INTRODUCING THE !CHAOS CONTROL SYSTEM FRAMEWORK

L. Catani, F. Zani, INFN-Roma Tor Vergata, Roma, Italy
C. Bisegni, D. Di Giovenale, G. Di Pirro, L. Foggetta, M. Mara, G. Mazzitelli, A. Stecchi
INFN-LNF, Frascati, Italy

Abstract

The analysis of most recent developments on high-performance software technologies suggests that new a design of distributed control systems (DCS) for particle accelerators and large experimental apparatuses can profit from solutions borrowed from cutting-edge Internet services. To fully profit from this new technologies the DCS model should be reconsidered, thus leading to the definition of a new paradigm. In this paper we present the conceptual design of a new control system for a particle accelerator and associated machine data acquisition system (DAQ), based on a synergic combination of a non-relational key/value database (KVDB) and network distributed object caching (DOC). The use of these technologies, to implement continuous data archiving and data distribution between components respectively, brought about the definition of a new control system concept offering a number of interesting features such as a high level of abstraction of services and components and their integration in a framework that can be seen as a comprehensive control services provider for GUI applications, front-end controllers, measurement and feedback procedures etc. The work is under development by a collaboration of INFN-LNF and INFN-Roma Tor Vergata with growing contributions from other academic and industrial partners.

THE !CHAOS FRAMEWORK

A typical example of software technology emerging from developments of Internet services is the class of non-relational databases known as key/value database. They offer an alternative to relational databases (RDMS) that is having a growing success and interest among developers of web services due to of their high throughput, scalability and flexibility. Another example are the distributed memory object caching systems. They provide in-memory key/value store for small chunks of frequently requested sets of information in order to both respond faster to requests and to distribute the load of the main server to a scalable cluster of cache servers.

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

In particular, the KVDB is used by DAQ for managing what we call *history data*, while the DOC implements the service for distributing *live data* from the front-end controllers to clients, thus replacing the client/server communication.

Datasets that need to be updated are identically pushed,

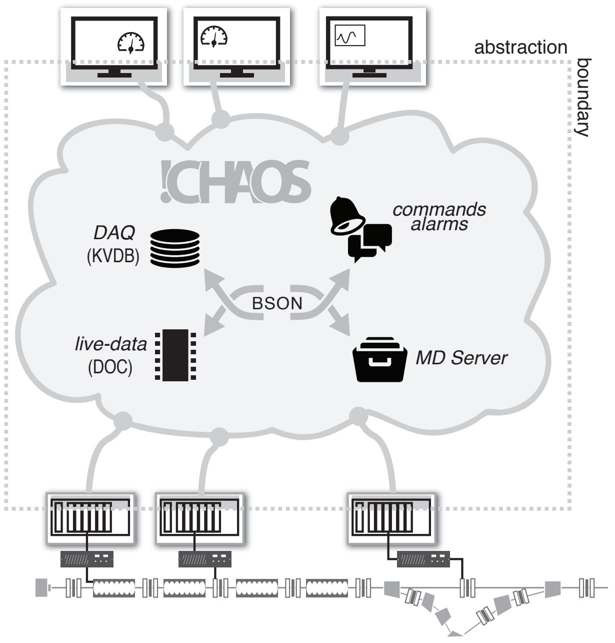

Figure 1: The "control service provider" model and the !CHAOS framework abstraction boundaries.

by front-end controllers, to both DOC and KVDB servers by issuing *set* commands. It means that data collection mechanism for DAQ is inherently included in the !CHAOS communication framework because both *live* and *history* data are pushed by the data source (the front-end controllers) to similarly distributed caching and storage systems. Moreover, since both DOC and KVDB use key/value data storage, formatting and serialization of datasets can be done once for both.

It is important noting that both the client applications and the front-end controllers are simple clients of the distributed object caching and DAQ. In particular, provided the DOC has an object container for each dataset of the DCS, defined by its unique key, a GUI client simply sends to the !CHAOS DOC service a *get* request for the object identified by that particular key, i.e. the dataset describing the associated device. On the other side the controller responsible for that device updates the correspondent dataset, according to the push rate defined for it, by issuing *set* commands to the DOC.

