SYSTEM RELATION MANAGEMENT AND STATUS TRACKING FOR CERN ACCELERATOR SYSTEMS

M. Audrain, D. Csikos, K. Fuchsberger, J.C. Garnier, A.A. Gorzawski,
G. Horanyi, J. Suchowski, P.C. Turcu, M. Zerlauth, CERN, Geneva, Switzerland

Abstract

The Large Hadron Collider (LHC) at CERN requires many systems to work together closely to allow reliable operation and at the same time ensure that the required protection systems function correctly when operating with the large energies stored in the magnet system and particle beams. Some examples of systems are magnets, power converters and the LHC quench protection system as well as higher level systems like Java applications or server processes. All of these systems have numerous and varied kinds of links (dependencies) between each other. The knowledge about the different dependencies are available from different sources like layout databases, Java imports, proprietary files etc. Retrieving consistent information is difficult due to the lack of a unified approach to collecting the relevant data. This paper describes a new approach to establish a central server instance, which allows collecting this information and providing it to different clients used during commissioning and operation of the accelerator. Furthermore, it explains future visions for such a system, which includes additional layers for distributing system information like operational status, issues or faults.

INTRODUCTION

The LHC is made of numerous software and hardware systems. Dipole magnets drive the beam into a circular path, while quadrupole magnets keep the beam focused. An electrical circuit contains a chain of magnets and is powered by a power converter. Magnets which store a very high energy are monitored by dedicated machine protection systems, like the Quench Protection System (QPS) controllers [1]. A QPS controller is intended to supervise up to four resistive voltage detectors. Each detector is related to a different physical circuit which is powered by an individual power converter.

The AccTesting [2] framework needs to know the relation between circuits and QPS controllers in order to schedule tests during LHC hardware commissioning campaigns [3] optimally. For instance, it must not schedule two tests in parallel on two circuits which are monitored by the same QPS controller, otherwise it would not be able to guarantee the diagnostic data consistency. Hence it needs to know, for a given circuit, all the circuits to which it is related, and through which QPS controller.

The AccTesting also needs to retrieve certain information about a system: The test history through all the previous hardware commissioning campaigns, test parameters, the current state of a system. All these information and their history are precious data to understand better the complex systems around the LHC.

The problem is that there is not a single data source to provide all the systems, their information and their relations. The software environment is made of multiple data sources which provide partial information. The AccTesting framework had to interface all the data sources providing partial information, using various different Application Programming Interfaces (API), in order to rebuild the full picture of the systems and their relations, as illustrated in Figure 1.

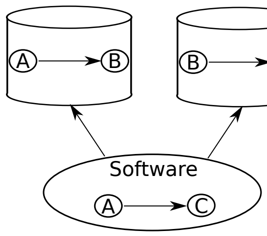

Figure 1: The software needs to know the relation from A to C, and it finds it by transitivity from two sources, first learning the relation from A to B and then from B to C.

The AccTesting is only an example and many applications have to address the same issue. A system relation management and status tracking framework was therefore designed. We will refer to it as the system framework. Its role is to retrieve all the records from all known data sources. It then provides a consistent API to allow any software to retrieve the systems, their relations and their status.

The first Section presents the overview of the framework and its usage. The second Section presents the architecture designed to handle systems, their relations and their attributes. The third Section presents the architecture of the system, how it serves clients and how it retrieves data. The fourth Section presents the investigations carried out at CERN to provide dynamic information, particularly for software components.

OVERVIEW

As explained before, the aim is to retrieve data from multiple data sources and to provide them in a consistent way to any user. The system framework provides a simple Java API so client applications can retrieve systems and their relations easily.

To feed data into the framework, a provider API was put

ISBN 978-3-95450-139-7

in place. Developers can design providers, based on any data source, and plug them into the system framework as illustrated by Figure 2. They can also define their own systems and relations.

Currently the system relation service is simply implemented in a library which can be embedded in client applications. In the future the plan is to provide a central and dependable service from which clients can access the information.

