16th Int. Conf. on Accelerator and Large Experimental Control Systems ICALEPCS2017, Barcelona, Spain JACoW Publishing
ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPL01

# C2MON SCADA DEPLOYMENT ON CERN CLOUD INFRASTRUCTURE

B. Copy, M. Braeger, E. Mandilara, F. Ehm, A. Lossent
CERN, Geneva, Switzerland

*Abstract*

The CERN Control and Monitoring Platform (C2MON) [1] is an open-source platform for industrial controls data acquisition, monitoring, control and data publishing. C2MON's high-availability, redundant capabilities make it particularly suited for a large, geographically scattered context such as CERN. The C2MON platform relies on the Java technology stack at all levels of its architecture, and previously imposed the deployment of binary archives that needed to be unpacked and executed locally. Since end of 2016, CERN offers a platform as a service (PaaS) offering based on RedHat Openshift [2]. Initially envisioned at CERN for web application hosting, Openshift can be leveraged to host any software stack due to its adoption of the Docker container technology, including the Java dependency stack that C2MON is based upon. In order to make C2MON more scalable and compatible with Cloud Computing [3], it was necessary to **containerize** C2MON components for the Docker container platform. Containerization is a logical process that forces one to rethink a distributed architecture in terms of decoupled micro-services and clearly identify dependencies in terms of services, storage requirements, configuration and connectivity, without ever imposing any physical considerations, which would in any case jeopardize the redeployment of the distributed architecture in another cloud environment. In return, the deployment of the said distributed architecture becomes reproducible and entirely automatable.

This paper explains the challenges met and the principles behind containerizing a server-centric Java application, demonstrating how simple it has now become to deploy C2MON in any cloud-centric environment (ranging from Openstack Magnum to Docker Swarm, and of course Openshift).

## C2MON USAGES AT CERN

C2MON [1] is a monitoring platform developed at CERN and since 2016 made available under an LGPL3 open source license. C2MON employs Java messaging, caching and clustering technologies to deliver robust, scalable and monitoring of data of any kind, with a particular focus on industrial control systems.

C2MON is at the heart of the CERN Technical Infrastructure Monitoring (TIM) that supervises the correct functioning of CERN's technical and safety infrastructure. TIM handles about three million messages per day.

C2MON is also used by DIAMON2 [4], CERN's accelerator infrastructure to monitor a large majority of the equipments that compose it, ranging from servers to consoles, through front-end computers and PLCs. DIAMON2 handles an average of twenty million messages per day.

## ADAPTING FOR THE CLOUD

Over the past couple of years, CERN has embraced cloud technology by replacing the majority of its computing infrastructure by Openstack at a record pace [3]. Cloud technology presents significant advantages for large organizations by allowing a more precise and more agile sharing of available resources. It promotes device and location independence by forcing users to design their software architectures in terms of remote resources. It also simplifies reusing and duplication of entire groups of machines for testing and validation purposes. Last but not least, cloud deployments introduce support for load balancing, circuit breaking and rolling updates in a near-transparent manner, which prior to this would have required the usage of proprietary, complex and technology-specific solutions.

Cloud technology is perfectly suitable for deploying pre-cloud era legacy applications thanks to virtualization technology. Legacy applications that rely on low-level operating system devices (such as storage or network adapters) can easily be relocated on a cloud and thus gain a new home away from any cumbersome hardware constraints.

However, with regards to this last point, a number of aspects need to be carefully considered in order to benefit more completely of a cloud infrastructure :

- Usage of storage, process and network resources.
- Support for failures, low availability, health metrics.
- Support for clustering and configuration injection.

### *Usage of Storage, Process and Network Resources*

Typical pre-cloud era applications expect a file system to be available along with one or more local network connections. Cloud-based deployment can certainly fulfil such expectations, but for scalability and relocation's sake, file systems are usually transient (i.e. they are reset upon restart) and network interfaces typically allocated on the fly with a randomly-generated hardware address and attached to a local private, non-routable network.

