MicroPlay: A Networking Framework for Local Multiplayer Games

Anh Le[1], Lorenzo Keller[2], Christina Fragouli[2], Athina Markopoulou[1]
[1] UC Irvine, USA
[2] EPFL, Lausanne, CH
{*anh.le, athina*}*@uci.edu*, {*lorenzo.keller, christina.fragouli*}*@epfl.ch*

ABSTRACT

Smartphones are an ideal platform for local multiplayer games, thanks to their computational and networking capabilities as well as their popularity and portability. However, existing game engines do not exploit the locality of players to improve game latency. In this paper, we propose MicroPlay, a complete networking framework for local multiplayer mobile games. To the best of our knowledge, this is the first framework that exploits local connections between smartphones, and in particular, the broadcast nature of the wireless medium, to provide smooth, accurate rendering of all players with two desired properties. First, it performs direct-input rendering (*i.e.*, without any inter- or extrapolation of game state) for all players; second, it provides very low game latency. We implement a MicroPlay prototype on Android phones, as well as an example multiplayer racing game, called Racer, in order to demonstrate MicroPlay's capabilities. Our experiments show that cars can be rendered smoothly, without any prediction of state, and with only 20–40 ms game latency.

1. INTRODUCTION

Smartphone ownership has seen tremendous rate of growth in recent years. According to the latest report by Nielsen [1], it has more than doubled in just two years. One of the fastest growing sector of smartphone applications is mobile games. In fact, mobile games consistently dominate the list of top applications downloaded in both Android and iOS markets. The popularity of mobile games have led to innovations in both hardware and software, *e.g.*, from the introduction of mobile quad-core CPUs to 3D sensor-based games [2].

Smartphones offer opportunities for mobile game players to engage in highly interactive, local multi-player game activities, also known as LAN parties. These activities offer high levels of satisfaction and entertainment to the participants, as they allow them to interact with each other on the spot, while playing the games. With a traditional gaming platform, such as PCs, gamers have to make a significant effort to setup a LAN game party, *e.g.*, planning the time and arranging a powered table space. In contrast, smartphones are usually carried by the owners, operate on battery, and have small footprints, which make them an ideal gaming platform for spontaneous, local multiplayer games.

One of the main challenges when developing multiplayer games is how to reduce *game latency*, *i.e.*, the delay between when one of the players inputs a command and when the results of the command are rendered on the screens of all players. Local multiplayer games on smartphones have two important characteristics: (i) the phones are connected via a common wireless network (*e.g.*, WiFi) and (ii) the players are typically close to each other. In this work, we exploit, for the first time, these two characteristics to significantly reduce game latency, at the same time, simplifying multiplayer games development. In particular, we eliminate the need for game state prediction.

Our contribution is the design and evaluation of a novel, comprehensive networking framework, MicroPlay, which aims at assisting mobile game developers in building highly interactive local multiplayer games.

Key Design Aspects and Benefits. We design MicroPlay so as to explicitly use local connections between smartphones and exploit overhearing over the shared wireless medium (WiFi). In particular, we host the games locally, *i.e.*, one of the phones acts as a server and the others act as clients. We also exploit, for the first time, that packets sent to the server can be overheard by other players. Using the overheard packets, the local game engine of each player can render the movements of the other players precisely, without the need to make predictions about the other players' movements. This design brings benefits both in terms of performance (*low game latency*, sufficient even for games with the most stringent delay requirements, such as first person shooter (FPS) and racing games) and *simplified game development* (unlike other engines [3, 4, 5, 6], it does not rely on game state prediction). Furthermore, it enables *spontaneous setup* of local multiplayer games. Indeed, MicroPlay provides a user interface that enables phones to easily setup a local network and establish all the necessary connections to play a game on-the-fly, without the need of existing network infrastructure, such as local access point or Internet connectivity. This significantly brings down the overhead of setting up multiplayer games.

Android Prototype and Example Application. We implemented MicroPlay in Java and C from scratch. To demonstrate the functionality of MicroPlay, we also developed an example multiplayer racing game called

Racer based on AndEngine [7], an open-source Android game engine. We compare the performance of Racer over MicroPlay against Racer over the vanilla multiplayer networking functionality provided by AndEngine. Our experiments show that MicroPlay is able to smoothly render car movements without any prediction code. Furthermore, MicroPlay is able to achieve very low game latency (20–40 ms compared to, *e.g.*, 100 ms when the server is hosted remotely), even when there are many (up to 6) players . We plan to make the source code of both MicroPlay and Racer publicly available.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we describe the MicroPlay networking model and the benefits of overhearing. In Section 4, we describe the architecture and implementation of MicroPlay on Android phones. In Section 5, we evaluate the performance of MicroPlay over an example game, Racer. Finally, in Section 6, we conclude the paper.

