Technical Note

Not peer-reviewed version

Controls-kt, a Next Generation Control System

Alexander Nozik [*]

Posted Date: 29 August 2023

doi: 10.20944/preprints202308.1746.v2

Keywords: SCADA; scientific software; Kotlin language

Technical Note

Controls-kt, a Next Generation Control System

Alexander Nozik

Moscow Institute of Physics and Technology, 9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russian Federation; nozik.aa@mipt.ru

Abstract: In this article, we discuss problems that exist in modern SCADA systems and present a Controls-kt (formerly DataForge-controls) software development kit, that allows both to create a control system from scratch and provide integration with existing systems via Magix specification and connector.

Keywords: SCADA; scientific software; Kotlin language

1. Introduction

Supervisory control and data acquisition (SCADA) systems are a necessary part of any scientific experiment or industrial factory. Half of the century of development produced a lot of different SCADA systems with different organization logic and different technologies. Sadly, this abundance stalls the development of a new generation of those tools. Modern SCADA is required to be scaleable both up (with more devices and a more robust distributed communication structure) and down (using the same systems for small setups and laboratory prototypes). Modern information technology gives a lot of tools to work with data: communication, storage, analysis, visualization, etc., but most current SCADA systems use outdated communication protocols and system design, which do not allow to use modern tools to the full extent.

Another problem is that existing systems are not designed to work with each other. If two parts of a system are implemented using different SCADA systems, it is usually impossible to make them seamlessly work together

In this article, we discuss the design principles of modern SCADA systems and present the Controls-kt software development kit, designed to provide solutions to both scalability and interoperability issues. The system consists of two parts:

- Controls-kt [1] core and modules, that allows creating device servers (software modules that provide communication with a physical device) with minimal effort.
- An implementation of Magix specification [2] discussed in [3].

Those two parts together allow the creation of both compact centralized systems for small experiments and large distributed control systems. One of the key points of the design is that both Controls-kt and its Magix implementation rely on existing industrial technologies and are easily adaptable to other technologies as well. For example, Magix could be used to bring together different systems by creating a simple event loop adapter for each of them.

The crucial feature of both Controls-kt and Magix in comparison with widely used systems is that they utilize an asynchronous communication approach that is used in most modern web services instead of a synchronous approach used in device communication.

2. SCADA systems by scale and application

SCADA systems are used at different levels of measuring, technological and production processes. As a result, they differ in the principle of their operation, as well as in scale and structure. SCADA can be divided according to the organization of device connection into:

- Centralized.

- Distributed.

Centralized systems use one central control unit, often integrated with the database and the operator's workstation. This central node directly manages all processes. Distributed systems consist of several interacting nodes. In such systems, operator workstations are one of the possible nodes. Centralized systems are easier to set up and maintain and are used in small measuring systems as well as in compact productions. Centralized control systems are also used as on-board control systems for autonomous hardware systems (such as complex medical equipment).

Distributed systems are essential to keep large systems and processes running. The use of centralized systems at such facilities is impossible due to the limitations of the communicative and computational capabilities of individual nodes, as well as due to the need to ensure the modularity of systems.

One can also classify SCADA by the type of message delivery as either synchronous or asynchronous. Synchronous systems are based on client-server architecture. A client forms a request and sends it to a server. The client waits (blocks an execution thread) until the response is received. Modern technologies like coroutines allow to do waiting without blocking actual physical OS thread ([4]), but the important part is that a client could get the value only by requesting it first. Synchronous systems usually use peer-to-peer communication, meaning that one client communicates only with one server at a time. It means that to collect data from several servers, one needs to call them sequentially.

Asynchronous communication implies that the device independently sends messages about changes in its properties to the control unit or other devices and receives similar messages. At the same time, messages from different devices can come to the control node in any order. Asynchronous systems are characterized by the concept of a subscription rather than a request. A subscription happens when one participant in the communication subscribes to changes in a particular parameter and receives all messages about changes in this parameter without additional requests.

Synchronous systems are less demanding in terms of software design and are more common. In addition, synchronous communication is convenient for identifying breakdowns in the message delivery system (if the answer does not arrive within the timeout, then the communication is broken or the device is out of order). However, synchronous systems have some limitations. For example, they have a scaling limit. When a single node (for example, a user's workstation) works with a large number of nodes synchronously, one must wait for the results of all requests on all properties before repeating the polling cycle. Also, the use of more complex communication schemes (for example, when one device accesses another device bypassing the central node) can lead to the "locking" of the system, when several requests are waiting for each other to be executed so none of them could move forward.

One should notice that there may be different combinations of these variants. For example, centralized synchronous systems are used in small setups, centralized asynchronous systems are used in smart homes and onboard vehicle systems, and distributed synchronous systems are used in the control of large industries. Distributed asynchronous systems are widely used in web technologies and data processing, but so far they are rarely found in SCADA.

3. SCADA architecture

SCADA communication with a device could be done in two ways:

- Communicate with devices directly (using the protocol provided by the device).
- Interact with devices through an intermediate hardware link in the face of a programmable logic controller (or PLC).

Most of the systems support both modes. They could communicate with PLC with PLC-specific protocols and with devices using device-specific protocols. Each device or PLC type requires a specific software module, which is used for communication. Those modules are frequently called Device Servers.

A PLC is a hardware solution that allows you to connect to devices that read information and control the process (so-called I/O modules) and to a higher-level process control system on the one hand. PLCs solve two key tasks:

- Standardization and unification of communication with I/O modules. This feature is important because device manufacturers use different specifications and communication protocols. Being able to simply plug these devices into a PLC without having to create additional device drivers is important for the industry.
- The PLC can be set up with simple process control instructions that can operate autonomously even in the event of an accident in the control system ([5]).

The use of PLCs is justified in systems that control hazardous production, as well as in systems where control is carried out in real-time and it is important to accurately observe the time of technological processes (with an accuracy of 50 ms or better).

The use of PLCs is not justified in laboratory setups, as well as in systems where complex instruments with integrated controllers are used.

One of the major problems of SCADA systems is the complexity of device server development. This is the reason, SCADA engineers tend to replace software solutions with much more expensive hardware.

4. Device server design

The device server is the primary abstraction used in defining the interaction between software and hardware. All modern open systems (TANGO, DOOCS, etc) define it in more or less the same terms:

- Device contains some properties (called attributes in TANGO).
- Properties have fixed types: numbers, strings, booleans, and lists of all of the above. Usually, there is also an option to define custom types as combinations of basic types.
- Properties could be writeable or read-only.
- Some systems ([6]) also allow to subscribe on properties changes. In this case, the device reports updates at fixed intervals or following device logic.
- In some systems it is also possible to define actions (or commands), which have an argument and a result (both use types that are available for properties).