Likewise, the life cycle of a cloud container hosting an application is linked directly to its main process ; this means that a web application server process that stops will immediately terminate its hosting container and be signalled to the cloud infrastructure as inactive, ready to be removed. This is an essential feature of a cloud infrastructure which allows for unused resources to be garbage-collected and reallocated immediately. Processes must therefore be managed not at the level of the host they are running on, but at the level of the hosting cloud : They must log their activity in a central location (any local activity logs written prior to the process stopping will be immediately lost) and must be able to clearly indicate their current status and general health through easily reachable metrics.

As such, pre-cloud era applications do not support the aforementioned features. Thankfully, numerous tools such as logstash [5] or telegraf [6] exist to perform log forwarding and health metrics publication in a non-intrusive manner and can be easily injected into container images to extract health metrics from legacy applications .

### *Support for Failures and Resource Scarcity Conditions*

Cloud environments aim to make the best usage of resources; commercial clouds indeed charge their users by the thousandth of CPU unit and per byte of RAM per second. After decades of Moore's law and multi-core servers, computing infrastructure users must once again think carefully about their resource usage. Another important aspect is the inherent lack of robustness - cloud machines are based on commodity hardware with a high failure rate. It is therefore essential for any application to expose its own load and health metrics, so that the hosting cloud can intelligently adjust the effort requested from a given running process.

### *Support for Clustering and Configuration Injection*

Deployments in a cloud are by definition approximative and elastic. To benefit fully from such a dynamic runtime environment, application processes should :

- accept to run in any location inside the cloud that satisfy its working conditions.
- be able to self-configure according to existing meta-data that is injected in their environment by the hosting cloud.

By accepting to run in any location, cloud deployments are more effective if they only specify "soft preferences" over "hard constraints". For instance, an application server node can deploy as part of a cluster by either :

- expressing the preference that it wants to run as close as possible to the database server nodes backing it.
- mandating the constraint that it must run on the same node as a database server (to avoid network latency for instance).

In the former, the hosting cloud will be more resilient, as it can relocate or re-instantiate the application server node anywhere within reach of the backing database with more flexibility, giving the service a better chance of remaining in nominal condition. In the latter, the application server node may have to wait until suitable resources on a database hosting node become available.

Adapting an existing application for the cloud requires the adoption of new habits with regards to resources, monitoring and configuration. While these points are nothing fundamentally new in the world of computing, these points have become a necessity in a cloud, and environments such as Red Hat Openshift [2] will not be able to deploy the application at all.

Once such considerations have been addressed, the application components must be "containerized" to be sent to the cloud. Containerization is a systematic process that can be both frustrating and rewarding, as it forces a software developer to inspect and sanitize entirely the technology stack he or she can easily take for granted. It always starts from identifying the most minimal operating system and set of dependencies generic enough to operate the application then refines the image in terms of software artefact injection, configuration support, user privileges and possible dependency on other containers.

## CONTAINERIZATION PROCESS

Cloud containers are wrappers around a single process, allowing said process to benefit from the operating system services it expects to find (e.g. access to a file system, access to network handles, access to low-level system libraries ).

In order to containerize an existing application, the following steps are necessary :

- Identify a minimal base image.
- Inject software artefacts via a delivery pipeline.
- Add support for external configuration.
- Review minimal required user privileges.

### *Identify a Minimal Base Image*

Cloud containers are essentially "a computer inside a computer". However the container does not need to replicate all functions and services of a computer - first off, it can perfectly forego the presence of unnecessary devices and services; second, it can piggyback on the host computer's essential features such as its kernel, process management and, if needed by the application it executes, even hardware devices. It is therefore recommended for the container to run off a base container image, exhibiting only minimal functions. As container technology emerged, dedicated operating system distribution such as *Alpine Linux* can fit in a few megabytes what is needed to execute a process. Naturally, these minimal base images need to be enriched with the required software artefacts, as very few modern applications are developed without relying on a large number of software dependencies.

