ATL-DAQ-PROC-2016-042
21 December 2016

The new inter process communication middle-ware for the ATLAS Trigger and Data Acquisition system

Serguei Kolos[1] and Reiner Hauser[2] on behalf of the ATLAS TDAQ Collaboration

1. University of California Irvine, 4172 Frederick Reines Hall, Irvine, CA, USA.
2. Michigan State University, Est. 1855. East Lansing, Michigan, USA.

serguei.kolos@cern.ch

Abstract. The ATLAS Trigger & Data Acquisition (TDAQ) project was started almost twenty years ago with the aim of providing scalable distributed data collection system for the experiment. While the software dealing with physics data flow was implemented by directly using the low-level communication protocols, like TCP and UDP, the control and monitoring infrastructure services for the TDAQ system were implemented on top of the CORBA communication middle-ware. CORBA provides a high-level object oriented abstraction for the inter process communication, hiding communication complexity from the developers. This approach speeds up and simplifies development of communication services but incurs some extra cost in terms of performance and resources overhead. Our experience of using CORBA for control and monitoring data exchange in the distributed TDAQ system was very successful, mostly due to the outstanding quality of the CORBA brokers, which have been used in the project: omniORB for C++ and JacORB for Java. However, due to a number of shortcomings and technical issues the CORBA standard has being gradually losing its initial popularity in the last decade and the long term support for the open source implementations of CORBA becomes questionable. Taking into account the time scale of the ATLAS experiment, which goes beyond the next two decades, the TDAQ infrastructure team reviewed the requirements for the inter process communication middle-ware and performed the survey of the communication software market in order to access the modern technologies which raised in the past years. Based on the result of that survey several technologies were evaluated for estimating the long-term benefits and drawbacks of using them as a possible replacement for CORBA during the next long LHC shutdown, which is scheduled in 2 years from now. The evaluation concluded recently with the recommendation of using communication library called ZeroMQ in place of CORBA. The article presents the methodology and the results of the evaluation as well as the plans of organizing the migration from CORBA to ZeroMQ.

1. Introduction

The TDAQ [1] online system of the ATLAS [2] experiment is composed of tens of thousands of software processes distributed over several thousand computers. For the system to function properly all of these processes must be operated in a coherent way, thus making Inter-Process Communication (IPC) a crucial task. The current implementation of the TDAQ control system, which was born in 1998, is based on the CORBA [3] communication middleware. Two CORBA implementations have been used: JacORB [4] for Java and omniORB [5] for C++. They both satisfied the performance and scalability requirements and simplified development and maintenance of the TDAQ software. However, after more than 10 years of successful experience with the CORBA software, we have decided that the time is right to explore if there are new products on the IPC software market which can improve our system performance and maintainability.

2. CORBA in the light of modern software practices

CORBA is an open standard for distributed object computing, which was proposed in 1991 by the Object Management Group (OMG). This was the first attempt to provide a broad high-level standard for information exchange in a distributed software environment. The standard was quite successful and played an important role in the overall evolution of distributed software systems. However many key features of the CORBA standard have a number of built-in drawbacks, which have become more and more prominent in recent years, making CORBA less attractive for modern software development.

2.1. The Interface Definition Language

CORBA proposed a dedicated language called Interface Definition Language (IDL) for communication protocol description. The code for a specific programming language can be automatically generated from such a description. This approach provided a powerful yet simple solution for establishing communication between different programming languages and operating systems. While IDL was originally one of the strongest points of the CORBA standard, the passage of time has seen it become one of the weakest. The issue was that the standard is very strict with defining the mapping from the IDL declarations to the programming languages, thus practically excluding any opportunity to benefit from language evolution. While the most popular programming languages, like C++ and Java, evolved significantly in the last decade the CORBA API couldn’t benefit from that so now the CORBA API looks archaic and awkward. The C++ IDL mapping efficiency also suffers from the absence of some recently introduced features like zero-copy or move-assignment semantics.

2.2. CORBA brokers interoperability

Another strong point of the CORBA standard was interoperability between different CORBA implementations, which is required by the standard. This is a good feature, which unfortunately led to some issues with communication efficiency as all CORBA implementations were forced to use the same communication protocol, called IIOP, to support interoperability. The problem is that IIOP has some drawbacks, like for example the data size and processing time overhead due to the data alignment requirements.

