Асинхронная система сбора данных. Сделай сам!

Александр Нозик, МФТИ

По плану

- Что такое система сбора данных?
  - Какие они бывают и какие с ними проблемы.
- Сделай сам:
  - Сервер устройства (на Kotlin).
  - Бонус – структуры метаданных.
  - Шина (асинхронная, разумеется, и на Kotlin).
  - Протокол коммуникации (тоже на Kotlin).
  - Поиграем в конструктор. Соберем систему сбора данных на вкус и цвет.
- Проблемы.

Обо мне

- Директор Центра Научного Программирования.
- К. ф.–м. н. по физике частиц.
- Преподаватель МФТИ.
- (Со-)руководитель московского KUG.

- https://sciprog.center/people/Nozik
- https://twitter.com/noraltavir
- https://t.me/noraltavir

Что такое система сбора данных?

Кто все эти люди?

- SCADA - Supervisory Control And Data Acquisition
- АСУ ТП - Автоматизированная система управления технологическим процессом
- САУ - Системы автоматического управления
- DCS - Distributed control system
- HMI – Human-Machine Interface
- PLC - Programmable logic controller (что оно тут делает?)

Какие они (не)бывают

Клиент-сервер

Распределенная

DOOCS

EPICS

Шина

Много закрытых HMI

TANGO controls

LabView

WinCC

Синхронная

Асинхронная

Монолитная

Задачи, которые они (не)решают

- Чтение и запись свойств устройства
- Обнаружение устройств в сети
- Распределение прав доступа к устройствам
- Централизованное хранение данных
- Локальное хранение данных
- Панели управления ← Все озабочены этим
- База данных конфигураций
- Интеграция с другими системами ← И никто этим

И в чем проблема?

- Системы строятся вокруг протоколов. Как только протоколы перестают поддерживаться, все становится плохо.
- Протоколы имеют ограниченную реализацию на разных языках программирования.
- Системы обнаружения сервисов и распределенные базы данных требуют сложной настройки.
- Инструментарий для создания серверов устройств очень сложный (часто система поддерживает только «свое» железо).

- В результате запуск простенького эксперимента с десятком датчиков требует профессиональной команды и года работы!

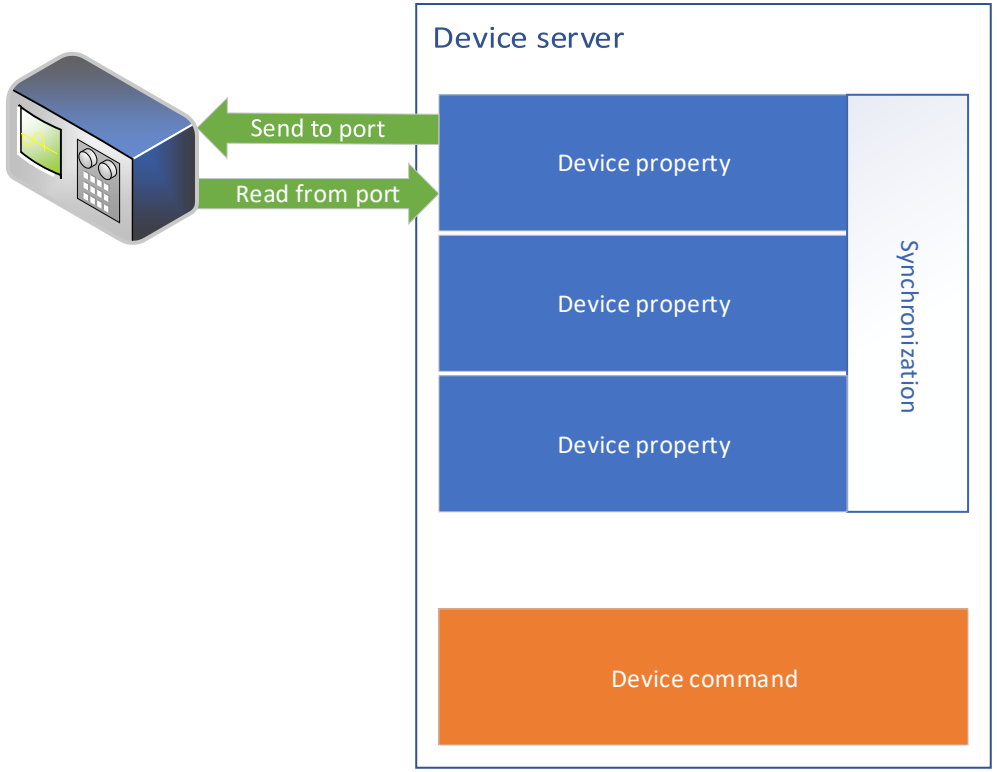
Photo by Rick Mason on Unsplash

Сделай сам!

https://github.com/SciProgCentre/controls.kt

Сервер устройства

Что такое сервер устройства



- Устройство состоит из свойств (типизированных).
- Свойство может быть доступно на чтение и запись.
- Свойство может быть связано с физическим состоянием прибора.
- Возможно наличие команд.

Интерфейс устройства

```
public interface Device : Closeable, ContextAware, CoroutineScope {
    public val meta: Meta
    public val propertyDescriptors: Collection<PropertyDescriptor>
    public val actionDescriptors: Collection<ActionDescriptor>
    public suspend fun readProperty(propertyName: String): Meta
    public fun getProperty(propertyName: String): Meta?
    public suspend fun invalidate(propertyName: String)
    public suspend fun writeProperty(propertyName: String, value: Meta)
    public val messageFlow: Flow<DeviceMessage>
    public suspend fun execute(action: String, argument: Meta? = null): Meta?
    public suspend fun open(): Unit
    override fun close(): Unit
}
```

