!CHAOS HISTORICAL ENGINE

M. Mara, A. Paoletti, INFN-AC, Frascati, Italy
C. Bisegni, G. Di Pirro, L.G. Foggetta, G. Mazzitelli, A. Stecchi, INFN-LNF, Frascati, Italy
L. Catani, INFN-Roma2, Rome, Italy

Abstract

!CHAOS is an INFN project aimed at creating the communication framework and the services needed for the definition of a new control system standard, mainly addressed to large experimental apparatus and particle accelerators. !CHAOS is under test at DAFNE accelerator and has been developed to overcome the strong requirements throughput of new accelerators, like superB. One of the main components of the framework is the historical engine (HST Engine), a cloud-like environment optimized for the fast storage of large amount of data produced by the control system's devices and services (I/O channels, alerts, commands, events, etc.), each with its own storage and aging rule. The HST subsystem is designed to be highly customizable, such to adapt to any desirable data storage technologies, database architecture, indexing strategy and fully scalable in each part. The architecture of HST Engine and the results of preliminary tests for the evaluation of performance are presented.

THE TECHNOLOGY BEHIND THE HST ENGINE

The !CHAOS framework has been designed giving a sight to the software technology emerging from the development of the new Internet services, non-relational databases (NRDB) and distributed caching system (DCS). Both allow a high level of horizontal scaling allowing the insertion and retrieval of the data as fast as possible, trying to saturate all the available bandwidth an all the network connections of the subsystem.

While the NRDB logics and techniques are used to achieve the indexes management and fast data retrieval, the DCS instead is used to achieve "live data sharing", a scalable system to share the real-time device data. This software provides in-memory key/value storage and permits fast multiple accesses on the same key by many requesters. For this reason, the usage of this cache is one-write (the device that update its data on cache) and multiple-reader (the user interface or algorithm that need to fetch device output channels). This permits to remove the load caused from multiple access reading from the host that controls the device.

These two software technologies represent the core components in the design of the new control system named !CHAOS [1] (i.e. "not" CHAOS, as the logical negation in many programming languages, where CHAOS stands for Control system based on Highly Abstract Open Structure) [2, 3, 4].

In !CHAOS architecture, the Front End Controllers (FEC) push acquired channels and alarm data into both live and history data cloud (DC), which means that data collection mechanism is inherently included in the !CHAOS communication. The user interfaces or controlling algorithms can obtain hardware data, from the DC, issuing a "get" command or registering into the push data services of the DC. The use of "get" command permits to regulate the effective refresh rate needed by every node, the push service instead, forwards the data at the same rate as it is pushed into the DC.

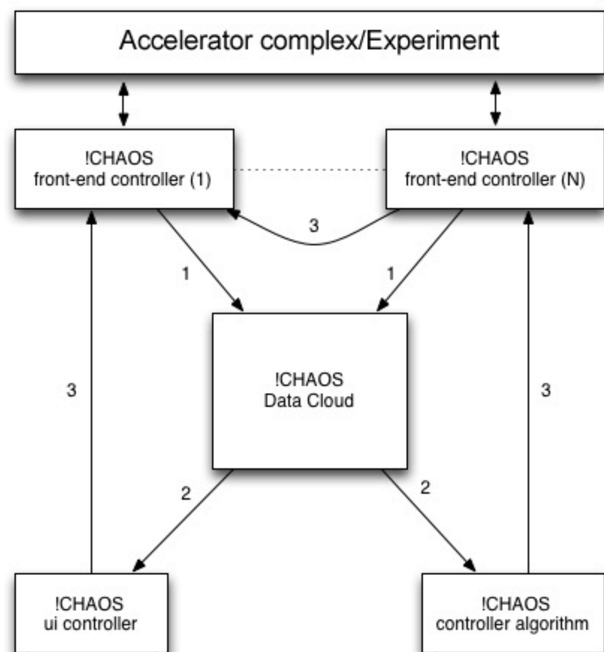

Figure 1: How the !CHAOS Data Cloud interacts with the other components.

The data payload sent by the front-end controller into the DC is serialized using the BSON spec[5]. By construction, the channel and alert description are considered using the maximum dimension that their values can have. Also the positions of the channel (within the payload) are observed between different push operations. In this way the BSON payload never changes its structure, but only the channels and alerts value changes. Within these considerations we can decide to update the entire payload (if needed), or his sub part. This permits to scale down the data bandwidth needed to

Copyright © 2013 by JACoW — cc Creative Commons Attribution 3.0 (CC-BY-3.0)

update the current device state into the DC.

