



FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

TECHNICAL REPORT

ICS - TESTBED

AUTHOR
AUTOR PRÁCE

PETER GROFČÍK

BRNO 2020

Contents

1	About	2
1.2	Running	3
2	Description of the framework packages	6
2.1	Relevant classes	6
2.2	First possible approach	8
2.3	Second approach	9
	Bibliography	10

Chapter 1

About

ICS-TestBed-Framework [2] is a scalable java based framework that consists of three types of virtual devices.

1. HMI (Human-machine interface) – periodically controlling other hosts
2. RTU (Remote Terminal Unit) – represents node for connection between sensors, etc. and HMI
3. Data Historian

The framework itself uses protocol IEC104[1] to communicate between its virtual machines (optionally with physical machines), even through non-virtual network.

The motivation for work with this framework is to create and observe corresponding communication, abnormalities in the common flow of communication or under attacks. Basically for the security analysis of IEC104 communication using virtual devices that corresponds to physical hardware.

1.1 Installation (based on Github repository¹)

1. `git clone --recurse-submodules https://github.com/PMaynard/ICS-TestBed-Framework.git`
2. `JAVA_HOME` variable is needed to be openjdk 1.8
 - `sudo apt install openjdk-8-jdk maven`
 - `export JAVA_HOME = /usr/lib/jvm/jre - 1.8.0 - openjdk.x86_64/`
3. add three dependencies to a pom.xml file in a root directory, because in a development state they were probably included in maven standard libraries, but got depreciated overtime
 - ```
<dependency>
 <groupId>com.sun.xml.bind</groupId>
 <artifactId>jaxb-core</artifactId>
 <version>2.2.11</version>
</dependency>
```

---

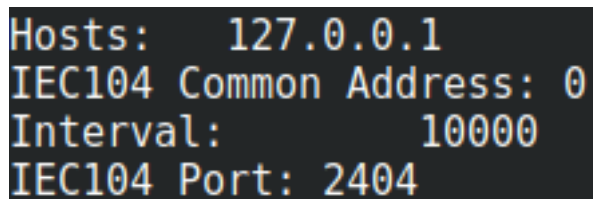
<sup>1</sup><https://github.com/PMaynard/ICS-TestBed-Framework>

- `<dependency>`  
`<groupId>com.sun.xml.bind</groupId>`  
`<artifactId>jaxb-impl</artifactId>`  
`<version>2.2.11</version>`  
`</dependency>`
  - `<dependency>`  
`<groupId>javax.xml.bind</groupId>`  
`<artifactId>jaxb-api</artifactId>`  
`<version>2.2.11</version>`  
`</dependency>`
4. `mvn package -DskipTests` (-DskipTests just skips some internal framework tests that end up failing in a wrong timezone)
  5. Firewall - some problems might occur with a firewall blocking virtual machines and might need to be disabled

## 1.2 Running

Each one of the virtual machines needs its shell terminal for deployment using a target node jar created after the successful build (`java -jar node/target/node-1.0.jar`). As for a default configuration, it is possible to use default local-host address 127.0.0.1 to deploy more virtual devices that will communicate just over a single physical device. As for the deployment of multiple devices that communicate over physical hardware, there is a need for correct LAN connection with IP address for physical hardware. After that, the virtual machine deployed on it can be started with a correct IP (that is set by commands listed below) that corresponds to physical hardware for the communication to be possible.

At the very beginning, there are commands RTU, HMI that is needed to choose a form of virtual device to be started from the exact terminal. The default configuration is loaded by framework after a form (HMI, RTU) is chosen.



```
Hosts: 127.0.0.1
IEC104 Common Address: 0
Interval: 10000
IEC104 Port: 2404
```

Figure 1.1: hmi-show command example for default settings

Default settings for RTU machine are the same for ip address and port used in communication. Common address is not needed, because it should be declared by HMI machine in a form of ASDU number. Interval is as well used only for HMI to determine pause time before next cycle in communication is invoked.

