ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA045

A NEW SIMULATION STRUCTURE TO IMPROVE SOFTWARE DEPENDABILITY IN COLLIDER-ACCELERATOR CONTROL SYSTEMS*

Y. Gao†, T. Robertazzi, Stony Brook University, Stony Brook, USA
K. A. Brown, J. Morris, R. H. Olsen, Brookhaven National Laboratory, Upton, USA

Abstract

The Collider-Accelerator Department (C-AD) at Brookhaven National Laboratory (BNL) operates a world-class particle accelerator – the Relativistic Heavy Ion Collider (RHIC). To ensure its safe and proper operations, C-AD develops its own control systems. It is a large distributed complex system consisting of approximately 1.5 million control points [1]. The system has two physical layers: the front end level and the console level. In work [2], a new simulation structure is proposed which aims to improve the console level codes reliability. The structure enables developers to conveniently do customized testing on ADO[1] codes, specifically ADOs using the General Purpose Interface Bus (GPIB) interface [3]. In this work, a new simulation framework is proposed. It extends the simulation structure in [2] by accommodating new types of ADOs that use the Ethernet connections. Together, they form a more comprehensive simulation environment which enhances the overall controls software dependability.

INTRODUCTION

The Relativistic Heavy Ion Collider (RHIC) at Brookhaven National Laboratory (BNL) is a world-class particle accelerator, which helps scientists to study what the university may have looked like in the first few moments after its creation. RHIC contains two 3.8 kilometers counter-rotating super-conducting rings to carry particle beams which can be collided in 6 crossing regions to provide possible interactions for experimenters to study.

The RHIC is operated by elaborate control systems at the Collider-Accelerator Department (C-AD) of BNL. Instances of the C-AD control systems are also applied in the Linear Accelerator (LINAC), Electron Beam Ion Source (EBIS), Tandem Van de Graff pre-accelerators, the Booster accelerator, and the Alternating Gradient Synchrotron (AGS). C-AD control systems provide operational interfaces to the accelerator complex. Its architecture is hierarchical and consists of two physical layers with network connections: the Front End Computers (FECs) level and the Console Level Computers (CLCs) level, as shown in Fig. 1. The front end level contains more than 500 FECs, each of which running on the VxWorks™ real-time operating system. Every FEC consists of a VME chassis with a single-board computer[2], network connection, and I/O modules [4]. The console level

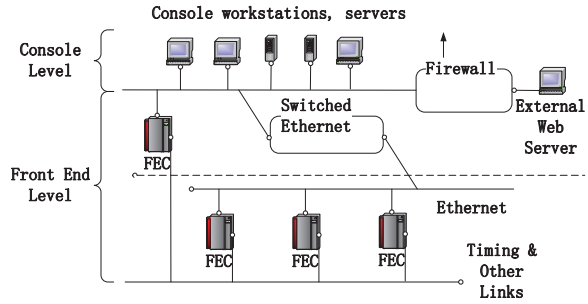

Figure 1: RHIC system hardware architecture.

is the upper layer of the control system hierarchy, which consists of operator consoles, physicist workstations and server processors that provide shared files, database and general computing resources.

There are several fundamental system components in the C-AD control systems.

Accelerator Device Object (ADO) It abstracts features from the underlying devices into a collection of control points (also known as parameters), and provide those parameters to the users of the control systems. ADO designers determine the number and names of parameters based on the needs of the system. ADO parameters can be viewed or edited by the Parameter Editing Tool (PET). ADOs provide the *set()* and *get()* methods as the controls interface to the accelerator devices. The accelerator complex is controlled by users or applications which set() and get() parameter values in instances of the ADO classes. A special preprocessor is used to help to convert ADO ".rad" files[3] into C++ files [5]. It takes care of the necessary details, and allows the ADO designers to focus on the more important parts: the ADO functionalities.

Controls Name Server (CNS) It is a centralized repository where unique name/value pairs can be efficiently managed and queried. Given an object's instance name[4], the CNS will provide enough information so that the associated data can be accessed. The CNS is session oriented which means several copies of it can be run at the same time as long as each of them has its own host. This feature allows developer to have a "private" CNS, which makes it possible to signal a process to look for an ADO instance in a different place from where it normally resides. The proposed

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
† ygao@bnl.gov
[1] ADO stands for Accelerator Device Object, see details below.
[2] They can have different processor architectures, e. g. POWER3E, MV2100, MV3100, XILINX, etc.
[3] It stands for RHIC ADO Definition file.
[4] That object can be an ADO parameter, a Complex Logical Device (CLD), a manager's parameter name, or an alias (a name used by developers which is more human-readable), etc.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA045

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

simulation framework leverages this property to redirect applications to interact with simulated ADOs instead of real ADOs.

Logging System The logging system in the C-AD control systems is used to save machine parameters and device values to provide a history of accelerator's performance and data to be analyzed for machine physics studies [4]. Logging requests are initiated by files that define what device parameters to be logged and the method of logging. Data can be logged upon demand or periodically. In the early years, data were logged by demand using the General Purpose Monitor (Gpm), hence also called Gpm logging. The data logged in this way are usually array data, so they occupy a large amount of storage space. The second logging method came later, which aggregates and updates information periodically. This type of logging method is called pool logging. Users can get their interested information from the pool logging data without the need of making individual logging requests, hence it reduces the access frequency of FECs. The pool logging data are typically stored in scalar format and hence enabling more data to be archived. For comparison of the two logging methods, 14% of items (such as ADO parameters, etc.) are saved by Gpm logging occupying 85% of data volume, while the rest of 86% items are saved by pool logging occupying only 15% of data volume [6]. Several applications can be used to view, edit and process logging data, such as Gpm, LogView, etc.

Motivation

In work [2], a simulation architecture was proposed to improve ADO codes reliability. The framework focuses on testing ADOs with the General Purpose Interface Bus (GPIB) [3] connections to devices. It consists of several function blocks, and has a switch mechanism which enables users to conveniently turn on and off the simulation mode without changing the ADO codes. In the simulation mode, each ADO parameter can be tested using three different sources of data, i. e. random data, file data or log data. Moreover, it contains a special module which automates a particular kind of testing on ADO codes. Testing results are summarized and presented to users for codes analysis.

In this work, a new simulation framework is proposed which covers a different type of ADOs that use the Ethernet connections (including[5] DIGI interface [7]) to devices. The simulation framework adopts a totally different structure. Specifically, it is based on a powerful networking engine called Twisted, which is an event-driven network programming framework developed by the Twisted Matrix Labs [8]. The simulation framework can handle multiple types of devices at the same time.

Together with the simulation structure in [2], they form a more comprehensive simulation environment which covers a large amount of ADOs frequently used by developers.

[5] Since eventually DIGI devices also need to communicate with hardware devices through Ethernet.

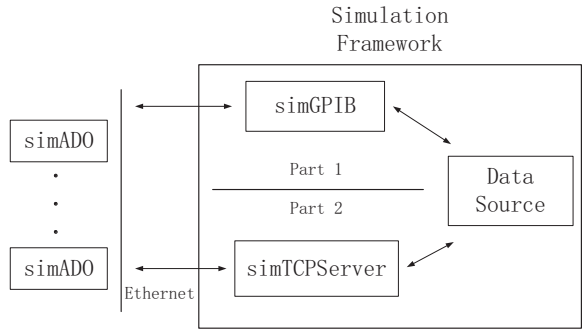

Figure 2: The overall simulation framework consists of two independent parts.

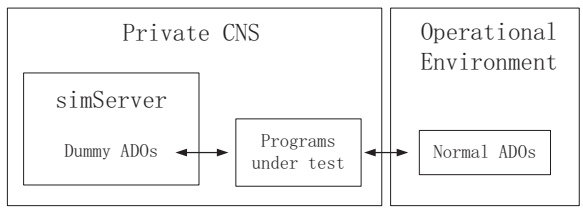

Figure 3: The structure of the simServer.

Figure 2 summarizes the two parts of the new simulation environment.

