HIGH-AVAILABILITY MONITORING AND BIG DATA: USING JAVA CLUSTERING AND CACHING TECHNOLOGIES TO MEET COMPLEX MONITORING SCENARIOS

M. Bräger, M. Brightwell, E. Koufakis, R. Martini, A. Suwalska, CERN, Geneva, Switzerland

Abstract

Monitoring and control applications face ever more demanding requirements: as both data sets and data rates continue to increase, non-functional requirements such as performance, availability and maintainability become more important. C²MON (CERN Control and Monitoring Platform) is a monitoring platform developed at CERN over the past few years. Making use of modern Java caching and clustering technologies, the platform supports multiple deployment architectures, from a simple 3-tier system to highly complex clustered solutions. In this paper we consider various monitoring scenarios and how the C²MON deployment strategy can be adapted to meet them.

INTRODUCTION

The world of Monitoring and Control software is vast, stretching from the advanced systems installed in today's modern factories and scientific installations, all the way to the software monitoring a small IT cluster or your home IP-enabled fridge. Within this diversity, many of the core functionalities remain the same, namely that of gathering, storing and displaying data and processing related alarms ("monitoring"), and taking appropriate actions - either manually or automatically - based on this information ("control"). The functional differences will often lie in the drivers for connecting to the specific hardware and the algorithms behind complicated control processes. That said, major differences may appear in the non-functional requirements, and this appears to be often overlooked. Of course, all users would hope for the software to be reliable! But when you start looking at availability, maintainability, adaptability or performance, major differences soon appear, depending on the monitoring service you are trying to run. For instance, in terms of maintainability, a software upgrade that puts your fridge temporarily offline is of little importance whereas the inability to introduce a critical patch at runtime may be a major downside of a 24/7 monitoring service. Similarly for adaptability, if your monitoring system doesn't have a predictably steady data flow, it must bust be designed to respond to a sudden avalanche of data without significantly degrading performance.

With these special requirements in mind, the C²MON was created in 2009 to design and implement a common monitoring and control platform [1]. Providing the core functionalities of a monitoring system, this software aims to be extendable and adaptable to a wide variety of monitoring requirements, both functional and non-functional. In this paper we focus on the non-functional requirements mentioned above, and how the C²MON architecture was designed to meet these. In particular, we illustrate how the integration of modern Java caching and clustering solutions allowed sufficient modularity in the deployment strategies, for most of the requirement scenarios to be met. In this case the overused term of "leveraging" is in fact appropriate, since the C²MON architecture builds on 3rd party products to provide a much more comprehensive software suite.

The paper is organized as follows: in the first section we review Java caching and clustering technologies and how they have evolved over the past few years. The second section gives a brief overview of the C²MON design, referring to other publications. The heart of the paper is in section 3, where we illustrate C²MON deployment architectures through a number of different monitoring scenarios.

REALIZING CACHING AND CLUSTERING ON JAVA ENTERPRISE TECHNOLOGIES

A major aim of the C²MON project was to provide a clustered server layer that was able to consume data updates in a load-balanced manner. This setup allows a better handling of data avalanches and a higher protection against network or hardware failures. Another big advantage of a clustered setup is the possibility of applying patches or functional upgrades transparently to the system without creating any service downtime. This can be achieved through rolling updates, which means that the servers are stopped and restarted one after the other. It also allows quick rollbacks in case of unforeseen problems. When realizing such a distributed setup, a key design issue is how the cache operates across the different nodes. The cache is usually placed between the application and the database, and a distributed design must consider how distributed updates and reads will affect the consistency of the data and overall performance of the system.

Review of Latest Evolution in Java Cache Technologies

One of the biggest constraints in Java is that the increase of allocated heap memory is tightly coupled to the time that the garbage collector running in all Java applications needs to collect unused objects. During its clean-up phase all other threads are paused by the Java Virtual Machine (JVM) and this can lead to unpredictable application response times. Even though in theory a 64bit Java virtual machine is open to allocate all available memory, it is not recommended to use more than four GB of garbage-collected heap. The main issue is not the object de-

ISBN 978-3-95450-139-7

letion itself, but the fact that the garbage collector compacts the heap afterwards. Large blocks of data allocated as buffers may then be moved around, and this can be very time-consuming. Tuning the garbage collector is possible, but non-trivial and is limited. Java however offers a backdoor to more memory use through direct *ByteBuffer*, which is nowadays widely used by standard caching technologies. Already introduced with Java 1.4, direct buffers allow creating and working with memory outside the garbage-collected heap [2]. Ehcache [3], one of the most used open-source caching libraries is leveraging this to offer Java applications the potential of making use of the entire heap space. Hazelcast [4] is another caching product making use of this off-heap possibility.

