ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBR006

TOWARDS ASYNCHRONOUS CONTROL SYSTEMS, AN ASYNCIO IMPLEMENTATION OF OPC UA USING TANGO GREEN MODES

E.J. Morales*, X. Mercadal, J.A. Ramos, Z. Reszela, A. Rubio, S. Rubio-Manrique†
N. Serra, J. Villanueva, ALBA Synchrotron, Cerdanyola del Vallès, Spain
G. Cuni, Paul Scherrer Institute, Villigen, Switzerland

Abstract

The ALBA Synchrotron (Barcelona, Spain) has been operating as a 3 GeV facility for over 10 years and is now preparing its transition to ALBA II, a fourth-generation light source. As part of this planned upgrade, we are evaluating state-of-the-art technologies that could shape the future of our Tango Control System. In particular, we investigate how asynchronous programming can enhance system responsiveness while reducing latency and resource usage. This study focuses on applying asynchronous communication paradigms at all levels between our Taurus SCADA UIs, Tango Control System and PLC-based systems — used for Equipment (EPS) and Personnel (PSS) Protection as well as automation. In this context, we explore the adoption of OPC Unified Architecture (OPC UA), a self-descriptive industrial standard for secure, platform-independent communication, alongside asyncio, the Python standard library for coroutine-based asynchronous programming, as supported by the FreeOpcUa library and "green" modes of PyTango, the Python binding for Tango Controls. Our goal is to demonstrate a modern, flexible, vendor-independent and high-performance control strategy for ALBA II Control System. We provide a comprehensive comparison and benchmark between the proposed solution and existing PyPLC Tango Device Servers.

INTRODUCTION

The ALBA synchrotron (Barcelona, Spain) [1], a third-generation electron accelerator, is preparing for its transition to a fourth-generation light source. A scheduled shutdown in 2030 will enable the upgrade and restart of the facility. Over the past decade, ALBA has relied on its original design technologies, demonstrating robustness through more than ten years of continuous service. The forthcoming upgrade requires modernizing supporting technologies and adapting the infrastructure to new operational demands.

ALBA Equipment Protection System

The Equipment Protection System (EPS) [2] safeguards the accelerator and beamline subsystems through a network of PLCs [3] interconnected via `Ethernet PowerLink` [4]. It monitors PLC variables and can interlock machine operation in under 50 ms. PLCs are integrated into the ALBA control system using Tango [5], a distributed control framework developed by the Tango Controls Collaboration [6, 7]. Each EPS PLC manages over 400 nodes and 1500 parameters, grouped into analog, floating-point, digital, and actuator types. The `EPS_LIBRARY` [5] standardizes access using 16-bit `StatusRead` and `StatusWrite` words for diagnostics and control, while additional words report CPU and network health.

Communication currently relies on `Modbus TCP` [8, 9], a simple and robust protocol widely used in industry. While effective for small to medium systems, its scalability in large or dynamic environments such as ALBA's EPS is limited, motivating the evaluation of newer approaches. All EPS configuration is centralized in a CSV [10] file mapping nodes to `Modbus` addresses and parameters.

Integration with Tango is handled by the `AlbaPLC` class, which parses the CSV at startup to generate Tango attributes [11]. The lower communication layer `PyPLC` manages `Modbus` transactions, while `AlbaPLC` exposes EPS states, parameters, and events. This modular design separates hardware-level management (`EPS_LIBRARY`) from control-system integration (`AlbaPLC`), enabling reliable and scalable operation.

OPC Unified Architecture

OPC UA (IEC 62541) [12], developed by the OPC Foundation [13], offers a modern, cross-platform, secure, transport agnostic and event-driven communication framework. It supports client–server and publisher-subscriber models, advanced information modeling, and hierarchical data representation. Being self-descriptive, OPC UA can export PLC internal structures directly, eliminating intermediate files and avoiding transcription errors. In this work, we rely on `asyncua` [14], an open-source Python library providing asynchronous OPC UA clients and servers.