Параметры устройства

Дескрипторы свойств

Дескрипторы действий

Асинхронное чтение (физического) свойства

Взять текущее (логическое) значение свойства

Сброс логического значения

Запись (физического) значения

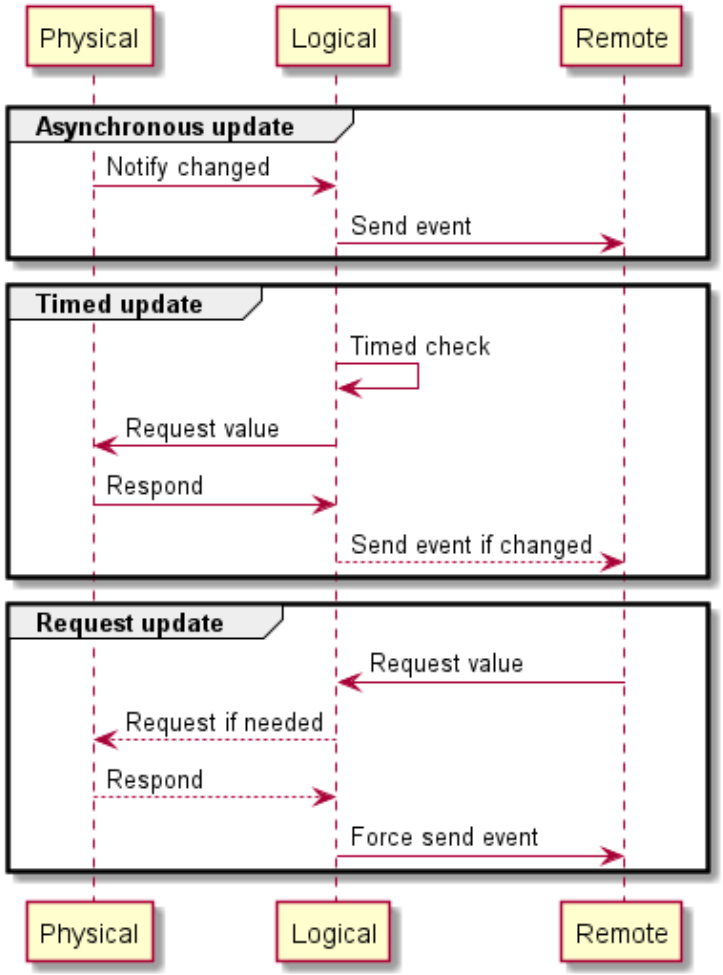
Подписка (многоразовая) на события

Стоп, стоп… назад

Какие такие физические и логические свойства?

Что такое Meta?
Почему там один и тот же тип.

Логический уровень свойства

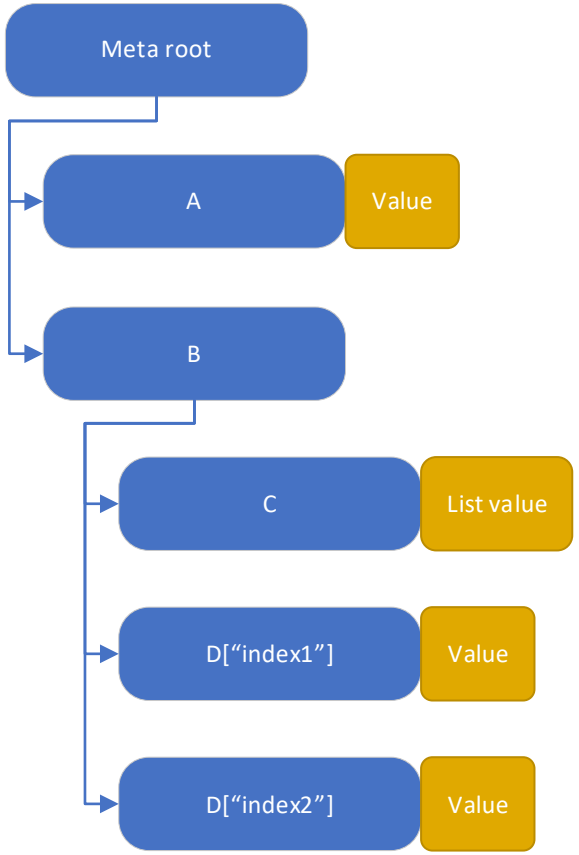


- Делаем запросы с той частотой, с которой удобно прибору.
- Храним последнее состояние в виде логического значения.
- Посылаем сигнал только когда логическое значение поменялось (экономим события).

Что за Meta?

Meta - дерево значений.

- Q: Почему не типизированный объект?
  A: Все равно придется обезтипливать при сериализазции.
- Q: Почему не JSON?
  A: JSON для текстового представления. А тут в памяти.



Добавим уюта

```
class DemoDevice(context: Context, meta: Meta) :
        DeviceBySpec<DemoDevice>(DemoDevice, context, meta) {
    private var timeScaleState = 5000.0
    private var sinScaleState = 1.0
    private var cosScaleState = 1.0

    companion object : DeviceSpec<DemoDevice>() {
        // register virtual properties based on actual object state
        val timeScale by mutableProperty(MetaConverter.double, DemoDevice::timeScaleState) {
            metaDescriptor {
                type(ValueType.NUMBER)
            }
            info = "Real to virtual time scale"
        }

        val sinScale by mutableProperty(MetaConverter.double, DemoDevice::sinScaleState)
        val cosScale by mutableProperty(MetaConverter.double, DemoDevice::cosScaleState)
```

Состояние виртуального прибора
или обращение к физическому состоянию

https://github.com/SciProgCentre/controls.kt/blob/dev/demo/all-things/src/main/kotlin/space/kscience/controls/demo/DemoDevice.kt

Добавим уюта

```
class DemoDevice(context: Context, meta: Meta) :
        DeviceBySpec<DemoDevice>(DemoDevice, context, meta) {
    private var timeScaleState = 5000.0
    private var sinScaleState = 1.0
    private var cosScaleState = 1.0
```

Спецификация устройства,
общая для всех устройств этого типа.
Не хранит состояния.

```
    companion object : DeviceSpec<DemoDevice>() {
        // register virtual properties based on actual object state
        val timeScale by mutableProperty(MetaConverter.double, DemoDevice::timeScaleState) {
            metaDescriptor {
                type(ValueType.NUMBER)
            }
            info = "Real to virtual time scale"
        }

        val sinScale by mutableProperty(MetaConverter.double, DemoDevice::sinScaleState)
        val cosScale by mutableProperty(MetaConverter.double, DemoDevice::cosScaleState)
```

https://github.com/SciProgCentre/controls.kt/blob/dev/demo/all-things/src/main/kotlin/space/kscience/controls/demo/DemoDevice.kt