### *Inject Software Artefacts via a Delivery Pipeline*

The application's compiled code (typically executable files) needs to be injected inside the container image. Build systems such as Apache Maven or Openshift S2I [7] already support the creation of a container image and injection of build artefacts directly into the image, as well as the deployment of the container to the cloud. Such a sequence is typically referred to as "a delivery pipeline" [8], whereby the source code of an application can be automatically delivered to a runtime environment without interruption of service via rolling upgrades.

### *Add Support for External Configuration*

At runtime, the application needs to obtain information to adapt its behaviour. For instance, it might want to know if it is a test or production instance, where it can store persistent data etc... Applications such as C2MON expect environment variables and Java system properties, along with well-defined files to be provided upon start-up. Container technology solves these problems by injecting file system mount points (typically referred to as "volumes") and environment variables. Modern cloud-centric applications prefer distributed key-value stores such as ZooKeeper [9] over configuration files and environment variables - this allows a centrally managed approach with support for versioning. Spring Cloud Configuration [10] is also a very compelling solution for applications written with the Spring Framework such as C2MON, it integrates transparently with source versioning systems such as Subversion or Git, making it very simple to deploy and with a lower cost of maintenance than ZooKeeper, and still allows to upgrade to a ZooKeeper backend should the situation require it.

### *Review Minimal Required User Privileges*

Containers in a cloud environment must be isolated from each other as much as possible to ensure they are easily replaceable and that they do not impact any other third-party applications that could be co-located in the same cloud. Container technology places a strong emphasis on security and stopping "container escape" - the possibility for a container application to escape its boundaries and interact with the host operating system, allowing an attacker to gain control of the cloud infrastructure the application is running on. The first line of defence against container escapes is to ensure the container runtime user has no privileges on its host system. One popular strategy for cloud implementations is to assign containers a random user identifier, ensuring the container runtime is unprivileged. It is rather unusual for traditional applications to operate that way, and steps must be taken to grant enough privileges upon start-up to this randomly assigned user [11].

### *Specify Dependencies on Other Containers or Non-Cloud Based Resources*

Applications in an enterprise-type environment such as CERN rarely operate by themselves : they depend on existing services and resources available on a large scale (for instance naming directories, enterprise database servers). Since containers aim to run a single process inside each instance, it is preferable to deploy one type of container per architecture element present in the application (e.g. application server, web front-end server, message broker). As stated earlier, containers cannot make any assumptions about the location of other containers they depend on. The hosting cloud environment is the one in charge of orchestrating deployment and only it knows where each container is actually located. As a result, cloud hosting platforms tend to group distinct types of containers into **services**. Services declare a known ingress point, from which requests are then dispatched to individual containers fulfilling the same function [12]. This allows containers to be added, destroyed or upgraded without any loss of service, and dependent containers always know how to obtain service. Furthermore, if the service implementation is stateless or capable of clustering, such an indirection can transparently support load-balancing and circuit breaking which further contributes to the robustness of the overall application.

Access to non-cloud based resources also relies on **service definitions** (typically through a known network address). In that case however, advanced features such as load-balancing or health monitoring cannot be provided by the cloud itself.

