Wendelstein 7-X's CoDaStation

A modular application for scientific data acquisition

T. Bluhm[a,*], P. Heimann[b], Ch. Hennig[a], G. Kühner[a], H. Kroiss[b], J. Krom[a], H. Laqua[a], M. Lewerentz[a], J. Maier[b], H. Riemann[a], J. Schacht[a], A. Spring[a], A. Werner[a], M. Zilker[b]

[a] *Max-Planck-Institut für Plasmaphysik, Euratom Association, Teilinstitut Greifswald, 17491 Greifswald, Germany*

[b] *Max-Planck-Institut für Plasmaphysik, Euratom Association, 85748 Garching, Germany*

Abstract

The Control and Data Acquisition Station (short: CoDaStation) is a modular application for continuous data acquisition. It is based on the idea of abstract signal sources and signal sinks that are connected to a signal processing network. The structure of this network is defined by a configuration description and the behavior is controlled by a control system. Due to the well-defined interface definitions for signal processing units, configuration sources and control systems it is possible to use various implementations of these components and easily switch between them. This means less effort on environment changes which is especially important for experiment sites with operation times of several years and even decades. Furthermore, it provides room for tests using mock-up implementations and so an improved testability and stability.

The *CoDaStation* will be an integral part of the control and data acquisition system of Wendelstein 7-X. However, the modular design allows the integration into very different scientific environments.

1 Introduction

The major goal of Wendelstein 7-X is to demonstrate continuous plasma operation with experiment durations up to 30 minutes. Although only pulses up to a length of 40s are planned during the first operation phase all involved components have to be prepared for steady state operation right from the beginning. Amongst others, this implies challenging requirements for the data acquisition system [1, 2].

1.1 Non-functional Requirements

The expected overall data rate of the attached diagnostic systems will be about 15 Gbyte/s. Because of the continuous operation mode it is necessary to stream all acquired data directly to a central archive and monitoring system for immediate inspection and processing.

Furthermore, the impact of failure rates increases dramatically in the continuous operation case. After entire completion of Wendelstein 7-X there will be roughly 100 data acquisition systems that have to deliver data reliably for approximately 1000 seconds of plasma operation. In this scenario an apparently moderate failure rate of 1 failure per 10000s cumulative discharge duration already means that about 10 data acquisition systems will presumably fail during one experiment. Because this is unacceptable the Wendelstein 7-X CoDaC group aims to assure a high reliability of the data acquisition systems by providing extensive testing and supervision possibilities of the corresponding hardware and software.

Finally, Wendelstein 7-X plans to provide feedback control loops for sophisticated control scenarios during steady state experiments. The input data used for these control loops must be provided by the data acquisition system in a deterministic way. As a consequence it must

*Corresponding author, E-Mail: torsten.bluhm@ipp.mpg.de

be possible to provide data reliably in predictable intervals.

1.2 Functional Requirements

On full operation of Wendelstein 7-X a high number of different diagnostics must be integrated which partly use very different acquisition technologies. In order to be able to create a coherent view on the experiment it is necessary to abstract from the concrete technologies and map them to a small set of common interfaces. This does not only apply to the data acquisition functions but also to the integration into the central control system of Wendelstein 7-X. Every diagnostic component must be able to participate in the segment sequence control which means react to corresponding commands as well as provide state information.

The continuous operation mode also requires every acquired value to be addressable by an absolute timestamp. To make these timestamps comparable between different data acquisition systems they have to be synchronized to a central time system [3, 4].

1.3 CoDaStation History

In order to meet the described requirements Wendelstein 7-X provides a special software called Control and Data Acquisition Station (short: CoDaStation). This software has already been used for several years for different prototype installations like WEGA for example [5]. Because of the experiences made with these prototypes a complete redesign of the CoDaStation has been done. The following chapters will describe the basic architecture of this new version.[1]

2 Basic Concepts

The general idea of the *CoDaStation* is to combine several signal sources and signal sinks to a signal processing network. Signals are time-variable functions of data samples of different kind. In the context of the *CoDaStation* a data sample may be a scalar value (e.g. a voltage measurement) or a set of values that are closely related (e.g. the pixels of a video image).

