DEVELOPMENT OF AN INNOVATIVE STORAGE MANAGER FOR A DISTRIBUTED CONTROL SYSTEM

M. Mara 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 the definition of a new control system standard for large experimental apparatus and particle accelerators based on innovative communication framework and control services concepts. !CHAOS has been developed to address the challenging requirements in terms of data throughput of the new accelerators under study at INFN. One of the main components of the !CHAOS 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, or indexing strategy and fully scalable in each part. The architecture of HST Engine and the results of preliminary tests for the evaluation of its performance are presented in this paper.

THE !CHAOS FRAMEWORK

The !CHAOS framework has been designed after an in-depth evaluation of the new software technologies for data transfer and data storage emerging from the development of high-performance Internet services, such as the non-relational databases (NRDB) and the distributed caching system (DCS). Both are designed for a high degree of horizontal scaling that allows the insertion and retrieval of the data at the highest possible throughput, limited only by the saturation of either the available bandwidth or the network connections of the subsystem.

While the NRDB logics and techniques are used to implement the indexes management and the fast data retrieval the DCS is used to provide the "live data sharing", a scalable service for sharing the real-time device data. This software provides in-memory key/value storage and permits fast accesses to the same key/value by many concurrent clients. This caching layer avoids overloading the front-end controller with multiple reading accesses from clients that need to fetch data of a device.

These two software technologies represent the core components in the design of the new control system named !CHAOS [1, 2, 3].

In the !CHAOS architecture, the Front End Controllers (FEC) push acquired I/O channels and alarms data into both live and history data cloud (DC), which means that data collection mechanism is inherently included in the !CHAOS communication layer. User interface applications, feedbacks or measurement algorithms can receive hardware data from the DC by issuing a "get" command or by registering to 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 from the FEC.

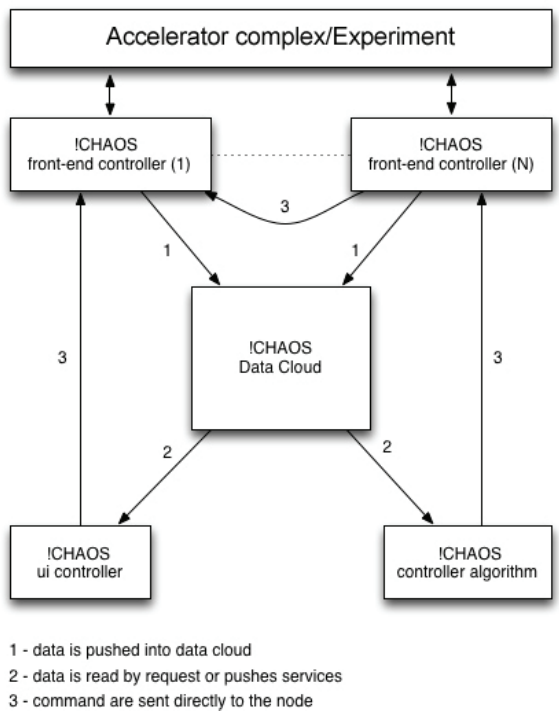

Figure 1: Data exchange between components by means of !CHAOS Data Cloud.

The data payload sent by the front-end controllers to the DC is serialized according to the BSON specifications [4]. By construction, the formatting structure of the serialized data, its length and the offset of each value within the string do not change if only the data values are changed. By taking advantage of these features, we can decide to update either the entire payload (if needed) or just a part of it. This permits to scale down the bandwidth requirements for updating, at a given refresh rate, the device state into the DC.

All others parameters of !CHAOS services and controlled devices such as data refresh rates, as well as

Copyright © 2014 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-139-7

other meta-data, configurations, commands, data syntax and semantic etc. are managed by the Meta-Data Server (MDS). It manages FECs registration at start-up and later provides directory services to clients that need, for instance, to locate the DCS server for pushing its data or a FEC's IP for sending RPC commands. Both static and dynamic configuration data of all !CHAOS services and nodes are managed by this central repository. The MDS has an important role also in the management of 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 achieved before accessing the Data Cloud.