Data refresh rates, as well as other meta-data such as global parameters, CU configuration, commands and data syntax and semantic etc. will be managed by the meta-data server (MDS). The Meta Data Server will be also the central authority for !CHAOS components. It will keep track, for instance, of the Control Units instantiations. As su-

ISBN 978-3-95450-124-3

pervisor of their initialization, it will manage, at start up, the registration of CUs' services and datasets providing them with a systemwide unique reference to be properly addressed by client applications.

It is worth mentioning that in !CHAOS the DOC layer is not operated as an object caching in a strict sense since the distributed memory is not populated after clients' requests. Instead, each dataset is by default stored in the DOC and continuously updated by the front-end controllers. Nevertheless the datasets transfer from front-end to clients can still profit from the high performance of the distributed caching systems that, in addition, prevents front-end controllers from overload originated by multiple clients' requests [4].

Another benefit of the !CHAOS design is that the front-end controllers don't need to run servers to provide data to clients since they themselves are clients of the data distribution and storage services. That improves their robustness and portability.

A fundamental property of both DOCs and KVDBs is their intrinsic scalability that allows distributing a single service over several computers. Moreover, dynamical keys re-distribution allows automatic failover by redirecting to other servers the load of a failed one.

The data-pushing strategy allows to further extend the abstraction boundary at the front-end. The Controller's functionalities can be simplified and standardized by introducing the *Control Unit* (see next paragraph), a manager and a supervisor of the software modules implementing the device's specific control procedures.

In addition, abstraction of services will be implemented throughout (Fig. 1). Access to live or history data, for instance, will be provided by !CHAOS APIs wrapping the service specific APIs in such a way that client's access to services, and even internal communications, will not be affected by the modification or replacement of any core component.

Serializations of datasets and of information passed between components (e.g. command's parameters, result of queries, etc.) will further improve the abstraction of services by using a binary string as the common format for the methods' payload [5].

The commands dispatching and the events notification services complement the communication and interconnection between !CHAOS components. A cross-language RPC-like software (i.e. *msgpack* [6]), included in the !CHAOS libraries, will be used by client applications for sending commands to front-end controllers.

In conclusion !CHAOS is a scalable control system framework providing, at a high level of abstraction, all the services needed for communication, data archiving, timing, etc.; GUI applications and front-end controllers access the framework services and expand its functionalities.

CONTROL UNITS

Figure 2 shows the logical structure of the software running in a front-end controller. The Control Unit (CU), the

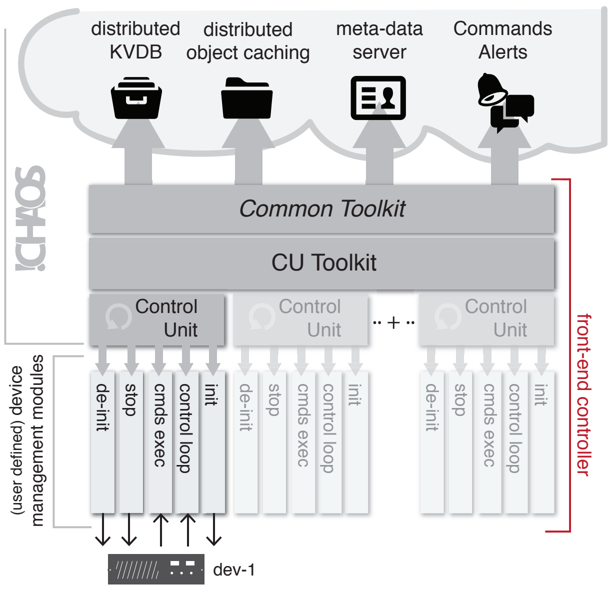

Figure 2: !CHAOS components for the front-end controllers.

CU Toolkit and the included Common Toolkit are components of the !CHAOS framework while the device management modules (DMM) are software modules that complement the !CHAOS framework functionalities by providing the interface to the device. The development of these components is expected either as a contribution, or as a responsibility, of the device experts. They will simply focus their work on the development of control loops, commands execution etc. while all the other operations will be delegated to the !CHAOS Framework.

