REAL-TIME DATA REDUCTION INTEGRATED INTO INSTRUMENT CONTROL SOFTWARE

P. Mutti*, F. Cecillon, C. Cocho, A. Elaazzouzi, Y. Le Goc, J. Locatelli, H. Ortiz
Institut Laue-Langevin, Grenoble, France

Abstract

The increasing complexity of the experimental activity and the growing raw dataset collected during the measurements pushed the integration of the data reduction softwares within the instrument control. On-line raw data reduction allows users to take instant decisions based on the physical quantities they are looking for. In such a way, beam time is optimise avoiding oversampling. Moreover, the datasets are more consistent and the reduction procedure, becoming now part of the sequencer workflow, is well documented and can be saved for future use. We will report on the implementation of the on-line data reduction on several instrument at the ILL as well as on the obtained performances.

INTRODUCTION

NOMAD is the instrument control software in use at the Institut Laue-Langevin. It has been designed about 10 years ago as a client/server application. The server is written in C++ to have a direct access to the C driver layer while the main client is written in Java to have a portable and reactive GUI application. A number of other client applications have been developed. Among them we can cite the Nomad Web Spy to refresh a web page that displays monitoring information, the Plot Screenshot Generator to generate offline acquisition images, etc. [1]. In the current NOMAD environment, processes and applications are running in different languages while the communication between them is based on CORBA [2]. Processes are started and stopped on demand and they can crash: - continuous development of a small team compared to the number of instruments - module-based architecture with hundred of classes The crashes are part of the problem and taken into account and may not be considered as development mistakes. It is impossible to perform the tests to ensure a 100% robustness.

The integration of reduction or more generally computation methods at the ILL supposes:

- run heterogeneous code (multiple languages e.g. Matlab, Python, etc.) owned by scientists
- run on other computers than the instrument control PC to avoid interferences
- running on different systems (Linux, Mac OS X, Windows)
- acquisition data and computation results must be exchanged in an effective way

* mutti@ill.eu

One solution is to have a monolithic centralised server that processes every computation request and sends the results asynchronously. This solution requires some extra resources to maintain the services (list of computation methods and their update, load-balancing, etc.). Another solution is to have a "microservices" approach [3]. The computation methods are distributed along the existing control and scientific computers. For that we need a fluid, flexible and easy integration. We need Erlang [4] distributed proces functionalities. In our case, those functionalities need to be integrated in C++ and Java, including multi-process, multi-environment, synchronisation and message queue as well as crash management. To be able to achieve this goal, for the development of NAPPLI we have decided to leave CORBA since it is declining technology [5].

WHAT IS NAPPLI

NAPPLI is a very lightweight application server, very easy to install and usable everywhere. NAPPLI stands for *N applications*, where N can be both interpreted as the first character of NOMAD as well as an unknown number. A NAPPLI server provides services for starting, stopping, synchronising and making distributed applications communicate. We can say that it is an application-oriented middleware. The lifecycle of remote applications can be entirely managed within the application. The server is accompanied with a client API in Java and C++ with a modern asynchronous programming model using the *future* concept [6]. The available communication patterns between the applications are request/response, publisher/subscriber (synchronised or not) and return value at the end of the execution of the application. It is possible to use the application server in a non-intrusive way. Existing applications can be called directly without using the provided API. In this case, the application itself is directly responsible of communication with the outside world. The NAPPLI services are intended to be logic and network fault tolerant. An application can terminate with an exception but the remote caller will be notified with an error, so that it will be able to take the decision to restart or not the application. The network layer also implements features to survive to failures.

Simple Example

Figure 1 describe in a simple example the way NAPPLI can be used.

The application *App1* started by NAPPLI on the computer A, requests the start of *App2* on the computer B (Fig. 1 (A)). Notice that there both the computers A and B are running NAPPLI servers. In Fig. 1 (B) once *App2* is running and

ISBN 978-3-95450-148-9

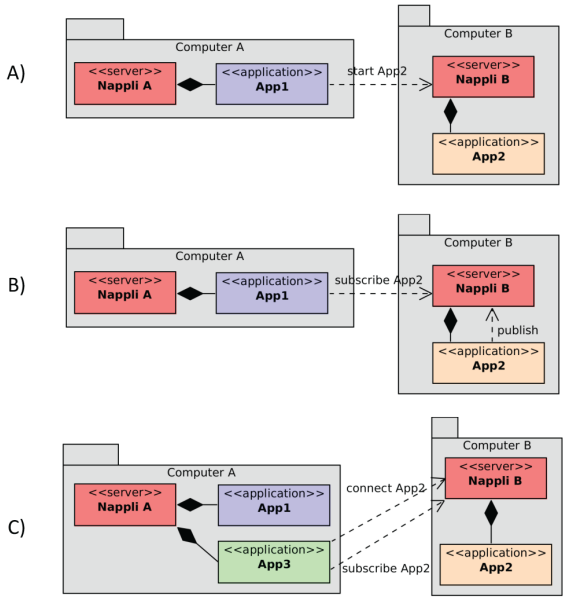

Figure 1: Example of the use of NAPPLI.

declared a publisher, *App1* subscribes to *App2* and it is ready to receive messages published by *App2*. In Fig. 1 (C) the new application *App3*, started on computer A, connects and subscribes to the existing *App2* and it is also ready to receive its published messages. Note that *App2* declares its publisher with a name (so that we can have multiple publishers in the same application) and a number of subscribers. When the number is a strictly positive n integer, the publisher can wait for the n subscribers to be registered before sending any message.

