Journal of Computer Engineering
ISSN 10003428

Original Research Article
Vol 10, Issue 7, 2021 | ISSN: 1000-3428

Design and Implementation of a Novel Communication Library for the COMPASS Experiment

Elena Rodriguez, Sofia Jensen

Department of Physics, University of Oslo, Norway; Institute for Particle Physics, ETH Zurich, Switzerland

Abstract—Modern experiments in high energy physics impose great demands on the reliability, the efficiency, and the data rate of Data Acquisition Systems (DAQ). This contribution focuses on the development and deployment of the new communication library DIALOG for the intelligent, FPGA-based Data Acquisition System (iFDAQ) of the COMPASS experiment at CERN. The iFDAQ utilizing a hardware event builder is designed to be able to readout data at the maximum rate of the experiment. The DIALOG library is a communication system both for distributed and mixed environments, it provides a network transparent inter-process communication layer. Using the high-performance and modern C++ framework Qt and its Qt Network API, the DIALOG library presents an alternative to the previously used DIM library. The DIALOG library was fully incorporated to all processes in the iFDAQ during the run 2016. From the software point of view, it might be considered as a significant improvement of iFDAQ in comparison with the previous run. To extend the possibilities of debugging, the online monitoring of communication among processes via DIALOG GUI is a desirable feature. In the paper, we present the DIALOG library from several insights and discuss it in a detailed way. Moreover, the efficiency measurement and comparison with the DIM library with respect to the iFDAQ requirements is provided.

I. Introduction

THIS paper presents development, deployment and performance of the communication library DIALOG for the intelligent, FPGA-based Data Acquisition System (iFDAQ) of the COMPASS experiment at CERN.

COMPASS (COmmon Muon Proton Apparatus for Structure and Spectroscopy) [1] is a high-energy particle physics experiment with fixed-target situated on the M2 beamline of the Super Proton Synchrotron (SPS) particle accelerator at CERN laboratory in Geneva, Switzerland. The scientific program of the COMPASS experiment was approved in 1997. The goal was to study the structure of gluons and quarks and the spectroscopy of hadrons using high intensity muon and hadron beams. By the year 2010 the experiment entered it's second phase COMPASS-II [2]. The COMPASS-II program started with a physics run for the study of the polarized Drell-Yan (DY) process in the years 2014 and 2015 followed by a run dedicated to Deeply Virtual Compton Scattering (DVCS).

The DIALOG library is designed and implemented to meet all necessary requirements, especially on high-performance, reliability and robustness. It was fully incorporated to all processes in the iFDAQ during the run 2016 and improved the stability of iFDAQ significantly.

The paper is organized as follows. The description of iFDAQ is stated in Section II. A very detailed overview of the iFDAQ from the hardware and software point of view is given, followed by a figure of the COMPASS iFDAQ topology which put all views together. Finally, the motivation for development of a new communication library is given, which concludes the iFDAQ part of the paper.

Section III deals with the design of the DIM library. It gives a description and deeper insight into the DIM library.

Section IV is concerned with the implementation of the DIALOG library. It presents all requirements, gives description, integration, robustness and implementation domains. The important subsection is Scenarios discussing all exemplary situations.

In Section V, the online monitoring of communication among processes via DIALOG GUI is stated. The DIALOG GUI allows the visualization of the processes involved in the application.

The final section, Section VI, presents the efficiency measurement and performance of the DIM and DIALOG library.

II. iFDAQ Architecture

The COMPASS iFDAQ is currently undergoing a major hardware and complete software replacement, the first part of which was finished in 2014, and the second part of which is planned to be completed in 2017.