Next to this use of off-heap memory, another major evolution in Java caching products has been the increase in distributed cache technologies. These tend to fall into two categories, either making use of a separate "caching" server, or connecting applications in a peer-to-peer architecture. Ehcache for instance, in combination with Terracotta's shared memory server technology called BigMemory [5], can be spread across multiple nodes. Although some of the Ehcache and BigMemory features require a license, the free edition already offers interesting functionality for projects that do not intend to share hundreds of gigabytes among their nodes. A remarkable one is the Ehcache search functionality, allowing SQL-like searches across the distributed memory, realized through the separate BigMemory server that serves as central cache store to which all nodes connect. The server can have any number of hot standby mirror nodes forming a so-called stripe. If the memory becomes too large, more stripes can be added horizontally to maintain the required performance. This server-centric design tends to suit high-update scenarios, since a received update will be communicated only to the server and does not get sent to all other nodes immediately.

Due to C^{2}MON's high-update requirement and also because of the fact that Ehcache is compliant with the upcoming JSR 107 in-memory caching standard [6], the framework was chosen as C^{2}MON's central cache, ensuring that the server remains compatible with other caching solutions. Finally, when looking at the caching options available, careful consideration must be put into understanding the data consistency and durability settings available. For Ehcache with Terracotta for instance, the cache can be set to remain at all times consistent and persisted to disk, which is the configuration usually used for C^{2}MON.

C^{2}MON ARCHITECTURE OVERVIEW

C^{2}MON started out of a refactoring of the Technical Infrastructure Monitoring (TIM) server [7]. It was based on the idea of reusing TIM's core components for creating a new modular and open monitoring platform. C^{2}MON uses standard three-tier architecture with the Java Messaging (JMS) framework ActiveMQ as middleware between its tiers [8]. The three layers are the Data Acquisition (DAQ) layer, the C^{2}MON server layer and the C^{2}MON Client API.

Data Acquisition Layer

The DAQ layer offers drivers to acquire data from a variety of sources, for instance the standard industry communication protocol OPC (DCOM and UA) as well as data retrieval from PLC, Oracle database or other CERN specific protocols. Each DAQ process runs a common DAQ core, which manages the lifecycle and communication with the C^{2}MON server tier. It also applies various filters to the data stream in order to reduce the volume and improve the quality of the data.

Business Layer

The C^{2}MON server runs as a standalone Spring application. No additional dependency to any enterprise server technology such as J2EE is needed. The heart of the system is based on a (optionally distributed) cache that is shielded by a wrapper interface. The modular design simplifies the writing of regression tests or simulation of certain parts of the system. The server comprises a core part that every C^{2}MON instance has to include and a set of optional modules. The core architecture contains the in-memory cache and provides a basic set of functionalities, that is:

- Communication and lifecycle management of all DAQ processes.
- Configuration of the individual DAQ modules.
- The initial load of the in-memory cache from the backup database, and recovery options in case of a crash.
- Evaluation of alarms and business rules, which can be defined for each data sensor.

Server modules can subscribe to all kinds of core events for further treatment, such as sensor or alarm updates, or the status change of an equipment or DAQ process.

Client Layer

Finally, the C^{2}MON client API is the Java interface for interacting with C^{2}MON. The API allows subscription to any kind of monitored data and securely triggers execution of pre-configured commands. It also checks the connection and health state of the server layer. An appreciated feature is the possibility to retrieve historical data from the database as well as to switch all subscribed data into playback mode and replay it on an application as if it was occurring in real time. The communication between server and client layer uses simple JSON messages broadcast through JMS topics.

C^{2}MON DEPLOYMENT SCENARIOS

Before describing the different C^{2}MON deployment scenarios, it is important to say a word about data partitioning in general.

ISBN 978-3-95450-139-7

C²MON Distributed Architectures: General Considerations

Often in distributed architectures, the performance of the system will depend highly on whether the data can be partitioned in some logical way, so that different nodes across the cluster can concentrate on managing a subset of the overall data set. This strategy is often referred to as data sharding. Indeed, successful partitioning will then limit the network traffic, with clients going directly to the node where the freshest version of the required data is stored. In fact, if your data can be neatly split, it may be worth running two entirely separate instances of C²MON, particularly if the clients can also be split along these lines. What's more, a future project is to enable the C²MON client API to acquire data from multiple C²MON-based services, so that all C²MON-acquired data can be seamlessly merged in common synoptic displays. In the end, the main constraint on partitioning the data will be the use of data in common business rules, which require the concerned data points to be served by the same C²MON cluster (for data synchronization reasons).