The core of the *CoDaStation* software is a generic framework that solely provides the necessary infrastructure to acquire and process signals. Additionally, it defines a fixed interface that is used to extend the *CoDaStation* with actual business functions like retrieval of data samples from data acquisition devices, archiving of acquired data to persistent storage systems and provision of control and configuration interfaces.

3 Signal Processing

3.1 Buffers

Signals are processed inside the *CoDaStation* in the form of *Buffers*. *Buffers* are containers for the values of one or more signal chunks where a chunk is a finite number of consecutive samples. All signals contained in a buffer must have been acquired using a common sample clock. This means that samples with the same index inside a buffer are assumed to be acquired simultaneously.

3.2 Signal Providers and Consumers

Signal sources and sinks are represented by *Signal Providers* and *Signal Consumers*. *Signal Providers* produce *Buffers* and *Signal Consumers* take *Buffers* for further processing.

There are different scenarios where *Signal Providers* and *Signal Consumers* are used. Sources for samples of *Signal Providers* may be such as:

- any kind of sensor that converts analog signals to digital values (ADCs, cameras, ...)
- time measurement devices, counters, etc.
- network sockets that receive data samples via Ethernet from a remote source
- pieces of software that simulate or process samples

Examples for *Signal Consumers* are:

- digital to analog converters (DACs) of different types

[1] A few aspects of the architecture will be illustrated with diagrams. These diagrams use the FMC notation [6, 7].

- network sockets that are used to send data via Ethernet to remote sinks
- pieces of software that process samples
- pieces of software that store samples in a specific format

Signal Providers and *Signal Consumers* are basically data input and output interfaces for third party components that acquire or process signals.

3.3 Routers

Signal Providers are connected to *Signal Consumers* by so-called *Routers*. Similar to common network routers that route network packets from a sender to a receiver *Routers* retrieve *Buffers* from one *Signal Provider* and transfer them to one or more *Signal Consumers*. Unlike network routers the information about sender and receiver as well as the routing path is statically defined and not part of the transferred *Buffer*.

The transfer steps are executed in a loop as long as the acquisition task is active. The number of samples to be transferred in one cycle is a configurable parameter allowing the number of cycles per second to be customized. When no samples can be retrieved from the source the *Router* will block until a timeout is reached.

3.4 Routing Jobs

Routers may act synchronously or asynchronously. This is defined by *Routing Jobs*. A *Routing Job* is a task that executes one or more *Routers* synchronously. Multiple *Routing Jobs* are executed asynchronously (Fig. 1). The default behavior is to run each *Router* in a separate *Routing Job* so that all *Routers* will be executed asynchronously. This will usually result in the maximum performance. However, there are cases where the execution of *Routers* has to be more deterministic. For example the feedback control loops mentioned in chapter 1.1 require all *Routers* to be executed in a limited time period. In this case all *Routers* have to be run synchronously in one *Routing Job* each having a defined timeout.