How to cite: Elena Rodriguez, Sofia Jensen (2021). Design and Implementation of a Novel Communication Library for the COMPASS Experiment. Journal of Computer Engineering, 10(7), 44–56. DOI: https://doi.org/10.5281/zenodo.19343290

A. Hardware Part

The iFDAQ of the COMPASS experiment consists of several layers [9], [7], [8]. The frontend electronics that form the lowest layer continuously preprocess and digitize analogue data from the detectors. There are approximately 300 000 detector channels; trigger rate can rise up to 50 kHz with 36 kB average event size. SPS accelerator operates in cycles that consist of 10 second long period with beam (called spill) followed by approximately 40 second period without beam. Data from multiple channels are readout and assembled by the concentrator modules called CATCH [13], HGeSiCA [15], and GANDALF [14]. These modules receive signals from the time and trigger system; when the trigger signal arrives, the readout is performed and data are sent over optical connection S-Link [17] to the following layer that is based on special FPGA DHC (Data Handling Card) cards. It is further divided into two layers and is responsible for building of complete events. It comprises eight FPGA (MUXs) and handles another level of multiplexing. S-Links are also used to connect the first sublayer to the second sublayer, which is made up of a single DHC with switch firmware (SWITCH) – this layer handles event building.

This newly designed event building part allows usage of more compact control system. The hardware event builder performs online verification of data consistency. The last layer of the system consists of eight readout engine computers equipped with spillbuffer cards that readout assembled events and transfer them to the CERN permanent storage (CASTOR)
[12].

That is a theoretical description of the iFDAQ full setup. In Fig. 1, the current state – used in the run 2016 and 2017 – is given. It consists of only six FPGA cards (MUXs) on the level of multiplexing and four readout engine computers.