The important difference between the subscription model compared to property model is that in the property model client decides at which rate to read the property. In a subscription model, the decision about the rate lies with the device server itself. The device server also does not report values if they have not changed, which could limit the total number of messages.

4.1. *Device property design*

Let's look into device property in more detail. To properly understand its dynamics we must separate three different facets of a property (see Figure 1:

- Remote state. The value that is sent via network and which is observed in monitor software and database.
- Logical state. The actual value is passed to a property setter or read from the property getter.
- Physical state. The state of a physical device that corresponds to a given property value.

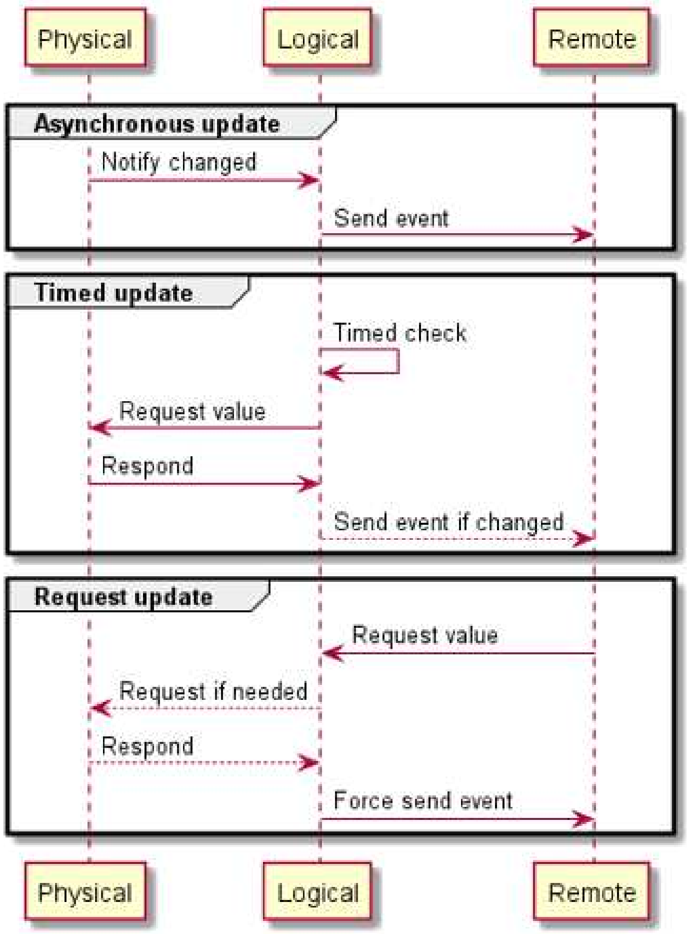

Figure 1. Levels of property definition: remote, logical and physical, and typical communication procedures.

All those states could diverge. For example, let's assume that we have a motor with a PID regulator. A remote user sends a target value to the device. The value could be outside of the possible range. In this case, either the logical value will be set to the appropriate threshold or the set will fail. In both cases, remote value will have a state, different from logical value. Even if the event is propagated back to the caller, there will be differences during that process. Now, logical value could diverge from physical value either because of limited precision or because of physical device inertia. Also, it is possible that physical value changes in time so the value that is "remembered" by the device server is different from the actual value.

The general way to treat the divergence problem is to make all calls synchronous. It means that when a remote process calls the device server, it waits for the device server's confirmation before confirming the change. The device server in turn waits for the device's physical value to stabilize before confirming the change to the remote caller. This approach imposes a lot of limitations. The main one is that remote must wait for the whole process to finish and change states sequentially for all devices and all properties. Another limitation is that the remote caller must know all properties it wants to request in advance. It is a serious limiting factor for database connections and visualization.

A solution to the problem that does not rely on synchronization is to allow different states to diverge and turn the control system into a fully-fledged distributed system. Such systems are studied in computer science ([7]). Terms of eventual consistency could not be fully applied to devices because the device's state changes on its own in time. Even if the user stops sending signals, the physical state will change over time. Still, one can ensure that all changes in the logical state are eventually sent to remote listeners.

To make the model more transparent, it is better to discuss changes in terms of events rather than the state (asynchronous communication). Each state change event contains the following fields:

- the name of the property,
- the time of the change,
- the value of the property after the change.

It is also possible to store the value before the change to establish proper history, but it significantly increases the volume of communication.

The external read or write operation also could be executed by receiving appropriate events. The difference between asynchronous request-response communication and event-based requests is that event is always sent in fire-and-forget mode, meaning that the caller does not expect immediate response, it triggers the generation of an opposing event, that could or could not happen.

Subscription capabilities ([8]) are available in most modern SCADA systems ([9,10]), but they do not embrace the event-based model fully.

4.2. Conversion between asynchronous and synchronous modes

Asynchronous communication does not cover all possible usage scenarios. One important case is synchronous communication between the device server and the device. Many devices do not support asynchronous messaging, so they need to be queried synchronously. In this case, the device server generates or receives a message to read or write the state, then creates a waiting task that blocks a thread and waits for the device response. The task then sends a message with the result of the measurement. It usually suffices to have only one thread per synchronous device (separate requests are processed synchronously) so the blocking is occurring only once in the whole system.

Another case is when a remote user or service wants to read or write state synchronously. In this case, the service sends a request to read or write a state and creates a thread that suspends until the client receives a message with a specific property value. One must note that a message with a property value is not a necessary response to the request message, it could be a part of a regular update. Additional limitations could be imposed to avoid using stale property value updates. Using coroutines ([4,11]) in this scenario could significantly limit the resource requirements because coroutines do not need to block a thread.

Figure 2 shows the scheme for converting asynchronous events to synchronous calls and back.

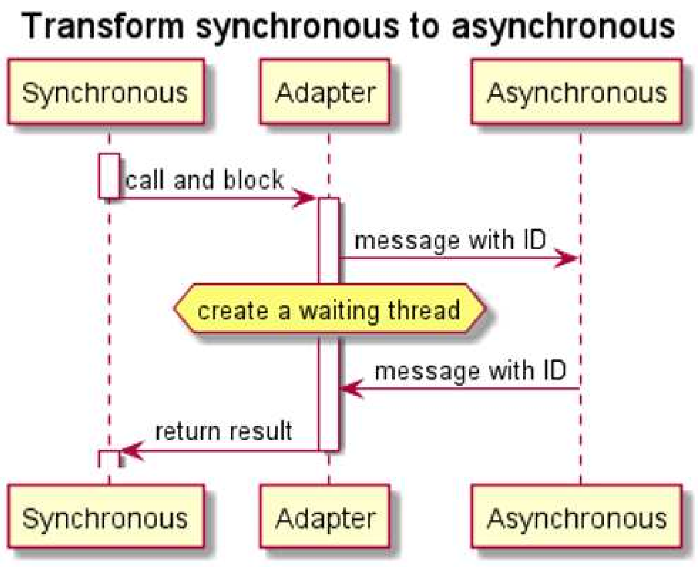

Figure 2. Conversion of synchronous request to an asynchronous one.