2. RELATED WORK

The strategies used to synchronize game state in multiplayer games can be divided in two groups: client-server and peer-to-peer. In the first approach, players synchronize their local game states with the authoritative state held by the server. This is the approach used by many commercial engines [3, 4, 5, 6] and taken by our framework. In the peer-to-peer approach, no single peer is responsible for the game state. Instead, the state is synchronized using a distributed protocol run by all players [8]. The advantages of the client-server model are that it is easier to implement and more resilient to cheating. The appeal of the second approach is that it obviates the need of expensive servers and no node in the network must be present for the duration of the game. There are also efforts to build peer-to-peer based games that are resilient to cheating [9].

The recent popularity and powerful capabilities of smartphones have enabled development of multiplayer games over wireless networks. Current commercial engines essentially re-use the approaches designed for the wired setup over wireless [6]. Highly interactive multiplayer games, such as FPS, have been implemented both over LAN and wide-area networks (WAN); the latter have much larger network latency. Commercial solutions, such as [3, 4, 5, 6], employ mechanisms that can cope with this large latency over WANs, and simply reuse these mechanisms over the lower latency LANs as well. In the academic community, [10] has proposed solutions to deal with the high latency of the wireless WAN links and [11] studied the performance of various broadcasting schemes, such as flooding and distance-based broadcast. Multicast has also been proposed in the literature [8, 11] to distribute game commands and states, but as far as we know, has not been used in commercial systems.

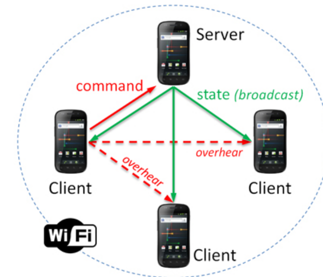

Figure 1: MicroPlay networking model.

The effect of network latency on players in various classes of games have been studied in [12]. A matchmaking system that assigns players to games such that the latency requirement is satisfied is proposed in [10]. Techniques to ensure a smooth gameplay can be grouped in three categories: client prediction [13], interpolation [13], and extrapolation (also called dead-reckoning) [14, 13], and are described in the next section.

In our recent work [15], we developed MicroCast – a framework for collaborative video streaming on mobile devices. MicroPlay is similar to MicroCast in that it exploits local connections and overhearing, for collaboration between devices within proximity of each other. However, the intended application for MicroCast is video streaming, and the objective is to maximize the common download rate. In contrast, MicroPlay's application scenario is multiplayer games and the objective is to deliver game-related packets, which are much smaller than video packets, to all players with minimum latency.

3. NETWORKING MODEL

In this section, we first describe the standard networking model for multiplayer games and the terminology commonly used. Then, we describe how overheard packets can improve both game rendering and latency.

3.1 Networking Model and Terminology

MicroPlay adopts the popular client-server model that is currently used by the majority of commercial game engines [3, 4, 5, 6]. In this model, one of the smartphones acts as the server and the rest act as clients. Both the server and the clients participate in the game. The server is responsible for updating and distributing the game state. To this end, the server and clients communicate with each other using a local Wifi network.

When a client inputs a command, such as "move up," this command is put into a command packet and sent to the server. The server processes the command and updates the game state. Then the server sends the updated game state, *i.e.*, snapshot of the game, in a state packet to all the clients. Afterwards, the clients render the new game state. The command packets are sent by the client very frequently, every 12–20 ms. The state packets are sent less frequently, every 50–80 ms, as illustrated in Fig. 2 (left).