B. Software Part

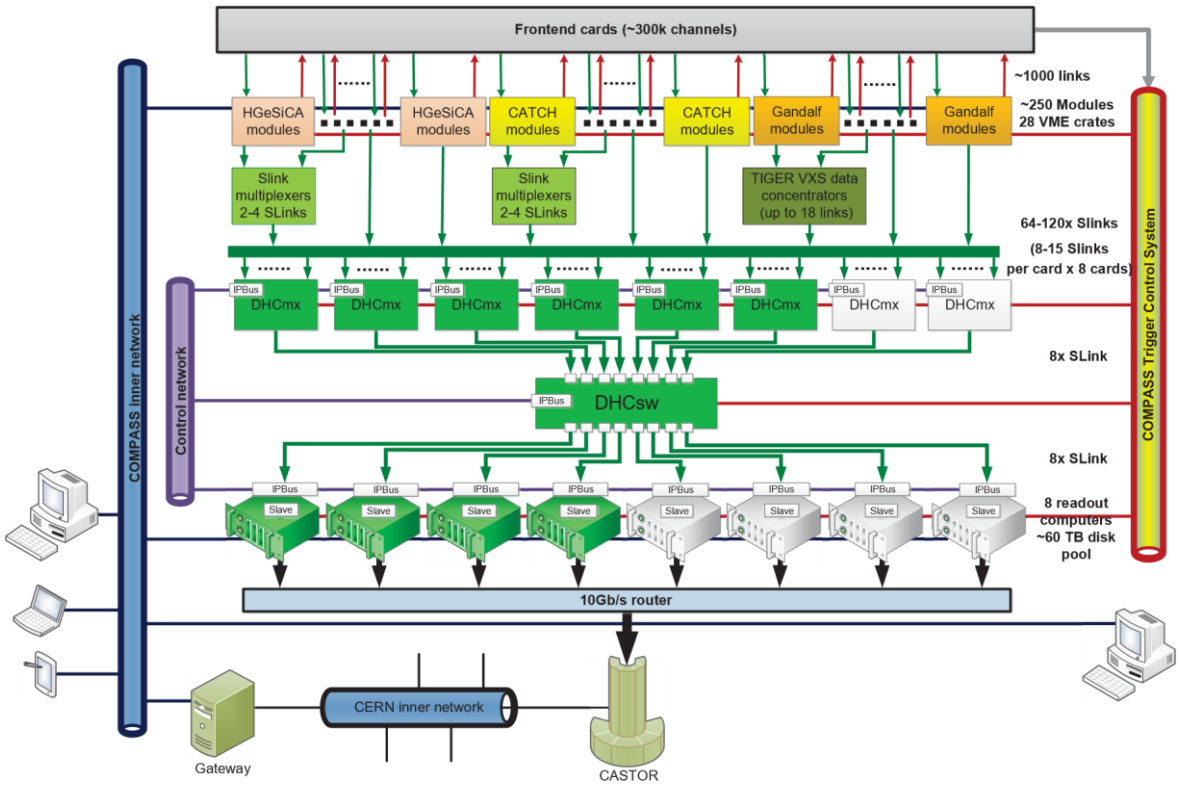


Fig. 1 The COMPASS iFDAQ topology

The iFDAQ software [8] is deployed on the readout engine, the individual computers of which run the Scientific Linux CERN 6 (SLC6) operating system [16]. The software is based on C++ and uses the Qt Framework not only for its GUI, but also for its threading. Furthermore, Qt data types and a variety of non-GUI classes are also used in the software. The Qt version used in the iFDAQ software is 5.5.1. Python and Bash script also find use in the iFDAQ, their scripts being particularly useful for starting processes remotely using SSH. Finally, XML is used to describe the hardware configuration of the iFDAQ in XML structure files and the IPBus [11] configuration in XML connection files and address files.

Six main functions are provided by the iFDAQ software: configuration of the hardware, monitoring of the data taking process, remote control of the hardware, data flow control, logging of information and errors and log browsing. The iFDAQ software also includes a connection to an MySQL database. The database is used to store,

among others: configuration information of the iFDAQ's hardware, information logs and error logs.

There are six types of processes fulfilling these six functions in the iFDAQ [6]: Master, Slave-control, Slave-readout, Runcontrol GUI, MessageLogger, and MessageBrowser. The Master process is responsible for control of the system by retranslation of messages from user to slaves according to configuration loaded from database. It has access to all slaves through DIALOG services and direct access to MySQL database. It also has integrated error recovery functions to cope with problems caused by misbehaving slave processes. The Slave-control process supervises connected FPGA card by accessing registers via IPBus. The full scale system will contain 17 Slave-control processes which will be distributed over the readout computers. The Slave-readout process is the most complex and demands most of CPU resources in the iFDAQ. It is a multi-threaded process that monitors readout activities and checks consistency of accepted data. A Spillbuffer card is used as the data source. The data are transferred between threads via signal-slot connections mechanism of Qt by blocks of about 512 events. Events are distributed to 10 processing threads before final checks and preparation of the final data format. Portion of data is, simultaneously with storing on the HDD, distributed to monitoring outputs. The main graphical user interface is implemented in Qt framework. It has been designed and developed with emphasis on ergonomy and flexibility. It provides iFDAQ status information for expert and nonexpert users. It runs in one of two modes: runcontrol and monitoring. There is only one runcontrol GUI allowed in the system; it controls and monitors state of system. The number of running monitoring GUIs is not limited, as they are used only for monitoring. MessageLogger and MessageBrowser ar the last two processes to be discussed. The MessageLogger receives messages from all parts of the system and stores them in the database. The MessageBrowser is a visualization tool for browsing through these messages. The master process and slave processes are based on state machines.

The original iFDAQ system of the COMPASS was based on the Data Acquisition and Test Environment (DATE) software [10], originally developed for the ALICE experiment at CERN for control of the hardware, therefore many user programs expect that data files are in the DATE data format. Transformation of read out data to DATE data format is needed because of this limitation.

C. The Motivation for the DIALOG Library Implementation

The DIM library was fully incorporated to all processes for the runs 2014 and 2015. The iFDAQ had to face several problems connected to the DIM library during that time. Messages were sometimes delivered truncated with length multiple 4 B. The iFDAQ solved that by adding artificial spaces to the end of messages. The next problem is more serious, the messages were sometimes not delivered at all.

However, the decision to implement a new communication library came with the last issue. Processes crashed without any obvious reason. Especially, Master process met this issue quite often. The debugging attempts sometimes terminated in the DIM library.

Unfortunately, the DIM library is a large package and to understand the source code is not a trivial task. The iFDAQ group made a decision to implement their own communication library.