This work concentrates on the second part. Key use cases in the control system are summarized. The next development phase is discussed.

RELATED WORK

There is another simulation structure in the C-AD control systems called simServer [9]. In this section, we briefly review this simulation work, and compare it with the proposed simulation structure.

simServer

The simServer is a simulation structure that is proposed in the controls system to simulate data for ADO parameters. The structure of the simServer is shown in Fig. 3.

It uses some special features of CNS and "adogen" to allow users to set ADO parameter (including measurement parameters) values. As introduced above, when applications (e. g. PET) need to communicate with an ADO, they first search through CNS to find out where the ADO is running and then make the connection. Moreover, users can create a private copy of CNS, which is operated on users' local machines (so it will not affect the normal operation of the system) and could contain different ADO information. By doing that, users could direct applications to interact with different ADOs without making any change on the applications' side.

"adogen" is the program to convert ".rad" files into C++ codes which are then compiled to generate ADOs. "adogen" has a runtime switch "-dummy", which generates a special version of ADOs (dummy ADOs). Those dummy ADOs have their "get()" and "set()" methods dummied-out,

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA045

i. e. they simply return OK. This feature effectively makes all ADO parameters simple memory containers. Users can write data into dummy ADO parameters (including measurement parameters), so that they can interact with applications without actually connecting to any hardware.

The simServer is a place (a server or specifically, an ADO manager) where it holds those dummy ADOs for applications to access. The simServer simulation structure is useful in situations where a programmer wants to force some particular parameter values (which are not likely to appear in the practical operation of the devices) for testing purposes. For example, giving a voltage measurement of a magnet a high value which exceeds its engineering limit.

The simServer can interact with applications through a special version of ADOs (without the need of any hardware). The building of a simServer is relatively simple and it works for different types of ADOs (since all ADOs are dummy ADOs in the simServer, their methods are dummied-out to be simple memory containers). However, the simplicity of omitting ADOs' code details limits its testing capability. It can only perform tests in the application level, since there is no way for it to test the ADO codes itself (the ADO codes are already modified to be dummy codes), which is more important to the overall system reliability.

The simulation structure proposed in this work can test both the ADO codes level and the application level. Users can run their customized testing data with ADO codes unchanged, and interact with the applications.

TWISTED FRAMEWORK

The simulation structure proposed in this work employs an event-driven networking engine called Twisted from Twisted Matrix Labs [8]. The Twisted framework is asynchronous, therefore it is very efficient and scalable. This section briefly introduces some important facts about the Twisted framework.

Reactor Pattern

The Twisted framework uses a design pattern called reactor pattern [10]. It works like a loop that waits for events to happen and then reacts to them. For that reason, the "reactor" loop is also known as an event loop.

The Twisted framework uses callback [11] to invoke user-defined function codes. It is a fundamental aspect of asynchronous programming with Twisted. Figure 4 shows how the Twisted framework uses callback to invoke users' codes. From the figure, we can see that the users' callback codes are in the same thread as the Twisted reactor loop. Thus, at anytime if the users' codes are running, the Twisted loop has to wait, and vice versa. The reactor loop resumes when the callback returns.

The reactor is the most important abstraction in Twisted. At the center of every program built with Twisted, there is always a reactor loop that making the whole thing go. In other words, writing programs with Twisted means choosing the

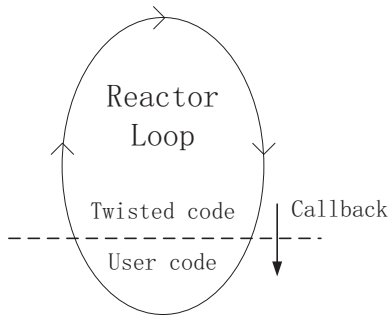

Figure 4: Callback pattern in the Twisted framework.

reactor pattern, which means programming in the "reactive style" using callbacks and cooperative multitasking.

Useful Concepts

The simulation structure in this work is essentially built as a TCP server. It (instead of real hardware) interacts with the ADOs (that using Ethernet connections) for testing purpose. There are several useful concepts in the Twisted framework that simplify building a TCP server greatly.