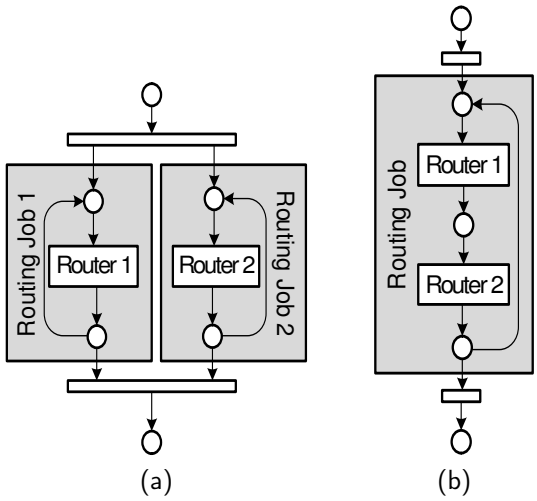

Figure 1: Asynchronous (a) vs. synchronous (b) control flow

3.5 Time Groups

As stated during discussion of the requirements all acquired samples have to be addressable by synchronized timestamps. Hence, the data acquisition system has to assure that the synchronized time information is attached to every acquired data signal. If that did not already happen outside of the *CoDaStation* (e.g. in customized devices) a separate time measurement must be added to the actual data acquisition. The time and data measurements must use a common sample clock and are combined to a common measurement by so called *Time Groups*.

Time Groups contain one *Signal Consumer* for the time measurement and one or more *Signal Consumers* for acquired data signals. Incoming *Buffers* from the different sources are checked for consistency (equal number of samples) and merged to a common *Buffer*. The merged *Buffers* are made available by a *Signal Provider*.

4 Control

4.1 Properties

The state of *CoDaStation* components is defined by *Properties*. *Properties* are sets of key-value pairs that

define the parameters of a specific component. The keys are always character sequences (which may be interpreted as numbers) whereas the values may be of any type. The component using the property values is responsible for checking the correct type and interpreting the content.

4.2 Controllables

Controllables retrieve *Properties* from the *CoDaStation* and translate them into component-specific actions. An action may be something like writing to a register of a device or execution of a software method. Furthermore, it provides status information (e.g. health status, fill level of internal buffers or similar). Like *Signal Providers* and *Signal Consumers Controllables* are input and output interfaces for the control specific part of third party components.

4.3 Station States

A *Station State* is basically an aggregation of *Properties* with references to corresponding *Controllables*. It is used to define the overall state of the *CoDaStation* at a given point in time. *Station States* do not have to be complete. If property sets or single property entries are omitted from the state definition the corresponding *Controllable* simply keeps its current state.

4.4 Controller

Controllers are the interface for external control systems to issue control commands. The main task is to select the current state from a pre-defined set of *Station States* (Fig. figure 2). The external control system can define the behavior of a *CoDaStation* over time by executing a sequence of *Station States* and observing the returned state information. Additionally, the *Controller* is used for maintenance tasks like reset or shutdown.

This mechanism reflects the ideas of the segment sequence control concept of Wendelstein 7-X [3] and of course a corresponding *Controller* has been implemented. Every data acquisition system based on the *CoDaStation* will automatically be able to participate in the control system of Wendelstein 7-X. However, the integration into alternative control systems (e.g. for laboratory setups or test environments) is still possible by simply replacing the *Controller* component.

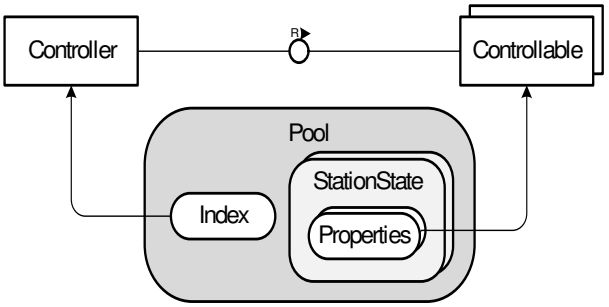

Figure 2: Control components

5 Modularization

5.1 AddOns

To make the *CoDaStation* modular a plug-in concept has been developed that is based on the strategy pattern [8]. These plug-ins are called *AddOns*. An *AddOn* is basically a package that extends the *CoDaStation* with new features (Fig. figure 3). It provides a set of predefined methods to inspect and retrieve these features.

In order to realize a toolbox-like modularization mechanism *AddOns* are designed to have only a few weak dependencies to the *CoDaStation*. These dependencies are defined as a fixed software interface. Hence, the *CoDaStation* does not have to know anything specific about the *AddOns* it uses. It will still work even if no *AddOn* is available but it will possibly not be able to provide specific functionalities. The *AddOn* on the other hand does not rely on changes of the *CoDaStation* other than changes of the interface definitions. This strong modularization concept permits extensive test scenarios on both sides of the interface.