IMPLEMENTATION

NAPPLI was designed taking into account the disadvantages of CORBA. Unlike CORBA which shares data references through a naming service, NAPPLI shares application instances. This is a real different approach. Thus the applications have the responsibility to organise the sharing of their data through persistent services where NAPPLI provides patterns for communication. Internally, a NAPPLI server is written in pure Java 8 so that it only requires a compatible virtual machine for running. That makes it very portable and easy to install. Moreover, NAPPLI application instances are processes started by the server. We take advantage of the continuous Java improvement in its process API to have a unified way to start and monitor processes on different platforms (Linux, Mac OS X, Windows). To organise the network services, we use the robust and reliable ZeroMQ [7] message queue for which a 100% Java implementation called JeroMQ [8] exists. ZeroMQ is not only an open-source library, it also provides a precise documentation on how to use it in different network contexts. For example we followed the recommendations to implement a real synchronised publisher/subscriber pattern. Internally, we use the Protocol Buffers [9] library to serialise and parse messages exchanged by the application instances. Protocol Buffers offers a portable and fast data encoding and decoding. The main feature of a NAPPLI server is to start and stop applications on its own system. For that, a NAPPLI server is configured with a list of runnable applications. Each runnable application has a list of attributes so that an application instance can be seen as an enriched system process. We won't provide all the available attributes here but we can cite:

- *Name*: String identifier used by clients to start an application instance
- *Multiple*: yes or no, no meaning that only a single instance of the application can run
- *Restart*: An application instance is restarted in case of error termination
- *Stream*: The application publishes its output and error streams to the clients
- *Executable*: The path to the executable
- *Args*: The list of arguments that are always passed to the executable

Note that it is really important to make the difference between an application configuration and its instances. Applications have a workflow state shown in Fig. 2.

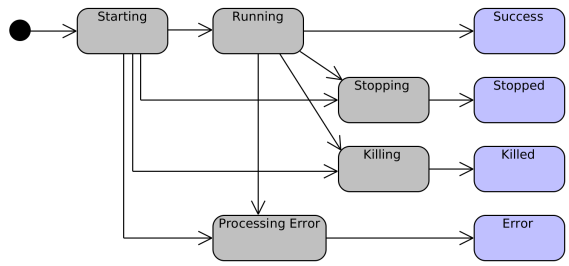

Figure 2: NAPPLI application state workflow.

Around the Running state, there are transitional states (Starting, Stopping, Killing, Processing Error) and terminal states (Success, Stopped, Killed, Error). Once the process of the application is launched, the application has the Starting state. It becomes Running when: it remains alive for a certain time, defined as an attribute of the application, or the application itself changes its state to Running. When a client requests the stop of the application, it immediately becomes Stopping until it terminates. At that moment, its state becomes Stopped. Notice that after the crash of an application (segmentation fault in C++ or exception in Java), its state becomes Error. The changes of state are all sent to the clients of the application.

The messages that are passed between applications can be of any type. NAPPLI provides binary messages as well as string messages so that the programmer can choose the encoding that can be Protocol Buffers or JSON. Arrays of

ISBN 978-3-95450-148-9

integer and floating point number are also provided by convenience to speed up the coding. Theses messages can be used in the different communication contexts:

- publisher/subscriber
- request/response
- return value

The return value of an application is implemented with a publisher/subscriber so that any connected client application receives the result. We provide a client API for C++ Java and a Python API is planned. A minimal example in C++:

```
// Get a reference to a remote NAPPLI server
Server server('tcp://computer.ill.fr:7000');

// Check its availability
If (!server.isAvailable()) {
  return;
}

// Start App and get a reference to the instance
auto_ptr<application::Instance> app;
app = server.start('App');

// Wait termination of app and get final state
application::State;
state = app->waitFor();

// Get the result from app with a text encoding
string result;
app  -> getResult(result);
cout << 'app returned ' << result << endl;
```

NAPPLI is clearly oriented towards a microservices software architecture rather than a monolithic one, where the various components (applications) are smaller and easy to update and replace. It becomes now obvious how NAPPLI fits perfectly the requirements for implementing data reduction within the instrument control workflow. Every scientific method, resident on remote computers, can, in principle, become a NAPPLI application providing it has the capabilities to accept command-line arguments or file inputs and returns results or file outputs.