Similarly, before choosing a given architecture for performance reasons, careful thought must be put into whether all the data propagated to the C²MON servers is required at that level. As described in the past chapter, C²MON provides a sophisticated data acquisition layer, providing a variety of data filtering mechanisms letting only essential data changes through to the actual server/cache.

In particular two hardware aspects are worth emphasizing for the importance they will have on the final performance and behavior of the system, namely memory and network. Of course, to benefit from the in-memory cache technologies described previously in this paper, it is important that the hardware on which the cache components run, both nodes and central cache server, dispose of sufficient memory to contain the entire cache. Similarly, it is quite clear that to optimize performance the cache nodes and servers should ideally be connected via a fast and reliable network. Finally, it is worth noting here that the most complicated part in tuning a distributed system will often be understanding the consequences of various hardware and network failures on the cluster, both in terms of service availability and data consistency. There is no general rule here, and obtaining the ideal configuration will require real-world tests to tune such things as cluster-monitoring timeouts, and fallback strategies when the network splits a cluster in two, for instance. Ehcache provide some good guidance on their website for addressing these issues correctly.

Deployment 1: Simple and Fast

The first scenario we describe corresponds to an existing C²MON deployment at CERN, that of the DIAMON monitoring service [9]. This service is used to monitor thousands of computers used to run the CERN accelerator complex. Both operating system variables and application-specific metrics are imported into the C²MON platform, using some custom data acquisition modules. In terms of non-functional requirements, reliability and high performance are the most important. The availability is less critical, and the service can live with short interruptions for software upgrades. To meet this scenario, the simplest C²MON deployment is sufficient, providing for easier management of the software components. In this architecture, a single server is deployed, with no redundancy or rolling updates possible (see Fig. 1 for a depiction of this architecture). On the other hand, the in-memory single-node cache immediately provides a very good performance and the setup remains very simple to configure and manage.

This deployment can be adapted in two important ways. Firstly, as described in the previous chapter, the cache can be kept off-heap using the Ehcache BigMemory technology. This means that this architecture can still be used for very large monitoring scenarios without compromising on the C²MON performance (currently a license is required if the cache size exceeds 32 GB). Secondly, the single ActiveMQ broker can also be upgraded to a cluster if the message load requires this, as is done in the DIAMON deployment.

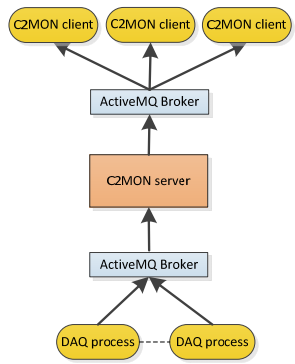

Figure 1: Single server setup.

Deployment 2: Available and Maintainable

The architecture presented in Fig. 2 combines high availability and maintainability with sufficient performance for most needs. It corresponds to the deployment used for TIM that benefits since the migration to C²MON from easier software upgrades and an increase the overall availability of the service. As compared to scenario 1, this deployment has added redundancy on the server level, allowing rolling updates and instant failover in case of a single server failure. With the separate Terracotta caching process, this architecture choice also allows for a larger number of configured data points out-of-the-box, since the Terracotta process will automatically overflow to disk if required. On the other hand, the performance will in general be slower with a lower throughput, since the transfer of cache changes from the nodes to Terracotta is transacted (including the disk cache persistence).

As in scenario 1, this distributed cache can be kept off-heap if it gets too large and performance needs to be maintained. This can be done both for the C²MON nodes and the Terracotta central server. However, this will re-

ISBN 978-3-95450-139-7

quire a BigMemory license, at least according to the licensing policy at the time of writing.

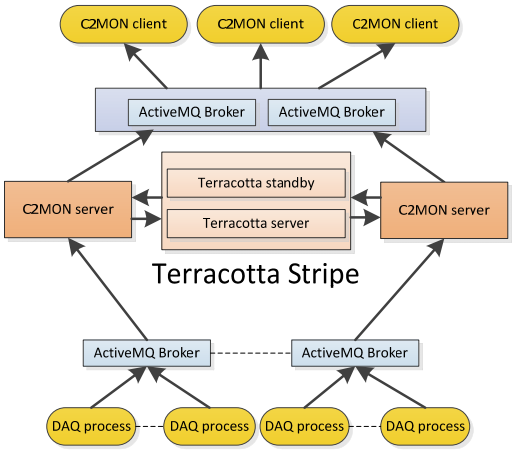

Figure 2: High availability architecture.

