Journal of Physics: Conference Series **664** (2015) 082039 doi:10.1088/1742-6596/664/8/082039

Generic OPC UA Server Framework

Piotr P Nikiel[1], Benjamin Farnham[1], Viatcheslav Filimonov[2,3] and Stefan Schlenker[1]

[1]CERN, Geneva, Switzerland
[2]PNPI, Gatchina, Russia

E-mail:
piotr.nikiel@cern.ch
benjamin.farnham@cern.ch
viatcheslav.filimonov@cern.ch
stefan.schlenker@cern.ch

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 necessary logic for integration to the target system or device.

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 OPC UA server 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 is detailed along with examples of actual production-level usage in the detector control system of the ATLAS experiment at CERN and beyond.

1. Introduction

Distributed control systems require middleware – software which transfers data between components of a distributed system.

The ATLAS Detector Control System (DCS)[1] is an example of such a distributed control system. Being organized as a hierarchical mesh of often heterogeneous components, the middleware must be capable of handling various data models, while being portable and performant at the same time.

For the ATLAS DCS, OPC Unified Architecture (UA)[2] has been selected as its new standard for middleware[3]. Thus it is necessary to provide OPC UA servers to numerous types of subsystems integrated into the DCS. A common approach to create these OPC UA servers allows to reduce development and maintenance costs, and provides added value in reusable software components and technology.

[3] Present address: CERN, Geneva, Switzerland

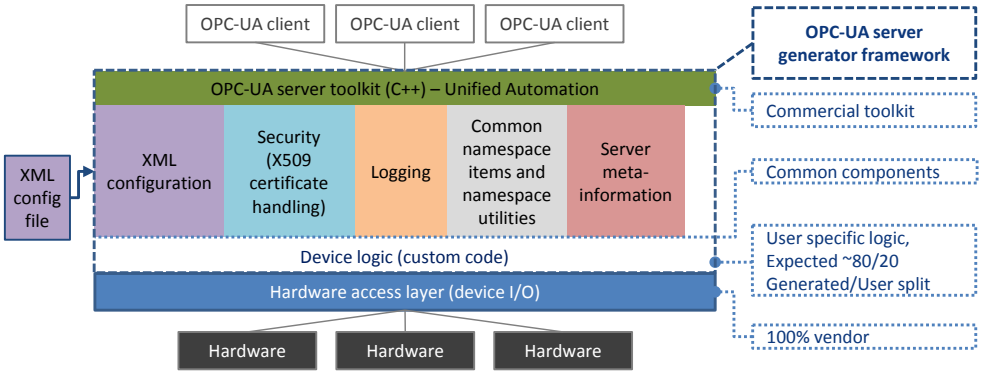

Figure 1. Overview of the generic server framework components.

Apart from obvious common functionality in which identical software parts were identified (e.g. server startup code, logging implementation, and others), it became evident that considerable saving of development efforts could be achieved if the data model of a given subsystems server was considered a parameter of a generalized OPC UA server. Such a data model, augmented with additional information, is called design in this paper.

If the format of the design is sufficiently rich to describe and model (potentially complex) subsystems, then outstanding parts of their OPC UA servers may be automatically created (generated). Thereafter, hand-written custom code is only necessary for providing high level business logic between the generated parts and software handling the specific subsystem type (e.g. a hardware access library or protocol implementation). Such hand-written code may obviously be very complex in order to provide additional functions. We chose to call this code *device logic*.

In the following chapters we explain the approach of generating OPC UA servers starting with the preparation of a server design and up to obtaining a functional application.

2. Framework architecture

Figure 1 gives an overview of the different layers of the generator framework put into context. Controllable devices are accessed using their specific access layer – often provided together with the device. The device logic layer functions as interface with the high level layers provided by the framework. The framework itself comes in several modules covering different functionality aspects. The address space module lies on the OPC UA end of the server, exposing data towards OPC UA clients, and is implemented using a commercial OPC 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. Further functionality such as logging, a certificate handling facility and server metadata are being developed currently.

3. Modelling the subsystem or device – server design

In the generic approach of the framework, an object oriented model of subsystem or device was chosen due to two reasons:

- Object orientation is well known and widely understood.
- OPC UA itself follows the object orientated paradigm.

The purpose of modelling is to establish a description of such a model using classes, variables and the relations between them. Classes are types of particular objects. Variables belong to classes and are factual vectors of data. The purpose of relations is to model aggregations and type hierarchies.

