An Distributed Architecture for Mobile Digital Games Based on Cloud Computing

Marcelo Zamith
UFF, Medialab

Mark Joselli
UFF, Medialab

Luis Valente
PUC-RIO, ICAD Games

Esteban Walter Gonzalez Clua
UFF, Medialab

Anselmo Montenegro
UFF, Medialab

Regina Celia P. Leal-Toledo
UFF, IC

Bruno Feijó
PUC-RIO, ICAD Games

Abstract

Several fields in Computer Science use distributed computing to solve many intensive computational problems. Digital games use this approach mainly in multiplayer games, where a mainframe or cluster processes the majority of game logic. Single player games can also use distribute computing to process game logic and visualization algorithms, usually the tasks where digital games spend most of the processing time. By applying an approach based on distributed computing, games would have softer requirements regarding hardware, since the network cluster would be responsible for processing parts of game loop tasks. With the concept of cloud computing, games could rely on other computers to aid in processing their tasks. This work presents game-loop architecture for single-player or multiplayer games, using automatic load balancing and distributing game logic computation among several computers.

Keywords:: Mobile Games, Cloud Computing, Distributed Systems, Parallel computing, Real-Time Systems

Author's Contact:

mjoselli,mzamith,esteban,anselmo,leal@ic.uff.br
lvalente,bruno@inf.puc-rio.br

1 Introduction

Digital games are real-time interactive multimedia applications. In other words, if the application is not able to perform the required tasks on time, it will fail. For game applications, this means not being able to sustain the interactive experience, due to factors like the game taking too much time to process the tasks, or delayed responses for user input.

Mobile games are applications that run on mobile devices as smartphones and tablets. Those devices offer opportunities to design novel gaming experiences due to the distinct characteristics of those devices, but at the same time, developing for those devices poses another set of challenges. For example, mobile devices have more constrained processing power and memory capacities than desktop computers and dedicated consoles, although this has been improving greatly over time.

Mobile devices also have more limited input methods than desktop computers or dedicated game consoles. For example, mobile phones were initially designed for making voice calls, meaning that their input method (numerical keyboard) was optimized to dial numbers. This has been changing as more and more devices have touch screens.

The characteristics of current mobile devices (especially smartphones) make it possible to design novel game experiences. As an example, smartphones provide a high degree of convergent features: multimedia capacities (producing and consuming audio, video), networking (local and global), and sensors (camera, accelerometers, GPS, etc). This opens up the possibility to create games as location based games [M1ndLab 2007], voice based games [Zyda et al. 2008], accelerometer based games [Chehimi and Coulton 2008], camera based games [Park and Jung 2009] and touch based games [Rohs 2007]. In order to develop good mobile games, they must be designed to take advantages of such unique characteristics into gameplay [Zyda et al. 2007].

Computer games are multimedia applications that employ knowledge of many different fields, such as Computer Graphics, Artificial Intelligence, Physics, Computer Networks and others. Moreover, computer games are also interactive applications that exhibit three general classes of tasks: data acquisition, data processing, and data presentation. Data acquisition in games is related to gathering data from input devices as keyboards, mice and joysticks, or any other kind of interaction. Data processing tasks consist on applying game rules, responding to user commands, simulating Physics and Artificial Intelligence behaviors. Data presentation tasks relate to providing feedback to the player about the current game state, usually through images, audio and vibration. In massive online games, there is also one more class for tasks: game distribution. Game distribution is the logical partitioning of the game world among multiple servers, computation distribution management according to actual game state, and communication [Glinka et al. 2008].

Games are interactive real-time systems and have time constraints to execute all of their processes and to present the results to the user. If the system is unable to do its work in real-time, it will lose its interactivity and consequently it will fail. A common parameter for measuring game and visual simulation performance is frames per second (FPS). The general lower acceptable bound for a game is 16 FPS. There are not higher bounds for FPS measurements, but when the refresh rate of the video output (a computer monitor or the mobile screen) is inferior to the game application refresh rate, some generated frames will not be presented to the user (they will be lost). One motivation for designing game loops is to better achieve an optimal FPS rate for the application.

Mobile phones are connected devices by definition. This means networking is an important component to consider for mobile applications, especially for games. Mobile phones are able to establish connection with local (co-located) or global peers. Local peers are connected through technologies as Bluetooth, while global peers are reached through WiFi and the mobile operator network (e.g. 3G). This built-in feature makes developing mobile multiplayer games a natural move.

Multiplayer online games have been contributing to increasing internet network traffic. In this kind of game, clients who are positioned across the Internet connect to a game server (or another game client acting as a server) to interact with other clients in order to be part of the game. In current architectures, clients and servers exchange messages directly. The architecture this work proposes shares similar concerns as cloud computing [Armbrust et al. 2009]. In cloud computing, computers across the internet share resources, software and information, while in our approach the mobile client is able to use resources available in the network to help in game processing. In our approach, a mobile client with less computing power could join a game session, by relaying the effort for game processing to the network cloud.

1.1 Motivation and Contribution

The problem this work aims at helping to solve is to have lower-powered devices taking part into complex games (e.g. with sophisticated visual effects), as well as allowing many players to play together through their mobile devices.

