VFC-reckon for Android Mobile Devices

Mário Santos, Luís Veiga, Paulo Ferreira
INESC-ID, Instituto Superior Técnico, Lisbon, Portugal
marios.santos@ist.utl.pt, [luis.veiga, paulo.ferreira]@inesc-id.pt

Keywords: Mobile Devices, Ad-Hoc Network, Consistency, Multiplayer Games, Interest Management, DeadReckoning

Abstract: Nowadays, mobile phones/tabs/netbooks are an essential part of our life and it is hard to find someone who does not have such a device. These devices are used to work, play games, listen to music, navigate on the internet, etc... Multiplayer games to such devices, while interesting and fun to play, raise serious scalability and playability difficulties as they require a massive communication flow between them, needed to maintain the game state consistent between players. The goal of this work is to raise the game scalability by reducing the number of messages exchanged between the devices which will reduce the bandwidth and CPU usage. Our work consists in the conception and development of a consistency model named VFC-reckon based on Interest Management and Dead Reckoning techniques. VFC-reckon allows us to reduce the number of messages exchanged between the devices without affecting the game playability. We developed the system for Android portable devices along with a distributed game that shows the advantages of the VFC-reckon model for ad-hoc networks.

1 INTRODUCTION

In the last years mobile phones/tabs/netbooks have acquired an important role in our society. Almost everyone has at least one such device which can be used either as a multimedia platform or as a work tool.

An interesting class of applications for such devices is multiplayer games in ad-hoc networks. In order to provide a shared sense of space among players, each player has a copy of the (relevant) game state on his device. When a player performs an action, the game state of all other players affected by that action must be updated. The simplest approach is for each player to maintain a full copy of the game state and all players broadcast updates to all other players. The disadvantage of this approach is that it does not scale: as the number of players increases, the number of messages sent over the network and to be processed by each client increases as well.

Interest management is a technique to select the updates that are relevant (or not) to a player based on multiple criterias. The most well known Interest Management criterias are Aura Of Interest[2], based on distances between objects, Line Of Sight[3] wich is based on the player line of sight and RegionParition[4] based on map division.

Dead Reckoning[6] is a technique that masks network latency by estimating future game events. With this technique we can reduce delay between messages as well as the their loss. The next object position is based on his current position and the collection of past positions.

Current solutions[1, 5, 3], regarding interest management techniques ignore mobile devices limitations such as bandwidth, CPU and battery consumption. In addition, most are focused on one type of game and are inflexible, making the programming of other applications a very hard and time consumption task.

The goal of this work is to develop a consistency model that raises the game scalability through the reduction in the number of messages among players. This reduction is done while ensuring an acceptable degree of consistency, i.e. without hindering game playability. For this purpose, we developed the VFC-reckon consistency model; this is based on the VFC[7] (Vector Field Consistency) consistency model improved with Dead Reckoning[6] techniques.

Our system is implemented as a middleware for the Android platform that handles multi-player game object consistency. The Asteroids game [1] was ported to VFC-reckon on Android mobile devices in order to test the VFC-reckon playability impact. The performance results obtained show that VFC-reckon significantly reduces the network traffic by exchanging less messages and, consequently, CPU and bandwidth usage.

2 VFC-RECKON CONSISTENCY MODEL

As aalready said, the VFC-reckon consistency model integrates VFC along with the application of

[1] http://www.goriya.com/flash/asteroids/asteroids.shtml

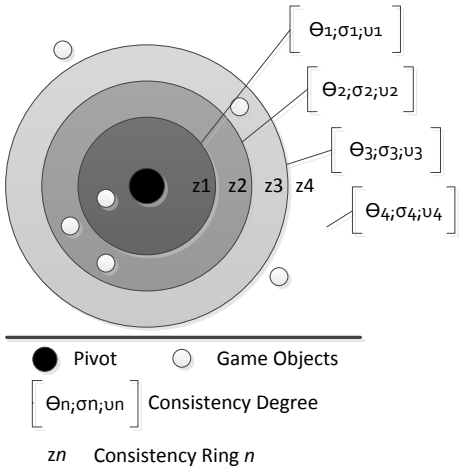

Figure 1: VFC example with three consistency rings and one pivot.

