UPPSALA
UNIVERSITET

Kubernetes for Game Development

Evaluation of the Container-Orchestration Software

Faculty of Arts
Department of Game Design

Author: Jonas Lundgren

Bachelor Thesis in Game Design, 15 credits

Program: Game Design & Programming

Supervisor: Masaki Hayashi

Examiner: Mikael Fridenfalk

June, 2021

Abstract

Kubernetes is a software for managing clusters of containerized applications and has recently risen in popularity in the tech industry. However, this popularity seems to not have spread to the game development industry, prompting the author to investigate if the reason is a technical limitation. The investigation is done by creating a proof-of-concept of a simple system setup for running a game server in Kubernetes, consisting of the Kubernetes-cluster itself, the game server to be run in the cluster, and a matchmaker server for managing client requests and creation of game server instances. Thanks to the successful proof-of-concept, a conclusion can be made that there is no inherent technical limitation causing its infrequent use in game development, but most likely habitual reasons in combination with how new Kubernetes is.

Keywords: containerization, game server, Kubernetes, virtualization

Table of Contents

Glossary

API	**A**pplication **P**rogramming **I**nterface, code base meant to be designated entry points for programmers to interface with the program.
Blade server	A particular physical server form factor and design.
C#	A programming language developed by Microsoft.
Client	A computer or software which connects to a server.
Cluster	Several nodes which run a specific software, usually distributed, to allow for redundancy and increased performance. Can be virtual or physical, or a mix.
CNCF	**C**loud **N**ative **C**omputing **F**oundation, a foundation under the Linux Foundation which focuses on incubating and developing software for use in computing native to cloud environments.
Container	A separated environment for an application, usually consisting of the application itself as well as the dependencies it has to be able to run.
containerd	A container runtime for Linux and Windows, which allows for the running of containers. Stylized with lowercase c.
Containerization	The act of creating, running and managing containers.
Data center	A room or building dedicated to physical servers and other related networking equipment in rows of rack cabinets.
Dedicated game server	A server application design where the server runs the game simulation itself with data input from the clients followed by output to the client after advancing the game simulation.
Docker	A software for containerization, developed by Docker Inc.
(API) Endpoint	The actual connection point of an API, which programs utilize to obtain, modify or remove data. For example, an endpoint named GetUser would return data for a user.
Flask	A HTTP framework written in Python.
FOSS	**F**ree **O**pen-**S**ource **S**oftware, software where the source code is freely available to use, copy, modify, study, etc.
Framework	A premade software intended to ease the development of new programs by providing a skeleton with certain functionality. This lets a developer focus on creating new functionality by filling out the “meat” of the skeleton.
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol, the data protocol used most commonly for web applications.
Hypervisor	Software responsible for managing and running virtual machines.
(Docker) Image	The complete software package needed to run a containerized application. Typically consists of a base image, into which all the code, libraries, software, etc., that the application needs to run is added.
Kubernetes/K8s	Software for managing clusters of containers. K8s is the shortened name of Kubernetes: K-8 letters-s, pronounced “Kates”.

Library	Similar to a framework, but usually focused on solving one or more specific problems.
.NET Core	An open-source cross-platform software framework for applications such as C#, currently developed by the .NET Foundation. Successor to the original .NET Framework.
Node	Another name for a server, often in a cluster. Can be either virtual or physical.
Python	Programming language currently being developed by the Python Software Foundation.
Server	A computer or application which serves data to a client.
Ubuntu	Operating system of the Linux variety, based off Debian. Currently developed and maintained by Canonical Ltd.
YAML	**Y**AML **A**in't **M**arkup **L**anguage, a human-readable data-serialization language.

This page is left intentionally blank.

1. Introduction

Running multiple applications on a single physical server became overly complex since applications would compete for resources such as processing power and memory but also conflicting libraries of code and operating system tweaks. Therefore, a technology known as *virtualization* was invented to separate applications and their dependencies from one another. Each separate instance can then be managed separately without any impact on the other small environments, both in resource allocation as well as the data inside.

One of the technologies under the term virtualization is *containerization*, where an application is separated into a container along with only its most required dependencies. Said container also has a pre-defined start point, making the application act the same way each time the container is started, no matter which hardware it is run on.

Eventually, organizations started getting too many containers to effectively manage by hand. Therefore, more technology was created in order to manage them. One of these technologies is Kubernetes, a free open-source software which was created by Google in 2014, building upon their numerous years of experience running data centers and containerized software.

Kubernetes let developers and engineers define ahead of time how a container should be run, such as how many simultaneous instances of the container should be active, how many resources a container is allowed to use, auto-scaling depending on load, and so on. Kubernetes would then do its best to match those requests, making it a very flexible tool to use.

Containerization on its own also lets developers focus on actually developing things, since they can be certain that their application will behave in the same way when deployed as it does during development, and there will not be issues caused by another application running on the same server.

Kubernetes is used for a wide number of purposes, such as CERN using it to analyze their data, or Spotify using it to manage their backend. The author noticed, however, that there was seemingly a lack of game companies among the numerous success stories relating to Kubernetes and containerization. This prompted the author to investigate whether the reason stems from a technical reason where Kubernetes and games were incompatible with one another.

This is especially relevant due to the ever-increasing popularity of games with online connectivity. Using a standardized server as a baseline with configuration done on the fly makes for a more uniform and simpler backend for a given game, where game instances are deployed when needed rather than having a bunch of instances idling in order to match peaks in load during certain times of day.

To investigate this, the author created a proof-of-concept consisting of a simple containerized game server to be run in Kubernetes and a matchmaker. The matchmaker is responsible for taking a request for a game instance from a client and requesting Kubernetes to create more instances if none are available. This setup acts as an example of how such a system might work in an actual game.

Based on the experience of creating the proof-of-concept, possible reasons for the infrequent use of Kubernetes for games beyond the technical issues is also discussed.

As a conclusion, there were no technical reasons found for Kubernetes' infrequent use in games, but rather hints of it being a cultural thing along with the newness of Kubernetes where organizations are simply not aware of it existing.

In this thesis, the three modern ways of deploying an application are explained followed by a high-level explanation of Kubernetes and its components as well as some examples of Kubernetes being used, both outside and within the games industry. The design of the proof-of-concept is explained and based on the results, along with discussions regarding games and Kubernetes, the final conclusion is drawn.

2. Background

Presently there are three main ways of hosting applications: on physical servers, in virtual machines or in virtual containers, with the specifics of each hosting option being discussed in the respective section of this chapter. A fourth method of hosting known as cloud hosting is in practice just hosting your apps on someone else's servers – be they physical, virtual or containerized – and is for this thesis considered synonymous to those.