In face of improving the game quality (visual effects, AI and so on) on mobile devices, the contribution of this work is a distribution architecture as cloud computing architecture. Basing on the the work proposed by [Joselli et al. 2010], which has presented a framework for game loops that use automatic task distribution be-

tween CPU cores and the GPU. This work discusses an approach of distribution game loop with mobile client through cloud computing context, as well as the previous concepts presented in earlier works, and new ones as follows:

- Overview of real-time game loops that can be applied in mobile platforms;
- Task distribution among computers, related to game loops;
- Load balancing of tasks;
- Mobile clients using the cloud to process some of their tasks.

Finally, the organization of this work is as follows: Section 2 presents the related works. Section 3 discusses the architecture this work proposes. Section 4 discusses and analyses the tests. Finally, Section 5 presents the conclusions of this work.

2 Related Work

Mobile devices constraints are a challenge for developing complex games. On the other hand, cloud computing services allow for processing tasks that have high processing demands. The infrastructure for those services could be based on computer clusters, for example.

Considering games, an ideal solution would be a cloud computing service allowing many players to join a multiplayer game, while being able to perform the heavy part of the information processing on behalf of the mobile clients.

In this regard, some works have shared concerns similar to the idea we have described, as the Onlive service (www.onlive.com). The Onlive service runs game loop tasks inside a computer cluster, while the clients are responsible for providing the visualization part and gathering user input. This approach requires less processing power from the game clients (as the servers perform the heavyweight part of processing), but it requires high-speed network connections and is subject to high latency, which for mobile clients can be a big issue.

The real-time loop represents the heart of real-time simulations and games.However, there are few works that discuss this subject. Among the few ones, are [Valente et al. 2005], Dalmau [Dalmau 2003], Dickinson [Dickinson 2001], Watte [Watte 2005], Gabb and Lake [Gabb and Lake 2005], and Mönkkönen [Mönkkönen 2006]. None of them discuss game loop models with focus on mobile devices.

The most straightforward approach to modeling real-time loops (for single-player games) is the Simple Coupled Model. Basically, this model consists of sequentially arranging the tasks in a main loop as Figure 1 illustrates. This is a basic approach that mobile games have already adopted in the past, due to its simplicity.

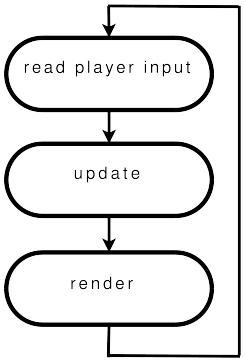

Figure 1: *Simple Coupled Model*

Dickinson [Dickinson 2001] proposed an extension to the Simple Coupled Model, named Single-thread Uncoupled Model. This model is namely the single-thread uncoupled models. This model has the rendering and updating stages uncoupled, i.e., rendering and updating are running independently of the power processing of CPU. Moreover, the single-thread uncoupled model tries to bring determinism to the game execution by feeding the update stage with a time parameter. For example, existing open-source game engines, as [COCOS2D 2011] adopts this model.

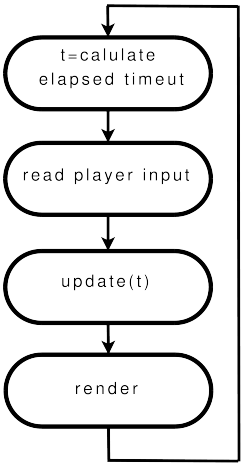

Figure 2: *Single-thread Uncoupled Model*

Single-thread Uncoupled Model has been improved by making it more adaptable to different machine capacities. In other words, with this model the application has chance to adjust its execution according to the capacity of the host machine, so the game runs the same way in different devices. More powerful devices will be able to run the game more smoothly, while less powerful ones should still be able to provide some experience to the user.

Although these are working solutions, time measuring may vary greatly in different hardware devices due to many reasons (such as process load), making it difficult to reproduce it faithfully. For example, a network module implementation and program debugging [Dickinson 2001] may be easier to implement if the loop uses a deterministic model. Another issue is that running some simulations too frequently, like AI and the game logic, may not yield better results.

Hence, research works as [Valente et al. 2005] propose models that try to address those issues. The Fixed-frequency Uncoupled Model outlined in [Valente et al. 2005] features another update stage that runs at fixed frequency, besides the time-based one. The work by Dalmau [Dalmau 2003] presents a similar model, although not naming it explicitly. Those works describe the model using a single-thread approach. Figure 3 illustrates the Fixed-frequency Uncoupled Model.

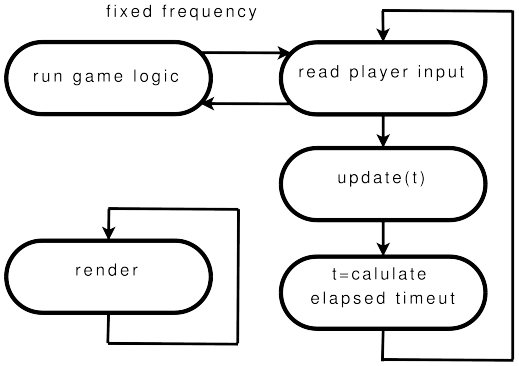

Figure 3: *Fixed-frequency Uncoupled Model*

Finally, Dickinson [Dickinson 2001] discusses another approach for fixed-frequency uncoupled models, which presents just one update stage that runs at a xed-frequency. The main objective of this model is to attain reproducibility.

