A novel solution for controlling hardware components of accelerators and beamlines

Igor Khokhriakov (a), Olga Merkulova (b), Alexander Nozik (c,d),
Victoria Mazalova (e)*

(a) Helmholtz-Zentrum Hereon, Outstation DESY, Notkestrasse 85, 22607 Hamburg, Germany
(b) IK Company, Botanicheskaya st. 5, 127015 Moscow, Russia
(c) Moscow Institute of Physics and Technology, Institutsky Lane 9, Dolgoprudny, Moscow Region 141700, Russia
(d) JetBrains Research – St. Petersburg, Kantemirovskaya st. 2, 194100, Russia
(e) Centre for Free Electron Laser Science, DESY, Notkestrasse 85, 22607 Hamburg, Germany
Correspondence e-mail: victoria.mazalova@desy.de

Abstract - A novel approach to the remote-control system for the compact multi-crystal energy-dispersive spectrometer for x-ray emission spectroscopy (XES) applications has been developed. This new approach is based on asynchronous communication between software components and on reactive design principles. In this paper we identify the challenges we have faced, our solution to them as well as the implementation and future development prospects. The main motivation of this work was the development of a new holistic communication protocol that can be implemented to control various hardware components.

Keywords: control system; system design; reactive system; physics facility; experiment control, x-ray spectroscopy

1. Introduction

Along with ongoing advances in the development of large-scale particle accelerators such as X-ray free electron lasers (XFELs) and synchrotrons, these facilities continue to operate in a "high demand and limited access" mode. The recent achievements in laser technologies open up new possibilities for the construction of compact accelerators, which will lead to an increase in the number and availability of such facilities for users delivering new technologies to universities, institutes, hospitals, etc. Two compact X-ray free electron lasers are almost built at the Deutsches Elektronen-Synchrotron (DESY) (Kärtner *et al.,* 2016) and Arizona State University (ASU) (Graves *et al.,* 2017). A mini-synchrotron, the Munich Compact Light Source (MuCLS), operates at the Technical University of Munich (TUM) (Günther *et al.,* 2020), and other developments are underway around the world. This growth in scientific instruments leads to a variety of implemented Supervisory Control And Data Acquisition (SCADA) systems to control equipment. Thus, the control of technical components of accelerators at DESY are implemented with the DOOCS (Grygiel *et al.*, 1996) and TINE (Bartkiewicz *et al.*, 2007) control systems, whereas equipment of beamlines are controlled with TANGO (Götz *et al.*, 2003), KARABO (Hauf *et al.,* 2019) or EPICS (Dalesio *et al.*, 1994). Such diversity of SCADA systems requires development of complicated communication protocols when synchronization and communication between different technical components is needed, especially in the facilities where multiple SCADA systems are used.

Frequently, the integration of new equipment at the large-scale facilities is done by adapting the existing control software used for other devices. Such a strategy often results in replication of large databases and huge software infrastructure. In our work, we propose a different approach based on the use of modern software solutions, with the help of which one can quickly achieve similar functionality with less resources.

While the number of various research techniques and analytical instruments constantly grows, the idea of equipment sharing may become reasonable in the near future. Following this concept, analytical instruments for compact accelerators, when possible, should also be sufficiently compact and versatile, which includes the possibility of integration of their control

system into various facilities. Whereas good and up-to-date equipment control practice in newly developed facilities should be envisaged.

The XES spectrometer was developed for the compact XFEL beamline at DESY (the AXSIS project) and can be used for the single-short collection of XES spectra simultaneously with X-ray diffraction/X-ray scattering experiments in a hard X-ray energy range (4-12 keV) with attosecond time resolution. One of the requirements for the XES spectrometer for the compact XFEL was its versatility, namely, the possibility of installing the spectrometer at different beamlines and X-ray facilities. This requirement limits not only the weight and size of the device, but also requires the spectrometer control system to be independent and easily integrated into various SCADA systems.

In our paper we present a new solution based on asynchronous communication between software components and on reactive design principles. Our approach to hardware control can be implemented for controlling and communicating any technical equipment in real-time regardless of its manufacturer, scale, and complexity. We show, as an example, the implementation of the developed protocol for controlling hardware components of the X-ray spectroscopy end station of the AXSIS beamline.

2. Challenges

2.1. Hardware challenges

To follow the ultrafast dynamics of chemical processes achievable with the pulsed X-ray source working at a high repetition rate, the XES spectrometer for the AXSIS beamline is based on the von Hamos geometry (von Hamos, 1932). Due to its cylindrical shape, each analyser crystal diffracts X-rays in a specific energy range along the cylinder axis (dispersion direction) and focuses the X-rays along the base of the cylinder (focusing direction), allowing the entire XES spectrum from each individual X-ray pulse to be measured.

The principle of operation of the XES spectrometer is based on the Bragg diffraction of the incoming X-ray beam from multiple analyser crystals. After interacting with the sample, the emitted X-ray photoelectrons are diffracted and focused from the crystal analysers into the position-sensitive 2D detector following Bragg's law $n\lambda = 2d\sin\theta$, where n – the diffraction order, λ - the x-ray wavelength, d – the distance between crystal planes, θ - the Bragg angle.

To provide high efficiency of the experimental data for the low-concentrated biological samples or catalysts, the XES spectrometer was built from multiple crystal analysers increasing a covered solid angle. Unlike other existing solutions (Alonso-Mori *et al*., 2012; Szlachetko *et al*., 2012), we aimed to develop a highly efficient, high resolution, compact and portable device at the same time. With this objective, the XES spectrometer for the AXSIS beamline consists of 8 (2 x 4 matrix) independent cylindrically bent to 250 mm crystal-analyser. Each crystal-analyser can be moved by three individual motors to perform three different tasks: 1) change the Bragg angle (and the energy range of detected spectra); 2) change the focus of the projection of XES spectra to the detector, or 3) position an individual XES spectrum on a different area on the detector.

