ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA100

quasar: THE FULL-STACK SOLUTION FOR CREATION OF OPC-UA MIDDLEWARE

P. P. Nikiel*, P. Moschovakos, S. Schlenker, CERN, Geneva, Switzerland

Abstract

quasar (Quick OPC-UA Server Generation Framework) started as OPC-UA server generation framework. The project evolved into a software ecosystem providing OPC-UA support for distributed control systems. OPC-UA servers can be modeled and generated and profit from tooling to aid development, deployment and maintenance. OPC-UA client libraries can be generated and published to users. Client-server chaining is supported. *quasar* was used to build OPC-UA servers for different computing platforms including server machines, credit-card computers as well as system-on-chip solutions. *quasar* generated servers can be integrated as slave modules into other software projects written in higher-level programming languages (such as Python) to provide OPC-UA information exchange. *quasar* supports quick and efficient integration of OPC-UA servers into a control system based on the WinCC OA SCADA platform.

The ecosystem is adapted to different OPC-UA stack implementations and thus can be used as fully free and open-source solution as well as with and for commercial applications.

The contribution will present an overview and the evolution of the ecosystem along with example applications from the ATLAS Detector Control System (DCS) and beyond.

PREVIOUS WORK

The OPC-UA Standard

The OPC-UA standard [1] is very attractive for information exchange between nodes of a distributed control system. The prime advantages are: information modeling, firewall friendliness, portability to different computing platforms, usage of open standards, security and scalability. The standard defines how information is to be exchanged but it leaves the corresponding aspects of software engineering undefined. Therefore software engineers are left without tools to make OPC-UA compliant software. In addition it bears the risk of multiple incompatible software architectures and implementations which are difficult to maintain in a big distributed system which is expected to be used for a decade or longer.

Quasar

quasar was born as an OPC-UA server generation framework [2, 3] to standardize the process of OPC-UA software creation. It was used to create, develop and maintain multiple OPC-UA server projects, primarily for the controls of the ATLAS [4] and other LHC experiments at CERN. Open-sourcing in 2015 [5] permitted its application to numerous projects beyond CERN, e.g. to create OPC-UA servers for commercial power supplies [6].

quasar profits from the Model-Driven Architecture. The models used in *quasar* are called *quasar designs*. *quasar designs* are on a conceptually higher level than OPC-UA information models. A number of dependent artifacts is generated from *quasar designs*: source code, model visualizations, documentation, SCADA integration, among other.

MOTIVATION FOR EVOLUTION

Due to the positive experience with a growing number of OPC-UA applications, the new standard gained wide interest. Consequently, new applications (and thus requirements) followed. Among the requirements the following were of the highest importance: liberation from restrictive software licensing, embedding of OPC-UA software components on embedded targets, distributing processing logic to a chain of serially or parallely connected OPC-UA components, quick integration into SCADA systems and integration of OPC-UA software components into other programming languages.

EVOLUTION OVERVIEW

Liberation from Restrictive Licensing

Initially, the Unified Automation's C++ OPC-UA SDK [7] (further referred to as *UA-SDK*) was supported as the only OPC-UA implementation. However, the *UA-SDK* requires a paid license to develop with, which is considered to be relatively costly, especially for multiple developers. This was considered a significant barrier to wider adoption of OPC-UA. In addition, the source code is closed-source, which poses a significant limitation for build platforms requiring access to the source code, like *Yocto* (detailed in the next subsection).

Thus an attempt was made to find a substitute for the *UA-SDK*. Among free and open-source OPC-UA protocol stacks, *open62541* [8] was considered the most promising choice. A compatibility library called *open62541-compat* [9] was made to adapt the API of *open62541* to the one offered by *UA-SDK*. As the result, while building OPC-UA software components made with quasar, it is possible to select the protocol stack: either *UA-SDK* (paid) or *open62541* (free and open-source).

The *open62541-compat* library covers most of the features achievable with the *UA-SDK* . However, at the time of writing, certain features were not yet available. For example, the internal architecture of the *open62541* leaves much less freedom on how to process incoming OPC-UA requests than the *UA-SDK*. This limits the distribution of requests processing into multiple threads and prevents job queuing. Nevertheless only few applications known to the authors

* Piotr.Nikiel@cern.ch

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA100

were impacted by this limitation; in most applications both *open62541* and *UA-SDK* did equally well.

Advancements in Server Generation

The *quasar ecosystem* as a whole profits from significant advancements in the core *quasar* itself. A summary of few most important items is given, with regard to the initial publication [2, 3].