Nowadays, mobile devices, like the Motorola Atrix, iPad 2, LG Optimus, HTC Pyramid and the Samsung Galaxy S-II, have multi-core processors. For this reason, real-time loops for mobile games that take advantage of those resources are likely to become important

in the near future. Therefore, making the tasks parallel in multiple threads is a natural step.

However, dealing with concurrent programming introduces another set of problems, such as data sharing, data synchronization, and deadlocks. Also, as Gabb and Lake [Gabb and Lake 2005] states, that not all tasks can be fully parallelized due to dependencies among them. As examples, in a game, characters cannot move until the game logic is computed, and rendering cannot be performed until the game state is updated. Hence, serial tasks represent a bottleneck to parallelizing simulation computation.

The Asynchronous Function Parallel Model [Mönkkönen 2006], which separates the tasks (input, update and render) in three threads. Another example is the Synchronous Function Parallel Model [Mönkkönen 2006], which processes the game physics in a separated thread while the main thread process the characters animations.

The Data Parallel Model [Mönkkönen 2006] uses a different paradigm where data are grouped in parallel sections of the application where they are processed. So, instead of using a main loop with concurrent parts that process all data, the Data Parallel Model proposes using separate threads for sets of data (like game objects). This way, the objects run their own tasks (like AI and animation) in parallel. Figure 4 depicts this approach.

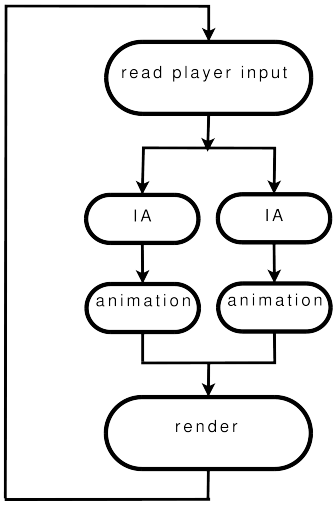

Figure 4: *Data Parallel Model*

According to the author[Mönkkönen 2006], this model scales well because it is able to allocate as many processing cores as they are available. Performance is limited by the amount of data processing that can run in parallel. An important issue is how to synchronize communication of objects running in different threads. The author states that the biggest drawback of this model is the requirement to have components designed with data parallelism in mind. This work has been inspirational for the distributed part of our architecture, where it splits a task into threads that run across the cloud.

Barbosa and co-authors [Barboza et al. 2010a] propose a cloud-computing approach for mobile devices, similar to the Onlive architecture. In their proposal, a mobile device sends user input information to a server, and later receives back the rendered images for the game as a streaming video. Figure 5 illustrates the game loop for this approach.

Another approach for game loop architectures adopts the GPU as a new resource in the computer. This resource can be used to process physics or any other massively mathematics problems apart from visualization task. This approach is based on GPGPU, whose importance has been increasing since graphics hardware became programmable. There are some works that discuss using GPGPU with game loops [Barboza et al. 2010b]. However, these works concentrate on game loops for desktop computers, and currently GPUs in mobile devices (smartphones, tablets) do not have capabilities for the GPGPU programming.

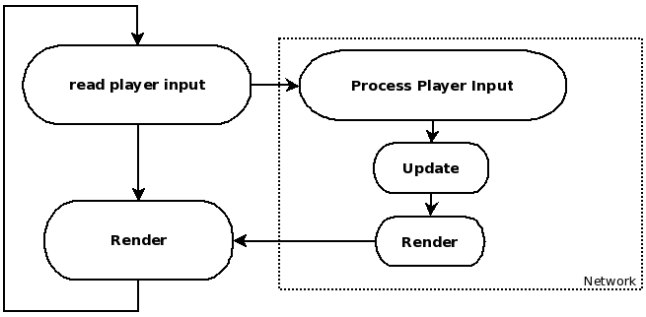

Figure 5: *The cloud game loop*

Using the GPU as a mathematics co-processor gives rise to a balanced architecture, i.e., a game loop architecture that is able to distribute tasks between the CPU and the GPU [Zamith et al. 2007]. It is possible to extend this approach for mobile devices, by proposing a game loop framework that is able to distribute update tasks among distributed computers (CPUs and GPUs), and mobile device CPUs, as Figure 6 illustrates.

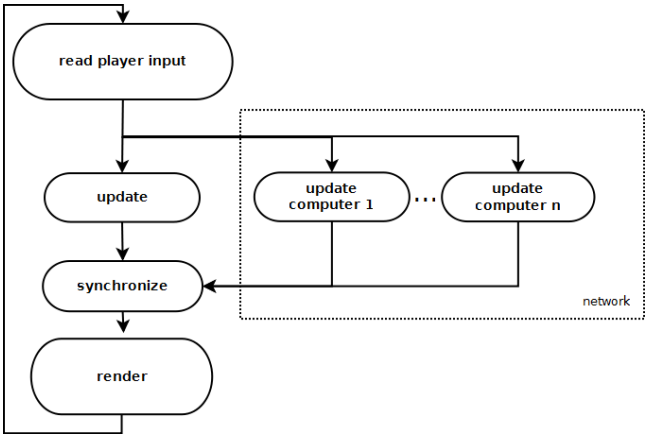

Figure 6: *The Distributed Mobile Game Loop*

3 A Cloud Computing approach for game loops

This work proposes an architecture for game loops that applies cloud computing concepts. This means that there is a server cluster and a set of mobile devices. The mobile devices connect to the server cloud to join multiplayer games and to use cloud computing services.