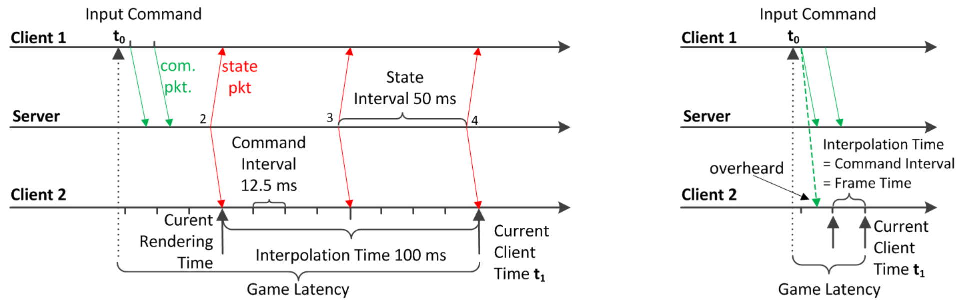

Figure 2: Game latency in a regular multiplayer networking framework (left) and in MicroPlay (right).

If a client rendered the game based only on the discrete game states sent by the server, moving objects would look choppy and jittery. Also, missing state packets would cause noticeable glitches. To address these problems, the game at each client buffers the received states and renders states that are slightly in the past. Fig. 2 (left) illustrates this technique: the client has received up to snapshot 4; however, it has only rendered up to snapshot 2 and still buffers 3 and 4.

With this buffering technique, first, animation of objects between game states can be smoothly rendered by *interpolating* the state of the objects between the latest rendered game state and the immediate next game state, *e.g.*, snapshots 2 and 3 in Fig. 2 (left). Second, loss of a single game state update can be tolerated. For instance, if snapshot 3 is lost, the client can still render the game by interpolating between snapshots 2 and 4. The interpolation time is usually chosen large enough to include two snapshots. Furthermore, when no information about an existing object is available in the buffered snapshots, the client will try to *extrapolate* the object state, *i.e.*, predicting its future state, using the currently known information (*e.g.*, velocity and position).

However, buffering also causes a constant *game latency* (defined in Section 1), as shown in Fig. 2 (left). The delayed visual feedback due to game latency makes it hard to move or aim precisely. To overcome this, each player's local game engine adopts a technique called *input prediction*, where the client predicts the results of its own user commands and renders them immediately. We will refer to this as *direct-input rendering*. This makes the game feel more interactive. For example, in Fig. 2 (left), although the rendering time only includes state 2, the player updates its own state using its own input commands up to the current time. With direct-input rendering, the client does not have to interpolate its own player since exact commands are executed. Thus, the rendering of the local player is accurate. Direct-input rendering, however, is not possible for other players, and the local game engine has to resort to inter- or extrapolation to render them.

3.2 Benefits of Overhearing

A key novelty of MicroPlay is that it exploits the nature broadcast of WiFi to enable direct-input rendering for all players. In particular, MicroPlay allows a client to overhear command packets sent by all other clients to the server and use them to perform direct-input rendering. Fig. 1 illustrates the MicroPlay networking model with overhearing enabled. Fig. 2 (right) illustrates how a client (client 2) performs direct-input rendering for objects of another client (client 1). There are two key benefits with this approach.

Precise rendering. When direct-input rendering is available for all players, the rendering of other players and their objects can be done similar to the local ones. Hence, there is no longer need for performing inter- or extrapolation. Note that direct-input rendering is strictly more accurate than rendering using these predictions. The elimination of state prediction code also brings many side benefits: it simplifies game development (simpler and more maintainable objects' update code), reduces processing needed at each phone, and improves scalability when the number of players or interactive objects increases.

Low game latency. The interpolation time could be eliminated since (i) no interpolation is needed and (ii) the loss of state packets is not as critical as before, thanks to the overheard command packets. In MicroPlay, we reduce the interpolation time to the average in-game frame rendering time, 20 ms (50 frames per sec.), as illustrated in Fig. 2 (right).

4. MICROPLAY IMPLEMENTATION

4.1 Architecture

Fig. 3 shows the architecture of an Android game built using AndEngine and MicroPlay.

4.1.1 MicroPlay Components

These are provided by MicroPlay and are depicted in light gray in Fig. 3.

Connection module: Phones participating in the game are organized in a 802.11 basic service set (BSS). One of the phones, the server, acts as an access point while the other phones act as clients and connect to it. The connection module takes care of setting up this network and configuring the IP connectivity when the game starts.

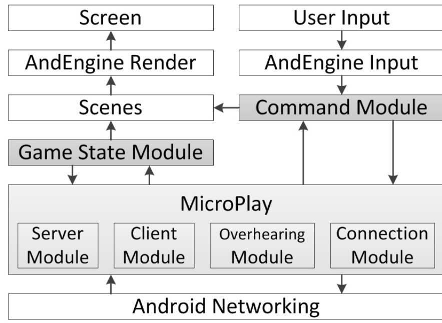

Figure 3: Architecture of an Android game based on AndEngine and MicroPlay. Components in light gray are provided by MicroPlay. Components in dark gray are implemented by the game developer. The rest are provided by Android and AndEngine.