Добавим уюта

```
class DemoDevice(context: Context, meta: Meta) :
        DeviceBySpec<DemoDevice>(DemoDevice, context, meta) {
    private var timeScaleState = 5000.0
    private var sinScaleState = 1.0
    private var cosScaleState = 1.0

    companion object : DeviceSpec<DemoDevice>() {
        // register virtual properties based on actual object state
        val timeScale by mutableProperty(MetaConverter.double, DemoDevice::timeScaleState) {
            metaDescriptor {
                type(ValueType.NUMBER)
            }
            info = "Real to virtual time scale"
        }

        val sinScale by mutableProperty(MetaConverter.double, DemoDevice::sinScaleState)
        val cosScale by mutableProperty(MetaConverter.double, DemoDevice::cosScaleState)
```

Регистрация изменяемых свойств. Хранение состояния в экземпляре

https://github.com/SciProgCentre/controls.kt/blob/dev/demo/all-things/src/main/kotlin/space/kscience/controls/demo/DemoDevice.kt

Добавим уюта

Похожая концепция используется в Plotly.kt:
https://www.youtube.com/live/8F0e_JaoUBU

```
val sin by doubleProperty {
    val time = Instant.now()
    kotlin.math.sin(time.toEpochMilli().toDouble() / timeScaleState) * sinScaleState
}
```

Чтение «физического» свойства

```
val cos by doubleProperty {
    val time = Instant.now()
    kotlin.math.cos(time.toEpochMilli().toDouble() / timeScaleState) * sinScaleState
}

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

https://github.com/SciProgCentre/controls.kt/blob/dev/demo/all-things/src/main/kotlin/space/kscience/controls/demo/DemoDevice.kt

Добавим уюта

```
val sin by doubleProperty {
    val time = Instant.now()
    kotlin.math.sin(time.toEpochMilli().toDouble() / timeScaleState) * sinScaleState
}

val cos by doubleProperty {
    val time = Instant.now()
    kotlin.math.cos(time.toEpochMilli().toDouble() / timeScaleState) * sinScaleState
}
```

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

Автоматически читаем свойство с той скоростью, с которой комфортно устройству.

https://github.com/SciProgCentre/controls.kt/blob/dev/demo/all-things/src/main/kotlin/space/kscience/controls/demo/DemoDevice.kt

Запись свойств

Входим в контекст устройства

В контексте устройства используем типо-безопасный дескриптор для записи значения.

```
button("Submit") {
    useMaxWidth = true
    action {
        controller.device?.run {
            launch {
                timeScale.write(timeScaleSlider.value)
                sinScale.write(xScaleSlider.value)
                cosScale.write(yScaleSlider.value)
            }
        }
    }
}
```