4.3. *Property value types and communication*

One of the major problems in SCADA system design is defining a way to propagate custom property values between remote nodes (this problem is not relevant for fully centralized systems). Most known systems use schema-based communication protocols:

- TANGO-controls uses CORBA industrial standard [12].
- DOOCS uses Sun ONC protocol [13].
- EPICS uses its unique protocol.
- Simatic WinCC primarily uses different versions of the OPC ecosystem, which also has an option of defining custom schemas.

The schema-based protocol assumes that both communication participants have access to a message schema. The message is serialized and deserialized according to this schema. Schema allows the production of a more compact binary representation (because it does not require field names, only their index), it also allows to validate input and work with objects in remote procedure call (RPC) mode. This idea was especially appealing in the 2000s when it allowed to apply general OOP techniques to remote objects.

Schema-based protocol has a lot of problems though. Especially when used in SCADA systems. Most of the schema-based approach benefits could be realized if all communication participants have access to all schemas in compile-time. This is not the case for distributed SCADA, where new devices could be added after some parts of the systems are up and running. The second problem is that schema negotiations require a central schema repository (so all participants have access to all schemas). Negotiating schema synchronization adds a lot of complexity to the system (it could be seen in TANGO-controls and DOOCS). In the end even after schema negotiation, one needs to access resulting deserialized objects via field names because objects are not accessible in compile-time.

An alternative is a tree-based property representation, where an object is represented as a tree of basic value types (like numbers, strings, booleans, and arrays).

The common argument that binary schema-based protocols are much faster and more compact than schema-less protocols is not quite correct. The direct measurement shows that there is no clear advantage of schema-based protocols in this regard: [14]. Using schema-less binary protocol also does not improve performance much over string-based formats (but impacts readability): [15].

Unlike regular in-memory structures, value trees could be traversed and compared without the use of reflection. It is also possible to compute a difference between two value-tree structures.

5. Controls-kt framework

Controls-kt framework ([1]) is designed to simplify the creation of devices servers and to provide ways to implement event-based (asynchronous) communicate between devices and with services like visualization and data storage.

The initial prototype was developed for Troitsk nu-mass experiment ([16]). The industrial systems could not be used in the experiment because it involved unique device and workflow. Controls-kt (called DataForge-control at that moment) was planned to be used in Baby-IAXO experiment to integrate custom devices with DOOCS system ([17]).

Controls-kt is written in Kotlin language ([18]). It allows to use modern language, build tools and libraries to easily create and maintain device servers. A unique advantage of Kotlin language is the Kotlin-multiplatform technology, that allows creating JVM, JS, Wasm distributions as well as platfrom-specific native binaries. So different parts of the SCADA system could be integrated with different platforms, using the same language. Kotlin is used a lot in the web backend development, so there are a lot of industrial tools for working with events and data streams available.

5.1. *Device API overview*

The simplified version of device API used in Controls-kt showed in the following code:

```
public interface Device{

    /**
     * Read the physical state of property and
     * update/push notifications if needed.
     */
    public suspend fun readProperty(propertyName: String): Meta

    /**
     * Get the logical state of property or return null if it is invalid
     */
    public fun getProperty(propertyName: String): Meta?

    /**
     * Invalidate property (set logical state to invalid)
     */
    public suspend fun invalidate(propertyName: String)

    /**
     * Set property [value] for a property with name [propertyName].
     */
    public suspend fun writeProperty(propertyName: String, value: Meta)

    /**
     * A subscription-based [Flow] of [DeviceMessage] provided by device.
     * The flow is guaranteed to be readable multiple times.
     */
    public val messageFlow: Flow<DeviceMessage>

    /**
     * Send an action request and suspend caller
     * while request is being processed.
     * Could return null if request does not return a meaningful answer.
     */
    public suspend fun execute(
      actionName: String,
      argument: Meta? = null
    ): Meta?
}
```

Meta is a read-only value tree (see [19] and [20]) that allows representing single values (numbers, strings, booleans), arrays, and composite structures. Meta objects could be serialized to JSON, XML, or CBOR (other tree-like schema-less formats are also possible). Meta could be accessed as a dynamic tree. DataForge also has tools to provide soft schema object, that allows to read and write Meta in a type-safe way. Soft schema is not required for serialization so it is not necessary to synchronize across nodes.

The difference between `readProperty` and `getProperty` is that `read` forces the device to synchronize the physical state (and update the logical state if synchronization was successful), whereas `get` returns the current logical value and `null` if it is not yet computed. `invalidate` method discards previously computed logical state so the next call to `getProperty` returns null, which forces to use `readProperty` to perform synchronization.

The messageFlow property returns a reactive stream ([21]) that in general contains not only property change events but also other device messages like lifecycle events (if they are defined). The device server could emit new events according to its internal logic. Unlike property-based subscriptions in TANGO-controls, Controls-kt shares all events in a single Flow with multiple subscriptions. A consumer could then filter and transform events using a filter-map-reduce methodology ([22]).

5.2. Device server implementation

One of the hardest things in mastering the SCADA system is to create a device server. For example, the TANGO server requires extensive infrastructure (configuration database, history database) as well as extensive study of its concepts ([23]). Currently, its generation is simplified via the POGO tool ([24]). Examples of device servers could be found in [25].