Last but not least, the advantage of understanding the own library also played a key role in the decision making.

III. DIM Library

DELPHI [5] was one of the largest physics experiments in the world, it's online control system was composed of many different components distributed over many machines. In order to allow for efficient communication among machines and processes a communication system – DIM – was developed.

The processes involved in the DELPHI Online System needed to communicate efficiently and reliably across the different machines. The Online System was responsible for Data Acquisition, Trigger, Control, Monitoring, User Interfacing, etc. The DIM (Distributed Information Management) [4] system was proposed and implemented in order to provide the required communication layer.

A generic design and implementation of the DIM offers a wide usage in other platforms and for other applications. For this reason, it provides an opportunity to use it also in other experiments at CERN, e.g., L3, L3 Cosmics and NA50 and by BaBar at SLAC [3].

A. Design Requirements

All different types of activities in a system define different demands on a communication system. From the Data Acquisition System point of view, transfer speed, reliability, handling of large amounts of data and access to all the information available in the experiment are the most important aspects [3]. In order to accomplish the above-mentioned demands the DIM was designed meeting following requirements [3]:

- *Efficient Communication Mechanism* – A communication system should provide with an asynchronous behavior in a message exchange among processes to offer a communication in a most efficient way. Once a message is available for sending, the sending procedure is processed. Similar approach should be implemented for the receiving procedure in a process.

- *Uniformity* – All processes should use the same communication mechanism in order to be able to exchange all information within a system. Then the implementation and support of such system is more manageable.
- *Transparency* – To satisfy the system independence, a distributed communication system must fulfil transparency. Any running process should be able to communicate with any other process in the system regardless of their current running location.
- *Reliability and Robustness* – In a system with many processes running on many machines connected by network links, it might happen that a process, a machine or a network itself breaks down. The application should be able to deal with the loss of one of these components, i.e., providing for system recovery in a self-recoverable manner from error situations or the migration of processes from one machine to another.

IV. DIALOG Library

The name DIALOG represents conversation or interview in English, both connected to communication. Moreover, each letter is a first letter of a word characterizing somehow the library itself (distributed, inter-process, asynchronous, library, open, general). DIALOG library is designed in order to meet the following requirements:

- Any process should be able to access any information it needs in order to perform its processing or display activities.
- The integration to running system requires interface for an easy use.
- The information gathered should be consistent over all processes running at a given moment.
- Any process should be able to move from one machine to another.
- The communication system should be very robust. Any process dying should not disturb the rest system.

A. Description

In order to provide an easy recovery mechanism from crashes and migrate processes to another machine if necessary and satisfy the requirement for transparency, i.e., a client does not need to know where a server is running, the Control Server was introduced.

Information inside the DIALOG library is handled as named services. A service is a set of data with a name. The data inside a service can be of any type and size, since they are transferred as bytes. The Control Server keeps an up-to-date directory of all the processes and services available in the system.

To control particular process, commands are introduced. They are declared by specifying a name for the command and command with the same name can be registered by more processes. Once the command is delivered to process, generally, some action is taken.

The DIALOG library uses a client/server mechanism. A provider (server) is a process that has information to publish. It sends the list of services it provides to the Control Server at startup. A subscriber (client) is any process that uses a service. When requiring a service the subscriber asks the Control Server which provider provides that service and from then on contacts directly the provider. The Control Server knows at any time which services are available in the system and who provides them.

A recovery procedure is started whenever one of the processes (any process or even the Control Server itself) in the system crashes or dies. It includes the notification to remaining processes connected to it about the crashed process and reconnection as soon as a spare process will be available again. Moreover, this feature provides the possibility of balancing the machine load of the different workstations. By stopping a process in the first machine and starting it in the second one, a process can be easily migrated.

B. Integration

The DIALOG library is designed bearing in mind that it has to be integrated in a running system, so it has to be made as easy to use as possible. The library takes care of all the communications with the Control Server and with the other processes.