COMPUTATION EXAMPLES

ZeroMQ provides different communication patterns that we have implemented in NAPPLI. It enables having different ways of writing the interactions between the instrument control software and the computation applications. We can define three interaction patterns depending on the computation purposes:

- Function application: the remote application is used as a function. The input data are passed to the application arguments and the return results are set by the NAPPLI result functionality. The application terminates after having set the result.
- Asynchronous server: the remote computation application is a server. The input data are passed by a subscriber connected to a publisher located in the control server. Another publisher is located in the application to publish the results asynchronously to the control server.
- Synchronous server: the remote application is a server. The input data are passed by a request of the control server and the results are returned by the response of the application. Even if the response can be asynchronous, we consider the entire procedure as synchronous as we need one response for one request that is not the case for the asynchronous server.

Matlab Q Space Transformation

A number of Matlab scripts were written at the ILL to transform raw detector data into Q space representations. We run the scripts using the Matlab engine library that we can access in C++ by a simple NAPPLI server application called *RemoteMatlab*. The sequence diagram in Fig. 3 illustrates the synchronous server as, in this case, we need to ensure to have one image for each acquisition.

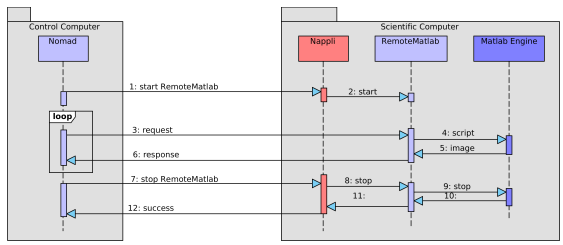

Figure 3: Synchronous server for Matlab Q space transformation.

Notice that there is a single *RemoteMatlab* execution for multiple acquisitions since the Matlab engine is very slow to initialise.

Nuclear Particle Physics Coincidences

When running long data acquisitions using a multi-detector system composed up to several hundred channels, the users need to have a multitude of monitoring information to survey the quality of the data taking. If some of those quantities can be obtained in real-time from the acquisition electronics, some others require the knowledge of the entire detection system and of the physical relation between the different elements. On the other hand, those indirect quantities that are not directly accessible in real-time, must be calculated and visualised within a reasonable time. Moreover, these computations must not disturb the live data taking which can consume lots of resources on the control server. An asynchronous NAPPLI server, as shown in Fig. 4, fits the requirements.

We obtain a two-way streaming of data. Partial acquisition data are sent to the *NPPCoincidence* application which computes and sends the results asynchronously. Notice that we can have a single execution of the computation application per acquisition.

ISBN 978-3-95450-148-9

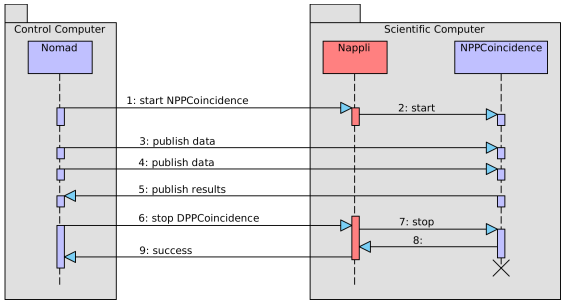

Figure 4: Asynchronous server for computing coincidence rates.

CONCLUSION

We developed successfully NAPPLI to manage and organise the execution of the different applications of the instrument control software environment at the ILL. More specifically it is the ideal solution to distribute and run the scientific computations on other computers without the need for a monolith server infrastructure. The NOMAD software organisation tends to a microservices architecture where different actors have more freedom to make evolve the services they are responsible for. At the ILL, we are at the first step of the integration of the scientific computations in the acquisition workflow. At present computations are used for monitoring the acquisitions but soon they will be used to take automatic decisions. NAPPLI is a generic tool that can be reused in larger environments than the ILL and is not reserved to data acquisition porposes only. Some more developments can be led to ensure its scalability and robustness. But further, the concept of an application-based middleware is proved.

REFERENCES

[1] P. Mutti et al., “Nomad more than a simple sequencer", Proc. ICALEPCS (2011), Grenoble, France.

[2] OMG CORBA, http://www.corba.org

[3] M. Fowler, “Microservices", http://martinfowler.com/articles/microservices.html

[4] J. Armstrong, “Erlang", (2010), http://cacm.acm.org/magazines/2010/9/98014-erlang/

[5] M. Henning, “The Rise and Fall of CORBA", (2006), http://queue.acm.org/detail.cfm?id=1142044

[6] Henry Baker et al.(1977), “The Incremental Garbage Collection of Processes”, Proc. of the Symposium on Artificial Intelligence Programming Languages, ACM Sigplan Notices 12, 8. pp. 55–59.

[7] iMatix ZeroMQ, http://www.zeromq.org

[8] JeroMQ, https://github.com/zeromq/jeromq

[9] Google Protocol Buffers, http://code.google.com/p/protobuf

ISBN 978-3-95450-148-9