!CHAOS: THE "CONTROL SERVER" FRAMEWORK FOR CONTROLS

L. Catani, INFN-Roma Tor Vergata, Roma, Italy
F. Antonucci, C. Bisegni, A. Capozzi, G. Di Pirro, L.G. Foggetta, F. Iesu,
N. Licheri, M. Mara, G. Mazzitelli, A. Stecchi, INFN-LNF, Frascati, Italy

Abstract

We report on the progress of !CHAOS [1], a framework for the development of control and data acquisition services for particle accelerators and large experimental apparatuses. !CHAOS introduces to the world of controls a new approach for designing and implementing communications and data distribution among control system's components and for providing the middle-layer services for a control system. Based on software technologies developed for high-performance Internet services !CHAOS offers, by using a centralized highly-scalable cloud-like design, all the services needed for controlling and managing a large infrastructure. It includes a number of peculiar features such as high abstraction of services, devices and data, easy and modular customization, extensive data caching for enhancing performances, integration of all services in a common framework. Since the !CHAOS conceptual design was presented two years ago the INFN group have been working on the implementations of services and components of the software framework. Most of them have been completed and tested for evaluating performance and reliability. Some services are already installed and operational in experimental facilities at LNF.

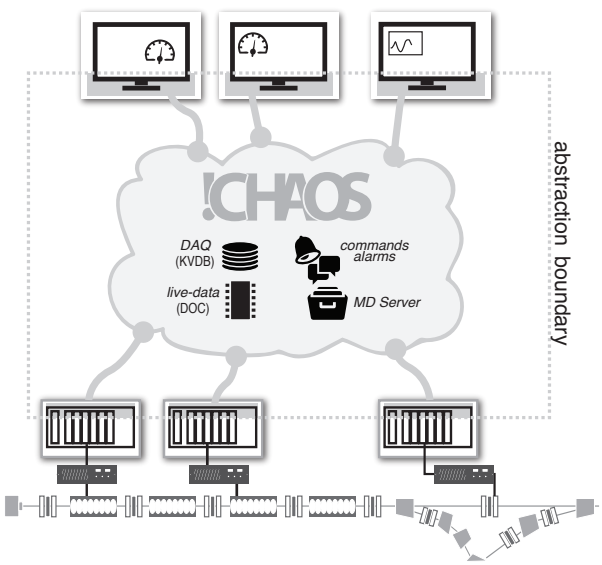

Figure 1: The !CHAOS "control server" model.

INTRODUCTION

One of the pillars of the !CHAOS framework is the high abstraction of services, devices and data such to obtain a high scalability of the system and an extreme flexibility in terms of its possible applications (see Fig. 1). Although particle accelerators and large experimental apparatuses have been always considered as the primary target, since the very beginning of the project we looked for other uses of !CHAOS with special attention to Cyber Physical Systems and, more in general, solutions for monitoring and managing devices and sensors over Local or Wide Area Networks that we already presented to some potential communities of users.

Recently, the Italian Ministry of Education, University and Research has approved a two-years project aimed at the development of a prototype of cloud-like monitoring services for multidisciplinary applications based on the !CHAOS framework.

At the same time the !CHAOS Group is continuing the development of framework's components, including an update of the design of the DAQ service, and the deployment of !CHAOS-based solutions at control systems of LNF's accelerators.

PROGRESS OF !CHAOS FRAMEWORK

Front-end

The refinement of the Control Unit design has brought to the definition of a CU as a "container" for a device's front-end controls. Beside the standard methods: INIT, START, STOP and DEINIT, CU's specialization is provided by the device's *dataset*, the *actions* implemented for it (e.g. commands and control loop) and the drivers for I/O components (see Fig. 2).

At startup (INIT) the CU receives from the Meta Data Server (MDS) the *dataset* of the accelerator's device (equipment, diagnostic, sub-systems, etc.) assigned to that unit and the related actions. The default action, i.e. RUN, implements the accelerator's device control loop; other actions implement, as usual, commands for modifying the working state of the device or for executing more complex procedures.

At START the CU begins looping on the active *actions* with only the RUN action tagged as active. When a new command is received, the CU appends it to the *priority queue* that is managed according to the commands' properties. If the action currently running is either *killable* or *stackable*, as RUN is, a new command from the *priority queue* can be moved to the actions loop and executed instead of the previous one or in sequence with it.

Simple *actions* are managed by handlers associated to *dataset*'s variables using C++ *accessors* to execute *set* or

ISBN 978-3-95450-139-7

get operations. If the variable is connected to an I/O channel then the correspondent *driver* is used to access the hardware.

