ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP076

STREAM-BASED VIRTUAL DEVICE SIMULATION FOR ENHANCED EPICS INTEGRATION AND AUTOMATED TESTING

M. Lukaszewski*, K. Klys, E9 Controls Limited, London, UK

Abstract

Integrating devices into the Experimental Physics and Industrial Control System (EPICS) can often take a suboptimal path due to discrepancies between available documentation and real device behaviour. To address this issue, we introduce “vd” (virtual device), a software for simulating stream-based virtual devices that enables testing communication without connecting to the real device. It is focused on the communication layer rather than the device’s underlying physics. The vd listens to a TCP port for client commands and employs ASCII-based byte stream communication. It offers easy configuration through a user-friendly config file containing all necessary information to simulate a device, including parameters for the simulated device and information exchanged via TCP, such as commands and queries related to each parameter. Defining the protocol for data exchange through a configuration file allows users to simulate various devices without modifying the simulator’s code. The vd’s architecture enables its use as a library for creating advanced simulations, making it a tool for testing and validating device communication and integration into EPICS. Furthermore, the vd can be integrated into CI pipelines, facilitating automated testing and validation of device communication, ultimately improving the quality of the produced control system.

INTRODUCTION

EPICS (Experimental Physics and Industrial Control System) is one of the most widely used frameworks for designing distributed control systems for large experiments such as accelerators and observatories [1].

Integrating new devices into the EPICS can be challenging. This is often attributed to the dissonance between documented device behaviours and their operations. This disparity can lead to sub-optimal integration paths, potentially compromising the robustness and reliability of the control system.

Another layer of complexity arises due to project delays, making it increasingly difficult to test the devices on time. Sometimes, specific devices can only be tested once they are delivered. This further exacerbates the challenges faced during integration, increasing the potential for integration errors and system malfunctions.

A solution to this problem could be writing device simulators and creating integration tests based on these simulators. While this may sound straightforward, crafting simulators is time-consuming. This is primarily because it necessitates extensive programming and a profound understanding of the device’s behaviour. The effort to develop a simulator can sometimes feel counterintuitive, as the time invested only sometimes aligns with the benefits received. Furthermore, developing simulators demands a different set of networking and programming skills than those used daily by control system engineers. It also requires additional effort to ensure the simulator remains updated and consistent with the device’s documentation.

In our pursuit to streamline and enhance the integration process, we are committed to simplifying the creation of simulators. In our solution, we deliberately bypass the portion of the simulator responsible for the device’s behaviour, concentrating solely on the communication layer. This approach allows us to create specific device states by externally adjusting parameter values. Simultaneously, we can verify that the simulated device communicates these changes appropriately using the designated protocol.

By modifying parameters through a separate channel from the one used by the client, we can establish an automated test suite as part of a continuous integration process. This allows us to set up a grid of tests covering all the essential device states without coding the device’s behaviour.

ARCHITECTURE

The software’s architectural design comprises four distinct layers delineated in Fig. 1. It employs the Golang programming language, leading to a single binary file as output. The absence of external dependencies in this configuration simplifies the distribution process and augments its compatibility, ensuring seamless execution across a diverse range of operating systems.

In the architectural design, the layers were interconnected via interfaces. Within the Golang programming language context, interfaces serve as a core way to construct complex systems. These systems comprise modular components, each of which can be independently developed and tested. Such an approach fosters flexibility and ensures that the Stream layer can be substituted in the future with an alternative layer accommodating different communication protocol formats, should the need arise.

Notably, Fig. 1 omits elements related to the reading of the configuration file in the TOML format and the logging mechanism. This intentional exclusion ensures clarity and prevents obfuscation of the primary architectural representation.

TCP Server

The first layer under discussion is the TCP Server. It is constructed based on the standard library available in Golang. Golang, known for its efficient concurrency management and clean syntax, offers developers a solid foundation to build robust systems.

* marcin.lukaszewski@e9controls.com

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP076

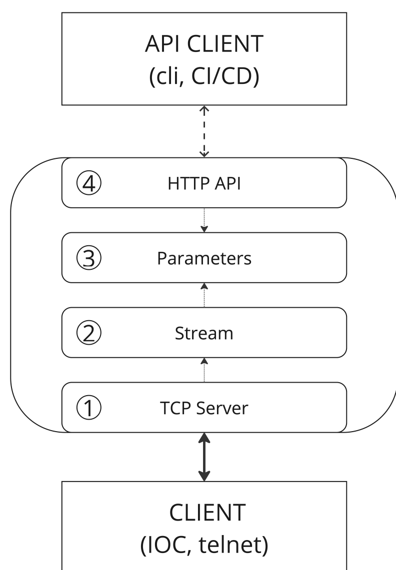

Figure 1: Software architecture.