DeadReckoning techniques. In our system the VFC role is to select the updates that should be sent to each player while DeadReckoning techniques are used to mask the lack of messages.

2.1 VFC

VFC[7] is a consistency model based on Interest Management[2] techniques. VFC dynamically adjusts the consistency of replicated objects, based on the current game state; it manages the degree of consistency of each object based on its distance to special game entities. With VFC we can reduce the number of messages between players in multiplayer games providing a reduction in network bandwidth and a less CPU load.

The VFC consistency model is based on two main concepts: Consistency Rings, and Consistency Degrees, described now.

2.1.1 Consistency Rings

In VFC, each player has a local view of the virtual game map and entities. Within each local view there are special game entities called pivots; consistency degree is defined around pivots for all the objects that surround them. A pivot can be any game entity like, such as the player Avatar[2]. Consistency rings are formed around a pivot; each ring defines a consistency degree that is applied to all game entities in that ring.

[2]Entity that represents the player in the game

Fig. 1 illustrates three consistency rings around a pivot object. The color intensity in the rings represents the consistency degree in that ring: the object located in *z1* ring will have a greater consistency degree then the other two objects located in the outer rings (*z2* and others). The *z4* represents the area outside the consistency rings. In this example, we only consider three rings but the VFC model allows the definition of any number of consistency rings.

VFC supports any number of dimensions. In our previous example we showed a bi-dimensional space, with 2-dimension rings, but we can use any arbitrary number of *D* dimensions like, for example, three dimensions in which case the rings would become spheres.

2.1.2 Consistency Degrees

Each consistency ring has a consistency degree associated with it. A consistency degree is a 3-dimensional vector that specifies the consistency deviation limits for objects within that ring. The 3-dimensional vector parameters are: Temporal (θ), Sequential (σ) and Value (υ).

Temporal Dimension θ specifies the maximum time a replica can be without being refreshed with the most recent value. In other words, this dimension specifies the maximum amount of time (seconds) that a replica of an object can remain without being updated.

Sequential Dimension σ specifies the maximum number of lost replica updates. With this dimension we guarantee that a replica of an object is outdated in a maximum of σ updates w.r.t. the original object.

Value Dimension υ specifies the maximum relative difference between a replica and its master contents. This dimension needs a function to calculate the difference, in percentage, between two objects which makes it dependent on the game implementation.

The multiple parameters in each consistency degree makes the VFC flexible, allowing to easily adapt the model to different game logics.

2.1.3 VFC specification

VFC consistency model corresponds to the set of all game objects, pivots, consistency rings and consistency degrees. From the programmer's viewpoint, to specify the entire model associated with any multiplayer game, he just specifies these four sets. The aggregation of these four sets corresponds to the *phi* specification.

2.2 Dead Reckoning

DeadReckoning techniques [6] are frequently used in multiplayer games due to its simplicity and gains. These techniques are commonly used as a method of reducing the latency impact by anticipating future object positions based on the current position and history of old positions. In this work the use of DeadReckoning is not to reduce the impact of latency, but to mask the lack of update messages.

Normally, a game requiresa rate of 25fps[3], which means that, each second, the object moves, at maximum, 25 times. To achieve this rate on a multiplayer game, messages between players must be sent within a 40ms interval. Using Dead Reckoning technique we increased this interval to 160ms without affecting, in a noticeable way, the game playability (from the point of view of the player).

To demonstrate the DeadReckoning estimation process we show in Fig. 2 an example of the prediction of three positions, according to a linear trajectory, where the interval between updates is 160ms. To start the estimation process we need at least two real points, in this case, $P1$ and $P2$. The estimated points $Pe(nv)$ in the figure are calculated from the real points Pn and $P(n-1)$. To calculate the position of a Pe is necessary to know the number of estimated points Pe between two Pn points. With the number of Pe between each Pn ($\#Pe$) we can calculate the distance between each Pe from the formula $\frac{\Delta v}{(\#Pe+1)}$ where Δv is the distance between two real points like $P1$ and $P2$. Finally, to discover the estimated point $Pe(nv)$ position, we simply add the offset $\frac{\Delta v}{(\#Pe+1)}$ to each position.