The introduction of *calculated variables* permits server users and administrators to quickly provide calibrations, conversions, unit changes and introduction of new quantities by defining OPC-UA variables with values based on analytic formula evaluation of other variables. The prime advantage is that recompilation of the server is not necessary - *calculated variables* are loaded from the configuration file which is parsed at runtime.

OPC-UA methods became supported, with any number of arguments and return values. In addition, methods in *quasar servers* can be executed in threads which are separate from the protocol stack's threads, so potentially long-lasting and CPU-intensive tasks can be spawned without affecting the processing of other OPC-UA requests.

The support for restrictions of the configuration schema appeared. In addition, generation of documentation for the configuration schema as well as address-space was added, to the benefit of users and server administrators.

quasar optional modules feature standardizes the way in which external dependencies of servers are used. In addition, a publicly-available repository of *quasar optional modules* was created.

OPC-UA on Embedded Targets

The ubiquity of embedded systems nowadays is visible also in the domain of detector control systems. Often the processing systems used for such devices are powerful enough to permit to embed OPC-UA software components directly within the devices. This is a significant improvement compared to previous approaches when only simple protocols were used for the embedded systems.

quasar was successfully used in multiple embedded software projects. Different build and deployment strategies were tested: cross-compiling, usage of Yocto and native compilation on the embedded target.

Cross-compilation: *quasar* projects can be cross-compiled for an embedded system of choice. The path(s) to the cross-compiler and the *sysroot* [1] are required to be configured and an OPC-UA server executable ready to be run on the target is produced.

Yocto-based: *Yocto* [10] is an open-source project (backed by numerous companies from the embedded sector) which creates highly customized Linux distributions. *Yocto* builds everything from sources (even the compiler used to build the target software is built from source) and therefore the achievable coverage of target systems is very wide.

quasar's build system was extended to support Yocto's CMake-based recipes. *quasar* can be used to create OPC-UA server applications for *Yocto*. Sample recipes for OPC-UA applications as well as all dependencies not handled by built-in recipes are provided.

In addition, *Yocto* is used as the work-horse of higher level environments, e.g. Xilinx's *PetaLinux* [11]. Therefore *quasar* OPC-UA servers can easily be built with *PetaLinux*.

Native Compilation on the Embedded Target: In certain situations the embedded systems were powerful enough to conveniently build the OPC-UA software natively. *quasar* was used in such situations exactly like if an OPC-UA server was being made for a desktop or a server-grade machine.

Note on Resource Footprint Measurements were performed to estimate executable size and memory consumption of a simple OPC-UA server made with *quasar* using the *open62541* as the protocol stack. Two environments were measured: on the armv7l architecture on an embedded board with Zynq SoC using *PetaLinux* and on the x86_64 architecture using *CentOS 7* on a desktop PC. In both cases the protocol stack was linked statically to the executable, while the remaining dependencies were linked to shared objects delivered by the operating system. For armv7l, the executable size was 1.4MB and the RSS[2] was 12M. For x86_64 the executable size was 2.6MB and the RSS was 7.5MB. Note that the usage of RSS is not an ideal measure of memory consumption on architectures in which virtual memory is used and swapping might take place.

Generation of Corresponding OPC-UA Clients

In many situations, an OPC-UA client for information exchange with an OPC-UA server made with *quasar* needs to be embedded into a computer program.

The *quasar* ecosystem delivers a solution to this problem called *UaObjects* (abbreviated UaO) [12, 13]. Using C++ variant of UaO, a client library for C++ can be generated, in which proxy (surrogate) classes are provided for every *quasar class* of the server for which the corresponding client is generated. The proxy classes expose methods to read, write or call OPC-UA variables or methods (respectively) declared per given *quasar class*. In the methods, OPC-UA client code is generated which deals with all aspects such as data encoding, calling the protocol stack, processing errors and other.

UaO users therefore do not need to know much about OPC-UA or its client APIs. The only requirement is that those users have access to the *quasar design* of the server for which the client is generated.

Within the ATLAS Detector, the UaO approach found applications for interconnecting the Data Acquisition (DAQ)

[1] sysroot (also known as rootfs) is the folder corresponding to root directory on the embedded device file system.

