A Distributed Game Engine for Mobile Games on the Android Platform

Davide Gadia[1], Dario Maggiorini[1], Davide Puopolo[2], Laura Anna Ripamonti[1] and Luca Ziliani[2]
Department of Computer Science, University of Milan, via Comelico 39, I-20135 Milano, Italy
[1]{*firstname.lastname*}*@unimi.it,* [2]{*firstname.lastname*}*@studenti.unimi.it*

Keywords: Game Engines, Game Development, Mobile Platforms, Distributed Systems.

Abstract: In the last few years we have witnessed a tremendous change in the way game developers are required to deal with software production. We moved from small groups building the application ground-up to large coordinated teams with hierarchical organisation. To support this transformation, game developers are now using integrated development and execution environments called *game engines*. Among all possible gaming platforms, mobile ones are proving to be a challenging ground due to their intrinsic requirement for game engines to deploy the final application on a distributed system. In this paper we discuss about requirements for next-generation game engines for mobile devices. In particular, we propose a variation of the standard approach for game engines architecture pushing from a monolithic architecture toward a distributed one. In our solution, the mobile game engine becomes modular and lower the distinction between client and server side.

1 INTRODUCTION

In these last years, the way developers implement video games is undergoing a tremendous change. If we look to historical blockbusters for home entertainment such as Pitfall!, Tetris, and Prince of Persia we can see the name of a single developer. As a matter of fact, in the '80s, all aspects of video game development were usually managed by a single person or, at best, a very small group. Today, with the evolution of the entertainment market and the rise of projects with seven (or eight) figures budget, this situation calls for a drastically different approach. Video game development is now a distributed collaborative effort involving tens to hundreds of programmers. This increasing complexity of teams organisation and the tremendous growth of projects size force development teams to organise themselves in a hierarchical way and to adopt software engineering practices enforcing string code reusability. As a result, modern video games are implemented by means of software environments called *game engines*.

A game engine, as largely discussed in literature, is a complex framework made of two main building blocks: a tool suite and a runtime component. In particular, the runtime component assembles together all the internal libraries required for hardware abstraction and provides services for game-specific functionalities. A variable portion of the runtime is usually linked inside the game and distributed along with the binary executable. As a matter of fact, deployment for different platforms requires to embed specific runtimes wrapping the same *game content*. The game content is made up of rules and assets created by developers and artists, which get managed via the tool suite component. When developing a game for a mobile device, the aforementioned architecture may really become an hindrance since we have a distributed platform as target environment.

Mobile applications in general, and games in particular, have an intrinsic requirement to be deployed on a distributed system due to the always-online nature of mobile devices and a strong prerequisite to play (or, at least, interact) online with other players. Developing any kind of application for a distributed environment involves managing data transmission between heterogeneous platforms and monitoring real-time operations to prevent failures due to undelivered or delayed packets. In particular, as also discussed in (Festa et al., 2017), distributed debugging is a complex problem due to the synchronisation required between network nodes to correctly reconstruct the sequence of events leading to a malfunctioning feature.

With the increasing presence of distributed applications and frameworks in online services (i.e., cloud computing), the constraints introduced by mobile de-

Gadia, D., Maggiorini, D., Puopolo, D., Ripamonti, L. and Ziliani, L.
A Distributed Game Engine for Mobile Games on the Android Platform.
In *Proceedings of the International Conference on Computer-Human Interaction Research and Applications (CHIRA 2017)*, pages 142-149
ISBN: 978-989-758-267-7

vices are now spreading to all other platforms. Under these lenses, current research faces an important task: to understand if classical client-server architectures for games are up to the challenge or something more modular and flexible is required. We strongly believe that the classical approach will not be enough for the next generation of game developers, for two reasons. The first reason is poor scalability. As a matter of fact, modern games are pushing to involve thousands to millions of players in Massively Multiplayer Online Games (MMOGs). The resources used by an MMOG must be shared among users and scale in a seamless way with the offered workload. If a standard client-server approach is adopted, the server is in charge for data exchange between all clients. Such server will also become a single – global – point of failure and a serious bottleneck. Load shedding between servers may not be an option when dynamic information must be shared to a worldwide player population. The second reason is the client-server strict dependency from a well known, and always available, service provider. Next generation games should be able to exploit the network whenever available, while allowing single player mode when offline, with minimal experience degradation for the user.