!CHAOS STORAGE SUBSYSTEM

In !CHAOS the data storage is provided by the service called History (HST) Engine. Its design (timed data oriented), will give !CHAOS an important technology advantage, in terms of performance, scalability and flexibility, against the most popular DAQ standards 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. At the moment, candidate technologies for these services are respectively Hadoop [5] and MongoDB [6] that we choose because of the large users community and the abundance of use cases that we used as references. The functionalities of the !CHAOS HST Engine are allocated into three dedicated components, or nodes, namely the !CHAOS Query Language (CQL) Proxy, the Indexer and the Storage Manager.

This document focuses on the flow of the data in the storage operation. Figure 2 shows the role of CQL Proxy and Storage Manager in the data acquisition and organization. As soon as the data has been organized it will be more easily indexed for be made available. Every CQL proxy works on its personal space on the cache area and the Collector processes (a part of storage manager), instead, are working on all of the cache areas.

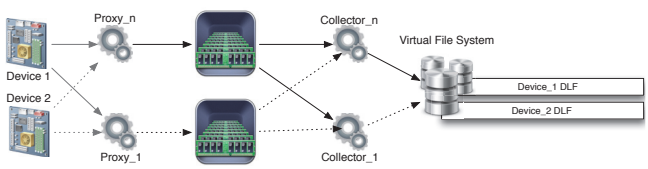

Figure 2: The !CHAOS multi-write concept.

Staging the Data

A Front End Controller, i.e. a !CHAOS Control Unit (CU), starts the writing process by pushing a dataset to one of the storage subsystem CQL Proxies. The CU already knows which CQL Proxy is better to contact because during the initialization of the system (or the boot of the device) an algorithm tries to allocate at the best the resources of the whole Storage system. By checking the medium data-pack size of the devices, and its pushing rate, the registered CQL Proxies are "allocated" to the devices in order to load balance the network infrastructure and the total computing power. This technique allows implementing an inherited hot swap in case of proxy down time because the users of the proxies (Data producers and consumers) already know the full list of accessible access point if their main one goes down. The CQL Proxy plays the role of an access point to the storage subsystem, hiding all the complexity of data storage procedures to the user. Upon receiving the package, e.g. a serialized device's dataset, the caching related logic inside the access point starts the data flow inside the Storage infrastructure. To ensure multi-write capabilities to the entire system we implemented a Cache layer such that all the packets received from clients (e.g. FEC) are stored by proxies in a common area (2) structured as the following. For each CQL Proxy a logical path is created in the distributed FS. To improve the performance of the system each proxy can allocate a pool of threads with the only task of getting the packets received by the proxy and start the allocation inside the file system. Each thread will fetch the data packets from the proxy regardless the device that produced them, starting to write them in files inside the path associated to the Proxy. The data pack wrote by a thread is stored into a private cache chunk (CC). Once a CC is no more valid (in terms of space or time elapsed), it will be available for the next phase described below.

Moving the Data to Device Logical File (DLF)

This phase consists of reading a closed cache chunk, read every packet and write it in the Logical File for the corresponding device. A specific process, contained into the Storage Manager, called "Collector" achieves this task. Every Collector process has a pool of thread and each one of these scans the cache directory to find a cache chunk to be processed. The selected cache chunk will be a closed one by the action of the proxy and not yet selected by other thread or other collector processes.

The selected chunk will be read pack by pack. In every pack, it will be found the "device id" that has generated the data then it will be moved to the device logical file. Current collector thread has its own LogicalFileWriter, allowing having one logical file chunk for thread. Only one thread writes on a single Logical File Chunk, the same technic used by the proxies. When all the packets into the cache chunk are read, the chunk is deleted.

This method allows improving the input performance of the system by increasing the number of proxies writing concurrently to the cache. Clearly, it is necessary to introduce appropriate strategies to allow packets reordering inside the file system such that each logical file, associated to a data producer, is chronologically ordered at any time.