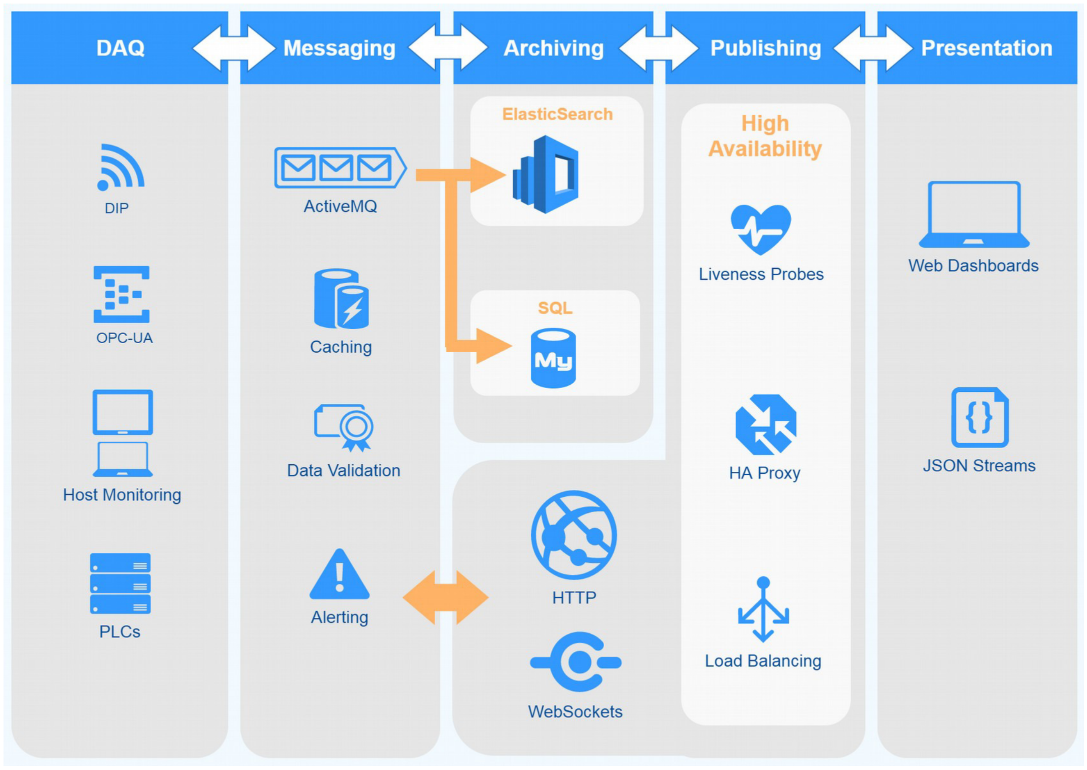

Figure 1: C2MON general architecture.

## C2MON TECHNOLOGY STACK

Using C2MON and the CERN cloud infrastructure as a case study, let us consider each of its constituting elements before evaluating how suitable they are for cloud deployment.

Figure 1 presents the general C2MON architecture.

The C2MON architecture relies on the following majors components :

- ActiveMQ Message Brokers
- Clustered caching
- DAQ processes
- Configuration persistence and archiving
- Client interfaces

### *ActiveMQ Message Brokers*

ActiveMQ is a message broker software. C2MON uses ActiveMQ brokers for all its message exchanges (with the exception of web broadcasting, which relies on the cloud-enabled Atmosphere framework). ActiveMQ is a robust and widely used messaging service. Its implementation however is synchronous, by opposition to asynchronous messaging facilities such as Apache Kafka [13], thereby limiting its general performance. The ActiveMQ project was introduced in 2001 and its architecture is not currently a natural fit for a cloud deployment in the CERN cloud environment : It only supports cluster member auto-discovery via multicast, which is not allowed in the CERN Openshift infrastructure, and its configuration relies on file system resources and system properties, which as explained in section "Add support for external configuration", reduces its scalability opportunities.

Another important limitation of ActiveMQ is that it relies primarily on TCP communications through well-known ports, which are inapplicable in a cloud environment, where hostnames and ports cannot be relied upon. ActiveMQ supports other transport protocols such as HTTP, but tests proved that this implementation is incompatible with high-availability proxies such as HAProxy - as long running DAQ sessions eventually cause HAProxy to run out of memory.

### *Clustered Caching*

C2MON uses EhCache and Terracotta to maximize its availability and performance. Similarly to ActiveMQ, Terracotta does not support cluster auto-discovery - it relies on hostnames or IP addresses, both inapplicable in a cloud environment. Redundant terracotta can however be deployed by declaring one cloud service per server, with the obvious consequence that such a setup would have to be scaled up manually.