As a result, we envision that there will be a strong demand for game engines with a flexible and modular architecture (such as the ones envisioned in (Maggiorini et al., 2015) and (Maggiorini et al., 2016)) which will deploy seamlessly on multiple platforms and allow component distributions over a network without asking for additional effort from the developer. A possible first step to understand how distributed game engine can be shaped in the future is to apply their model to the current mobile ecosystem (Gerla et al., 2013). For this reason, we propose here a modular architecture for game engines targeting the Android operating system.

To deploy our architecture on a distributed system, we lower the distinction between client and server architecture and make it evolve in a similar direction as a peer-to-peer network. Where required, a functionality can be provided both locally and remotely with the same API. Scalability will benefit from this approach thanks to transparent service-points relocation. A service may be provided from an arbitrary point of the peers mesh based on resources availability and response time. Service point relocation can be achieved using local policies leveraging on neighbours node discovery. As a result, from a game developer's point of view, no network management will be required and platform-dependent code can be limited to a minimum.

2 RELATED WORK

In the past, a fair number of scientific contributions has been devoted to improve the architecture of game engines. Nevertheless, at the time of this writing and to the best of our knowledge, only a very limited number of papers are specifically addressing issues related to scalability and cross-platform portability.

A significant share of existing literature seems to be focused on optimising specific aspects or services, such as 3D graphics (e.g., (Cheah and Ng, 2005)) or physics (e.g., (Mulley, 2013; Coumans, 2006; Maggiorini et al., 2014)).

Issues related to portability on different platforms have been addressed, among the others, by (Darken et al., 2005), (Guana et al., 2015), (Munro et al., 2009), and (Carter et al., 2010). Authors of (Darken et al., 2005) propose to improve portability by providing a unifying layer on top of other existing engines. In fact, they extend each architecture with an additional platform-independent layer and assume a share set of functionalities to be available on all platform. This approach is feasible for multi-platform deployment but may reduces performances and set all platforms to use a shared set of basic features. Authors of (Guana et al., 2015) focus on development complexity and propose a solution based on modern model-driven engineering while in (Munro et al., 2009) an analysis of the open source version of the Quake engine is performed with the purpose to help independent developers contribute to the project. In the first case, developers may not be able to implement any possible game mechanics while, in the second case, results are limited to a specific game and they stay on the same platform. A different approach has been followed by (Carter et al., 2010) where authors envision convergence to a web-based platform. In this case, we have to outline mobile devices are still lacking web-based 3D support and, also, many mobile features (e.g., bluetooth peer-to-peer connection) may not be accessible as part of the gaming experience.

If we focus strictly on the adoption of distributed systems as viable platforms for games, research is currently pushing toward two directions: improving network performances and providing distributed services.

Improvement of network performances is mostly related to reduce transmission latency by optimisation (Chen et al., 2016b) or offloading (Chen et al., 2016a; Zucchi et al., 2013) in order to achieve the same result as in a local system (Chen et al., 2014). Unfortunately, it has been proved long ago that Internet is not following a WFQ service model and a reliable estimation strategy for available resources is

still to come. For this reason, many papers are focusing on how existing game engines are adapting to the cloud (Messaoudi et al., 2015; Kim, 2013) or a distributed system in general (Deen et al., 2006) and evaluating player experience in these environments (Sabet et al., 2016; Suznjevic et al., 2016; Wen and Hsiao, 2014).

When providing distributed services, the main idea is to make available a game-related feature using the cloud (Aly et al., 2016). Many research efforts have been devoted to create distributed implementations of existing engines. Unfortunately, many of them apply a distributed system approach only to a specific internal service (such as simulation pipeline or shared memory (Gajinov et al., 2014; Lu et al., 2012)) while others offload non time-critical but computational-intensive tasks to the cloud (e.g., avatar animation (Li et al., 2015)). While these approaches are greatly increasing data processing power, they do not provide a solution to the problem of creating a completely distributed game engine.