2.3. The source code compatibility

The standard precisely defines the API for any possible operation including object creation, registration, activation and so on for assuring source code compatibility between different CORBA implementations. This requirement forced any CORBA broker to provide a high-level object-oriented API, which completely hides all aspects of the underlying communication. This of course simplifies software development and maintenance but at the same time adds a noticeable performance overhead and reduces the flexibility of the communication implementation. In practice that would also make any end user CORBA applications strongly dependent on the quality of the chosen CORBA implementation, thus making problematic a transparent migration from one CORBA broker to another.

3. The modular communication architecture

The rapid evolution of programming languages and the increasing popularity of the open source software development model drastically changed the landscape of the IPC software domain in the last decade. The modern open source market offers a large variety of high quality software packages, which can be used for implementing the basic components of a high-level communication system. A combination of such packages is an attractive alternative to a monolithic heavyweight communication system like CORBA. Such a solution implies that the communication system is organized into a hierarchy of software layers with well-defined interfaces between them, which allow changes to the implementation for these layers in a transparent way for end user applications. Another important property of this design is the transparency of the layers, which means that the APIs of all layers are exposed to the end users and can be used independently of each other. This gives the full advantage of using the high-level API for a simple communication implementation while is still leaving open the possibility of implementing performance-critical applications using the low-level communication API to increase efficiency.

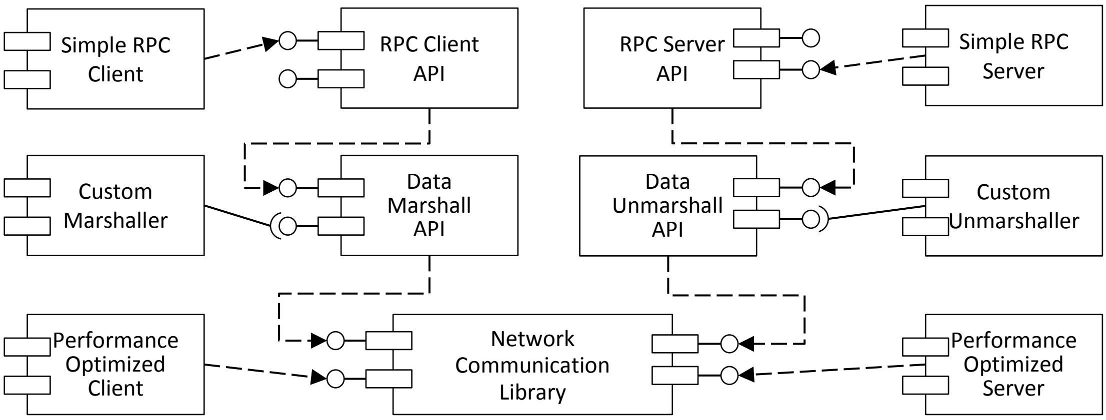

Figure 1. Modular communication architecture for the new ATLAS TDAQ IPC software.

Figure 1 shows the architecture of the new IPC software for the ATLAS TDAQ system, which has been designed using this approach. This software consists of three main layers:

1. A low-level communication library, which provides a simple and efficient API for exchanging unstructured data, i.e. messages, between software applications independently of their location. This component provides abstraction for the network communication layer.
2. The data marshalling and unmarshalling components provide a way of passing structured information between applications. These components define the API for the conversion of an arbitrary data structure into a message, which can be passed over a network and vice versa. This API can have multiple implementations, which may be interchanged transparently for the end users. The only limitation is that any implementation of this API has to support all programming languages used in the ATLAS TDAQ system. A user application can use these components directly to convert arbitrary information to the network specific format before passing it to another application using the first layer API.
3. The Remote Procedure Call (RPC) layer provides the top level API for inter-process communication, making remote calls look like the normal local ones in a given programming language. This is the top-most API layer, which is simple to use but incurs a relatively high overhead with respect to the lower layers due to the generic code, which is capable of transparently mapping an arbitrary user function to an RPC procedure.

4. The new IPC software implementation

To simplify implementation and maintenance of the new IPC software we decided to use the existing open software projects as much as possible. As a result the fully functional IPC implementation has only a few hundred lines of custom code, which was provided mostly for the RPC layer implementation, while the implementations of the first two layers are almost entirely based on external software.

4.1. Using ZeroMQ as the network transport layer

After comparing a number of communication libraries available on the open software market we have chosen ZeroMQ [6] as the implementation of the network communication layer. ZeroMQ is a low-level C-style library for I/O based on an object resembling a standard socket, but which hides all the state management and error handling complexity one would normally expect. ZeroMQ supports most of the widely used programming languages including C, C++, C#, Java, Python, Ruby, and many others. ZeroMQ is open source software with large and very active user and developer communities.
ZeroMQ provides a simple yet mature API for remote communication, offering the full power of a classic socket, becoming a de facto standard for many modern communication software projects. In order to support some common services used in the TDAQ system, some extra classes have been

added on top of the ZeroMQ API, but in agreement with our design approach they don't imply any limitations to the direct use of the ZeroMQ native classes.