Asynchronous Programming and Tango Green Modes

Asynchronous programming puts its focus on allowing multiple operations to progress concurrently without blocking, a paradigm suited for large event-driven systems. `asyncio` [15] is Python's standard library for asynchronous programming. Based on coroutines and event loops, it allows to manage tasks and asynchronous events concurrently while keeping resource usage low. The Python binding of Tango Control System, `PyTango` [16], and its `asyncio` Green Mode have been selected for scalable data handling at ALBA. This approach is focused on improving responsiveness, reducing CPU load, and replacing polling-based communications with data-change event subscriptions.

This article presents the evaluation and benchmarking of a modernization path for EPS communication, comparing the

* emorales@cells.es
† srubio@cells.es

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBR006

legacy Modbus TCP solution with an asynchronous OPC UA approach using asyncua and PyTango, and highlighting the expected benefits for ALBA's fourth-generation upgrade [17].

OPC UA IMPLEMENTATION IN TANGO

At ALBA, an initial OPC UA implementation is under testing for a new pulsed magnet in the storage ring, controlled by a Siemens S7-1500 PLC providing both a traditional socket interface and an OPC UA server. As this was ALBA's first OPC UA deployment, the legacy socket communication was kept as a fallback option.

The implementation, developed in Python 3 with asyncua and PyTango, subscribes to a vendor-provided list of nodes using an asyncua Subscriber object. Incoming data are cached via a callback and trigger a PyTango push_change_event only when the received value differs from the cached one, while attribute reads return the last cached value. Due to server constraints, the minimum publish interval is 500 ms, which defines the subscription parameters. Commands are sent by directly writing to OPC UA nodes, and the server provides methods to verify the last written node and value.

Although the Tango community lists OPC UA device classes in its *Device Classes Catalog* [18], existing developments such as SOLEIL Synchrotron [19] Java implementation using Eclipse Milo [20, 21], or Python 3 implementations at Max IV [22] and Solaris [23] (based on Max IV), they didn't match our constraints for our existing EPS clients. To enable in-place performance evaluation and address potential concurrency issues under high read/write loads, ALBA developed its own AlbaPLC replacement implementation.

This implementation focuses on three main objectives. First, it leverages the asyncua library and PyTango asyncio Green Modes to avoid the deadlocks and timeouts frequently experienced with Modbus during simultaneous read and write operations. Second, it supports self-discovery of PLC data structures, eliminating the need for intermediate files (e.g., CSV) to exchange information between PLC programmers and control engineers, thus reducing the maintenance effort. Finally, it is designed to integrate not only EPS PLCs but also any OPC UA–based solution provided by suppliers, such as safety PLCs, temperature controllers, gas purification systems, or operator interfaces. To achieve this, the implementation adopts a two-layer architecture: a generic PyPLC layer and an EPS-specific AlbaPLC layer.

PyTango Asyncio Green Mode

PyTango *Green Modes* enable cooperative multitasking in the Tango control system by leveraging Python's asynchronous frameworks such as asyncio [24] and gevent [25]. Unlike traditional synchronous implementations, which rely on explicit thread management and can suffer from blocking operations, Green Modes provide efficient task scheduling and improved scalability, particularly for I/O-bound applications.

They allow multiple operations to run concurrently within a single-threaded event loop, reducing the need for complex synchronization and minimizing idle waiting. Transitioning to Green Modes, however, requires adopting async/await patterns and carefully managing event loops to avoid issues such as subtle bugs, resource leaks, or task starvation.

asyncua: An Asynchronous OPC UA Library for Python

asyncua is a Python library for implementing OPC UA (IEC 62541) clients and servers using the asyncio framework. It supports secure communication, node browsing, variable read/write operations, subscriptions and event notifications, being successfully used in scientific environments [26].

Unlike the old synchronous python-opcua implementation, asyncua performs non-blocking operations and handles multiple concurrent connections efficiently, improving scalability and responsiveness while reducing CPU overhead from polling loops. This makes it particularly suitable for large-scale or high-frequency systems, such as accelerator protection infrastructures, where numerous subscriptions and events must be processed with minimal latency.

PyPLC OPC UA Communication Layer

The PyPLC [27] device server is a PyTango Device Server [28] that exploits the versatility of the Python language to create Tango interfaces on-the-fly using the Fandango library [29, 30]. While the original Modbus implementation relied on pre-configuring a list of Modbus addresses to be accessed, this new version of the PyPLC device server relies instead on OPC UA self-descriptive quality. Once OPC UA connection is created at Tango device server initialization time, OPC UA nodes are obtained from the PLC by exploring child nodes from the pre-specified root node, thus generating a node-tree to be converted into Tango attributes and configuration parameters.