3 BACKGROUND ON GAME ENGINES

Although game engines have been studied and perfected since mid-'80s, a formal and globally accepted definition is yet to be found for them. Despite this lack of definition, the function of a game engine is fairly clear: it exists to abstract the (platform-dependent) details of doing common game-related tasks, like rendering, physics, and input; so that developers can focus on implementing game-specific aspects.

In particular, inside a game engine we can find a runtime component. The runtime component is a collection of software libraries required for hardware abstraction. Since a variable portion of the runtime is usually linked inside the game or get distributed along with the executable, mobile devices pose serious problems. These problems may be coming from very specialised hardware drivers, different technical specifications such as screen size and aspect ratio, or limitations imposed by online market owners. There might be huge differences in term of capabilities for runtimes used in phones with different brands. Sometimes, even within different models under the same brand.

The drawback on the developer is that many times the game code must be platform aware and perform operations based on the underlying device due to feature missing or not working in the local runtime.

3.1 A Brief History of Game Engines

The first example of game engine dates back to 1984 with the game Doom (ID Software, 1993) Despite the fact Doom was not designed around the concept of game engine, it paved the way to a number of best practises for their definition. In the Doom architecture, we could observe a strict separation between software modules. To each module, a precise function and location in the architecture was assigned. In particular, there was clear distinction between code and data assets. This approach enforced code reusability and made for an easy portability on different platforms. The rules we just described are nowadays standard best practices for software engineering; however, it was not uncommon in the '80s for one-man projects to have hardware-specific self-modifying code. To actually see the first example of game engine we had to wait until mid-'90s with the release of Quake Engine (ID Software, 1996) and Unreal Engine (Epic Games, 1998). Quake and Unreal have not been designed as environments to hold a specific game rather than as collections of software components useful to create First Person Shooter (FPS) games. As a matter of fact, the business model of ID and Epic was not just about selling games to end users rather than selling an engine to other developers. Starting from this point, games opened up to user customisation and, most important of all, their engines have been regarded by software companies as an asset. As of today, we have companies making games and selling their internal technology along with companies (such as Unity Technologies (Unity Technologies, 2005)) focused only on distributing and supporting a game engine platform. Of course, game engines have not been immune to the *open source* movement. Many open source projects exist providing specialised gaming functionalities to be used in third parties projects (e.g., Ogre3D (Torus Knot Software, 2013) for graphics and Bullet (Coumans, 2006) for physics) or a complete application stack (e.g., Cocos2D (Chukong Technologies, 2010) and Torque (Garage Games, 2012)). Today, we can observe on the market a growing number of game engines with specific goals and adopting different approaches. In particular, for mobile devices, many of them focus on the optimisation of hardware performances and cross-platform deployment (such as the Marmalade game engine (Marmalade Technologies, 1998)) or try to achieve a distributed platform leveraging on the we infrastructure by means of javascript (see e.g., the Turbulenz game engine (Turbulenz l.t.d., 2009)).

4 A DISTRIBUTED ANDROID GAME ENGINE

As already mentioned in the previous sections, the relevant diffusion and success of mobile games is pushing the need for advanced, robust, and flexible game engines, which able to address the peculiar characteristics of the multiple available platforms and of the intrinsic difficulties of a distributed system. In this section, we present our proposal for an engine architecture for mobile games. Our aim is to address in a simple but effective way the different possible gaming configurations (online and offline) as well as roles (client and server) while feeing the developer from explicit network management.

The architecture described in the following subsection has been implemented in java using the Android Development Toolkit (ADK). By using ADK, our prototype is leveraging on the android hardware abstraction to provide a single-layer class library and to offer basic functionalities to create games. A game can just import our class library and provide its own asset management and game logic. The asset management part may be an ad-hoc implementation for the game or exploit a third party library; the android execution environment will take care of the integration.

To prove the feasibility of our approach, we used our class library to implement *True Believers Mobile*: an online distributed mobile game. As a matter of fact, the engine prototype can be used as a foundation for any kind of player vs player strategy game.