2.2. Software challenges

2.2.1. Lightweight and compact

Thinking of how the hardware tasks match with the existing software solutions we face the following. Usually existing SCADA systems, which operate with hardware, are huge and have solutions for lots of problems. Most of them are of great complexity with their own ecosystems such as libraries, servers, special tools, graphical user interfaces (GUIs), etc., which require efforts and resources to get into, and in-house support and development from time to time. There is no possibility to use only a part of it if the task is not big and if there are not so many drawbacks to solve. We do not need special complicated features from SCADA systems. As we have a **compact** multi-crystal energy-dispersive spectrometer there is also **no need to create**

and maintain a dedicated database which is usually necessary when using a SCADA system. So most of the existing solutions were with redundant functionality for our task.

2.2.2. Portable across different beamlines

Possibility of **installing the spectrometer at different beamlines** and X-ray facilities was another important task to solve. Incompatibility between different systems is the real case in this situation. Devices from one SCADA system can't be easily connected to another one because they depend on different transport protocols which are used in each SCADA system (for example TANGO uses CORBA (Henning, 1999), whereas DOOCS uses SUN ONC (Srinivasan, 1995)). The same applies for visualization services. Indeed, there are some wrappers from the SCADA systems which give the possibility to have connections with other systems but deep knowledge of systems is required to implement these connectors. So when one wants to bring together two devices that use different SCADA systems, the SCADA integration has to be written from scratch. This is not convenient if the device is to be used in different facilities where different SCADA solutions are operated.

2.2.3. Minimizing third party dependencies and vendor lock in a device server

Interconnecting different SCADA systems and providing visualization are not the only challenges to be solved. We also want to provide a common interface and tools to aggregate new device servers, not connected to any established platform. Those stand-alone device servers have general use-cases:

- Providing a way to create a device server that can interact with several control systems. Currently, one needs to create a custom device server implementation for each platform in order to make the device pluggable into them. It requires a significant effort and limits the number of devices that could be used in a specific experiment.
- Using device servers in a stand-alone mode, without integration with the other devices and very simple read-out and visualization. Currently, most systems do not allow running devices in this mode without setting-up central naming and communication services, configuration database, storage database, etc.

The first case is rather self-explanatory. Device manufacturers obviously want to write a device server once using a convenient tool-set instead of supporting a device server for each platform (usually in different languages).

The second use-case mostly appears in small experiments and during component testing. It does not make sense to establish the whole SCADA system for a single device, but the drivers (device servers) are already integrated in those systems and could not be used alone. The LabVIEW platform (Travis *et al*., 2007) provides some means to manage small-scale systems, but is also limited when it comes to custom hardware.

2.2.4. Making asynchronous communication a first class citizen

Most existing control systems are using synchronous (request-response) and often one-direction communication protocols when data is propagated from the lower layers to the upper ones, e.g. when data is propagated from the hardware to the GUIs on request (Birrell *et al*., 1984; Gotz *et al*., 2013) also known as client polling. This means that we either receive a command execution's result or an error, not knowing what is going on in between. Request-response models are very good in peer-to-peer communication (Schoder *et al*., 2005) when every node talks to another one and only one at a time, but not in multiple peer connections, when a node talks with multiple nodes simultaneously. Request-response based systems are good until we want to scale up in a multi-directional communication way. For instance, in case we need to add more motors, devices or software components that talk to each other, the code base quickly becomes complicated with lots of if-else statements and nested loops.

Summarizing, the synchronous communication scheme is easy to implement, but it has several disadvantages (Hintjens, 2013):

- Synchronicity in most cases means blocking control flow, see Figure 1a. This leads to latency and freezes in the whole system;
- Synchronous systems based on peer-to-peer communication do not **scale** well, e.g. dozens of peers talking to each other in a blocking way;
- Finally, such systems are very hard to load balance.

In order to avoid the mentioned above drawbacks, one should use **asynchronous communication** (McCool *et al*., 2012) and **reactive design** (Kuhn *et al*., 2017), (see detailed description in the next chapter 2.2.5).
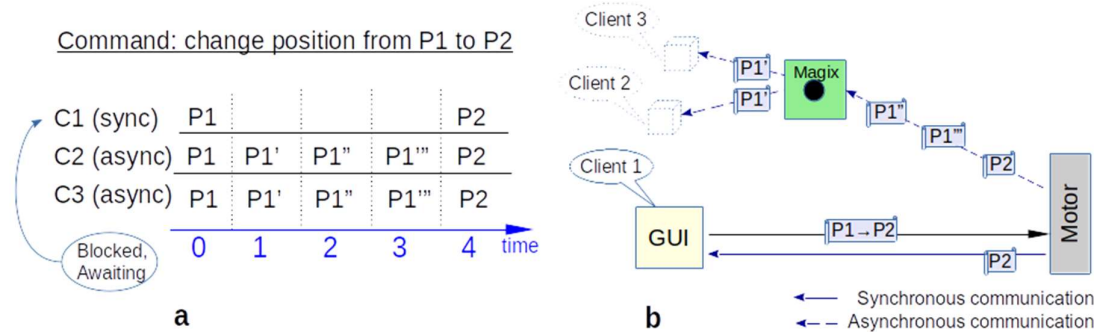

Figure 1   *An example of the difference between synchronous and asynchronous communication. Client 1 (GUI) sends a message with a command to change the state of the motor from the current P1 position to P2 at the t0 time. After receiving the message, the motor begins to execute the command and changes its states. The motor moves through intermediate positions P1', P1", P1'" to reach the P2 position. Via synchronous communication the Client 1 (GUI) receives response P2 at the t4 time. Whereas via asynchronous communication Client 2, Client 3 (these may be another GUI, other motors or systems), which want to receive information about intermediate motor's states, receive these intermediate positions simultaneously: P1' at t1 time, P1" at t2, P1'" at t3 and P2 at t4 time.*