It gives all the DIALOG functionality to an existing process just by inserting one or two lines of code. This is possible because the system is completely asynchronous. Then all messages are sent from the OutgoingThread object and delivered to the IncomingThread object. According to the message header, one can recognize the message type easily.

C. Robustness

The DIALOG library has become the most important means of communication between processes in the iFDAQ, so a very special care has been taken on the recovery from error situations.

The establishment of a communication channel between processes is independent of the order in which they are started. The Control Server keeps track of the subscribers for “non-available” services and contacts them as soon as the providers start up. More generally when any provider or subscriber dies its partners will reconnect as soon as it comes back up.

When the Control Server starts all the providers will re-register their services. And the subscribers will re-request the services they need.

In order to make sure that the processes are in a good state, each process sends a heartbeat to the Control Server,

this way the Control Server can disconnect from a process or kill it if its behaviour is anomalous.

The communication between providers and subscribers once established is independent unless the Control Server dies. If the Control Server dies, the processes delete all information about other processes and try to reconnect to the new spare Control Server. Otherwise the behavior of the whole communication system would become unpredictable without any heartbeat check. Once the Control Server is on again, services are registered and subscribed as in a fresh start.

D. Implementation

1) Providers: Providers are processes that have information to provide. A process becomes a provider by declaring any services it can provide and any commands it is willing to accept. It sends this information to the Control Server.

A service is declared by specifying a name for the service, which is unique in the DIALOG system scope. A command is declared by specifying a name for the command and a command with the same name can be registered by more processes. Once the command is sent, it is forwarded by the Control Sever to all provider processes that registered it.

2) Subscribers: Subscribers are processes that need the available information in order to accomplish their tasks, that being display, monitoring or processing. In order to become a subscriber a process has to specify the service name it is interested in and requesting for it.

From then on the subscriber can go on with its work, any service message will automatically be processed whenever a service is received. At any time, a process can send a command to a provider by specifying the command name and the command message.

Any process can be a provider and a subscriber at the same time.

3) The Control Server: The Control Server keeps an up-to-date list of all the servers and services in the system, it receives registration messages from providers and service requests from subscribers. All processes send heartbeats at regular intervals so that the Control Server can be assured that they are functioning. If a process fails sending heartbeats the Control Server marks its services as not available, send the information concerning the crashed process to processes providing something to it and subscribing something from it. Once a spare process is started, it overtakes the same functionality as the crashed one.

The service uniqueness based on their names is a basic requirement for the system reliability. Any process trying to register a service being already registered is killed by a kill signal from the Control Server.

If the Control Server dies, the processes delete all information about other processes and try to reconnect to the new spare Control Server. Otherwise the behavior of the whole communication system would become unpredictable without any heartbeat check. When it comes back up all providers re-register all their services (they have been trying at regular intervals) and all the subscribers re-request the services they are waiting for and all connections are then established.

E. Scenarios

In the following section, we present the most typical scenarios the DIALOG library is dealing with. Each scenario is displayed in a particular data flow diagram followed by discussion.

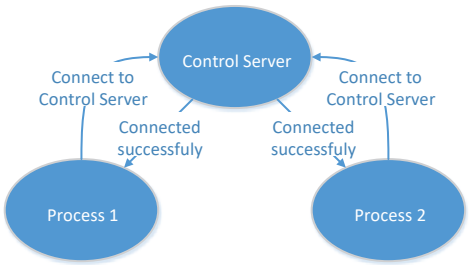


Fig. 3 The DIALOG connection to the Control Server diagram

In Fig. 3, data flow diagram shows the connection mechanism to Control Server for each process. Once the connection procedure is successful, the Control Server notifies to the particular process. The messages from