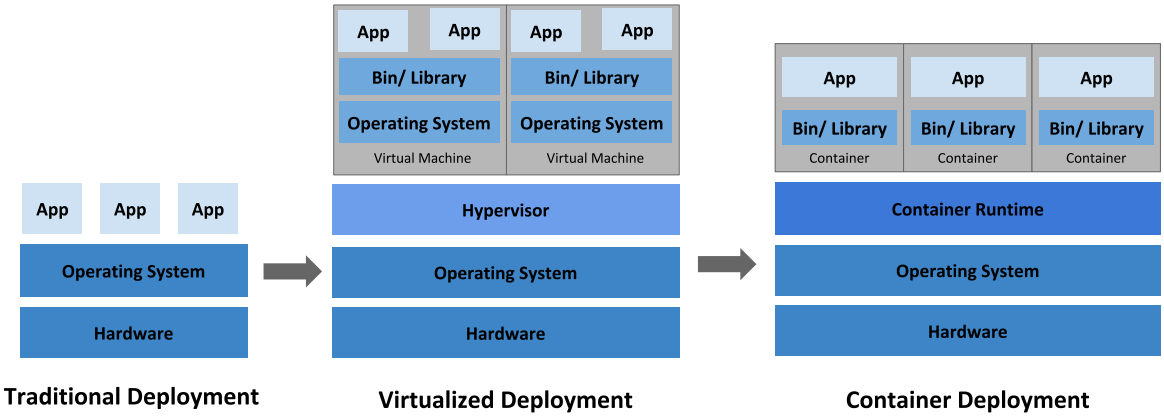

Figure 1: Visualization of application deployment evolution [8]

Application deployment has evolved over the years. As shown in Figure 1, it has transitioned from traditional development using physical servers, to a virtualized deployment using virtual machines running on physical servers, to running containers on either virtual or physical machines.

2.1 Physical servers

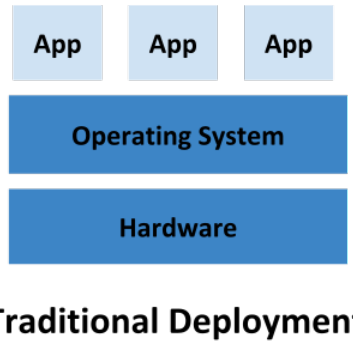

Figure 2: Visualization of applications on physical servers [8]

Physical servers (also referred to as *bare-metal* or just *metal*) are fairly self-explanatory, in essence consisting of an everyday computer but with more throughput thanks to noise and heat not being as big of an issue. Deploying an application to a physical server only requires an operating system on top of the hardware, and multiple applications can run on the same physical server.

Physical servers do come with some drawbacks:

- Cost: a physical server is generally fairly expensive on its own, and usually comes with additional costs such as support contracts and part replacements.

- Space: since a server is just that – physical – it takes up actual space which needs to be accounted for. Additionally, power and network need to be available.
- Time: physical servers require some maintenance from time to time, in particular part replacements (fans, hard drives and power supply units are common replacements)
- Resource allocation: it is hard to allocate or limit resources between applications. If multiple applications were running on the same server, they could end up fighting over resources and negatively affect each other. To work around this, a physical server could run only one application, but that approach does not scale particularly well due to the previous points mentioned as well as not fully utilizing the available resources of each server.

Physical servers can be the right choice if there is a team of technicians and engineers who can optimize both the hardware and the application to ensure that each server is being used to its fullest extent. Despite the drawbacks, every data center needs the physical hardware to be able to provide virtual machines or containers.

Application examples for the physical machine pattern are the actual simulation-running parts of a game. These could be dungeon instances in games such as World of Warcraft, or the actual match in a game such as Dota 2 or Counter-Strike where calculations need to be done quickly to provide a smooth gaming experience.

2.2 Virtual machines

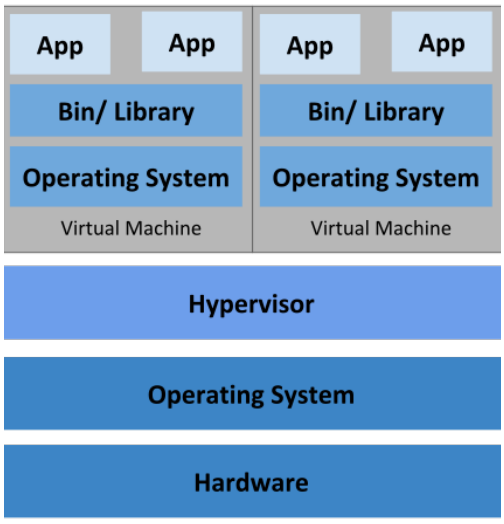

Figure 3: Visualization of virtualized deployments [8]

As computer performance increased and resource allocation became a significant problem due to an increasing number of applications needed for an organization, a solution was developed where you could run another operating system inside the original physical computer. This secondary operating system is known as a *virtual machine* or a *guest operating system*, while the physical computer is known as a *node* or *host*.

This works by having a software known as a *hypervisor* which translates each virtual machine's functions from the software down to an equivalent function on the host system. Thanks to this

translation layer, one host can run several virtual machines, each of them running their own operating system and applications [13].

By having this separation of applications and virtual machines, resources can be allocated to each virtual machine independent of other virtual machines, based on the application's need. One virtual machine running a simple application could be fairly lightweight and only require a small amount of memory or CPU power, while a virtual machine running a database can make use of significantly more memory and CPU to improve performance of other applications.

This deployment pattern also allows more efficient usage of hardware, since applications in different virtual machines are unable to interfere with one another. However, there is a slight decrease in performance when compared to running the application directly on the physical servers, due to each virtual machine on a given hypervisor sharing the bandwidth, memory and processing power of the underlying host.

Another benefit of virtual machines is that they are not bound to a specific host and can be moved between hosts without the virtual machine itself being affected.