This implementation has been designed to be vendor-independent, and to be easily configurable. PLC nodes to be read or subscribed are filtered using lists of regular expression patterns stored as device properties in the Tango database. The PyPLC parent class (fandango.DynamicDS) provides convenient helper methods to convert PLC data types into any Python type, allowing either direct reading of PLC data or the usage of single-line Python code expressions to customize the final value of each attribute.

AlbaPLC Architecture

AlbaPLC device server is a derived subclass of PyPLC that provide a deeper level of introspection into the OPC-UA nodes structure, implementing Python equivalents of the data types defined in the PLC EPS LIBRARY. Instead of just exposing the raw values of OPC UA nodes, it parses timestamps, buffers, alarm ranges, and status bits of each PLC variable to convert this information into Tango qualities and attribute config parameters; that will be send along the

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBR006

node value in the same Tango event. This upper layer of data structures allows to reduce the total number of events to be subscribed from clients, and allows to group all the signals that are part of movable objects (valves, shutters, masks having multiple in/out position sensors and actuators) into single-attribute entities much easier to operate by users.

COMPARING OPC UA VS MODBUS PERFORMANCE

For testing and compare OPC UA and `Modbus TCP`, a dedicated PLC infrastructure has been developed using a B&R `X20CP1584` CPU as used in the ALBA EPS System. The tests were executed using a PLC with OPC UA subscription update of 50 ms and a `Modbus` update thread reading bunches of 112 holding registers with 20 ms time waits between commands. The details of PLC data structures are listed in the Table 1.

Table 1: Test PLC Registers

Nodes	Num. of Registers
Analog Float	51
Analog Integers	64
Digital Registers	84
Movable elements	7
Others	5
Total	**211**

Regarding `Modbus TCP` implementation at `PyPLC`, each type of variable maps to a different set of registers, being the more extensive the Analog Float type, that uses a 16 bit register flag for its status (alarm, warning, interlock, ...), 2*16 bits for its value and 7*16 bits for its settings (alarm ranges, force value and status write). `Modbus TCP` communications are processed every 2 CPU cycles, and are processed in bunches of no more than 120 registers. Types of registers are grouped in memory in order to optimize communications time, these allow to read Analog Input values at a higher frequency (up to 5 Hz) than its settings, which are stored in a different area of memory read every 5 seconds on average.

Advantages and Limitations of Modbus TCP

`Modbus TCP` is valued for its simplicity of implementation, operating as a lightweight client–server protocol with short binary messages and minimal stack requirements. It reuses standard Ethernet infrastructure, supports multiple simultaneous connections, and has low network overhead, making it ideal for small data exchanges with high reliability.

Despite these benefits, `Modbus TCP` presents several drawbacks for large or dynamic systems. Its data model is limited to four basic tables with 16-bit registers, lacking explicit typing or semantic description. The protocol has no native security, provides only restricted event-driven capabilities, and most implementations rely on polling. In practice, scalability is constrained by device connection limits as simultaneous reads/writes increase latency, making Modbus TCP hard to tune for high-volume or high-frequency data streams.

Advantages and Limitations of OPC UA

`OPC UA` offers high interoperability, enabling secure communication between heterogeneous devices and simplifying system integration. The protocol is scalable and flexible, supporting client–server and publisher-subscriber models, and its hierarchical information modeling simplifies data management. Being self-descriptive, it can export PLC internal data structures directly, removing the need for intermediate files (e.g., CSV) and avoiding transcription errors. Combined with `asyncua`, `OPC UA` enables non-blocking, asynchronous communication that enhances responsiveness and optimizes resource use in I/O-bound operations.

However, its adoption in large-scale or time-sensitive control systems requires careful consideration. Without prior know-how, implementation can be complex, demanding precise configuration and specialized training. Asynchronous communication introduces potential concurrency challenges, where poor task coordination may cause race conditions or subtle timing issues. Furthermore, the maximum number of variables that can be subscribed simultaneously is limited by the specific `OPC UA` server implementation, which may constrain performance under heavy subscription loads. Subscription to more than 1000 variables at rates below 500 ms is known to be troublesome, and data to be subscribed must be carefully selected.