ISBN 978-3-95450-139-7

Copyright © 2014 CC-BY-3.0 and by the respective authors

Chunk Fusion of the Logical File (LF)

After the "Moving" phase (Figure 3), every chunk of the logical file will be ordered in time (every data packet has timestamp >= than the previous one). Anyway two or more LF chunks can be overlapped in time (OLFC – Overlapped Logical File Chunk, a side effect of the fast moving phase). These overlapped chunks are reorganized by another process called "Fuser" (also contained into the Storage Manager). Every DP of every chunk are "fused" in a unique chunk that contains DP of every OLFC time ordered. If the cache subsystem, in a fixed point in time, receives an "old" packet, the fuser ensures the proper chronological order of packets inside the device's logical files, by applying another merge operation on the right chunks of the Logical File. After that system updates, for each device, the timestamp of the newest packet effectively stored in the file system, providing the data consumers with quasi-real time information about the packets effectively stored inside the Cloud.

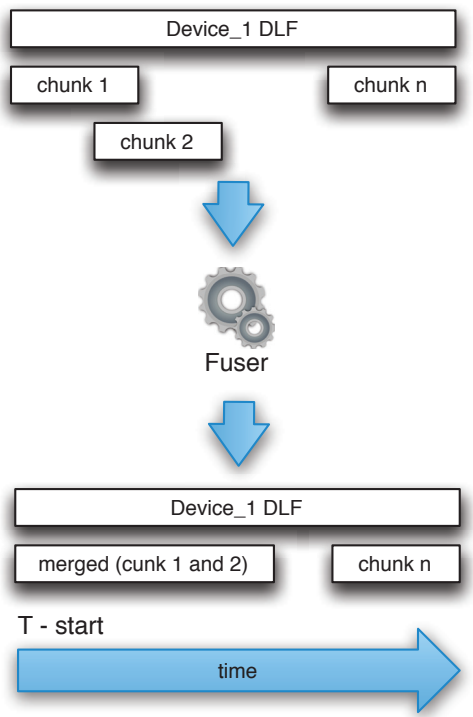

Figure 3: The "Chunk Fusion" logic.

Both the cache chunk and the device logical files are stored inside a distributed file system. Hadoop, our current best candidate, is a distributed file system that provides high throughput access to data, by automatically replicating it in the other servers of the cluster 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 written chunk and the first available Indexer appends the task to its queue. When processing the chunk, the Indexer first reads the packet (i.e. the dataset), analyzes it and, according to the indexing rules, updates the corresponding indexes. The default indexing strategy will be by 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 stored data is ordered according to a continuous timeline such that the timestamp is the primary key for all the data fetched by a single device. The TMI is intended as multilevel such to allow 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 the retrieval of particular data patterns.

Storage test

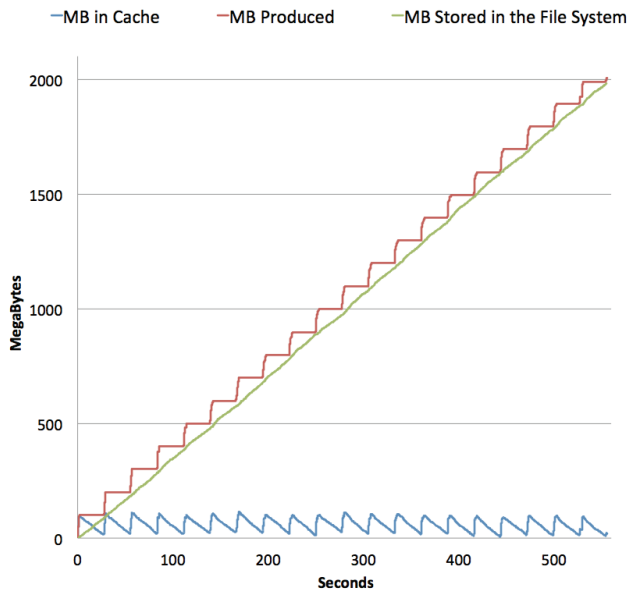

Figure 4: The output of the staging and moving tests, showing the MBs produced, in cache and stored in the FS.

It's worth stressing that the solution we have just described allows increasing any of the system's performance independently by scaling selectively its components. A faster and higher data throughput from front-end controllers, for instance, can be achieved by increasing the number of proxies writing concurrently to the cache. On the other hand the data throughput between caches and device's logical files can be increased by growing the number of queues' managers checking the packets acquired and the indexing procedure can be improved by increasing the number of indexer nodes.