The architecture is scalable, according to the number of mobile clients. Each server is able to handle a certain number of mobile clients. When a specific server is at full capacity, the cluster forwards client requests to another available server. The motivation for this approach is to avoid having overloaded servers and to support having a huge amount of clients

The architecture has two main components: the game interface (i.e. the game visualization, a component responsible for delivering the results of processing visualization algorithms to clients), and a component responsible for cluster processing that is responsible for implementing the distributed game loop.

The current implementation uses HTML 5 for the mobile client and C++ with MPI (Message Protocol Interface) for processing tasks (as AI) in the cluster. The web server is Apache.

The mobile application uses HTML 5 to implement the user interface, gather user input, rendering, and to connect to the server cluster. The mobile application sends to the server the current game state for each frame, and receives from the server a new version of game state from de server. Basically, the game loop for the client

includes those tasks. We have chosen HTML 5 as it is a portable alternative for mobile device browsers.

The component responsible for cluster processing has two parts: the master process, which is responsible for answering client requests, and the slave process, which runs the distributed game loop.

Initially, the master process waits for incoming connections from clients. When a client connects, it creates slave processes that run game loop tasks such as physics simulations, AI processing, and other related to the game update stage. The Figure 7 illustrates the proposed architecture.

For each game loop step, the master process receives the current game state from a client, and updates it in each slave process, dividing the problem with all slave processes. Then, the master process merges the solution of each slave process, composing a new game state. Finally, the master process sends to client this new game state.

The processes communicate among themselves using MPI. MPI is a powerful framework for developing parallel and distributed applications. Among its features are dynamic process creation, different synchronization object types, and different communication modes (asynchronous and synchronous).

In summary, the master processes communicates with clients (through sockets) and slave processes (through the MPI library).

The architecture has some constraints, as it needs to copy some game information to the cluster and client applications. For instance, in the test application (a pac-man game) presents the game scene in clients and in the cluster. The motivation for this approach is to minimize communication between cluster and clients.

In order to guarantee the real-time requirements for the game loop, the master process is able to create other master processes, as a master process can handle a limited number of clients. Hence, whenever a master process notices the game loop update has slowed down, the master process creates another master process from one of the existing slave ones. When this happens, it creates another slave process to replace the one that has been transformed.

The motivation for this strategy is to decrease the communication between master process and clients, balancing the load dynamically.

3.1 Load Balancing Strategy

The load balancing strategy aims at providing a management layer analyzes the hardware performance dinamically and adjusts the amount of tasks to be processed by the resources (computers, CPUs and GPUs).

To make a correct task distribution, it is necessary to run an algorithm. In the current implementation, a script is responsible for this. The load balancing applies the scripting approach because the loop can be used in many simulations, and each simulation requires different algorithms and parameters.

The load balancing core corresponds to the Task Manager and Hardware Check classes. The Task Manager schedules tasks in threads and changes which processor handles them whenever it is necessary. The Hardware Check detects the available hardware configuration capabilities.

Additionally, the load balancing cores applies heuristics presented in work develop by Joselli [Joselli et al. 2008]. Whereas, work [Joselli et al. 2010] discuss the concept of tasks. A task corresponds to some processing that the application need to execute. Examples of tasks include reading player input, rendering and updating game objects, as illustrated by Figure 8.

In this work, as in previous approach, task represents anything that the application should process. However, not all types of processors are able to process any type of tasks. Usually, the application defines three groups of tasks. The first group consists of tasks that only CPUs are able to run. The second group consists of tasks suitable for running in the GPU, like presenting a scene. The third group consists of tasks that both CPUs and GPUs are able to run. The tasks in this group can be distributed among computers, and are responsible for processing operations as Physics and AI.

The Task class is an abstract base class and has six subclasses: Input Task, Update Task, Presentation Task, Hardware Check Task, Network Check Task, and Task Manager. The first three classes are also abstract classes. The Hardware Check Task and Network Check Task classes are responsible for checking hardware and network connection speeds. The Task Manager class is a special class that performs the task distribution. The Automatic Update Task and Distribution Task use the Task Manager class services. The first one distributes tasks between CPU cores and GPU, while the second distributes tasks among computers.

The Input Task classes and subclasses handle user input related issues. The Update Task classes and subclasses are responsible for updating the game loop state. Tasks related to CPUs should use the CPU Update, while tasks related to GPUs should use the GPU Update class. Tasks that run in multiple CPU cores should use the CPU Multithread task class.

The Presentation Task and its subclasses are responsible for presenting information to the user, which can be: visual (Render Task), with images, 3D models and visual effects; audio (Sound Task), with music and sound effects, or motion (Vibrate Task), with vibration feedback on the mobile phone.

The Update Task classes and its subclasses are responsible for updating the game loop state. In this case, there is only one subclass the Network Update class.

3.2 The Mobile Architecture

The mobile architecture applies the task concepts that the previous Section discussed.

For mobile devices, the Input Task classes handle user input that comes from several sources. For example, the accelerometer sensor, microphone (voice commands), keyboard, touch screens, camera (using camera images to estimate device motion), and location (using GPS and WiFi, for example).