The critical aspect of DeadReckoning is the function used to predict an object's position. Although DeadReckoning is able to estimate any number of points we need to keep in mind that these are estimated values. In any estimated value there is always a difference between the estimated and actual value. In this case, the estimation error leads to weird object movements.

In this work we developed two types of prediction engines: a linear engine, used when an object has a linear trajectory, and a circular engine used when an object has a circular trajectory.

2.3 VFC-reckon

VFC-reckon combines the VFC consistency model and DeadReckoning prediction techniques. The number of points, estimated by the prediction engine, are defined in the VFC-reckon model by simply adding a new consistency parameter, called *estimationPoints*, into VFC Consistency Degrees. Now, the specification of each consistency degree, has the following parameters: (θ), (σ), (υ), and *estimationPoints*.

Basically, to use the DeadReckoning techniques, the programmer specifies how many points should be estimated in each ring between each server message. For example, if we specify 3 *estimationPoints* for a consistency degree this means that, within each message received by a client, the system estimates 3 points for objects inside that ring.

[3]25 frames-per-second corresponds to the minimum update rate according to human eye

3 VFC-RECKON ARCHITECTURE

VFC-reckon architecture is client-server based (shown in Fig. 3) and serves as a middleware to multiplayer games for the Android Platform. Using the VFC-reckon middleware, the programmer just specifies the consistency model specification *phi* according to the game logic.

It's the server that applies the VFC-reckon model. This makes it simple to implement and run making all the information needed by the model to be available in the server node. The client-server protocol is implemented in the Session Manager component. The server component has the responsibility to process update requests from clients and propagate the updates to clients selected by the VFC-reckon consistency model. Communication between client and server is implemented at the Communications Layer following a star-type topology.

3.1 Reading and Writing objects

Game objects shared between clients are called primary objects and replicas. The primary objects are stored on the Primary Object Pool in the server side, while the replica objects are stored in the Replica Object Pool and are local to each client. The server performs periodic rounds; on each round, it selects and sends, based on the VFC-reckon model, the relevant updates to clients so that they can update their replicas. Both Write and Read operations are applied to local replicas. Writes to local replicas are propagated later in the background to the server.

3.2 Updates propagation

There are two types of updates propagated in our system: Server to Client and Client to Server.

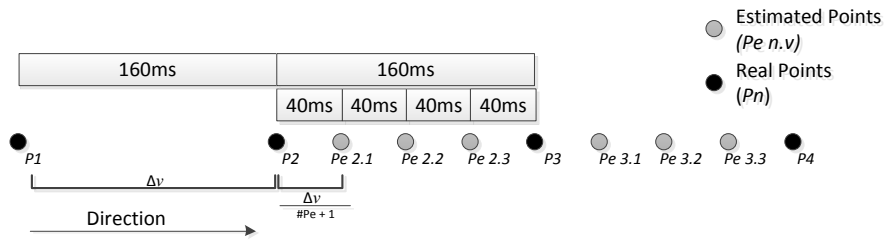

Figure 2: DeadReckoning linear trajectory example.

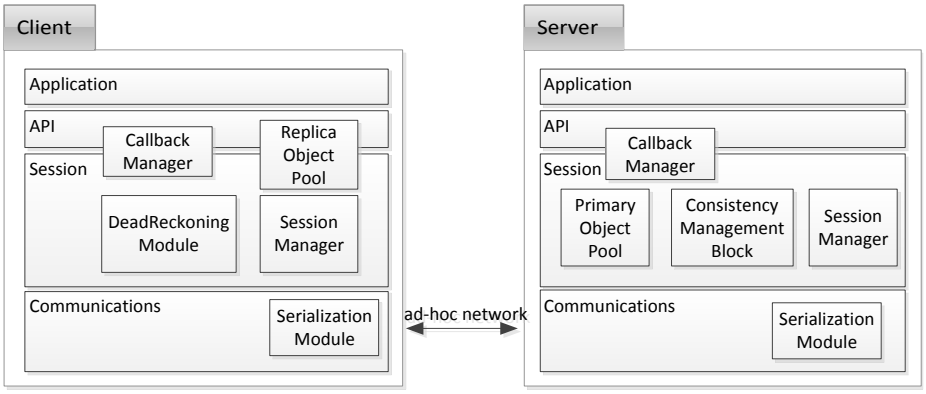

Figure 3: VFC-reckon middleware architecture

3.2.1 Update Propagation - Server to Client

For Server to Client updates propagation, the Consistency Management Block applies a concept of periodic round. Basically, the Consistency Manager Objects periodically applies the VFC-reckon model, in order to send to clients their selected updates.