Transports A Twisted Transport represents a single connection[6] that can send and/or read bytes. The Transport abstraction represents any such connection and handles the details of asynchronous I/O automatically. It is usually common to use the Transport implementations that Twisted provides. This way, the Transport objects will be created automatically whenever the reactor makes new connections.

Protocols As the name suggests, Twisted Protocol objects implement protocols (e. g. TCP, FTP, etc.). Strictly, each instance of a Twisted Protocol object implements a protocol for one specific connection. This makes Protocol instances the natural place to store the accumulated data of partially received messages (since the Twisted framework is asynchronous, the data are received in arbitrary-sized chunks). The way the Protocol instances decide what connections they are responsible for is through a callback method which is called by the Twisted codes with a Transport instance as the only argument. That Transport instance is the connection the Protocol instance is going to use.

Twisted includes a large number of ready-built Protocol implementations for various common protocols [12]. Depending on the demands, it is also common to implement new Protocol classes.

Protocols Factories Each new connection needs its own Protocol instance, and Twisted handles creating new connections. Thus, it will be convenient to also let Twisted make the appropriate Protocol "on demand" whenever a new connection is made. That is the job of Protocols Factories.

As the name implies, Protocols Factories follow the Factory design pattern [13] and they work in a straightforward

[6] The connection can be a TCP connection, I/O over UNIX pipes and UDP sockets, etc.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA045

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

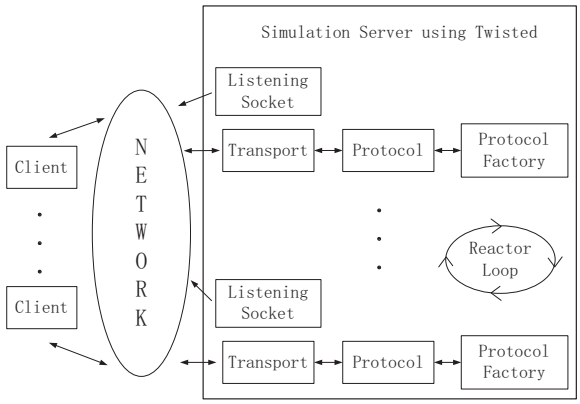

Figure 5: Architecture of the simulator using Ethernet connections.

way. Twisted invokes a Protocols Factories method to create a new Protocol instance for each new connection.

Next, we discuss in detail about the simulation architecture.

SIMULATION STRUCTURE OVERVIEW

The simulator's structure is shown in Fig. 5. It is implemented using the Twisted framework and serves as a simulation server, interacting with clients from the network. The simulation server can handle requests from multiple clients at the same time (clients are usually different types of devices). Each client is served with a Protocol Factory instance, which contains information about how to perform tests for a particular type of device.

The simulation structure supports three testing modes (and a default mode if the testing mode is unspecified), i. e. log mode, random mode, and file mode. There are several features about this simulator that are worth mentioning. First, parameters from the same ADO can run different simulation modes. For example, some parameters may use random mode to test on different data points, while other parameters from the same ADO may use file mode to run user-specific testing data. Users can decide what simulation modes to be used for which parameters depending on the testing purpose. Second, for the log mode, the logged data are fetched from the remote data servers [14]. Using data servers is an efficient way to handle users' requests for logged data. It reads the data quickly and culls the returned data so that it sends the user only the data that can reasonably be distinguished on the display applications (e. g. LogView).

Moreover, users can specify a time delay value, which will be added to the communications between the client and the simulator. The granularity of the time delay can be in second, millisecond, microsecond, or nanosecond[7]. Setting parameter values is also supported in the simulation structure, users can set ADOs' parameters for testing purpose. All of those functionalities will be illustrated with an example below.

Simulation Protocols and Protocols Factory

Twisted provides a rich library of Protocol implementations for various kinds of demands. Since commands coming from devices usually end with a delimiter (e. g. a newline character "\n", or a carriage return "\r"), "LineOnlyReceiver" is adopted as the simulation's Protocol [12], which only receives lines. "ServerFactory" is an implementation of Twisted Protocols Factories [15], which is adopted as the simulation's Protocol Factory (since the simulator is essentially a TCP server).