In our case, this means that if we send a command for the motor to change position from P1 to P2, we want to get intermediate positioning values of the motor (P1', P1", P1'"). Asynchronous way of communication can give us this possibility without sending constant requests from clients' side. Moreover, when having something like a broker (see Magix in Figure 1b), all clients that want to get all information from the motor, receive it at the same time and this information is the same for all of them. In case of peer-to-peer/synchronous communication if one of the clients requests the motor's position it may get a response with intermediate positioning value at a time while the other client requesting for the position value will get another one. This leads to information inconsistency at some time points, see Figure 2.
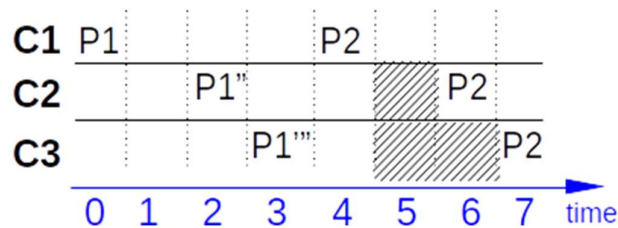

Figure 2   *Example of synchronous communication and polling. Three clients at a different time start to ask for the motor's position value with the time slot polling of 5. The first client C1 receives the final value of P2 at the t4 time while the second client C2 and the third C3 would receive this value only at time t6 and t7. Until the t7 time all three clients see different position values.*

Thinking of how these tasks could be addressed, we came up with an idea of creating an open source interconnection platform - the place where information from different SCADA systems could meet and share information.

2.2.5. Being reactive by design

According to our tasks, we need:

- to have a *responsive* solution when command execution takes time. This is solved by relying on asynchronous communication;

- to continue to operate in case one or more components fail or, in other words, to have a *resilient* system. This is provided by the way the errors are treated. Errors are messages, but not exceptions that stop the process;

- to operate without problems and delays when a considerable number of clients (e.g. status monitors) are connected to the instrument—have an *elastic* system. This is accomplished by dynamically deploying multiple broadcasters (see Magix in Figure 1b) instances;

- to use messages for communication between different devices and SCADA systems.

The above statements indicate that our solution should be *responsive*, *resilient*, *elastic* and message *driven*. This means that it fits into a reactive definition or reactive manifesto (Bonér *et al.,* 2014). As we design a reactive system, message flow or message transport is to be a reactive stream which is specified in the ReactiveX Streams description.

One of the benefits of reactive streams is that they have a very rich ecosystem (i.e. libraries) also known as reactive extensions (Rx) - libraries with Rx supported platforms that can be attached to the reactive streams to extract data (information in messages) from them. These reactive extensions exist for almost all platforms and are written in most of the known software languages and provide similar functionality due to the standard reactive paradigm.

3. Solution

The main idea of the solution is to make it possible for the spectrometer and other hardware components to communicate with different control systems. Data from one control system should be accessible for another control system in case several systems are used at once. This solution provides a generic way of communication between different SCADA systems and the instrument because there is no need to know each SCADA system to operate it. The approach, as it is, is quite widely used in web-development (especially in so-called microservice architecture (Wolff, 2016), but is not yet fully adopted in SCADA systems. The solution is based on work done by Khokhriakov *et al*., 2014, 2017 and the "Troitsk nu-mass" experiment in search of masses of active and sterile neutrinos (Abdurashitov *et al*., 2015).

3.1. Message specification

The elementary unit of the communication process is a message. Each message is a predefined structure filled with information. This structure has been defined after analysis of SCADA systems (TANGO, TINE, DOOCS, EPICS) for more than 5 years and written in a Request for Comments (RFC) way (Daigle, 2007) which means that the structure can be updated if needed. Thus, for different SCADA systems the structure of the message is the same. Once a message is created it can be used/applied by clients (other SCADA systems, GUI or algorithms), see Figure 3.
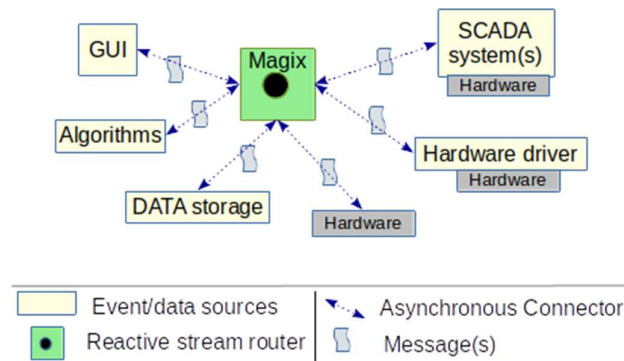
Figure 3   *Overview of the interconnection platform. Here Magix is a core component responsible for the message flow between components. Each software component, shown as yellow boxes, deals only with incoming messages without knowing any details of the counterparty component. All communication is asynchronous.*

Each box in Figure 3 (green, yellow or grey) is a solution's component which is a stack of technologies. Depending on functional requirements a set of components can be adjusted, thus, allowing to build the required system as from Lego blocks.

3.2. Asynchronous communication

By design the solution implies an asynchronous way of messages' interchange between components (see Figure 3). When required, an asynchronous way can be always transformed into synchronous. Components may also establish peer-to-peer communication as in existing SCADA systems because the solution extends SCADA systems, not replacing them.

The core component of the solution—Magix—provides broadcasting of messages and subscribing capabilities. Messages are delivered by a transport which can be implemented by any existing transport frameworks (e.g. ZMQ, Kafka, etc) depending on the requirements. For instance, for small instruments when the whole system is on a single host the solution can be packed into a single process. In this case, the transport may be implemented by ZMQ in-process sockets. For mid to large instruments the transport may be implemented by Apache Kafka—an event streaming platform or D-Bus (Palmieri, 2005) or ZMQ interproc.