[2] RSS, Resident Set Size - part of the memory occupied by a process held in the RAM.

systems with the Detector Control Systems (DCS). Namely, often the DCS runs an OPC-UA server which is responsible not only for controls and monitoring but also has hardware interfaces to configure the detector. However, the configuration data is often in possession of the DAQ system, thus it needs to find its way to hardware controlled by DCS via OPC-UA. Therefore, an OPC-UA client generated by UaO is integrated into the DAQ software.

Chaining Multiple Clients and Servers

quasar can be used to create both OPC-UA servers and corresponding OPC-UA clients. Therefore, it is possible to easily create multi-stage information exchange in which one OPC-UA server integrates an OPC-UA client which connects to another OPC-UA server. Such an arrangement is depicted in the Figure 1.

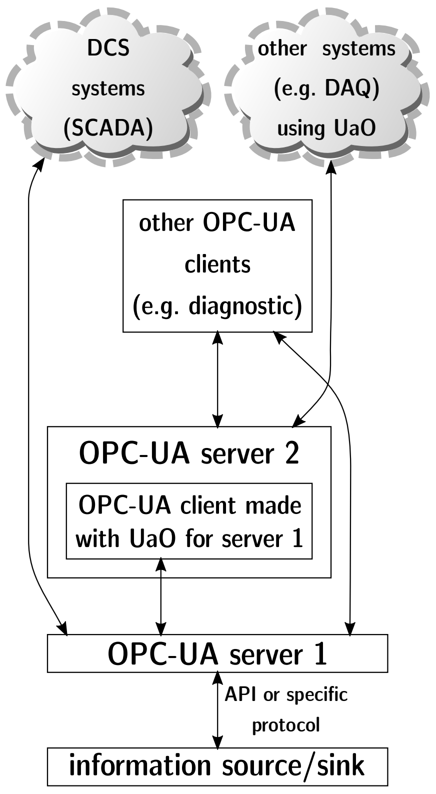
DCS systems (SCADA)
other systems (e.g. DAQ) using UaO
other OPC-UA clients (e.g. diagnostic)
OPC-UA server 2
OPC-UA client made with UaO for server 1
OPC-UA server 1
API or specific protocol
information source/sink

Figure 1: An example of chained servers and clients. The client generated using UaO for server 1 is used from within of the server 2. All arrows without label indicate OPC-UA.

The possible use cases for such an architecture are:

- providing different levels of abstraction for different stages of information exchange.
  For example, server 1 (in the figure) exposes data at very low level of abstraction (e.g. data read directly from hardware, like contents of registers of a chip). Server 2 processes data such that they are exposed on much higher level of abstraction, for example as results of measurements which can be expressed in physical quantities (volts, amperes, etc).
- intermediate data processing.
  For example, server 2 performs statistical operations (e.g. average, median, FFT, etc.) which significantly reduce the amount of published information.
- separation of management/maintenance/responsibility.
  For example, server 1 is within the responsibility of the experiment infrastructure while server 2 is within the responsibility of data processing teams. The separation allows for the modifications of the algorithms used in server 2 without impact on server 1, and especially without impact on other clients that server 1 might have.

Both servers of the example can be created using *quasar*. The OPC-UA client made for server 1 and used in server 2 can be obtained using UaO.

The schema can be much more advanced, for example: one server might connect to many different clients and the number of processing stages might by freely chosen.

A more realistic example of such an application is given in the examples section of this contribution.

Quick Integration into SCADA Systems

Integration of OPC-UA connectivity to a SCADA system depends on chosen SCADA platform. For the WinCC OA SCADA [14], which is the chosen SCADA for all Detector Control Systems of LHC experiments, the OPC-UA connectivity is achieved by configuring so-called peripheral addresses[3] per each data point element[4] which is mapped to a given OPC-UA variable. Such task might often be tedious, especially for big systems. In addition, coherency between peripheral addresses and OPC-UA address-space must be maintained. Both factors prompt for automation of peripheral address assignment.

The *quasar* extension Cacophony [15] allows for the automatic creation of SCADA data structures and peripheral addresses using the *quasar design* and configuration file of the corresponding server by generating SCADA program instructions (i.e. CTRL code for WinCC OA). The approach saves relatively big effort for control system developers, because all OPC-UA servers made with *quasar* can have their SCADA counter-part done at negligible cost.

Integration into other Programming Languages

All components of the *quasar* ecosystem are primarily oriented towards the C++ programming language. However, often the integration into other programming languages is necessary. Since at the time of writing Python was perceived to be the most common programming language into which the OPC-UA server needed to be integrated, the *quasar* ecosystem was extended with a module called Poverty [16].