2.3 Virtual containers

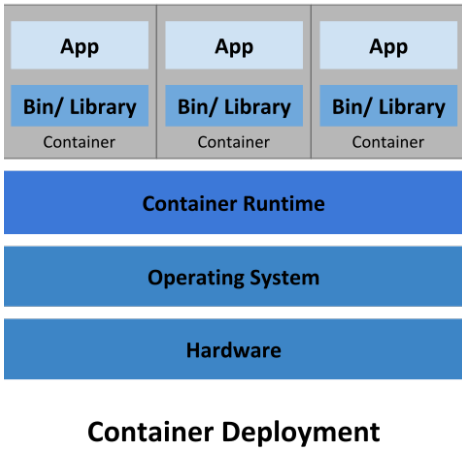

Figure 4: Visualization of application deployment using containers [8]

Virtual containers are a technology which allows the separation of applications from one another, by putting them in separate *containers*. These containers can then contain only the bare minimum dependencies which comes with its own benefits such as different versions of code libraries not interfering with one another. Another key difference compared to virtual machines is that containers reuse the kernel and other common functions of the host server, further focusing on what just the singular application needs.

Thanks to only having the minimum number of dependencies, containers are typically small in size, with a container of the minimalist operating system Alpine only being 8MB in size [1].This approach also creates a consistent environment for an application to run, reducing issues where an application works on one computer but not another.

The typical way a modern container works is by running a *container engine* such as *Docker* or *containerd* on a computer or a server. These engines allow the creation, running and management of *container images*, which have been defined beforehand in a file known as a Dockerfile. The Dockerfile consists of a base image, followed by additional steps which are executed in order, from top to bottom. Those steps may include moving files from the building

computer into the image (such as source code files) or defining commands to run (such as the command to build or run the source code).

There are also drawbacks with containers. For one, they add an additional level of complexity when trying to solve a problem with the application compared to running it on a physical server or virtual machine, where the developer first has to determine if an error is in the application itself or if the container itself is misconfigured.

The application itself should also be coded in such a way that it is not dependent on a previous state to get the most advantage from the containerization pattern, stemming from the best practice surrounding containers to discard the old instance when it finishes running its application, be it due to application crash or a requested shutdown. Since the instance gets discarded, any data not explicitly written to another location (such as a separate database or a networked storage) is lost. This distinction needs to be accounted for in the application itself.

To help visualize how a container works, imagine a match in a two-player game. Two players A and B wish to play, so they select their characters and a map, and the players' game clients connect to a server with the players' chosen configuration, and the game is played. What happens on the backend is that a container is created from a common container image and configured with the players' configuration after which the match is available for the clients to connect to. This exact same backend process is repeated for every following pair of players who wish to play, with the same container image being used with different configuration.

2.4 What is Kubernetes?

Kubernetes is a software for managing clusters running virtual containers. This includes functions such as ensuring a certain number of pods (explained below) of a given deployment are running, monitoring pod health, load balancing and much more.

Kubernetes itself was originally created by Google based on their experience of running very large datacenters and clusters of various applications and services. Along with its version 1.0 release in 2015, it was handed over to the Cloud Native Computing Foundation (CNCF), a subsidiary of the Linux Foundation, to act as a seed technology in order to further the development of cloud native computing.

Core to Kubernetes is the concept of a *pod*. A pod is a group of one or more containers which share storage and network resources, along with a specification of how to run those containers. Pods are the smallest deployable unit in Kubernetes, and other types of workloads are extensions of pods. A pod typically only contains a single container running an application, with additional containers running utility software such as monitoring or sending logs to a logging platform.

Configuration of pods and their extensions is done in YAML and defines exactly how a workload should run, such as the name and type of a workload, the ports used, the number of simultaneously running instances and much more. Kubernetes also exposes what is known as the Downwards API to the YAML, which allows the containers in the pods to access pod information such as their configured CPU or memory limits, or the IP of the node the pod is running on [6].

A Kubernetes-cluster consists of a control plane (which manages nodes and pods in the cluster) and one or more worker nodes on which the pods are run. The control plane consists of several components:

- An API server, which acts as a frontend for the control plane itself
- One or more instances of *etcd*, a software for key-value storage in which Kubernetes stores cluster data such as current state of the cluster as well as the desired state
- A scheduler, which checks requests for new pods and determines on what node said pod should run
- A controller-manager, running controller processes, examples of which are:
    - A node controller, responsible for noticing and responding when nodes go down
    - A job controller, responsible for creating and running jobs which are meant to do some tasks, only once
    - An endpoint controller which adds services and pods to endpoint objects, which is a list of IP addresses and ports of services/pods, either internal in the Kubernetes cluster or external.
- An optional cloud controller manager that enables connection to a cloud provider

Each worker node also runs some components of Kubernetes:

- A *kubelet*, an agent which ensures that containers are running in a pod and in accordance with the specification
- A *kube-proxy*, which allows network communication to and from pods, from inside or outside the cluster
- A container runtime to actually run the containers.

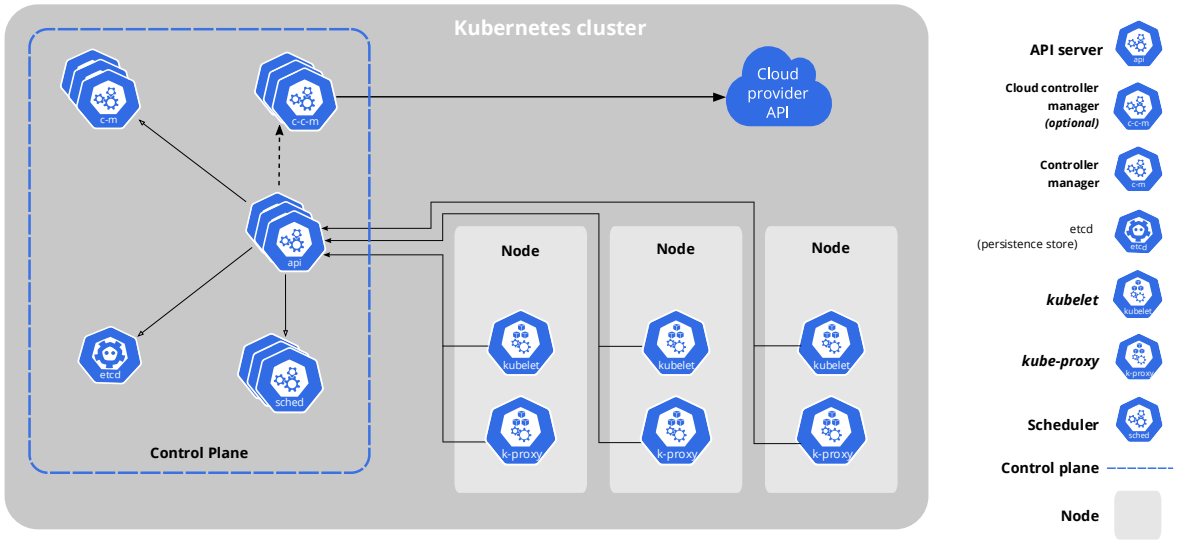

Figure 5: Overview of the components in a Kubernetes cluster [7]

2.5 Real-life use cases of Kubernetes

Kubernetes is a very flexible software thanks to abstracting away the containers themselves and can fit the needs of many differing organizations with some clever planning and work. One of the companies using Kubernetes is CERN, the European Organization for Nuclear Research, who needed their data processing to scale in particular during the periods prior to big conferences, and Kubernetes was able to fill this requirement by letting their data analysis workload scale on both their on-premises datacenter as well as public clouds [3].