Reading Comparison, Modbus vs OPC UA

As shown in Table 2, measurements confirm that `asyncua` (`OPC UA`) outperforms `Modbus TCP` in most read operations, particularly for batch and subscription-based access. Single-node reads are faster with `OPC UA` (4.14 ms vs. 60 ms), and grouping readings in single-call batch operations show even a clearer advantage: reading 206 values on a single `OPC UA` read call takes only 18 ms compared to 342 ms with `Modbus` (that required several calls due to being limited to 120×16 bit registers per command). Larger reads, such as 782 configuration nodes, complete in 325 ms with `OPC UA` versus 626 ms for `Modbus`.

Table 2: Reading Comparison of `asyncua` and Modbus TCP

Operation	Nodes	asyncua	Modbus
Read 1 node	1	4.14 ms	60 ms
Read all nodes	1194	11562 ms	24720 ms
Values batch reading	206	18 ms	342 ms
Settings batch reading	782	325 ms	626 ms
Values subscription	412	74 ms	–
Cache refresh period	412	112 ms	342 ms

For a more realistic approach, in our studies we differentiate the data that needs to be refreshed ("values") from the data that does not change unless written ("settings").

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBR006

Subscription-based access to 412 values is achieved in 74 ms, refreshing the cached values in 112 ms with OPC UA, substantially better than the 342 ms observed with Modbus. Even for full index reads (all values, settings and internal flags), OPC UA completes 1194 nodes in 11.5 s, less than half the 24.7 s required by Modbus. These results highlight the superior responsiveness and scalability of asynchronous OPC UA architecture.

Writing Comparison, Modbus vs OPC UA

We conducted tests comparing the performance of Modbus and OPC UA when writing the position of a movable element, specifically a pneumatic valve controlled by two limit switches, as summarized in Table 3. The measurements included the time required for the Tango Device Server to (i) issue the command to open or close the valve, (ii) receive the first status event once the valve started moving, and (iii) obtain the final value update once the valve reached its end position. These operations were performed while continuous variable updates were kept running in the background.

Table 3: Comparing Writing and Refresh Times While Operating Movable Elements

Protocol	min-max refresh time	average
OPC UA	90 - 270 ms	168 ms
Modbus TCP	170 - 1000 ms	499 ms

Validating that the moving operation was started is a required step; because we observed that, due to its asynchronous nature, OPC UA commands are not necessarily executed in the same order than they are sent to the PLC. Whenever two commands are sent during the same CPU cycle, their execution order may be arbitrary.

OPC UA reacted at 168 ms on average, although with certain variability in response times (90–270 ms). Modbus exhibited generally higher and more irregular response times (170–1000 ms, 499 ms on average), as readback time depended on the cycling across all modbus registers. Across all 100 tests performed, only five instances were observed where the Modbus reply was faster than OPC UA.

WRAPPING UP

The tests demonstrated the reliability and stability of both the Tango Device Servers and the PLCs, since concurrent reading and writing of values using both protocols did not affect the continuous update of other variables. Measurements were repeated under different conditions: writing via OPC UA while monitoring responses from both protocols, writing via Modbus while measuring responses from both, and writing from a Taurus [31] client using a separate Modbus port. No differences were observed when using different methods of writing. Changes on valve position and status bits were monitored using an additional Taurus client subscribed to Tango events. It didn't interfere with neither writing nor reading times.

SYNCHRONOUS VS ASYNCHRONOUS PROGRAMMING

Previous implementations of PLC communications relied on internal polling threads, sending synchronous commands at fixed intervals to refresh each variable. This approach was hard to scale, since the total read time increased proportionally with the number of variables, and performance degraded further when write operations had to be interleaved with reads, often causing timeouts. The problem was exacerbated when multiple clients attempted to read synchronously from the PLC, adding new read cycles to the queue.

We wanted to demonstrate that these limitations are better addressed with an event-driven communication model, where data are pushed from the hardware layer to clients. This approach balances scalability and concurrency by allowing reads and writes to coexist while events are forwarded asynchronously to clients. Although still dependent on server-side load balancing, asynchronous programming frees the communication channel and reduces the risk of timeouts or deadlocks. This freedom comes at a certain risk, as it implies that concurrent actions may be executed simultaneously or not in the same order than expected, adding the need of a careful state machine implementation or semaphores when needed.

