Towards Application Programming Interfaces for Cloud Services Orchestration Platforms in Computer Games

Markus Schatten, Igor Tomičić, Bogdan Okreša Đurić
Artificial Intelligence Laboratory
University of Zagreb
Faculty of Organization and Informatics
Pavlinska 2, 42000 Varaždin, Croatia
{markus.schatten,igor.tomicic,dokresa}@foi.unizg.hr

Abstract. *Services orchestration platforms which allow for the implementation of scalable, distributed and containerized multi-agent systems are gaining momentum. Lots of contemporary computer games, especially massively multi-player on-line (MMO) games and game streaming platforms are such systems and can have major benefits in using orchestration platforms. In order for a game to use such a platform its game engine has to have the necessary prerequisites including an application programming interface (API) which we analyze in this paper. Additionally, we provide four proof-of-concept implementations of such APIs in four different game development platforms (namely Godot, RPG Maker, Ren'Py and Blender Game Engine).*

Keywords. services orchestration, computer games, multi-agent systems, game engine, application programming interface

1 Introduction

The challenge of building scalable, robust, large applications that can be incrementally upgraded, yielded a need for a concept called microservices (Fowler and Lewis, 2014). Within such concept, the core functionality of an application is decomposed into many smaller units, usually packed as containers, that are in some way able to communicate within themselves.

As (Khan, 2017) notes, such applications can be "composed of clusters of hundreds of instances of containerized services", and the cluster of containers must be "fault tolerant, available, and potentially geographically dispersed". Because of the sheer complexity of such systems, the notion of orchestration platforms became highly significant within the domain – such platforms can simplify container management and are generally used for integrating and managing containers at scale. Some of the general container orchestration platforms in use at the time of this writing are Kubernetes, Amazon Elastic Container Service, Docker Swarm, Mesosphere Marathon, RancherOS, etc.

Blanchet et al. (Blanchet et al., 2005) introduce an intelligent-agent framework where web services are wrapped in a conversation layer and service orchestration is presented as a conversation among intelligent agents, each of which is responsible for delivering services. Their effort was to avoid inflexible composition evolution where all parties must be updated concurrently in order to avoid interaction problems.

Hahn and Fischer (Hahn and Fischer, 2007) present a model-driven approach to design interoperable holonic multiagent systems in Service-oriented Architectures, illustrating how a choreography model can be easily conceptually implemented through holons.

On the other hand, computer games as applications are usually monolithic with a thick client handling most of the computations. Network based games including massively multi-player on-line (MMO) and streamed games only recently shifted their attention to microservices, and there are not many reports on using orchestration platforms for game servers and game system architectures. For example (Yahyavi and Kemme, 2013) note only three types of system architectures for MMO games: (1) client-server; (2) distributed multi-server and (3) peer to peer. The distributed multi-server architecture is most closely related to the concept of microservices, but not equivalent. According to (Yahyavi and Kemme, 2013) there are two types of such an architecture: (a) shards - each server contains an own copy of the game world and (b) one world - each server contains only one part of the game world. An industry expert (Walker, 2020) mentions even six types of game server architecture patterns:

Monolithic Architecture resembles a server on which the entire game is implemented in a single process on a single computer. First-person-shooters like *Quake 2*, *Unreal Tournament*, *Counter Strike* and *Call of Duty* use such a pattern.

Distributed Network Connections pattern usually implements a server which accepts and manages client connections in order to multiplex client