```
public suspend fun <T> WritableDevicePropertySpec<D, T>.write(value: T) {
    invalidate(name)
    write(self, value)
    //perform asynchronous read and update after write
    launch {
        read()
    }
}
```

Моделирование приборов

Photo by Atish Sewmangel on Unsplash

- Создание модели прибора – важный этап разработки прибора, его отладки и поддержки.
- Встраивание модели прибора в рабочую систему – важный этап отладки всей системы.

Сервера устройств: выводы

- Сервер устройства состоит из двух частей: спецификация и экземпляр.
- Спецификация содержит описания свойств.
- Экземпляр хранит логические свойства.
- Используем делегаты в Kotlin для того, чтобы объявлять и сразу регистрировать свойства.
- Делаем сервер устройства в 50 строк.

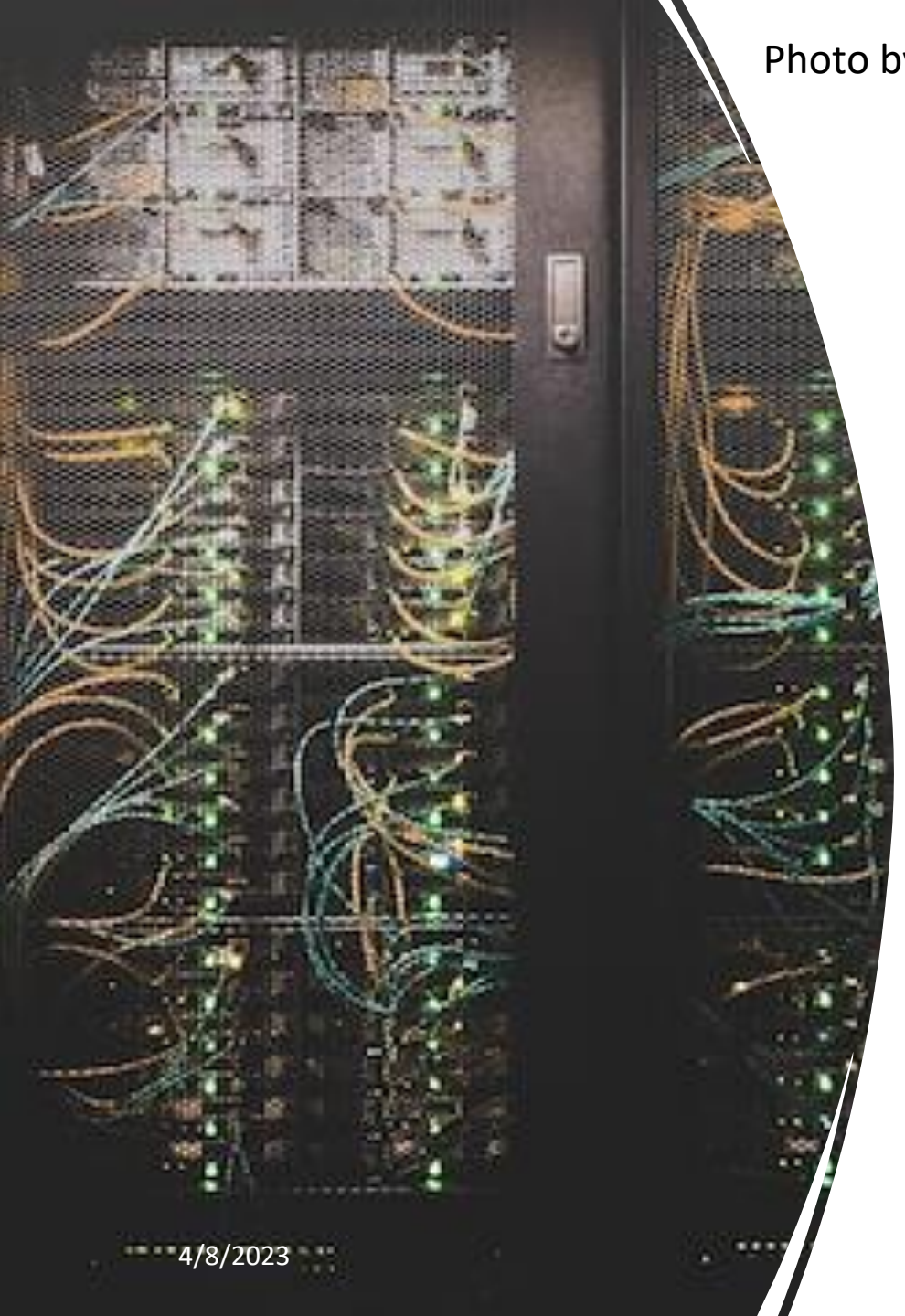
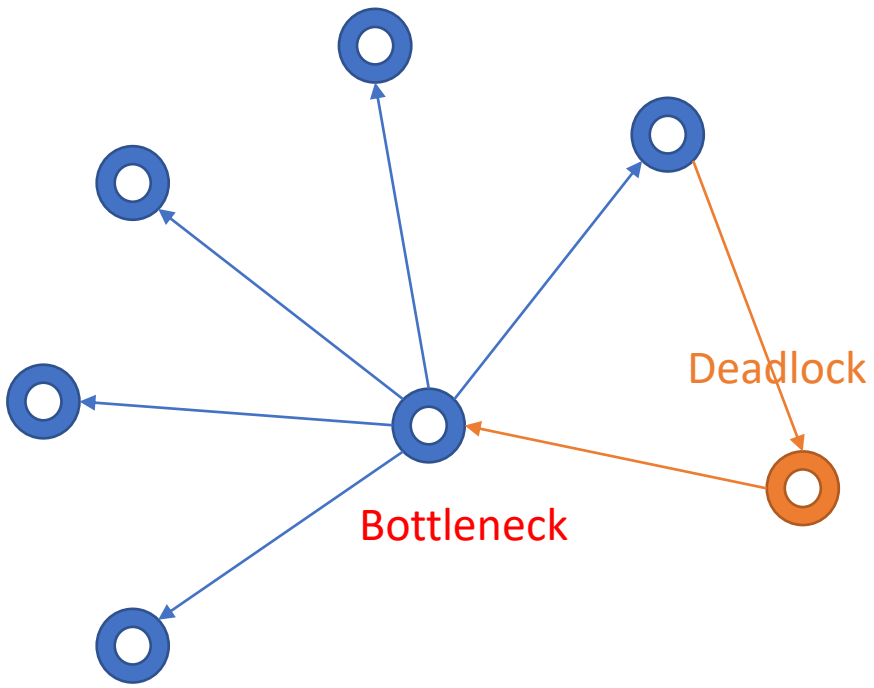


Photo by Taylor Vick on Unsplash

Шина

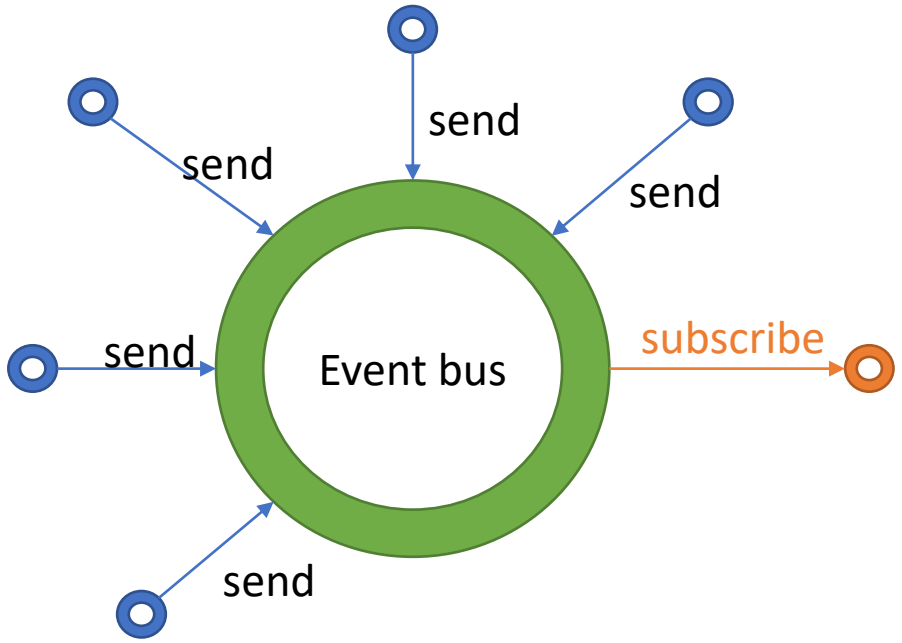
Шина? Какая шина?



- Большинство SCADA систем не используют шину данных.
- Используются синхронные P2P запросы.
- Клиент отличается от сервера.
- SCADA система предоставляет сервисы для обнаружения устройств.

Асинхронная шина

- Есть единый «сервер» - (распределенная) шина и множество «клиентов» с двусторонней коммуникацией.
- Сообщения отправляются когда хочет клиент, а не когда спросили.



Что лучше?

P2P

- Синхронные запросы с гарантией доставки и гарантией времени отклика.
- Плохо масштабируется.
- Нужен сервис обнаружения устройств.
- Каждое сообщение доставляется только адресату.

Шина

- Асинхронные события. Нет (в общем случае) гарантий доставки.
- Хорошо масштабируется.
- Не нужен сервис обнаружения устройств.
- Сообщения доставляются всем, кто подписан.

It’s Magix

```
public interface MagixEndpoint {

    public fun subscribe(
        filter: MagixMessageFilter = MagixMessageFilter.ALL,
    ): Flow<MagixMessage>

    public suspend fun broadcast(
        message: MagixMessage,
    )

    public fun close()
}
```

Подписка на события

Отправка событий

Magix server

Волшебная штучка:

```
val magixFlow = MutableSharedFlow<MagixMessage>(
    replay = buffer,
    extraBufferCapacity = buffer,
    onBufferOverflow = BufferOverflow.DROP_OLDEST
)
```

Спецификация тут:

https://github.com/waltz-controls/rfc

Пример сообщения:

```
{
  "id": 1235,
  "origin": "waltz",
  "format": "dataforge",
  "target": "192.168.111.132:8882",
  "payload":{
    "type": "property.set",
    "targetDevice":"my-device",
    "property": "a",
    "value": 11,
    "comment": "pretty please!"
  }
}
```

Реализация для RSocket

```
RSocketRequestHandler(coroutineContext) {
    //handler for request/stream
    requestStream { request: Payload ->
        val filter = magixJson.decodeFromString(
            MagixMessageFilter.serializer(),
            request.data.readText()
        )
        magixFlow.filter(filter).map { message ->
            val string = magixJson.encodeToString(MagixMessage.serializer(), message)
            buildPayload { data(string) }
        }
    }
    //single send
    fireAndForget { request: Payload ->
        val message = magixJson.decodeFromString(MagixMessage.serializer(), request.data.readText())
        magixFlow.emit(message)
    }
}
```

Реализация для RSocket

```
public fun subscribe(
    filter: MagixMessageFilter = MagixMessageFilter.ALL,
): Flow<MagixMessage>
```

```
RSocketRequestHandler(coroutineContext) {
    //handler for request/stream
    requestStream { request: Payload ->
        val filter = magixJson.decodeFromString(
            MagixMessageFilter.serializer(),
            request.data.readText()
        )
        magixFlow.filter(filter).map { message ->
            val string = magixJson.encodeToString(MagixMessage.serializer(), message)
            buildPayload { data(string) }
        }
    }
    //single send
    fireAndForget { request: Payload ->
        val message = magixJson.decodeFromString(MagixMessage.serializer(), request.data.readText())
        magixFlow.emit(message)
    }
}
```

Реализация для RSocket

```
RSocketRequestHandler(coroutineContext) {
    //handler for request/stream
    requestStream { request: Payload ->
        val filter = magixJson.decodeFromString(
            MagixMessageFilter.serializer(),
            request.data.readText()
        )
        magixFlow.filter(filter).map { message ->
            val string = magixJson.encodeToString(MagixMessage.serializer(), message)
            buildPayload { data(string) }
        }
    }
    //single send
    fireAndForget { request: Payload ->
        val message = magixJson.decodeFromString(MagixMessage.serializer(), request.data.readText())
        magixFlow.emit(message)
    }
}
```

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

Протокол коммуникации

“Биологическое” разнообразие

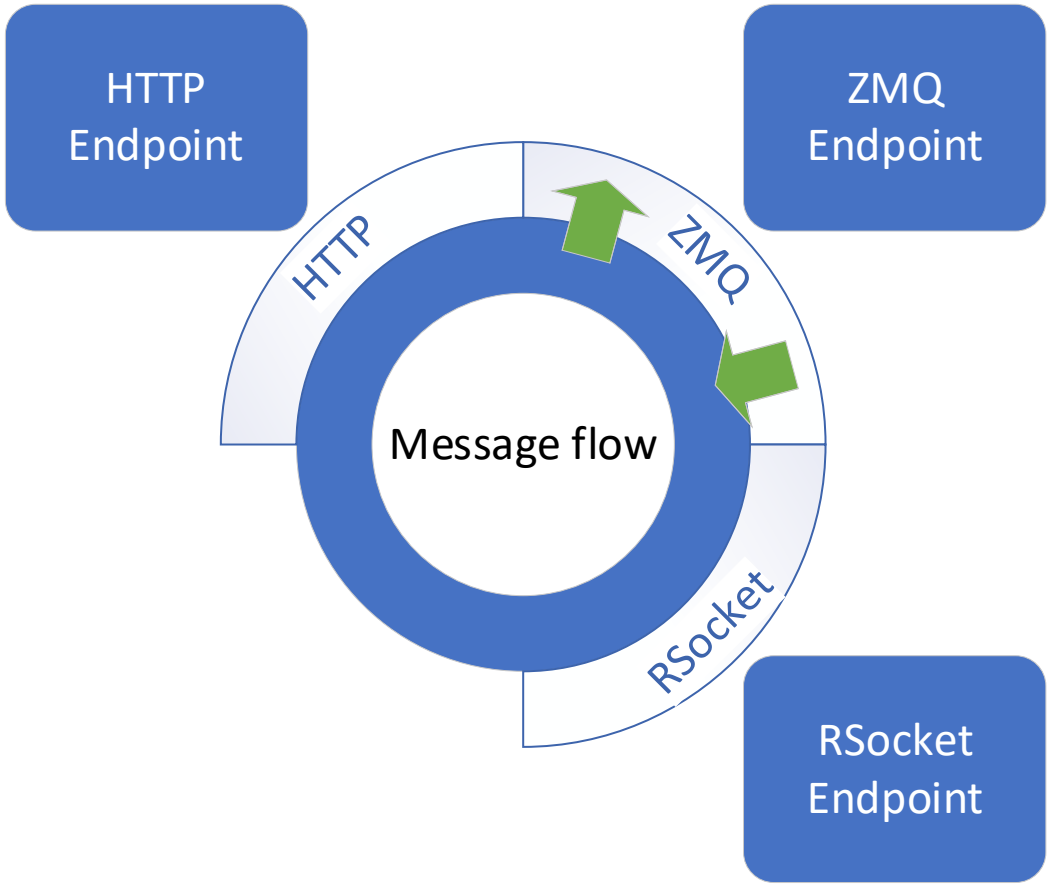
- EPICS
- Sun ONC (DOOCS)
- CORBA (TANGO controls)
- OPC-UA
- Protobuf
- HTTP/SSE
- WebSocket
- ZMQ
- …

Первое, что делает разработчик SCADA системы-
изобретает собственный протокол коммуникации

Почему протокол – это сложно?

- RPC протокол с безопасным типом требует схемы.
- Схему надо передать всем участникам коммуникации.
- Протокол надо поддержать на всех языках программирования, на которых написаны сервера устройств.
- Если библиотека, обеспечивающая протокол «протухла», надо поддерживать ее самостоятельно.

А почему один протокол?



- Общая шина позволяет реализовывать подключения по разным протоколам.
- Использование не-типизированных наполнений сообщений позволяет не думать о схеме.
- Конвертация протоколов не бесплатная, но очень дешевая.

Собираем все вместе

Система как конструктор

https://ideas.lego.com/projects/b383b238-c159-41e4-b4b9-7354240a890e