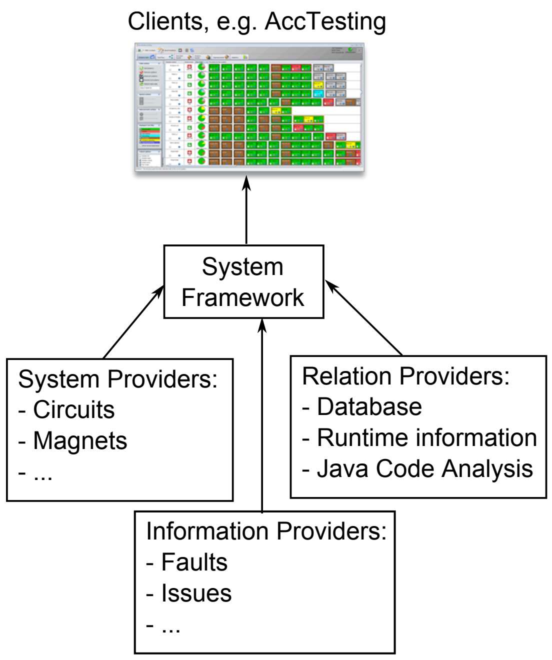
Clients, e.g. AccTesting
System Framework
System Providers: - Circuits - Magnets - ...
Relation Providers: - Database - Runtime information - Java Code Analysis
Information Providers: - Faults - Issues - ...

Figure 2: On top, the system framework serves multiple clients, like the AccTesting framework. At a lower level, the framework uses providers to retrieve information from multiple sources. The design was made to allow developers to add new providers easily.

SYSTEMS, RELATIONS, INFORMATION AND ATTRIBUTES

A System represents basically any part of the entire CERN accelerator complex. More generally *everything is a system*. It can be a magnet, a power converter, a device controller, a front-end-controller, a host or even a process. From a software point of view, this is an interface that can be implemented by developers who want to define their own systems, as shown in Figure 3. Most of the hardware systems which are used by the Machine Protection software are already implemented, and developers started to add other systems like hosts and processes.

The basic information about a system is the *SystemKey*, which defines a unique way to identify the system across multiple databases. This is the role of the implementation to make sure that the system key will be unique and will identify a same system across multiple data sources. For example for most of the hardware systems, we use the type of the system and the hash of its name, as names are unique within a given system type. However, a process key will have to use the process ID and the host IP address, as there might be multiple processes with the same name on the same host, and multiple processes with the same process ID on multiple hosts.

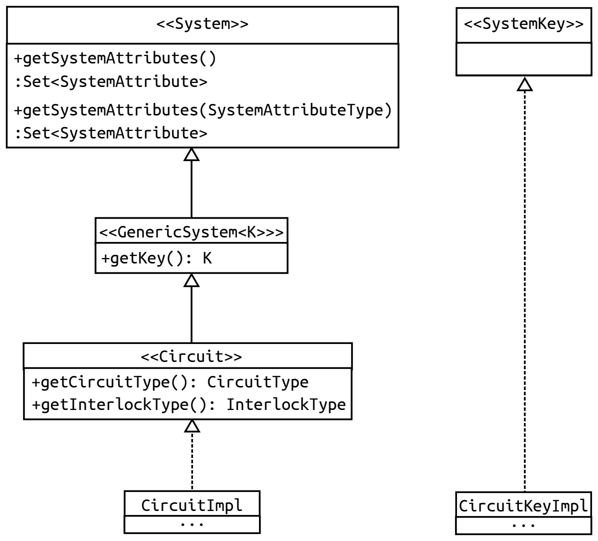
<<System>>
+getSystemAttributes() :Set<SystemAttribute>
+getSystemAttributes(SystemAttributeType) :Set<SystemAttribute>
<<SystemKey>>
<<GenericSystem<K>>>
+getKey(): K
<<Circuit>>
+getCircuitType(): CircuitType
+getInterlockType(): InterlockType
CircuitImpl
...
CircuitKeyImpl
...

Figure 3: The *System* interface allows users to define their own implementations, here is a Circuit. A System is composed of a *SystemKey* which is also implemented by the user. The *SystemKey* is what allows the application to identify the instances in a unique way in multiple databases.

System Attribute A system has attributes. They are characteristics of a system. Attributes are somehow static and almost never change. The perfect example in the LHC accelerator domain is the physical location of a magnet. For most of the LHC life time, the magnet will have the same location.

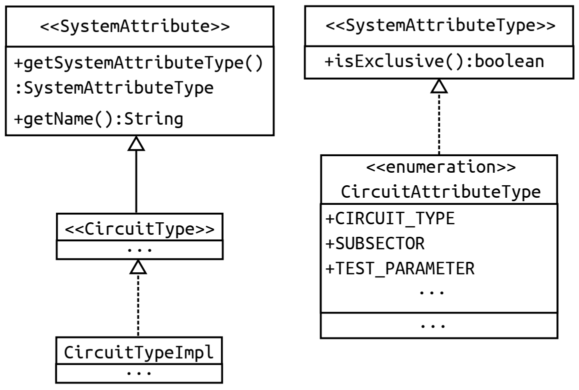

Figure 4: The System Attribute hierarchy is simple. The interface forces one to provide a name and a type to the attribute, then the implementations are free to add any information they want.