Thus, at each round, the server selects the updates that must be sent to each client, based on the VFC-reckon *phi* specification. After this selection, the server sends messages (to the concerned clients) with all updates. When a client receives such messages (called round messages), it updates its local replicas at the Replica Object Pool.

By receiving periodic round messages from the Server, the VFC-reckon middleware at the client may notify the application through callback mechanisms implemented by the Callback Manager. The notifications inform the application that the game state has been changed, and based on that, the application can perform several actions, such as, update the player's score on the screen.

3.2.2 Update Propagation - Client to Server

Sending updates (from clients) to the server each time an update occurs (locally on a client) would be a bad choice due to the amount of updates in each game cycle[4]. This would cause the client to send N messages, with N equal to the number of objects updated in that game cycle. A more efficient way is to periodically propagate a client updates to the server (similarly to the server's round already described). Updates are grouped in one message and periodically sent to the server in background, avoiding interfere with the game cycle.

3.3 Client and Server Session Manager

The Client and Server Session Manager are implemented according to a state machine. Due to space constraints it is not possible to fully explain all the

[4]Normaly, a game cycle is a repetition of updating game objects physical components and drawing entities at the screen.

details of the Session Manager state machines.

Briefly, the state machine can be divided into two phases: *setup* phase and *active* phase. In the *setup* phase the client registers and sends the necessary game specifications like the *phi* specification and shared objects. The *active* phase occurs during the game execution and it's when the server periodic rounds take place. In the *active* phase the client can still publish new objects or remove shared objects.

3.4 VFC-reckon application

The application of VFC-reckon model depends on two main modules: Consistency Management Block (CMB) and the DeadReckoning Module.

Rounds are performed by the CMB at the server. The VFC-reckon model is applied, at each round, by selecting the updates that should be sent to clients based on the consistency model specification *phi* defined (by the clients) during the game *setup* phase.

The DeadReckoning module exists only on the client side and has the responsibility of estimating objects new positions based on their current positions and historical positions that have been stored from the server.

3.5 Message Serialization

The serialization module in VFC-reckon has a great performance impact. With a high transmission rate between clients, the operation of serializing messages can easily become the system bottleneck. Java default serialization demonstrates to be a bad solution with an high transmission rate between players.

The VFC-reckon serialization module, that we developed, serializes the messages into arrays of bytes and delivers them to the communication layer. To identify the object types inside a message we specify the object class type using integers instead of strings like Java does. This choice speeds up messages serialization and deserialization operations; it also makes messages much smaller.

4 IMPLEMENTATION

The VFC-reckon system, and a demonstrative game were developed in the platform Android 2.2 Froyo (but are compatible with later versions).

The system supports communication through the WiFi protocol with TCP/IP sockets as well as via the Bluetooth protocol with Bluetooth sockets. Only these two communication protocols are currently supported but others can be easily suported as well given the extensibility of the VFC-reckon implementation.

To use the DeadReckoning module, game objects must specify a trajectory type. Each trajectory is associated with a PredictionEngine which is specified, in game objects, invoking a method named *setPredictionEngine()*. The programmer uses as argument the type of engine he wishes. During the game execution the engine can be switched invoking the *setPredictionEngine()* method. Regarding new PredictionEngine types, our system is extensible enough so that the programmer can easily create his own (based the game's logic). The new engine is created simply by extending a class named PredictionEngine and implementing the prediction method.

VFC-reckon is parameterized using Java Annotations. To define the *phi* specification the programmer must annotate a game class with *@PhiAnnotation* and set the VFC-reckon parameters such as Consistency Degrees, Consistency Rings and Rings dimensions.