Simulation Configuration File

Following the same principle as in [2], to generalize the simulation structure to accommodate various types of devices, a simulation configuration file is applied. Each type of device has its own configuration file. The file is written in standard XML format [16], and contains a list of information about how to interact with a particular type of device using user-customized testing data.

To better understand the kind of information a configuration file maintains, consider the following example. There is an ADO called "ampmotion" that is interacting with an applied motion controller [17] through Ethernet.

The first part of the configuration file lists several individual settings, such as the delimiter that ends the commands sent from the device (e. g. a newline character "\n"), the delimiter that ends the responses from the simulator (device uses it to detect messages received), and the time delay value (can be 0).

The second part lists information about all the device commands and the corresponding responses for each simulated ADO parameter. It has several sections. Take the command "IP" for example[8].

Section "CmdResp" lists hardware command elements. Each element has the following information:

- The name of the hardware command (such as "IP");
- The response format for this command (such as "IP={}");
- Number of parameters needed to form the response ("1" for the example, since there is only one pair of "{}" needs to be filled to form a response);
- Simulation mode (e. g. "randRangeInt", which means generating a random integer in a range);
- Simulation mode parameters (e. g. "15000; 20000", which sets the range of "randRangeInt" to be (15000, 20000)).

Section "Default" lists default responses for all the simulated parameters ("16500" for the example "IP"), which will be used if the simulation mode is unspecified when the simulation starts.

[7] The time delay function is implemented using the "sleep" function of the "time" module in Python. The accuracy of the function depends on the underlying Operating System (OS)'s sleep accuracy. On Linux, the granularity of the system clock can be around 1 ms. Using a real-time system will further increase the accuracy of the "sleep" function.

[8] This command is used to query the immediate position of the controller.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA045

Section "Log" lists the information about how to get the logged data for any parameter that uses log simulation mode. It contains the directory of a log file, a start time and a stop time to locate the part of the logged data that are interested, and the ADO parameter name for the command ("positionM" for the command "IP"). The logged data will be played repeatedly (in loops) from the start time to the stop time during the entire simulation.

Section "File" lists the information about how to get testing data from a file for parameters that use file simulation mode. Information includes the directory of the file that contains the testing data, a pair for each parameter consisting of command name and an identifier ("IP;ip" for the example). The identifier "ip" will be used in the simulation to locate the testing data from the testing file.

The third part of the configuration file lists simulation configurations for setting commands. It contains key-value pairs for all the setting commands that will be used in the simulation. The key is the command's format used for setting the ADO parameter, and the value is the command used for getting the ADO parameter value. For the "IP" example, the key-value pair is "FP{d}-IP". "FP{d}" is the setting command's format, which means the command is composed of "FP" plus an integer number. Later in the simulation, the integer number will be used as the response to the query command "IP", and returned to the requester.

Simulation Procedure

The simulation procedure can be described as follows.

First, there is a server configuration file to be created, which contains necessary information to start the simulation server, such as the name of the host the server is running on, the name of the data server, listening port numbers, and the directories of the simulation configuration files.

The simulation server starts based on the information in the server configuration file. Once it starts, a "simData" abstraction instance is created for each type of device. The instance is created with a simulation configuration file as the only argument. It includes the user-customized testing data and appropriate methods to generate and organize responses (in the right format that the device supports) based on the testing data. In other words, the "simData" instance has the necessary knowledge about how to interact with the underlying device using the testing data. Then, a simulation Factory instance is created by loading a "simData" instance. The Factory instance generates new Protocols for new connections and serves as the bridge between the Protocol instance and the simulation data. One Factory instance is responsible for one type of device. The Protocol instance is dealing with all the low level work. It is in charge of handling new connections, receiving commands from ADOs, getting responses from Factory instances, sending responses back to clients, and managing connection lost. One Protocol instance is created for each new connection from the network. The listening socket tells Twisted to monitor the network for new connections on a specific port, and use the Factory to make new Protocol instances for new connections.