```
context.launch {
    device = deviceManager.install("demo", DemoDevice)
    //starting magix event loop
    magixServer = startMagixServer(
        RSocketMagixFlowPlugin(), //TCP rsocket support
        ZmqMagixFlowPlugin() //ZMQ support
    )
    //Launch device client and connect it to the server
    val deviceEndpoint = MagixEndpoint.rSocketWithTcp("localhost")
    deviceManager.connectToMagix(deviceEndpoint)
    //connect visualization to a magix endpoint
    val visualEndpoint = MagixEndpoint.rSocketWithWebSockets("localhost")
    visualizer = visualEndpoint.startDemoDeviceServer()

    //serve devices as OPC-UA namespace
    opcUaServer.startup()
    opcUaServer.serveDevices(deviceManager)
}
```

Система как конструктор

https://ideas.lego.com/projects/b383b238-c159-41e4-b4b9-7354240a890e

```
context.launch {
    device = deviceManager.install("demo", DemoDevice)
    //starting magix event loop
    magixServer = startMagixServer(
        RSocketMagixFlowPlugin(), //TCP rsocket support
        ZmqMagixFlowPlugin() //ZMQ support
    )
    //Launch device client and connect it to the server
    val deviceEndpoint = MagixEndpoint.rSocketWithTcp("localhost")
    deviceManager.connectToMagix(deviceEndpoint)
    //connect visualization to a magix endpoint
    val visualEndpoint = MagixEndpoint.rSocketWithWebSockets("localhost")
    visualizer = visualEndpoint.startDemoDeviceServer()

    //serve devices as OPC-UA namespace
    opcUaServer.startup()
    opcUaServer.serveDevices(deviceManager)
}
```

Система как конструктор

https://ideas.lego.com/projects/b383b238-c159-41e4-b4b9-7354240a890e

```
context.launch {
    device = deviceManager.install("demo", DemoDevice)
    //starting magix event loop
    magixServer = startMagixServer(
        RSocketMagixFlowPlugin(), //TCP rsocket support
        ZmqMagixFlowPlugin() //ZMQ support
    )
    //Launch device client and connect it to the server
    val deviceEndpoint = MagixEndpoint.rSocketWithTcp("localhost")
    deviceManager.connectToMagix(deviceEndpoint)
    //connect visualization to a magix endpoint
    val visualEndpoint = MagixEndpoint.rSocketWithWebSockets("localhost")
    visualizer = visualEndpoint.startDemoDeviceServer()

    //serve devices as OPC-UA namespace
    opcUaServer.startup()
    opcUaServer.serveDevices(deviceManager)
}
```

Система как конструктор

https://ideas.lego.com/projects/b383b238-c159-41e4-b4b9-7354240a890e

```
context.launch {
    device = deviceManager.install("demo", DemoDevice)
    //starting magix event loop
    magixServer = startMagixServer(
        RSocketMagixFlowPlugin(), //TCP rsocket support
        ZmqMagixFlowPlugin() //ZMQ support
    )
    //Launch device client and connect it to the server
    val deviceEndpoint = MagixEndpoint.rSocketWithTcp("localhost")
    deviceManager.connectToMagix(deviceEndpoint)
    //connect visualization to a magix endpoint
    val visualEndpoint = MagixEndpoint.rSocketWithWebSockets("localhost")
    visualizer = visualEndpoint.startDemoDeviceServer()

    //serve devices as OPC-UA namespace
    opcUaServer.startup()
    opcUaServer.serveDevices(deviceManager)
}
```

Система как конструктор

https://ideas.lego.com/projects/b383b238-c159-41e4-b4b9-7354240a890e

```
context.launch {
    device = deviceManager.install("demo", DemoDevice)
    //starting magix event loop
    magixServer = startMagixServer(
        RSocketMagixFlowPlugin(), //TCP rsocket support
        ZmqMagixFlowPlugin() //ZMQ support
    )
    //Launch device client and connect it to the server
    val deviceEndpoint = MagixEndpoint.rSocketWithTcp("localhost")
    deviceManager.connectToMagix(deviceEndpoint)
    //connect visualization to a magix endpoint
    val visualEndpoint = MagixEndpoint.rSocketWithWebSockets("localhost")
    visualizer = visualEndpoint.startDemoDeviceServer()