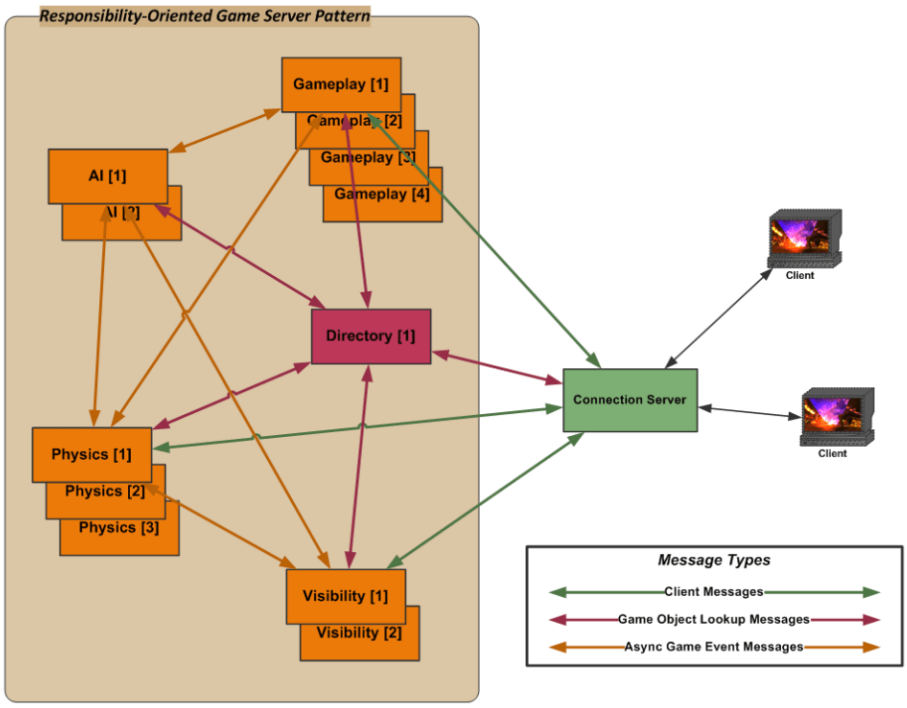

Figure 1: Responsibility-Oriented Game Server (Walker, 2020)

network traffic to several game servers (other names for such a server are connection server, user server, gateway server, front-end server or player server). Most commercial MMO games use this kind of pattern.

Client Side Load Balancing is an implementation pattern in which the game client contains a mechanism to randomly choose between a number of available connection servers described above. Some large-scale MMO game solutions use this kind of pattern.

Map-Centric Game Server (Walker, 2020) describes this pattern as follows: "*Develop a map-centric game server strategy that concentrates core game play activity with the maps in which it occurs. Do this by creating two server types, area server and world server. The game's server cluster contains many area server processes connected to a single central world server process. One or more connection servers generally manage client network connections, but are not part of this pattern.*". From our perspective these area and world server types could be viewed as service types in an orchestrated architecture. Such an implementation pattern is used by *Richard Garriott's Tabula Rasa*, *Hero Engine* (an MMO server framework) and *Face of Mankind*.

Seamless World Game Server is a variant of the previous pattern in which the world map is divided into smaller region maps distributed across a uniform server grid. Such a pattern is used by games like *Ultima Online*, *Ultima Online 2* (cancelled), the mentioned *Hero Engine* and *Star Wars Galaxies*.

Responsibility-Oriented Game Server is a pattern in which server types responsible for specific subsets of game play functionality (like gameplay, artificial intelligence (AI), visibility, physics, game state directory etc.) are defined. Such a pattern is used by *Rift (Trion Worlds)* and *Lineage*. This type of architecture closely resembles the idea of a microservices architecture as shown on figure 1.

In the following we will argue that orchestration platforms, especially when modeled and implemented as holonic and organization centered multiagent systems (MASs) can profoundly benefit the implementation of game server architectures by introducing various *out-of-the-box* features which are usually manually implemented in MMO game servers.

The rest of this paper is organized as follows: firstly in section 2 we provide an overview of requirements for both orchestration platforms and game engines to make a large-scale system work. In section 3 we provide four proof-of-concept game implementations that allow using existing microservices on any given orchestration platform. In the end in section 4 we draw our conclusions and provide guidelines for future research.

2 Requirements Definition

As already outlined in the introduction, a microservices orchestration platform allows for building scalable, robust, large applications that can be incrementally upgraded (Fowler and Lewis, 2014). In order to implement a computer game that make use of such an architecture, there are a number of requirements that have to be adhered and a number of features the actual game engine has to offer.