In Controls-kt, one can directly implement and interface mentioned above, or use a helper device definition via specification. The specification is a singleton template that defines properties and actions without storing resources associated with the physical device like connection details. It allows to implement several devices with the same external API but different connection details.

```
/**
 * An connection-agnostic interface to access the physical state
 */
interface IDemoDevice: Device {
    var timeScaleState: Double
    var sinScaleState: Double
    var cosScaleState: Double

    fun time(): Instant = Instant.now()
    fun sinValue(): Double
    fun cosValue(): Double
}

/**
 * A specification that defines properties
 */
object DemoDeviceSpec : DeviceSpec<IDemoDevice>() {

    val timeScale by mutableProperty(
      MetaConverter.double,
      IDemoDevice::timeScaleState
    )
    val sinScale by mutableProperty(
      MetaConverter.double,
      IDemoDevice::sinScaleState
    )
    val cosScale by mutableProperty(
      MetaConverter.double,
      IDemoDevice::cosScaleState
    )

    val sin by doubleProperty(read = IDemoDevice::sinValue)
    val cos by doubleProperty(read = IDemoDevice::cosValue)

    val resetScale by unitAction {
        write(timeScale, 5000.0)
        write(sinScale, 1.0)
        write(cosScale, 1.0)
    }

    override suspend fun DemoDevice.onOpen() {
        doRecurring(50.milliseconds) {
            read(sin)
            read(cos)
        }
    }
}

/**
 * An implementation of device for specific connection (or virtual)
 */
class DemoDevice(context: Context, meta: Meta) :
  DeviceBySpec<IDemoDevice>(Companion, context, meta), IDemoDevice {

    override fun sinValue(): Double = sin(
      time().toEpochMilli().toDouble() / timeScaleState
    ) * sinScaleState

    override fun cosValue(): Double = cos(
      time().toEpochMilli().toDouble() / timeScaleState
    ) * cosScaleState
    /*...*/
}
```

`DemoDeviceSpec` is a singleton object, that defines the following specifications:

- Three mutable properties: `timeScale`, `sinScale` and `cosScale`. Mutable properties could be read and written. Meta converter ensures that one could use a type-safe way to access them.
- Two mutable properties: `sin` and `cos` that could not be written, but could only be read.
- An action `resetScale` that has no arguments or outputs and just resets the scale.

It also defines an initialization logic that forces re-read of `sin` and `cos` values each 50 ms. This behavior does not depend on a specific implementation of `sinValue` and `cosValue` readers. The `DemoDevice` class implements those properties (in the current implementation, it just computes the sine and cosine of current time in milliseconds, adjusted by scale). Another implementation could do the same by requesting values from a device.

The reading and writing of properties could be done like:

```
demoDevice.read(sin)
demoDevice.write(sinScale, 2.0)
```

5.3. Implementation of device communication protocols in Controls-kt

One of the most complicated tasks in developing a device server is to support a communication protocol provided by a device. In Controls-kt it is supported by dedicated modules to support the most frequently used protocols:

- Direct serial interface. The Controls-serial module provides synchronous and asynchronous phrase-based communication with a device.

- Network interfaces. Controls-kt supports synchronous and asynchronous communication with raw TCP and UDP protocols. There are two implementations of such communication: one uses JVM internal implementations and is available only on JVM. Another one uses the Ktor-network module ([26]) and will be available also for native compilation in the future.
- Modbus. Controls-Modbus module supports Modbus-RTU and Modbus-TCP protocols via `j2mod` library ([27]). Modbus is the most used protocol for low-level communication. The Controls-kt device could be used both as a Modbus-master (client) and Modbus-slave (server). The interaction is based on a type-safe registry mapping.
- OPC-UA. Controls-kt supports OPC-UA client and server modes with Eclipse Milo ([28]). In a server mode Controls-kt devices server connects to a device and translated OPC-UA properties to DataForge Meta objects. In server mode, it exposes the Controls-kt device server as an OPC-UA device.

5.4. A real-life example of stand-alone device server

One of the current industrial applications of the Controls-kt framework is the software for the adjustment of civil ship-based satellite antennas. The purpose of the software is to perform automatic adjustment of an antenna to compensate for the ship movement and automatically search for the proper satellite (the work is performed in MIPT under the research center for telecommunication).

The antenna control structure is the following (see Figure 3):

- Antenna motors are controlled directly with a PCB based on an STM32 processor ([29]). The embedded software is written in C++ and performs basic movement and gyro-stabilization tasks.
- STM32 board is connected via 2-wire RS485 interface to Raspberry Pi 4 microcomputer ([30]). The communication is done with Modbus-RTU protocol.
- The Controls-kt software runs on Raspberry Pi JVM and consists of several modules:
  - A Modbus registry that allows to access Modbus properties by name instead of register number and allows storing complex objects in Modbus registers.
  - A device server that wraps the registry (the tools for creating this server are already included in Controls-kt SDK).
  - A tuner controller that allows measuring the level of the signal for automatic antenna position adjustment.
  - Visualization server based on VisionForge framework ([31]).

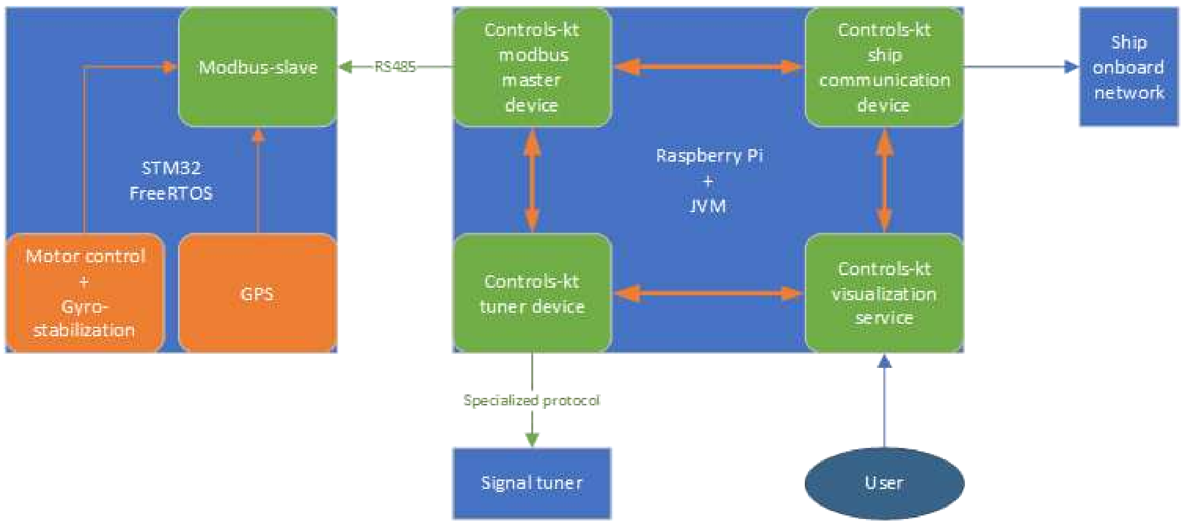

Figure 3. The architecture of two-part communication with ship-based satellite antenna. Fast operation is programmed on an STM32 processor with a FreeRTOS operation system. The user logic is implemented in several Controls-kt devices that communicate with each other.

Figure Figure 4 shows a visualization dashboard for antenna tuning. The plot shows the signal versus time. The 3D visualization shows the antenna direction. Controls allow to direct the antenna manually.