    //serve devices as OPC-UA namespace
    opcUaServer.startup()
    opcUaServer.serveDevices(deviceManager)
}
```

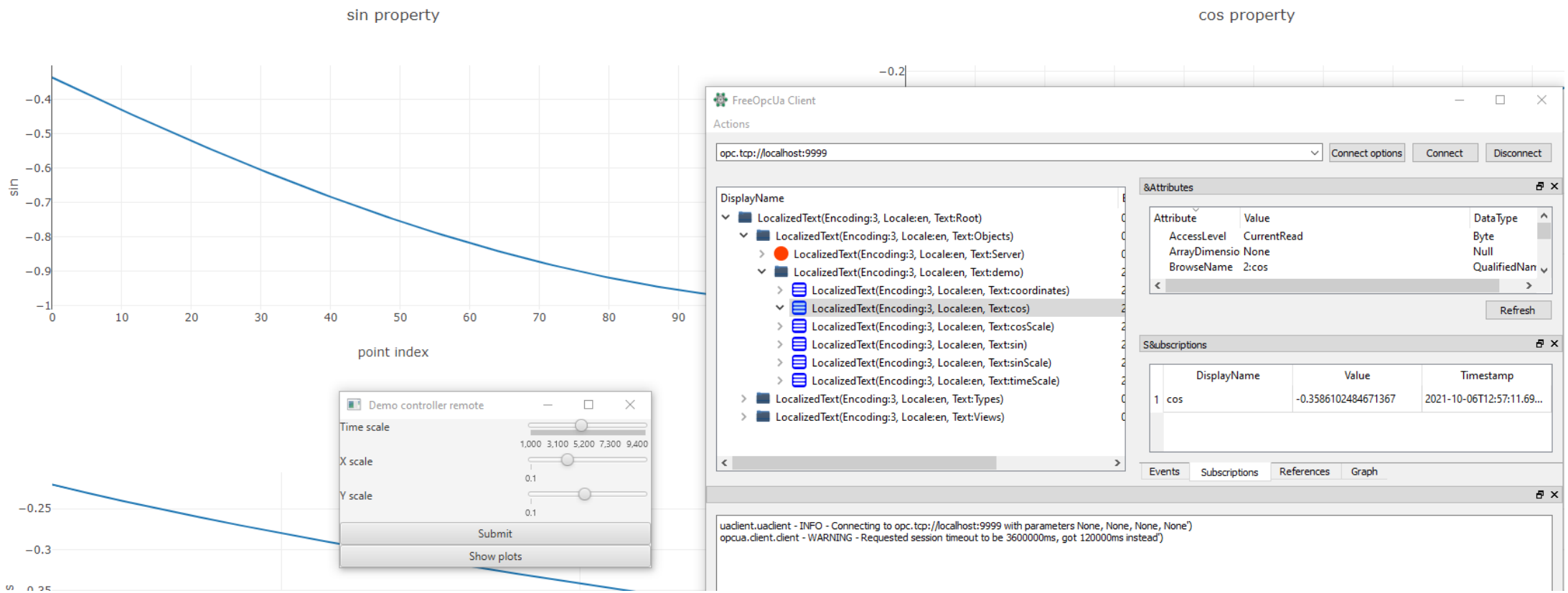
Система как конструктор

https://ideas.lego.com/projects/b383b238-c159-41e4-b4b9-7354240a890e

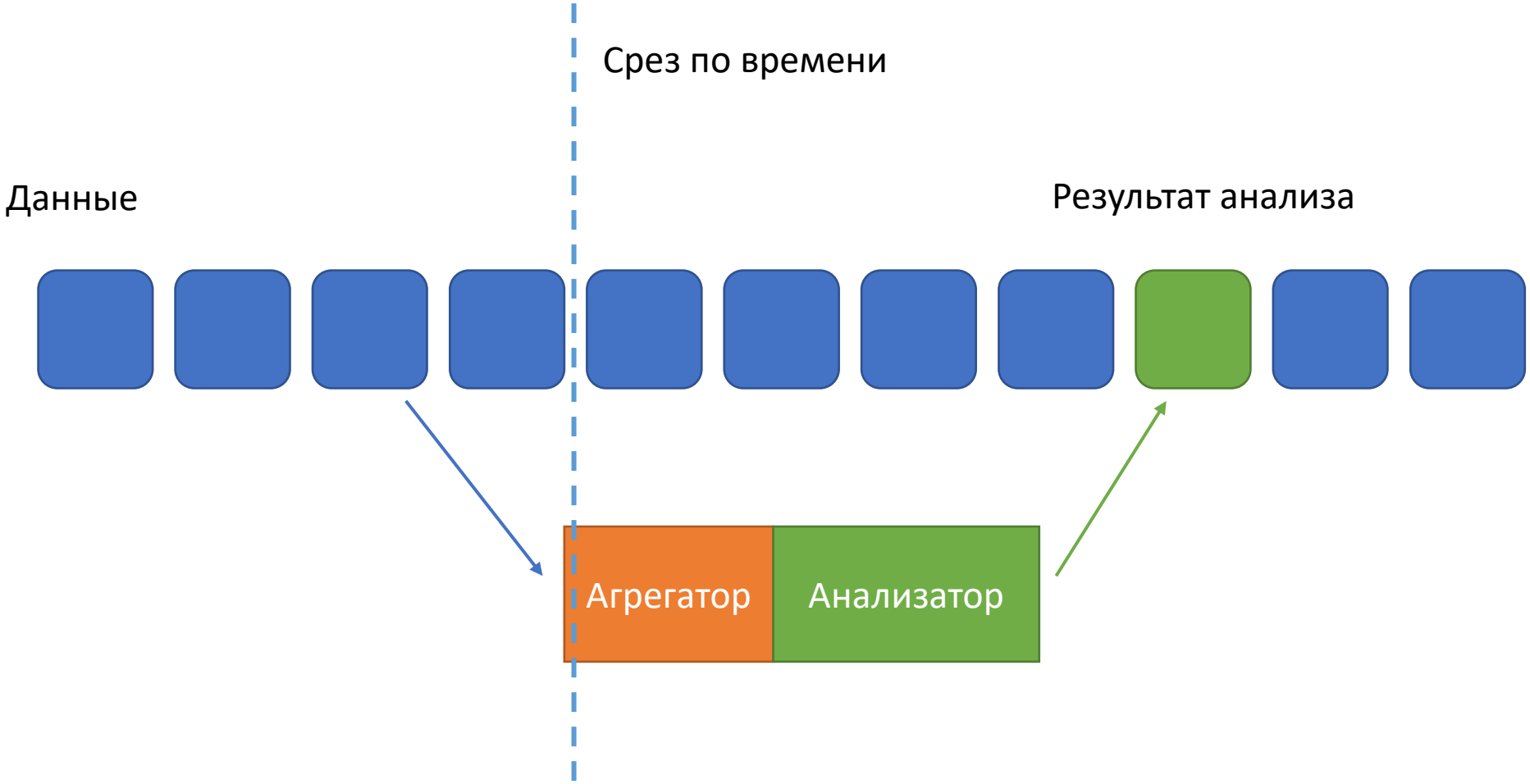
```
context.launch {
    device = deviceManager.install("demo", DemoDevice)
    //starting magix event loop
    magixServer = startMagixServer(
        RSocketMagixFlowPlugin(), //TCP rsocket support
        ZmqMagixFlowPlugin() //ZMQ support
    )
    //Launch device client and connect it to the server
    val deviceEndpoint = MagixEndpoint.rSocketWithTcp("localhost")
    deviceManager.connectToMagix(deviceEndpoint)
    //connect visualization to a magix endpoint
    val visualEndpoint = MagixEndpoint.rSocketWithWebSockets("localhost")
    visualizer = visualEndpoint.startDemoDeviceServer()