4.1 General Architecture

Our proposed architecture tries to disengage from the classical client-server approach: all functionalities should be present in every node and used seamlessly by the runtime component of the game engine. This way, we lower the boundaries between client and server components and create generic software nodes to build a distributed system. Each node is hosting both a server- and client-side component to be used depending on the situation. The client component can connect to a remote server (online mode) or locally (offline mode) to request gaming service. The software implementing the game client is not actually required to know – or manage – the actual service mode. The server component will accept one or more incoming connections to provide a gaming service. The software implementing the server is not required to know if a contacting client is local or remote. The server side must manage multiple connections in the most effective way, set up games between clients, collect and validate each action or request sent by clients, and then update the overall state of the game. In any case, more than one match may take place in a server at a given time. Each match will hold its own game state. The equivalent of a classic server is simply a software node where the client component is inactive.

Of course, one of the key element of any distributed architecture is the transmission reliability for game data. Since foundation classes are implemented directly in ADK, we can use both TCP or UDP transport protocols for data exchange. TCP offers a reliable transmission over packet switched networks by means of packets retransmission after a timeout. Despite the additional delay, game interactions time proved to be bounded to an acceptable level for strategy games. When dealing with strict real-time games, UDP may be preferred over TCP depending on game mechanic time constraints. Real-time gaming on a wide area network is usually addressed with specific game mechanics and prediction mechanisms. Both of these solutions are implemented in the higher levels of the application stack.

One important topic when dealing with distributed games is cheat prevention. We are not going to address this problem here since it is usually solved through code signing and peer authentication mechanisms in the network layer. Our prototype is, in fact, assuming a trusted network platform. Nevertheless, we acknowledge the importance of this problem and plan to address it in a future work.

4.1.1 Server Functionalities

In Fig. 1 we show a scheme representing the modules of the server component and their interconnections. Basically, the Connection Manager and Match Manager modules are instantiated at game server startup, while the other submodules are instantiated on request.

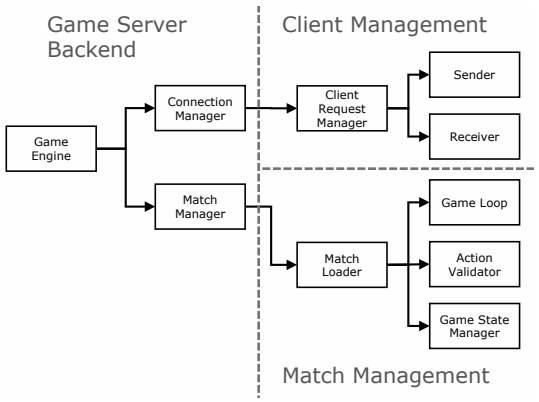


Figure 1: Dependencies of Game Server modules.

The role of the Connection Manager is to manage all the connection requests and, in case of acceptance (usually through an authentication process), to instantiate all the resources needed by each connection. After a connection is established, all messages exchanged between client and server are managed by a dedicated instance of the Client Request Manager. A Client Requests Manager manages the messages from and to a single client through a Sender and a Receiver modules. Once a request is received (e.g., to begin a new game, or to perform a particular action during a game), the message is forwarded to the Connection Manager, which, in turn, will contact the Match Manager module. The Match Manager module, and its submodules are the core of the server side of our architecture.

Similarly to the Connection Manager, the Match Manager is a management module and takes care of the instantiation and management of Match Loader modules; one for each active match. When the Match Manager receives a request to begin a new game, it checks if other players are available (using a matchmaking algorithm on the connected clients), and creates a new game by instantiating a Match Loader module. The Match Loader module is in charge of the actual creation and management of the resources for the specific match between the clients involved in the game. The actual management of a specific game is actually performed by the Game Loop, Action Validator, and Game State Manager submodules.