One or more instances of CU can run simultaneously, although completely independent, in a front-end controller. Each CU should be dedicated to a particular device or a family of devices and specialized for that particular component by means of appropriated device management modules. The latter is a set of routines implementing the device's specific functionalities grouped into five general modules: *initialization*, *de-initialization*, *stop*, *control loop* and *commands execution*.

When a command issued by a client application is received by the CU, the *command execution* module will be invoked for executing it.

The command is delivered to the CU Toolkit running the *command server* for all the CUs managed by that particular controller. The CU Toolkit, by analyzing command's *domain*, identifies the CU to which it is targeted and appends it to the correspondent command's queue.

If the application issuing the command requires a direct read-back from the CU this can be provided by returning the command's results to the client's call. Alternatively, since also the UI Toolkit will host a *msgpack* server, the client application could be notified, yet asynchronously, on the results of the command's execution by a message delivered from the CU. The latter, actually, is the preferred solution.

ISBN 978-3-95450-124-3

Upon receipt of the command the CU verifies that the method *alias* indicated in the command's header is available and it can be executed (i.e. there is no other blocking method pending) and then launches the *commands execution* module. The rest of the serialized information containing the instructions for the action to be taken is passed as-it-is to the command's execution module.

The implementation of separated threads assures that requested periodicity of dataset refreshing is preserved even during any commands' execution. In addition the serialization of the command's descriptor (i.e. the command's header, method name, parameters etc.) passed to the execution module allows a common interface for all the methods to be implemented.

During the command execution, if needed, the refresh rate of the device can be set, at least temporary, to an higher value providing the operator and the history data archiving system with a more detailed description of the attribute evolution. Modification of parameters like the data refresh rate are superintend by the Meta Data Server. All components concerned with this change will receive notification by means of the events notification system.

HST ENGINE

In !CHAOS the DAQ, i.e. the machine data acquisition system, is provided by the service we call History (HST) Engine. A distributed file system is used to store data produced by machine operations while a KVDB manages the indexes structure; candidates are Hadoop [7] and MongoDB [8] respectively. The functionalities of !CHAOS HST Engine are allocated to three dedicated components, or nodes, namely the CQL Proxy (where CQL stands for CHAOS Query Language), the Indexer and the Maintainer.

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

A CU starts the writing process by sending a dataset to the CQL Proxy indicated, from the MDS, as its primary HST server (1). Upon receipt of the package, the proxy interprets the CQL command and writes the data into the file system (2). Hadoop automatically replicates the data in the other servers of the cluster (grey lines). Then the CQL Proxy informs the pool of Indexer nodes about the new entry (3) 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 (4), analyzes it and, according to the indexing rules, updates the corresponding indexes on the MongoDB (5). 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

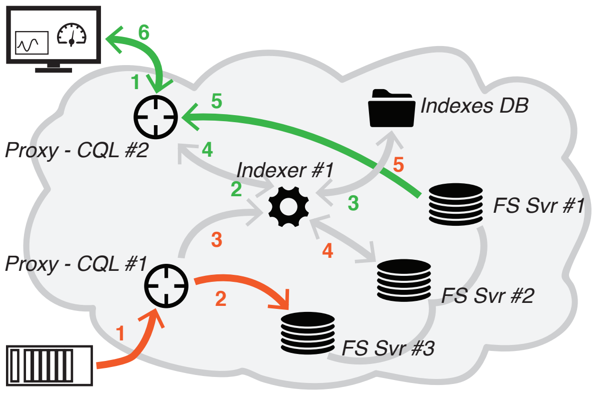

Figure 3: !CHAOS History Engine and its components.

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.

CONCLUSION

The design of the !CHAOS Framework is approaching its final stage. The whole architecture has been completed in details and prototypes of the main services are already under test. Other components and concepts not presented in this paper, namely Execution Unit, I/O Unit, Notification Service etc., have also been developed. Porting of !CHAOS libraries to various platforms, including ARM based boards, has been successfully completed. Moreover some !CHAOS components have been already adopted by the DAFNE and SPARC control systems at INFN-LNF and successfully operated since several months.

REFERENCES

[1] http://chaos.infn.it

[2] 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

[3] 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

[4] http://memcached.org

[5] http://bsonspec.org

[6] http://msgpack.org

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

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

ISBN 978-3-95450-124-3