### *DAQ Processes*

DAQ processes are configured via property files and environment variables. They do not support redundancy or clustering, deploying them in a swarm would therefore bring no benefit.

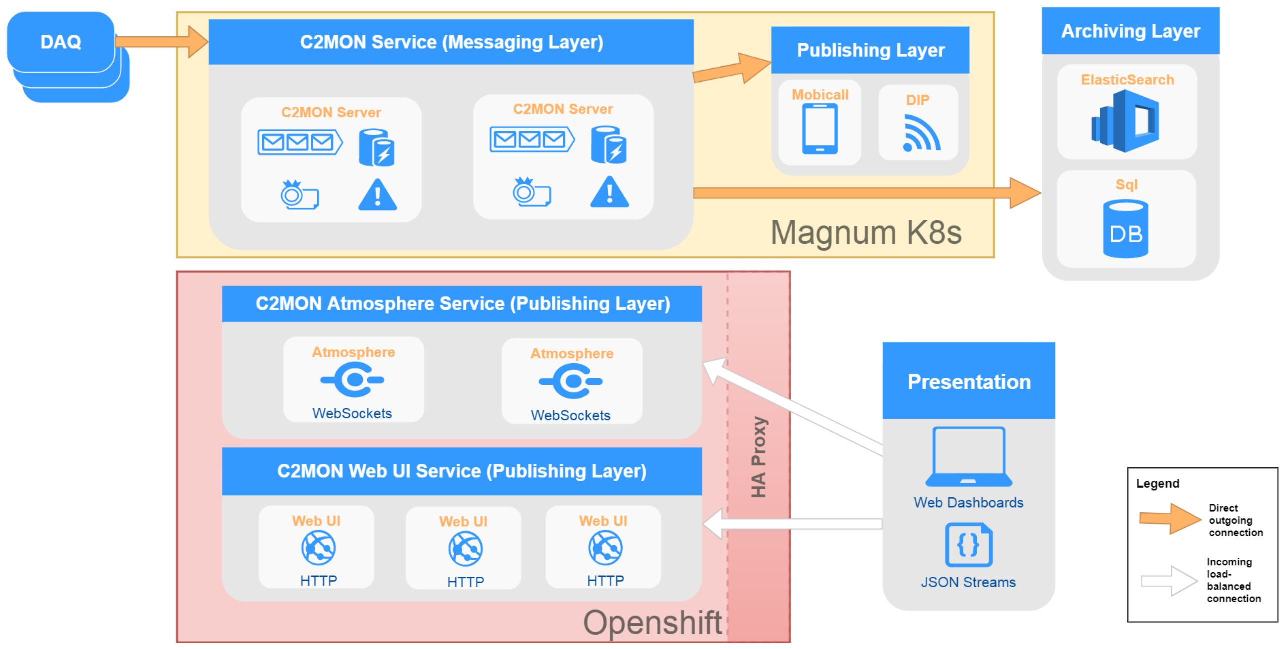

Figure 2: C2MON deployment over the CERN cloud infrastructure.

### *Configuration Persistence and Archiving*

C2MON can archive data to ElasticSearch or an Oracle or MySQL database. It also persists its runtime configuration to a relation database such as Oracle or MySQL. CERN already provides a high-availability ElasticSearch cluster service, as well as a database on demand service that supports MySQL and Oracle. In the case of an environment were neither would be provided, new high-availability database implementations such as CockroachDB that is fully compatible with PostgresDB is an ideal candidate to move relational databases to a cloud environment. In our case, this option was not explored due to lack of resources.

### *Client Interfaces*

C2MON can expose its data to users through two mechanisms :

- the Web UI, which is an administrative interface to C2MON internals. Not destined for end-users, it is an essential tool to monitor a C2MON cluster. It relies on a stateless application server, requires little to no configuration (since it merely reflects the one contained in its C2MON cluster) and is an ideal candidate for cloud deployment.
- the WebSocket client extension which provides scalable streaming support for end-user web interfaces. Based on the Atmosphere framework and compatible with numerous clustering back-ends, it is also ideal for cloud deployment.

