FAULT TOLERANT, SCALABLE MIDDLEWARE SERVICES BASED ON SPRING BOOT, REST, H2 AND INFINISPAN

W. Sliwinski[*], K. Kaczkowski[†], W. Zadlo[††], CERN, Geneva, Switzerland

Abstract

Control systems require several, core services for work coordination and everyday operation. One such example is Directory Service, which is a central registry of all access points and their physical location in the network. Another example is Authentication Service, which verifies caller's identity and issues a signed token, which represents the caller in the distributed communication. Both cases are real life examples of middleware services, which have to be always available and scalable. The paper discusses the design decisions and technical background behind these two central services used at CERN. Both services were designed using the latest technology standards, namely Spring Boot and REST. Moreover, they had to comply with demanding requirements for fault tolerance and scalability. Therefore, additional extensions were necessary, as distributed in-memory cache (Infinispan), or Oracle database mirroring using H2 database. Additionally, the paper will explain the tradeoffs of different approaches providing high-availability features and lessons learnt from operational usage.

INTRODUCTION

The potential for rapid growth of controls systems implies that every service has to be built to scale nearly instantly in response to growing requirements. CERN controls services are required to implement a high level of reliability, agility, and scale expected of modern computer systems.

High availability is a quality that aims to increase the time a service is available and it refers to systems that are durable and able to operate continuously without failure for a long time. It is generally achieved by scalability, failover and monitoring.

Two important CERN controls services, namely Authentication Service and Directory Service are examples of central, core services, which have to be always available, even during scheduled infrastructure upgrades or unexpected failures of dependent services. Both services are used in this paper to illustrate different architectural and design choices aiming at providing highly available, fault tolerant architecture, satisfying service requirements.

Authentication Service

Authentication Service (AS), at CERN part of the RBAC [1, 2] infrastructure, is a central authority, which verifies caller's identity, be it a human or an application, and issues a signed token, which represents the caller in the distributed communication. Users token holds several pieces of information, which are necessary to obtain access to protected resources, including: username, account type, list of roles, IP address and location name. AS provides several different types of authentication: explicit (username and password), location (trusted hosts by IP address), Kerberos [3] and SSO (SAML based Single-Sign On). This is made possible by aggregating different authentication mechanisms available at CERN and providing a common REST API [4] to all users. Figure 1 depicts the service architecture:

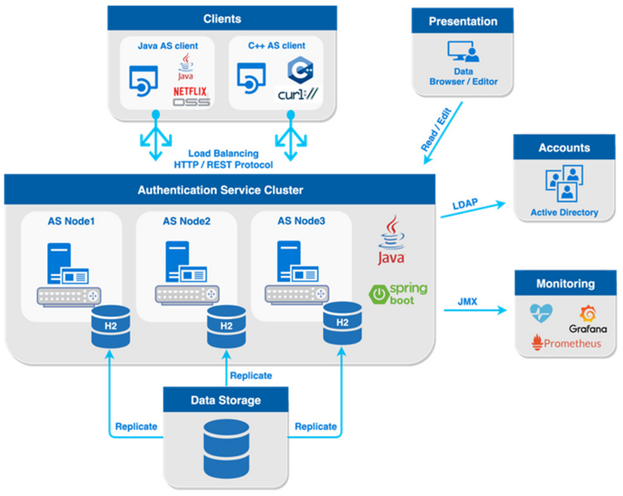

Figure 1: Authentication Service architecture.

CERN's controls middleware framework RDA3 [5] integrates with AS to provide security facilities (authentication and authorization) for RDA3 clients and servers.

Directory Service

Directory Service (DS) is a central registry of all access points in the distributed control system (Fig. 2). It provides up-to-date information about the actual physical location of a device server in the network. This is possible, because each device server has to register its current location during the start-up phase. Additionally, DS resolves logical device names to actual device servers and returns the location information to the client. Thanks to this, high-level applications don't need to know any information related to a device server; only unique device name is sufficient to initiate communication.

The RDA3 communication stack depends on DS for server's binding registration, device to server resolution and server's location and device lookup queries.

** Wojciech.Sliwinski@cern.ch
† Konrad.Kaczkowski@cern.ch
†† Wojciech.Zadlo@cern.ch

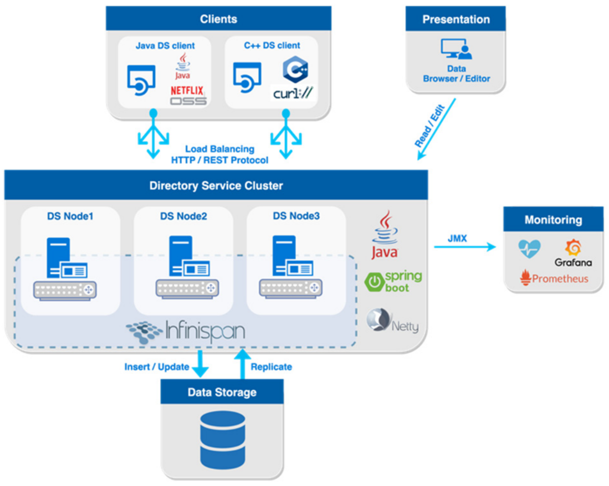

Figure 2: Directory Service architecture.