An example of topology created with this framework might look just like in a picture 1.2. I also used this topology in later steps after my changes in framework based on captured traffic example from real devices that I used as a model sample of IEC104 communication between RTU and HMI machine

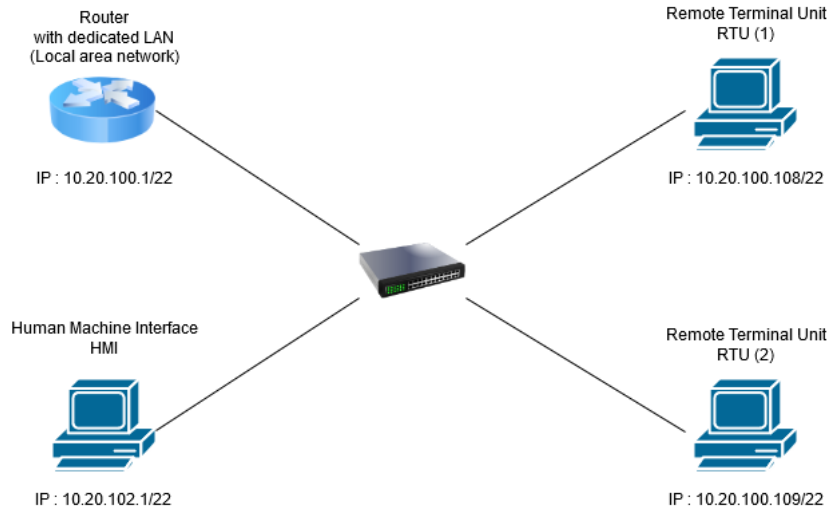


Figure 1.2: Simple example of topology consisting of one HMI and two RTU machines

**Possible commands for HMI machine:**

- hmi-iec104port [port number] – port number that HMI will connect to
- hmi-interval [number (ms)] – interval/timing for cyclic connection from HMI to other devices
- hmi-common-address [number] – common ASDU address of origination for connection
- remote-hosts [IP address] – IP address/addresses of devices that the HMI will connect to (every single input will override the old one for more devices at the moment needs an input of all desired IP addresses split by comma)
- hmi-show – shows current HMI configuration

**Possible commands for RTU machine:**

- rtu-iec104port [port number] – port number on which RTU will expect connection
- rtu-listen [IP address] – IP address on which the RTU will expect connection (as said for connection over physical hardware must correspond with IP address of device of which it will be started on)

For both devices, there is a single command “run” when the configuration is finished. HMI machine must be last to start because it requires listeners on the other side, which means that it will fail as soon as there are none.

**Already prepared code consist of four default interrogation command communication:**

- Act – beginning of interrogation ( $C\_CS\_NA\_1$  - Clock synchronization command)
- ActCon – reply to beginning of interrogation ( $C\_CS\_NA\_1$  - Clock synchronization command)

- Inrogn – interrogation command ( $M\_ME\_NB\_1$  - Measured value, scaled value)
- ActTerm – termination if interrogation ( $C\_IC\_NA\_1$  - (General-) Interrogation command)

This sequence is just a simple example of IEC104 interrogation communication between HMI and RTU device, but it is not possible to change it anyhow without change of a corresponding machine code explained below. As said, the framework does not consist of any sort of commands that can change the flow of communication between virtual (physical) devices. Change of flow in communication can be achieved by change on code inside some of the relevant classes created for it, which are explained below.

$C\_CS\_NA\_1$ ,  $M\_ME\_NB\_1$  and  $C\_IC\_NA\_1$  stands for ASDU (Application Service Data Unit ) types that can be used in communication in the right order. Reference about possible combinations can be found for example in a technical report *Description and analysis of IEC 104 Protocol*<sup>2</sup>.

---

<sup>2</sup><https://www.fit.vut.cz/research/publication-file/11570/TR-IEC104.pdf>

## Chapter 2

# Description of the framework packages

Framework consists of three main and two secondary directories:

- Main directories:
  1. *j60870* - The directory containing a structure of source and build files for the main functionality of a framework. Most of the files were not made to be changed in any way, because it may cause a malfunction of the framework, but in some cases, it might be a good way to alter or add some helpful functions for the creation of customized packets or even sequences.
  2. *node* - This directory contains the source and built files that can be used to change the flow of communication mentioned above. A sub-directory *src* contains relevant classes to do so. A sub-directory *target* contains build files as well as a file *node-1.0.jar* that is of executable java type format for framework execution.
  3. *UA-Java* - This directory contains „*Unified Architecture Java Stack and Sample Code to the community*“. To my understanding, it was taken from other project repository and can be altered to a user's needs if needed, but for experiments with devices created by this framework will not be necessary.
- Secondary directories:
  1. *attacks* - This directory contains python scripts as well as logs about attack experiments carried by the creators of framework.
  2. *scripts* - This last directory contains script examples, that can be loaded after the execution of the java executable file to start single virtual devices.