As stated earlier, the Presentation Task and subclasses are responsible for presenting information to the user. This information can reach the user through several modalities, as visual (Render Task, with images, 3D models and visual effects), aural (Sound Task, with music and sound effects), or haptics (Vibrate Task, controlling the vibration motors in mobile phones).

The Update Task classes and subclasses are responsible for updating the game loop state. In this case there is only one subclass, the Network Update Task class.

3.3 The Network Update Task

The Network Update Task is the core component in the architecture. It is responsible for interacting with Distributed Architecture and updating the game state according to the data received from it.

This task acts as a client for the Distributed Architecture. It gathers player input and send this information to the distributed architecture through the network.

Network Update Task includes as socket as MPI. In doing so, Network Update Task uses socket to communication with mobile client. Whereas, the master e slaves processes communicates thought MPI.

3.4 The Distributed Architecture

The core of the proposed architecture corresponds to the Task Manager and the Hardware Check class. The Task Manager schedules tasks in threads and changes which processor handles them whenever it is necessary. The Hardware Check detects the available hardware configuration capabilities.

In the proposed architecture, like in the mobile architecture, a task can be anything that the application should process. The Task class

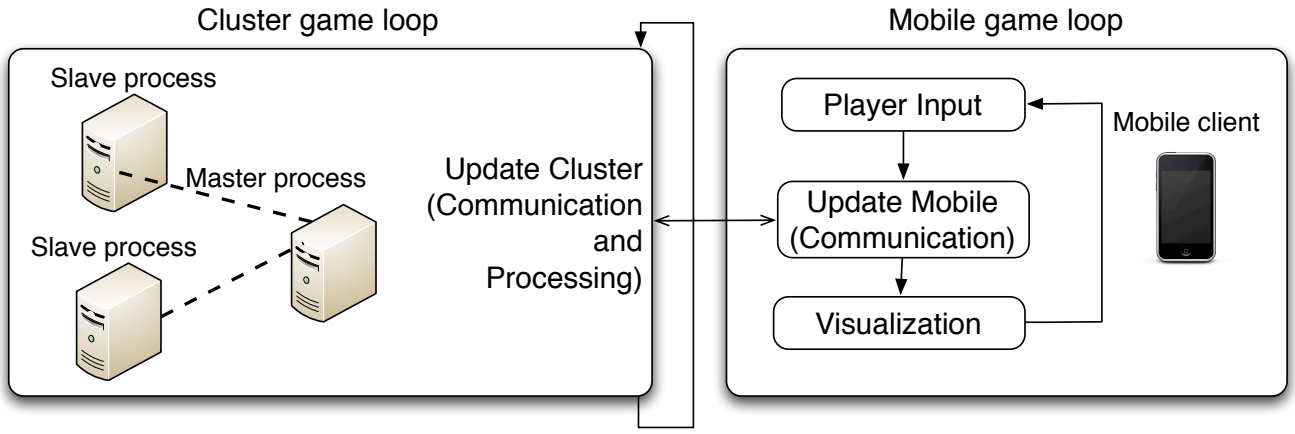

Figure 7: *Distributed Game Loop Architecture*

is the abstract base class and has four subclasses: Update Task, Hardware Check Task, Network Check Task and Task Manager. The first is also an abstract class. The second and third is a special classes are special classes to check the hardware and network connection speed. The Task Manager is a special class that is responsible for performing the task distribution. This special class is used by the Automatic Update Task, which distributes tasks between CPU cores and GPU, and Distribution Task, which distributes tasks among computers.

The Update Task classes and subclasses are responsible for processing the new loop state, which are presented by the mobile device. The CPU Update class should be used for tasks that run on the CPU, the GPU Update class corresponds to tasks that run on the GPU, and the CPU Multithread task class correspond to task that can be distributed among CPUs cores.

4 Test Case: Distributed Pac-man

The test case consists of an example classic game: pac-man. The architecture is composed of four computers: one is dedicated to a web server and the three others compose the cluster. This cluster is responsible for running the AI for the ghosts in pac-man. Figure 9 depicts the architecture. The web server, the cluster and mobile clients are connected through the internet. In spite of native code in mobile device provides better performance, the authors chose HTML 5 as client mobile. This language is portable and also provides the proposed architecture with necessary features [W3C 2011].

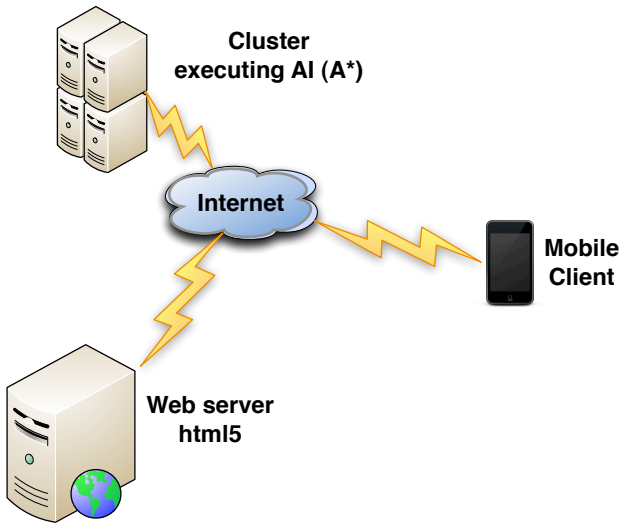

Figure 9: *The Cluster Architecture*

The architecture is composed of two parts: The first is the web server which hosts a HTML 5 web page with the pac-man game. The second part is the cluster. This cluster executes the ghost AIs, where each process corresponds to an instance of the AI algorithm. Each instance corresponds to one ghost. However, only one process is responsible for communicating with the mobile client. Finally, the ghost AIs are based on the A* path-finding algorithm. The web server hardware consists of three computers. All of them have an Intel Core 2 Duo CPU (E6750) running at 2.66GHz, 4MB of cache memory and 1GB of RAM memory. The operating system is Ubuntu 4.4.1-4ubuntu9 64 bits, and the Apache server is of version 2.2.