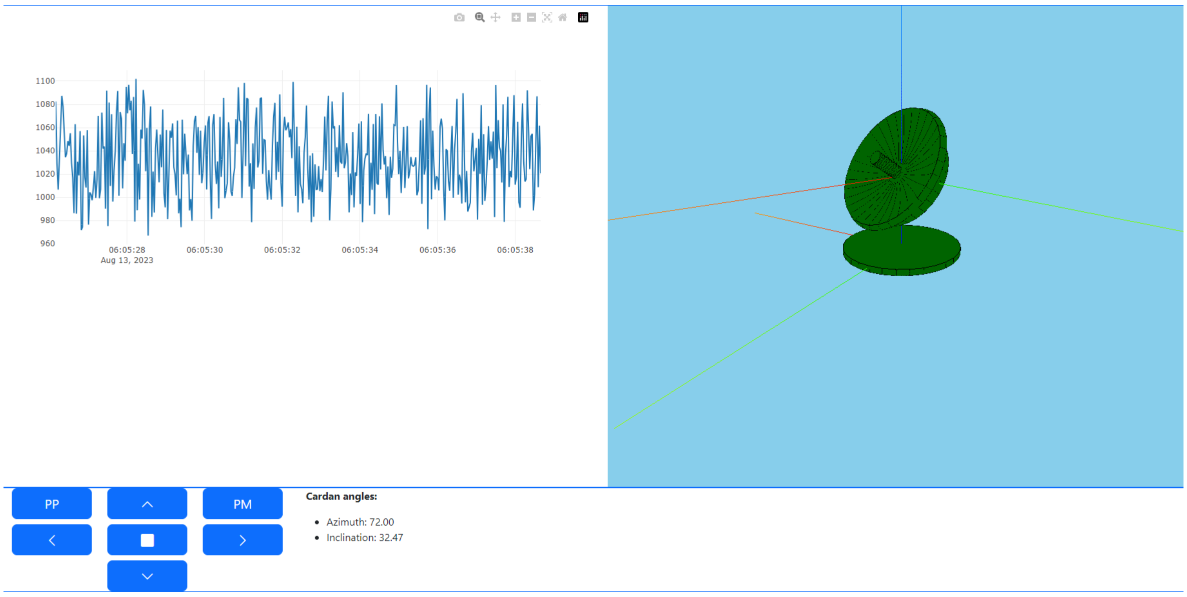

Figure 4. The dashboard for the antenna controls program. It shows the 3D model of the antenna, the plot of signal strength versus time, and simple manual controls for antenna position.

6. Distributed system implementation with Magix

In the previous section, we discussed the design for a Device server and the separation between device specification and its implementation on specific hardware. But the device server alone does not allow to create a distributed control system (DCS). The distribution is done via Magix specification ([2]), its design and applications are discussed in [3]. The Magix specification is based on reactive streams principles and consists of an interface for the so-called endpoint:

```
public interface MagixEndpoint {
    public fun subscribe(filter: MagixMessageFilter): Flow<MagixMessage>

    public suspend fun broadcast(message: MagixMessage)
}
```

The `subscribe` method allows to subscribe receive all messages from all sources that correspond to the given filter criterion. The `broadcast` method allows sending a message to all devices and services, attached to the event loop. The `MagixMessage` is generally represented by JSON string (see [2] for details).

```
{
  "id":"string|number[optional, but desired]",
  "parentId": "string|number[optional]",
  "sourceEndpoint":"string[required]",
  "format":"string[optional, but desired]",
  "targetEndpoint":"string[optional]",
  "user":"object[optional]",
  "payload":"object[optional]"
}
```

The message format does not specify the data that should be transmitted. The data is stored in the `payload` field and could have different structures for different endpoint types. The only mandatory field is the `sourceEndpoint` field that shows the origin endpoint of the message.

It is important to note that the communication with Magix does not follow client-server conventions. Endpoints do not connect directly. Instead, they utilize a publish-subscribe pattern ([8]). In this architecture, each endpoint communicates only with a central bus/event loop and can both asynchronously send and receive data. There is several advantages to this approach:

- Network discovery, which is one of the most complicated parts of traditional SCADA systems is not required. The endpoint needs only the event loop connection address to run.
- The event loop is highly scalable. One can use modern industrial message brokers like Apache Kafka ([32]), RabbitMQ ([33]), or ZMQ ([34]) to handle any amount of workload.
- A subscriber does not have to know all properties and devices it listens to. For example, the database can record all events from all sources without the discovery of all of the devices.
- Asynchronous communication allows to avoid deadlocks and other synchronization issues ([35]).

In the case of Controls-kt, the Magix message payload is a JSON representation of `DeviceMessage` mentioned above.

Asynchronous communication has two major deficiencies:

- Lack of delivery guarantees. Since messages are sent in asynchronous mode, the sender does not receive a notification if the message is not delivered. The communication protocol usually guarantees that the message is delivered to the event loop, but there are no simple ways to determine if the message is received by the specific subscriber.
- Multiple replications of messages. The message is delivered to all subscribers, who have filters that accept the message.

To provide delivery guarantees, one needs to employ additional services like heartbeat monitor ([36]) and watchdog([37]). Those services are currently not the part of official Magix specification, but it is under development.

The replication could be limited by using proper filters. Another way is to segment the loop by separating different parts of the loop with so-called portals, which are made of two Magix endpoints connected to different loops with a filter between them

```kotlin
public fun CoroutineScope.launchMagixPortal(
    firstEndpoint: MagixEndpoint,
    secondEndpoint: MagixEndpoint,
    forwardFilter: MagixMessageFilter = MagixMessageFilter.ALL,
    backwardFilter: MagixMessageFilter = MagixMessageFilter.ALL,
): Job = launch {
    firstEndpoint.subscribe(forwardFilter).onEach {
        secondEndpoint.broadcast(it)
    }.launchIn(this)

    secondEndpoint.subscribe(backwardFilter).onEach {
        firstEndpoint.broadcast(it)
    }.launchIn(this)
}
```

6.1. Magix loop connections implementations

Magix loop is not limited by the specification to use specific protocols. Moreover, the default Magix loop implementation provided with Controls-kt (which is not intended to be used for large-scale applications) uses several protocols. Currently, there are the following Magix loop client implementations:

- RSocket ([38]). There are both endpoint and loop-side connections for RSocket. The protocol is recommended for small and medium systems.
- ZMQ ([34,39]). The ZMQ protocol is much harder to handle, but it is optimized for maximum performance and zero-copy communication. There are both endpoint and loop-side implementations.
- HTTP/2 connection (using SSE for backward communication). This mode is implemented in [3], but is not recommended for larger systems.
- RabbitMQ (AMQP protocol). Controls-kt supports only endpoints-side integration. The loop is the RabbitMQ itself.
- MQTT. Controls-kt supports only endpoints-side integration.