The Game loop module is the main component for the actual game. It manages the synchronization, and performs the simulations aimed at determining the new states for game and players. On the basis of the nature and complexity of the game, it can be structured in several dedicated submodules (e.g., for artificial intelligence, or for the automatic generation and validation of new levels), to be executed in different threads. Every time a client performs an action during a match, the Action Validator module checks for its validity. This module stores all the controls and procedures (i.e., the game rules). If the action is valid, the Action Validator module communicates the result to the Game State Manager. If the action is refused, the failure is notified through the Sender module, or some other feedback can also be considered (e.g., if cheating or hacking attempts are detected). The Action Validator must also validate actions and events generated by the Artificial Intelligence submodule of Game Loop module. Finally, the Game State Manager updates the overall game state, on the basis of the actions validated from the Action Validator, and uses the Sender module to synchronise game states on the client side.

4.1.2 Client Functionalities

In Fig. 2 a scheme presenting the modules hierarchy of the Client component is reported. As we can see,

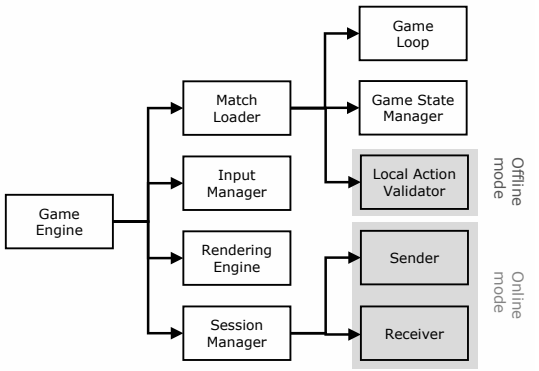


Figure 2: Dependencies of Game Client modules.

the same modular approach as in the server component is adopted. Moreover, a subset of modules is also shared between client and server sides. Remote or local instantiation and execution of these modules at runtime is based on the actual game mode (e.g., online multiplayer or offline single player). This approach allows for a great flexibility in the application of the engine to different games, and a more efficient implementation for the programmers, who have access to those functionalities through a common and unified API, with less efforts required in the explicit management of the network features.

The Input Manager and Rendering Engine modules are specific to the Game Client implementation. The Input Manager manages the input from the player (through keyboard, mouse, or other input devices), and on the basis of the type of the input and of the current state of the game, it communicates to the Session Manager module the message to be sent to the Game Server. The Rendering Engine manages all the aspects related to the graphics of the application: rendering loop, GUI updates, camera movements, animations, etc. With respect to the game complexity and need, it is usually subdivided in several specific submodules, each dedicated to a specific aspect. The final result generated by the Rendering Engine depicts the updated state of the game, once all the actions have been validated, and the corresponding results simulated and calculated.

Regarding the network modules, the execution context of the Game Client is very different than the one of the Game Server. In case of an online game, there is only one connection (to the Game Server) to manage. The Session Manager module has the same role and functionalities than the Client Requests Man-

ager module described before (including its Sender and Receiver submodules). Actually, from a development point of view, they can also provide the same APIs. When online, the Session Manager module receives information from the Input Manager and sends requests to a remote server via the Sender module. Answers, received through the Receiver module, are then sent to the Game State Manager module in order to update the local game state.

As it can be noticed from Fig. 2, the Game Client presents the same Match Loader module than the Game Server, with the same duties (i.e., the allocation and management of the local resources needed by the game from a client-side perspective). Same observation holds for the Game Loop and Game State Manager submodules.

In an online gaming session, the Game State Manager receives update from the Receiver module (after validation from the Action Validator on the server side) and updates the local game state. If the game is used offline, a Local Action Validator module is instantiated. This module shares the same code and most of the functionalities of the Action Validator module on the server side (even if some specific control or procedure may be different due to the different networking context), but, in this case, it communicates directly to the Game State Manager module. Moreover, on the basis of the nature of the game, some of the functionalities of the Game Loop (i.e., advanced artificial intelligence) may also need to be instantiated locally.