To validate that our implementation transmits OPC UA server events to the Tango control system with minimal latency through AlbaPLC, data change rates were emulated at 5, 10, and 20 Hz on the PLC side using internal clocks. On the asyncua client side, a 50 ms data change subscription period captures these signals and queues them, then Tango processes the queue every 50 ms ensuring reliable publishing of updated values and quality to Tango without saturating the channel. Drawing on prior experience with the DDK project, the system was tuned quickly and reached stable operation after few subscription adjustments.

We archived all PLC-generated events using an HDB++ [32] event-based system while running the tests. The received data matched expected frequencies at 5 and 10 Hz, and were limited to 19.28 Hz for the fastest signals over a one-week period. Same signals when archived using Modbus communications were not able to refresh at more than 1.52 Hz, limited by the total refresh time of the Modbus cycle. These results confirmed both the reliability of the setup and compliance with the expected constraints.

CONCLUSIONS

OPC UA vs Modbus TCP

The performance of OPC UA depends strongly on the PLC implementation, with notable differences between vendors and models due to variations in their stacks, scheduling, and available resources. Stable operation requires careful tuning of subscription periods and notification policies on both the server and client sides. Although OPC UA generally outperforms Modbus, it remains constrained by limits on

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBR006

publishing frequency and the number of nodes per connection.

In practice, OPC UA is reliable for medium-size datasets at update rates up to 20 Hz. Beyond this, neither OPC UA nor `Modbus` seem suitable, as both are restricted by design. Such high-frequency use cases correspond more to Data Acquisition than to Protection Systems and are better served by alternative solutions such as socket protocols or deterministic field buses.

In some situations, particularly when OPC UA subscription intervals are limited to around 500 ms, `Modbus` can still deliver faster updates for small sets of variables (1–50). In this specific context—fast access to a reduced dataset on non-performant hardware—simpler protocols such as `Modbus` may remain the preferable choice.

A practical advantage of OPC UA, however, lies in its self-descriptive nature. Data definitions are embedded within the protocol itself, eliminating the need for additional configuration files that must otherwise be generated and maintained across PLCs, Tango devices, and clients. This avoids transcription errors, ensures consistency, and significantly reduces development and integration time. In fact, the combination of this feature with the adoption of the PyTango HL API has reduced the code base of the `AlbaPLC` Tango device by more than 70%.

Synchronous vs Asynchronous Communication

Synchronous polling of hundreds of variables has traditionally been used in hardware communications based on single-channel, point-to-point links such as serial lines. While effective for simple, non time-critical applications, this model becomes inefficient and difficult to scale when the number of variables or the timing requirements increase.

In our evaluation we compared synchronous polled communications with asynchronous approaches at two levels: hardware communication (`Modbus` vs. OPC UA) and higher-level software communication (Tango synchronous mode vs. `asyncio` Green Mode). Although these layers employ different protocols and channels, many of the observed advantages of asynchronous communication apply to both.

Asynchronous communications —on which devices push data to clients on data-change— improved throughput and reduced the chance of losing critical events due to limited reading frequencies. On the other side, clients may have less control over which data are published and at what frequency. As a result, high publication rates may overload clients with unnecessary traffic.

Achieving both fast update rates and meaningful, manageable data streams requires additional strategies. These include buffering mechanisms, event-driven notifications, and filtering and change-detection methods that reduce overhead, prevent redundant traffic, and improve scalability. Both OPC UA and Tango allow clients to configure filtering criteria and publishing frequencies in advance, ensuring that only relevant information is delivered. Our study shows that, when combined, these mechanisms provide fast, reliable, and scalable communications across both layers.

Being a non-deterministic approach, Semaphores and State Machines have been needed to ensure proper order of execution for critical tasks in the PLCs. The execution and status of these tasks is monitored by the events sent by their status bits, ensuring a continuous feedback between client and PLC.

Future Developments

OPC UA is rapidly becoming a standard for industrial communications, extending beyond PLCs to many other hardware and software solutions such as gas analyzers, temperature controllers, databases, and sensors. Furthermore, open-source applications can be exposed as OPC UA servers using libraries such as FreeOpcUA, broadening its adoption. In order to provide a reusable interface to OPC UA devices within the Tango Community, we will publish the generic PyPLC OPC UA implementation in the Tango Device Classes repository [33], with the aim to converge with the rest of the community on sharing a common solution.