In !CHAOS, I/O modules drivers are managed by CU-like components called *Driver Unit*. Similarly to CU they are specialized for a particular I/O module, or more in general a I/O service, by means of the module's *dataset* including specifications such us the IP address and port for a network device or the Controller address and Node number for CAN units etc., and by specific management software. *dataset* also includes unique identifiers for each I/O channel of the module (e.g. each input channel of an ADC module).

At startup the *Driver Unit* initialize the I/O module and starts populating the embedded key/value shared memory with default values of its I/O channels. The key is provided by the unique identifier assigned to the I/O channel; it is defined in the module's *dataset* received by the *Driver Unit*. After receiving the START signal, the *Driver Unit* begins looping on the I/O module acquisition procedures triggered either by machine timing signals or by or software.

It means the Control Unit and the Driver Unit will run, asynchronously and independently, their own tasks governed by their respective timing: the device's refresh rate and command execution the first, the machine's timing the second. If synchronization is needed, as is for the case of readout from a serial line I/O, dedicated variables shared among CUs and DUs can be used, similarly to a bus control line, for triggering an acquisition (by the CU) and for asserting the *data ready* signal (by the DU).

The solution that was just described allows to clearly, and conveniently, separate the development of the software implementing the logic of the controls (control loop, commands etc.) from the I/O modules software. In other words, it extends the !CHAOS abstraction concept down to the interface between the device's logic and the I/O driver's management.

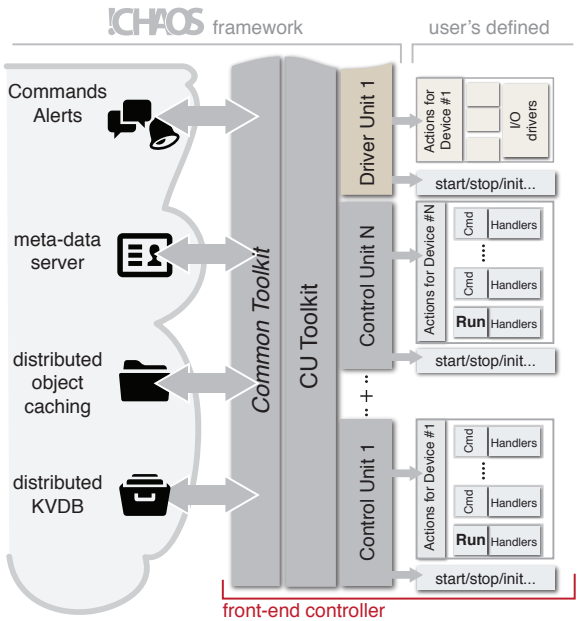

Figure 2: !CHAOS front-end and its components.

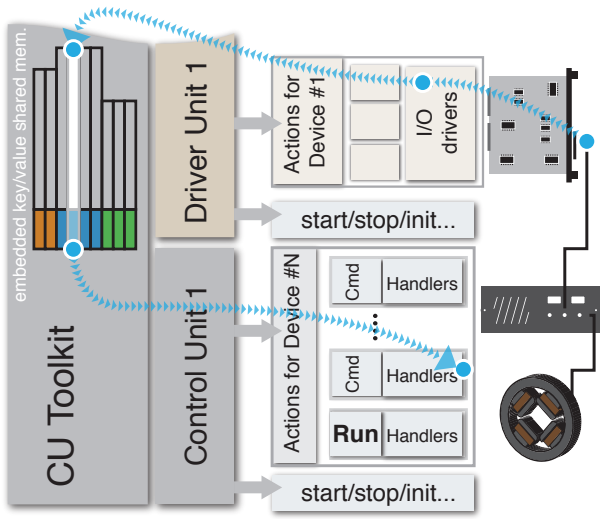

Figure 3: Role of the key/value shared memory to pass data between Driver Units and Control Units.

In fact, the unique identifier, defined for each I/O channel of any I/O module, is the only link between the user of the I/O data, e.g. a CU action by means of a get handler, and the producer of the I/O data, the *Driver Unit* managing the I/O module to which this particular I/O channel belongs. When a new accelerator's device is added to the control system, the programmer searches the database for the I/O modules used by this device and links the I/O variables of the *dataset* with the unique identifier associated to the channel it uses.

At the time of writing of this paper, this solution is yet to be fully implemented mainly because it strongly relays on services of the MDS that have not been completed. To achieve a fully functional CU for the pilot installations on Dafne Control System, the Driver Unit functionalities have been split into methods of the CU and, instead of the key/value shared memory, pointers to memory locations of I/O variables are passed between the handlers and the I/O drivers.

DAQ

Compared to both the strategies and the technologies implemented in the first prototypes, the DAQ has been subject to a deep redesign with the aim of creating a solution for the infrastructure and the *history* service that is able to offer the maximum scalability in terms of speed, throughput, capacity and, especially, the depth of database.

It doesn't mean that the solution based on distributed KVDB was dropped. Instead, exploiting the abstraction of services that is guaranteed at all levels in !CHAOS, the former is now one of the possible options within the set of data storage technologies that are supported by a more flexible and performant design.

ISBN 978-3-95450-139-7

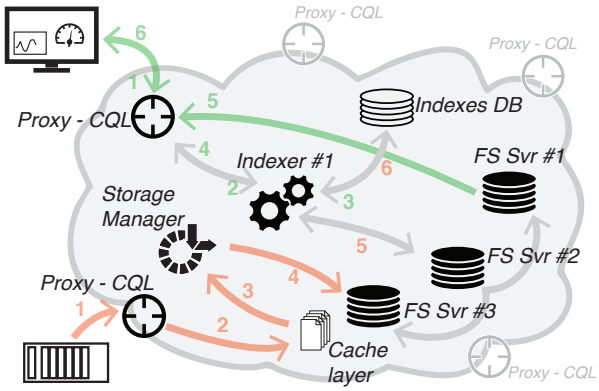

Figure 4: !CHAOS History Engine and its components.

In !CHAOS the DAQ, i.e. the machine data acquisition system, is provided by the service we call History (HST) Engine. A distributed network File System (FS) is used to store data produced by machine operations while a KVDB manages the set of indexes that are produced by the indexing rules; candidates are Hadoop [2] and MongoDB [3] respectively. The functionalities of !CHAOS HST Engine are provided by three dedicated components namely, the CQL Proxy (where CQL stands for CHAOS Query Language), the Indexer and the Storage Manager.

Figure 4 shows the data flow and the role of the before before mentioned components in data writing (red) and reading/querying operations (green). Grey lines are used to indicate HST commands and internal data flow.

A CU starts the writing process by sending a *dataset* to the CQL Proxy indicated, by the MDS, as its primary HST server (1). Upon receipt of the package, the proxy interprets the CQL command and starts the process of writing the data into the file system.

To ensure multi-write capabilities we implemented a Cache layer such that all the packets received from clients are stored by proxies in a common area structured as the following.

For each CQL Proxy a logical path is created in the distributed FS (see Fig. 5). At the same time, each proxy launches a pool of threads having the only task of allocating (2) the packets received by the proxy into files within the logical path associated to the Proxy, regardless from the device that sent the packet. Once a predefined maximum size, in terms of either disk space or time period is reached, the file is marked as “closed" and another one is opened for writing.

In parallel the Storage Manager looks for closed files (3) and empties them by distributing the packets to the FS file they belong since for each device there is a dedicated logical file (4). A logical file is made of physical files, or "chunks", ordered in such a way to be seen by clients as a single file of chronologically ordered chunks.

Ordering is done by another process called âĂIJFuserâĂİ. Since each chunk is a chronologically ordered sequence of *datasets*, the Fuser checks if there is an over-

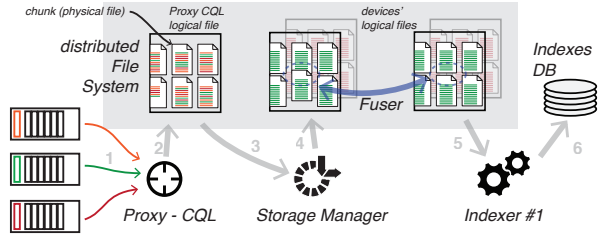

Figure 5: The !CHAOS History Engine storaging and indexing sequence.

lapping between the oldest and the newest timestamp in two adjacent chunks and, in that case, reorders them before they are definitely stored in the file system. Hadoop automatically replicates the files in the other servers of the FS (grey lines).

Next the CQL Proxy informs the pool of Indexer nodes about the new entry 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 Indexes DB (6). The default indexing strategy will be by chronological order, i.e. based on the timestamp and bunch/packet number within timestamp intervals.

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 positions of data packets (3) satisfying the query’s conditions (e.g. all data packets within a certain time interval) 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’s worth mentioning that since responses to queries are asynchronous and tasks can be distributed among different nodes, data packets resulting from a query can be provided to the client application also by CQL Proxies different from the one that originally received the request.

Other Developments

The detailed implementation of the !CHAOS framework revealed possible improvements to the basic design and optimizations that we already started to investigate. These tasks offered opportunities of research arguments for graduate or doctoral thesis that have been undertaken in the last months.

The first work investigated the possibility of introducing a second level caching at the client layer to minimize the data fetching from the main key/value database. We studied the possibility to introduce a circular buffer for sharing data among client applications. As an example, consider a GUI with a graph showing the last N values of a certain variable. When the GUI starts to fetch data from the KVDB, the User Interface Toolkit underneath allocates a lock-free circular buffer and provides the graph’s applica-

ISBN 978-3-95450-139-7

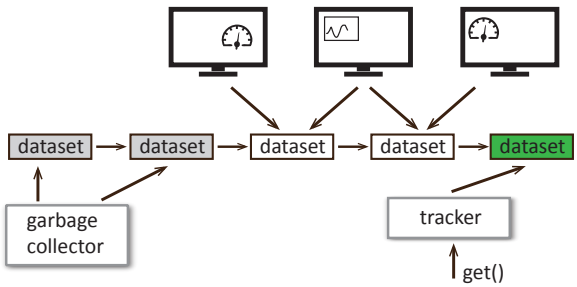

Figure 6: The UI Library circular buffer for caching the data fetched from the key/value database.

tions with the pointer to the buffer (see Fig. 6). The buffer, with size equal to the depth of the graph, is updated by the *tracker* taking into account the refresh rate of the device and the sampling of the graph.

If another GUI panel or application needs the same data or a part of it, e.g. the most recent value of the buffer or a downsampling, the User Interface Toolkit instead of opening another stream of data uses the buffered data to feed the second client. Alternatively, if the second client needs a longer buffer or higher sampling frequency the buffer is resized accordingly and the original "owner" of the data buffer will extract data from it.

A second thesis addressed the optimization of *datasets*' distribution, i.e. of the Control Units' data pushing processes, to the pool of Proxy CQL for refreshing the main cache memory. *dataset* assignment (i.e. distribution of keys) to memcached servers is managed by the MDS that provides Control Units and User Interface Toolkit with the list of primary and secondary servers for each *dataset*. In the case of failure of a primary server both writing (typically CUs) and reading clients automatically switch to the first secondary server in the list and then to the following one in case the latter also fails.

An algorithm has been developed to create the list of primary and secondary servers for each CU that equally distribute the load among the Proxies. To achieve the optimal balancing, the algorithm distributes *datasets* by taking into account the product (*dataset* size) $\times$ (nominal refresh rate) such that bytes per seconds written to each server will be uniformly distributed. Also secondary keys will be chosen to preserve a reasonable balancing in case of failure of one or more servers in the cluster.

The results of the algorithm and the failover scheme have been tested in a multi-CU and multi-Proxy system. The result confirmed the possibility to manage Proxies failure with limited impact on performance and data loss because switching to secondary server never exceeded 2 ms. In other words the failover scheme can guarantee data integrity for refresh rates up to 50 Hz.

Another work aimed at developing a tool for benchmarking and fine-tuning the communication framework based on the central caching of *datasets* by using the key/value database.

By using the theory of controls we simulated on a CU an unstable physical process. A feedback algorithm, running on a client application at the consoles level, is used to stabilize a given working point. The simulated physical process was designed in such a way to be sensitive to relevant indicators of performance of the communication framework. In other words if the communication framework is not able to support the feedback algorithm with sufficient transmission speed and throughput the process will lose its stability.

A number of CUs running this physical process can be activated and controlled by an equivalent number of client applications. By increasing the number of such processes and/or modifying their stability conditions it is possible to get the measure of performance of the communication framework.

Finally, another task has dealt with the development of an embedded version of *memcached* [4] for implementing local shared memories. The work consisted in isolating the code for the key/value storage by removing either dependencies from libraries such as *libmemcached* and *libevent* or resource consuming services and features in order to optimize its performance. A typical use would be, for instance, the management of shared memory for exchanging data between Driver Units and Control Units, as it was previously described.

CONCLUSION

The !CHAOS Group is progressing in completing the services of the !CHAOS Framework and, consequently, in the definition of details of its design.

The innovative approach at the basis of the !CHAOS framework requires a number of well established control system's components to be deeply redesigned. Nevertheless, prototypes of the main services are already under test and demonstrated the validity of design in terms of performances and robustness. Future plans foresee a smooth migration of the LNF accelerators' controls to !CHAOS-based solutions.

In parallel the !CHAOS Group is involved in the development of a multidisciplinary application of the !CHAOS framework aimed at prototyping a cloud-like monitoring service for devices and sensors distributed over a Wide Area Network.

REFERENCES

[1] 10.1103/PhysRevSTAB.15.112804.

[2] http://hadoop.apache.org.

[3] http://www.mongodb.org.

[4] http://memcached.org.

ISBN 978-3-95450-139-7