The server can broadcast frames to all clients; clients can only unicast frames to the server. Each client maintains a TCP connection to the server, used as a reliable control channel, and also listens for UDP packets. The server broadcasts UDP messages using IP broadcast; the clients unicast UDP packets to the server.

Overhearing module: This module provides an API to let each client overhear UDP command packets sent by all other clients to the server.

Server module: This module runs on the server. It periodically takes snapshots of the game and sends them in state packets to all clients using IP broadcast. It detects client disconnection as well as accepts new client connections.

Client module: This module runs on the server (as the server is also a player) and all clients. It periodically samples input commands of the local player and sends these commands in command packets to the server. It parses overheard packets to obtain command packets.

4.1.2 Developer Components

These are the components that must be provided by the game developer and are in dark gray in Fig. 3.

Command module: This module serializes commands performed by the local user to a binary representation that is then put in command packets and sent by the client module of MicroPlay. It also parses the serialized commands contained in the command packets received from MicroPlay. Finally, it takes care of applying the commands generated by the local player and overheard from the other players to the local game scene that is rendered on screen. Notice that command packets contain a generation time and are applied to the local scene only when the rendering time reaches it.

Game state module: On the server, this module serializes the current local scene to a binary representation that is then sent in state packets by MicroPlay. On a client, it updates the local scene using the content of state packets received from MicroPlay. Every state packet is accompanied with a timestamp that indicates when it has been created. Similar to a command, a game state is only processed when the rendering time reaches it.

Furthermore, the game state module only applies a game state if the current scene, which is the result of applying the overheard command packets, is significantly different from the received game state. We call this situation a *sync error*. This error may come from a variety of sources, such as missing command packets or diverging results of physics simulation on the server and the clients due to difference of system clocks. The game developer decides when to correct the local scene based on the type of games since in many games, minor errors can be tolerated.

4.2 Implementation Details

Connection module. If the player wants to start a new game session, the connection module takes care of starting a software-based access point (soft AP) and a DHCP server. This is done using non-public APIs provided by the Android `WiFiManager`. Once the local AP starts, the module opens a UDP socket that is subsequently used to broadcast state packets and to receive command packets. It also binds a TCP socket to receive connections from clients. If the player wants to join an existing game, the standard API of `WiFiManager` is used to discover the BSS created by the server, to connect to it and to obtain an IP address. After acquiring an IP address, the module establishes a TCP connection to the server. The TCP channel is used to transmit control packets that notify clients of connections and disconnections of players.

Overhearing module. To enable overhearing of UDP packets, this module uses a technique developed in [15]: it opens a raw socket and sets the network interface into promiscuous mode by running an external executable daemon called `overhearingd` as root. Overhearing in MicroPlay is necessary because when connected to an AP, the clients cannot broadcast frames, they have to first send them to the to the AP that will then broadcast them. Therefore, without overhearing, it is not possible to achieve a direct client-to-all communication.

The `overhearingd` daemon is implemented in C. It filters packets received from the raw socket and delivers the relevant packets to the overhearing module using a named pipe. The overhearing module contains a JNI library that wraps the system call `select` to perform non-blocking reads from the named pipe, which is not possible using only the Android Java API.

Client and server modules. These modules are implemented as update handlers of AndEngine. Update

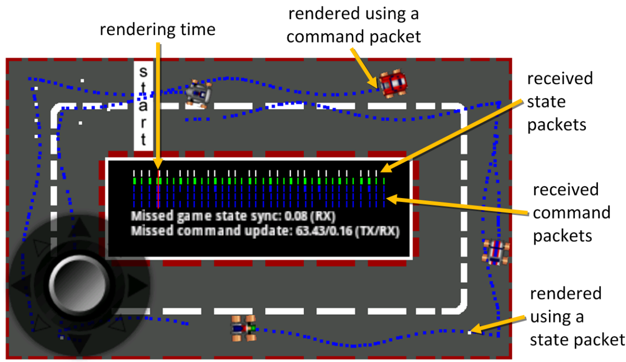

Figure 4: A Racer game session with 4 players.

handlers are called once per screen redraw. This single thread approach was chosen to tightly control the scheduling of network operations. Since the handler code is run in the main game rendering thread, it cannot block. For this reason, all I/O operations are performed using non-blocking calls provided by the Android NIO API. We note that both client and server modules could be ported to other game engines as independent threads.