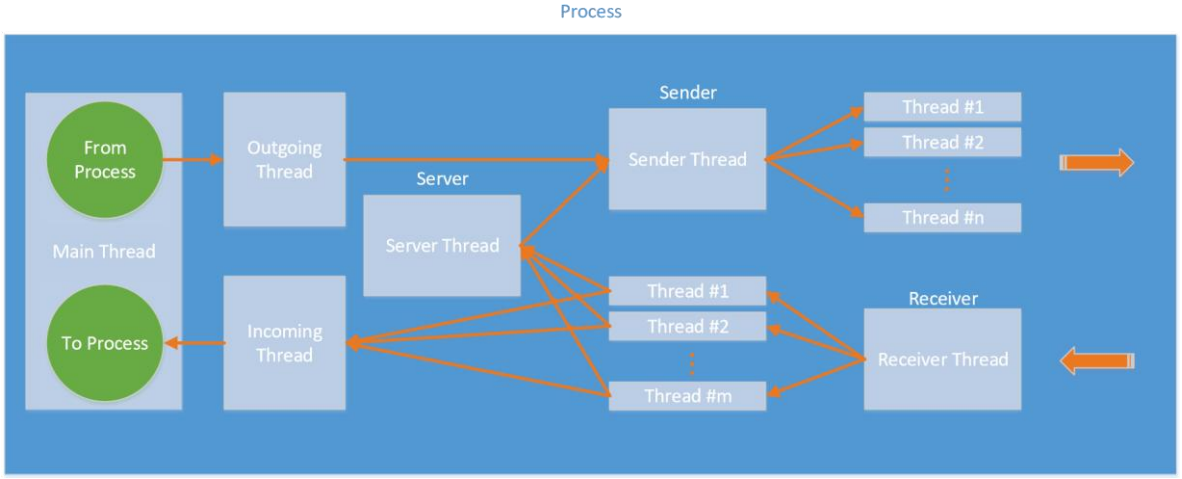


Fig. 4 The DIALOG system diagram

processes, which are not connected to the Control Sever, are ignored. It prevents the misleading or malicious behavior of processes not belonging to the DIALOG at all.

If the process is not connected to the Control Server, nevertheless it is still sending messages to the Control Server, the Control Server sends the message to the process, that it is not connected and probably it would like to connect.

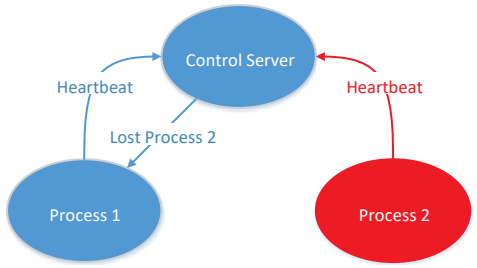


Fig. 4 The DIALOG heartbeats diagram

The heartbeat procedure is stated in Fig. 4. All connected processes are sending heartbeats to the Control Server at regular intervals. The Control Server is checking whether the hearbeat is received in a given checking interval for each process. In Fig. 4, the red color indicates a process which did not deliver its hearbeat in time. The process could fail or be stuck. Regardless of the real reason of the undelivered heartbeat, the Control Server considers the Process 2 as a lost one and notifies to all remaining processes.