The cluster is composed of three computers: two are Intel Core 2 Duo CPU (E6750) with 2.66GHz, 4MB cache memory and 1GB RAM memory; and one has an Intel, Core 2 Quad CPU (Q6600) with 2.40GHz, 4MB cache memory and 4GB RAM memory. All cluster computers run Ubuntu 4.4.1-4ubuntu9 64 bits. The graphics card is a NVIDIA GTX480. The cluster uses a chance message protocol to execute the communication among them. Although all processes are similar, there is one that is responsible for the mobile client connection. This process is named as master process and the others are named as slaver processes. The test case uses an iPhone second generation, with 16GB of memory and iOS is 4.3.3, as the mobile device.

A possible scenario for this architecture is as follows: the mobile client connects to the web server, which sends the client a HTML 5 page with the pac-man game. When the game is loading in the mobile client, the HTML 5 page requests a socket connection to the cluster, using TCP/IP. On the cluster side, there is a process that answers mobile client requests.

After the game has started, the master process receives mobile client requests (e.g. update pac-man position), distributes the request to the slave processes, and waits for new positions of the ghosts. Each slave process is responsible for one ghost in the game. In other words, each slave process runs the A* path-finding algorithm for one ghost.

The slave processes send the updated ghost position to the master process, after the results of the path-finding algorithm. The master process then attaches a time stamp to this information and sends it to the mobile clients. Finally, the mobile clients update the game presentation with this information received from the cluster.

The basis for the test case is varying the number of ghosts in a game. A dedicated cluster process is responsible for running the AI for each ghost. The test starts with 2 ghosts and ends with 32. The test disregards the time it takes for mobile clients and the cluster to communicate, as this counts for less than 1% of the game loop elapsed time. Also, the size of message should not be taken in account, because of 32 ghosts do not represent a large message. Thus, table 1 depicts performance based on number of ghost. The columns are:

- **NG**: Number of ghosts;