It is also planned to add Apache Kafka ([32]) integration for large-scale industrial applications. The principal scheme of Magix loop plugin connections is presented at Figure 5.

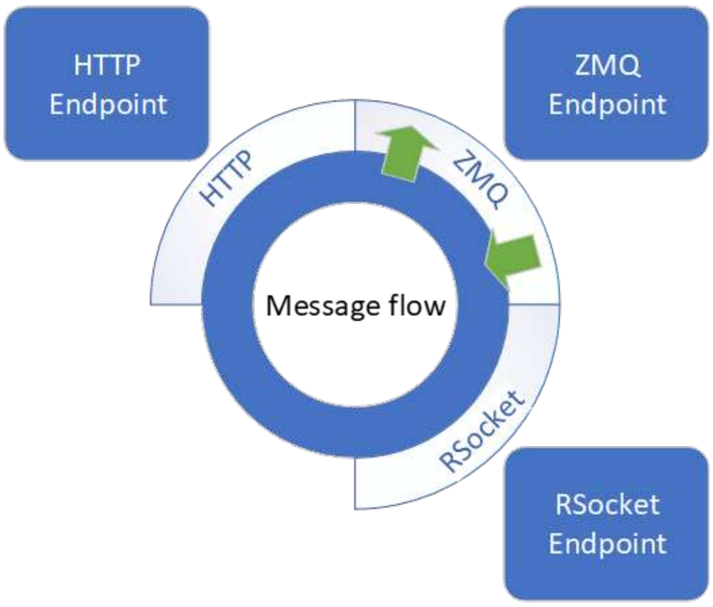

Figure 5. The principal scheme of Magix loop design and plugins.

6.2. *Inter-system connections with Magix*

One of the primary features of Magix specification is that it allows interconnecting different systems. An example of such interconnection is shown in [3]. A system has an adapter to connect to a Magix endpoint and the target system. Magix does not discriminate between clients and servers, but target systems usually do. So one needs to implement a client-side connector (Magix connects to a system as a client) and a much harder server-side connection (Magix loop-based device is connected to a system as a device server). Currently, there are implementations for Controls-kt, TANGO-controls, and DOOCS. It is planned to add client-side support for EPICS and Symatic WinCC in the future. The adapter transforms system-specific message protocols into JSON and vice versa. To connect two systems one needs only to implement a conversion tool that changes the payload to be compatible with a target system. The tool could be implemented on the endpoint side (transform it in place, after the message is received), or as a service that transforms all messages with a given format to a new format and sends them back into the loop. The service is less reliable but allows one to connect different systems to a single loop without knowing about each other.

7. Visualization tools

Visualization tools are considered an important part of the SCADA system. Visualization systems could be divided into two groups:

- No-code or low-code tools to design connections and system interactions. This kind of system is frequently incorrectly called HMI (human-machine interface).
- Dashboards for monitoring and controlling specific devices and their measurements.

Neither Controls-kt nor Magix has a dedicated no-code system for the graphical design of connections at the moment. Designing such a system is greatly time-consuming and requires tuning to work with a specific set of devices. Instead, the ideology recommended to be used with Controls-kt and Magix is to utilize modules' flexibility to easily create custom solutions for specific cases. Some examples could be found in the Controls-kt repository ([1]).

For dashboards, there are several different solutions:

- Waltz-controls framework ([40]) was specifically designed to work with Magix. It allows to design of client-side dashboards using a variety of widgets.
- Plotly-kt framework ([41]) that allows both client-side and server-side dashboard management.
- VisionForge framework ([31]) that includes Plotly alongside tables, 3D rendering and other capabilities (Figure 4 is created using VisionForge framework).
- Any other framework used for dashboards like Plotly-dash, Grafana, etc.

8. Performance tests for Magix and Controls-kt

One of the important requirements for the SCADA system is the ability to support many connections with a high event rate. Performance usually could not be estimated from the software code, so to measure performance we created a stand-alone demonstration module (the code for the testing module could be found at [42]). The testing module consists of the following components:

- A virtual device specification with a single floating point value property, which gives a random value on each read. A random value is used because the algorithm does not send an event if the new value is the same as the one before read. The device is configured to re-read the value each 10 ms, which is considered a very good time sensitivity for DCS (the usual requirements start from 50 ms).
- A bootstrap procedure to load 100 such devices simultaneously.
- A Magix event loop, supports RSocket-TCP, RSocket-WS, and ZMQ protocols. The connection with devices is done via RSocket-TCP protocol.
- A visualization service based on Plotly-kt, which is connected to the Magix loop with RSocket-WS. The Plotly-kt uses WebSocket inside to communicate between the backend and frontend. The visualization service receives all events from all devices, computes the difference between the event origin time and the time it has been received (latency), then shows the maximum difference accumulated in 200 ms time for each device.

All modules run inside the same JVM instance.

Figure 6 shows the typical latency for all devices which is about 3-4 ms. The latency occasionally rises to 15 ms, which is still very good for DCS communication rates.

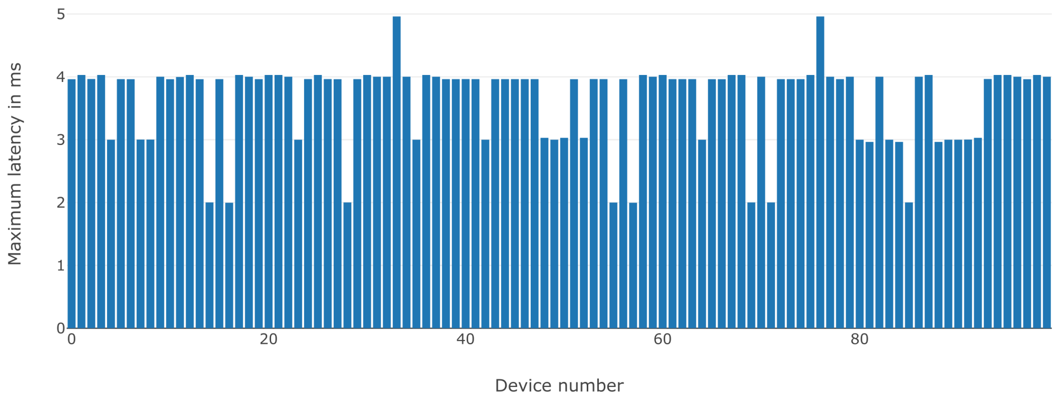

Figure 6. The maximum latency for 100 devices simultaneously connected to a local event loop with a 100 Hz event rate each.