    //serve devices as OPC-UA namespace
    opcUaServer.startup()
    opcUaServer.serveDevices(deviceManager)
}
```

Система как конструктор

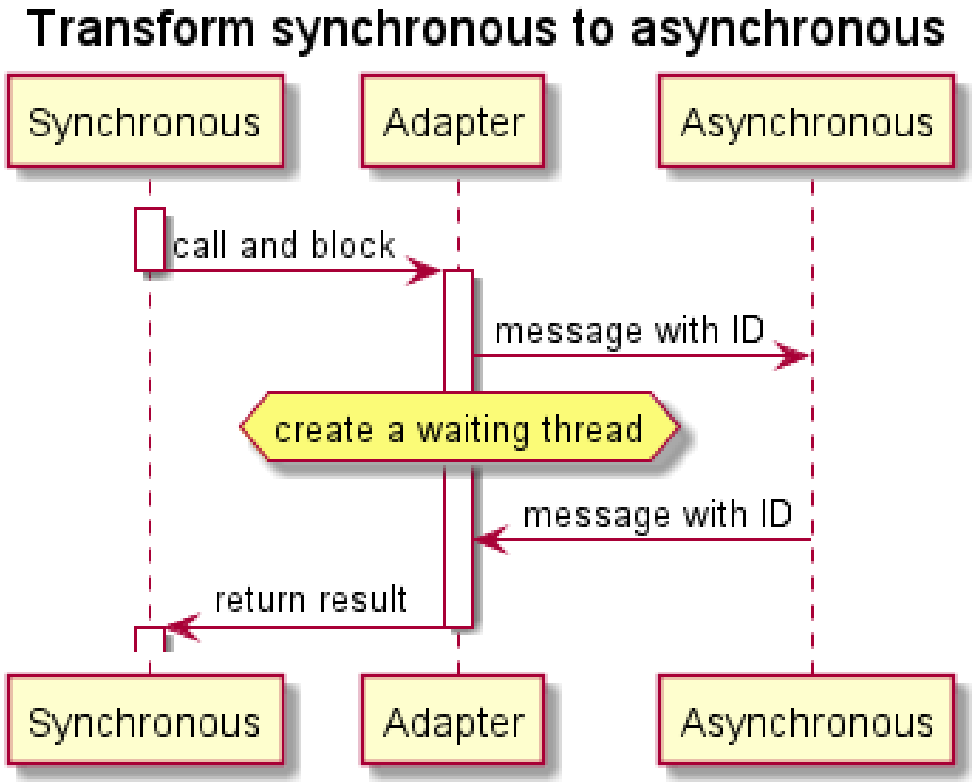


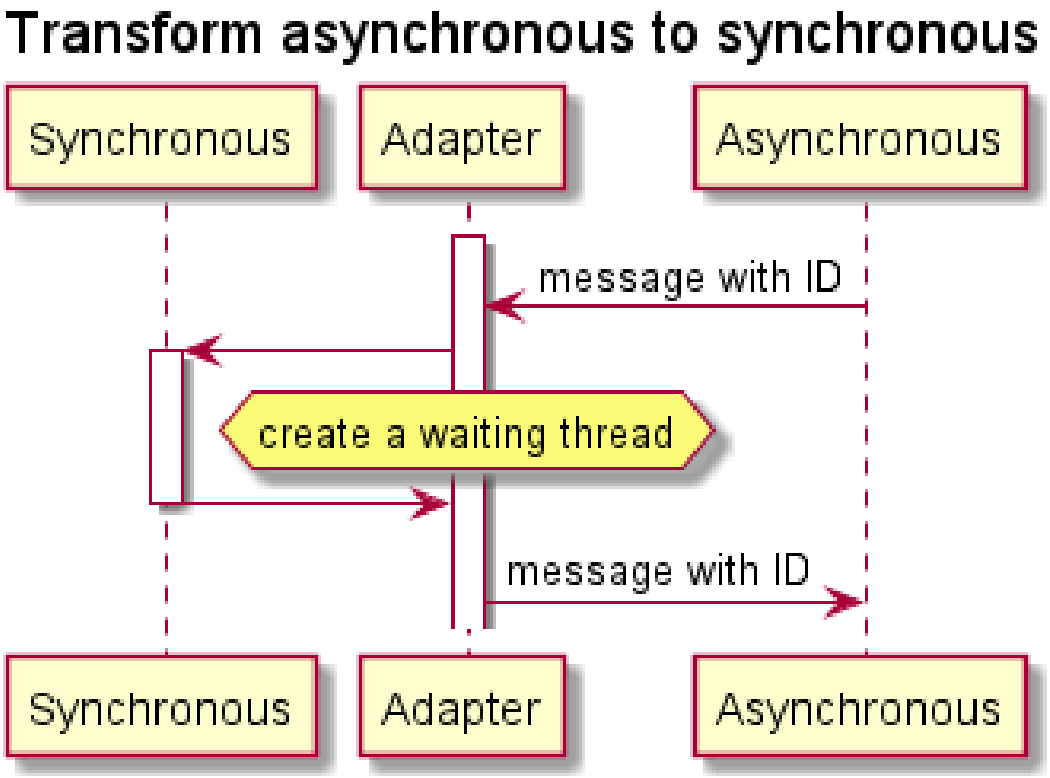
Асинхронный анализ данных



Проблемы

Синхронный запрос





Гарантии доставки

Проблема

- Асинхронная система (в отличие от синхронной) не дает гарантий доставки сообщений.
- Сервисы могут самопроизвольно подключаться и отключаться от шины.
- Или вовсе не поддерживать определенные типы сообщений.

Решения

- Использовать шину с гарантиями доставки (например Apache Kafka).
- Использовать сервис для проверки подключения (watchdog).
- Регулярно подавать признаки жизни (heartbeat).

Много сообщений

Проблема

- В худшем случае на одно сообщение от источника N пересылок (где N – количество сервисов).
- То есть общее количество пересылаемых сообщений N^2

Решения

- Использовать фильтр на стороне шины.
- Использовать распределенную шину (если не знаешь что делать, бери железо посильнее).
- Можно сегментировать шину…

Большие бинарные данные

Проблема

- Шина не годится для передачи больших бинарных данных.

Решения

- Передавать по шине только сигнал о появлении файла.
- Запрос файла делать напрямую к серверу устройства по другому протоколу.

В остатке

Выводы

- Мир систем сбора данных большой и дивный (но не новый).
- Хороших общепринятых решений там нет.
- Те, что есть в основном синхронные.
- Мы сделали конструктор на основе асинхронной шины.
- И вроде получилось (сейчас стадия MVP).

- Есть еще хранение, про него не успел рассказать.
- И OPC-UA.

Ссылки

- Сам проект: https://github.com/SciProgCentre/controls.kt
- TANGO: https://www.tango-controls.org/
- EPICS: https://epics.anl.gov/
- Визуализация на Walz: https://github.com/waltz-controls/waltz

- Обсудить: https://t.me/SciProgCentre