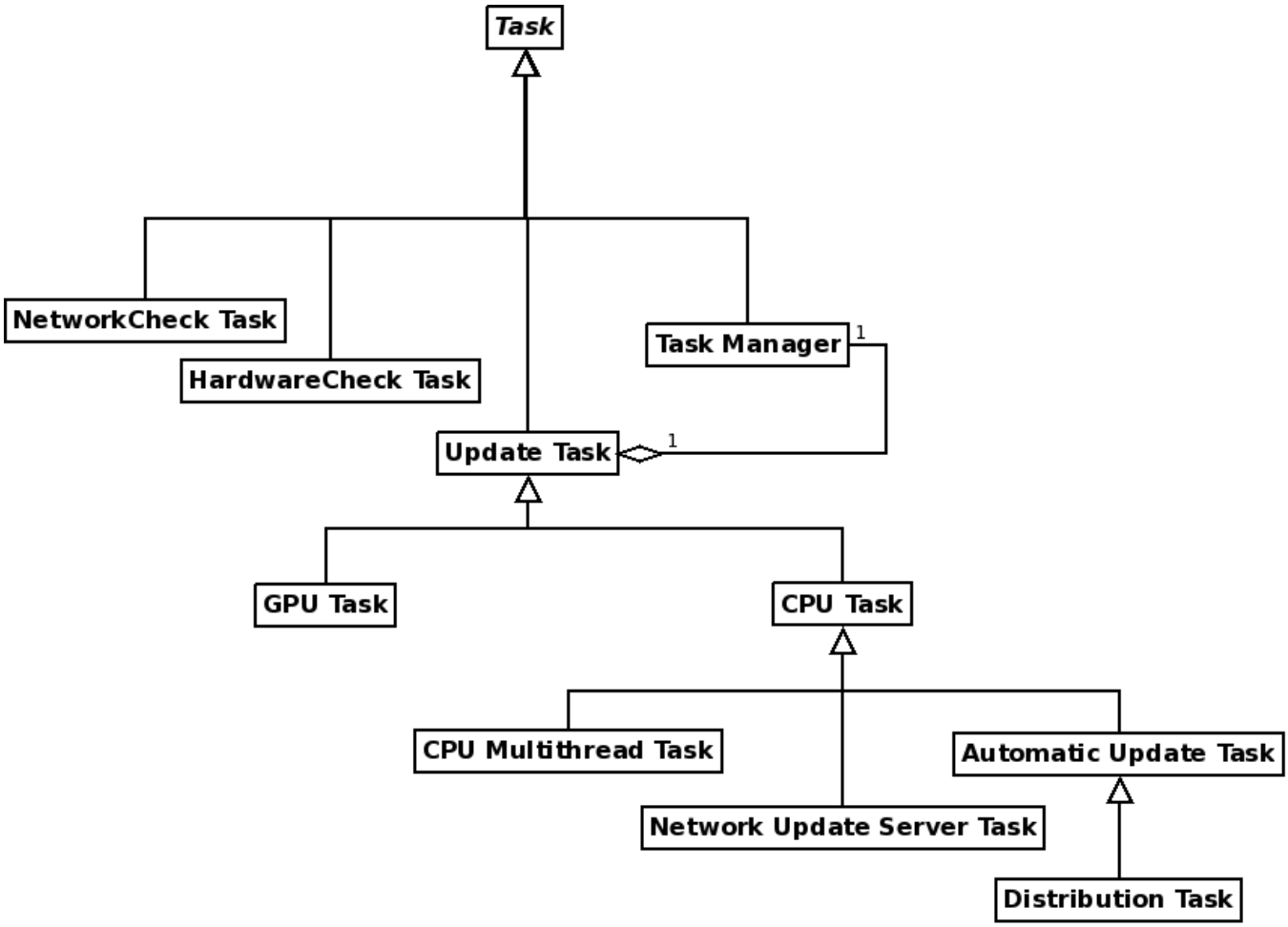

Figure 8: *Framework UML Diagram*

- **RB**: Total received bytes from mobile client;
- **SB**: Total sent bytes to the mobile client;
- **AI** : Time elapsed in the AI processes;
- **COMM** : Time elapsed in the communication between cluster and mobile client;
- **ET**: Game loop elapsed time, considering time elapsed in artificial intelligence (AI) and communication between cluster and mobile client (COMM).
- **FPS**: Frames per second rate in the game loop.

According to the tests, even when there are twice as many ghosts, the performance remains satisfactory. As the worst-case scenario has 32 ghosts and the FPS rate corresponds to 270.5, the architecture is scalable both in number of CPUs and ghosts. An environment with 32 ghosts is similar to an environment of crowd simulation [Passos et al. 2008]. In this case, the ghosts would correspond to people (the crowd).

NG	RB	SB	AI	COMM	ET	FPS
2	8	8	1.94209	0.00850	1.95059	512.67
4	8	20	2.39055	0.00800	2.39855	416.92
8	8	44	2.94858	0.00800	2.95658	338.23
16	8	92	1.86514	0.00833	1.87347	533.77
32	8	212	3.61611	0.01367	3.62978	275.50

Table 1: *Cluster performance based on number of ghosts*

Finally, message optimization collaborates to the architecture performance. Each mobile client sends the cluster the pac-man position, which corresponds to 8 bytes (column RB in Table 1). On the other hand, the cluster sends to mobile clients the current position of each ghost. This represents a byte quantity that is proportional linearly to the number of ghosts (column SB in Table 1). Moreover, the cluster should have a copy of the scene to calculate collisions of ghosts and walls. The game loads this information when it is starting.

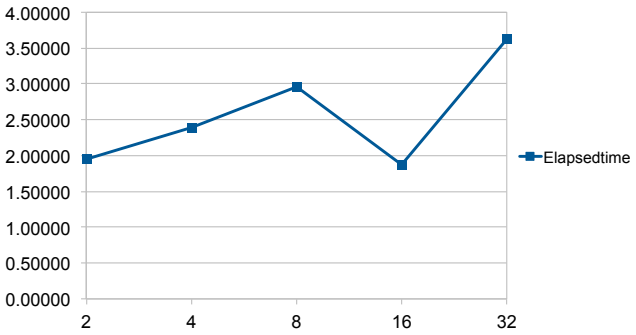

Figure 10: *Performance based on elapsedtime (ms)*

The Figures 10 and 11 illustrate the performance fluctuation when the message increases in size. The communication between master process and mobile client increases linearly according to the number of ghosts.

However, the tests also suggested a fluctuation in the game loop elapsed time. This happened because the cluster library the test application has used. The cluster library is MPI (Message Protocol Interface), which changes the way it encapsulates messages according to the message size. In this case, message size has increased according the number of ghosts, as each ghost is associated with a dedicated process.

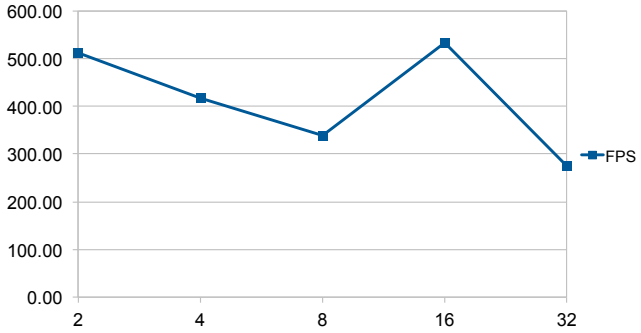

Figure 11: *Performance based on FPS*

Furthermore, the test results have suggested that this approach looks promising, as the communication part does not consume significant time in the game loop. This opens up opportunities to improve the quality of physics simulation or visual details on the mobile device, for example, as cluster can take care of the heavy-weight tasks.

5 Conclusions

With the evolution of networks and mobile devices, distributing computation will become more in evidence, even for mobile games.