In the following we will describe a cloud service orchestration platform as a holonic multi-agent organisation or holarchy (see Horling and Lesser, 2005; Rodriguez et al., 2011). The latter idea, describing services as holons (agent that can be comprised of other agents), provides for the mechanism of nesting services, i.e. making it possible for the platform to run services either as individual services (agents) or complex organizations of agents. In this way it is possible to define multiple levels of holonic organization (see Schatten, 2014 and Schatten, 2013 for details on this approach).

The platform that contains several redundant agents is therefore the back-end of the platform described in this paper. Such agents consist of a chosen process, an input, and an output and can be instanced on demand – usually when load on a given agent becomes to high for it to process it adequately. Holons are, therefore, the structure of the back-end module of the cloud services orchestration platform - they take care of instancing agents when there is need, provide communication and coordination facilities and load balancing. Usually orchestration platforms are *invisible* to the client – the client communicates with a service (an agent in our case) which might be just one such service in an ensemble of agents in the background. Thus, any service that could be offered through the network can be orchestrated in this way, and this includes various aspects of gameplay. Thus, for a game engine in order to communicate with a microservice orchestration platform it needs to feature a simple application programming interfaces (APIs) that is able to communicate with a service needed by the game engine. Most prominent possible APIs are representational state transfer (REST), Web Socket or NetCat (simple TCP or UDP connections) that will enable developers to implement connections between their game and the cloud services orchestration platform. Therefore, in most game development engines all necessary prerequisites are already established since most engines allow for various types of network connections.

On the other hand, in order to make a game that puts an orchestration platform to good use, there are a number of requirements on the actual game architecture. The most important one is modularity – possibly implemented to an actor model (which is basicaly an agent model just in terms of game development usual terminology) in which each actor connects to some kind of service (agent) in order to act upon the game world and react to certain game events. Thus, an agent based approach to implementing both games and their necessary server architecture allows us to handle fairly complex terminology in terms of agents communicating. coordinating, competing etc.

Networked game server architectures including MMO and game streaming services can have major benefits from using a containerized microservices orchestration platform. Khan, 2017 outline that there are seven key capabilities of a container orchestration platform:(1) cluster state management and scheduling, (2) providing high availability and fault tolerance, (3) ensuring security, (4) simplifying networking, (5) enabling service discovery, (6) making continuous deployment possible, (7) providing monitoring and governance – all of which would have to be implemented manually when implementing a game server architecture from scratch. Thus, orchestration services provide game backend developers and administrators with a higher-level of abstraction in implementing their desired architecture.

3 Proof-of-concept Examples

In the following we will present four proof-of-concept APIs which were made in four different game engines for a number of games which have been developed or are in development. As already stated most general game engines already have the necessary prerequisites for communicating with services. Herein we will show how such communication can be established with four game engines: Godot, RPG Maker, Ren'Py and Blender Game Engine.

3.1 Godot

Godot is a cross-platform, free and open-source game engine that allows for the implementation of both 2D and 3D games (Wikipedia contributors, 2020b). Figure 2 shows the main graphical user interface (GUI) of Godot when a game project is loaded.

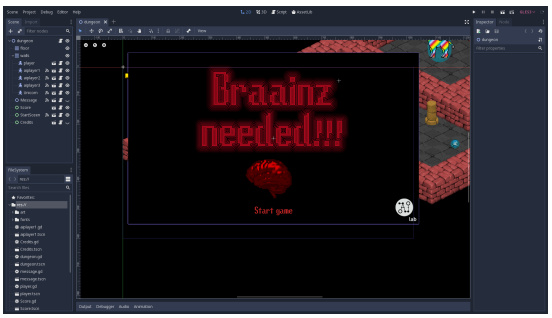

Figure 2: Godot – main user interface

As an example we have created a simple role-playing game (RPG) like game in which each non-player character (NPC) interacts with the player and gives her/him

simple tasks to solve. Figure 3 shows the main screen of the game.[1].

