QUASAR - A GENERIC FRAMEWORK FOR RAPID DEVELOPMENT OF OPC UA SERVERS

P. P. Nikiel, B. Farnham, S. Schlenker, C.-V. Soare, CERN, Geneva, Switzerland
V. Filimonov, PNPI, Gatchina, Russia
D. Abalo Miron, University of Oviedo, Spain

Abstract

This paper describes a new approach for generic design and efficient development of OPC UA servers. Development starts with creation of a design file, in XML format, describing an object-oriented information model of the target system or device. Using this model, the framework generates an executable OPC UA server application, which exposes the per-design OPC UA address space, without the developer writing a single line of code. Furthermore, the framework generates skeleton code into which the developer adds the required target device/system integration logic. This approach allows both developers unfamiliar with the OPC UA standard, and advanced OPC UA developers, to create servers for the systems they are experts in while greatly reducing design and development effort as compared to developments based purely on COTS OPC UA toolkits. Higher level software may further benefit from the explicit device model by using the XML design description as the basis for generating client connectivity configuration and server data representation. Moreover, having the XML design description at hand facilitates automatic generation of validation tools. In this contribution, the concept and implementation of this framework named `quasar` (acronym for **q**uick OPC **UA** server gener**a**tion f**r**amework) is detailed along with examples of actual production-level usage in the detector control system of the ATLAS experiment at CERN and beyond.

INTRODUCTION AND MOTIVATION

Distributed control systems require middleware – software which transfers data between system components. The ATLAS Detector Control System (DCS) [1] is an example of such a distributed control system, organized as a hierarchical mesh of heterogeneous components. The middleware must be capable of handling numerous data models, while being portable and performant at the same time. For the ATLAS DCS, OPC Unified Architecture (further on: UA) [2] has been selected as its new standard middleware [3] for device integration mainly due to its object oriented design and platform independence. A common approach to create UA servers for the various device types allows to reduce development and maintenance costs.

Apart from obvious common functionality in which identical software parts were identified (such as server startup code, logging implementation etc.), it became evident that development efforts could be largely reduced if the data model was considered a parameter of a generalized UA server. Such a data model, augmented with additional information, is subsequently called *design*. If the format of the design is sufficiently rich to describe and model (potentially complex) subsystems, big parts of an UA server implementation may be automatically created (generated). Thereafter, hand-written custom code is only necessary for providing high level ‘business logic’ between the generated parts and the handling of the specific subsystem type (e.g. a hardware access library or protocol implementation). Such hand-written code may be very complex depending on its functionality requirements. We chose to call this code *device logic*.

In the following sections we explain the approach of generating UA servers from the preparation of a server design up to obtaining a functional application.

QUASAR ARCHITECTURE

Figure 1 gives an overview of the different layers of `quasar` put into context. Controllable devices or systems are accessed using their specific access layer – often provided together with the specific device. The device logic layer functions as interface with the high level layers of `quasar` which comes in several modules covering different functionality aspects. The address space module lies on the UA end of the server, exposing data towards UA clients, and is implemented using a commercial UA SDK [4]. A configuration module facilitates address space and device instantiation and the definition of their relations. XML is used as configuration format backed by XML schema definitions. A XML schema to C++ mapping generator (here: `xsd-cxx`) is used to build actual instances from configuration files. An additional subsystem called ’calculated items’, operating entirely in the address space, enables creation of new variables which are derived from existing ones using mathematical functions. `quasar` comes further with optional modules such as component based logging, certificate handling, server metadata and embedded python processing.

MODELLING DEVICES OR SYSTEMS

In the generic approach of `quasar`, an object oriented model was chosen for two reasons: object orientation is well known and widely understood, and UA itself follows the object orientated paradigm. The purpose of modelling is to establish a comprehensive device or protocol characterization using classes, variables, methods and the relations between them. Classes are types of particular objects. Variables belong to classes and are factual vectors of data while class methods process the associated data. The purpose of relations is to model aggregations and type hierarchies.

