latency may give the game master an advantage compared to the other players. He will see the power-ups once they are generated, while the other players must wait for the notification.

A better solution is to let the server handle power-up generation. Some latency is still involved, and participants with slow connections may receive the notifications a little later than others. This was also the case in the former. In addition, none of the clients need to use their valuable resources for power-up generation, but delegates this responsibility to the far more powerful server.

There are two possible scenarios that cause the removal a power-up object from the game board. The *first* is when the power-up times out without having been picked up by any of the players. This is quite similar to generation of power-ups, and should be handled by the server for the same reasons. The *second* is when power-up objects are removed when a player picks up a power-up. This event was discussed earlier in this paper, and the conclusion was that the clients should themselves detect when they collide with a power-up object. When such an event is detected, the client sends a notification to the server, requesting permission to activate the power-up. If the request is approved by the server, this power-up activation is forwarded to all players, along with the attribute increment provided by the power-up object. The clients are then responsible for removing the power-up in question from the game board.

When a power-up has been activated, it remains active for the player for a set time interval. Detection of when a power-up is deactivated is also a task that can be performed both server and client side. However, if this detection is performed client side, the exact same calculation has to be performed on every one of the clients. Of course, each player can be responsible for his own power-ups and send a notification when a power-up times out. Still, this check has to be run rather often, and will occupy more of the mobile phones limited resources.

If this check is performed on the server, it only has to be performed each time the active power-ups are checked. Since the server also has the most available resources, detection of timed out power-ups should be performed on the server. Hence, the server runs through all the active power-ups for all the players with set intervals, and if a power-up deactivation is detected, a notification is sent to all players, who then set the affected player's attributes accordingly.

B. Game Settings

To give all the players a feeling of being equally involved in the game, it is important that all players have the same opportunity to change the settings of the game. Examples of such settings are how long a game should last, how many players are allowed in the game, or possible score limits. Changing such settings should naturally be performed on the local client, and the changes in settings should then be transmitted to the other players through the server.

However, the control of these settings can be implemented in various ways. In a peer-to-peer Bluetooth version of BrickBlock, the game master was responsible for handling these settings, and detecting whenever a change in the game occurred (such as a time-out or reached score limit).

This solution is also possible to use for the server-client architecture, by letting the game initiator be game master. The problem with this approach is the latency discussed previously. But for this kind of events, this latency is not critical. It is not critical if a game finishes a little bit earlier for one player than

for another.

Another client-based approach for this kind of events is letting all the clients be their own game master. All the clients have control over the different settings, and if a limit is reached, the game is simply closed locally. However, both of these client-based approaches require some background calculations. Even though these calculations are not very demanding, a server-based approach is not in any way worse, and in addition, frees the client from having to perform the calculations.

A server-based approach to this task requires the server to have a complete model of the game, such as player scores, number of players in the game, and time elapsed. Some of these elements are naturally stored on the server (such as players connected to the game), whereas others can be implemented with a minimal amount of effort. In this way, the server can continuously check the state of the game, and (close to) immediately send a "game over"-notification when the game should be ended. As mentioned, this takes some calculation load off the clients. Furthermore, like with power-up objects, this approach reduces the small downside of delayed "game over"-notifications mentioned for the game master client-side approach.

VI. Experiences

Our experiences from running BrickBlock showed that the game was playable even over GPRS and EDGE networks and that our measures for limiting latency issues mostly works well. However, there were some issues that will be presented 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. Then, a new movement vector is calculated, and the player is moved along this 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 as well as we could have wished for.

A. 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. Unfortunately, the players do not always necessarily move in straight lines. If a player feels like it, he may change direction as often as he likes. This may lead to need for correction of other player's positions on 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. However, both of these methods have their downsides. If position updates are sent more often, the amount of data sent per game will increase correspondingly.

On the other hand, reducing the player's speed leads to the players' bricks moving slower on the game board. This is very likely to decrease the playability. For these reasons, it is important to find appropriate values for the frequency of position updates and the players' speed. Some warping will have to be allowed as a compromise.

Another solution to minimize the warping is interpolation. This is a technique that reduces the amount of warping significantly, or removes it entirely. However, interpolation and smooth turning is not suitable for the kinds like BrickBlock where the player changes directions very suddenly.

The movement prediction is definitely a problem in our current implementation of BrickBlock, because of the warping problem. We have not found a completely satisfactory solution to this problem.

B. 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. The receiving clients then calculate the player's movement vector based on these positions. Since the positions are equal, the movement vector will be 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 never stopping the sending of position updates, or sending them less frequently. 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.

A sort of middle way would be sending a larger number of position updates before stopping the transmission. In this way, the probability of at least two equal position updates reaching

their destinations would be improved. However, from our tests we have discovered that one packet loss often is followed by several more packet losses. As a consequence of this, if two packets are lost, it is likely that five packets would also be lost.

For our prototype game the problem with players not stopping because of lost packets is a non-critical problem. Firstly, such packet losses are rare; at least with the network conditions we have tested the game. Secondly, and more importantly, BrickBlock is not designed for static play. Players that are not moving cannot push other players. At the same time, they are easy targets for other players seeking to push them into the trap.