[3] In WinCC OA, a peripheral address for OPC-UA uniquely identifies given variable in a chosen OPC-UA server. In addition, supplementary data to ensure connectivity is also often there, like the subscription to be used.
[4] A data point element is an equivalent of a process variable in the WinCC OA architecture.

The Poverty module is capable of generating a dynamically-loaded library per chosen *quasar design* which can be imported into any Python program. The library delivers methods to manage (e.g. start) an OPC-UA server for which it was created as well as methods to operate on its address-space. Note that in this scenario the Python application is considered a parent application to the *quasar*-made OPC-UA server.

However, the Python application is limited to publishing or reading information to or from the address space while the embedded OPC-UA server cannot be used to invoke parent application Python code.

From a user point of view, equipping a Python program with an OPC-UA server (for example, to publish data from Python over OPC-UA) only requires to use the Poverty module to obtain the library, load it into Python, start the server and call a function to publish data. It is therefore a very efficient approach for integrating OPC-UA servers into Python[5].

APPLICATIONS

At the time of writing, 16 OPC-UA server projects were used in production or were in pre-production use (each of the projects is usually deployed in multiple instances - with different configuration - on different nodes of the distributed control system). Many more are in the development phase awaiting future upgrades of the ATLAS detector.

Among these projects, the server for the Slow Controls Adapter (SCA) OPC-UA is the most interesting example application [17]. It is the most advanced OPC-UA server in terms of *quasar* features used (different types of variables and methods, advanced threading, relatively large OPC-UA address-space etc.).

In addition, the project profits from many extensions detailed in this writing. In the New Small Wheel (NSW) upgrade project, UaObjects is used to provide OPC-UA clients for DAQ software performing detector configuration. The DAQ configuration process is run in parallel with the DCS control and monitoring activities with both systems accessing the same SCA instances. In this case the concurrency is handled entirely by the OPC-UA server. In the Liquid Argon (LAr) upgrade project, UaObjects is used to make an OPC-UA client for a second-stage OPC-UA server called LAr LTDB Peripheral OPC-UA server. The peripheral server provides high-level abstraction of system-specific electronics boards containing several SCA chips; each command sent to the peripheral server translates into multiple commands sent to the generic SCA OPC-UA server. This gives freedom of development (i.e. separation of responsibility) to the LAr software developers because the generic server (SCA OPC-UA) is used by multiple several other ATLAS upgrade projects.

[5] Note that one can find OPC-UA implementations written purely in Python. The advantage of using *quasar* is that one stays within the *quasar* ecosystem, so it is possible to quickly integrate such an embedded OPC-UA server into a SCADA system (using aforementioned Cacophony module) or use (aforementioned) UaObjects to make a client for it.

Cacophony (detailed above) was used to create SCADA integration code, for rapid integration of SCA connectivity into SCADA projects for the NSW detector. In addition Cacophony is planned to be used to integrate the aforementioned LAr LTDB Peripheral OPC-UA server into the DCS.

Other OPC-UA software solutions created with *quasar* include:

- OPC-UA servers for commercial industrial power supplies used by all LHC experiments: CAEN, Iseg, Wiener [6],
- OPC-UA server for the monitoring and control of ATCA shelves, compliant to the PICMG standard,
- several servers for specialized hardware in ATLAS,
- OPC-UA servers for integration of system-on-chip boards into ATLAS DCS: gFEX and eFEX OPC-UA servers,
- ATLAS version of Wiener VME crates OPC-UA server,
- Generic IPbus [18] server,
- Generic SNMP server.

FUTURE OUTLOOK

Different ideas were recognized as future *quasar* ecosystem development directions. Bridging the gap of supported features between *UA-SDK* and *open62541* is important because in certain applications only the latter can be used (e.g. due to licensing). Support for OPC-UA *events* could help to cleanly implement event-type notifications. Internally within *quasar*, the code generation engine will shift from XSLT (majority today) to Jinja2 [19], which is easier to maintain (ongoing development). *quasar* servers will also be able to load arbitrary additional OPC-UA address-spaces to support alternative ways of information modeling.

CONCLUSION

Since the initial publication, *quasar* evolved significantly and well-beyond sole server generation framework. Nowadays, control systems made of multiple OPC-UA clients and servers, potentially involving many stages of data processing, can be made with *quasar*. Such systems might involve software components written in programming languages different from C++ (e.g. Python) with seamless integration with the other components. The Cacophony extension frees control system developers from tedious task of SCADA systems configuration to provide OPC-UA connectivity. Multiple new features seen in the core *quasar*, like *calculated variables*, bring added value to server users and administrators without need to write any code.

quasar was successfully applied to a wide range of applications; many of the applications are seen in the ATLAS DCS. Beyond the DCS, applications are seen CERN-wide and also in different projects collaborating with CERN.