Now that we have reviewed each architectural element, we can review our actual implementation in the CERN environment.

## CLOUD MIGRATION RESULTS

Figure 2 presents the final C2MON cloud deployment as performed over the CERN cloud infrastructure. The CERN cloud offers, amongst others, two implementations particularly suited for C2MON deployments : Openshift and Openstack Magnum [14].

Openshift, as implemented at CERN, is aimed at web hosting. All incoming traffic is handled by HAProxy, a high-availability load balancer that also prevents other types of trafic. As explained in the “ActiveMQ Message Brokers” section, the ActiveMQ HTTP transport is ill-suited for long-running sessions and causes HAProxy to run out of memory.

The Openstack Magnum hosting infrastructure on the other hand supports general purpose cloud hosting via the Docker swarm [14] technology, which allows TCP traffic and well-known ports and services, making it possible to deploy C2MON server instances, albeit without any support for high-availability and load balancing.

Other elements such as the C2MON Atmosphere service or the C2MON Web UI service are ideally suited for Openshift, supporting rolling upgrades and automated deployments out of the box.

Migrating existing applications to make the best out of a cloud environment, if not effortless, is nevertheless possible with the help of a suitable build and deployment infrastructure, and provided enough attention is brought to deployment specificities, security requirements and load balancing constraints imposed by the hosting cloud.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THBPL01

## REFERENCES

[1] M. Braeger *et al.*, “High availability monitoring and big data : using Java clustering and caching technologies to meet complex monitoring scenarios”, MOPPC140, Oct 2013, ICALEPCS’13, San Francisco, USA.

[2] A. Lossent and A. Peon, “PaaS for web applications with OpenShift Origin”, Oct 2016, CHEP 2016, San Francisco, USA.

[3] T. Bell, “Cloud Computing Infrastructure at CERN”, March 2015, HEPTech conference, Budapest, Hungary.

[4] W. Buczak *et al.*, “Diamon2- Improved Monitoring of CERN’s Accelerator Controls Infrastructure”, THCOBA03, Oct 2013, ICALEPCS’13, San Francisco, USA.

[5] J. Hamilton *et al.*, “SCADA Statistics Monitoring Using the Elastic Stack”, TUPHA034, Oct 2017, ICALEPCS'17, Barcelona, Spain.

[6] A. Lahiff, “Monitoring with InfluxDB and Grafana”, Fall 2015, HEPiX Workshop, New York, USA.

[7] S. Picozzi *et al.*, “Source to Image”, in *DevOps with OpenShift*, O'Reilly Media, Sebastopol, CA, 2017.

[8] V. Naik, “Architecting for Continuous Delivery”, *Thoughtworks continuous delivery blog*, 11 Jan 2016, `https://www.thoughtworks.com/insights/blog/architecting-continuous-delivery`

[9] V. Farcic, “Service Discovery – the key to distributed services”, in *DevOps 2.0 Toolkit*, Birmingham, UK, Packt Publishing, 2016.

[10] J. Carnell, “Controlling your configuration with Spring cloud configuration server”, in *Spring Microservices in action*, Greenwich, USA, Manning Publications, 2017.

[11] *Openshift container platform v3.6 documentation*, , “Creating images : guidelines”, Red Hat Software, Raleigh, USA, Jul 2017, `https://docs.openshift.com/container-platform/3.6/creating_images/guidelines.html`

[12] *Openshift container platform v3.6 documentation*, , “Deployment”, Red Hat Software, Raleigh, USA, Jul 2017, `https://docs.openshift.com/container-platform/3.6/creating_images/guidelines.html`

[13] D. Vohra, “Messaging and indexing : Apache Kafka”, in *Practical Hadoop Ecosystem*, New York, USA, 2016.

[14] S. Trigazis, “OpenStack Magnum and the CERN cloud”, OpenStack User Group, Paris, France, 2017.