### 2.1 Relevant classes

To alter the flow of communication itself there are few classes in sub-folder */node/src/main/java/xyz/scada/testbed/node*. The main classes for doing so are HMI, HMIConnect-Interrogate, and RTU.

HMI class consists of default settings that we can alter yet again just to change IP address, port, etc. for communication, but also a sleep time that separates a sequence of interrogation commands inside of the cycle.

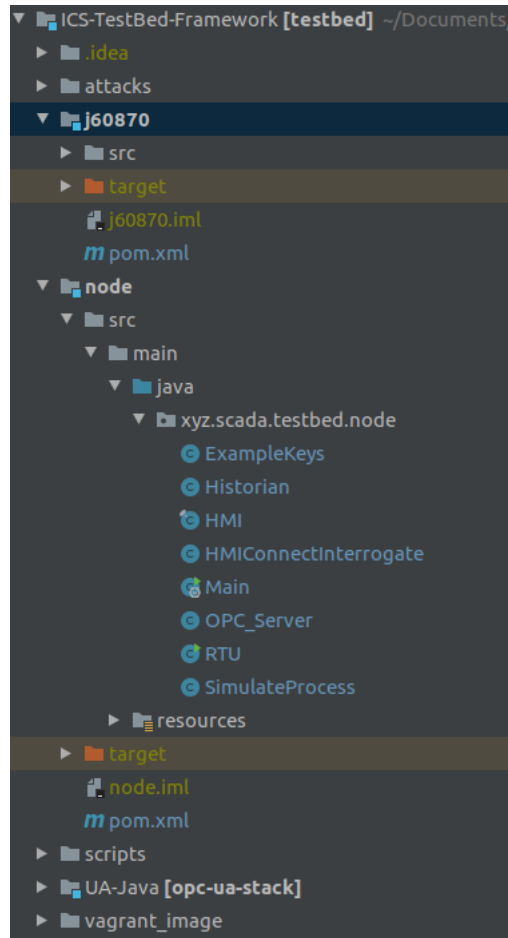


Figure 2.1: Framework structure with relevant classes unrolled

On the other hand, HMIConnectInterrogate class contains this so-called cycle, inside of which we can add some interrogation command sequences. With a use of some inner logic even for more types of devices to connect to.

RTU class is mostly one big logical switch that will make an RTU response depending on the Act command. It is possible to consider other aspects of incoming activation, but this should always be the most important one because it separates possible responses based on the IEC104 standard.

For most of possible interrogation commands or responses, there are various classes in a sub-folder `/j60870/src/main/java/org/openmuc/j60870`. Based on those and with them, I was able to alter the flow of communication to correspond with one that was captured from real devices. A problem in this state is that it will create the corresponding “static” communication that maps only a small-time on communication with static values to be sent between virtual devices, and it’s also impossible to make it work with some predefined physical hardware like that.



## 2.2 First possible approach

Based on already captured traffic of real devices that communicate using IEC104 it might be possible to automatically create corresponding code for virtual devices by filtering important values such as:

- Type of command (*Act*, *ActCon*, *Init*, etc.)
- TypeID of command (*C\_CS\_NA\_1*, *M\_ME\_NB\_1*, etc.)
- OA – Address of originate
- IOA – objects of interest for command send with all it's various different values

This approach will only work if it simulates all the devices because it creates specific communication that will require specific responses from the other side. It will not guarantee that if we use some real devices that it will not fail. It might be possible if we map just an HMI side that will dictate RTU devices (real device used), but only if we use exact device that was used to capture from and yet it might still fail, because it's not only getting orders to do something but also obtain some other values from sensors and so, in that case, it will respond with no mapped values.

```
> Frame 39: 70 bytes on wire (560 bits), 70 bytes captured (560 bits)
> Ethernet II, Src: AsustekC_56:0b:54 (00:22:15:56:0b:54), Dst: ZatAS_00:09:05 (00:16:d1:00:09:05)
> Internet Protocol Version 4, Src: 10.20.102.1, Dst: 10.20.100.108
> Transmission Control Protocol, Src Port: 46413, Dst Port: 2404, Seq: 133, Ack: 1689, Len: 16
> IEC 60870-5-104-Apci: <- I (6,45)
v IEC 60870-5-104-Asdu: ASDU=10 C_RC_NA_1 Act IOA=1 'regulating step command'
 TypeId: C_RC_NA_1 (47)
 0... = SQ: False
 .000 0001 = NumIx: 1
 ..00 0110 = CauseTx: Act (6)
 .0.. = Negative: False
 0... = Test: False
 OA: 0
 Addr: 10
 v IOA: 1
 IOA: 1
 > RCO: 0x02
```