All others parameters of !CHAOS infrastructure, such as data refresh rates, as well as other meta-data, configurations, commands and data syntax and semantic etc. are managed by the Meta-Data Server (MDS). It permits to manage where a FECs needs to push data and help the other nodes to find FECs IP for sending RPC commands. All kind of nodes are managed by this central repository. The MDS has an important role also regarding the access to the Storage subsystem: it stores the logic and data used by the FECs and user interfaces to identify the appropriate access point to the Cloud. Thanks to this information a first level load balancing is already done before accessing the Data Cloud.

!CHAOS HISTORICAL ENGINE

In !CHAOS the data storage is provided by the service called History (HST) Engine. This conceptual design will allow an innovative storage system for a Distributed Control System, giving !CHAOS an important technology advantage against other equivalent most popular standard for controls. The main ideas at the base of the data acquisition process are the following: a distributed file system is used to store data produced by machine operations while a KVDB manages the indexes structure (nowadays candidates are Hadoop [7] and MongoDB [8] respectively). These tools have been chosen thanks to their diffusion in the scientific community for solving similar problems and the abundance of use cases to which learn from. The functionalities of the !CHAOS HST Engine are allocated to three dedicated components, or nodes, namely the !CHAOS Query Language (CQL) Proxy, the Indexer and the Storage Manager.

Figure 3 shows the data flow and the role of the before mentioned nodes in data writing (red) and reading/querying operations (green). Grey lines are used to indicate internal actions and data flow.

A Control Unit (CU), FEC !CHAOS process, starts the writing process by sending a dataset to the CQL Proxy indicated from the MDS, as its primary HST server (1). The CQL Proxy has the role of an access point to the storage subsystem, hiding all the complexity of data storing to the user. Upon receiving the package, the proxy interprets the CQL command and starts the data flow inside the Storage infrastructure. To ensure multi write capabilities to the entire system a Cache mechanism has been chosen: all the packets received are pushed inside a common area (2) structured as follows. For each CQL Proxy a logical file is allocated inside the distributed FS, once a determined size has been reached the file is marked as "closed" signaling the storage manager to move the packets from the actual cache file to the producer device file inside the file system (3). This method allows to improve the writing capability of the system by increasing the number of proxies writing at the same time inside the cache. It is necessary to introduce appropriate policies allowing packets reordering inside the file system. Every logical file related to a data producer is time ordered at any time, the storage system for each device maintains the information containing the latest packet stored in the file system, if the cache subsystem receives an "old" packet, not yet stored, adequate procedures are implemented to ensure the time contiguity inside the device logical files. When moving the packets inside the device logical files, the system updates, for each device, the latest useful timestamp effectively stored in the file system, giving the data consumers a semi real time information about the packets readable from the Cloud.

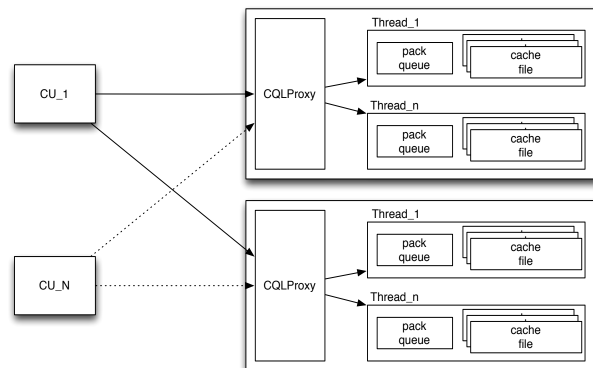

Figure 2: The !CHAOS multi-write idea.

Figure 2 shows the idea behind the multi-write caching design, it permits to minimize the collision of the data packs into a single proxy, an algorithm based on the default push rate of every CU, performs the load balancing between data producers and Proxies. Every proxy can dynamically allocate multiple threads and for each thread a lock free queue is created. Here are inserted the data packets obtained by the proxies, every thread is responsible for consuming the data inside its queue by moving them inside the proxy associated cache file.

The cache garbage collection mechanism is obtained thanks to this feature, all the proxy files containing packets already available on the Cloud, and marked as closed, are removed from the file system, ensuring the resources reuse as soon as they become available again. Both the cache files and the device logical files are stored inside a distributed file system, at the moment Hadoop seems the best option. Hadoop is a distributed file system that provides high throughput access to data, it automatically replicates the data in the other servers of the cluster (grey lines) ensuring a full redundancy of the system. Once the data has been stored, the CQL Proxy informs the pool of Indexer nodes about the new entry (4) and the first available Indexer appends the task to its queue. When processing the entry, the Indexer first reads the packet (i.e. the dataset) from the first available Hadoop node (5), analyzes it and, according to the indexing rules, updates the corresponding indexes on the MongoDB (6). The default indexing strategy will be by

Copyright © 2013 by JACoW — cc Creative Commons Attribution 3.0 (CC-BY-3.0)

chronological order, i.e. based on the timestamp and bunch/packet number within timestamp intervals. The indexing procedure allows a faster retrieve of the stored data by providing two different Indexes, the Time Machine Index (TMI) and the Value Based Index (VBI). The TMI is the default index in !CHAOS, because all the acquired data can be placed in a continuous time line in which the acquisition time can act as a primary key for all the data fetched by a single device. The TMI is intended as multilevel and allows choosing the desired granularity for every query forwarded by the proxies. The second index, based on the values of the data stored, will be available only on demand, allowing to retrieve particular data patterns like spikes or results of a chosen indexing function.