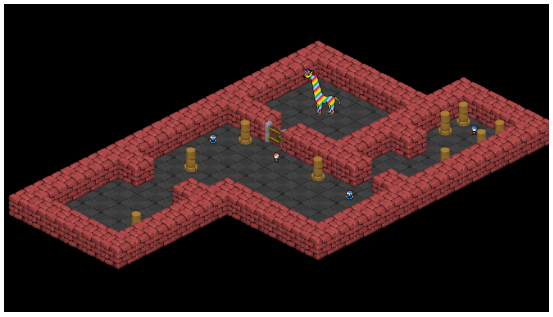

Figure 3: Godot – proof-of-concept game *Braainz needed!!!*

Godot can use a number of progamming languages to implement the actual game, but the default language is GDScript. Listing 1 shows how communication to a server can be implemented by using the built-in `HTTPRequest` object.

Listing 1: Excerpt from game code related to server connection

```
func get_message():
 var data = RQ_RESULT.result
 if data:
  data[ 0 ][ "msg" ] = data[ 0 ][ "msg"
      ↪ ].replace( '\\n', '\n' )
  if data[ 0 ][ "cmd" ] == "giveItem":
   emit_signal("give_item",NAME)
   DONE = true
  return data[ 0 ]
 return { "msg":"Zombies ate my brain!!!\
     ↪ nCan you program me a new one?\
     ↪ nPretty please ...?", "btns":"
     ↪ Exit", "cmd":"null"}

func send_request( caller ):
 CALLER = caller
 $HTTPRequest.request("http://localhost
     ↪ :2709/query/" + NAME.
     ↪ percent_encode() + "/" +
     ↪ LAST_COMMAND)

func _on_HTTPRequest_request_completed(
    ↪ result, response_code, headers,
    ↪ body):
 RQ_RESULT = JSON.parse(body.
     ↪ get_string_from_utf8())
 CALLER.show_message( self )
```

3.2 RPG Maker

RPG Maker is a tool for creating 2D RPG games (Wikipedia contributors, 2020d). Figure 4 shows the main interface of RPG Maker MV.

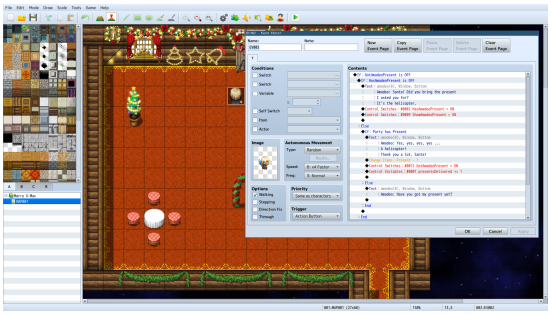

Figure 4: RPG Maker – main user interface

In its current version (RPG Maker MV) the scripting language of choice is JavaScript which makes communication to various types of services fairly easy. For this proof-of-concept game we have implemented a simple greeting card game for Christmas shown on figure 5.[2]

Figure 5: RPG Maker – proof-of-concept game *Marry X-Mas*

In order to communicate to a REST service, one can use the built-in `XMLHttpRequest` and `JSON` objects to communicate and parse results respectively, as shown in listing 2.

Listing 2: Excerpt from game code related to server connection

```
$gameMessage.setFaceImage('Actor1',0);
$gameMessage.setBackground(1);
$gameMessage.setPositionType(2);
let xhr = new XMLHttpRequest();
xhr.open('GET', 'http://localhost:5000/
    ↪ query/ivek', false);
xhr.send();
var ans = JSON.parse( xhr.response )
$gameMessage.add( ans[ 'msg' ] )
```

3.3 Ren'Py

Ren'Py is an open source visual novel engine written in Python (Wikipedia contributors, 2020c). The main

[1]The game uses art by Deesiv @ Deviantart, and is based on an example game by Filipe Mice.