Time synchronization. The authoritative game time is the server time. Clients synchronize their game time based on the time indicated in the periodic state packets sent by the server. This approach does not take into account the one-way trip time between the server and the clients, which is negligible (a few ms) in our setting. With this approach, sporadic time synchronization errors due to transient network conditions and systematic errors due to differences in frequency of the client and server clocks are quickly fixed.

5. EVALUATION

5.1 Racer: A multiplayer racing game

In order to evaluate MicroPlay, we implemented a multiplayer racing game called Racer using AndEngine. Racer serves as an example application to demonstrate key properties of the underlying MicroPlay. However, we envision that MicroPlay will be used to support many different games that can benefit from its capabilities. Fig. 4 depicts an annotated screenshot of Racer. In a game session, several players race their cars around a terrain; each player controls her own car using an on-screen analog joystick (at the bottom left).

In the implementation, the command module samples an input command by recording the current position of the joystick, which is just its x and y positions. It applies a command by adjusting the velocity and angle of the car based on the pair (x, y). The game state module takes a game snapshot by recording the positions, velocities, and angles of all cars. It applies a snapshot by updating all the players cars from the recorded values.

In Racer, we keep track and plot all the received state and command packets. Whenever a car state is updated as a result of applying a command, we draw a blue dot on the screen, at the current position of the car. On the other hand, whenever a car state is updated as a result of applying a game state, we draw a white dot. In this way, we keep track of how the cars are rendered.

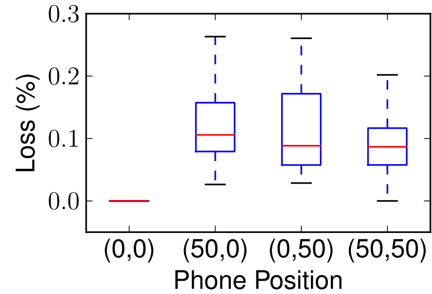

Figure 5: Loss percentage of command packets when 4 phones are arranged at 4 corners of a 50-cm-side square.

5.2 AndEngine Multiplayer Extension

AndEngine provides a multiplayer extension of its own, which we refer to as AndExt, to support development of multiplayer games. This extension facilitates client-server connection establishment, including automatic server discovery. All players must join an existing WiFi network, and use unicast TCP connections to send all packets from clients to the server and vice versa. As a baseline for comparison, we adapt Racer so that it works with AndExt; *i.e.*, when using AndExt, all packets of Racer are sent using unicast sessions over TCP.

5.3 Performance Evaluation

We evaluate MicroPlay by running multiple sessions of Racer, with up to 6 players. The command sampling interval is set equal to the redrawing interval (frame rate) of the game, at 20–30 ms, and the state sending interval is set at 80 ms. For the box plots in Fig. 5 and 6, the box upper and lower bounds are the first and third quartiles, and lines in the boxes are the medians.

Overhearing quality. MicroPlay relies on the assumption that every client can overhear command packets sent by the others. In order to verify this assumption, we measure the percentage of command packets that each node cannot receive. To detect packet loss, we attached a sequence number to every command packet. We ran Racer for 20 minutes, and we measured the packet loss every minute. The experiments were conducted in a residential area with a high number of co-located WiFi networks.

Fig. 5 shows the percentage of command lost, averaged over all players when 4 players are sitting around a table. Phones are positioned in a square whose side is 50 cm. The node at position $(0, 0)$ is the server. In this experiment, the server does not lose any command packet since all players send command packets to the

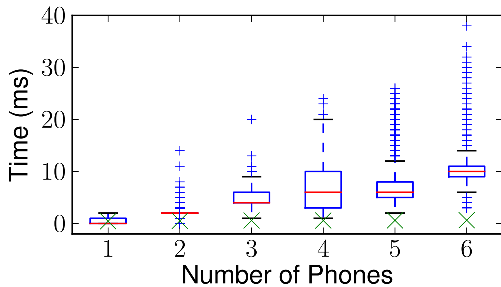

Figure 6: Time the server spends to send a state packet to all clients using AndExt. X denotes the performance of MicroPlay.

server as UDP unicast packets and therefore they are retransmitted multiple times if the server does not ACKs them. Other phones lose some of the updates but no more than 0.25% of them.