DESIGN FOR HIGH AVAILABILITY

Performance

Performance means system throughput under a given workload for a specific time frame. It is an ongoing process and not an end result. DS and AS have been designed taking into consideration scalability and reliability of hardware, software and network. Their requirements oblige to handle hundreds of requests per second and to accommodate specific business requirements.

Communication Style

Communication between services and execution flow is a fundamental decision for a distributed system. It can be synchronous or asynchronous in nature. Both approaches have their trade-offs and strengths.

Asynchronous communication allows for better use of available system resources (CPU, threads) and is indispensable for scalable systems as it allows to serve more concurrent requests with less resources.

Synchronous communication is closely associated with HTTP protocol and REST API. It is used for both AS and DS REST clients. Synchronous communication leaves upstream services susceptible to cascading failure in a microservices architecture. If downstream services fail or even worse, take too long to respond back, it might lead to client breakdown. For DS, although all communication is over HTTP/REST, it is necessary to use asynchronous server-side processing for registering new device server's bindings to avoid the failure of application requests due to the breakdown of one DS node. For AS, all update operations on the database are done asynchronously to not block application threads in case of database failures.

Caching Architectures

Caching is a mandatory requirement for building scalable, resilient distributed systems. As data and applications continue to get larger and faster, the data has to be available instantly to users. Even with a relatively fast storage layer, loading big amounts of data can take significantly more time than the desired time to serve a user request.

DS requires a great deal of data to be loaded into memory in order to respond to user calls as fast as possible. Depending on the need, the data can be cached in many different ways.

A local cache is the simplest approach, when the data is stored locally and its changes are not replicated. Another option is a remote, central cache available to many users. The major advantage of a centralized cache is that once the data is loaded into the common data store it can be queried by many clients.

In Tables 1 and 2, all evaluated cache products are summarized together with their main features.

Table 1: Comparison of Distributed Caches, Part 1

	Infinispan	**Ignite**
Transport	JGroups (TCP/UDP)	TCP
Storage	memory, db, file	memory, db, file
Deployment	in-process	in-process, server
Strategy	replication, distributed	replication, distributed
Client API	Java, C++, Python	Java, C++, Python

Table 2: Comparison of Distributed Caches, Part 2

	Hazelcast	**EhCache**	**Redis**
Transport	UDP, multicast	RMI, JMS, JGroups (TCP/UDP)	TCP
Storage	memory, db	memory, file	memory, file
Deployment	in-process	in-process, server	server
Strategy	distributed	replication, distributed	replication
Client API	Java, C++, Python	Java	Java, C++, Python

For DS the chosen solution is an in-process cache combined by the Red Hat Infinispan cluster [6]. Its data is fully replicated; therefore, every DS node has quick access to the whole data set and all reads are always local. In the distributed cache approach, changes are replicated to a fixed number of nodes and reads request the value from at least one of the owner nodes in the cluster.

On top of mentioned aspects, we differ also client-side and server-side caching. Client-side approach can be observed with most web browsers and has the benefit of reducing network latency and remote storage I/O. It also protects a server from being overloaded from client requests.

Server-side caching on the other hand is by far the most reliable and fastest method of caching available. It is useful for high volume transactions that can be kept secure. It also provides the highest degree of control over invalidation. In

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOBPP03

DS, both client-side (included in the client API library) and server-side caching are employed.

Another important aspect that has to be considered is the expiration policy for cached data. A solution implemented in DS is a periodic refreshing of the whole data set. As an alternative strategy, a time to live (TTL) on each record can be set.

Data Replication

When the data set size is too large or it may grow with an unpredictable rate, using a distributed, fully replicated in-memory cache is not an optimal approach. The main problem is that the whole data set might not fit into the available system memory and moreover cache refresh operations may incur heavy network traffic due to significant data size, which has to be transported to all cluster nodes.

This scenario is the case for AS where the configuration data needed for operation might have very big size (>1GB). Moreover, the data can be frequently changed by users and therefore AS should always use the latest values. Therefore, a periodic upload of the whole data set into a common cache, as in DS, is not an appropriate strategy as the cached data would immediately become stale and that would cause wrong results to user's requests. However, in order to survive possible, temporary unavailability of the master database it was decided to investigate full data replication mechanism into a local database instance, fully managed by each AS node.