[2]The game contains a number of game resources avaliable online, namely Deep male burp sound by Mike Koenig, Jingle Bells by Kevin MacLeod, Christmas sprites by PandaMaru, Santa face by Idril's Grove, Santa sprite by slimmmeiske2.

interface of Ren'Py is shown on figure 6.

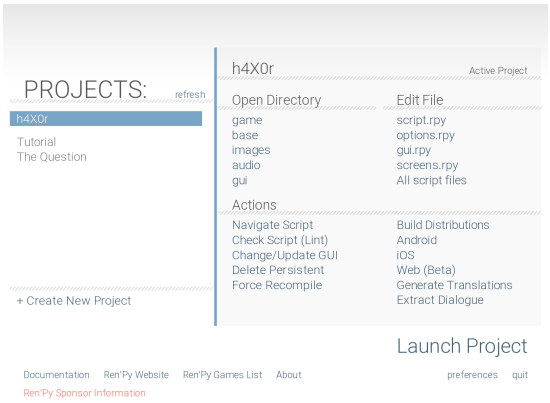

Figure 6: Ren'Py – main user interface

As a proof-of-concept game we have implemented a visual novel game (which is still work in progress) shown on figure 7.

Figure 7: Ren'Py – proof-of-concept game *h4X0r*

Ren'Py uses its *Animation and Translation Language* and its *Screen Language* for game implementation, but it is possible to embed Python code into the script. Listing 3 shows an example of how Python code inside Ren'Py script can be used to access a REST service by using Python's `urllib` and `json` modules.

Listing 3: Excerpt from game code related to server connection

```
init python:
 import urllib.request, json
 def get_request( req ):
  with urllib.request.urlopen( "http://
      ↪ localhost:8001/q=%s" % req ) as
      ↪ url:
   data = json.loads( url.read().decode()
       ↪ )
   return data

 config.grequest = get_request
```

3.4 Blender Game Engine / UPBGE

Blender Game Engine (BGE) component of Blender, a free and open-source 3D production suite, used for making real-time interactive 3D and 2D games (Wikipedia contributors, 2020a). Since BGE was discontinued, UPBGE (Uchronia Project Blender Game Engine) which is a fork of Blender has been developing the engine further. The main user interface (UI) of UPBGE is shown on figure 8.

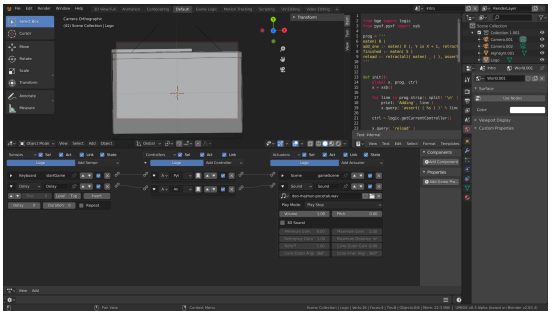

Figure 8: Blender Game Engine – main user interface

We have created a small proof-of-concept game in UPBGE shown on figure 9.[3]

Figure 9: Blender Game Engine – proof-of-concept game *Monkey Doghnut*

Since BGE and consequently UPBGE uses Python as its main implementation and scripting language, the implementation of a sensor communicationg with a service is straightforward and similar to the Ren'Py implementation (listing 4).

Listing 4: Excerpt from game code related to knowledge base connection

```
from bge import logic
import urllib.request, json

def query( req ):
 with urllib.request.urlopen( "http://
     ↪ localhost:8001/q=%s" % req ) as
     ↪ url:
  data = json.loads( url.read().decode()
      ↪ )
  return data
```

[3]The game uses the Public Domain image of a monkey by frankes.

```
def main():
 if ctrl.sensors[ 'eating' ].positive:
  act = ctrl.actuators[ 'njam' ]
  ctrl.activate( act )
  x.query( 'add_one' )
  result = x.query( 'finished' )
  if result:
   act = ctrl.actuators[ 'finish' ]
   ctrl.activate( act )
```

4 Conclusion

In this paper we have argued that containerized microservices orchestration platforms can benefit game server architectures on a number of ways by allowing to use already established services provided by the orchestration platform. We have shown that both the backend server-side as well as the actual game engine can (and should) be modeled and implemented as a (holonic) multi-agent system. In this way the same simplified and much more human understandable abstraction is used in both sides of the development.