Like *phi*, the specification of pivots is made through a simple annotation. The programmer annotates the game object that he wants to become a pivot with the annotation *@PivotAnnotion*.

5 EVALUATION

To evaluate the VFC-reckon system we developed a distributed game (based on the well known Asteroids game[5]). The map in which spaceships and asteroids move spans several mobile devices. On each player screen, we can see other players spaceships and asteroids as long as they are close enough in the overall map.

The tests were performed on 2 Samsung Galaxy tablets with 512MB of RAM and a 1GHz CPU each, using Bluetooth as the communication protocol.

To evaluate the system we used three types of consistency models: Basic, VFC and VFC-reckon:

- In the Basic model there is no update selection. The server simply broadcasts all the updates to all the clients.
- In the VFC model, the server selects the updates to send to each client based on *phi* specification.
- VFC-reckon applies the VFC model with DeadReckoning techniques allowing to mask the intentional lack of messages between server and client.

The three models have a common configuration parameter: the time interval between each round. The round time of each model is specified after the name; for example, the Basic 40 where the number 40 indicates that the time interval is 40ms round.

[5]http://www.goriya.com/flash/asteroids/asteroids.shtml

For this work we adopted two types of round: 40ms and 160ms. The 40ms value is related to the game frame-rate. A game running at 25fps (frames-per-second) means that there is, at maximum, a new position each 40ms. The 160ms interval (a multiple of 40ms) means that the interval between each round is 160ms. The fact that this range is 4 times greater than 40ms generates a discontinuity in the movement of entities in the game due to the lack of messages.

We observed that for intervals longer than 200ms[6] the error in the estimation of new positions begins to be too noticeable, thus dramaticaly reducing the game playability.

It's important to keep in mind that the most important consistency models to evaluate are VFC-reckon 160 and Basic 40. VFC-reckon 160 corresponds to our proposed optimal solution where we reduce the number of messages exchanged without affecting the gameplay. The Basic 40 is the simplest, and most used solution to send game events by simply broadcasting all the game messages to everyone.

5.1 Qualitative Evaluation

In this analysis we pretend to measure the impact on the gameplay using our Asteroids game with VFC-reckon 160.

For this evaluation we invite 10 people to test the game and give their feedback on the gameplay impact, when we use reckon VFC-reckon 160 and a Basic 40 solution. All tests were conducted under the same scheme. The players first test the game using the Basic 40 solution and the next game the with VFC-reckon 160 solution. In the end we asked if there was a noticeable difference in gameplay between the two versions like, for example, in the movement of objects.

Tests results show the differences between the two versions are practically undetectable in the gameplay. Players didn't detect any discontinuity in the objects movement, or other types of differences. The player focus is mainly in the center of the screen, more precisely, the area closer to the player's ship. As almost the entire area of the screen is covered by the most consistent rings, the object movement in this area is fluid.

With this evaluation we have proved that the impact of VFC-reckon 160 model, in the gameplay, is almost imperceptible. VFC-reckon Interest Management techniques based on distance, shows to be a good heuristic as we proved during the tests. The use of DeadReckoning techniques, especially in the most consistent rings, makes the object's movement being as fluid as the Basic version 40.

5.2 Quantitative Evaluation

To evaluate VFC-reckon system scalability we measured the main resources used by the system: Bandwidth usage, CPU and Memory.

5.2.1 Bandwidth usage

The bandwidth is one of the resources most used by our system. To measure the bandwidth used by the system, we measure the bandwidth used during the a game execution, varying the model, round time and the number of DataUnits[6].

In terms of bandwidth, the results of VFC 160 and VFC-reckon 160 are exactly the same. DeadReckoning techniques role is to mask the intentional lack of messages, in the client side, created by the increase in the interval between each server message, which means, that it does not influence system bandwidth usage.

To measure the gains of VFC-reckon 160 vs Basic 40 model in a real game, we measured the number of messages received on the client side, through the byterate. We perform 12 tests, 3 for each number of DataUnits (50, 100 and 150) per client, using 2 clients.

The first conclusion that stands out in Fig. 4 is that the reduction in bandwidth is about 50% when comparing an VFC model type vs Basic model type with the same round time. These results show us that the Interest Management criteria used by VFC, based on the distance of objects, is extremely useful.

Analysing the behaviour of the two principal consistency models, VFC-reckon 160 and Basic 40 solution, the reduction in bandwidth with VFC-reckon 160 is, in average, 88%. The reduction stems from the greater interval between each message, from 40ms to 160ms, and the uage of VFC consistency model. Other conclusion is that the gain is independent of the number of DataUnits used, ie, use 50, 100 or 150 DataUnits usually corresponds to a gain of about 88%.

The gains of using VFC depend fundamentally on three factors: size of the rings, consistency degrees and size of the map. Decreasing the size of the rings or increase the map size will decrease the number of entities that enter the consistency rings, which will reduce the number of messages that clients receive. The area of the rings, in our tests is about 30% of the map size which, in our opinion, is a pessimistic view because in a real game it is expected that the

[6]DataUnit is the representation of one game object that a player shares with all the other players like, for example, his ship location or score

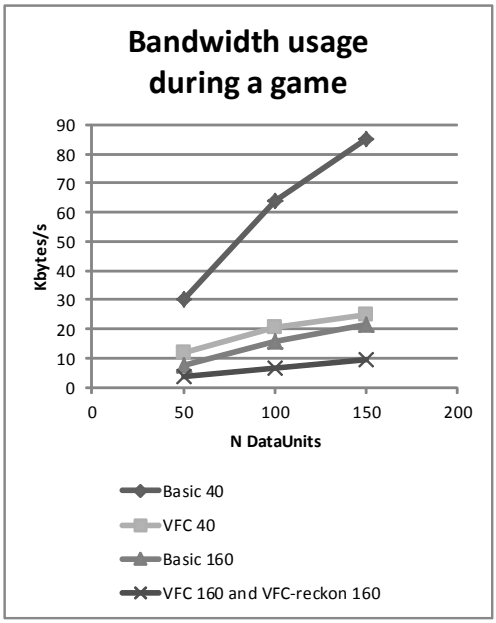

Figure 4: Bandwidth usage during a game

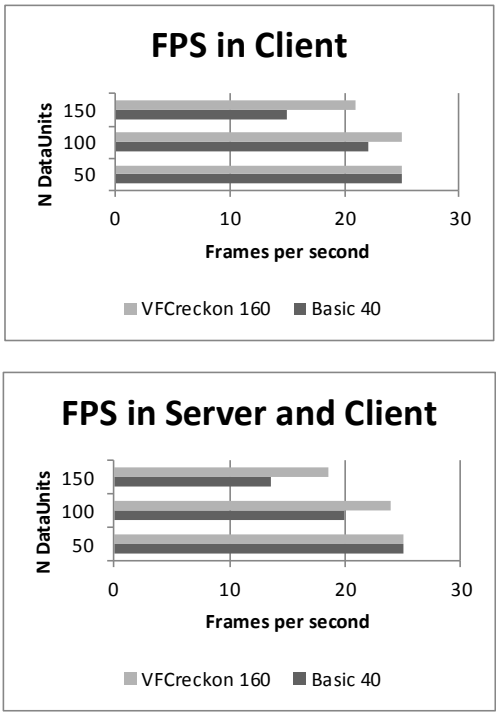

Figure 5: FPS measurements during a game execution

ratio between the area of the rings and the map area is larger.

5.2.2 CPU usage

To measure the CPU load reduction with the usage of VFC-reckon 160 e Basic 40 solution, we analysed the game's frame rate. Despite the game frame-rate a game is variable, we did about 10 measurements and it's correspondent averages in order to accomplish the results.

Fig. 5 demonstrates the benefits of using the model VFC-reckon 160 when compared to the model Basic 40. With 100 DataUnits per client, having 2 clients, is no longer possible to keep the ideal frame-rate, ie, 25fps. With 100 DataUnits the message processing load begins to be noticeable, as well as the benefits of using VFC-reckon 160. Although we already have gains with 100 DataUnits, the gains in CPU grows with the number of DataUnits. Again this is because the CPU reduction comes from the reduction in the message processing load.

Results show gains between 12% and a maximum of 24%. Although the results do not seem satisfactory it's important to keep in mind that the messages processing load, is not the only one that increases with the growing number of DataUnits. The increase in the number of DataUnits also increases the processing load of the game as, for example, calculations of physical updates to the objects.

Through a profiling tool we verified that the CPU load of our system, with a Basic 40 model, in the device that acts as a server is about 55%, and in the device that acts as a client about 40%. This means that when we apply the VFC-reckon 160 model, these are the components that we are reducing by decreasing the number of messages. When the client have an overall decrease of 24% in CPU time with 150 DataUnits, using VFC-reckon 160, this means that we subtract 24% to 40% since the other 60% correspond to the game CPU load. If we consider only our system CPU load, the overall reduction of 24% in CPU usage, means that our system uses less 60% of CPU time with VFC-reckon 160 when compared with Basic 40.

Another important conclusion that can be deduced from Fig. 5 is that the processing load of a device that has the role of server is not too different from the processing load of a device that has the role of client. This is visible through the game frame-rate, where the difference is nearly 3fps. These results confirms that the server role doesn't have a significant impact on the system performance.

5.2.3 Memory Allocation

To evaluate the memory used by the system we measured the heap memory, used by the game and system, through a profiling tool in two Android emulators. The two devices where running a client instance and a Server+Client[7] instance. In this evaluation we compare the models VFC-reckon 160 and Basic 40, switching the number of DataUnits between 25 and 100.

Due to space constraints we will briefly present the memory measurements conclusions.

The measurements show that the amount of heap memory used by the game and system, regardless the number of DataUnits, is relatively small. The minimum value in the measurements was 2.4 Mbytes, while the maximum was of 2.65 Mbytes.

Comparing the VFC-reckon 160 and Basic 40 measurements, the results demonstrate the expected. VFC-reckon 160 need more memory due to the application of VFC on the server and, the application of DeadReckoning techniques on the client side. VFC on the server side needs more memory because it performs more functions such as the selection of updates. On the client side we need to allocate more memory mainly due to the history points record in DeadReckoning module.

Although VFC-reckon 160 need more memory than Basic 40, the difference between them is relatively small when we consider the heap size. Comparing the measurements, the overhead for the client is about 50Kbytes, and for the Server+Client 100Kbytes corresponding to 2% and 4%.

6 CONCLUSIONS

We presented VFC-reckon, a consistency model based on VFC consistency model with the addition of DeadReckoning techniques. VFC-reckon is well adapted to distributed multi-user games in mobile devices for an ad-hoc network environment. The system is a middleware, implemented with a client-server architecture on which a distributed version of the Asteroids game runs. When compared to other approaches, our system provides smaller bandwidth and CPU consumption (which results from the fact that less messages are exchanged) while ensuring good game playability.

[7] Server+Client means that the device was running a server and a client instance at the same time.

References

[1] Carlos Eduardo Bezerra, Fábio R. Cecin, and Cláudio F. R. Geyer. A3: A novel interest management algorithm for distributed simulations of mmogs. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, pages 35–42, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Jean-Sébastien Boulanger, Jörg Kienzle, and Clark Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM.

[3] Thomas A. Funkhouser. Ring: A client-server system for multi-user virtual environments. In *Symposium on Interactive 3D Graphics*, pages 85–92, 1995.

[4] Dugki Min, Donghoon Lee, Byungseok Park, and Eunmi Choi. A load balancing algorithm for a distributed multimedia game server architecture. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems - Volume 2*, ICMCS '99, pages 882–, Washington, DC, USA, 1999. IEEE Computer Society.

[5] Jeffrey Pang. Scaling peer-to-peer games in low-bandwidth environments. In *In Proc. 6th Intl. Workshop on Peer-to-Peer Systems (IPTPS*, 2007.

[6] Lothar Pantel and Lars C. Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, pages 79–84, New York, NY, USA, 2002. ACM.

[7] Nuno Santos, Luís Veiga, and Paulo Ferreira. Vector-field consistency for ad-hoc gaming. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 80–100, New York, NY, USA, 2007. Springer-Verlag New York, Inc.