C. Pushing Other Players

The main goal of BrickBlock is pushing other players into the trap, and by doing so causing a negative point for the pushed player. Making this force pushing work in a satisfactory way has proved to be a challenge in our implementation. In the current implementation of BrickBlock, 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, the pushing does not work as well as could be desired. Every time a position update is sent from the pushed player, his position is corrected with all the other clients. Since these position updates are not necessarily synchronized with the force vector, this correction may lead to pushed player being corrected to a position he has actually been pushed past. When this happens several times, the player will gradually be placed more and more under the pushing player. When a player has picked up a speed power-up, this problem is even worse, as the difference between the positions in the updates are even greater.

There are several possible methods that can be used to reduce this problem. However, we have not been able to find any solutions that solve the problem satisfactory.

One solution is to *increase the transmission frequency* to reduce this problem. If position updates were sent more often, the deviance between the different clients' model of the game board would be reduced. Collisions would be detected closer to the same time with the involved players, and the weakest player would not be allowed to move toward the stronger. This increases the amount of data transmission in the game.

Another solution is to *forbid position updates from pushed players*. This could be done by either the server or by the pushing player. The server would likely be the best alternative, as this would be the fastest way to notify all connected clients. However, the consequence of this approach would be that the pushed player could not move away from the stronger player. Still, a variant of this method, where only limited movement from the pushed player is allowed, would probably be the best way to improve the force push functionality of BrickBlock. For example, the server could calculate the positions of the pushed player based on his previous location, his current movement vector, and the force vector received from the pushing player. Then the server could transmit this position to the other clients, instead of forwarding the position update from the pushed player immediately.

The current version of BrickBlock does not work as well as it should because of this problem. Players can push each other around the board, but sometimes there will be some player control issues. It is unlikely that there is one approach that can make the pushing perfect. However, an approach involving the server calculating the pushed player's position should be investigated.

VII. 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.

In [4], Busse et. al describe experiences from running a ported online game over GPRS and UMTS from an quality of service perspective. The game setup consisted of a 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, where as 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 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 compensates for a less frequent update of game states than traditional wired online games.

In [1], 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. The paper describes how introduction of loss and latency in the network affects the players when they performed simple movement (following trails), complex movement (navigate an obstacle course), precision shooting and a death-match. 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.

In [6], Chen et. al investigate the network traffic of a Massive Multi-player Online Role Playing Game (MMORPG) called ShenZhou Online. 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 behaviour, and the self-similarity of the aggregate traffic due to the heavy tailed activity/idle activities of individual players.

In [8], Dick et. al analyse how network latency and jitter affects the performance and perception in multiplayer online games (wired). This paper presents a survey where players state their subjective perceptions for how network latency and jitter affects the performance and game play 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 characterised with a slower gameplay.

In [14], 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. The experiment considered three different type of activities in the game: (1) building - players gather resources, construct defences 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). 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.

Other work related to network issues and games can be found in [2], [12], [13], [9], and [5].

VIII. Conclusion

In this paper we have presented issues and solutions for managing latency issues related to mobile multiplayer real-time games. The main problem is to ensure a coherent representation of the game world for all involved players and to ensure that player movement and actions are smoothly and correctly represented for all players. The simplest solution to this problem is of course to play the game over wireless networks with low latency like WLAN. However, it is possible to minimise the problems also for slower networks like GPRS or EDGE, by choosing the right balance between client collision detection and server management. The BrickBlock game presented in this paper was a very demanding game in terms management of object collision. More work should be done on other types of games to improve the solutions for latency in real-time games.

References

[1] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151, New York, NY, USA, 2004. ACM.

[2] Y. W. Bernier. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. In *Game Developers Conference*, February 2001.

[3] Thomas W. Brignall and Thomas L. Van Valey. An online community as a new tribalism: The world of warcraft. *Hawaii International Conference on System Sciences*, 00:179b, 2007.

[4] Marcel Busse, Bernd Lamparter, Martin Mauve, and Wolfgang Effelsberg. Lightweight QoS-support for networked mobile gaming. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 85–92, New York, NY, USA, 2004. ACM.

[5] Wu chang Feng, Francis Chang, Wu chi Feng, and Jonathan Walpole. A traffic characterization of popular on-line games. *IEEE/ACM Transactions on Networking*, 13(3):488–500, 2005.

[6] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. Game traffic analysis: an MMORPG perspective. *Computer Networks*, 50(16):3002–3023, 2006.

[7] CipSoft. Tibia Micro Edition - the first mobile online roleplaying game. Web: http://www.tibiame.com/home/?language=en, 2007.

[8] Matthias Dick, Oliver Wellnitz, and Lars Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7, New York, NY, USA, 2005. ACM.

[9] Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. CAN mobile gaming be improved? In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, pages 44–47, New York, NY, USA, 2006. ACM.

[10] Jan Krikke. Samurai Romanesque, J2ME, and the Battle for Mobile Cyberspace. *IEEE Computer Graphics and Applications*, 23(1):16–23, 2003.

[11] mDisney Studios. Pirates of the Caribbean Multiplayer Game. Web: http://disney.go.com/disneymobile/mdisney/pirates/, 2007.

[12] Y. S. Ng. Designing Fast-Action Games for the Internet. In *Gamasutra*, September 1997.

[13] L. Pantel and L. C. Wolf. On the Impact of Delay on Real-Time Multiplayer Games. In *Systems Support for Digital Audio and Video*, May 2002.

[14] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in Warcraft III. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 3–14, New York, NY, USA, 2003. ACM.

[15] Alf Inge Wang, Anne Marte Hjemås, Martin Jarrett, and Eivind Sorteberg. Performance of Mobile Multiplayer Real-time Games over Wireless Networks. In *Submitted to 9th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2008.