We have also argued that most general game engines already provide the necessary prerequisites for implementing game infrastructures as orchestrated systems, namely: (1) support for actor model (i.e. agent model) and (2) network communication abilities.

In the end we have provided four proof-of-concept games implemented in four different game engines to show how simple it is to implement orchestration server communication from a game engine, since orchestration server behave as simple REST, WebSocket or even Netcat (TCP / UDP) services. Additionally, programming examples have been shown for the interested reader to try out.

Our future research will be focused on implementing valid APIs for a number of game engines to communicate with an orchestration platform focused on hybrid artificial intelligence (HAI) services that shall be implemented as part of the O_HAI (4) Games project sponsored by the Croatian Science Foundation.

Acknowledgement

This work has been fully supported by the Croatian Science Foundation under the project number IP-2019-04-5824.

References

Blanchet, W., Stroulia, E., & Elio, R. (2005). Supporting adaptive web-service orchestration with an agent conversation framework, In *Ieee international conference on web services (icws'05)*. IEEE.

Fowler, M., & Lewis, J. (2014). Microservices a definition of this new architectural term. *URL: http://martinfowler. com/articles/microservices. html*, 22.

Hahn, C., & Fischer, K. (2007). Service composition in holonic multiagent systems: Model-driven choreography and orchestration, In *International conference on industrial applications of holonic and multi-agent systems*. Springer.

Horling, B., & Lesser, V. (2005). A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, *19*(04), 281. https://doi.org/10.1017/S0269888905000317

Khan, A. (2017). Key characteristics of a container orchestration platform to enable a modern application. *IEEE cloud Computing*, *4*(5), 42–48.

Rodriguez, S., Hilaire, V., Gaud, N., Galland, S., & Koukam, A. (2011). Holonic Multi-Agent Systems (G. Di Marzo Serugendo, M.-P. Gleizes, & A. Karageorgos, Eds.). In G. Di Marzo Serugendo, M.-P. Gleizes, & A. Karageorgos (Eds.), *Self-organising Software: From natural to Artificial Adaptation*. Berlin, Heidelberg, Springer. https://doi.org/10.1007/978-3-642-17348-6_11

Schatten, M. (2013). Reorganization in multi-agent architectures: An active graph grammar approach. *Business Systems Research Journal*, *4*(1), 14–20.

Schatten, M. (2014). Organizational architectures for large-scale multi-agent systems' development: An initial ontology, In *Distributed computing and artificial intelligence, 11th international conference*. Springer.

Walker, M. (2020). Game server architecture patterns [[Online; accessed 15-July-2020]].

Wikipedia contributors. (2020a). Blender game engine — Wikipedia, the free encyclopedia [[Online; accessed 17-July-2020]]. https://en.wikipedia.org/w/index.php?title=Blender_Game_Engine&oldid=967025085

Wikipedia contributors. (2020b). Godot (game engine) — Wikipedia, the free encyclopedia [[Online; accessed 15-July-2020]].

Wikipedia contributors. (2020c). Ren'py — Wikipedia, the free encyclopedia [[Online; accessed 15-July-2020]]. https://en.wikipedia.org/w/index.php?title=Ren%27Py&oldid=960243958

Wikipedia contributors. (2020d). Rpg maker — Wikipedia, the free encyclopedia [[Online; accessed 15-July-2020]]. https://en.wikipedia.org/w/index.php?title=RPG_Maker&oldid=967986970

Yahyavi, A., & Kemme, B. (2013). Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Computing Surveys (CSUR)*, *46*(1), 1–51.