System Information defines the runtime information about a system, such as its actual status, parameters, issues or non conformities. A system information basically represents information about the system which does not characterize the system itself, unlike attributes. The number and type of information may vary depending on the system, if it is hardware or software.

A System Relation defines a relation between two systems. The interface *SystemRelation* offers two methods to retrieve the source and the target of a relation. This comes from the way data is stored in system and relation data sources, i.e. one looks up a target system from a source system. Relations can be extended to more complex definitions, like bidirectional, dependence, composition, connection, etc. Developers can implement their own relation to add them into the system framework. Figure 5 shows the hierarchy which implements a simple bidirectional relation.

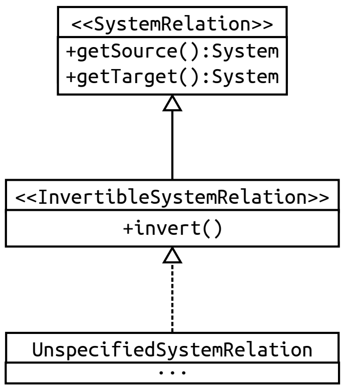

Figure 5: A *SystemRelation* is a simple interface which provides a way to retrieve the source and the target of the relation. A specific interface defines a simple bidirectional relation, its implementation will basically be able to reverse the source and the target of the relation.

COLLECTING, CACHING AND SERVING

The system framework retrieves systems, relations, information and attributes from multiple data sources in a unified and consistent way. Large scale applications need to access only one data source now, the system relation framework. The framework is layered as shown in Figure 6. Providers implement data collection, managers use these providers to retrieve data and cache them so that clients can retrieve them using the controllers.

Architecture From our experience, systems and relations were loaded from multiple text files, e.g. CSV, and from a couple of databases. Thus we decided to use loose coupling in order to define multiple providers. Providers can then be seen as plug-ins that any developer could implement. The managers are then able to use all the providers in a consistent way and get all the data. The managers will just handle objects through the interfaces presented in the class diagrams of Figures 3, 4, and 5. Meanwhile, the users know the types of their systems and relations. So they are able to retrieve data from the controllers with the correct type in a safe way thanks to Java Generics [4].

Collaborators already implemented their own modules, for instance to provide hosts and processes. In order to contribute, they have to implement their own systems, relations, information and attributes providers, and their own domain object if they are not already defined.

Behaviour At start-up, the framework loads all the systems, relations and attributes from the providers that are plugged in. It then stores the data in a multi-map [5] which represents an unlabelled directed graph: keys are mapped to a set of values. The multi-map is synchronized in order to serve multiple clients concurrently. Then every subsequent request from clients will hit the cache. A read-through caching still needs to be implemented to take into consideration data source updates occurring during the service uptime.

In order to integrate a new provider the service has to be redeployed and restarted. To overcome this, dynamic class loading is under investigation. Developers would be able to inject new classes directly into the server. The aim is to declare the server a critical service and therefore provide high availability.

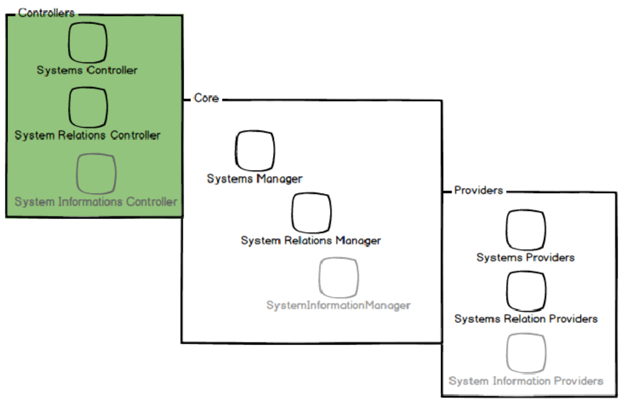

Figure 6: The system relation application is divided in 3 layers. Controllers allow users to get systems, relations or information. The core layer contains managers which perform caching. The lower layer consists of plug-in providers that are used by the managers to retrieve data. The System Information vertical slice is not implemented yet.