To better explain the characteristics and modularity of the presented approach, we present in Fig. 3 a diagram of the data flow between client and server. In the figure, the client side is depicted on the left while the server side (when playing online) is reported in the right side. If the client is operating offline, messages are routed through the Local Action Validator module while if the game is online Sender and receiver modules are involved and talking with the remote counterpart (large shaded rectangle in the center). About the big picture, there are two important things to point out. First, once the engine infrastructure is set up all the game code is just sitting in the Game Loop module. From a server-side perspective, the Game Loop will generate actions (to be validated) for the evolution of environment and Non-Playing Characters (NPCs) while, on a client-side perspective, it will react to player's input. The game loop will be the same on both side; actually, there will be only one Game Loop module since we are not implementing strictly client or server nodes. Second, for sake of clarity, Fig. 3 is not drawn in full. As a matter of fact, all modules are presents on both sides of the communication and it is possible to also have a player on the network node acting as server (hosting the game). In such case, the Input Manager module will be active also on the server side and the Game Loop will react to user input as well as generate NPCs actions.

4.2 True Believers Mobile

To test the feasibility of the proposed architecture, we have implemented an online Real-Time Strategy game called True Believers Mobile. True Believers Mobile uses the class library described in the previous section in order to deliver online functionalities to the user. Each game level is a maze populated by enemies, traps, and power-up items where the goal is to to reach a given position or to acquire an object within a given time limit. To do this, the player places her own characters in the maze in a strategical way. The game can be played by a single (offline) player, or against another online player. With respect to the proposed architecture, character availability for each level is managed by the Match Loader module. The Game Loop module implements a dedicated Artificial Intelligence submodule to manage NPCs behaviour and movement.

The maze description is managed by a specific submodule of the Match Loader called Maze Manager. This module is responsible to parse a configuration file and load all the needed resources. Moreover, it generates a new set of rules specific for the loaded maze and sends them to the Action Validator module. As already said, these procedures can be executed locally on the client, or remotely on the Game Server, without an explicit management from the developer.

Finally, a stand-alone Level Editor tool has been implemented in order to allow the players to design and generate their custom mazes to be used during game sessions with other players. The user is shown a schematic view of the maze with the possibility to define the characteristics of each position. Once completed, the level description is saved in a local store. In an online configuration, the new level is sent to the Game Server when starting a game with another online player. The Level Editor shares with the game engine a subset of modules needed for its execution; in particular, the Match Loader functionalities are used to load the graphical resources of the tool, and the Input Manager and Render Engine are needed to manage the user interaction and the editor interface. In Fig. 4(a) we show the Level Editor tool, while in Fig. 4(b) we show a screenshot of a True Believers Mobile game session based on the level generated by the Level Editor tool.

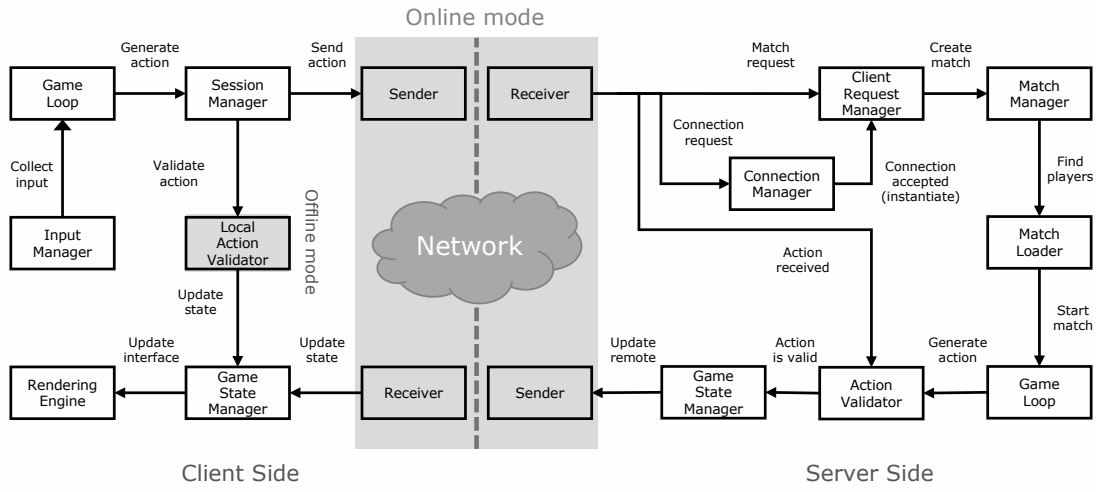
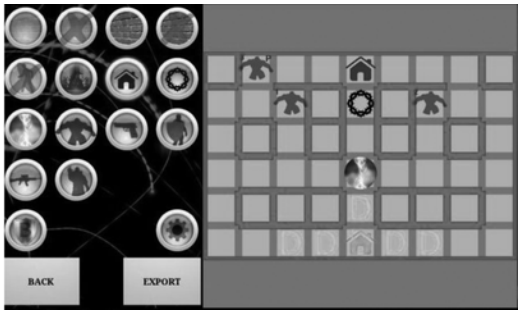