Table 3: Comparison of Embedded Databases, Part 1

	H2	HSQLDB
Storage	memory, file	memory, file
Deployment	in-process, server	in-process, server
Oracle syntax similarity	high	exact
SQL support	procedure, trigger, sequence	procedure, trigger

Table 4: Comparison of Embedded Databases, Part 2

	SQLite	Derby
Storage	memory, file	memory, file
Deployment	in-process	in-process, server
Oracle syntax similarity	low	medium
SQL support	low	procedure, trigger, sequence

Finally, after an evaluation of several different products H2 database [7] was chosen to implement a local, embedded database mirror (Table 3). It was configured to run as a private in-memory database, with file-based storage. Every 15 minutes a new upload of the full data set is performed to update the local H2 instance on each AS node. In operation, AS by default accesses the master database, however when it becomes unavailable it switches immediately to the local H2 instance. Therefore, there is no failure reported to end user as data is always available. Evidently, locally replicated data could be outdated by a maximum of 15 minutes, but it is acceptable in view of achieving better system availability.

Scalability

While performance is a measure of how efficiently an application processes the requests, scalability is related to how to divide and conquer the processing of incoming tasks. Infinispan that has been chosen for implementing the cache layer allows to discover neighboring instances on the same local network and forms a cluster of multiple nodes. Therefore, DS is able to scale horizontally and can be expanded at any time with additional nodes with identical functionality, redistributing the load among all of them. Similarly, AS distributes the load among all available nodes. However, since every AS node has a direct access to the datastore, additional nodes increase load on the database.

Load Balancing

Load balancing allows for distributing incoming requests among all healthy cluster nodes, so no single node gets overloaded. It also provides an ability to self-heal after a particular node becomes again available, without any service downtime for end users. Load balancing models can be divided into server-side (Fig. 3) and client-side (Fig. 4).

Popular choices for server-side load balancing include HAProxy [8], Nginx [9] and Amazon's Elastic Load Balancer (ELB) [10].

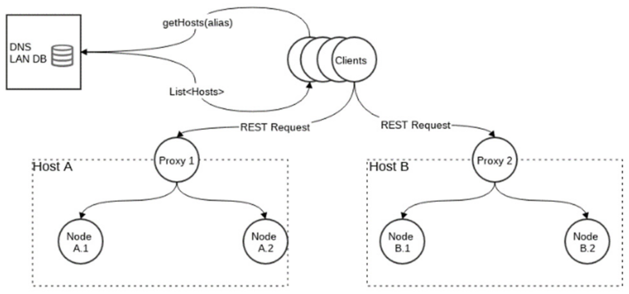

Figure 3: Server-side load balancer.

Unfortunately, using a single central load balancer for an entire service ecosystem can be a single point of failure, leading to a failure of the entire service. Also, this single load balancer can very quickly become a major bottleneck, since all traffic to every microservice has to pass through it. Another possibility is to use clustered multiple load-balancers, however, what has to be considered is that it comes with a lot of additional configuration overhead. Another issue with central load balancers is scalability. Thus, for AS and DS it has been decided to use client-side load balancing.

There are multiple client-side solutions available on the market. For the new DS Java client, the Netflix Ribbon [11] product has been chosen. In this approach, load balancing is fully distributed, with each client directly responsible for routing requests to an available microservice (Fig. 4).

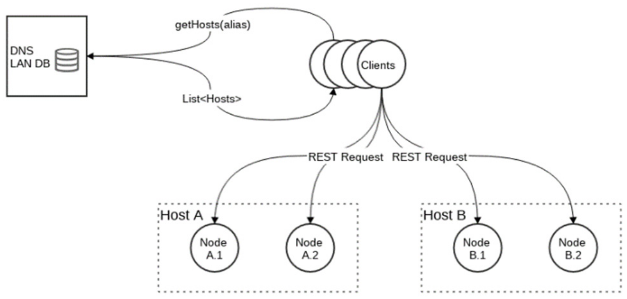

Figure 4: Client-side, in-process load balancer.

It simplifies service management and automatically scales the system in accordance with the number of available instances and eliminates all single points of failure.

On the other hand, AS client uses a custom in-house load-balancer, but it is planned to upgrade it to Netflix Ribbon as well. For C++ clients a custom solution based on libcurl [12] was implemented with a similar functionality.

Redundancy

It is important to build enough redundancy into the system to ensure that the service does not fail. To improve the availability, it is essential to eliminate all single points of failure and create, using a server load balancer, clusters in which all nodes are stateless and completely equivalent. A successful microservice implementation has redundant copies of each service.