One of the distinct advantages of using Golang's standard library for the TCP Server is the multi-platform compatibility it brings to the table. This means that applications built with this foundation can run seamlessly on various platforms without significant modifications. This multi-platform nature aligns well with today's need for diverse computing environments, from traditional servers to modern cloud infrastructures.

Moreover, the number of dependencies in the project is minimised by utilising the standard library. Reducing external dependencies not only simplifies the maintenance and updating process but also enhances the security and stability of the software. Fewer dependencies mean a lower risk of encountering compatibility issues or vulnerabilities associated with third-party libraries.

Upon reading data from the client, the TCP Server forwards it to the subsequent layer using the "Handle" method, which is part of the Handler interface. This method, acting as an intermediary function, has a crucial role in the entire data processing chain, ensuring that data flows smoothly from one component to another.

```
type Handler() interface {
    Handle([]byte) []byte
}
```

Due to this specific data structure, the TCP Server does not undertake the task of protocol decoding. Instead, its primary role is to serve as a bridge, facilitating the transfer of raw, unprocessed data. This design choice aids in keeping the server's functions modular and reduces the complexity of integrating multiple functionalities into a single unit.

The layer referred to as "Stream" subsequently receives this array of bytes. The architecture ensures a clear separation of concerns by keeping the TCP Server decoupled from the decoding process. This separation enables easier debugging, modification, and potential system expansion. With each layer having its distinct responsibilities, developers can modify or enhance a specific part without causing disruptions to the entire ecosystem.

Stream Layer

The "Stream" is a module in which the analysis of incoming data to the simulator takes place. It assigns the incoming data to the respective command, parameter, and value (if applicable). The simulator maintains a clear separation of responsibilities by centralising the data analysis and routing functions within the Stream module.

Lexer We have decided to write the custom lexer inspired by Rob Pike presentation titled "Lexical Scanning in Go" [2]. Its task is to convert byte messages into understandable for parser tokens. It is based on the state machine that goes to the next logic state by returning the state function depending on the current result of scanning. The change of the state is triggered by the scanner that divides bytes messages into runes (Golang alias for int32 representing Unicode CodePoints). Thanks to that, the lexer can detect not only standard characters but also symbols. The state function is a piece of code that emits tokens and detects errors. When the message is parsed and converted into tokens, they are transferred to the parser.

Parser This parser must determine the exact transmitted command based on the tokenised input. By comparing the incoming data against the list of recognised tokens, the parser can quickly and accurately ascertain the nature of the command, allowing the simulator to respond or act accordingly.

Parameters

The "Parameters" layer is responsible for managing the values of the parameters of the simulated device. When the simulator is launched, these parameters are dynamically created from a configuration file. They can be of various types: int, int32, int64, float32, float64, bool, and string. During the setting of a value, the new value type is verified, ensuring that one cannot assign an inappropriate value to a specific parameter type. The design considerations for this dynamic creation are rooted in the need for adaptability and customisation, allowing the simulator to be versatile in representing different device configurations and behaviours.

Furthermore, the parameters employ a mechanism called mutual exclusion. This is crucial as it simultaneously permits safe access to a parameter from multiple points within the program. Mutual exclusion mechanisms ensure that data races and other concurrency-related issues are avoided, maintaining the integrity and reliability of the parameter values, even in complex, multi-threaded environments.

This safety mechanism is of paramount importance for our simulator. Since parameter values can be altered concurrently by a TCP client and through a dedicated API, there

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP076

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

is an inherent risk of conflicts or unsynchronised changes. The simulator guarantees consistent and accurate updates by utilising mutual exclusion, ensuring that the simulated device behaves reliably, irrespective of the number or nature of simultaneous interactions.

