Distributed caching mechanism for various MPE software services

Summer student 2017 project

Summer student: Andrej Svec
Supervisor: Kamil Henryk Krol
Section: TE-MPE-MS
Date: 18. 8. 2017
Acknowledgment

I would like to express thanks to my supervisor Kamil Henryk Krol and my second supervisor Maciej Krzysztof Osinski for being the guiding light when I was struggling with my project.
Overview

In this section we will give a motivation for this summer student project. We will describe existing systems and their state before the start of the project. In the next sections we will focus on this project and describe our solution.

Assignment

The MPE Software Section provides multiple software services to facilitate the testing and the operation of the CERN Accelerator complex. Continuous growth in the number of users and the amount of processed data result in the requirement of high scalability. Our current priority is to move towards a distributed and properly load balanced set of services based on containers. The aim of this project is to implement the generic caching mechanism applicable to our services and chosen architecture. The project will at first require research about the different aspects of distributed caching (persistence, no gc-caching, cache consistency etc.) and the available technologies followed by the implementation of the chosen solution. In order to validate the correctness and performance of the implementation in the last phase of the project it will be required to implement a monitoring layer and integrate it with the current ELK stack.

Post Mortem (PM)

Post Mortem is a system that collects data about interesting and important events happening in the LHC, e.g. beam dump, powering, injection events etc. These events are called Post Mortem events and the data is called Post Mortem data (PM data). After the PM data is collected from the LHC, it is stored as a raw files in a machine on the CERN technical network from where it is accessible by Network File System (NFS). We will call this machine NFS server or PM data storage.

The data stored in PM system is interesting both for online analysis (automatic tools helping operators during daily operation) and hardware experts (long term analysis of devices’ performance).

Post Mortem does not collect data continuously. Its event based nature and irregular data flow make the data collection task very challenging. Heavy load on the current PM storage implementation can result in long access times and it may lead to missing some important data dumps.

Post Mortem REST API

To decouple users from the underlying data storage a middle layer between NFS server and end users was introduced in 2015. This layer is a java based REST API which reads the PM data from the NFS and serves it to the clients. One of the most important parts of the PM REST API is a caching mechanism. The PM REST API uses an implementation of Spring cache by Ehcache. The cache is held on the heap of the JVM.

The PM REST API layer is very important for future work as it decouples the users from the NFS server and abstracts the PM data storage. It enables to change the PM data storage easily
without stopping all user applications from working. This is necessary for the future because the PM data raw file storage will be probably changed for other types of storage, e.g. Hadoop.

Distributing PM REST API

By now not all of the users who access PM data from the NFS server are using PM REST API. Some of them still access the data directly by connecting to the NFS server. However in the future, this behavior should change because all users will be migrated to the PM REST API. With increasing number of users we know that one instance of PM REST API will not be able to serve all the requests. That is why we need to distribute the PM REST API in a cluster provided by CERN Openstack infrastructure.

Docker container

First of all we needed to wrap the existing PM REST API application in a Docker container so that we would be able to run it in a cluster. There were several issues solved when containerizing the application.

First of all the application relies on having the PM data storage mounted at /nfs. However the NFS is not available on machines running within Openstack as the NFS server is on CERN technical network and the NFS communication is not allowed. After discussion with IT support we decided to mount the PM data storage via SSHFS which is a much slower connection than NFS but it is safer. In the future the mounting method could be preferably changed to NFS but security aspects must be discussed.

Other issues were rather simple. Docker container always uses GMT and the PM data is in GMT+2 timezone so sometimes we were looking for data in wrong day. Exposing the JMX metrics from Docker container was very difficult and required proper setup of JMX hostname and port to allow the communication. This will be discussed in more detail in monitoring section.

Deployment in Kubernetes

CERN Openstack infrastructure enables to create a cluster with a variety of orchestration managers. These orchestration managers are comparable. We decided to use Kubernetes due to existing knowledge about this, and recommendation and support from the IT department.

As mentioned above Kubernetes is an orchestration manager for containerized applications. It automatically restarts any container if it dies. If a node dies it rebalances the cluster and starts new containers. As a result all containers need to be stateless because a container can get removed at any time and can be respawned on another node. However this does not represent any problem for PM REST API as it requires read-only access to the PM data storage.

We run the application in a cluster with twenty nodes. Exactly one application instance fits on one node. The application can be scaled easily using Kubernetes CLI or Web UI.
Distributed caching

The main part of this summer student project was implementing a distributed caching mechanism. In this section we will first describe the different options for caching and the final chosen solution. In the end we will discuss the performance of this solution.