Furthermore, users can use Twisted to create and run the simulation server as a daemon process [18], so that the simulation server can continue its services even the terminal session is closed or the user logs out.

Simulation results can be verified through different ways. Viewing the parameter values of the simulated ADOs is an easy and efficient way. Standard controls systems tools such as PET or LogView can be applied.

Key Use Cases

The main use cases of this simulation framework along with the one in work [2] are summarized as follows:

- Improve ADO codes reliability;
- Replace hardware devices when they are not available;
- Test failure conditions without interrupting normal systems operations;
- Automated system and unit test of software;
- Validate the upgrade of software.

FUTURE WORK

Our focus thus far has been on defining the overall structure of an useful control system simulation environment within the framework of the existing, real control system. One of the future goals is to construct a version of the system that can be completely self-contained. This would allow working in an environment in which all aspects of the system can be controlled, without impact on the actual running control system.

Another interesting topic is the combination with machine learning. A possible extension of the simulation architecture is the construction of a special client that monitors both the inputs and outputs of the simulated ADOs. Since the special client has the complete information of both the data feeding in and the data coming out, machine learning techniques such as anomaly detection [19] can be applied with the special client to further test the ADO codes reliability. We imagine this special client to become the main user interface developers could use to construct and perform testing suites and even develop regression testing groups.

REFERENCES

[1] K. A. Brown, "C-AD Controls Systems various notes on history, architecture, and modern systems", Collider-Accelerator Department Documentation, unpublished, 2012.

[2] Y. Gao, T. G. Robertazzi, K. A. Brown, J. Morris, and R. H. Olsen, "A New Simulation Architecture for Improving Software Reliability in Collider-Accelerator Control Systems", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 1261–1265. doi:10.18429/JACoW-ICALEPCS2017-THMPL03

[3] NI-488.2 User Manual, http://www.ni.com/pdf/manuals/370428v.pdf

[4] D. S. Barton *et al.*, "RHIC control system", *Nucl. Instrum. Methods Phys. Res., Sect. A*, vol. 499, no. 2–3, pp. 356–371, Mar. 2003.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA045

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2019). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI.

[5] R. H. Olsen, L. Hoff, and T. Clifford, "Code Generation of RHIC Accelerator Device Objects", in *Proc. ICALEPCS'95*, 1995.

[6] T. D'Ottavio, "Controls Logging System", Collider-Accelerator Department Documentation, unpublished.

[7] DIGI PortServer TS, https://www.digi.com/products/networking/serial-connectivity/serial-device-servers/portserverts

[8] Twisted Matrix Labs, https://twistedmatrix.com/trac/

[9] R. H. Olsen, "simServer Simulation Environment", Collider-Accelerator Department Documentation, unpublished, 2008.

[10] Reactor pattern, https://en.wikipedia.org/wiki/Reactor_pattern

[11] Callback (computer programming), https://en.wikipedia.org/wiki/Callback_(computer_programming)

[12] Twisted base protocols, https://github.com/twisted/twisted/blob/twisted-8.2.0/twisted/protocols/basic.py

[13] Factory (object-oriented programming), https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)

[14] T. D, B. Frak, J. Morris, and S. Nemesure, "Improving Data Retrieval Rates Using Remote Data Servers", in *Proc. ICALEPCS'11*, Grenoble, France, Oct. 2011, paper MOMAU002, pp. 40–43.

[15] Twisted protocol-related interfaces, https://github.com/twisted/twisted/blob/twisted-8.2.0/twisted/internet/protocol.py

[16] Extensible Markup Language (XML), https://www.w3.org/XML/

[17] Si™ Command Language (SCL) Software Manual, https://www.applied-motion.com/sites/default/files/920-0010B_SCL_manual.pdf

[18] Daemon (computing), https://en.wikipedia.org/wiki/Daemon_(computing)

[19] Anomaly detection, https://en.wikipedia.org/wiki/Anomaly_detection