Taurus is a GUI framework designed for rapid development and widely used within the Tango community. It supports interaction with multiple protocols and control systems [34] through schema definitions. An OPC UA schema for Taurus could streamline client integration by mapping data from PLCs and other OPC UA sources directly into the Taurus model, ensuring consistency and reducing the need for custom development. Such a schema would also enable rapid prototyping of graphical user interfaces, diagnostic tools, and monitoring applications for OPC UA–enabled solutions, while leveraging the rich Tango/Taurus ecosystem.

Another Tango tool that would benefit from OPC UA and PyTango `asyncio` integration is the Panic Alarm System [35,36]. Since Tango 10.0 [37], new Alarm Events allow device servers to forward alarms to clients immediately via a dedicated channel. Subscriptions to alarm flag changes will enable instantaneous notification of PLC-triggered alarms without intermediate layers, reducing the Panic response time from approximately 500 ms to below 100 ms.

Personal Conclusions

The transition from synchronous to fully asynchronous communications has been a long-term objective of the ALBA Controls Section during the last two decades. ALBA was the first Tango institute to widely adopt direct event pushing in its Device Servers, but progress has been gradual and challenging, as much of the hardware and software stack was not initially prepared for such an approach. The continuous upgrade of operating systems, middleware, and tools has represented a major and sustained effort [38].

With the adoption of technologies such as `ZeroMQ` [39] in Tango and `asyncio` in Python, full asynchronous communication from hardware to clients has now become a reality, allowing developers to integrate fast solutions into tools such as HDB++ archiving, the Panic Alarm System, and the Taurus toolkit. This milestone consolidates years of incremental progress and keeps the ALBA Control System at the fore-

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBR006

front of control system paradigms, laying the foundation for the forthcoming ALBA II accelerator.

ACKNOWLEDGEMENTS

We would like to thank our colleagues in the Tango Collaboration, who have always been receptive in sharing their work on this topic and answering our questions. In particular, we acknowledge the meeting with Johan Forsberg from the MAX IV synchrotron and Corne Lukken from ASTRON, who presented their developments and provided valuable insights. We also thank Guillaume Pichon from SOLEIL for the meeting in which he shared his work and offered useful advice for our implementation.

REFERENCES

[1] ALBA website, https://www.albasynchrotron.es

[2] A. Rubio, G. Cuni, D. Fernandez-Carreiras, S. Rubio-Manrique, N. Serra, and J. Villanueva, "ALBA Equipment Protection System, Current Status", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 1599–1603. doi:10.18429/JACoW-ICALEPCS2017-THPHA096

[3] Programable Logic Controller, https://en.wikipedia.org/wiki/Programmable_logic_controller

[4] Ethernet PowerLink protocol, https://en.wikipedia.org/wiki/Ethernet_Powerlink

[5] S. Rubio-Manrique, M. Broseta, G. Cuni, D. Fernandez-Carreiras, A. Rubio, and J. Villanueva, "Integration of PLC's in Tango Control Systems Using PyPLC", in *Proc. ICALEPCS'15*, Melbourne, Australia, Oct. 2015, pp. 413–416. doi:10.18429/JACoW-ICALEPCS2015-MOPGF140

[6] Tango Controls website, https://www.tango-controls.org/.

[7] T. Juerges *et al.*, "The Tango Controls Collaboration Status in 2023", in *Proc. ICALEPCS'23*, Cape Town, South Africa, Oct. 2023, pp. 1100–1107. doi:10.18429/JACoW-ICALEPCS2023-TH1BCO03

[8] Modbus TCP, https://en.wikipedia.org/wiki/Modbus#Modbus_TCP

[9] Modbus, https://gitlab.com/tango-controls/device-servers/DeviceClasses/communication/Modbus

[10] Comma-separated values file, https://en.wikipedia.org/wiki/Comma-separated_values

[11] S. Rubio-Manrique, G. Cuni, D. Fernandez-Carreiras, Z. Reszela, and A. Rubio, "PyPLC, a Versatile PLC-to-PC Python Interface", in *Proc. PCaPAC'14*, Karlsruhe, Germany, Oct. 2014, paper FPO011, pp. 179–181.