Types of caches

We analysed different types of caches. The results of this analysis are written below. When choosing the solution we focused on following properties:

- Easy setup
  - Kubernetes. The cache should be easy to deploy in Kubernetes.
  - Spring integration. PM REST API is a Spring Boot application. We focused on solutions that are easy to integrate with Spring Boot, preferably those that can be integrated by only adding a Spring Boot Starter.
- Free to use. We wanted the solution to be either open-source or free to use.
- Cache size. We want to maximize the cache size as we want to save as much data as possible.

Standalone vs embedded

**Standalone cache.** Standalone cache runs in a process external to the application. This type of cache is more difficult to set up as it requires to run the application as well as the cache in the cluster. On the other hand this solution may be more stable because the PM REST API would be running even if the cache died. We considered following standalone solutions: Redis, Memcached, Infinispan.

**Embedded cache.** Embedded cache runs within the application. The disadvantages include the fact that it consumes CPU and RAM of the application but the setup is easier. Amongst embedded solutions we considered: Infinispan, JBossCache, Hazelcast or Java Caching System.

We chose embedded cache as an suitable solution for us as it was a high priority for us to deploy the application easily.

Replicated vs distributed

**Replicated cache.** Replicated cache is a type of cache where every node has all the data. This is useful when there is a need to cache relatively small amount of data which needs to be accessible very quickly. It also increases redundancy and data will not be lost if one of the nodes dies.

**Distributed cache.** Distributed cache is a cache where the cache entries are not stored on every node but only on a subset of nodes. This way we can achieve larger cache size because it gets spread around the different nodes.

**Distributed and replicated cache.** This type of cache lets its' users define how many owners should each cache entry have. If this number is set to the number one the cache becomes a
distributed cache, if the number of owners is set to the number of cache instances it becomes a fully replicated cache.

We chose to implement a distributed cache because it is not a problem if some of the data is lost from the cache. We prefer larger cache to redundancy because the data can be read from the PM data storage at any point.

On-heap vs off-heap

**On-heap.** Cache entries are stored on heap of JVM. Advantage is they do not have to be serialized but when the cache is full and there are cache evicts (removal of an entry) this data needs to be garbage collected which might take a lot of time if the cache is large.

**Off-heap.** Cache entries are stored in a serialized way off the heap. There is much less garbage collection when cache evictions occur and the cache is more stable when it is full. These were the main reasons for choosing off-heap cache.

Our solution

We analysed various distributed caching mechanisms and gathered advantages and disadvantages of all of them. The results of the analysis can be found in appendix A. We decided to implement Infinispan embedded distributed cache in PM REST API for following reasons:

- Integration with Spring. Infinispan integrates easily with Spring Boot because it provides a Spring Boot Starter. It even supports Spring's Cacheable annotation so no changes in the code were required.

- Integration with Kubernetes. Infinispan uses Kube Ping to discover other instances when it is run in Kubernetes cluster. This way it is able to connect to other instances itself.

- Clear documentation. [http://infinispan.org](http://infinispan.org)

Implementation

We integrated Infinispan cache into the PM REST API using Infinispan Spring Boot Starter. Spring Boot Starters are meant for easy set up. In our case it required only including the Starter and creating a configuration file for Infinispan. In this configuration file we set up distributed asynchronous caches which would discover other instances using Kube Ping and we defined sizes for them.

Fine tuning cache sizes is very important because it leads to increased stability and performance. We assigned smaller cache for storing indexes as these entries are small. For storing PM data we assigned more space because these files can go up to few megabytes. In the future the cache sizes will need to be fine tuned even more because only from real operation we will be able to see how individual caches are being used and whether they are too small or too big.
Performance testing

For determining the performance of our solution we created a test suite. This test suite consists of 3,700 requests which are run against PM REST API. Requested data is related to one beam dump event in the LHC. We run the test suite against the API twice. During the first run we populate the cache and we measure throughput without the cache because all the requests are unique. During the second run we measure how much the cache can accelerate the test suite. During the second run the requests are randomly shuffled.

During the implementation process we tried following setups of the cache and compared its results.

- **S2.** Distributed synchronous cache, two owners of a cache entry
- **A2.** Distributed asynchronous cache, two owners of a cache entry
- **A1.** Distributed asynchronous cache, one owner of a cache entry

\[\text{\textsuperscript{1}}\text{ replication factor}\]
From the charts above we can see that during the first run, when the cache is empty, the solution scales linearly until ten instances. Then we probably hit the limit of the PM data storage or networking limit and the solution no longer scales linearly. These limits can be pushed further by switching from SSHFS connection to NFS connection as the latter is around five times faster. The storage limit can be solved by changing the type of the storage to a different type of storage or to load less data from the storage.
If we compare our solution to the baseline solution which is the non-distributed version of PM REST API we can achieve almost the same results with one instance.