VOLATILE DATA COLLECTION

Investigations The information about volatile systems like processes will be based on data sources very different from databases. Running processes will be provided by DIAMON [6], while stopped processes will be provided by log files and Splunk [7]. The network hardware connections will be provided by the central IT database, while the

ISBN 978-3-95450-139-7

interprocess communication will be provided by the CERN Control Middleware [8]. JIRA issues [9], and more generally all the CERN accelerator development and build environment will be used to provide software system information and attributes.

In addition, volatile information calls for date and history management, pulling or pushing of data into the framework, and challenging caching strategies in order to ensure a high level of dependability and accurate answers to client requests. This is currently under investigation.

Complementary study This project is carried out in collaboration with the accelerator controls group, with the same requirements, but with a slightly different environment to start with. They use a complementary approach as their environment is mainly concerned by managing dynamic information, so they investigated further this problematic.

The alternative solution depends on code generation and metamodelling (describing a general and object-oriented entity-relationship (E-R) model, similar to to the interfaces presented in Section 1). The domain model of the system is defined with an instance model, which is loaded and processed by a Java code generator (implemented with Acceleo [10]). The models are defined with Eclipse Modelling Framework [11], but the generated code consists of Plain Old Java Objects (POJO), without any runtime dependencies. With the usage of a domain model and a code generator, not just the domain objects, but a strongly typed data access layer (DAL) and a remote access layer is generated fully automatically. The DAL uses a Neo4j [12] graph database back-end, which provides an efficient way for storing and accessing E-R models. The remote access layer is used for client-side applications (mainly graph-based data visualizations).

The data collection uses a similar loosely coupled data provider architecture. The providers will be executed by a central scheduler component. Three customisable options are available for defining the execution time for the providers:

- Timer-based triggering, where the provider can specify the frequency of execution.
- Event-based triggering, where the provider can specify, that if a given entity (or a set of entities) was added to the system (where entity is basically a *System* from Section 1), then the provider should be executed.
- On-demand triggers, where the clients can send request to the system to update one specific element in the database.

These options ensure that all kind of providers can be defined and added to the system, providing an efficient and easily extensible system.

CONCLUSION

Retrieving information from systems and their relations is a very common need at CERN. This paper describes the approach to a generic solution in the machine protection group, with the effort to bring a single solution fulfilling the needs of the whole CERN accelerator environment.

This solution can be seen as a proxy that interfaces all the possible system and relation data sources to provide a unified and consistent way to retrieve information from all these sources. It is able to store any information as it can be extended at will. Investigations are currently being performed in order to record the ephemeral information from the software world, to store a history in order to perform status tracking, and to be able to collect data dynamically.

This solution currently works embedded as a module in the applications which want to use it, and it will soon be released as a server as it becomes a more and more central system in machine protection software.

In parallel, the accelerator controls group investigated a different approach based on metamodelling, code generation and graph database persistence, in order to meet their strong requirement in handling ephemeral relations.

The next step will be to review the two existing solutions and converge to a single solution.

REFERENCES

[1] R.Denz et al., “Performance of the Protection System for Superconducting Circuits during LHC Operation”, TUPS071, proc. of IPAC 2011, San Sebastian, Spain.

[2] D.Anderson et al., “The AccTesting Framework: An Extensible Framework for Accelerator Commissioning and Systematic Testing”, THPPC071, proc. of ICALEPCS 2013, San Francisco, CA, USA.

[3] M. Solfaroli Camillocci et al., “Commissioning of the LHC Magnet Powering System in 2009”, MOPEB045, proc. of IPAC 2010, Kyoto, Japan.

[4] James Gosling, Bill Joy, Guy Steele, Gilad Bracha , “Java Language Specification, Third Edition”, Prentice Hall PTR 2005.

[5] https://code.google.com/p/guava-libraries/wiki/NewCollectionTypesExplained

[6] M.Buttner et al., “Diagnostic and Monitoring CERN Accelerator Controls Infrastructure : The DIAMON Project First Deployment in Operation”, proc. of ICALEPCS 2009, Kobe, Japan.

[7] http://www.splunk.com/

[8] A.Dworak et al., “The new CERN Controls Middleware”, proc. of CHEP 2012, New-York, NY, USA.

[9] https://www.atlassian.com/software/jira

[10] http://wiki.eclipse.org/Acceleo

[11] http://www.eclipse.org/modeling/emf/

[12] http://www.neo4j.org/