Once the model is prepared, it has to be codified in a common format – we call this the *design file*. `quasar` uses the

Copyright © 2015 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-148-9

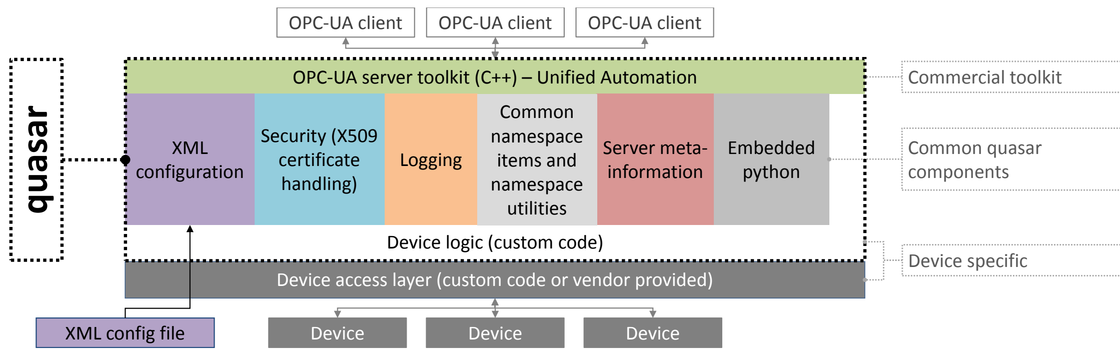

Figure 1: Overview of the quasar components.

XML format for its designs and comes with a ready made XML Schema allowing for easy design creation with commonly available XML editors. A format was required which parametrizes the generic server such that specific instances (design files) match particular subsystems/device types. The most important ingredient of such a design file is the data model handled by the server (discussed in the previous section) augmented with processing-specific attributes.

The first mandatory attribute is capturing the difference between distinct classifications of variables:
Cache Variable: the factual data resides in RAM. Since accessing RAM-based data is a primitive operation for computers, get/set/monitor are just trivial operations handled behind the scenes by the UA toolkit and UA stack.
Source Variable: the factual data resides in undisclosed location (might not even be RAM-based and for our use cases it typically was outside given server computer). An access interface is necessary to read or modify such data, for which glue logic has to be coded in the server. Moreover accessing such data may be a very time-consuming process and often has to be executed in separate thread of execution. Note that these classifications are not seen from the perspective of an UA client, all variables are simply queried (using write/read UA transaction types) or monitored (by creating a monitored item for this variable). However at the UA server implementation side it is beneficial to differentiate between classifications. Another important design attribute is the handling of concurrence inside the UA server. quasar provides capabilites to model domains of mutual exclusion to prevent race conditions in case two objects were to be accessed at the same time.

Finally, the configuration of objects needs to be specified. By configuration we understand a set of values that do not change between creation of an instance and its deletion. The primary configuration parameter is an unique object name, allowing e.g. to traverse the hierarchy of objects using dot as a separator. Additionally, objects often require additional configuration by assigning values to specific attributes, either constants or e.g. for initialization purposes. This is also handled in the design stage through items called "config entries". Config entries belong to classes.

In summary – the design represents the description of types while the configuration is the description of instances.

DESIGN TRANSFORMATION

quasar generates a number of distinct elements based on the server design:
- source code (mostly C++),
- dependent XML schemas,
- design-dependent parts of the build system,
- visualizations of the object structure,
- UA address configuration for quick integration into higher level control system layers (e.g. SCADA system),
- additional utilities (e.g. for testing the address space).

Almost all of these tasks are achieved by XSLT transforms, either to text output (e.g. C++ code) or to XML (e.g. XML schema). Figure 2 illustrates these transformations.

The full procedure of creating an UA Server using the framework is as follows:

1. Create or modify the design.
2. Request creation of device logic stubs for classes defined in the design (initially empty stub implementations are generated, thereafter existing implementations are merged with respect to the new design).
3. Extend device logic stubs by providing factual implementation.
4. Build the server (e.g. by using the provided CMake based build system).
5. Develop by re-iterating the steps 1 $\rightarrow$ 4.

In the following, the individual transformation steps performed by quasar are detailed:

Generation of Address Space C++ classes: For each class in the server design, one C++ class is generated, conforming to the interface provided by the UA Toolkit (therefore instances of these classes can be directly "injected" into the server address space). For every variable of the class, appropriate setters, getters and/or write/read handlers are generated; this ensures that code outside of the Address Space module (typically, hand-written code in the Device Logic module) can interface with the address space class using straightforward C++ function calls.

Information model: UA has rich modelling capabilities from which smart UA clients may profit. quasar exposes

ISBN 978-3-95450-148-9

Copyright © 2015 CC-BY-3.0 and by the respective authors

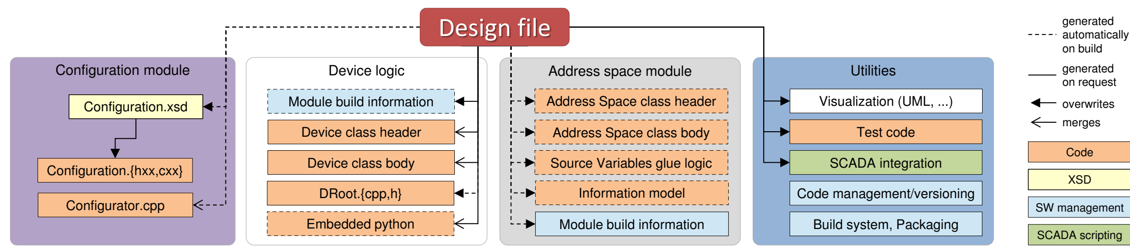

Figure 2: Transformation diagram illustrating the software generation or modification process for a given design file.

the information model derived from the design both in the generated code and within the UA address space at server runtime.