Spotify went from running their containers using a homegrown orchestration software to using Kubernetes, in large due to the surrounding community [4].

For games, Niantic's *Pokémon GO* is built on services managed by Kubernetes, running in Google Cloud, which allowed them to quickly scale to meet 50 times higher load than anticipated during their launch [11]. Google have also in collaboration with Ubisoft created a framework for dedicated game servers running in Kubernetes, named *Agones* [2]. Blizzard Entertainment uses containerization at least for their upcoming title *Diablo IV* [12].

CNCF hosts a collection of case studies of companies who share how Kubernetes helped them with their problems on the official Kubernetes homepage: https://kubernetes.io/case-studies/

3 Method

The evaluation of containerization and Kubernetes for game development is in this work based on the author's own experience of setting up a Kubernetes cluster, followed by creating the game server and manually deploying it to the cluster, before finally automating the game server deployments through a matchmaker. The advantages and disadvantages as well as issues encountered are discussed in the next chapter.

3.1 The proof-of-concept

In order to try to evaluate the whole Kubernetes workflow from scratch, an implementation of the systems involved was done to roughly demonstrate how development using containers and Kubernetes works. Given the flexibility of containerization, the applications themselves are not meant to be taken as production ready, and instead to serve as inspiration.

The proof-of-concept has the following goals:

- The game server deployed in a K8s cluster, and connection made directly to the game server
- A matchmaker able to create new game servers at will
- The client only need a single API call to the matchmaker to get a server to connect to

In order to meet these goals, three main steps had to be taken: setting up the cluster, creating a game server along with the files necessary for deploying it to a K8s cluster, and creating a matchmaker which would act as the deployer for the game server.

Some parts of the actual work were stubbed out due to the author's workplace already providing said functionality, in order to save time. The first thing that was stubbed was the underlying, as well as the baseline setup for the virtual machines for things like security. Neither the cluster for running virtual machines nor the baseline setup is strictly required to run Kubernetes but are nonetheless part of a good IT setup.

The second thing that was stubbed was the setup of a container image registry which would store container images and is necessary for Kubernetes to be able to download and run the image itself. There are a lot of variants of container image registries for both self-hosting and cloud, each with their own features, benefits and drawbacks, but the basic functionality of managing container images is the same.

To test the connectivity and act as a client, the author made use of the program netcat, a network utility program for creating network connections, but any program capable of creating a TCP network connection to an arbitrary address should work.

3.1.1 The cluster

The Kubernetes cluster was set up on three virtual machines (nodes) running Ubuntu 20.04, in a VMWare cluster. Each node had 2 virtual CPU's as well as 4GB RAM. The initial setup of Kubernetes on these nodes was done with the help of Rancher Kubernetes Engine (RKE). Rancher Labs, creators of RKE, describe RKE as follows:

> Rancher Kubernetes Engine (RKE) is a CNCF-certified Kubernetes distribution that runs entirely within Docker containers. It works on bare-metal and virtualized servers. RKE solves the problem of installation complexity, a common issue in the Kubernetes

community. With RKE, the installation and operation of Kubernetes is both simplified and easily automated, and it's entirely independent of the operating system and platform you're running. As long as a supported version of Docker can be ran, Kubernetes can be deployed and ran with RKE [10].

Running RKE and answering the questions it asks (such as number of nodes, node IP addresses, each node's role in the cluster) lets RKE do the setup of the cluster. It also comes with a web interface for management of the Kubernetes-cluster, although this feature was not utilized by the author.

3.1.2 The game server

After the setup of the Kubernetes-cluster, development started on the game server, a simple version of the fictional game *Numberwang!* originally from the British comedy TV series *That Mitchell and Webb Look*. The basic idea is that two contestants call out seemingly random numbers which occasionally are said to be "Numberwang" by the host. As such, it seemed simple enough to implement in code, with the player clients being the contestants and the server being the host. The original plan was to support multiple connections and by extension multiple players but ended up pivoting to a single player due to time and knowledge constraints.

The game server was developed using C# and .NET Core 5.0, primarily due to the authors familiarity with them. It followed the tutorial on Microsoft's documentation on an asynchronous socket server [9], due to how straightforward it was. The server was then modified to fit the needs of the experiment such as adding on the actual game state as well as some rudimentary connection tracking.

The game server also runs the actual game state. At the start, it waits for an incoming connection. When a connection has been made, it starts the Numberwang game, where it sends the message "Let's play Numberwang!" to the client before waiting for the client to send a message. When the client sends any message, the server has a one in four chance of sending back "That's Numberwang!". The game continues until the client sends "end" which ends the game state, closes any connections, and shuts down the server.

Upon creation, the game server registers itself with the matchmaker by sending an HTTP POST-request to the /register endpoint (explained in the next section), with the data consisting of the IP of the node currently running the instance and the port that the instance is currently listening on. The client can then use this data to create a TCP connection and start the game.

3.1.3 The matchmaker

The matchmaker is a simple HTTP server built using Python 3.9 and the web server framework Flask. At the core of the matchmaker is a shared queue, holding game servers which have not yet been used. The server needs a minimum of two threads to run: one to handle the incoming request from the client, and one to handle registrations from newly created instances of game servers. The queue is Python's own synchronized queue, a first-in-first-out data structure, where the first data that comes in is also the first data that gets returned to the user.

The matchmaker responds on three endpoints:

- GET /
    - Returns a message conveying that the server is OK; serves as a health/uptime check
- GET /game
    - Requests a game server instance
- POST /register
    - Used by the game servers to register itself

The /game endpoint is called with a standard HTTP GET request and is meant for requesting a game server instance. When called, the matchmaker checks if there are any game servers already in the queue. If there are none, it connects to the Kubernetes cluster using the official Kubernetes Python client and requests for a new game server instance to be created.

The /register endpoint takes a HTTP POST, and checks if the data from the POST contains data for an IP address as well as a port. If either of those are missing, it returns a HTTP status code of 400 Malformed Request. If the data is available, it adds it to the queue of game servers to be sent to clients.