There are some consequences from replicated nodes approach to managing data. Replication causes redundancy across the data stores, as the same item of data is appearing in multiple places. However, this allows the system to withstand errors and crashes in individual service nodes and simplifies the architecture.

Both AS and DS clusters, are organized as replicated load-balanced services, where every server node is identical to every other node and all are capable of supporting the traffic. This approach has a major impact on a general service reliability in case of unavailability of some server nodes, e.g. due to: HW failure, OS upgrade, etc.

Failover

It is important to design critical services in a fail-safe manner. In both AS and DS services fallback mechanisms are used to fail gracefully when external services (e.g. database, LDAP [13]) are not available.

Health checks can help us detect failed hosts so the load balancer can stop requests to them. A host can fail for many reasons, such as simply being overloaded, the server process may have stopped running, it might have a failed deployment, or broken code to list a few reasons. We distinguish between passive and active health checks.

In active health checks, the load balancer periodically "probes" upstream servers by sending a special health check request. If the response is not received back from the upstream server, or if the response is not as expected, the load balancer disables traffic to the server. In passive health checks, the load balancer monitors real requests as they pass through. If the number of failed requests exceeds a threshold, it marks the host as unhealthy. In AS and DS there are passive health checks with detecting the unhealthy server nodes. For AS and DS Java clients it was implemented using Netflix Ribbon, similarly as for load balancing mechanism. For C++ clients a custom solution based on libcurl was implemented with similar functionality. If the request sent from a client fails, it is automatically re-sent to another server nodes, and the previous one is temporarily banned from further requests. This reduces the load on the unhealthy node, and prevents resource exhaustion in the client.

Monitoring and Logging

One of the biggest challenges due to the very distributed nature of microservices deployment is monitoring and logging of individual service nodes. Since both AS and DS are Java based applications, JMX [14] was chosen to implement and expose service metrics. Additionally, many useful JMX metrics are already provided by Java platform and dependent 3rd party components: Spring [15], Infinispan, H2. Next, they are exposed and ingested by a Prometheus [16] server instance, which records them in a time-series datastore. Prometheus allows for easy querying of stored metrics and creation of alerts for detecting system anomalies. This proved to be very efficient in both production and test environments to be able to spot errors before they affect users. Next step was to configure monitoring dashboards in Grafana [17] based on Prometheus data source. Visually appealing dashboards in Grafana help to get an overview of the services' health and allow for browsing historical data, very much needed for issue troubleshooting.

Without monitoring in place, an operation team may run into trouble managing a large-scale microservice. An efficient monitoring helps to understand the behavior of a system from a user experience point of view. This will ensure that the end-to-end behavior is consistent and is in line with what is expected by the clients.

CONCLUSIONS

Both services Authentication Service and Directory Service have successfully passed many testing phases, including: integration, performance and stability testing. Before the final production deployment, the test setup ran continuously for several days without any problems proving that the new, highly available system architecture can handle reliably twice the production load. Finally, both services were successfully commissioned and deployed in operation: Authentication Service in 2017 and Directory Service in 2018.

REFERENCES

[1] P. Gajewski, S. R. Gysin, and K. Kostro, "Role-based authorization in equipment access at CERN", in *Proc. 11th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'07)*, Oak Ridge, TN, USA, Oct. 2007, paper WPPB08, pp. 415-417.

[2] S. R. Gysin, C. L. Schumann, and A. D. Petrov, "User authentication for role-based access control", in *Proc. 11th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'07)*, Oak Ridge, TN, USA, Oct. 2007, paper TPPA12, pp. 111-113.

[3] Kerberos, `https://web.mit.edu/kerberos/`

[4] REST, `https://en.wikipedia.org/wiki/Representational_state_transfer`

[5] J. Lauener and W. Sliwinski, "How to design & implement a modern communication middleware based on ZeroMQ", in *Proc. 16th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17)*, Barcelona, Spain, Oct. 2017, pp. 45-51. `doi:10.18429/JACoW-ICALEPCS2017-MOBPL05`

[6] Infinispan, `https://infinispan.org/`

[7] H2, `https://www.h2database.com/`

[8] HAProxy, `http://www.haproxy.org`

[9] Nginx, `https://nginx.org/en`

[10] Amazon ELB, `https://aws.amazon.com/elasticloadbalancing`

[11] Netflix Ribbon, `https://github.com/Netflix/ribbon`

[12] libcurl, `https://curl.haxx.se/libcurl`

[13] LDAP, `https://ldap.com`

[14] JMX, `https://openjdk.java.net/groups/jmx`

[15] Spring Boot, `https://spring.io/projects/spring-boot`

[16] Prometheus, `https://prometheus.io`

[17] Grafana, `https://grafana.com`