5.2 Resources

Every *AddOn* provides the functionality of one *Resource*. A *Resource* may be a device or some service. Some examples are:

- a data acquisition device
- a manipulator or other active device
- a signal processing framework
- a web service

Multiple instances of a *Resource* may exist. For example a Resource might provide the functionality of a device of a certain type but multiple devices of this type are installed.

5.3 Features

The functions of a *Resource* are provided as *Features*. A *Feature* can be a *Controllable*, a *Signal Provider* or a *Signal Consumer*. The list of supported *Features* is open for extension. Other *Features* may be provided as soon as they are included in the *AddOn* interface definition.

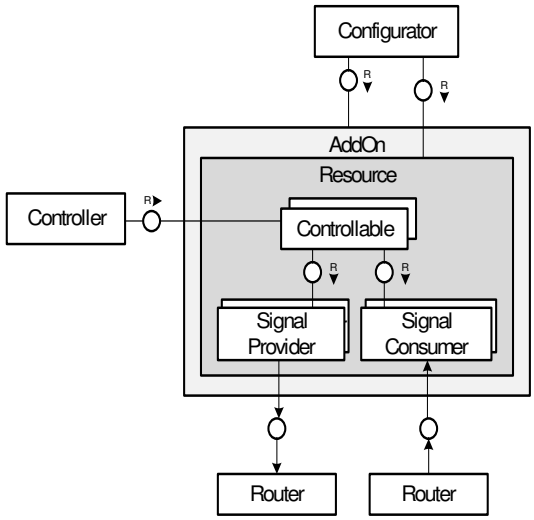

Figure 3: AddOn-centric view of the CoDaStation

6 Configuration

6.1 Configuration

In order to make the *CoDaStation* execute a specific task the required *Resources* and *Features* as well as the allowed *Station States* have to be configured. In different environments the way in which the configuration is provided may also be different. Therefore, a software interface has been defined to pass configurations in different formats to the *CoDaStation*.

The *Configuration* interface describes a specific setup of a *CoDaStation*. The setup includes:

- all used resources and features
- static properties that do not change at runtime (for example device identifications)
- the static setup of the signal processing network by specification of
  - *Routers* with corresponding *Signal Providers* and *Signal Consumers*
  - *Routing Jobs* and attached *Routers*
- the allowed *Station States*

6.2 Configurator

To retrieve *Configurations* and interpret specific formats the *Configurator* is used. It reads a *Configuration* from a given source and initializes *Resources*, *Features*, *Routers* and *Routing Jobs* accordingly (Fig. figure 4). The configuration source may be a database, a file or even an interactive application. Again, this supports test scenarios due to the possibility to create simple file-based test configuration sources for example.

In the same way like *Configurations* the *Configurator* reads and interprets *Station States* from the configuration source. The retrieved *Station States* are stored into an internal pool in the memory of the *CoDaStation* for later use. Which *Station States* exactly are read from the configuration source is defined as part of the *Configuration*.

7 Supervision

7.1 Supervisor

When many *CoDaStations* are running at an experiment site supervision becomes an important point. The *CoDaStation* provides a *Supervisor* interface based on the

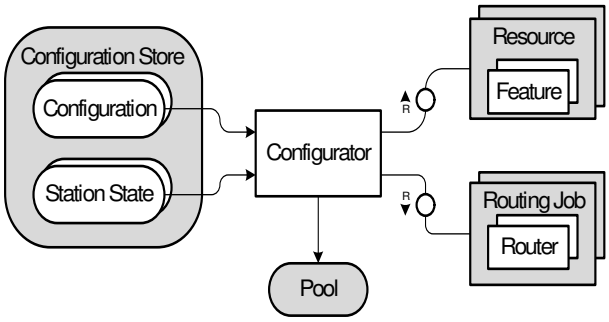

Figure 4: Configuration

observer pattern [8]. Aside from the status information that can be retrieved via the *Controller* interface other internal components of the *CoDaStation* can directly request information from the *Supervisor* or register and get notified on specific events. Typical examples for information provided by the *Supervisor* are:

- the error state of a *Resource*
- the fill-level of an internal buffer of a *Resource*
- the throughput of a *Router*

Using this supervision mechanism the *CoDaStation* can be diagnosed more easily and errors can be detected in a short time or even prevented.