[12] OPC Unified Architecture, https://en.wikipedia.org/wiki/OPC_Unified_Architecture

[13] OPC Fundation, https://opcfoundation.org/.

[14] FreeOPCUA library, https://freeopcua.github.io/

[15] asyncio, https://docs.python.org/3/library/asyncio.html

[16] PyTango library read-the-docs, https://tango-controls.readthedocs.io/projects/pytango/en/latest/.

[17] S. Rubio-Manrique, F. Becheri, G. Cuni, R. Homs-Puron, Z. Reszela, "Preliminary design for the Alba II Control System Stack", in *Proc. ICALEPCS'23*, Cape Town, South Africa, Oct. 2023, pp. 685–690. doi:10.18429/JACoW-ICALEPCS2023-TUPDP076

[18] Tango device classes catalog website, https://www.tango-controls.org/developers/dsc/.

[19] SOLEIL Synchrotron website, https://www.synchrotron-soleil.fr/en

[20] SOLEIL Synchrotron OPC UA implementation, https://gitlab.synchrotron-soleil.fr/software-control-system/tango-devices/communication/opcuaproxy/-/tree/main/src/main

[21] Y.-M. Abiven *et al.*, "Integrating OPCUA Devices in TANGO framework", presented at ICALEPCS'23, Cape Town, South Africa, Oct. 2023, paper THPDP008, unpublished.

[22] Max IV Synchrotron website, https://www.maxiv.lu.se/.

[23] Solaris National Synchrotron Radiation Center website, https://synchrotron.uj.edu.pl/en_GB/.

[24] asyncio library documentation, https://docs.python.org/3/library/asyncio.html

[25] gevent library documentation, https://www.gevent.org/.

[26] W. Duckitt, J. Abraham, "OPC UA EPICS Bridge", in *Proc. ICALEPCS'23*, Cape Town, South Africa, Oct. 2023, pp. 681-684. doi:10.18429/JACoW-ICALEPCS2023-TUPDP075

[27] PyPLC, https://gitlab.com/tango-controls/device-servers/DeviceClasses/InputOutput/PyPLC

[28] Welcome to PyTango documentation!, https://pytango.readthedocs.io/.

[29] S. Rubio-Manrique *et al.*, "Dynamic Attributes and other functional flexibilities of PyTango", in *Proc. ICALEPCS'09*, Kobe, Japan, Oct. 2009, pp. 824–826. https://proceedings.jacow.org/icalepcs2009/papers/thp079.pdf

[30] fandango, https://gitlab.com/tango-controls/fandango

[31] taurus, https://taurus-scada.org/.

[32] D. Lacoste *et al.*, "HDB++, a retrospective on 5+ years using Timescale", presented at ICALEPCS'25, Chicago, IL, USA, Sep. 2025, paper TUMR004, this conference.

[33] InputOutput, https://gitlab.com/tango-controls/device-servers/DeviceClasses/InputOutput/.

[34] A. Hoffstad *et al.*, "Taurus integration to ELT control software", *Proc. SPIE*, vol. 13101, p. 131013R, Jul. 2024. doi:10.1117/12.3019717

[35] S. Rubio-Manrique, G. Cuni, D. Fernandez-Carreiras, and G. Scalamera, "PANIC and the Evolution of Tango Alarm Handlers", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 170–175. doi:10.18429/JACoW-ICALEPCS2017-TUBPL03

[36] panic, https://gitlab.com/tango-controls/panic

ISBN: 978-3-95450-255-4 ISSN: 2226-0358 doi: 10.18429/JACoW-ICALEPCS2025-THBR006

[37] T. Jurges *et al.*, "The Tango Controls Collaboration status in 2025", presented at *ICALEPCS'25*, Chicago, IL, USA, Sep. 2025, paper WEAG001, this conference.

[38] G. Cuni *et al.*, "ALBA Controls System Software Stack Upgrade", in *Proc. ICALEPCS'21*, Shanghai, China, Oct. 2021, pp. 222–229. doi:10.18429/JACoW-ICALEPCS2021-MOPV037

[39] ZeroMQ, https://zeromq.org/.