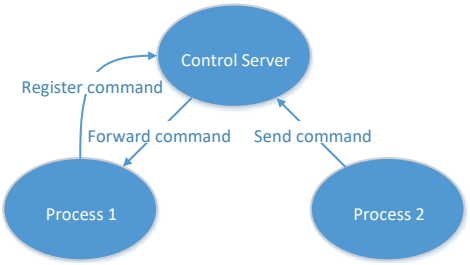


Fig. 5 The DIALOG command diagram

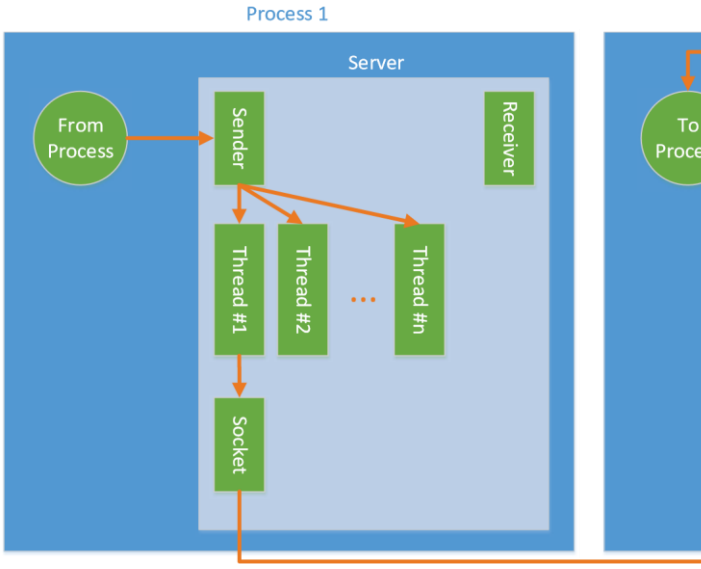


Fig. 6 The DIALOG communication between pr

The commands are significant part of the DIALOG library. In Fig. 5, the data flow diagram for commands is shown. Process 2 is sending a command with the command name and the command message. The command is sent from Process 2 to the Control Server, which knows all processes registering the command. The Control Server forwards the command to these processes and each process takes some action after the command delivery.

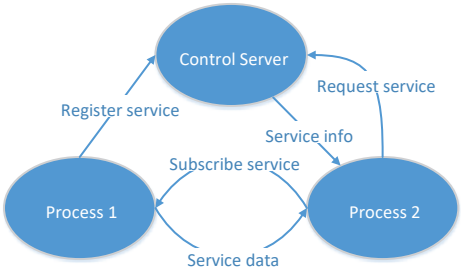


Fig. 7 The DIALOG service diagram

In Fig. 7, data flow diagram shows the control and data flow among the basic components of the DIALOG, the Control Server receives service registration messages from providers and service requests from subscribers. Once a subscriber obtains the “Service Info”, i.e. the service co-ordinates (hostname and port), from the Control Server it can then subscribe to services provided by provider process. If a subscriber sends a “Request Service” for a service that is not (yet) known to the Control Server a not-yet-provided “Service Info” is sent back to the subscriber but the request stays queued in the Control Server and when the service is made available a new “Service Info” is then sent to the subscriber and the subscriber proceeds to connecting to the provider.

In last two diagrams, we look deeper inside process and the DIALOG integration. In Fig. 2, the threads and

communication among them are described. We can divide the diagram into two parts. The first part is sending part, the second part is receiving one.

The Sender, running in the SenderThread, is taken care of dispatching messages among $n \in N$ threads and load balancing. These n threads are establishing connections to other processes, writing data to sockets and keeping sockets open until timeout. The socket is not closed immediately. It remains open for next messages to the particular process. If no message is sent to the particular process for some time, the socket is closed by timeout. That means, there is always one or no open socket from one process to the other. If the socket is already closed and new message must be delivered to the particular process, the socket is re-established again.

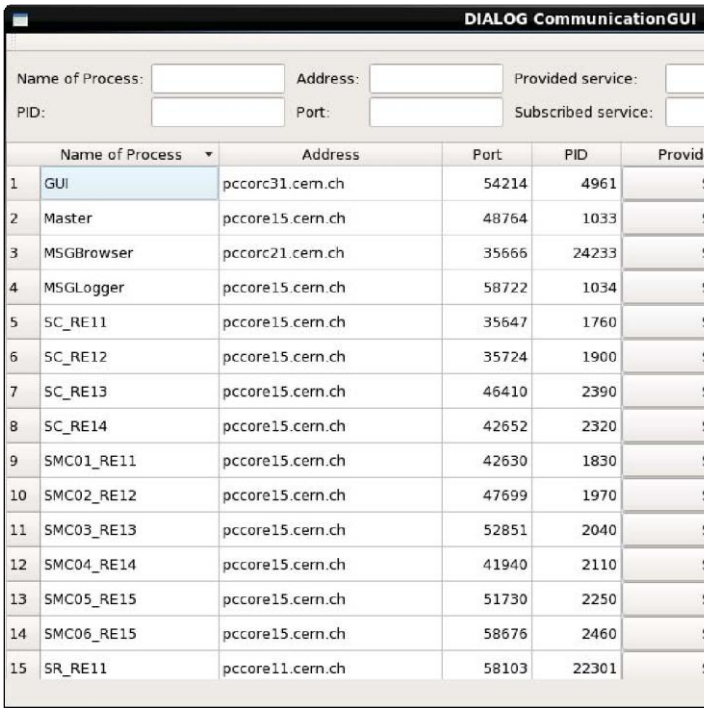
The message consists of two parts – from message header and message data. It is handling by pointers to them. Once the message is created, it is leaving the process as soon as possible. All these aspects – open socket, pointers to messages, sending as soon as possible – speed up the performance and reduce the latency significantly.