Generation of the configuration XSD schema: Each class from the design is transformed into a complexType in an XSD schema while aggregation relations between classes are respected (e.g. complexType may have a sequence of elements of a different class). Moreover, config entries specified in the design which provide configuration data to specific instances become attributes within the given relevant complexType.

Generation of the configuration loader: C++ code is generated to handle parsing a given XML configuration file to create instances of objects at the startup of the server. This code builds on top of code generated using xsd-cxx with Configuration schema as a parameter. Furthermore, an additional validator is generated to check for constraints that are not easily explained through Configuration schema but easy to explain through server's design.

Device Logic transformation: The Device Logic is the only `quasar`-provided module in which the developer is expected to write C++ source code – the device specific implementation – starting from the generated stubs. The stubs contain skeleton classes and methods according to the design along with an interface to the corresponding address space items. As development progresses, the design may be fluid; classes may be added or removed; variables may change type or perhaps are suppressed completely. After a change of the design file, the developer re-runs the Device Logic generation. If user-modified class sources exist already, the hand-written code will not be overwritten – a merge tool opens to facilitate the adaption of code to the new design.

SCADA integration code transformation: Additionally, `quasar` may generate tools which let the server be easily integrated into some SCADA systems. The typical use case at CERN is as follows: scripts are generated which allow creating the corresponding data structures and their UA addresses within the SCADA system (here: Siemens WinCC OA). This step may not be necessary if a SCADA is used which provides information model aware UA clients.

ADDITIONAL TOOLS FOR DEVELOPERS

Significant effort has been invested to avoid duplication of work for `quasar` users. Thus a number of tools are provided, helping to carry out the following tasks:

- Visualizing object structures: an UML-like diagram creator is provided which helps to visualize the design.
- Validating and upgrading design files.
- Managing consistency of source files: a tool is provided which ensures that source files are properly versioned and that certain files (e.g. XSLT transformations) are not accidentally modified.
- Building binaries: a build system based on CMake is provided along with pre-configured toolchains for several platforms such as `x86_64` or ARM-based Linux and Microsoft Windows.
- Creating installers: a preconfigured spec file for creating RPM packages.
- Testing the address space: a tool is added which continuously pushes random data into the address space. This can be used e.g. to test client mappings and server/client performance under specific conditions.

EXAMPLES

At the time of writing, `quasar` had already been used to create 11 different UA server implementations which cover numerous use cases, applications and various subsystems which are interfaced. These servers are currently in production use at CERN (in numerous instances). We briefly discuss two of these servers – the VME crates server and SNMP server – as their architecture and use case are very different demonstrating the flexibility of the `quasar` approach.

VME crates server: The VME crates UA server implements full monitoring and control of specific VME crates using a proprietary, polling-based communication protocol [5] via CAN bus interfaced to a COTS rack server of the control system. The object model of the design is defined by CAN buses formed by a chain of VME crates which in turn contain channels, fans and temperature probes, c.f. the `quasar`-generated class hierarchy diagram (Fig. 3). Each of these elements (buses, crates, etc.) is to be hierarchically declared in the configuration file allowing a variety of configurations, from very simple single-crate systems up to multi-bus, multi-crate systems with thousands of channels.

From the hardware point of view, each crate has a communication module which has to be polled for the status of the crate itself and its channels and sensors. From the device logic point of view, a software entity at the server side, called communication controller, manages communication between factual crates and the device logic of the UA server.