As an example, let's prepare a very simplified, informal model of car. A car has a couple of variables: `currentSpeed` and `acceleration`. A car is composed of an engine, which itself has variables: `currentRpm` and `currentPower`, a battery (variable: `voltage`) and wheels (variable: `tirePressure`). We could say that a particular car is identified by the vehicle identification number (VIN) which is considered constant and therefore could be specified by the configuration. In addition, there might be some operations defined, e.g. the car can be started and stopped. This description could obviously be expressed in some formal notation like e.g. UML.

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

4. Server design file

A format was required which parametrizes the generic server such that specific instances (design files) match particular subsystems/device types. The biggest ingredient of such a design file is the data model handled by the server (discussed in the previous section) augmented with hints which are normally hidden behind the OPC UA server abstraction, but are necessary to create a working server.

Probably the most outstanding hint is capturing the difference between distinct classifications of variables – two distinct classifications were identified:

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 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 OPC UA client, all variables are simply queried (using write/read OPC UA transaction types) or monitored (by creating monitored item with this variable). However at the OPC UA server implementation side it is beneficial to differentiate between classifications.

Another important hint (again, not visible from OPC UA client perspective) is handling of concurrence inside OPC UA server. The framework provides capabilites to model domains of mutual exclusion to prevent race conditions in case two objects were to be accessed at the same time.

Specifying how to configure objects is yet another aspect. 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 the object name. It was decided that all objects are identified by a unique name (which is not so obvious – OPC UA allows to identify also by a number, for example) so that a hierarchy of objects may be easily traversed using dot as a separator (e.g. `car1.wheelFrontLeft.tirePressure`). Apart from the name, objects often require additional configuration. Following the former example, an instance of `Car` may need to know its colour, vehicle identification number etc., which are constant and need to be specified on instance creation. This is also handled by the design of a server in the framework through items called "config entries". Config entries belong to classes.

We chose to store the server design in a XML file. The XML Schema of this file is distributed with the framework.

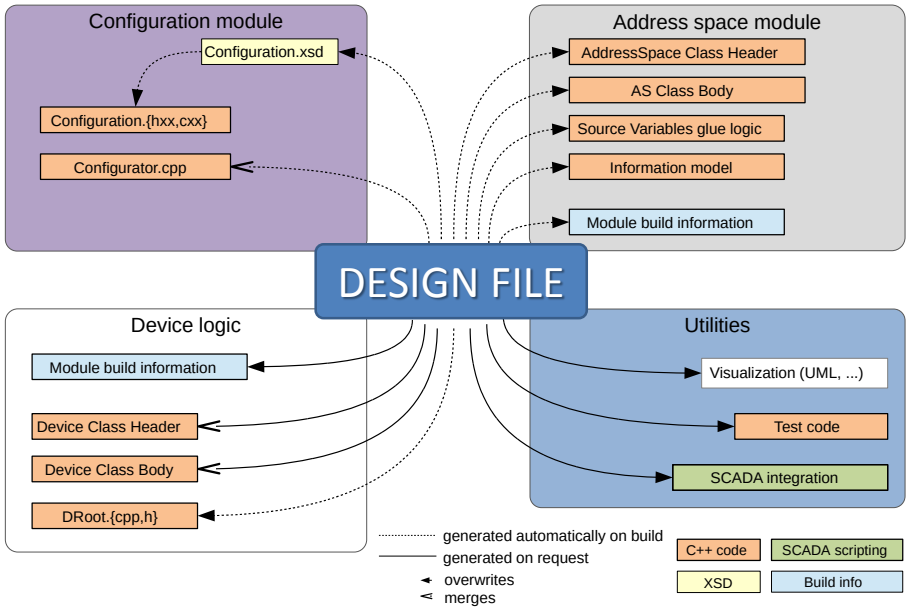

Figure 2. Transformation diagram illustrating the software generation or modification process for a given design file.

Please note that design XML file is not the configuration XML file which was briefly aforementioned. The difference may be best described as:

- the design is a description of types,
- the configuration is a description of instances.

5. Transforming the server design into runnable software

The framework 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,
- OPC UA address configuration for quick integration into a 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 OPC UA Server using the framework is as follows:

(i) create the design,

(ii) 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),

(iii) extend device logic stubs by providing factual implementation,

(iv) build the server (when using the provided build system – based on CMake),

(v) develop by re-iterating the steps (i) $\rightarrow$ (iv).

5.1. Address Space module transformation

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. For example, imagine that the class `Car` has an output variable `currentSpeed`, then there will be a method `Car::setCurrentSpeed(double speed)` which, when invoked from C++ code, pushes the new value into the address space (i.e. the new speed will be available to OPC-UA clients).

Information model: OPC UA has rich modelling capabilities from which smart OPC UA clients may profit. The framework exposes the information model derived from the design. For example, when class `Car` is part of the design, an OPC UA object type called `Car` becomes available in the address space within the folder for object types. Moreover all instances of class `Car` declared in the configuration file expose that their type is the same OPC UA object type `Car`.