During the performance tests we also measured cache hit ratio. This value is 0.5 if all the requests are cached because there will be only cache misses during the first run and only cache hits during the second run. Results are shown in following charts. Read PM data cache is used to store actual PM data and Search PM data cache is used for storing indexes of PM data.
We can see from the charts above that the Search PM data cache performs very close to 0.5 independently from the number of instances. This means that every request is stored in cache.

As far as Read PM data cache is concerned, when using two owners (S2, and A2) we are able to cache every request when running ten instances. If the number of owners is set to one we are able to achieve the same results with half of the nodes, i.e. five.

**Monitoring**

Monitoring is useful for discovering possible issues in applications. The old application exposed metrics from itself via JMX. Logstash was used to collect these metrics at fixed time intervals and store them in Elasticsearch. Metrics are afterwards fetched from Elasticsearch and displayed in Grafana dashboards.

However a solution like this was not possible in Kubernetes because from outsides neither we can see the application instances running nor can we connect to any of them directly. For these reasons we knew that we would need some application running inside the Kubernetes cluster collecting the metrics. We analysed possible solutions and we have created a summary in CERN Wiki which is also available in Appendix B.

Based on this analysis we proposed a solution for collecting the JMX metrics from individual instances. First of all we attached JMX Exporter to the PM REST API JVM. It is a java agent which collects JMX metrics and exposes them as HTTP. Scraping all the metrics took around 7 seconds and that is why we limited the scraping only for ObjectNames beginning with java_lang and org.infinispan. This filtering resulted in scraping duration around 500 ms.

Metrics exposed by each java agent are then collected by Prometheus deployed inside the cluster. As Prometheus needs to store the metrics in a persistent storage and Kubernetes
application instance can be killed at any moment and respawned on a different node, we decided to create an Openstack volume, mount it to one of the nodes and run Prometheus exclusively on this node.

Final step was creating a dashboard. Up to this point we created only a proof of concept displaying data from all instances in one chart. This test was however important to show that the data from Prometheus can be fetched and displayed by Grafana.

Conclusion

During my summer student stay we analysed possible solutions of distributing PM REST API and different solutions for distributing the PM REST API cache. We successfully implemented and deployed chosen solution.

We also fine tuned the caching mechanism and set up monitoring of the PM REST API instances deployed in Kubernetes. The results of the caching are promising and will lead to the API serving users faster when the system is in production.

The PM REST API serves requests around 15 times faster thanks to Search Raw Data Cache as this caches the indexes of PM data. These indexes are often tens of megabytes large so we do not need to go and fetch all the data. It also means that the Search Raw Data Cache significantly reduces the load on NFS server.

The Read PM Data Cache accelerates the application another two times but compared to the cache for indexes, this is not very important. However we want to cache as much as we can because we want to minimize the number of requests on NFS server.
Further work

Further work comprises more fine tuning of the cache sizes. It is important to analyse the behavior of the system and based on that edit change the sizes of the caches so that we minimize the number of read operations on the NFS. This is a simple metric to minimize. It is only necessary to monitor this metric and information about cache hit ratio in individual caches. Afterwards an improvement on cache sizes can be proposed and evaluated again using the same metrics.

Summary

During my summer student stay I learned a lot about clustering and solved many challenging problems in the distributed world. In the end the results are above my expectations because I didn't expect to finish all that have been done in a period of ten weeks.
Appendix A. Distributed caching analysis

Overview

We will discuss a variety of options for distributed caching mechanisms. We mainly focus on in-memory caches. The mechanism must be compatible with Java Spring. There are 2 types of caches we can use:

- database connection caching
- request caching

Database connection caching

JBoss Cache

Link: http://jboss.cache.jboss.org

- distributed (data not on all nodes) or replicated (every node can have the same data) cache
- 100% open source
- peer-to-peer

⚠️ network traffic when there are many nodes

http://docs.jboss.org/jbosscache/3.2.1.GA/faq_en/html_single/index.html#d0e362

- objects are cached in-memory for efficient, thread-safe retrieval
- replicated to some or all cache instances in a cluster
- persisted to disk and/or a remote, in-memory cache cluster ("far-cache")
- garbage collected from memory when memory runs low, and passivated to disk so state isn’t lost
- attach to JMX consoles and provide runtime statistics on the state of the cache

Hazelcast In-Memory Data Grid (IMDG)

Link: https://hazelcast.com/use-cases/caching/

- easy setup with Spring @Cacheable annotation (just replace CacheManager) - https://hazelcast.com/use-cases/caching/spring-cache/
- Docker image - https://github.com/hazelcast/hazelcast-docker
- open source
- peer-to-peer
- Kubernetes discovery plugin

⚠️ quite complicated - not only distributed cache, but also distributed synchronization, clustering, processing, pub/sub messaging, etc.