Figure 2.2: Example of captured packet activation from HMI machine

Based on captured traffic we can as a first step determine IP addresses of HMI and RTU in communication, thanks to the type of commands used in it. For example, *Act* command used in figure 2.2 means that this captured packet was sent by HMI machine because only HMI initiates communication (represented by *Act* or *Init* command). Then we can see that it used *OA* (*ASDU*) of value 10 and TypeID of command *C\_RC\_NA\_1* as well as all values used in *ASDU* itself. Also, each *ASDU* can contain any number of *IOA*, so we need to parse all of them. These invocations of communication are not conditional and will only be sent in a cycle after a certain time period.

On the other hand in figure 2.3 we can see a reply from RTU device, generated after HMI invoked communication with TypeID (*C\_RC\_NA\_1*), which can be seen in *ActCon* reply. Based on that we can determine what kind of reply takes place after what kind of invocation is received from HMI machine and use that to create a case in class RTU that will correspond to it, also with parsing other information from *ASDU* as done for HMI. This needs to be done for all *ASDU* received in packet as well. *ActTerm* type of command contains the same TypeID as had an invocation so it can be parsed even if it came in a

different packet then ASDU with *ActCon* type of command, but *Spont* should and will be only right after *ActTerm* ASDU in the same packet.

```

> Frame 40: 103 bytes on wire (824 bits), 103 bytes captured (824 bits)
> Ethernet II, Src: ZatAS_00:09:05 (00:16:d1:00:09:05), Dst: AsustekC_56:0b:54 (00:22:15:56:0b:54)
> Internet Protocol Version 4, Src: 10.20.100.108, Dst: 10.20.102.1
> Transmission Control Protocol, Src Port: 2404, Dst Port: 46413, Seq: 1689, Ack: 149, Len: 49
> IEC 60870-5-104-Apci: -> I (45,7)
√ IEC 60870-5-104-Asdu: ASDU=10 C_RC_NA_1 ActCon IOA=1 'regulating step command'
 TypeId: C_RC_NA_1 (47)
 0... = SQ: False
 .000 0001 = NumIx: 1
 ..00 0111 = CauseIx: ActCon (7)
 .0.. = Negative: False
 0... = Test: False
 OA: 0
 Addr: 10
 √ IOA: 1
 IOA: 1
 > RCO: 0x02
> IEC 60870-5-104-Apci: -> I (46,7)
> IEC 60870-5-104-Asdu: ASDU=10 C_RC_NA_1 ActTerm IOA=1 'regulating step command'
> IEC 60870-5-104-Apci: -> I (47,7)
> IEC 60870-5-104-Asdu: ASDU=10 M_ST_NA_1 Spont IOA=1 'step position information'

```

Figure 2.3: Example of captured packet response from one RTU

## 2.3 Second approach

We can assume that the HMI device is sending it is commands in a cycle and from that just create a logical device RTU on the other side that will hold it is own values for interrogation and can work with them depending on commands obtain from HMI. For that, it might help to use some specifications of RTU device<sup>1</sup>. Problem is that even if this logic is implemented well, it expects direct connections to other devices (sensors, switches, thermometers, etc.). For that, we can use some kind of randomization. It should be enough for a simulations of IEC104 communication, even if it does not fully correspond with an exact communication that might or might not have taken place with the real devices, because for this simulation it does not matter if our virtual RTU should generate values that would change the behavior of already mentioned sensors.

<sup>1</sup>[https://www.vfservis.cz/files/000290\\_RTU560\\_SD\\_R6.pdf](https://www.vfservis.cz/files/000290_RTU560_SD_R6.pdf)

# Bibliography

- [1] *INTERNATIONAL STANDARD IEC60870-5-104*. Jun 2006. [Online; 25.03.2020]. Available at:  
[https://webstore.iec.ch/preview/info\\_iec60870-5-104%7Bed2.0%7Den\\_d.pdf](https://webstore.iec.ch/preview/info_iec60870-5-104%7Bed2.0%7Den_d.pdf).
- [2] MAYNARD, P., McLAUGHLIN, K. and SEZER, S. An Open Framework for Deploying Experimental SCADA Testbed Networks. In:. 2018. Available at:  
<https://ewic.bcs.org/content/ConWebDoc/59846>.