Figure 7 shows the CPU load for the whole setup (AMD Ryzen 5 3600 processor) in time. The CPU load does not exceed 10% at each moment and is much lower on average. The memory consumption does not exceed 300 Mb (even with JVM overhead) and could be improved further. Such low resource requirements allow running the framework on microcomputers. We tested the same demonstrator on Raspberry Pi 4. The relative CPU consumption is much higher and has risen to 50-70%, but is still within the capabilities of the device.

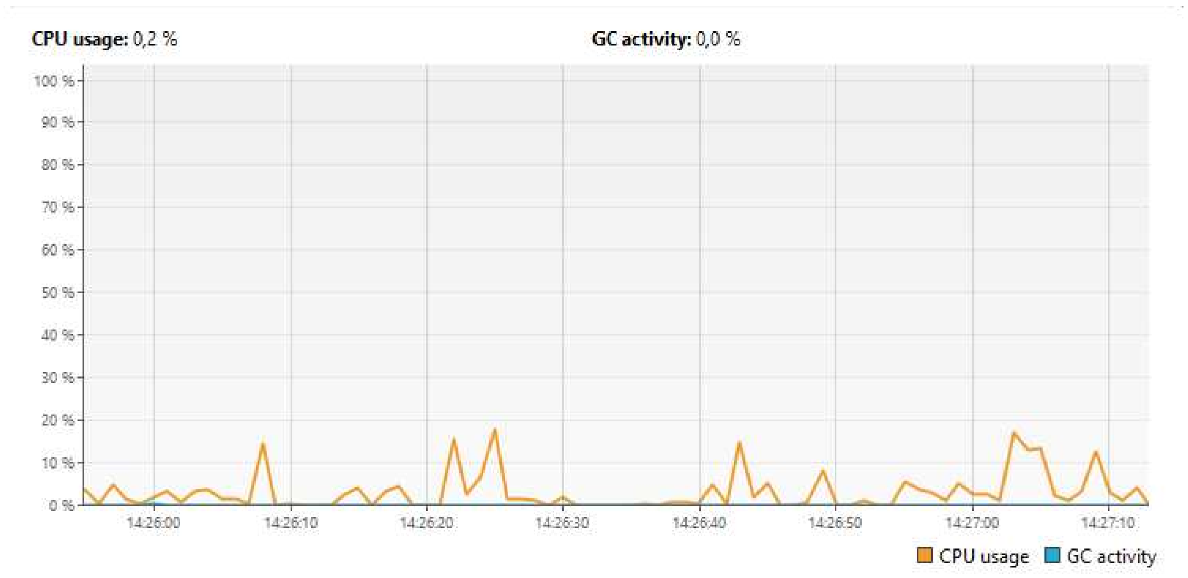

Figure 7. The CPU load in time for a system of 100 devices with a 100 Hz event rate, loop service with plugins, and visualization service

9. Conclusion

Controls-kt and Magix provide building blocks to create a SCADA system for both small-scale experiments with custom hardware, and industrial setup. For smaller experiments, Controls-kt device servers could be used in a stand-alone mode with local data storage (some data storage integrations are already provided in the Controls-kt repository. For distributed systems, the Magix event loop could be used both to connect separate Controls-kt nodes and to integrate it with different systems.

On the one hand, current integrations allow us to communicate with existing SCADA systems, on the other - asynchronous event-based communication requires a change of system design paradigm. It forces the designer to think more about delivery guarantees and the ordering of messaging in time-critical cases but allows much more flexibility, including online data processing and visualization services and integration with cloud platforms.

Right now different parts of the Controls-kt framework are being tested in the industrial environment. Research projects could benefit from it much more because research frequently requires much more flexibility and custom devices and visualization tools. What is more important is that open standards and cross-language and cross-platform support allow to significantly increase the effectiveness of SCADA system developers and lower the requirements for dedicated engineers.

One of the more frequent arguments for using schema-based binary protocols like CORBA is the performance limitations. Simple tests we performed show that there is no significant impact on performance when using JSON-based protocol for communication, yet it increases readability and more importantly, an ability to log and debug the communication. One could even use well-developed industrial tools for that.

This development of the framework received no external funding. Some development tools was provided free of charge by the JetBrains company. The development of antenna controller in section 5.4 was supported by the Federal Academic Leadership Program Priority 2030.

I would like to thank Igor Khokhriakov for his role in the development of the Magix standard, Olga Merkulova for pushing the work forward; Peter Klimai and Victor Samoilov for contributing directly to Controls-kt; all other kscience project contributors for their work.

I would also like to thank JetBrains Research organization for supporting kscience open-source projects from 2019 to 2022.

References

1. Nozik, A.; Samoilov, V.; Klimai, P.; SPC-code. SciProgCentre/controls.kt: 0.2.0, 2023. https://doi.org/10.5281/zenodo.8276733.
2. Khokhriakov, I.; Nozik, A. Magix-RFC. https://github.com/waltz-controls/rfc, 2023. [Online; accessed 20-August-2023].
3. Khokhriakov, I.; Merkulova, O.; Nozik, A.; Fromme, P.; Mazalova, V. A novel solution for controlling hardware components of accelerators and beamlines. *Journal of Synchrotron Radiation* **2022**, *29*, 644–653. https://doi.org/10.1107/S1600577522002685.
4. Koval, N.; Alistarh, D.; Elizarov, R. Fast and Scalable Channels in Kotlin Coroutines, 2022, [arXiv:cs.DS/2211.04986].
5. Wikipedia contributors. IEC 61131-3 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=IEC_61131-3&oldid=1161871613, 2023. [Online; accessed 23-August-2023].
6. Götz, A.; et al. TANGO Heads for Industry. In Proceedings of the 16th International Conference on Accelerator and Large Experimental Physics Control Systems, 2018, p. THCPL05. https://doi.org/10.18429/JACoW-ICALEPCS2017-THCPL05.
7. Burckhardt, S. Principles of Eventual Consistency. *Foundations and Trends® in Programming Languages* **2014**, *1*, 1–150. https://doi.org/10.1561/2500000011.
8. Lazidis, A.; Tsakos, K.; Petrakis, E.G. Publish–Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems. *Internet of Things* **2022**, *19*, 100538. https://doi.org/https://doi.org/10.1016/j.iot.2022.100538.
9. collaboration, T. TANGO PubSub. https://tango-controls.readthedocs.io/projects/rfc/en/latest/12/PubSub.html, 2023. [Online; accessed 20-August-2023].
10. Lange, R.; Johnson, A.N. ADDING FLEXIBLE SUBSCRIPTION OPTIONS TO EPICS*. 2011.
11. Breslav, A.; Elizarov, R.; Reshetnikov, V.; Erokhin, S.; Ryzhenkov, I.; Zharkov, D. Kotlin coroutines design proposal. https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md, 2023. [Online; accessed 20-August-2023].