Figure 3: Client-Server data-flow diagram.



(a)

(b)

Figure 4: True Believers Mobile: level editor tool (a), and a screenshot of a game (b).

5 CONCLUSION AND FUTURE WORK

In this paper we discussed about the requirements needed by a next-generation game engine to support mobile devices and a distributed solution feasible for the Android platform has been proposed. In our solution, server and client functionalities are hosted in every node, and they are allocated and used by the runtime component of the game engine depending on the current game configuration (online or offline). The overall approach we suggest allows to blur the distinction between client and server roles and to evolve the system toward an architecture similar to a peer-to-peer network. This evolution does not require any longer the developer to manage the network by implementing a distributed system. In the manuscript, we also present a Real-Time Strategy game implemented using our architecture to demonstrate its feasibility.

The next step of our research aims to understand the relation between network performances and player experience as well as to understand how our architecture can dynamically adapt to different network technologies e.g., roaming from WiFi to LTE and back. Moreover, we are also planning to implement other kind of games on top of our proposed architecture in order to evaluate by comparison how various genres are going to benefit from a distributed game engine approach. Last but not least, cheating prevention mechanisms should also be integrated in our game engine architecture.

REFERENCES

Aly, M., Franke, M., Kretz, M., Schamel, F., and Simoens, P. (2016). Service oriented interactive media (soim) engines enabled by optimized resource sharing. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 231–237.

Carter, C., Rhalibi, A. E., and Merabti, M. (2010). Development and deployment of cross-platform 3d web-based games. In *2010 Developments in E-systems Engineering*, pages 149–154.

Cheah, T. C. S. and Ng, K. W. (2005). A practical imple-

mentation of a 3D game engine. In *Computer Graphics, Imaging and Vision: New Trends, 2005. International Conference on*, pages 351–358.

Chen, K. T., Chang, Y. C., Hsu, H. J., Chen, D. Y., Huang, C. Y., and Hsu, C. H. (2014). On the quality of service of cloud gaming systems. *IEEE Transactions on Multimedia*, 16(2):480–495.

Chen, M. H., Dong, M., and Liang, B. (2016a). Multi-user mobile cloud offloading game with computing access point. In *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, pages 64–69.

Chen, Y., Liu, J., and Cui, Y. (2016b). Inter-player delay optimization in multiplayer cloud gaming. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 702–709.

Chukong Technologies (2010). Cocos2D-x. http://cocos2d-x.org/.

Coumans, E. (2006). Bullet physic library. http://www.bulletphysics.org/.

Darken, R., McDowell, P., and Johnson, E. (2005). Projects in VR: the Delta3D open source game engine. *IEEE Computer Graphics and Applications*, 25(3):10–12.

Deen, G., Hammer, M., Bethencourt, J., Eiron, I., Thomas, J., and Kaufman, J. H. (2006). Running quake ii on a grid. *IBM Systems Journal*, 45(1):21–44.

Epic Games (1998). Unreal Engine. http://en.wikipedia.org/wiki/Unreal_Engine.

Festa, D., Maggiorini, D., Ripamonti, L., and Bujari, A. (2017). Supporting distributed real-time debugging in online games. In *2017 IEEE Consumer Communications and Networking Conference (CCNC)*.

