Synchronization Medium : A Consistency Maintenance Component for Mobile Multiplayer Games

Abdul Malik Khan and Sophie Chabridon
GET INT, CNRS UMR SAMOVAR
9 rue Charles Fourier
91011 Evry cedex, France
{Abdul_malik.Khan,Sophie.Chabridon}@int-edu.eu

Antoine Beugnard
ENST Bretagne, Computer Science Department
CS83818
F-29238 Brest cedex 3, France
antoine.beugnard@enst-bretagne.fr

ABSTRACT

In multiplayer games, where many players take part in a game while communicating through a network, the players may have an inconsistent view of the game world because of the communication delays across the network. This problem of inconsistency is even more crucial when playing on a mobile phone via a 3G network where the communication delays can be of several seconds. Consistency maintenance algorithms must be used to have a uniform view of the game world. These algorithms are very complex and hard to program. In this paper, we discuss different consistency maintenance algorithms from the point of view of mobile devices and present an approach where the consistency concern is handled separately by a distributed component called *synchronization medium*, which is responsible for communication as well as consistency maintenance. The game logic components interact with the *synchronization medium* to communicate between them and synchronize their data. We argue that this separation of concerns reduces the burden on the game developer. Moreover, a medium offers a generic interface and is designed to be easily reused for different game applications. Finally, using a medium, different consistency maintenance approaches can be tested and compared easily for experimentation.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, component oriented design methodologies.*

General Terms

Design, Algorithms

Keywords

Multiplayer Mobile Games, Latency Hiding, Data Synchronization, Communication Abstraction, Medium

1. INTRODUCTION

Consistency maintenance is a complex issue in network games. It becomes even more difficult to solve when playing on a mobile phone because of the high latency of mobile networks which can be of several seconds in 3G networks. [4] discusses the effect of high latency on players and observes that a high communication delay can cause the player quit the game. A player normally tolerates a maximum of 250ms reaction time from the system [9]. In Multiplayer online games, where many players take part in a game at the same time, it is very difficult to maintain consistency in a high latency network. To hide latency and maintain consistency, algorithms such as Dead-Reckoning[1] and Trailing State Synchronizations[5] may be used. These algorithms are very complex and are, therefore, hard to program. It is thus desirable to separate the code of these algorithms from the game logic. The concept of Medium, which is a communication component, has been proposed recently [3] to deal with interaction and distribution as non-functional aspects. In this paper, we extend this concept by inserting prediction and synchronization algorithms into a medium that we call a *synchronization medium.* This way a game programmer can concentrate on the game logic, and is relieved of synchronization and communication concerns. We also consider that synchronization is an off-shoot of communication delays and hence it is better to deal with as a communication concern rather than as a game problem. Apart from handling consistency management, another advantage of a synchronization medium is its reusibility. It offers a generic interface that does not change when replacing a synchronization algorithm by another one inside the medium. A same medium can thus be used by different game applications; the application code is not impacted even if a different synchronization algorithm that best suits the applications needs has to be plugged into the medium. This is an important property of mediums that can be used for dynamicly adapting applications. With mechanisms for dynamic adaptability inserted inside the medium, it can adapt itself by using different algorithms according to the context of the game.

In this paper, we present different synchronization algorithms from the point of view of a mobile terminal. We then discuss the separation of the code of these algorithms from the game logic and their insertion into a distributed communication component responsible for non-functional, non-game issues. We present the design of such a communication component using different synchronization algorithms.

The paper is structured as follows. First, we discuss different synchronization algorithms and their relevance for high latency mobile networks. Then, we discuss the medium approach which represents communication as a distributed component. Then, we present our approach, called synchronization medium, and show its design with UML diagrams. In the final part, we conclude and discuss our future work.

2. DATA SYNCHRONIZATION IN MOBILE GAMES

In this section, we discuss some synchronization algorithms used in distributed systems such as military simulation and games from the point of view of mobile devices. Dead-Reckoning [1] is used to reduce bandwidth consumption and hide network latencies, by sending update messages less frequently and estimating the state information between the updates, using the already received information such as position, velocity and/or acceleration of the object. The predicted value can be different from the actual value which is received through the next update message. In this case, some convergence method can be used to arrive at the actual value. The importance of dead-reckoning in mobile games is that it permits a mobile terminal not to be blocked and to continue even if it is not receiving the data in case of a disconnection, for instance. The convergence method must fuse the actual value and the predicted value in a smooth way, so that there is no abrupt effect on the game user. [12] proposes a dead-reckoning protocol based on the position history of the object being dead-reckoned. It makes sense to use the simple dead-reckoning algorithm when the path is smooth and straight such as in car race games, and to use the position-based history protocol when the path is more of a zig-zag type and hardly predictable.

Time Warp (TWS) [8] is a synchronization mechanism for parallel/distributed simulation. It allows logical processes to execute events without the guarantee of a causally consistent execution. TWS takes a snapshot of the state at the reception of a command and issues a rollback to an earlier state if a command earlier than the last executed command is received. On a rollback, the state is first restored to the previous snapshot and then all the commands that occurred between the snapshot and the execution time are re-executed. Two problems arise with this method when used for mobile games. First, it needs to store previous states, for which memory is required. Second, rollback requires processing power which can be limited in case of mobile devices.

Trailing State Synchronization (TSS) [5] also executes rollback when inconsistency is detected. However, it implements rollback so as to avoid high memory and processor overheads demanded by Time Warp Synchronization. Instead of keeping snapshots of every command, TSS keeps two copies of the same game world, each at a different local simulation times separated by some synchronization delay. The latest one in the time domain is called the leading state. The other one is called the trailing state. When an inconsistency is detected in the leading state and rollback is required, instead of copying the state from a snapshot as TWS does, TSS just simply copies the game status from the trailing state to the leading state, and then performs all commands between the inconsistency point and the present point again. The difference between a trailing state and a snapshot is that a trailing state is a complete state with all the received commands but executing with a delay d, while a snapshot is a stored state of the system after the reception of a command. A new snapshot is taken for a command received after a snapshot has been taken. TSS does not actually solve the rollback problem originated in TWS but it will have better performance when the following two situations are present. First, the game state is large and it is expensive to store the snapshot. Second, the gap between states' date is small. In order to rollback, we need to have copies of the past checkpoint. It is still a challenge to do it with less memory and processing power as on mobile phones.

Perceptive Consistency (PC) [2] provides an ordering of updates and avoids potential conflicts. Before discussing Perceptive Consistency, the two properties of *legality* and *simultaneity* need to be defined. The property of *legality* requires that the latency for a given media instance between two remote processes must be kept constant. For example, in the case of a car racing game, the *legality* property is respected if the time between two successive positions of a given car is the same for the two users, and thus the speed is the same for both users. The *simultaneity* property states that the physical time between the playouts of two updates is the same for all users. In the case of a car racing game, the *simultaneity* property is hold if in the case of a collision between two cars, the two cars are considered at the same place at a given time for all the users. For a system to be perceptive consistent, it must satisfy both properties of *simultaneity* and *legality*. The algorithm implementing PC has three phases. In the first phase, the algorithm calculates, for a given player, the maximum communication delay between this local player and all the remote players. In the second phase, the algorithm calculates the local lag to be introduced locally to order the events before the playout of a media instance. In the third phase, the message is played out. In case of mobile games, as network delays can be quite longer, the local lag introduced by PC can have bad effects on the players. Hence, dead-reckoning can be combined with PC to hide the effect of local lag by adding predicted intermediate states.

3. MEDIUM : A COMMUNICATION COMPONENT

Presented in [3], the concept of communication component or *medium* is to separate interactional details from the functional details of a component. These interactional details can be handled separately by the medium. Hence, a medium is the reification of an interaction, communication or coordination system, protocol or service in a software component. This architecture has the advantage that the medium can be reused for different types of applications.

A medium is logically a single component but physically it is a combination of different components spread across a network. Like any software component, a medium can take different shapes according to the level at which it is considered. It exists as a specification describing the communication abstraction that it reifies, but also at implementation and deployment times. It is indeed possible to manipulate a high-level communication abstraction at all stages of the software development cycle. At the specification level, a medium is specified using a UML class diagram. Then a

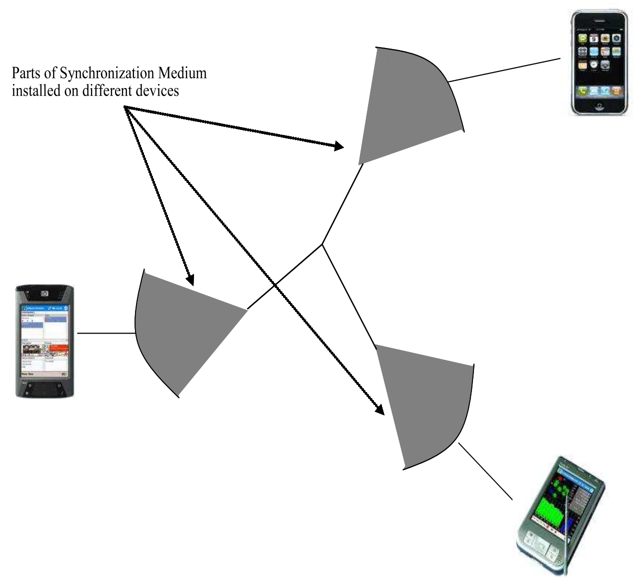
Parts of Synchronization Medium
installed on different devices

Figure 1: Synchronization Medium

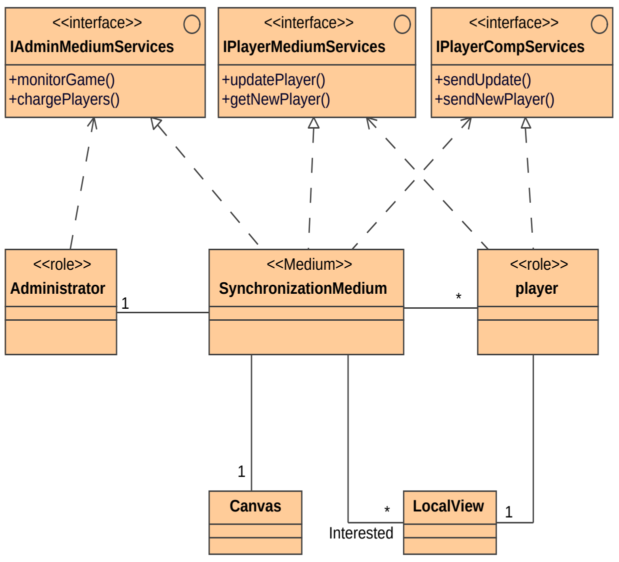

Figure 2: Abstract specification of Synchronization Medium

refinement process transforms this specification into a low-level implementation design. This refinement process is carried out in three phases. In the first phase, for each component interacting with the medium, a component called *Role Manager* is produced. This component is responsible for all the interactions with its corresponding component. A medium is an aggregate of these role managers. Logically, a medium is a single component while physically it is an aggregate of different components interacting with each other. In the second phase, the class representing the medium is removed, and only the role managers are left interacting with their corresponding components and with each other. Depending on the non-functional constraints, this phase can lead to many design choices [3]. For instance, if there are some data to be managed by the medium, we have the choice between either the data are handled centrally by a specific single manager, or they are distributed equally among several instances of a role manager. The third and final phase defines, for each design specification in the preceding phase, one or more deployment diagrams, describing how different role managers and components are distributed and grouped at the time of the deployment of a medium. For example, if we have two role managers, one playing the role of client and interact with the client component and the other playing the role of server by interacting with the components server, we can deploy the server and client role managers on different machines in case of a client server architecture or we can deploy these two role managers on the same machine in case of peer-to-peer architecture, as a peer acts both as a server and as a client. In the next section, we discuss the insertion of synchronization algorithms in the medium.

4. SYNCHRONIZATION MEDIUM

We present in this section the design of Synchronization Medium which handles consistency management as well as communication aspects. It allows to hide from the game clients, the latency compensation and synchronization mechanisms. Thus, a synchronization medium is an abstraction of the communication for data consistency across a network. This has the advantage to offer game developers a synchronization tool that can be picked up and used directly. This idea is shown in Figure 1 in the case of a deployment on heterogeneous devices.

The synchronization medium is logically a single component which is distributed physically across the network offering services required by the components interacting with it and requiring services offered by the interacting components. A two-way interaction actually takes place between the synchronization medium and application components. An abstract specification of a synchronization medium is given in Figure 2. Two components with the *Player* and *Administrator* roles interact with the medium. The *Player* role corresponds to the game client residing on the mobile terminal and uses the "IplayerMediumServices" services offered by the medium. The "IPlayerMediumServices" interface offers services like provision of information regarding a new player as soon as it joins the game session and the reception of update messages from remote players. These services are implemented through functions such as *getNewPlayer* and *updatePlayer*. The *administrator* role is the game server which uses the "IAdminMediumServices" services offered by the medium. These may include administrative services such as minotoring the game to prevent cheating and to charge fees from the users. The medium may need some services to interact with the player. These services are offered by the "IplayerComponentServices" interface of the *Player* role. The *Canvas* class represents the overall game data, while the *LocalView* class corresponds to the part of the game data that the player is supposed to receive. A player may, indeed, not be interested in all the game data but only in a subset of the canvas.

As mentioned in section 3, in the second phase of the reification process, the medium is represented as an aggregate of role managers. This is shown in Figure 3. There are two role managers namely "Player Manager" and "Game Manager". The "Player Manager" is the representative of

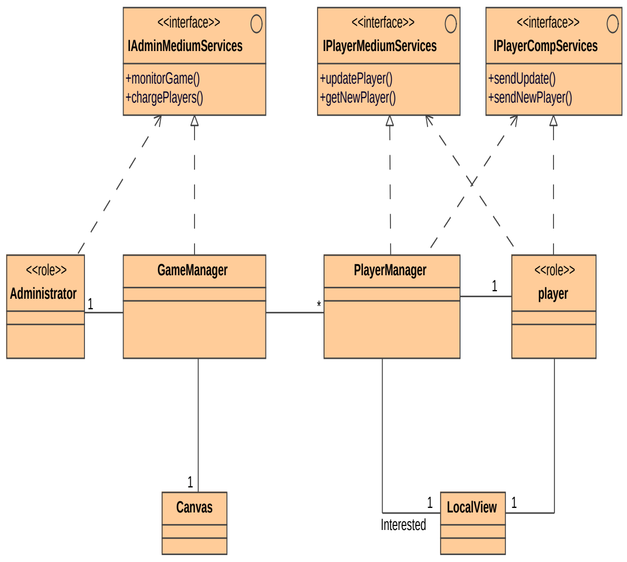

Figure 3: Introduction of role managers

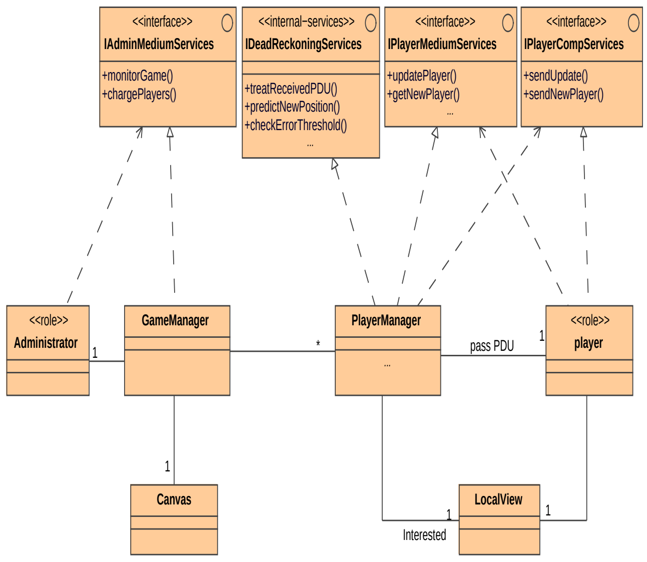

Figure 4: Synchronization Medium using dead-reckoning algorithm

the medium on each game client and offers/requires services offered/required by the game client. In the case of a client-server game, the two role managers may communicate through a middleware. Synchronization algorithms are then integrated in the medium as internal services which will be invoked by the medium transparently to the player.

We now discuss the design of a synchronization medium with different choices of synchronization algorithms.

Figure 4 shows a class diagram for a medium using dead-reckoning algorithms. The medium receives a Protocol Data Unit (PDU) concerning a remote object from a remote player. The PDU contains information that uniquely identifies an entity, such as its position and velocity. The PDU may contain an identifier telling which dead-reckoning algorithm to use. The Player Manager passes this PDU to the "IDead-ReckoningServices" interface of the medium. This interface is an internal interface of the medium because it is not used directly by any component interacting with the medium. This service predicts the new position of the entity using its *predictNewPosition* function and passes it back to the Player Manager. This updated position is then passed to the player via "IPlayerMediumServices" interface. In the case of a PDU emitted by the local player, it is first passed to the local Player Manager which in turn, passes it to the remote Player Manager. Before passing a PDU to the remote Player Manager, the *checkErrorThreshold* function is called to check whether the predicted value is different from the actual value by a certain margin. A PDU is passed to a remote Player Manager only when the *checkErrorThreshold* return true. Note that a PDU passed by a player is received by a remote Player Manager and not the remote player itself. Hence, the process is hidden from the player component in the game logic. A collaboration diagram showing the dynamic view of how the messages are passed during the dead-reckoning process between a player and the medium is shown in Figure 5. The interface connecting a role manager and a player is an external interface represented by a small black rectangle, while the interface between the local and the remote role managers is an internal interface of the medium represented by a small white rectangle.

Another choice could be to use the Perceptive Consistency algorithm within the medium for maintaining the consistency between different players. A diagram showing the synchronization medium using the PC algorithm is presented in Figure 6. A player passes an artificial message *(u, t(u))*, *u* being an artificial update issued at time *t*. The Player Manager receives such an artificial message from the remote players and passes it to the "ICalculateLocalVector" services of the medium. As discussed in Section 2, this process calculates the latency between the local player and all remote players. This service returns a vector containing these delays. The player manager then passes this vector to the "ICalculateLocalLag" service, which calculates, through its *CalculateLocalLag* function, the local lag to be introduced locally, and returns it to the "PlayerManager" via its *returnAdjust* function. The *adjust* factor is passed to the player through "IPlayerMediumServices" interface of the medium.

The basic difference between the two mediums, one using dead-reckoning and the other one using PC is in their internal services. The interfaces with the outside components remain the same. Thus, for example, the code for the game client almost remains the same except for the small changes for the type of message to be passed to the synchronization medium.

Note that it is possible for a medium to use a combination of synchronization algorithms for the same game. For example, Perceptive Consistency can be combined with dead-reckoning to compensate for high latencies in mobile networks. Dead-reckoning can help to provide predicted intermediate states during the local lag period introduced in PC.

Also, variations of the same algorithm can be used for the same application depending on the context. For example, a dead-reckoning algorithm using position-based history can

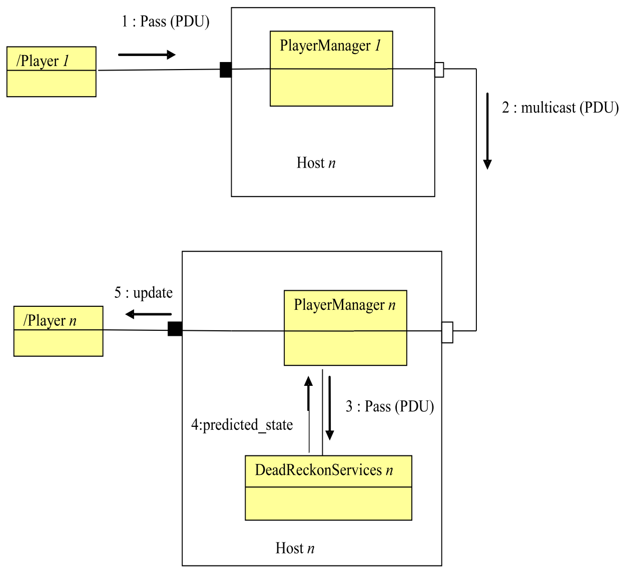

Figure 5: Dynamic view of message passing in case of Dead-reckoning algorithm

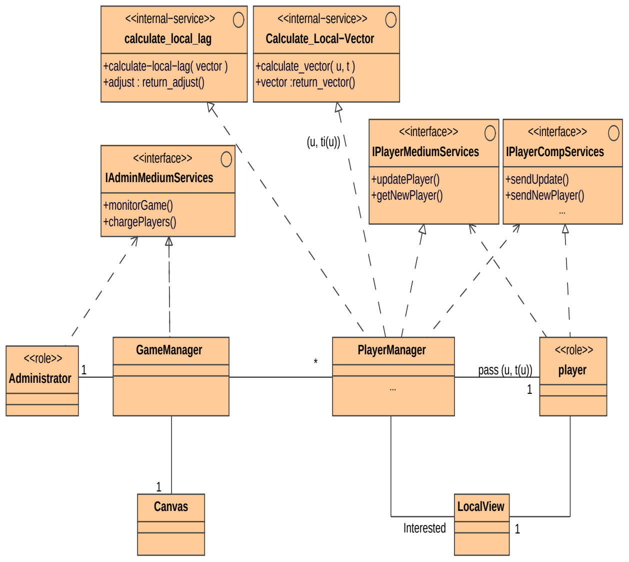

Figure 6: Synchronization Medium using PC

be used when the motion of an object is unpredictable, and a single-update-based dead-reckoning can be used when the motion of the object being predicted is smooth.

A deployment diagram for a deployment example of a synchronization medium in a client-server architecture is shown in Figure 7. The *playerManager* resides on the mobile device while *gameManager* resides on the server. Depending on the deployment constraints, one can have many deployment choices. A discussion of deployment constraints is outside the scope of this paper.

5. DISCUSSION

The primary advantage of the approach we proposed in this paper is reusability. A given synchronization medium with a specific synchronization algorithm can be reused many times by different applications. We also believe that separating the synchronization concern from the game logic, and putting it in a communication component facilitates game development by letting the developers concentrate on the game logic only. Programmers are not concerned by synchronization issues. They can use an off-the-shelf medium with a given synchronization technique by looking at its specification. Hence, the evolution of the game program becomes simpler during the course of time. Furthermore, the synchronization medium facilitates the experimentation of different synchronization algorithms. Having different synchronization mediums with different algorithms available, one can use them in the same game and compare and analyze the results. Dynamic adaptability mechanisms can also be inserted in the medium thus allowing to have more than one algorithms in the medium and use the algorithm(s) according to the context of the game. For example, a measure of the mobile network latency can help in the choice of the most appropriate algorithm. When the latency is below some threshold but still noticeable by the player, the Perceptive Consistency algorithm can give good results. When the latency increases, PC can be combined with a dead-reckoning algorithm by adding predicted intermediate states during the local lag period introduced by PC.

6. RELATED WORK

[6] proposes a Concurrency Control and Consistency Maintenance (CCCM) component to handle consistency issues separately from the game logic. The CCCM component implementing the consistency management algorithms resides between the game logic and the game data. We take a step further by decoupling the synchronization issues completely from the game logic and data, and injecting it into a communication component which handles the consistency management and returns the results to the players. Another important difference with our approach is that CCCM component is initially targeting client server architectures, while the synchronization medium is not limited to a centralized architecture. From the same abstract specification we can implement a synchronization medium for Peer-to-Peer or PP-CA (Peer-to-Peer with Central Arbiter)[10] architecture during the design process.

7. CONCLUSIONS AND PERSPECTIVES

In this paper, we argue for the separation of synchronization algorithms from the game logic. We propose to insert

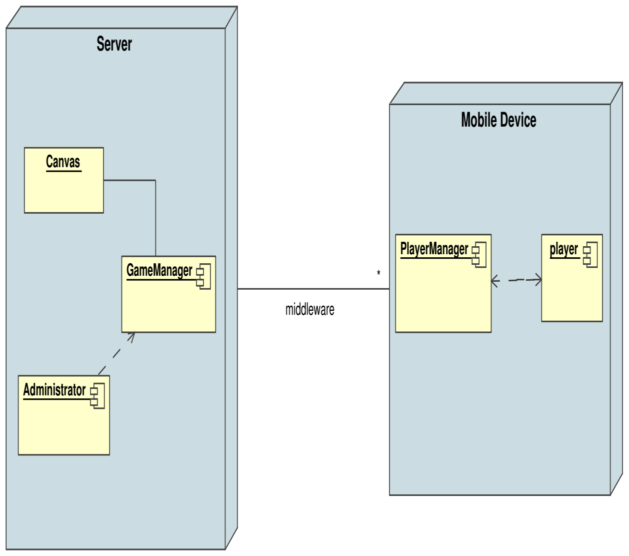

Figure 7: Deployment of synchronization medium in case of client server architecture

them into a communication component which we call synchronization medium. Before going into the details of our approach, we presented different synchronization algorithms used in multiplayer games and their pertinence for the case of mobile phone terminals. We showed different diagrams for the specification of a synchronization medium during the process of its reification. As future work, we intend to implement synchronization media using different algorithms on top of the GASP [11] framework, a middleware dedicated to the development of multiplayer games on mobile phones. A comparative analysis of these algorithms is also in perspective. Other issues, such as the Adaptive Focus Control [7] method for interest management and dynamic adaptability can be included in the medium thus further separating the game logic from non-game issues and offering more flexibility to game developers.

8. ACKNOWLEDGMENTS

This work is a part of the project JEMTU which aims to design solutions for the technological and psychosociological issues that slow down the development of multiplayer games on mobile phone (http://proget.int-evry.fr/projects/JEMTU/).

9. REFERENCES

[1] Application protocols. In *IEEE Standard for Distributed interactive Simulation*. IEEE Std 1278.1-1995, 1995.

[2] N. Bouillot. Fast event ordering and perceptive consistency in time sensitive distribued multiplayer games. In *7th International Conference on Computer Games (CGAMES'2005)*, pages 146–152, 2005.

[3] E. Cariou, A. Beugnard, and J.-M. Jézéquel. An archictecture and a process for implementing distributed collaborations. In *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002)*, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, September 17 - 20 2002.

[4] K.-T. Chen, P. Huang, and C.-L. Lei. How sensitive are online gamers to network quality? *Commun. ACM*, 49(11):34–38, 2006.

[5] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 67–73, New York, NY, USA, 2002. ACM Press.

[6] R. D. S. Fletcher, T. C. N. Graham, and C. Wolfe. Plug-replaceable consistency maintenance for multiplayer games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 34, New York, NY, USA, 2006. ACM Press.

[7] C.-c. A. Hsu, J. Ling, Q. Li, and C.-C. J. Kuo. The design of multiplayer online video game systems. In A. G. Tescher, B. Vasudev, V. M. J. Bove, and A. Divakaran, editors, *Multimedia Systems and Applications VI. Edited by Tescher, Andrew G.; Vasudev, Bhaskaran; Bove, V. Michael, Jr.; Divakaran, Ajay. Proceedings of the SPIE, Volume 5241, pp. 180-191 (2003).*, volume 5241 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 180–191, Nov. 2003.

[8] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6(1):47–57, Feb. 2004.

[9] L. Pantel and L. C. Wolf. On the impact of delay on real-time multiplayer games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA, 2002. ACM Press.

[10] J. D. Pellegrino and C. Dovrolis. Bandwidth requirement and state consistency in three multiplayer game architectures. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 52–59, New York, NY, USA, 2003. ACM Press.

[11] R. Pellerin, F. Delpiano, E. Gressier-Soudan, and M. Simatic. Gasp: A middleware for multiplayer games in mobile phone networks (in french). In *UbiMob '05: Proceedings of the 2nd French-speaking conference on Mobility and uibquity computing*, pages 61–64, New York, NY, USA, 2005. ACM Press.

[12] S. K. Singhal and D. R. Cheriton. Using a position history-based protocol for distributed object visualization. Technical Report CS-TR-94-1505, Stanford University, Stanford, CA, USA, 1994.