4.2. Data serialization

Serialization is an important ingredient of a software communication system, which has a major impact on its performance. For the sake of flexibility we have defined a simple interface for *marshalling,* i.e. converting a programming language structure to the sequence of bytes and *unmarshalling*, which does the opposite operation.

```
public interface ISerializer {
  public class Request {
    private static final AtomicLong gid = new AtomicLong(0);

    public final Method method;
    public final List<Object> params;
    public final long id;
  }

  String serializeRequest(Request req);

  Request deserializeRequest(String str, Map<String, Method> methods);

  String serializeResponse(Object resp, Exception e, long id);

  <T> T deserializeResponse(String str, Class<T> clazz) throws Exception;
}
```

Figure 2. Example IPC API serialisation interface (Java).

Figure 2 shows how such an interface is declared for the IPC API in Java. The interface has four methods: two for marshalling and unmarshalling requests on the client side and another two doing the same operations for a server response. The default implementation of this interface uses the Google Gson [7] library to pass the data over a network in Json format. The default implementation works fine for simple requests, which don't carry too much structured data. On the other hand for performance-optimized applications one can provide another implementation of this interface, which can be easily plugged into the IPC library and used transparently without affecting the code of the communication applications.

4.3. The RPC API

The TDAQ online software system consists of a number of communication services providing specific APIs for exchanging different types of information. To simplify development and maintenance of such services we tried to implement the RPC communication layer for the IPC software in such a way that a remote request would look as much as possible like a local one. It wasn't possible to achieve this goal completely in the C++ API, but in Java this was successfully implemented using the Java Reflection API [8].
Following the standard Java approach a user shall first declare a new interface with all methods, which can be called remotely. Figure 3 shows a simple example of such an interface.

```
public interface IHello {
  void say(String greetings);
}
```

Figure 3. A simple custom communication interface (Java).

Figure 4 demonstrates how a simple IPC Java server can be implemented. The *HelloImpl* class implements the *IHello* interface in the same way as a normal Java interface. Then the instance of the

HelloImpl class is given as a parameter to the IPC Server class constructor, which is bound to the specific network endpoint.

```
public class HelloImpl implements IHello
{
  @Override
  public void say(String greetings) {
    System.out.println(greetings);
  }

  public static void main(String[] a){
    Server server = Server.build(
         “tcp://localhost:5555”,
         new HelloImpl());

    server.wait();
  }
}
```

Figure 4. Simple RPC server (Java).

```
public static void main(String[] a)
{
  IHello hello = Client.build(
         IHello.class,
         "tcp://localhost:5555");

  hello.say(“Hello, World”);
  …
}
```

Figure 5. Simple RPC Client (Java).

The client implementation is shown in Figure 5. It uses the static *Client.build* function giving it two parameters: the server’s endpoint and the interface class. The call returns an instance of the special implementation of the *IHello* interface, which can be used to translate a local call to an interface’s method to the invocation of the corresponding function on the remote server.

5. ZeroMQ performance and scalability tests

Before choosing ZeroMQ as the communication layer implementation for the TDAQ IPC software we performed several tests to verify its performance and understand how it scales with the number of communicating applications. For comparison we also repeated the same tests with the omniORB CORBA broker and the ICE [9] framework from ZeroC company. ICE is a modern CORBA-like object-oriented RPC framework, which is free of many CORBA drawbacks, but still provides a complex high-level monolithic IPC solution.

In order to test how well the communication software scales with the number of communicating clients, the following configurations was tested for each of them:

- A single server application running on a dedicated computer and answering to all the clients’ requests. For all tests the servers were running the same number of threads: 1 I/O thread for incoming request routing and 20 worker threads for executing request code.
- Client applications were equally distributed on a cluster of computers connected to the server via the local network. Each client sent a request containing a 1-byte string to the server and received the same string back as fast as possible.

5.1. Hardware configuration for the tests

All tests were performed at a facility comprising around 200 computers connected to a local network via 10Gb Ethernet. Server applications ran on a commodity server with Intel Xeon E5645 4 core 2.4 GHz CPU [10] and 24 GB RAM. Client applications were equally distributed over 100 computers with Intel Xeon E5420 4 core 2.5 GHz CPU [11] and 16 GB RAM.