Gajinov, V., Eric, I., Stojanovic, S., Milutinovic, V., Unsal, O., Ayguad, E., and Cristal, A. (2014). A Case Study of Hybrid Dataflow and Shared-Memory Programming Models: Dependency-Based Parallel Game Engine. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 1–8.

Garage Games (2012). Torque-3D. http://www.garagegames.com/products/torque-3d.

Gerla, M., Maggiorini, D., Palazzi, C., and Bujari, A. (2013). A survey on interactive games over mobile networks. *Wireless Communications and Mobile Computing*, 13(3):212–229.

Guana, V., Stroulia, E., and Nguyen, V. (2015). Building a Game Engine: A Tale of Modern Model-Driven Engineering. In *Games and Software Engineering (GAS), 2015. IEEE/ACM 4th International Workshop on*, pages 15–21.

ID Software (1993). DOOM. http://en.wikipedia.org/wiki/Doom_(1993_video_game).

ID Software (1996). Quake Engine. http://en.wikipedia.org/wiki/Quake_engine.

Kim, H. Y. (2013). Mobile games with an efficient scaling scheme in the cloud. In *2013 International Conference on Information Science and Applications (ICISA)*, pages 1–3.

Li, M., Cai, W., Wang, K., Hong, J., and Leung, V. C. M. (2015). Prototyping decomposed cloud software: A case study on 3d skeletal game engine. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 192–195.

Lu, H., Yijin, W., and Hu, Y. (2012). Design and implementation of three-dimensional game engine. In *World Automation Congress (WAC), 2012*, pages 1–4.

Maggiorini, D., Ripamonti, L. A., and Cappellini, G. (2015). About Game Engines and Their Future. In *Proceedings of EAI International Conference on Smart Objects and Technologies for Social Good (GOODTECHS 2015)*, pages 1–6.

Maggiorini, D., Ripamonti, L. A., and Sauro, F. (2014). Unifying Rigid and Soft Bodies Representation: The Sulfur Physics Engine. *International Journal of Computer Games Technology*, pages 1–12.

Maggiorini, D., Ripamonti, L. A., Zanon, E., Bujari, A., and Palazzi, C. E. (2016). SMASH: A distributed game engine architecture. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 196–201.

Marmalade Technologies (1998). Marmalade game engine. https://marmalade.shop/en/.

Messaoudi, F., Simon, G., and Ksentini, A. (2015). Dissecting games engines: The case of unity3d. In *2015 International Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6.

Mulley, G. (2013). The Construction of a Predictive Collision 2D Game Engine. In *Modelling and Simulation (EUROSIM), 2013 8th EUROSIM Congress on*, pages 68–72.

Munro, J., Boldyreff, C., and Capiluppi, A. (2009). Architectural studies of games engines: The quake series. In *Games Innovations Conference, 2009. ICEGIC 2009. International IEEE Consumer Electronics Society's*, pages 246–255.

Sabet, S. S., Hashemi, M. R., and Ghanbari, M. (2016). A testing apparatus for faster and more accurate subjective assessment of quality of experience in cloud gaming. In *2016 IEEE International Symposium on Multimedia (ISM)*, pages 463–466.

Suznjevic, M., Slivar, I., and Skorin-Kapov, L. (2016). Analysis and qoe evaluation of cloud gaming service adaptation under different network conditions: The case of nvidia geforce now. In *2016 Eighth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–6.

Torus Knot Software (2013). Ogre. http://www.ogre3d.org/.

Turbulenz l.t.d. (2009). Turbulenz game engine. http://biz.turbulenz.com/.

Unity Technologies (2005). Unity3D. http://unity3d.com/.

Wen, Z. Y. and Hsiao, H. F. (2014). Qoe-driven performance analysis of cloud gaming services. In *2014 IEEE 16th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6.

Zucchi, A. C., Gonzalez, N. M., d. Andrade, M. R., d. F. Pereira, R., Goya, W. A., Langona, K., d. B. Carvalho, T. C. M., and Mngs, J. E. (2013). How advanced cloud services can improve gaming performance. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 289–292.