- deployment: embedded (in application) or client/server
Java Caching System (JCS)

Link: [https://commons.apache.org/proper/commons-jcs/](https://commons.apache.org/proper/commons-jcs/)

- peer-to-peer (TCP Lateral Cache) or client/server (RMI Remote Cache)

Apache Ignite

Link: [https://ignite.apache.org/](https://ignite.apache.org/)

- integration with Spring
- standalone app

Infinispan

Link: [http://infinispan.org/](http://infinispan.org/)

- integration with Spring
- replicated or distributed
- embedded cluster with auto-discovery
- integration (+ auto discovery) in Kubernetes cluster

Memcached

Link: [https://memcached.org/](https://memcached.org/)

- fast: 200,000+ requests per second
- standalone app
- no official integration with Spring - [https://github.com/ragnar/simple-spring-memcached](https://github.com/ragnar/simple-spring-memcached), [https://thysmichels.com/2015/04/03/setup-spring-enablecaching-for-memcached/](https://thysmichels.com/2015/04/03/setup-spring-enablecaching-for-memcached/)
Redis

Link: [https://redis.io/](https://redis.io/)
- standalone app
- max value size 512 MB: [https://redis.io/topics/data-types](https://redis.io/topics/data-types)
  - Usage as a cache (LRU or LFU): [https://redis.io/topics/lru-cache](https://redis.io/topics/lru-cache)

Ehcache + Terracotta Server

- easy setup with existing Ehcache
- only one "stripe" (server + mirrors)
- client/server (but with mirrors)
- standalone app (not embedded)
  - multiple tiers for storing cache entries
- every application node contains subset of data
- other data accessible at terracotta

**Ehcache with replication**

Link: [http://www.ehcache.org/documentation/2.7/replication/rmi-replicated-caching.html](http://www.ehcache.org/documentation/2.7/replication/rmi-replicated-caching.html)

- simple setup
- reduced scalability - only replication (all keys on all nodes), not distributed
- docs found only for Ehcache v 2.7

**TayzGrid**


- peer-to-peer
- JCache API compatible
- easy Spring setup
- not sure about Docker compatibility
- not maintained (last version in 2015)

**NCache**

only .NET client (Java client commercial)
not maintained (last version in 2015)

Request caching

Squid

Link: http://www.squid-cache.org/
+ cache server hierarchy
+ distributed, replication
+ Docker image + Kubernetes
- bad docs

Varnish

Link: https://www.varnish-cache.org/
+ simple setup
- bad scalability - not distributed (single point of failure)
Appendix B. Expose JMX metrics from Kubernetes

Overview

When the PM API was deployed there were JMX metrics collected to Grafana to be able to see if the application is healthy. When the application is deployed in Kubernetes it is impossible to connect to individual pods as we don't know where are the pods and we cannot access them directly from outside. That is why we need an application inside the Kubernetes cluster collecting the JMX metrics for us exposing them as a single endpoint. For this purpose we use Prometheus pod running inside cluster, collecting the metrics and exposing them.

There are two ways how to collect JMX metrics from pods to Prometheus:

- Prometheus establishes JMX connection to each one of the PM API pods, gets the metrics and exposes them.
- We expose JMX metrics from the pod as an HTTP endpoint. Prometheus connects to the HTTP endpoint of each one of the pods and exposes them.

We decided to go for the second option as we only need to use existing solutions.

Requirements

- Checkout `svn+ssh://svn.cern.ch/reps/mpe-java/trunk/mpe/research/mpe-research-pm-api-docker`.

Deploy JMX Exporter

For exporting the JMX metrics as HTTP API we use JMX Exporter. We use it as a javaagent attached to the JVM. This javaagent is automatically included in PM API Docker image. It exposes the metrics on port 9954.