5.2. Configuration module transformation

Generation of the configuration XSD schema: Each class from the design is transformed into a complexType in an XSD schema. Aggregation relations between classes are respected in generated XSD (e.g. complexType may have a sequence of elements of different class). Moreover, config entries (specified in the design), which provide configuration data to specific instances become attributes within given the relevant complexType. An example of a config entry for a class Car could be vin (vehicle identification number), nominalPower or colour.

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.

5.3. Device Logic transformation

The Device Logic is the only framework-provided module in which the developer is expected to write C++ source code – the device specific implementation – starting from the stubs generated by the framework.

For example, imagine a class `Car` has a variable named `acceleration` and assume that every time the variable is altered by an OPC UA Client, the OPC UA Server shall instruct the engine to alter fuel flow accordingly. Once the appropriate design is created, the developer requests the framework to create a device logic class for `Car`. Inside the generated class there is a stub method called `Car::writeAcceleration(...)`. Following the initial code generation this stub will be empty. The developer shall add implementation to the class such that it opens a communication channel with the engine and creates and sends the relevant command to increase/reduce the fuel flow.

As development progresses, the design may be fluid; classes may be added or removed; variables may change type or perhaps are suppressed completely. After every change of design, the developer re-runs the Device Logic generation. If class sources exist already (i.e. user stub implementations), the hand-written code will not be overwritten – a merge tool opens to facilitate the merging of the hand-written code for the new design.

5.4. SCADA integration code transformation
Additionally, the framework may generate tools which let the server be easily integrated into some SCADA systems. The typical use case at CERN is as follows: SCADA (here: Siemens WinCC OA) scripts are generated which can create Datapoint[4] types according to server's design, instantiate the actual Datapoints and set up appropriate OPC UA addressing. This step may not be necessary if a SCADA is used which provides information model aware OPC UA clients.

6. Additional tools for developers
Significant efforts have been invested to avoid duplication of work of users of the framework. Thus it provides a number of tools helping to carry out the following tasks:

- Visualizing object structures: an UML-like diagram creator is provided which helps to visualize the design. It was used for example to generate Fig. 3 and 5.
- 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.
- Creating installers: a preconfigured spec file for creating RPM[5] packages.
- Testing the address space: a tool is added which keeps on pushing random data into the address space. This can be used e.g. to test client mappings and server/client performance under specific conditions.

7. Examples
At the time of writing, the framework had already been used to create nine different OPC 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) or will enter production soon. We briefly discuss two of these servers – VME crates server and SNMP server – as their architecture and use case are very different, demonstrating the flexibility of the framework.

7.1. VME crates server
VME is a computer hardware bus standard, where VME-based boards are hosted by so called VME crates which provide a common back-plane, power supplies and a VME control module. In order to integrate the monitoring and control of VME crates into a parent control system, the crate needs to have a communication interface and communication protocol for which an industry standard is unfortunately lacking. Many experiments at CERN use VME crates from the WIENER Plein & Baus GmbH manufacturer. They can be interfaced using CAN bus and using a manufacturer-developed polling-based protocol[5]. The OPC UA server implements full monitoring and control based on this protocol and is in production use within the ATLAS DCS since the end of 2014.

The object decomposition used in the server models all elements of the VME crates monitoring and control chain including CAN interfaces for communication via COTS rack servers of the control system. The chosen object model is as follows: root instances are CAN buses which contain VME crates. The crates in turn may contain channels, fans and temperature probes. Note that each of these elements (buses, crates, etc.) is to be hierarchically declared in the

[4] Datapoints are structures to store data in SCADA systems.
[5] Common Linux software packet format compatible with numerous Linux distributions