7.2 Remote Supervision

Special *Supervisor* clients are used to make supervision information available from remote via network connections. The current Java-based *CoDaStation* implementation provides a JMX interface [9] to connect to the *CoDaStation* remotely and request status information. Because JMX is a widely used protocol this interface can easily be integrated into common monitoring frameworks like Nagios for example [10]. Thus, even a high number of *CoDaStations* can be observed centrally.

8 Testability

As mentioned before a reliable data acquisition system requires extensive testing possibilities. Because of its well-defined interfaces a lot of the *CoDaStation* components can be replaced by mockups. This allows an easy integration into test environments.

For tests of the basic *CoDaStation* functions a simulator *Resource* has been implemented that provides a *SignalProvider* mockup. It calculates simple configurable data signals in software (e.g. sine or sawtooth functions) and so allows system independent tests of the basic *CoDaStation* software.

9 State and Future Work

The implementation of the described architecture for Wendelstein 7-X is in progress. The basic functions are available and several essential *AddOns* are already provided (time measurement devices, analog to digital converters, data collecting resources using network interfaces). Currently tests concerning the reliability and performance of the implemented software components are done.

With Wendelstein 7-X nearing completion additional data acquisition devices will have to be implemented and a scalable deployment strategy for the *CoDaStation* software has to be defined.

References

[1] P Heimann, S Heinzel, Ch Hennig, H Kühntopf, H Kroiss, G Kühner, J Maier, J Reetz, M Zilker, Status report on the development of the data acquisition system of Wendelstein7-X, Fusion Engineering and Design, Volume 71, Issues 1–4, June 2004

[2] Jörg Schacht, Heike Laqua, Marc Lewerentz, Ina Müller, Steffen Pingel, Anett Spring, Andreas Wölk, Overview and status of the control system of WENDELSTEIN 7-X, Fusion Engineering and Design, Volume 82, Issues 5–14, October 2007

[3] Jörg Schacht, Helmut Niedermeyer, Heike Laqua, Anett Spring, Ina Müller, Steffen Pingel, Andreas Wölk, Tasks and structure of the WENDELSTEIN 7-X control system, Fusion Engineering and Design, Volume 81, Issues 15–17, July 2006

[4] Jörg Schacht, Helmut Niedermeyer, Christian Wiencke, Jens Hildebrandt, Andreas Wassatsch, A trigger–time–event system for the W7-X experiment, Fusion Engineering and Design, Volume 60, Issue 3, June 2002

[5] Marc Lewerentz, Dieter Aßmus, Torsten Bluhm, Stefan Heinrich, Christine Hennig, Uwe Herbst, Christiane Meyer, Eric Köster, Ina Müller, Heike Laqua, Matthias Otte, Steffen Pingel, Jürgen Sachtleben, Jörg Schacht, Anett Spring, Andreas Wölk, First experiences with the new W7-X like control system at the WEGA stellarator, Fusion Engineering and Design, Volume 84, Issues 7–11, June 2009

[6] Knöpfel, A., Gröne, B. and Tabeling, P. (2005). Fundamental Modeling Concepts: Effective communication of IT systems. Chichester Hoboken, NJ: J. Wiley & Sons.

[7] Fundamental Modeling Concepts
`http://www.fmc-modeling.org`

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Java Management Extensions (JMX)
`http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`

[10] Nagios Infrastructure Monitoring
`http://www.nagios.org`