HTTP API

The "vd" exposes the parameters of the current simulator instance as a simple REST (Representational state transfer) API (Application Programming Interface). One can modify the virtual device parameters with the HTTP POST method and read them by sending a GET verb. The HTTP routing is done with a "chi" router for GO HTTP services [3]. Users can send queries using external tools like CURL, web browser, or CLI (Command Line and Interface) commands.

Configuration

Configuring the simulator involves creating a text file in the TOML format. By default, this file is named "vdfile", but it can have a varied name and extension since users can direct the simulator to a specific file for simulation. TOML aims to offer a streamlined configuration file format, emphasising simplicity and readability with its easily interpretable semantics. The essence of TOML's design philosophy is to provide users with a configuration language that is machine- and human-friendly, striking a balance between complexity and comprehensibility.

The configuration file encompasses information about terminators for both input and output commands. Additionally, it contains an array of tables where each table describes a particular parameter.

```
# This is vdfile config

[terminators]
intterm = "CR LF"
outterm = "CR LF"

[[parameter]]
name = "current"
typ = "int"
req = "CUR?"
res = "CUR %d"
set = "CUR %d"
acq = "OK"
val = 300

[[parameter]]
name = "version"
typ = "string"
req = "VER?"
res = "%s"
val = "version 1.0"
```

The use of tables in TOML aids in organising the data hierarchically, making it easier for control system engineers to navigate and modify the configuration as required.

Each parameter must possess both a name and a type. Depending on the device's simulated functionality, a given parameter can be read, written, or both. To specify the communication method for these functionalities, one needs to configure the "req", "res", "set", and "ack" attributes, respectively. Familiar placeholders, akin to those used in the "printf" and "scanf" functions, can also be utilised.

It is worth mentioning that the names of the keywords and the way the communication methods are configured are designed to resemble as closely as possible the configuration of an EPICS protocol file used in the device support module called StreamDevice. This makes the integration with EPICS less time-consuming and more intuitive for EPICS users.

USAGE

Local Development

The "vd" aids during local development. It allows quick verification of the communication layer between IOC and the device emulator. To properly design the data transmission on the IOC side, debugging the "vd" can be helpful. While running the simulator with the proper flag, the console will present all the data traffic between the "vd" instance and the IOC (in the bytes, hex, or string format, depending on the user configuration). This facilitates IOC debugging and allows potential errors to be eliminated.

CI/CD

In the CI/CD (Continuous Integration/Continuous Delivery) systems, the "vd" can be used as part of the IOC integration tests in the pipeline. The testing scenario may involve launching IOC with the "vd" based simulator, modifying setpoint PVs, and verifying changes with corresponding readback PVs. This simple procedure can detect communication misconfiguration on the IOC side or errors in IOC database files at an early stage of development.

Thanks to such tests, the IOC integration with the physical device can be more fluent and less time-consuming. Suppose the device documentation of that device is complete and without mistakes. In that case, it can be stated that no modifications to the IOC communication layer can be required after tests with "vd" tool.

The test procedures based on "vd" can be part of Gitlab CI/CD or Github Actions if the IOC is stored in the code repository. After each commit, the pipeline with the test steps can be run, and the user will be quickly informed about the status of the introduced changes. Another alternative is to launch tests with automation software like Jenkins or Ansible on the local setup.

The "vd" tool is just a binary file without any external dependencies, so it can be easily adapted to the existing infrastructure or encapsulated into Docker images without prior preparations.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP076

CONCLUSIONS

The paper precisely describes "vd" tool for simulating stream-based virtual devices for software tests. It presents the motivation behind the design of such a tool and how it has been developed. The paper depicts the vd's architecture together with packages and modules. It describes the communication layer based on TCP protocol and interfaces exposed for parameter control.

The first version of the "vd" tool has been released. Nevertheless, the process of introducing new features is still in progress. It is planned to add new communications interfaces such as Modbus and other binary protocols. We are working on new configuration parameters that allow users to customise error messages when the wrong query has been sent to the device or to accept arrays as parameter values.

In the meantime, we are creating new configuration files for the devices used in the control systems we are familiar with, where we have been involved in their development. This allows us to verify and ensure that "vd" is able to simulate any device based on the stream protocols.

REFERENCES

[1] EPICS about, https://epics-controls.org/about-epics/

[2] Lexical Scanning in Go, https://go.dev/talks/2011/lex.slide

[3] Lightweight, idiomatic and composable router for building Go HTTP services, https://github.com/go-chi/chi

© Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI