BlazeBox: The all-in-one suite for Minecraft cluster management

Abstract

BlazeBox is an application that helps users manage Minecraft clusters effortlessly with the easy-to-use API that can be called by any operators(service). It features a Continuous Deployment which can help developers deploy their plugins(typically Bukkit or Paper ones) through the easy-to-use REST API or gRPC endpoint. Also it monitors the servers(and plugins if connected to its API) and exports the metrics to Prometheus(*https://prometheus.io/)[1]*.

Introduction

For example, we managed to create the cluster of a few Minecraft servers with Deployment(k8s resource type)*. And then we uses ConfigMap to pass in the server.properties* file. And then we mounts the volumes to put them inside the ./plugins folder, (we use dockerfile to simplify this).

You can easily find this hard and it is truly at very low level of abstraction if you try to explain what it does by manipulating how it was built. But with BlazeBox, we can define a MinecraftCluster CRD, a MinecraftServer CRD, a MinecraftProxy CRD to finish all these up, if we need plugins, it would be just as simple as creating another field in the CRD of the server to specify the plugin’ s metadata and its location: either a URL or a Git/Maven ref.

For example, when we are creating such complex infrastructures to provide an abstraction of an entire managed cluster, it is difficult to do such things:

1. How hard it is to allocate resources for these servers.
2. Minecraft Server cores are not multi-threaded, they can be under-optimized if we don’ t patch them or remake news ones that are multi-threaded.
3. For most users, getting their hands on such compiled server cores are nearly impossible.
4. For multiple reasons, creating such a core isn’ t possible
5. The server cores are hardly coded, multiple behaviors must be patched before connecting to the actual BlazeBox service, this cannot be done by Bukkit plugins nor Paper ones.

This paper is about how we managed to overcome the difficulties and how to implement such a system with a patched server core. We are willing to create a complex cluster in any time soon so that a demo could be hosted by any of our sponsors to demonstrate the features of BlazeBox. BlazeBox uses BlazeSpan [2] to store its data and BlazeDome [3] to manage domains.

And it also features customization with ease, including the Data Access Interface(DAI) and supports OpenTelemetry(https://opentelemetry.io/) [4] to export its metrics. DAI is designed to replace the BlazeSpan infrastructure managed by your cluster management system, like kubernetes in our case.

It is hard to say that any of these are the hard-dependency that cannot be replaced, we considered loosely-coupled structures on the first day of the development.

BlazeBox is also an operator-based system that operates the cluster and provides a higher level of abstraction compared to the classic Kubernetes method.

Managing Servers

Since BlazeBox is operator-based, it uses the Kubernetes API to interact with it. It creates Pods dynamically and assigns them into different nodes. Since the nodes are managed by Kubernetes, it is way easier to manage them.

Replacement of RPC

When we are managing these servers, we use a protocol called *BlazeCom* that interacts with the server cores and tells it what to do. It has a basic data model represented in JSON or YAML.

```
{
      “blazebox” : “<timestamp>”,
      “domain” : “<BlazeDome domain selector>”,
      “action” : “<action name>”,
      “receiver” : “<receiver selector>”,
      “issuer” : “<issuer name>”,
      “data”: “<data>”
}
```

Note:

1. <timestamp> is the timestamp returned by the time service
2. <BlazeDome domain selector> is the selector that selects only one domain from BlazeDome
3. <action name> is the name of one of: { push , kill , exec , enable_plugin, disable_plugin, update, map_update, plugin_update}

4. <receiver selector> is the selector of servers who receives this string and parses it before executing, note that this data is broadcast to all servers that is willing to receive(this kind of message).
5. <data> is the additional data sent to the receiver.

This is a broadcast replacement of gRPC, this also allowed the metrics system to catch every message without the receiver broadcasting it again. It also allowed the server to present all valid data. None of the data posted should contain sensitive data, also fields like plugin names, map names are spoofed(typically hash with salt) in practice to secure itself.

Handler

Below is a code snippet for how a handler is built in Java:

```
import org.serenity.blazebox.handler.Handler;
import org.serenity.blazebox.handler.RegisterFacade;
import org.serenity.blazebox.records.BlazeComData;
public class MyHandler extends Handler {
	public boolean isSupported(BlazeComData data){}
	public void handle(BlazeComData data){}
}
```

This is an example of handler that handles BlazeCom data. The BlazeCom API is made for handling data sent from the operator, so further on this is what a handler looks like in our production code:

```
----- import is removed -----

public class HandleKill extends Handler{
	priority = Priority.HIGHEST;
	public boolean isSupported(BlazeComData data){
		return data.action == Actions.KILL;
	}
	public void handle(BlazeComData data){
		serverManager.stop(1); //exit code for killed is 1
	}
}
```

Note: serverManager is a Spring(https://spring.io) [5] bean.

So the code snippet handles a kill signal sent by the BlazeBox operator and kills itself, well somethings will be on the filter chain to authenticate the authenticity of the operator and all the other things you can customize or prefer the defaults.

It contains a record of BlazeComData which is the record class for BlazeCom messages, we uses that to communicate between servers. And the above handler implements the interface and is registered as a component, this makes it possible to be indexed by the Node and can be callable in a list which makes it way easier to debug and manage.

Integration with Kubernetes

BlazeBox depends on Kubernetes to orchestrate the clusters, in this section we are going to talk about the CRD format and how BlazeBox uses the defined resource to spin up a cluster of servers.

```
apiVersion: blazebox.serenity.dev
kind: MinecraftCluster
metadata:
      name: <cluster-name>.<BlazeDome-domain-name>
      namespace: <namespace>
spec: {}
```

Sure that if you don’ t want to setup BlazeDome, it is okay to just remove the BlazeDome field from the snippet above. Thanks to Kubernetes, it will still work just fine and is as stable.

```
apiVersion: blazebox.serenity.dev
kind: MinecraftServerDeployment
metadata:
      name: <name-of-choice>.<BlazeDome-domain-name>
      namespace: <namespace>
      labels:
            clusterRef: <cluster-name>.<BlazeDome-domain-name>
spec:
      replicas: 10 #how many replicas would you prefer
      template:
            spec:
                  core:
                        version: "1.20.4"
                        channel: "Paper"
                  config: {}
```

Again, if you don’ t prefer BlazeDome, feel free to remove it, BlazeDome is not a hard dependency for BlazeBox, but BlazeDome is better than Kubernetes when it comes to domain management and resource allocation. So as above noted, the snippet will create a MinecraftServerDeployment object inside the domain if you have BlazeDome.

So when we finished, setting up proxies is also important, which can be done with only:

1. Change *kind* to MinecraftProxyDeployment,
2. Change *spec.template.spec.core.version* and *channel*

So you finished everything you have to do to spin up a normal Minecraft Cluster. Then we need plugins to add custom contents to the server.

```
------- Previous Declaration fields are purged --------
spec:
      replicas: 10 #how many replicas would you prefer
      template:
            spec:
                  core:
                        version: "1.20.4"
                        channel: "Paper"
                  config: {
                        plugins:
                              - name: something
                                reference:
                                    mvn:
                                          repository: https://serenity.dev/mvn
                                          groupId: org.serenity.example
                                          artifectId: example_plugin
                                          version: "1.0.0"
                                          cred_secret: <secret for credentials>
                              - name: something-else
                                    url: https://serenity.dev/example.jar
                  }
```

So you added plugins to your server and now you are ready to boot up your servers. And then all of your setup work is done by default, but still we are missing the custom world.

Though the complexity of world folders, plugins can define their own file paths and there isn’ t a well-known standard for this, so instead of using a world-oriented model for the CRD, we chose the *file patch* instead. According to its name, it should make patch to the server’ s root directory. We define it as extracting an archive provided to the root directory. For a majority of reasons, we would like to choose *.zip*, *tar.gz* and *tar.xz*. They are the most used compression protocols in the world while providing incredible performance.

Practices for speed

Asynchronous World Loading

We made that the world can be loaded automatically with another thread and be passed into the server-core’ s thread, basically, BlazeBox daemon will be created in another container in the same pod with the server, and it will try to load the world automatically, after that, it will push the loaded world directly to the server core’ s memory (memory access bypassing java).

Basically, Java offered a utility called *Unsafe*, Which is a class that cannot be accessed outside the jar of the server core, since plugins are loaded with *ClassLoader*, this will be impossible to do so, this requires modifying the server core to be done.

Modifications will reveal vulnerabilities, but we made this with the method of Inter-Process-Communication(IPC), we provided the API for plugins to let plugins access the Unsafe methods, which made possible for the plugin to hot swap the world.

With this method of loading, instability issues are spotted. So when we saw this issue, we load chunks only, and then we sync the blocks to the current world the server has loaded with shadow paging. Then we flag the server offline(calling BlazeCom operators API). And then we restart the server, after that, we put it online again.

Dynamic World Partitioning

Making so, servers are no longer heavily stated, this means that different servers can dynamically pass worlds over in isolation and create replicated environments.

So we made worlds being loaded by multiple servers and if one fails, another accepts all the players on the server that just crashed(requires proxy support), and the BlazeBox daemon of the crashed server hands the world to its successor, also it analyzes the crash, it will walk along the thread dump and find the issue. We' ve made that it can detect the issue, if its caused because the main thread is blocked, then it can identify that, for example, a player ran a command that halts. So it reacts based on the users configuration, which could be firing an alert for example. And if it crashed because the Java Virtual Machine ran out of memory, based on configuration, the new server could receive more memory.

Also, for use cases like mini-game servers, it can also pass maps between instances and do basically the same thing as described above.

Content Delivery Network for static files

Content Delivery Network(CDN) makes static files easier to distribute. With this, we can easily store all the world files and plugins inside a S3 bucket with automatic CDN pushes. This makes the world load way faster than a simple FTP or HTTP file server. Also storing them on public CDN services are still an efficient way to boost loading speed. With all of these working, the world should load at blazing fast speed and a server could be online in a blink.

Asynchronous world generation

For those use-cases that needs new worlds to be generated frequently, this can be helpful. With this feature, the world can generate before it is needed and be passed to the server directly(or via a CDN) when its loaded. This eliminates the time needed for server to generate a world which may take very long for *Spigot* servers.

Git based backup

For these cases of use, we might also need to put the world files under version control, well git is good but it does not fit our use case, we just need the region to be backed up and rolled back, or we might need the entire thing, but either case, we still prefer git, because all those necessary files can be backed up automatically with CRON based Job system automatically committing the changes.

Though this part still lacks implementation, we still didn' t implement our own at the end of the research, but a good idea is: create a repository on a git server, and every time a client(BlazeBox Daemon) pushes a commit to any of the machines in the cluster, that machine will use the *Gossip* protocol to inform other machines about the commit, and one of the machines will push one commit to master. The cluster may be managed by either ZooKeeeper or BlazeDome.

Conclusion

This is the draft of the paper, version 1. Meaning that this paper requires update, some pages are written a few months ago, and our team are rewriting them. Soon the final draft will be released, and later will be published.

Words here are the conclusion, which could be made easily, we implemented such a system that can help greatly improve performance, and made a skeleton of how such system are by design made. Which later a paper will explain all the complex infrastructure of a Minecraft game server network.

We will keep working on this, and we will rewrite this paper soon to provide up-to-date work.