The matchmaker is designed to be easily tweakable. While it currently only requests one new instance at a time, it can be changed to request for example 3 new instances, which then register themselves the same and get handed out one at a time to the next requesting client. This should probably be stored in a proper database for actual production use in order to not lose track of game server instances, but for this small-scale experiment, an in-memory queue was considered to be sufficient.

3.2 Limitations

There are some limitations in the implementation of the proof-of-concept, such as the small scale of the experiment as a whole, where only one proof-of-concept is made. Part of this is a fairly naïve implementation which does not consider performance or security. This should not affect the results in any meaningful way but should be kept in mind if the proof-of-concept were to be expanded to a setup for use in a customer-facing environment.

The ‘game’ is single player only but is more an issue from the application coding rather than a limitation in Kubernetes or containerization. There are a number of containerized applications which can handle multiple simultaneous connections which have been deployed in Kubernetes.

A fair amount of the design choices made were influenced by how things are done at the author’s workplace, such as using RKE instead of another Kubernetes engine. These should once again not affect the result other than some software having solved problems in slightly different ways such as having a graphical interface versus using the command line.

4 Experimental results

4.1 Result of the proof-of-concept

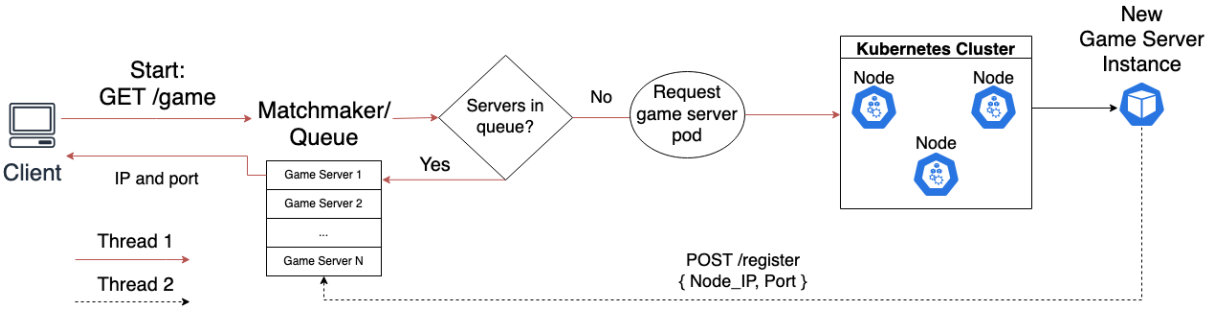

Figure 6: Full data flow in the proof-of-concept

Setting up a development environment for containers showed to be straightforward, all that was needed was a container engine of choice. Docker Engine is free and supports macOS and Windows, along with the CentOS, Debian, Fedora, Raspbian and Ubuntu Linux distributions with varying CPU architecture. Dockerfiles for creating containers are standard text files, and as such only require a raw text editor in order to be created.

Outside of the containerization, the standard development environment for a given programming language (such as compilers or interpreters, depending on the language) is suggested in order to actually develop the application itself outside of the container environment. Some languages such as C# offer pre-built container images for building and running an application, respectively, which help when making containerized applications.

To install Kubernetes, some familiarity with Linux-esque operating systems is recommended. Using RKE simplifies the process, but some knowledge of general server management and the terminology therein is still helpful in order to understand what information RKE is requesting. Kubernetes itself can run on any platform that can run containers.

While the proof-of-concept was created with Python and C#, they are not required for containerization. Kubernetes abstracts away the application inside the container from its responsibilities – it straight up does not care about the application, only the pod containing the container. Therefore, the application becomes the programmer's responsibility to ensure that it is working as intended.

Having both the game server and the matchmaker be containerized sped up the development in this work quite a bit. After setting up the Kubernetes-cluster and uploading the container image for the game server to the registry, it became easy to create new instances of the game server.

4.2 Code

All code for this experiment was written by the author, with the code for the asynchronous socket server being heavily inspired by Microsoft's asynchronous server socket example [9], in particular the functions for setting up a socket, as well as sending and receiving data over it. The rest of the code such as the game state and the matchmaker can be seen as approximations of how such a system might work. This section highlights some particular solutions done in the code, with the full code available in appendixes.

4.2.1 Flask Routes

```
@app.route("/", methods=["GET"])
def status():
[…]
@app.route("/register", methods=["POST"])
def register():
[…]
@app.route("/game", methods=["GET"])
def getGame():
[…]
```

Figure 7: Examples of Flask routes

Flask was chosen as the web server framework for this experiment due to its ease of use when developing. A route is defined as a function decorator, taking an endpoint path along with which HTTP verb it should respond to as arguments, as seen in figure 7. The endpoint can also contain parameters in the form <param_name>, which gets passed to the function argument of the same name, but this function was not used for this experiment.

4.2.2 Matchmaker Queue

```
# At program start
q = queue.Queue()

# When a game server is registers itself
q.put((registration_data["host"], registration_data["port"]), block=False)

# When a client requests a game server
try:
    server = q.get(block=True, timeout=30)
    return jsonify(server), 200
except queue.Empty:
    return "No pods available, try again", 503
```

Figure 8: Example Queue usage

Python's built-in queue is a synchronized queue which handles multiple data producers as well as multiple data consumers. Therefore, it worked well for this experiment since one thread could handle the client and act as a data consumer, while another thread would handle game server registrations acting as a data producer.

The put function inserts data into the queue, in this case a tuple consisting of the host IP and port for the registering game server. The block-parameter specifies what to do if the queue is full. If block is true, it waits for a duration specified by a timeout-parameter before throwing a queue.Full-exception. If block is false, it puts the data into the queue if a slot is currently available, else it also throws a queue.Full-exception. The get function works similarly, where if block is true, it waits for the duration of the timeout-parameter for data to become available.