3.3. Reactive design

Thinking about the solution, we rely on the reactive design approach. Another key feature of our solution is a transport agnostic paradigm, i.e. irrelevance to the underlying transport implementation. The solution is implemented using the reactive stream paradigm and is not based on any existing Remote Procedure Call models (Corbin, 2011) or frameworks but based on microservices which asynchronously communicate with each other. Thus, the solution is very container friendly and can be packed into widely used Docker (Poulton, 2016) containers and orchestrated using e.g. Kubernetes (Poulton, 2019) to manage multiple Docker containers.

Summarizing, the main important features of the solution are:

1.   transport agnostic;
2.   reactive system by design;
3.   no need of creating database;
4.   its role is to be a middleware between client applications and upstream control system(s) or third-party components.

It gives the possibility of an agile approach to software development through microservices within the reactive paradigm and makes a more convenient workflow for operating experiments, see Figure 4.
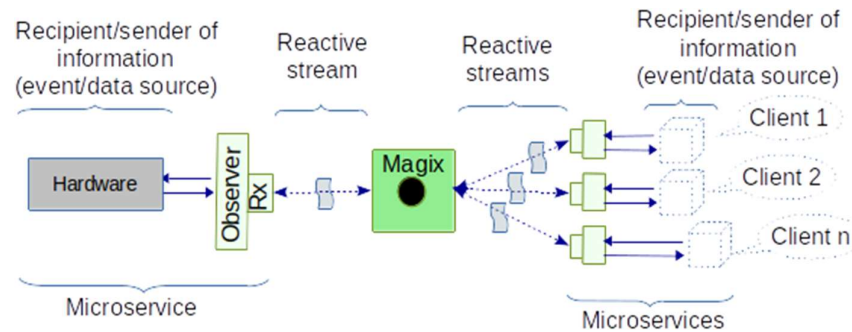
Figure 4 *As in Figure 3, here Magix is a core component responsible for the message flow between counterparts. Each counterpart is a software component or a microservice. Every microservice has its own lifecycle independent from others. The only thing they have in common is that they all attach to a reactive stream produced by Magix using reactive extensions (Rx) and defining observers. Observers "glue" Rx and microservice's logic.*

Hence, our solution gives a variety of possibilities to connect to the streams with messages which can come from SCADA systems, drivers or any other third-party components or microservices. We deal only with a reactive stream that brings standard defined messages and there is no need to know the details of SCADA systems, drivers, or any other third-party components or microservices, we only need to understand data from messages. The messages' structures are defined in the RFC's and the structure of the message is known in advance. Therefore, the approach allows us to use different SCADA systems as stated among the challenges.

3.4. Device server

Microservices are independent functional units (see green, yellow or grey boxes in Figure 3 and Figure 4). In order to address the challenge of reducing proprietary third-party dependencies, and especially to avoid vendor lock, we have started to design a new framework which is a combination of Rx extension (gives us possibility to connect to the data stream and get information from messages) and Rx observer (transforms message's information in the understandable for the recipient way) which is attached to the reactive streams. The framework supports passing device property changes in and out of the device via reactive streams and Magix.

By design, the solution does not require any dedicated component that will do the name resolution service. In other words, it does not require any particular database to be running. Small installations may benefit from it as less resources are required to set up and maintain the system. However, for large scale installations the name resolution service can be implemented.

The key details of implementation of this approach are shown in the next chapter.

4. Implementation

4.1. Hardware

In order to comply with the tasks, three main scenarios for positioning of each individual crystal of the XES spectrometer are envisaged:

- Positioning each crystal in the dispersion direction (changing the Bragg angle) by moving two motors sequentially or simultaneously in positive and negative directions relative to the reference position (see green arrows in Figure 5).
- Change the position of each crystal in the focusing direction by moving the three motors sequentially or simultaneously in positive and negative directions relative to the reference position (see red arrows in Figure 5).
- Moving each crystal as a whole in a positive or negative direction relative to its original position.

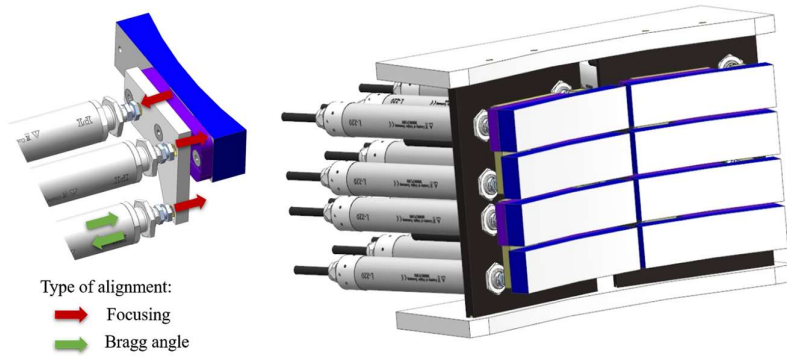All of the described above displacements can be different for each crystal.