Deployment 3: All Bells and Whistles

This is the ultimate scenario with the most stringent requirements, with an architecture intended to maximize performance and availability of the service. Of course, the downside is that the service becomes both more complicated to deploy and manage, since more hardware and processes are now involved. Also, the cache setup now requires a license, the fee currently depending on the number of C²MON servers running. Before choosing this option, careful consideration should be given to the points detailed in the first paragraph of this section, as it may be possible to achieve sufficient performance using the setup in scenario 2.

The first step when setting up this architecture is to optimally partition the data: large data sets can normally be broken down in some way, with minimal dependencies between the partitions. Let us now detail the different layers in turn.

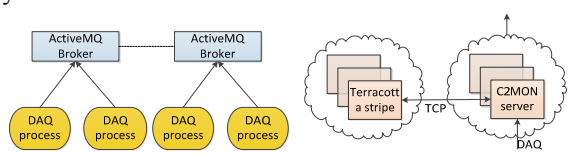

Figure 3: Data grouping within C²MON cluster.

On the DAQ layer, a single or small group of DAQs is dedicated to each data partition. Enough DAQ groups should be added to handle the incoming data load. Each group of DAQs then connects to a group of ActiveMQ brokers dedicated to its data fragment (see Fig. 3). These form part of the wider ActiveMQ cluster that handles DAQ to server communication. The data is then forwarded to a group of C²MON servers, which similarly form part of the wider C²MON cluster. It is important to understand here that all the C²MON nodes form a single cluster, but that subgroups of this cluster are dedicated to a single data partition of the shared memory (see Fig. 3). This means that while they will usually only deal with incoming data from this partition, they remain capable of dealing with any data coming from the DAQ layer. Equally importantly, they can handle any client requests, which transit through a separate ActiveMQ cluster and can end up on any C²MON node. The grouping of the broker and server nodes according to data partitions is simply a strategy to help the distributed cache optimize the data distribution, since C²MON nodes "specialize" in certain data points. It assumes client requests are not the main performance constraint, since they do not necessarily land on a node responsible for the requested data. Ehcache is configured as a Terracotta server array, providing for horizontal scaling of the Terracotta server to which the C²MON nodes connect. As in the other scenarios, the cache can be kept off-heap if it gets too large. Notice the illustration only displays a couple of horizontal data partitions, but all levels can be horizontally scaled along the dotted lines.

CONCLUSION

Redundancy and simplicity are key concepts for realizing high-availability services, but not trivial to achieve. The biggest challenge is to limit the consequences of disruptive incidents, such as hardware, network or database failures. Over the last decade the implementation of highly available, clustered applications in Java has become much simpler. One of the reasons is the increased offer of sophisticated third party in-memory distributed caching products. Another important factor is the use of simpler frameworks and design patterns with clear interfaces that avoid tight dependencies to third party libraries

Choosing Ehcache with Terracotta BigMemory as distributed in-memory framework proved to be the right decision for the C²MON project. It provides sufficient flexibility for realizing all the different C²MON use cases presented in the previous section, which is emphatically demonstrated through CERN's two critical monitoring projects TIM and DIAMON [10].

REFERENCES

[1] M. Bräger, M. Brightwell, A. Lang, A. Suwalska, "A customizable platform for high-availability monitoring, control and data distribution at CERN", ICALEPCS'11, 2011, p.418ff
[2] Keith Gregory, "Byte Buffers and Non-Heap Memory", http://www.kdgregory.com/?page=java.byteBuffer
[3] Ehcache Homepage, http://ehcache.org
[4] Hazlecast Hompage, http://www.hazelcast.com
[5] Terracotta Homepage, http://terracotta.org
[6] JSR 107: JCACHE - Java Temporary Caching API, http://jcp.org/en/jsr/detail?id=107
[7] J. Stowisek, A. Suwalska, T. Riesco, "Technical Infrastructure Montoring at CERN", EPAC'06, p.233ff
[8] ActiveMQ Homepage, http://activemq.apache.org
[9] W. Buczak, M. Buttner, F. Ehm, P. Jurcso, M. Mitev, "DIAMON2 – Improved monitoring of CERN's accelerator controls infrastructure", ICLAEPCS'13, 2013
[10] C²MON Homepage, http://cern.ch/c2mon

ISBN 978-3-95450-139-7