4.2.3 Requesting Kubernetes pods

```python
from kubernetes import client, config
config.load_kube_config(config_file="kube_config_cluster.yml")

def requestNewInstance():
    print("Requesting new pod", flush=True)
    with open(path.join(path.dirname(__file__), "gameserver_deploy.yaml")) as f:
        dep = yaml.safe_load(f)
        k8s_core_v1 = client.api.core_v1_api.CoreV1Api()

        resp = k8s_core_v1.create_namespaced_pod(body=dep, namespace="default")
        print("Pod created, name='%s'" % resp.metadata.name, flush=True)
```

Figure 9: Requesting a pod with the official Kubernetes Python library

This experiment made use of the officially maintained Kubernetes-library for Python in order to create new pods when needed. This is done by first loading a *kubeconfig*-file for a given cluster. This kubeconfig contains information such as IP addresses of the control plane, along with private access keys (and will therefore not be shown in this thesis). The kubeconfig is automatically generated after creating the cluster.

```yaml
apiVersion: v1
kind: Pod
metadata:
  generateName: "game-"
  labels:
    foo: bar
spec:
  hostNetwork: true
  restartPolicy: Never
  containers:
  - name: "numberwang-numberwang"
    image: "<REPLACE_ME>"
    env:
      - name: SESSION_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
      - name: HOST_IP
        valueFrom:
          fieldRef:
            fieldPath: status.hostIP
      - name: MMS_IP
        value: "<REPLACE_ME>"
  topologySpreadConstraints:
    - maxSkew: 1
      topologyKey: zone
      whenUnsatisfiable: DoNotSchedule
      labelSelector:
        matchLabels:
          foo: bar
```

Figure 10: gameserver_deploy.yaml, containing the specifications for the gameserver deployment

After loading the kubeconfig, the file detailing the game server deploy is read and parsed, before being sent to the Kubernetes-cluster for the pod to be created. This deploy-file contains information on how the pod should work, such as which container image it should use (and where to get it), what name the pods should get and such. Environment variables are also created from the pod name, the IP of the node running it and the IP address of the matchmaker. The

topologySpreadConstraints-section tells Kubernetes to spread out the pods between nodes to even out the load of the nodes. If Kubernetes is unable to spread it due to no available nodes or other reasons, it simply does not create a new pod (DoNotSchedule).

4.2.4 Registering the game server

```
private static void Register(GameServer serv) {
    String host = Environment.GetEnvironmentVariable("HOST_IP");
    String mms = Environment.GetEnvironmentVariable("MMS_IP");
    if (host == String.Empty) {
        host = "127.0.0.1";
    }

    Dictionary<String, String> data = new Dictionary<string, string>() {
        {"host", host},
        {"port", serv.Port.ToString()}
    };

    HttpClient client = new HttpClient();
    StringContent content = new StringContent(JsonSerializer.Serialize(data), Encoding.UTF8,
"application/json");
    var result = client.PostAsync($"http://{mms}:8000/register", content).Result;
    Console.WriteLine(result.StatusCode);
}
```

Figure 11: Code for registering the game server with the matchmaker

After the game server has started its socket server, it registers itself with the matchmaker to signal that it is up and running and ready to be connected to. This registration is done with a simple HTTP POST request, using .NET's own HttpClient. The data sent is the host IP and the port that the socket is listening on, with the host IP being retrieved from the environment variable HOST_IP.

4.2.5 Game server events

```
public event Action<Player> ConnectionAccepted;
public event Action<String> MessageReceived;

server.ConnectionAccepted += OnConnectionAccepted;
server.MessageReceived += OnMessageReceived;

public void OnConnectionAccepted(Player newPlayer) {
    players.Add(newPlayer);
}

public void OnMessageReceived(String message) {
    if (message.Contains("end")) {
        StopGame();
    }

    if (RNG.GetIntRange(1, 4) == 4) {
        Server.SendToAllClients("That's Numberwang!");
    }
}
```

Figure 12: C# events being used to notify the game state

The game server makes use of C#'s events to notify the game state when something happens. For this experiment, only two events were used: one for when a client connection is completed, and one for when the server receives a message from the client. OnConnectionAccepted adds the connecting player to a list of connected players, which in turn starts the game of Numberwang after enough players are connected (for this experiment, just one player). OnMessageRecieved became the main logic of the game state. If the received message contains "end", the game is stopped, and the server is shut down. For any other message, there is a one in four chance that the server sends back a reply stating "That's Numberwang!", roughly mimicking the behavior of the original gameshow.

4.2.6 Random Number Generator

```
using System;
using System.Security.Cryptography;

public static class RNG {
    private static readonly RandomNumberGenerator rng = RandomNumberGenerator.Create();

    public static int GetRandomInt() {
        byte[] rno = new byte[4];
        rng.GetBytes(rno);

        return BitConverter.ToInt32(rno);
    }

    public static int GetIntRange(int lower, int upper) {
        return RandomNumberGenerator.GetInt32(lower, upper + 1);
    }
}
```

Figure 13: Cryptographically strong random number generator

Mostly for fun, the RNG class in figure 13 generates cryptographically strong random values. In the experiment, this class gets used in two places: once when generating which port to listen to, and once when determining if a message is Numberwang. Using a cryptographically strong random number generator has no discernable benefit in either of those uses over a typical seeded random number generator.

5 Discussion and conclusion

So why is Kubernetes so infrequently used? One reason could be how new Kubernetes is as a software, having only been around since 2014, and only gained a lot of steam in the other tech industries since late 2017 or so.

There is also the point that large portions of game development are done on Windows rather than a Unix-based operating system that lends itself better to the typically command line-based management of containers. Even the components of the proof-of-concept are running in containers which are based on Ubuntu. That is not to say that one cannot run containers on Windows, but more that there are more issues to go through compared to a Unix-based operating system.

There is also the learning curve associated with containerization and Kubernetes. Learning to write Dockerfiles in order to create the containers themselves, learning how to debug an application running in a container, learning how to create a Kubernetes-deployment to run the application in Kubernetes and so on.

For Kubernetes and containers to be of great use for non-technical positions in an organization, a fair amount of tooling will probably be needed as well, either third-party or written in-house. The proof-of-concept created by the author was managed entirely through command line, which in itself is prone to errors and mistyping when compared to a singular button running the correct command. The time cost for the creation of these tools adds up and come at the cost of not working directly on the game or project at hand.

Another point to be considered is the company size. A larger organization with data centers all over the world most likely already has systems in place for deploying and running applications, making migration to a new platform such as Kubernetes a non-trivial task, especially if the organization is just getting started with containerization. Smaller organizations, however, may not have the manpower or budget to run their own Kubernetes cluster and would therefore rather prefer to pay for a service to manage their global connectivity.

On the other hand, for smaller organizations they may not have the manpower or budget to run their own Kubernetes-cluster and would rather pay for a service to manage global connectivity for them.

While there seem to be no technical reasons preventing Kubernetes from being used in game development as evidenced by the proof-of-concept produced for this paper, its seemingly low adaption rate can thus probably be attributed to habit in combination with the game industry not marketing their use of it as extensively as companies in other industries.