The components described so far, related to the staging and moving mechanism, have been tested off-line by using a software simulation of these two phases. The fusing phase has been removed from the numeric tests because is not fundamental for the data acquisition process: it is used once the data is already safe on the file system. The tests have been run on a mid level Mac Pro

Copyright © 2014 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-139-7

with two 2,8Ghz Quad-Core Intel Xeon, 18GB of DDR2 RAM, and a SATA2 SSD hard disk. The graph shown here (fig. 4) is obtained by using two Producer processes simulating ten devices running with 50 threads each, and a single consumer process running on five threads. The average data produced by the simulated devices is 3,5 MB/s simulating 515 channels pushing data packets of 68 B at 100Hz. The test environment is like a worst-case scenario for this algorithm, because it cannot gain performance by a distributed file system and a multitude of proxy machines. In fact the data rates obtained can grow almost linearly by increasing the number of proxy machines and using a more appropriate file system. The graph in figure 4 shows the three fundamentals measurement in the caching system: the data produced by the devices (in red), the data actually in the cache files (in blue) and the data actually stored inside the device logical files (in green). More intensive tests will be run in the next months on the other parts of the storage system now under development.

ACKNOWLEDGMENTS

The work is partially supported by FP7 Research Infrastructures project AIDA, grant agreement no. 262025.

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] L. Foggetta *et.al.*, "Progresses on !CHAOS development", Proceedings of IPAC2012, New Orleans US, http://www.JACoW.org
[4] http://bsonspec.org
[5] http://hadoop.apache.org
[6] http://www.mongodb.org

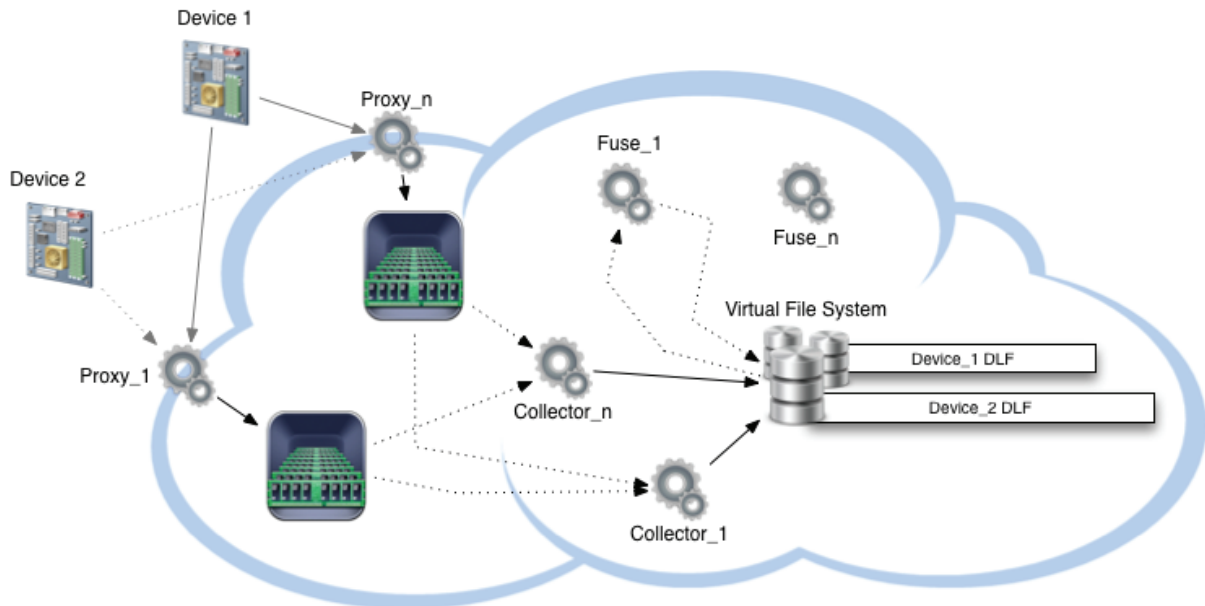

Figure 4: The !CHAOS Storage Infrastructure event list.

Copyright © 2014 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-139-7