Figure 5   *(left panel): one crystal driven by three linear actuators; (right panel): a view of the XES spectrometer.*

Figure 6 shows a schematic representation of the implemented communication network for controlling the XES spectrometer. To realize these scenarios, we used 24 linear actuators driven by two controller modules (Motor 1 and Motor 2 in Figure 6). To ensure the possibility of integrating the spectrometer into any equipment control system, a local network that includes a Raspberry PI computer and a router has been built. The Raspberry PI computer contains all the required software and libraries to control 24 linear actuators of the XES spectrometer. The router is used to provide a network communication between a remote PC and constituent elements of the spectrometer.
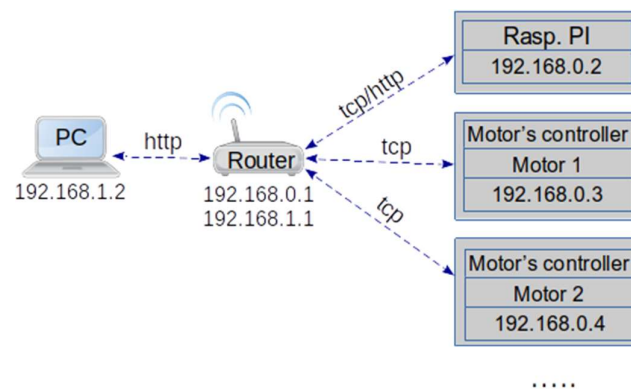

Figure 6   *Schematics of the local network to control the XES spectrometer.*

Any command received from the GUI on the PC is transmitted to the server deployed on the Raspberry PI. Once a command is assigned, it is redirected to the appropriate motor (or device) and its current status is returned to the user. All communications are performed asynchronously, which allows various actions to be performed in parallel.

4.2. Software

In this section we give an overview of the key details of the software solution's implementation.

First, let's have a look at the GUI end of the system. As stated within tasks, our GUI must provide a way for the user to adjust the position of each crystal as a whole by changing the Bragg angle or the focus, as well as interact with every motor independently. Obviously, it also has to be able to initiate the experiment session, connecting to the upstream components, in our case, Magix and software device controllers. Information from the device server is packed into a message described in Waltz-Controls RFC-1 Message. This information can be unpacked by the control system's connector or the GUI. The software device server is running on a dedicated host, communicating with upstream hardware devices via TCP over Ethernet or directly

through USB ports. The data from devices can be visualized directly on the computer where it is acquired, see Figure 7.
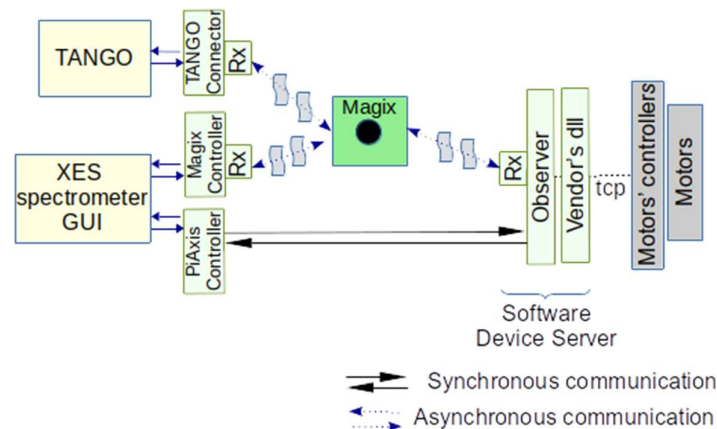

Figure 7   *Control system implementation for XES spectrometer. The software components may communicate both asynchronously and synchronously with other microservices via message's flow which is used to transfer data.*

The XES spectrometer GUI is implemented on the Waltz GUI platform (Götz *et al.,* 2019; Khokhriakov *et al*., 2019) as it is possible to create a user interface conforming to our needs. Moreover, it is an open source web-based solution. As a web application it does not depend on the system and can be run on Windows, Linux and mobile platforms. The XES spectrometer GUI is a typical front-end project that utilizes the ecosystem of NodeJS and JavaScript (Brown, 2014), i.e. dependencies management and builds are done using the npm package manager. The XES spectrometer GUI's source code resides in GitHub. For the graphical part we use the Webix UI library with its widgets collection that provides a huge set of predefined widgets as well as a very intuitive and straightforward way of defining custom widgets and it is also distributed under an open source license.

Two points of interest within the GUI source code are asynchronous and synchronous communication with the upstream device controller. Synchronous communication with the upstream device controller is encapsulated in Pi device synchronous controller (PiAxsisController), asynchronous communication—in the MagixController component.

A part of the MagixController component code example is presented in Figure 8 to show the reactive paradigm approach and asynchronous communication. Figure 8 gives the full implementation of the "move" action. For asynchronous communication, reusable Waltz GUI Magix plugin (waltz-magix-plugin) was independently developed. It provides an implementation of the Magix API specification defined in the Waltz-Controls RFC-2, Magix client API. We get a Waltz-magix-plugin from the application's context, see line 46. In lines 48-53 and 54-59 we observe incoming messages from the upstream device controller and either update the GUI (48-53) or show an error message (54-59). Finally, in lines 61-71 we broadcast a "move" request which is a non-blocking call and we do not expect any response here. Synchronous communication is implemented in the system in a standard way using JavaScript Promises (Rauschmayer, 2018).

```
45    async move(values){
46        const magix = await this.app.getContext(kMagixContext);
47        const id = +new Date();
48        magix.observe(kChannel).pipe(
49            tap(msg => console.debug(msg)),
50            filter(msg => msg.parentId === id)
51        ).subscribe(msg => {
52            this.dispatch(`Moving controller ${this.controller.ip}...`, kPiAxisControllerDone, kPiAxisController);
53        })
54        magix.observe(kChannel).pipe(
55            tap(msg => console.debug(msg)),
56            filter(msg => msg.parentId === id && msg.action === 'error')
57        ).subscribe(msg => {
58            this.dispatchError(new Error(msg.payload.error));
59        })
60        this.dispatch(`Moving controller ${this.controller.ip}...`, kPiAxisControllerDo, kPiAxisController);
61        magix.broadcast(new Message({
62            id,
63            target: kTarget,
64            origin: 'axsis-gui',
65            payload: {
66                ip: this.controller.ip,
67                port: this.controller.port,
68                action: 'MOV',
69                value: values
70            }
71        }), kChannel)
72    }
```

Figure 8 *Example of asynchronous communication for "move" action for motors.*

The software device controller in this setup is implemented using Python 3 and Pi Python library i.e. vendor library. The source code for this component resides in XES device controllers' source code on GitHub.