This work has discussed the concept of game loops with focus on mobile games a subject that has not been discussed in the literature, to the best of our knowledge.

This paper contribution lies on extending a previous work, by providing an architecture for game loops that is able to distribute tasks in mobile device games among computers in a network, which in turn uses CPUs, CPU cores, and GPUs for processing. With this approach a game is able to use more resources (local and remote), reducing the system requirements.

The framework and concepts this work has presented can be applied to any game or real-time simulation task that is able to run in a parallel mode. With distributing tasks across the internet, mobile devices could run more processing-hungry games with softer minimum requirements.

Finally, the architecture this work has proposed is scalable in number of ghosts, as in a crowd simulation. This means it is possible to increase the number of processors (CPUs) and the number of master processes. Moreover, two aspects of this work will extend as cloud computing server as mobile client developed based on native code.

References

ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R. H., KONWINSKI, A., LEE, G., PATTERSON, D. A., RABKIN, A., STOICA, I., AND ZAHARIA, M. 2009. Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb.

BARBOZA, D. C., JUNIOR, H. L., CLUA, E., AND REBELLO, V. E. F. 2010. A simple architecture for digital games on demand using low performance resources under a cloud computing paradigm. *Proceedings of the IX Brazilian Symposium on Computer Games and Digital Entertainment*, 38–45.

BARBOZA, D. C., JUNIOR, H. L., CLUA, E. W. G., AND REBELLO, V. E. 2010. A simple architecture for digital games on demand using low performance resources under a cloud computing paradigm. *Games and Digital Entertainment, Brazilian Symposium on 0*, 33–39.

CHEHIMI, F., AND COULTON, P. 2008. Motion controlled mobile 3d multiplayer gaming. In *ACE '08: Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*, ACM, New York, NY, USA, ACE, 267–270.

COCOS2D, 2011. Cocos2d. Avalible at: `http://www.cocos2d-iphone.org/games/`. 30/09/2011.

DALMAU, D. S. C. 2003. *Core Techniques and Algorithms in Game Programming*. New Riders Publishing.

DICKINSON, P., 2001. Instant replay: Building a game engine with reproducible behavior. Available at `http://www.gamasutra.com/features/20010713/dickinson_01.htm/` .

GABB, H., AND LAKE, A., 2005. Threading 3d game engine basics. Available at `http://www.gamasutra.com/features/20051117/gabb_01.shtml/` .

GLINKA, F., PLOSS, A., GORLATCH, S., AND MÜLLER-IDEN, J. 2008. High-level development of multiserver online games. *International Journal of Computer Games Technology 2008*, 5, 1–16.

JOSELLI, M., ZAMITH, M., CLUA, E., PAGLIOSA, P., CONCI, A., MONTENEGRO, A., AND VALENTE, L. 2008. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 115–120.

JOSELLI, M., ZAMITH, M., CLUA, E., LEAL-TOLEDO, R., MONTENEGRO, A., VALENTE, L., FEIJO, B., AND PAGLIOSA, P. 2010. An architeture with automatic load balancing for real-time simulation and visualization systems. *JCIS - Journal of Computational Interdisciplinary Sciences*, 207–224.

M1NDLAB, 2007. Alien revolt: Location-based massive-multiplayer online rpg. Avalible at: `http://www.alienrevolt.com`.

MÖNKKÖNEN, V., 2006. Multithreaded game engine architectures. Available at `http://www.gamasutra.com/features/20060906/monkkonen_01.shtml` .

PARK, A., AND JUNG, K. 2009. Flying cake: Augmented game on mobile devices. *Comput. Entertain. 7*, 1, 1–19.

PASSOS, E., JOSELLI, M., ZAMITH, M., ROCHA, J., MONTENEGRO, A., CLUA, E., CONCI, A., AND FEIJÓ, B. 2008. Supermassive crowd simulation on gpu based on emergent behavior. In *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 81–86.

ROHS, M. 2007. Marker-Based Embodied Interaction for Handheld Augmented Reality Games. *Journal of Virtual Reality and Broadcasting 4*, 5 (Mar.). `urn:nbn:de:0009-6-7939`, ISSN 1860-2037.

VALENTE, L., CONCI, A., AND FEIJÓ, B. 2005. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, 89–99.

W3C, 2011. The websocket api. Available at `http://dev.w3.org/html5/websockets/`.

WATTE, J., 2005. Canonical game loop. Available at `www.mindcontrol.org/~hplus/graphics/game_loop.html/` .

ZAMITH, M., CLUA, E., PAGLIOSA, P., CONCI, A., MONTENEGRO, A., AND VALENTE, L. 2007. The gpu used as a math co-processor in real time applications. *Proceedings of the VI Brazilian Symposium on Computer Games and Digital Entertainment*, 37–43.

ZYDA, M., THUKRAL, D., JAKATDAR, S., ENGELSMA, J., FERRANS, J., HANS, M., SHI, L., KITSON, F., AND VASUDEVAN,

V. 2007. Educating the next generation of mobile game developers. *IEEE Computer Graphics and Applications 27*, 2, 96, 92–95.

ZYDA, M. J., THUKRAL, D., FERRANS, J. C., ENGELSMA, J., AND HANS, M. 2008. Enabling a voice modality in mobile games through voicexml. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, ACM, New York, NY, USA, Sandbox, 143–147.