Certain applications are in production for more than 4 years and no design flaws or instabilities were found which could be attributed to the *quasar architecture*. Multiple new applications will enter production in the years to come.

REFERENCES

[1] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Berlin: Springer, 2009.

[2] P. P. Nikiel, B. Farnham, V. Filimonov, and S. Schlenker, "Generic OPC UA Server Framework," J. Phys.: Conf. Ser., vol. 664, no. 8, 082039. 8 p, 2015. doi:10.1088/1742-6596/664/8/082039

[3] S. Schlenker, B. Farnham, P. P. Nikiel, C.-V. Soare, D. Abalo Miron, and V. Filimonov, "quasar - A Generic Framework for Rapid Development of OPC UA Servers", in *Proc. ICALEPCS'15*, Melbourne, Australia, Oct. 2015, pp. 602–605. doi:10.18429/JACoW-ICALEPCS2015-WEB3O02

[4] A. Barriuso Poy *et al.*, "The detector control system of the ATLAS experiment," *Journal of Instrumentation*, 2008. doi:10.1088/1748-0221/3/05/P05006

[5] the quasar team. (2019). quasar: Quick OPC-UA Server Generation Framework, https://github.com/quasar-team/quasar

[6] B. Farnham, F. Varela, and N. Ziogas, "A Homogenous Approach to CERN/vendor Collaboration Projects for Building OPC-UA Servers", in *Proc. ICALEPCS'17*, Barcelona, Spain, Oct. 2017, pp. 1352–1355. doi:10.18429/JACoW-ICALEPCS2017-THPHA009

[7] Unified Automation GmbH. (2019). C++ Based OPC UA Client and Server SDK (Bundle), https://www.unifiedautomation.com/products/server-sdk/c-ua-server-sdk.html

[8] open62541 contributors. (2018). open62541: an open source implementation of OPC UA, https://open62541.org/

[9] P. P. Nikiel *et al.* (2019). open62541-compat: adapts open62541 API to Unified Automation C++ OPC-UA Toolkit API, the quasar team, https://github.com/quasarteam/open62541-compat

[10] Linux Foundation. (2019). Yocto project – it's not an embedded linux distribution – it creates a custom one for you, https://www.yoctoproject.org/.

[11] Xilinx Inc. (2019). Petalinux tools, https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html

[12] P.P. Nikiel and K. Korcyl, "Object mapping in the OPC-UA protocol for statically and dynamically typed programming languages," Computing and Informatics, vol 37, September, 2018, pp. 946-968, ISSN: 2585-8807, http://www.cai.sk/ojs/index.php/cai/article/view/2018_4_946/915 , doi:10.4149/cai_2018_4_946

[13] P. P. Nikiel. (2019). UaObjects client generation facility for C++, for Quasar, https://github.com/quasar-team/UaoForQuasar

[14] ETM professional control GmbH, "Modern, efficient and flexible simatic wincc open architecture v3.15," ETM professional control GmbH, a Siemens Company, Tech. Rep. 6ZB5370-1EG02-0BA0-V2, Jan. 2017. https://siemens.com/wincc-open-architecture

[15] P. P. Nikiel. (2019). Cacophony generates WinCC OA bindings for quasar-based OPC-UA servers, https://github.com/quasar-team/cacophony

[16] P. P. Nikiel. (2019). Poverty, a quasar module for trivial integration of quasar servers into Python, https://github.com/quasar-team/Poverty

[17] P. Moschovakos, P. P. Nikiel, S. Schlenker, H. Boterenbrood, and A. Koulouris, "A Software Suite for the Radiation Tolerant Gigabit Transceiver - Slow Control Adapter", presented at the ICALEPCS'19, New York, NY, USA, Oct. 2019, paper WEPHA102, this conference.

[18] C. G. Larrea et al., "IPbus: A flexible ethernet-based control system for xTCA hardware," Journal of Instrumentation, vol. 10, no. 02, pp. C02019–C02019, Feb. 2015. doi:10.1088/1748-0221/10/02/c02019

[19] A. Ronacher. (2014). Welcome | jinja2, http://jinja.pocoo.org/.