To export a synchronous communication channel, we use the Flask framework for building REST services (Richardson *et al*., 2013). This provides a very convenient way to communicate with the device controller as HTTP eco-system, on which REST services are based, is extremely rich. For example, we can use just a plain browser to communicate with the device. In general, REST is more preferable to conventional request-response based solutions like CORBA, see e.g. RESTful Architectural Principles by Burke, 2009.

To export asynchronous communication, we developed a reusable Python client implementation for Magix (RxPython MagixClient) and attached an observer to it (see Figure 7). The observer extracts incoming messages and delegates execution. All routines are non-blocking and performed within the async.io event loop.

An alternative implementation of the software device controller (Data acquisition framework based on DataForge) is being developed in Kotlin using kotlin-multiplatform (Nozik, 2019) technology, which has perfect instruments for implementing asynchronous communication. It allows us to share code between the Java virtual machine (JVM), browser and native targets. The key feature of the alternative implementation is bypassing vendor's dll and direct communication with the upstream hardware devices, thus eliminating 3rd party dependency and fulfilling one of our challenges. The project is developed in JetBrains Research fellowship and currently features a library for creating asynchronous device servers of different complexity as well as communication plugins that follow Magix specification both for REST/SSE and RSocket transport layers.

As for the Magix component, in addition to the Kotlin implementation, we developed an experimental reusable implementation based on HTTP2.0 and SSE written in Java (source code is on Magix component's GitHub page). Following RFC-2, it implements two methods "subscribe" and "broadcast" as REST endpoints.

The implementation of the solution can be considered as a reactive system. The system has to fulfill certain requirements (Bonér *et al.,* 2014) to be reactive: *message driven*, *elastic*, *resilient*, and *responsive*. The implementation is naturally *message driven*. Synchronous HTTP communication is, in fact, a message driven due to the asynchronous nature of JavaScript (ZeoLearn, 2018). It is *elastic* as it is easy to imagine horizontal scalability by increasing the number of Magix instances and the balance load between them. Natural limitation for scalability is the upstream hardware devices as we have a fixed number of motors. However, it

is easy to leverage the load applying back-pressure strategies (Phelps, 2019) provided by Reactive extensions. *Resilience* is achieved by early failure interception in the software components and emitting an "error" message. Finally, it is *responsive* as there are no blocking calls in the communication channels.

An important feature of our implementation is that it does not require the setup, the only thing one needs is to plug in the router into the on-premise network and access the GUI using a browser. Naturally, one can switch to a different computer without stopping the measurements and restarting the acquisition.

5. Conclusions and perspectives

In this paper, we have presented tasks, challenges, solutions, and implementation of the XES spectrometer and its control system. Specifically, our hardware solution has to be compact and versatile, also in terms of software, and can be easily integrated into existing instruments.

In terms of hardware tasks for the short-working distance multi-crystal von Hamos spectrometer we ended up with a non-commercial customized solution that meets the requirement of the AXSIS beamline.

The software control system must provide the possibility to understand what is going on with the hardware while executing the commands. This gives us the following challenges: the software solution must be responsive, resilient, elastic and message driven.

To meet the software challenges, we developed a new control system based on the reactive paradigm as existing solutions are typically RPC systems and are not suitable as they do not provide asynchronous communication and also typically are very complex and complicated for our compact device. They also demand the creation and maintenance of a database, which we would like to avoid.

Our solution is a transport agnostic and we foresee components implemented in other languages, technology stacks e.g. Python and Kotlin-Native. Moreover, using Magix we can extend capabilities of any existing SCADA, distributed control system (DCS) solution allowing easy integration of new protocols, smooth migration to new protocols, or event tech stacks.

The resulting implementation is open source. Most of the components are reusable and reside under Waltz-Controls organization on GitHub. Magix related components are aggregated in the Piazza project. Software device server is implemented in both Kotlin and Python languages. This implementation can be easily scaled to integrate more hardware and software components.

The XES spectrometer and its control system have been successfully tested during a research and development (R&D) beamtime at Pohang Accelerator Laboratory X-ray Free Electron Laser (PAL-XFEL).

References

[Abdurashitov, D., Belesev, A., Berlev, A., Chernov, V., Geraskin, E., Golubev, A., Koroteev, G., Likhovid, N., Lokhov, A., Markin, A., Nozik, A., Pantuev, V., Parfenov, V., Skasyrskaya, A., Titov, N., Tkachev, I., Tkachov, F., Zadorozhny S. (2015). *JINST*, **10**, T10005, https://doi.org/10.1088/1748-0221/10/10/T10005]

[Alonso-Mori, R., Kern, J., Gildea, R. J., Sokaras, D., Weng, T.-C., Lassalle-Kaiser, B., Tran, R., Hattne, J., Laksmono, H., Hellmich, J., Glöckner, C., Echols, N., Sierra, R. G., Schafer, D. W., Sellberg, J., Kenney, C., Herbst, R., Pines, J., Hart, P., Herrmann, S., Grosse-Kunstleve, R. W., Latimer, M. J., Fry, A. R., Messerschmidt, M. M., Miahnahri, A., Seibert, M. M., Zwart, P. H., White, W. E., Adams, P. D., Bogan, M. J., Boutet, S., Williams, G. J., Zouni, A., Messinger, J., Glatzel, P., Sauter, N. K., Yachandra, V. K., Yano, J., Bergmann, U. (2012). *Proc. Natl Acad. Sci. USA*, **109(47)**, 19103–19107, https://doi.org/10.1073/pnas.1211384109]

[Alonso-Mori, R., Kern, J., Sokaras, D., Weng, T.-C., Nordlund, D., Tran, R., Montanez, P., Delor, J., Yachandra, V. K., Yano, J. & Bergmann, U. (2012). *Rev. Sci. Instrum.* **83**, 073114, https://doi.org/10.1063/1.4737630]