Queries to HST are triggered from client applications by sending a CQL command (1) to the proxy with the highest priority in its list. The proxy node decodes the request and passes it to the first available Indexer (2) that in turn, by querying the Indexes DB, receives the coordinates of data packets (3) satisfying the query's conditions (like data packets within a certain time interval, or data packets that reflects a particular pattern) and sends them to the CQL Proxy (4). The packages are then collected (5) from various FS Servers and sent (6) to the client.

It is worth mentioning that since responses to queries are asynchronous and tasks can be distributed among different proxies, data packets resulting from a query can be provided to the client application also by other CQL Proxies different from the one that originally received the request, allowing to improve the responsiveness of the whole subsystem.

Thanks to the chosen implementation, it is possible to increase the overall performance of the system by scaling different components. A faster writing mechanism for the devices can be ensured by increasing the number of proxies writing in a parallel way inside the cache. The transfer process between cache and device logical files can be increased by incrementing the number of managers checking the packets acquired; the indexing procedure can be improved by increasing the number of indexer nodes.

The tests on this conceptual design will be executed by creating a software simulation of the FECs writing mechanism, each step, from the production of data to the write inside the file system will be analyzed scaling the numbers of threads, workers, and indexer nodes. The data obtained will show if the multiple redundancy and parallelization of the components allow to improve the performances of the system even if the data produced by the FECs is increased. The effective results will come in few months thanks to the collaboration with student of various Italian universities.

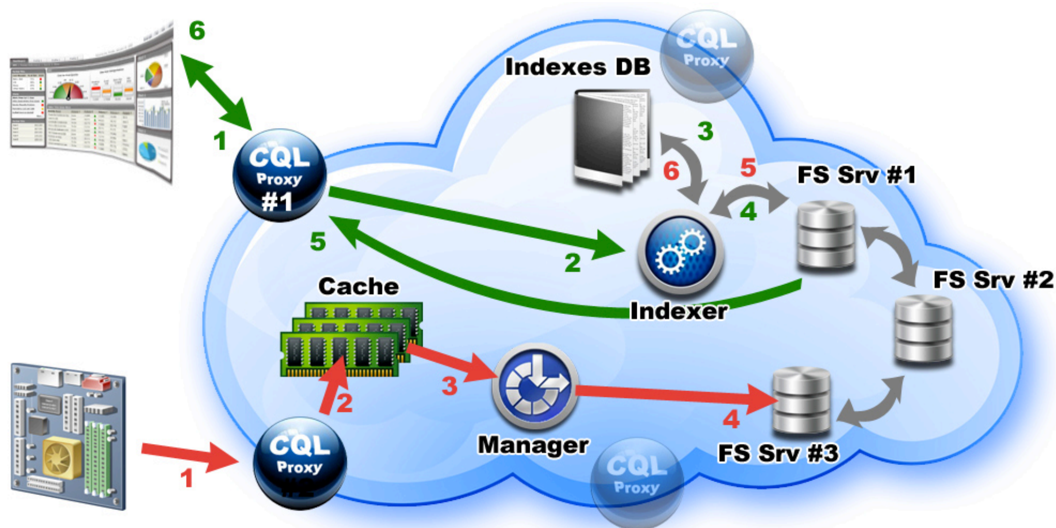

Figure 3: The !CHAOS Storage Infrastructure event list.

REFERENCES

[1] http://chaos.infn.it
[2] L. Catani *et.al.*, "Introducing a new paradigm for accelerators and large experimental apparatus control systems", Phys. Rev. ST Accel. Beams 15, 112804 (2012).
[3] G. Mazzitelli *et.al.*, "High Performance Web Applications for Particle Accelerator Control Systems", Proceedings of IPAC2011, San Sebastian, Spain, pp.2322-2324, http://www.JACoW.org
[4] L. Catani *et.al.*, "Exploring a New Paradigm for Accelerators and Large Experimental Apparatus Control Systems", Proceedings of ICALEPCS2011, Grenoble, France, http://www.JACoW.org
[5] http://bsonspec.org
[6] http://msgpack.org
[7] http://hadoop.apache.org
[8] http://www.mongodb.org

Copyright © 2013 by JACoW — cc Creative Commons Attribution 3.0 (CC-BY-3.0)