Modify the port

This port can be modified by changing it in java-vm-args in product.xml.

```
<java-vm-args>-javaagent:"lib/jmx_prometheus_javaagent-0.10.jar=9954:src/resources/jmx-exporter-config.yml"</java-vm-args>
```

Deploy Prometheus

Mount volume for Prometheus data
The Prometheus has to store the time series data in some place off the pod because the pod can get killed any time. That is why we need to use Kubernetes volumes, more specifically hostPath volume. HostPath volume is a directory mounted from the host i.e. node. We decided to use an Openstack volume for this purpose which needs to be attached and mounted to the node.

So in summary we attach and mount an Openstack volume to one of the Kubernetes nodes and then from the node we mount it to the pod.

The volume for Prometheus data is already created in Openstack (https://openstack.cern.ch/project/volumes/950d393d-40c9-4ef1-9097-ed56cd77465a/) and set up to be attached to node mp-q5eld6zwg5-2-pi2ervjivzed-kube-minion-jo4if2uwnpaz.

It is already set up to be automatically mounted to /mnt/prometheus-data on this VM startup.

Instructions to mount a fresh Openstack volume to a VM: http://www.darwinbiler.com/openstack-creating-and-attaching-a-volume-into-an-instance/

The second part of mounting (Kubernetes node to pod) is already configured in prometheus.yml Kubernetes config and will be done during the deployment.

Deploy Prometheus Docker image in Kubernetes

1. Create configmap containing config for Prometheus from file prometheus-config.yml.
   This config sets up Prometheus to scrape every service endpoint in Kubernetes that has annotation prometheus.io/scrape = true.
   Read more: https://prometheus.io/docs/operating/configuration/#<kubernetes_sd_config>  
   Example
   https://github.com/prometheus/prometheus/blob/master/documentation/examples/prometheus-kubernetes.yml
   kubectl create configmap prometheus-config --from-file=prometheus-config.yml

2. Provide external IP (your master node IP) in prometheus.yml. You can get your master node IP from Openstack.
   You can also change port on which the prometheus will be published.
   externalIPs:
   - <MASTER_NODE_IP>

3. [Optional] When a new cluster is created Prometheus needs to be assigned a VM that mounts the Openstack volume for Prometheus data. This is done by specifying the node for Prometheus in prometheus.yml config.
   a. Find labels for node on which the Prometheus data volume is mounted.
kubectl get node --show-labels

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>AGE</th>
<th>LABELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>188.185.68.54</td>
<td>Ready</td>
<td>10d</td>
<td>beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=188.185.68.54</td>
</tr>
</tbody>
</table>

b. Specify the label of selected node in node selector property in prometheus.yml. We choose label kubernetes.io/hostname because it is unique per each node. This way Kubernetes will always schedule the pod to run on this particular node.

```yaml
nodeSelector:
  kubernetes.io/hostname: 188.185.68.54
```

4. Deploy Prometheus.

```bash
kubectl create -f prometheus.yml
```

```bash
kubectl create --validate=false -f prometheus.yaml # if the first command fails
```

5. Test prometheus is running by connecting to http://<MASTER_NODE_IP>:9090 in your browser. You should see the Prometheus dashboard.

Navigate to Status -> Targets and you should Prometheus found your PM API pods and is scraping them.

## Targets

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>State</th>
<th>Labels</th>
<th>Last Scrape</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://10.100.17.3:9954/metrics">http://10.100.17.3:9954/metrics</a></td>
<td>UP</td>
<td>kubernetes_namespace=&quot;default&quot;</td>
<td>3.006s ago</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>instance=&quot;10.100.17.3:9954&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://10.100.19.3:9954/metrics">http://10.100.19.3:9954/metrics</a></td>
<td>UP</td>
<td>kubernetes_namespace=&quot;default&quot;</td>
<td>2.811s ago</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>instance=&quot;10.100.19.3:9954&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Navigate to Graph and you should be able to query the JMX statistics from the endpoints.
Configure available metrics

There are two options how to configure metrics sent to Prometheus.

- filter what gets scraped in JMX Exporter: 
  [https://github.com/prometheus/jmx_exporter#configuration](https://github.com/prometheus/jmx_exporter#configuration)
- filter data when it arrives to Prometheus: 
  [https://prometheus.io/docs/operating/configuration/](https://prometheus.io/docs/operating/configuration/)

We use the first option because scraping all the metrics from JMX takes a long time (~ 7s) and is useless if we filter it anyway when it arrived to Prometheus.

JMX Exporter config can be found in [mpe-pm-api-server-docker](https://mpe-pm-api-server-docker) repository in src/resources folder. We use whitelistObjectNames property to provide ObjectNames to scrape.