5.2. Test results

Figure 6 shows the average request execution time for all three implementations with different numbers of concurrent clients. The results clearly indicate that all the systems have excellent scalability and offer very good performance. ZeroMQ uses a bit more time for a single request, which can be explained by the differences in handling requests on the client side between ZeroMQ and the other two systems.

Contrary to omniORB and ICE brokers ZeroMQ does not write data to the network socket from the user thread. Instead the *write* function just places the data into a queue which is then handled by another thread dedicated to IO operations. This thread reads this data from the queue and sends it to the network.

Figure 7 shows the average CPU time which the servers spent in handling a single remote request. For these tests the implementation of the remote method did nothing apart from returning its input parameter. The results therefore show the pure overhead of the communication software itself. While ICE and omniORB are very similar in that respect, ZeroMQ has much smaller overhead due its simplicity.

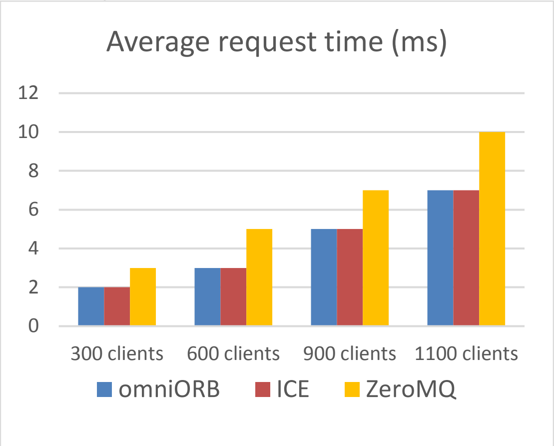

Figure 6. Average wall clock time for a single request execution (measured on the client).

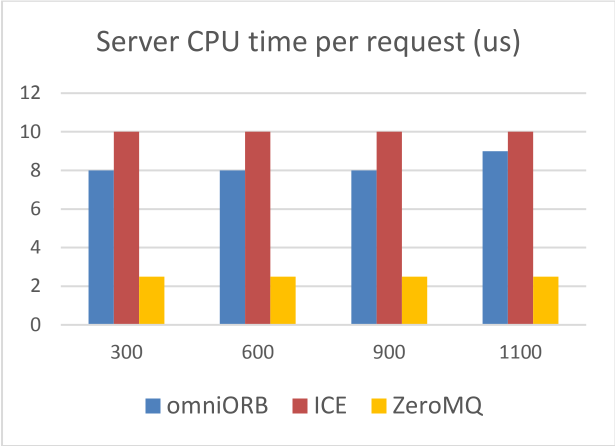

Figure 7. Average CPU time for a single request execution (measured on the server).

6. Conclusion

The CORBA standard has a long and successful history, but now the interest of the IPC software development community has been shifting away from universal frameworks to relatively low-level messaging systems. Many such systems are so simple that they are distributed in the form of a library with a simple yet very powerful API. The new implementation of the ATLAS TDAQ IPC software uses such a library called ZeroMQ in conjunction with a modular software architecture approach. This approach gives an attractive alternative to the usage of a traditional high-level object-oriented communication framework by offering a high performance, simple and flexible solution for communication system development. The new IPC software offers a RPC-style API for implementing simple communications, but at the same time makes it possible to customize data serialization along with providing access to the low-level communication library API for performance optimization.

7. References

[1] ATLAS Collaboration 2003 ATLAS high-level trigger data-acquisition and controls *Technical Design Report* ATLAS-TDR-016 CERN-LHCC-2003-022.

[2] ATLAS Collaboration 2008 The ATLAS experiment at the CERN Large Hadron Collider *Journal of Instrumentation* JINST 3 S08003.

[3] Common Object Request Broker Architecture, www.corba.org

[4] JacORB, www.jacorb.org

[5] omniORB, omniorb.sourceforge.net

[6] Distributed Messaging, www.zeromq.org

[7] A Java serialization/deserialization library, https://github.com/google/gson

[8] Java Reflection API tutorial, https://docs.oracle.com/javase/tutorial/reflect/

[9] Ice - Comprehensive RPC Framework, https://zeroc.com/products/ice

[10] Intel Xeon Processor E564, http://ark.intel.com/de/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI

[11] Intel Xeon Processor E5420, http://ark.intel.com/products/33927/Intel-Xeon-Processor-E5420-12M-Cache-2_50-GHz-1333-MHz-FSB