Since missing command packets can cause sync error (as discussed in Section 4.1.2), we also record the number of times that state packets were used to resync game state. A sync error in Racer is when a car position is 10 pixels off of its correct position. Our measurement shows that on average, less than 3% of state packets were used for resyncing. This shows that majority of the time, it is sufficient to render all players using just the (overheard) command packets.

In another experiment, we looked at the effect of distance between the clients and the server. We put the phones on a line with distance between two consecutive phones of 8 cm. The phone at the beginning of the line is the server. In this experiment, we observed that the loss rate increases with the distance from the server. Although the loss rate was higher than in the previous experiment, the upper quartile was still below 1%.

Server processing latency. A significant advantage of our approach is that we use broadcasting to efficiently disseminate state packets to the clients. To the best of our knowledge, this is something that all popular game engines, such as [3, 5, 4, 6] and AndExt, currently do not exploit (even the few ones that use a local server). Fig. 6 shows the time needed to send one state packet as the number of players increases. We observe that the median time increases with the number of players when AndExt is used, but stays flat when MicroPlay is used (less than 2 ms to broadcast the state packet).

Game latency. We also observed in our experiments of up to 6 phones that players are rendered at the rate at which their inputs are sampled, which equals the game's frame rate (at 20–40 ms for 25–50 frame per sec.). This latency is much smaller than the latency typically needed for highly interactive multiplayer games (∼100 ms). We note that this smooth rendering is possible *without the need of any prediction* (interpolation or extrapolation) code.

6. CONCLUSION

We present MicroPlay, a novel networking framework for local multiplayer games. MicroPlay exploits, for the first time, the broadcast nature of the wireless medium to allow for smooth, accurate rendering of all players without the need of inter- or extrapolation techniques. MicroPlay achieves very low game latency, sufficient to support games with the most stringent delay requirement, such as FPS and racing games. We demonstrate these properties of MicroPlay using a real racing game, called Racer, that we developed. We plan to make the code of both MicroPlay and Racer publicly available.

7. REFERENCES

[1] Nielsen. The mobile media report: State of the media, q3 2011. http://goo.gl/5Cv6S. [Online; accessed 13-mar-2012].

[2] Z. Zhang, J. Qiu, D. Chu, and T. Moscibroda. Demo: Sword fight with smartphones. In *Proc. of ACM SenSys*, 2011.

[3] Valve Inc. Source multiplayer networking, developer guide. http://goo.gl/sK7Tz. [Online; accessed 13-mar-2012].

[4] EpicGames Inc. Unreal networking architecture. developer guide. http://goo.gl/B7CmI. [Online; accessed 13-mar-2012].

[5] F. Sanglard. Quake engine code review: Prediction, blog. http://goo.gl/Nnvej. [Online; accessed 13-mar-2012].

[6] Unity Technology. State synchronization details, developer guide. http://goo.gl/YFDSn. [Online; accessed 13-mar-2012].

[7] Android game engine. http://www.andengine.org/. [Online; accessed 13-mar-2012].

[8] L. Gautier and C. Diot. Design and evaluation of MiMaze a multi-player game on the internet. In *Proc. of IEEE Conference on Multimedia Computing and Systems, 1998.*, pages 233–236., July 1998.

[9] N. E. Baughman, M. Liberatore, and B. N. Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions on Networking*, 15(1):1–13, Feb. 2007.

[10] J. Manweiler, S. Agarwal, M. Zhang, R. Roy Choudhury, and P. Bahl. Switchboard: a matchmaking system for multiplayer mobile games. In *Proc. of ACM MobiSys*, pages 71–84., 2011.

[11] A. Sardouk, S. M. Senouci, N. Achir, and K. Boussetta. Assessment of MANET broadcast schemes in the application context of multiplayer video games. In *Proc. of ACM SIGCOMM NetGames Workshop*, pages 55–60, New York, NY, USA, 2007.

[12] M. Claypool and K. Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, Nov. 2006.

[13] Y. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, 2001.

[14] E. J. Berglund and D. R. Cheriton. Amaze: A multiplayer computer game. *IEEE Software*, 2(3):30–39, May 1985.

[15] L. Keller, A. Le, B. Cici, H. Seferoglu, C. Fragouli, and A. Markopoulou. Microcast: Cooperative video streaming on smartphones. *Accepted to ACM MobiSys*, 2012.