References

[1] Alpine Linux Development Team. about | Alpine Linux. Retrieved May 14, 2021, from https://www.alpinelinux.org/about/

[2] Google. Agones Start Page. Retrieved May 29, 2021, from https://agones.dev/site/

[3] Kubernetes Authors. Case Study: CERN. Retrieved May 12, 2021, from https://kubernetes.io/case-studies/cern/

[4] Kubernetes Authors. Case Study: Spotify. Retrieved Jun 6, 2021, from https://kubernetes.io/case-studies/spotify/

[5] Kubernetes Authors. Kubernetes User Case Studies. Retrieved May 15, 2021, from https://kubernetes.io/case-studies/

[6] Kubernetes Documentation. Expose Pod Information to Containers Through Files. Retrieved May 14, 2021, from https://kubernetes.io/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information/

[7] Kubernetes Documentation, Kubernetes Components. Retrieved May 10, 2021, from https://kubernetes.io/docs/concepts/overview/c omponents/

[8] Kubernetes Documentation. What is Kubernetes?. Retrieved May 16, 2021, from https://kubernetes.io/docs/concepts/overview/w hat-is-kubernetes/

[9] Microsoft. (2017, Mar 30). Asynchronous Server Socket Example. Retrieved Apr 8, 2021, from https://docs.microsoft.com/en-us/dotnet/framework/network-programming/asynchronous-server-socket-example

[10] Rancher. Overview of RKE. Retrieved Apr 23, 2021, from https://rancher.com/docs/rke/latest/en/

[11] Stone, L. (2016, Sep 29). Bringing Pokémon GO to life on Google Cloud [Blog post]. Retrieved May 4, 2021, from https://cloud.google.com/blog/products/containers-kubernetes/bringing-pokemon-go-to-life-on-google-cloud

[12] Sweet, E. (2021, Jan 26). Blizzard Diablo IV debugs Linux core dumps from Visual Studio [Blog post]. C++ Team Blog. Retrieved May 26, 2021, from https://devblogs.microsoft.com/cppblog/blizzar d-diablo-iv-debugs-linux-core-dumps-from-visual-studio/

[13] VMWare Documentation. What is a Hypervisor?. Retrieved Apr 21, 2021, from https://www.vmware.com/topics/glossary/content/hypervisor

Games

Valve. (2012). *Counter-Strike: Global Offensive*.

Blizzard Entertainment. (2004). *World of Warcraft*.

Niantic. (2016). *Pokémon GO*.

Appendix A

Matchmaker source code

```
import queue
import yaml
import time

from os import path
from flask import Flask, request, jsonify
from kubernetes import client, config

q = queue.Queue()

app = Flask(__name__)

config.load_kube_config(config_file="kube_config_cluster.yml")

def requestNewInstance():
    print("Requesting new pod", flush=True)
    with open(path.join(path.dirname(__file__), "gameserver_deploy.yaml")) as f:
        dep = yaml.safe_load(f)
        k8s_core_v1 = client.api.core_v1_api.CoreV1Api()

        resp = k8s_core_v1.create_namespaced_pod(body=dep, namespace="default")
        print("Pod created, name='%s'" % resp.metadata.name, flush=True)

@app.route("/", methods=["GET"])
def status():
    return "Everything is fine", 200

@app.route("/register", methods=["POST"])
def register():
    print("Register called")
    registration_data = request.get_json()

    if not ("port" in registration_data and "host" in registration_data):
        return "Malformed request", 400

    q.put((registration_data["host"], registration_data["port"]), block=False)

    return "Registered successfully", 200

@app.route("/game", methods=["GET"])
def getGame():
    if not q.empty():
        server = q.get()
        return jsonify(server), 200
    else:
        print("queue empty, requesting new instance", flush=True)
        requestNewInstance()
        try:
            server = q.get(block=True, timeout=30)
            return jsonify(server), 200
        except queue.Empty:
            return "No pods available, try again", 503
```

File A1: mms.py

```
ARG $REGISTRY

FROM <CHANGE_ME>/python/v3.7:3.7.5-runtime
WORKDIR /app

ADD requirements.txt .
ADD mms.py .
ADD kube_config_cluster.yml .
ADD gameserver_deploy.yaml .

RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 8000
CMD ["gunicorn", "-b", "0.0.0.0:8000", "--threads", "2", "mms:app"]
```

File A2: Dockerfile

```
apiVersion: v1
kind: Pod
metadata:
  generateName: "game-"
  labels:
    foo: bar
spec:
  hostNetwork: true
  restartPolicy: Never
  containers:
  - name: "numberwang-numberwang"
    image: "<REPLACE_ME>"
    env:
      - name: SESSION_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
      - name: HOST_IP
        valueFrom:
          fieldRef:
            fieldPath: status.hostIP
      - name: MMS_IP
        value: "<REPLACE_ME>"
  topologySpreadConstraints:
    - maxSkew: 1
      topologyKey: zone
      whenUnsatisfiable: DoNotSchedule
      labelSelector:
        matchLabels:
          foo: bar
```

File A3: gameserver_deploy.yaml

Appendix B

Game server source code

```
ARG REGISTRY

# Build
FROM <REPLACE_ME>/dotnet:5.0-sdk AS build
RUN ln -sf /usr/share/zoneinfo/Europe/Stockholm /etc/localtime
WORKDIR /app
COPY . ./
RUN dotnet publish -c Debug -o output

# Run
FROM <REPLACE_ME>/dotnet:5.0-runtime AS runtime
RUN ln -sf /usr/share/zoneinfo/Europe/Stockholm /etc/localtime
WORKDIR /app
COPY --from=build /app/output ./
RUN chmod +x /app/exarb.dll

ENTRYPOINT ["dotnet", "/app/exarb.dll"]
```

File B1: Dockerfile

```
using System;
using System.Security.Cryptography;

public static class RNG {
    private static readonly RandomNumberGenerator rng = RandomNumberGenerator.Create();

    public static int GetRandomInt() {
        byte[] rno = new byte[4];
        rng.GetBytes(rno);

        return BitConverter.ToInt32(rno);
    }

    public static int GetIntRange(int lower, int upper) {
        return RandomNumberGenerator.GetInt32(lower, upper + 1);
    }
}
```