[Apache Kafka - an event streaming platform. Apache Software Foundation, https://kafka.apache.org/]

[Back-pressure strategies, Reactive Manifesto Glossary, https://www.reactivemanifesto.org/glossary#Back-Pressure]

[Bartkiewicz, P., Duval, P. (2007). *Meas. Sci. Technol.***18,** 2379; https://tine.desy.de]

[Birrell, A.D. & Nelson, B.J. (1984)..*ACM Transactions on Computer Systems,* **2**, 39–59, https://doi.org/10.1145/2080.357392]

[Bonér, J., Farley, D., Kuhn, R., Thompson, M. (2014). The Reactive Manifesto, https://www.reactivemanifesto.org/]

[Brown, E. (2014). *Web Development with Node and Express: Leveraging the JavaScript Stack*. ISBN-10:1491949309]

[Burke, B. (2009). *RESTful Java with JAX-RS 2.0: Designing and Developing Distributed Web Services*. O'Reilly Media, Inc. ISBN-10 144936134X]

[Corbin, J.R. (2011). *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls*. Springer. ISBN-10:1461277531]

[Daigle, L. (2007). *The RFC Series and RFC Editor*. IETF. https://doi.org/10.17487/RFC4844]

[Dalesio, L., Hill, J., Kraimer, M., Lewis, S., Murray, D., Hunt, S., Watson, W., Clausen, M., Dalesio, J. (1994). *Nucl. Instrum. Methods Phys. Res. A*, **352**, 179–184, https://epics.anl.gov/]

[Data acquisition framework based on DataForge, Source code, https://github.com/mipt-npm/controls.kt/tree/dev]

[Hauf, S., Heisen, B., Aplin, S., Beg, M., Bergemann, M., Bondar, V., Boukhelef, D., Danilevsky, C., Ehsan, W., Essenov, S., Fabbri, R., Flucke, G., Fulla Marsa, D., Göries, D., Giovanetti, G., Hickin, D., Jarosiewicz, T., Kamil, E., Khakhulin, D., Klimovskaia, A., Kluyver, T., Kirienko, Y., Kuhn, M., Maia, L., Mamchuk, D., Mariani, V., Mekinda, L., Michelat, T., Münnich, A., Padee, A., Parenti, A., Santos, H., Silenzi, A., Teichmann, M., Weger, K., Wiggins, J., Wrona, K., Xu, C., Youngman, C., Zhu, J., Fangohr, H. & Brockhauser, S. (2019). *J. Synchrotron Rad.* **26**, pp. 1448–1461, https://doi.org/10.1107/S1600577519006696]

[Henning, M., Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Addison Wesley. ISBN-13 978-0201379273.]

[Hintjens, P. (2013). *ZeroMQ*. O'Reilly Media. ISBN: 9781449334062, https://zguide.zeromq.org/docs/chapter3/#The-REQ-to-REP-Combination]

[Götz, A., Bourtembourg, R., Chaize, J-M., Verdier, P., Coutinho, T., Moldes, J., Pivetta, L., Khokhriakov, I., Merkulova, O., Gara, S., Goryl, P., Liszcz, M., Joubert, A., Abeille, G., Mant, G., Braun, T., Hardion, V., Bartolini, M. (2019). *Proc. ICALEPCS'19*, 1234-1239. https://doi.org/10.18429/JACoW-ICALEPCS2019-WEPHA058]

[Götz, A., Taurel, E., Pons, J., Verdier, P., Chaize, J., Meyer, J., Poncet, F., Heunen, G., Götz, E., Buteau, A., Leclercq, N., Ounsy, M. (2003). *Proceedings of the 2003 International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS2003)*, 220–222. MP705]

[Gotz, A., Taurel, E., Verdier, P., Abeille, G. (2013). *Proc. ICALEPCS'13*, TUCOCB07. ISBN 978-3-95450-139-7, https://accelconf.web.cern.ch/ICALEPCS2013/papers/tucocb07.pdf]

[Graves, W., Chen, J., Fromme, P., Holl, M., Kirian, R., Malin, L., Schmidt, K., Spence, J., Underhill, M., Weierstall, U., Zatsepin, N., Zhang,C., Brown, D., Hong, K., Moncton,D., Nanni, A., Limborg-Deprey, C. (2017). *38th International Free Electron Laser Conference,* 225-228, https://doi.org:10.18429/JACoW-FEL2017-TUB03]

[Grygiel, G., Hensler, O. & Rehlich, K. (1996). DOOCS: A Distributed Object-Oriented Control System on PC's and Workstations. *Proceedings of the 1st International Workshop on Emerging Technologies and Scientific Facilities Controls (PCaPAC96)*, https://doocs-web.desy.de]

[Günther, B., Gradl, R., Jud, C., Eggl, E., Huang, J., Kulpe, S., Achterhold, K., Gleich, B., Dierolf, M., Pfeiffer, F. (2020). *J. Synchrotron Rad.* **27**, pp. 1395-1414, https://doi.org/10.1107/S1600577520008309]

[JavaScript Promises, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise]

[Kärtner, F.X., Ahr, F., Calendron, A.-L., Çankaya, H., Carbajo, S., Chang, G., Cirmi, G., Dörner, K., Dorda, U., Fallahi, A., Hartin, A., Hemmer, M., Hobbs, R., Hu, Y., Huang, W.R., Letrun, R., Matlis, N., Mazalova, V., Mücke, O.D., Nanni, E., Putnam, W., Ravi, K., Reichert, F., Sarrou, I., Wu, X., Yahaghi, A., Ye, H., Zapata, L., Zhang, D., Zhou, C., Miller, R.J.D., Berggren, K.K., Graafsma, H., Meents, A., Assmann, R.W., Chapman, H.N., Fromme, P. (2016). *Nucl. Instrum. Methods Phys. Res. A* **829**, pp. 24–29, https://doi.org/10.1016/j.nima.2016.02.080]