Copyright © 2015 CC-BY-3.0 and by the respective authors

ISBN 978-3-95450-148-9

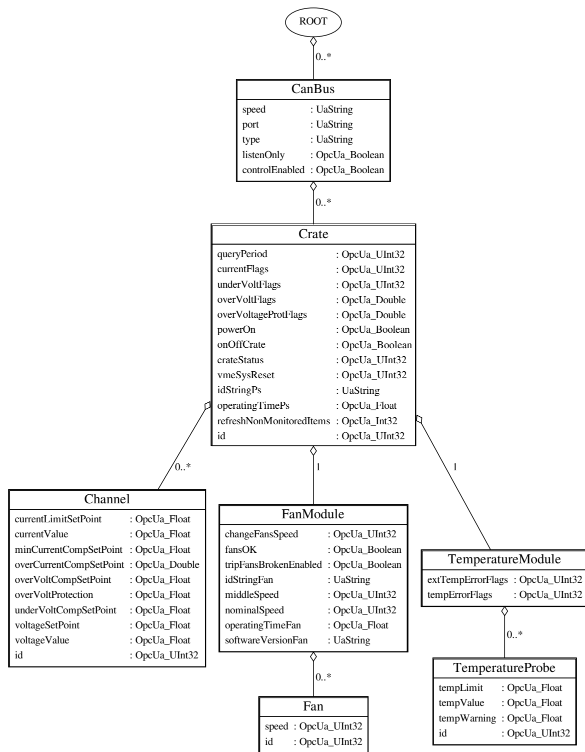

Figure 3: Generated design diagram of the VME server.

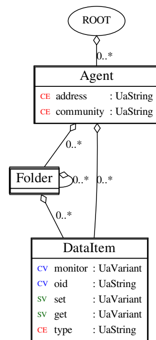

Figure 4: Generated design diagram of the SNMP server.

Even though the crate communication protocol is polling based, the software of the device logic is event based and as soon as new data arrives from a crate, it is pushed into variables of the UA address space and then further to any subscribed UA client – in UA terms "monitored data". The server uses a “CAN interface” module – an optional `quasar` module – a hardware access library supporting CAN interfaces from numerous vendors. A second, server-specific module implements the specific VME crate communication protocol facilitating encapsulation of data in the server device logic. The VME crates server proved to work stably in production and is used in a wide span of configurations – the biggest being a system of 62 VME crates of the ATLAS trigger and data acquisition system with ~5000 channels for which the server CPU load and memory consumption is negligible on a modern server computer.

SNMP UA server: The SNMP (Simple Network Management Protocol) is an Internet-standard protocol for managing devices on IP networks. The server exposes a tree-shaped address space in which data items are basic building blocks (leaves of the tree). Figure 4 shows the design of the server: each data item is bound to one SNMP object identifier in the configuration. Each read/write request coming from an UA client is transformed into a get/set SNMP operation.

Compared to the previous example where cached data coming from the hardware is pushed into memory, this server makes extensive use of Source Variables, which assumes that only the data provider (here: SNMP agent) has the most up-to-date contents of variables. Since a SNMP transaction is a blocking operation (which, compared to in-memory access, may fail), appropriate processing has to be estabilished in order to avoid blocking the whole server while transactions are ongoing. `quasar` automatically generates the corresponding code: each SNMP transaction is spawning a job belonging to a thread pool and synchronization might be automatically applied depending on the synchronization attributes of each hierarchy level in the design file.

CONCLUSIONS

`quasar` has been already successfully used for the creation of a number of UA servers by non-expert developers which demonstrates its advantages: versatility and efficiency of development. The former has been proven by a vast span of applications, from custom devices to generic designs for well-known protocols. The efficiency of the development process becomes evident by the reduction of development efforts since up to 90% of source code can be generated which results in a lower chance of bugs being introduced and at the same time achieving better source code managability. `quasar` doesn't add any overhead compared to classic server development using an UA toolkit only and thus no performance penalty is expected nor observed. Further exploitation of `quasar` at CERN and beyond is envisaged while its functionality being further expanded.

REFERENCES

[1] Barriuso Poy A, Boterenbrood H, Burckhart H J, Cook J, Filimonov V, Franz S, Gutzwiller O, Hallgren B, Khomutnikov V, Schlenker S and Varela F “The detector control system of the ATLAS experiment”, Journal of Instrumentation, Vol. 3, May 2008, doi:10.1088/1748-0221/3/05/P05006.

[2] The OPC Foundation, “OPC Unified Architecture”, http://opcfoundation.org/opc-ua/

[3] Nikiel P P, Farnham B, Franz S, Schlenker S, Boterenbrood H and Filimonov V “OPC Unified Architecture within the Control System of the ATLAS Experiment”, Proceedings of ICALEPCS2013, San Francisco, CA, USA, p 113-6.

[4] Unified Automation GmbH, “C++ based UA Server SDK”.

[5] W-IE-NE-R Plein & Baus GmbH, “CAN-BUS Interface for W-Ie-Ne-R Crate Remote Control”.

ISBN 978-3-95450-148-9

Copyright © 2015 CC-BY-3.0 and by the respective authors