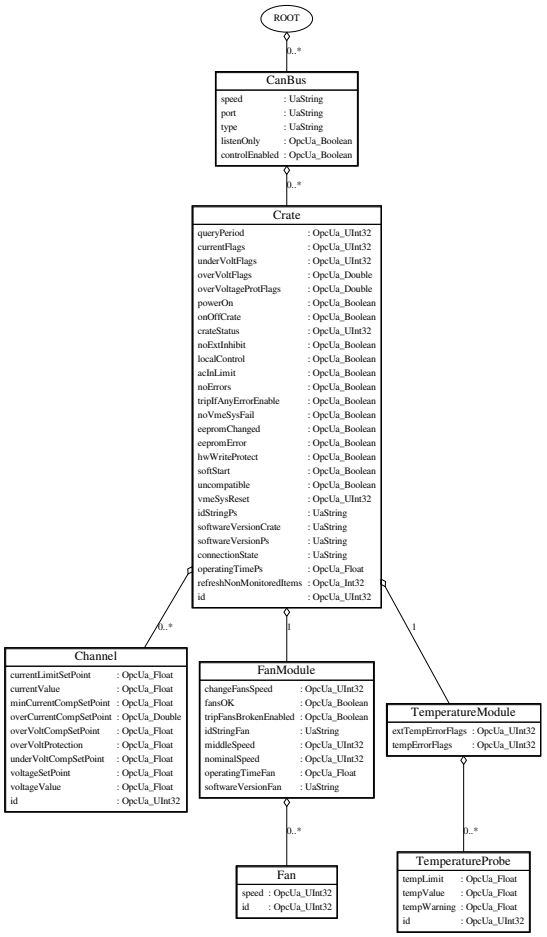

Figure 3. Generated design diagram of the VME crates server.

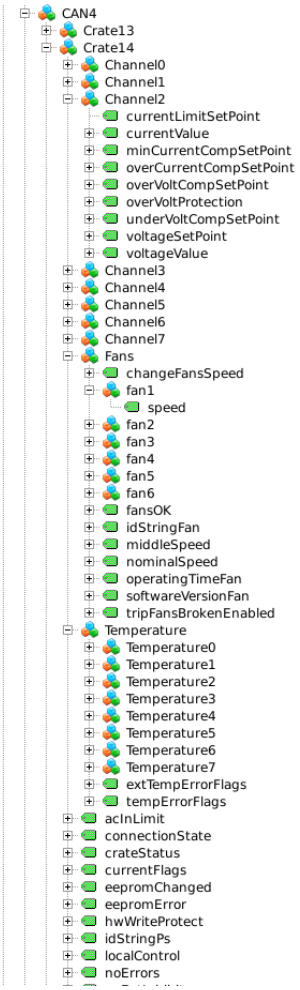

Figure 4. OPC UA client view of a running VME crates server.

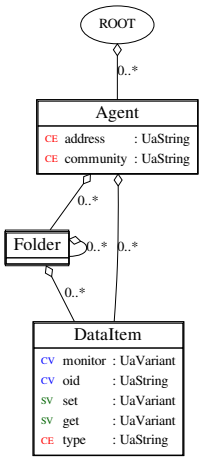

Figure 5. Design diagram of the SNMP server.

configuration file, so essentially any configuration may be obtained, from a very simple single-crate systems up to multi-bus, multi-crate systems which have thousands of channels. Figure 3 shows the class hierarchy generated directly from the design file.

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 device logic of the OPC UA 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 OPC UA address space and then further to any subscribed OPC UA client – in OPC UA terms "monitored data". Figure 4 shows a screenshot of an OPC UA client browsing through the address space of one of instances of the server used in production.

In addition to the framework-provided modules, the server uses a "CAN interface" module – a hardware access library supporting CAN interfaces from numerous vendors (e.g. SYS TEC electronic, Kvaser, PEAK-System, Analytica Anagate and others). This module is a common software component developed at CERN used for many projects, including other OPC UA

servers. A second module represents the specific VME crate communication protocol facilitating encapsulation of data.

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 TDAQ system which effectively has ∼5000 channels. Even for such a configuration its load on the CPU and memory is negligible on a modern server computer.

7.2. SNMP OPC UA server

The SNMP (Simple Network Management Protocol) is an Internet-standard protocol for managing devices on IP networks. At CERN it has numerous applications, e.g. to monitor and control the experiment computing infrastructure.

We developed a server which exposes a tree-shaped address space in which data items are basic building blocks (leaves of the tree). Figure 5 shows the design of the server. Each data item is bound to one SNMP variable by specifying its OID (object identifier) in the configuration. In such configuration each read/write request coming from an OPC UA client is transformed into a get/set SNMP operation.

Compared to 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. Such functionality is provided by the framework through the means of spawning each SNMP transaction as a job belonging to a thread pool. Please note that no user code is needed to profit from such concurrent execution of SNMP transactions – the only thing the developer has to provide is a function which implements such a transaction and setting the appropriate variable options in the server design file.

8. Conclusions

Having applied the framework to the creation of a number of OPC UA servers demonstrates its advantages: versatility and efficiency of development. The former has been demonstrated by a vast span of applications, from custom devices to generic designs for well-known protocols like SNMP. 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 better source code managability. The framework 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 expansion of the framework at CERN and beyond is envisaged while more features are currently being developed.

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 OPC UA Server SDK"
[5] W-IE-NE-R Plein & Baus GmbH, "CAN-BUS Interface for W-Ie-Ne-R Crate Remote Control"