[Khokhriakov, I., Lottermoser, L., Beckmann, F. (2017). Proceedings Volume 10391, Developments in X-Ray Tomography XI; 103911H]

[Khokhriakov, I., Lottermoser, L., Gehrke, R., Kracht, T., Wintersberger, E., Kopmann, A., Vogelgesang, M., Beckmann, F. (2014). Proceedings Volume 9212, Developments in X-Ray Tomography IX; 921217, https://doi.org/10.1117/12.2060975]

[Khokhriakov, I., Wilde, F., Merkulova, O. (2019). *Proc. ICALEPCS'*19, pp. 1519-1524. WESH3003, https://doi.org/10.18429/JACoW-ICALEPCS2019-WESH3003]

[Kotlin, Multiplatform programming, https://kotlinlang.org/docs/multiplatform.html]

[Kuhn, R., Hanafee, B. & Allen, J (2017). *Reactive Design Patterns*. Manning Publications. ISBN: 9781617291807]

[Magix component's GitHub page. Source code for HTTP2.0 and SSE written in Java, https://github.com/waltz-controls/magix-war-plugin]

[MagixController component, Source code, https://github.com/Ingvord/axsis-xes-gui/blob/master/src/controllers/magix_pi_controller.js]

[McCool, M., Reinders, J. & Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. ISBN-10: 0124159931]

[Nozik, A. (2019). AIP Conference Proceedings 2163, 040004, https://doi.org/10.1063/1.5130103]

[npm package manager, https://www.npmjs.com/]

[Palmieri, J. (2005) Get on D-BUS. Red Hat Magazine, https://web.archive.org/web/20151023072022/http://www.redhat.com/magazine/003jan05/features/dbus/]

[Phelps, J. (2019). Backpressure explained — the resisted flow of data through software, https://medium.com/@jayphelps/backpressure-explained-the-flow-of-data-through-software-2350b3e77ce7]

[PiAxsisController, Pi device synchronous controller source code, https://github.com/Ingvord/axsis-xes-gui/blob/master/src/controllers/pi_controller.js]

[Piazza project, https://github.com/waltz-controls/piazza]

[Poulton, N. (2016). *Docker Deep Dive: Zero to Docker in a single book*. ASIN:B01LXWQUFF]

[Poulton, N., Joglekar, P. (2019). *The Kubernetes Book: The Fastest Way to Get Your Head Around Kubernetes*. Packt Publishing, Limited. ISBN:1838984380]

[Rauschmayer, A. (2018) *Exploring ES6*. Ch. 25, https://exploringjs.com/es6/ch_promises.html]

[ReactiveX Streams description, https://www.reactive-streams.org/]

[Richardson, L., Amundsen, M., Ruby, S. (2013). *RESTful Web APIs: Services for a Changing World*. O'Reilly Media; 1st edition. ISBN-10: 1449358063]

[Rx observer. ReactiveX. Observable, http://reactivex.io/documentation/observable.html]

[Rx supported platforms, http://reactivex.io/languages.html]

[RxPython MagixClient, Python client implementation for Magix, https://github.com/waltz-controls/magix-python-client https://pypi.org/project/magix-client/0.4/]

[Schoder, D., Fischbach, K., Schmitt C. (2005). *Core Concepts in Peer-to-Peer Networking*, pp. 1-27. Idea Group Inc, https://doi.org/10.4018/978-1-59140-429-3.ch001]

[Srinivasan, R. (1995). *Request for Comments: 1831, Remote Procedure Call Protocol Specification Version 2*, https://www.freesoft.org/CIE/RFC/1831/index.htm]

[Szlachetko, J., Nachtegaal, M., de Boni, E., Willimann, M., Safonova, O., Sa, J., Smolentsev, G., Szlachetko, M., van Bokhoven, J.A., Dousse, J.-Cl., Hoszowska, J., Kayser, Y., Jagodzinski, P., Bergamaschi, A., Schmitt, B., David, C. & Lücke, A. (2012). R*ev. Sci. Instrum.* **83**, 103105.]

[Travis, J. & Kring, J. (2007). *LabVIEW for everyone : graphical programming made easy and fun. Kring, Jim. (3rd ed.)*. Upper Saddle River, NJ: Prentice Hall. ISBN 0131856723. OCLC 67361308, https://www.ni.com/en-us/solutions.html]

[von Hamos, L. (1932). Naturwiss. **20**, 705.]

[Waltz-Controls organization on GitHub, https://github.com/waltz-controls]

[Waltz-Controls RFC-1 Message, https://github.com/waltz-controls/rfc/tree/master/1]

[Waltz-Controls RFC-2, Magix client API, https://github.com/waltz-controls/rfc/tree/master/2]

[Waltz GUI platform, https://github.com/waltz-controls/waltz]

[waltz-magix-plugin, Waltz GUI Magix plugin source code, https://github.com/waltz-controls/waltz-magix-plugin]

[Webix UI library, https://webix.com]

[Webix UI widgets collection, https://docs.webix.com/desktop__components.html]

[Wolff, E. (2016) *Microservices: Flexible Software Architecture*. ISBN-13: 978-0134602417]

[XES device controllers' source code, https://github.com/Ingvord/axsis-xes]

[XES spectrometer GUI's source code, https://github.com/Ingvord/axsis-xes-gui]

[ZeoLearn (2018). Asynchronous Nature of JavaScript, https://medium.com/@zeolearn/asynchronous-nature-of-javascript-d7d18a71aa35]

[ZMQ intreproc, http://api.zeromq.org/2-1:zmq-ipc]

[ZMQ inprocess sockets, http://api.zeromq.org/2-1:zmq-inproc]