File B2: RNG.cs, utility class for generating random numbers

```
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text.Json;
using System.Text;

namespace exarb
{
    class Program
    {
        public static void Main()
        {
            GameServer serv = new GameServer();
            GameState gameState = new GameState(serv);

            Console.CancelKeyPress += (sender, eventArgs) => {
                if (gameState.IsRunning) {
                    gameState.StopGame();
                }

                if (serv.IsRunning) {
                    serv.Stop();
                }
            };

            serv.Start();

            Register(serv);

            gameState.StartGame();

            while (gameState.IsRunning) {
                gameState.Update();
            }

            serv.Stop();
        }

        private static void Register(GameServer serv) {
            String host = Environment.GetEnvironmentVariable("HOST_IP");
            String mms = Environment.GetEnvironmentVariable("MMS_IP");
            if (host == String.Empty) {
                host = "127.0.0.1";
            }

            Dictionary<String, String> data = new Dictionary<string, string>() {
                {"host", host},
                {"port", serv.Port.ToString()}
            };

            HttpClient client = new HttpClient();
            StringContent content = new StringContent(JsonSerializer.Serialize(data),
Encoding.UTF8, "application/json");
            var result = client.PostAsync($"http://{mms}:8000/register", content).Result;
            Console.WriteLine(result.StatusCode);
        }
    }
}
```

File B3: Program.cs, the program entrypoint

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Collections.Generic;

public class StateObject {
    public const int BufferSize = 1024;
    public byte[] buffer = new byte[BufferSize];
    public StringBuilder sb = new StringBuilder();
    public Socket workSocket = null;
    public DateTime lastDataReceived = DateTime.UtcNow;
}

public class GameServer {
    public bool IsRunning { get; private set; }
    public int Port { get; private set; }
    private Dictionary<String, StateObject> currentConnections = new Dictionary<String,
StateObject>();

    public event Action<Player> ConnectionAccepted;
    public event Action<String> MessageReceived;

    public GameServer() {
        Port = RNG.GetIntRange(11000, 11010);
    }

    public void Start() {
        IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
        IPAddress ipAddr = ipHostInfo.AddressList[0];
        IPEndPoint localEndPoint = new IPEndPoint(ipAddr, Port);

        Socket listener = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
        try {
            listener.Bind(localEndPoint);
            listener.Listen(100);

            Console.WriteLine("Bound socket to IP {0} and port {1}", localEndPoint.Address,
localEndPoint.Port);

            listener.BeginAccept(new AsyncCallback(AcceptCallback), listener);
            IsRunning = true;
        } catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }

    public void Stop() {
        foreach (KeyValuePair<String, StateObject> connection in currentConnections) {
            Socket sock = connection.Value.workSocket;

            SendToAllClients("That's it for Numberwang!");

            sock.Shutdown(SocketShutdown.Both);
            sock.Close();
        }

        IsRunning = false;
    }
```

```
    private void ReadCallback(IAsyncResult ar) {
        Console.WriteLine("Read callback...");
        String content = String.Empty;

        try {
            StateObject state = (StateObject) ar.AsyncState;
            Socket handler = state.workSocket;

            if (handler.Connected) {
                int bytesRead = handler.EndReceive(ar);

                if (bytesRead > 0) {
                    state.sb.Append(Encoding.UTF8.GetString(state.buffer, 0, bytesRead));

                    content = state.sb.ToString();
                    state.sb.Clear();
                    Console.WriteLine("Read {0} bytes from socket. \n Data : {1}", content.Length,
content);


                    MessageReceived(content);
                    state.lastDataReceived = DateTime.UtcNow;

                    handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new
AsyncCallback(ReadCallback), state);
                } else {
                    Console.WriteLine("0 bytes read, receiving again...");
                    handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new
AsyncCallback(ReadCallback), state);
                }
            }
        }
        //This is apparently intended design.
        // https://stackoverflow.com/questions/4662553/how-to-abort-sockets-beginreceive
        catch (ObjectDisposedException) { };
    }

    private void AcceptCallback(IAsyncResult ar) {
        Console.WriteLine("Accepting connection...");

        Socket listener = (Socket) ar.AsyncState;
        Socket handler = listener.EndAccept(ar);

        StateObject state = new StateObject();
        state.workSocket = handler;

        Player player = new Player();
        player.UUID = Guid.NewGuid();

        currentConnections.Add(player.UUID.ToString(), state);
        handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new
AsyncCallback(ReadCallback), state);

        ConnectionAccepted(player);
    }
```

```
    public int GetNumCurrentConnections() {
        return currentConnections.Count;
    }

    public void SendToAllClients(String data) {
        foreach (KeyValuePair<String, StateObject> connection in currentConnections) {
            Socket sock = connection.Value.workSocket;

            Send(sock, data);
            Send(sock, "\n");
        }
    }

    public void SendToSpecificClient(String uuid, String data) {
        Socket sock = currentConnections[uuid]?.workSocket;

        if (sock != null) {
            Send(sock, data);
        }
    }

    private void Send(Socket handler, String data) {
        byte[] byteData = Encoding.UTF8.GetBytes(data);

        handler.BeginSend(byteData, 0, byteData.Length, 0, new AsyncCallback(SendCallback),
handler);
    }

    private void SendCallback(IAsyncResult ar) {
        try {
            Socket handler = (Socket) ar.AsyncState;

            int bytesSent = handler.EndSend(ar);
            Console.WriteLine("Sent {0} bytes to client.", bytesSent);
        }
        catch (Exception e) {
            Console.WriteLine(e.ToString());
        }
    }
}
```

File B4: Server.cs, the socket server responsible for network communication

```
using System;

public class Player {
    public string userName;
    public Guid UUID;
}
```

File B5: Player.cs

```
using System;
using System.Threading.Tasks;
using System.Collections.Generic;

public class GameState {
    public GameServer Server { get; private set; }
    public bool IsRunning { get; private set; }

    private List<Player> players = new List<Player>();

    public GameState(GameServer server) {
        Server = server;
        server.ConnectionAccepted += OnConnectionAccepted;
        server.MessageReceived += OnMessageReceived;
    }

    public void OnConnectionAccepted(Player newPlayer) {
        players.Add(newPlayer);
    }

    public async Task WaitForPlayers() {
        while (players.Count < 1) {
            Console.WriteLine("Waiting for players...");
            await Task.Delay(2 * 1000);
        }
    }

    public void OnMessageReceived(String message) {
        if (message.Contains("end")) {
            StopGame();
        }

        if (RNG.GetIntRange(1, 4) == 4) {
            Server.SendToAllClients("That's Numberwang!");
        }
    }

    public void StartGame() {
        WaitForPlayers().Wait();

        Server.SendToAllClients("Let's play Numberwang!");
        IsRunning = true;
    }

    public void StopGame() {
        IsRunning = false;
    }

    public void Update() {
        System.Threading.Thread.Sleep(250);
    }
}
```

File B6: Game.cs, containing the code for the game state