12. Chaize, J.M.; Gotz, A.; Klotz, W.D.; Meyer, J.; Perez, M.; Taurel, E. TANGO - an object oriented control system based on CORBA. *Conf. Proc. C* **1999**, *991004*, 475–479.
13. Goloborodko, S.; Grygiel, G.; Hensler, O.; Kocharyan, V.; Rehlich, K.; Shevtsov, P.V. DOOCS : an Object Oriented Control System as the Integrating Part for the TTF Linac.
14. Sam, S. A performance study of protocols used in a print on demand server, 2015.
15. Viotti, J.C.; Kinderkhedia, M. A Benchmark of JSON-compatible Binary Serialization Specifications, 2022, [arXiv:cs.SE/2201.03051].
16. Nozik, A.; Pantuev, V. Direct Search for keV-Sterile Neutrino in Nuclear Decay. Troitsk Nu-Mass (Scientific Summary). *JETP Letters* **2019**, *110*, 91 – 96. https://doi.org/10.1134/S0021364019140042.
17. Abeln, A.; et al. Conceptual design of BabyIAXO, the intermediate stage towards the International Axion Observatory. *JHEP* **2021**, *05*, 137, [arXiv:physics.ins-det/2010.12076]. https://doi.org/10.1007/JHEP05(2021)137.
18. Nozik, A. Kotlin language for science and Kmath library. *AIP Conference Proceedings* **2019**, *2163*, 040004, [https://pubs.aip.org/aip/acp/article-pdf/doi/10.1063/1.5130103/14196335/040004_1_online.pdf]. https://doi.org/10.1063/1.5130103.
19. Nozik, A.; Postovalov, I.; Zelenyi, M. mipt-npm/dataforge-core: 0.5.2, 2021. https://doi.org/10.5281/zenodo.5742651.
20. Nozik, Alexander. DataForge: Modular platform for data storage and analysis. *EPJ Web Conf.* **2018**, *177*, 05003. https://doi.org/10.1051/epjconf/201817705003.
21. Stein, B.; Clapp, L.; Sridharan, M.; Chang, B.Y.E. Safe Stream-Based Programming with Refinement Types. In Proceedings of the Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, New York, NY, USA, 2018; ASE '18, p. 565–576. https://doi.org/10.1145/3238147.3238174.
22. Wikipedia contributors. MapReduce — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=MapReduce&oldid=1171373785, 2023. [Online; accessed 22-August-2023].
23. TANGO collaboration. TANGO device server. https://tango-controls.readthedocs.io/en/latest/development/device-api/device-server-writing.html, 2023. [Online; accessed 20-August-2023].
24. TANGO collaboration. TANGO pogo class generator. https://tango-controls.readthedocs.io/en/latest/tools-and-extensions/built-in/pogo/description.html, 2023. [Online; accessed 20-August-2023].
25. TANGO collaboration. TANGO device examples. https://gitlab.com/tango-controls/device-servers, 2023. [Online; accessed 20-August-2023].
26. Ktor contributors. Ktor raw sockets. https://ktor.io/docs/servers-raw-sockets.html, 2023. [Online; accessed 20-August-2023].
27. O'Hara, S. J2mod. https://github.com/steveohara/j2mod, 2023. [Online; accessed 20-August-2023].
28. Heron, K. Eclipse milo. https://projects.eclipse.org/projects/iot.milo, 2023. [Online; accessed 20-August-2023].
29. STMicroelectronics. STM32. https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html, 2023. [Online; accessed 20-August-2023].
30. Raspberry pi foundation. Raspberry pi 4 model b. https://www.raspberrypi.com/products/raspberry-pi-4-model-b, 2023. [Online; accessed 20-August-2023].
31. Nozik, A.; Dmitrieva, K.; Klimai, P.; Postovalov, I. mipt-npm/visionforge: 0.2.0, 2022. https://doi.org/10.5281/zenodo.5896056.
32. Narkhede, N.; Shapira, G.; Palino, T. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*, 1st ed.; O'Reilly Media, Inc., 2017.
33. Dobbelaere, P.; Esmaili, K.S. Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper. In Proceedings of the Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems, New York, NY, USA, 2017; DEBS '17, p. 227–238. https://doi.org/10.1145/3093742.3093908.
34. Wikipedia contributors. ZeroMQ — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=ZeroMQ&oldid=1170894095, 2023. [Online; accessed 22-August-2023].
35. Coulouris, G.; Dollimore, J.; Kindberg, T. *Distributed Systems: Concepts and Design (International Computer Science)*, 4th rev. ed. ed.; Addison-Wesley Longman, Amsterdam, 2005.

36. Mohd. Noor, A.S.; Mat Deris, M. Extended Heartbeat Mechanism for Fault Detection Service Methodology. In Proceedings of the Grid and Distributed Computing; Ślęzak, D.; Kim, T.h.; Yau, S.S.; Gervasi, O.; Kang, B.H., Eds., Berlin, Heidelberg, 2009; pp. 88–95.
37. Wikipedia contributors. Watchdog timer — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Watchdog_timer&oldid=1162902143, 2023. [Online; accessed 22-August-2023].
38. RSocket contributors. RSocket. https://rsocket.io/, 2023. [Online; accessed 20-August-2023].
39. Persaud, A.; Regis, M.J.; Stettler, M.W.; Vytla, V.K. Control Infrastructure for a Pulsed Ion Accelerator. *IEEE Transactions on Nuclear Science* **2016**, *63*, 2677–2681. https://doi.org/10.1109/TNS.2016.2594243.
40. Khokhriakov, I.; Merkulova, O.; Wilde, F. Waltz - A Platform for Tango Controls Web Applications. In Proceedings of the Proc. ICALEPCS'19. JACoW Publishing, Geneva, Switzerland, 08 2020, number 17 in International Conference on Accelerator and Large Experimental Physics Control Systems, pp. 1519–1524. https://doi.org/10.18429/JACoW-ICALEPCS2019-WESH3003, https://doi.org/10.18429/JACoW-ICALEPCS2019-WESH3003.
41. Nozik, A.; Samorodova, E.; ArtificialPB.; SPC-code.; Joseph-Hui.; Nieboer, T.; Henry, Z. SciProgCentre/plotly.kt: 0.6.0, 2023. https://doi.org/10.5281/zenodo.8195312.
42. Nozik, A. MassDevice.kt. https://github.com/SciProgCentre/controls.kt/blob/dev/demo/many-devices/src/main/kotlin/space/kscience/controls/demo/MassDevice.kt, 2023. [Online; accessed 20-August-2023].