According to the message header, we distinguish message types and how to deal with them. Messages with header CONNECT TO CONTROL SERVER, REGISTERSERVICE, REQUESTSERVICE, REGISTER COMMAND, SERVICE MESSAGE and COMMAND MESSAGE are coming from the OutgoingThread to the SenderThread, to one of n threads and leaving the process. Messages with header HEARTBEAT and SUBSCRIBE SERVICE are coming from the ServerThread to the SenderThread, to one of n threads and leaving the process.

There are also message headers being sent only from the Control Server, e.g. SUCCESSFULY CONNECTED, CONNECTION LOST, INFO SERVICE, LOST SENDER or LOST RECEIVER.

All these message headers are created only once and used until the process terminates.

The second part is the receiving one. Once the Receiver, running in the ReceiverThread, receives a new socket descriptor, the socket descriptor is dispatched to one of $m \in N$ threads and the socket is created and opened. The Receiver is taking care of the new sockets only. The already established ones are keeping open until they are closed by sender process. These m threads are responsible for reading data out from sockets. Once a new message is read out, based on its message header, it is sent either to the ServerThread or to the IncomingThread.



	Name of Process	Address	Port	PID	Provid
1	GUI	pccorc31.cern.ch	54214	4961	
2	Master	pccore15.cern.ch	48764	1033	
3	MSGBrowser	pccorc21.cern.ch	35666	24233	
4	MSGLogger	pccore15.cern.ch	58722	1034	
5	SC_RE11	pccore15.cern.ch	35647	1760	
6	SC_RE12	pccore15.cern.ch	35724	1900	
7	SC_RE13	pccore15.cern.ch	46410	2390	
8	SC_RE14	pccore15.cern.ch	42652	2320	
9	SMC01_RE11	pccore15.cern.ch	42630	1830	
10	SMC02_RE12	pccore15.cern.ch	47699	1970	
11	SMC03_RE13	pccore15.cern.ch	52851	2040	
12	SMC04_RE14	pccore15.cern.ch	41940	2110	
13	SMC05_RE15	pccore15.cern.ch	51730	2250	
14	SMC06_RE15	pccore15.cern.ch	58676	2460	
15	SR_RE11	pccore11.cern.ch	58103	22301	

Fig. 8 The DIALOG GUI

The messages with message header CONNECT TO CONTROL SERVER, HEARTBEAT, SUCCESSFULY CONNECTED, REGISTER SERVICE, REQUEST SERVICE, REGISTER COMMAND, INFO SERVICE, CONNECTION LOST, LOST SENDER or LOST RECEIVER are sent to the ServerThread. Only SERVICE MESSAGE and COMMAND MESSAGE are heading to the IncomingThread directly.

We have not discussed the establishment of communication between two process in a precise way yet. In Fig. 6, the establishment of communication between two processes is presented. Process 1 is on left and Process 2 is on the right. Process 1 is trying to send a message to Process 2.

Process 1 sends message to the Sender dispatching it one of n threads. If the connection is not yet established, the object socket is created and opened in Process 1. In Process 2, if the connection between these two processes is not established yet, the Receiver receives the socket descriptor trying to connect to Process 2. The socket descriptor is

dispatched to one of $m \in N$ threads and object socket is created and opened. Based on the message header, then the message is sent either to the ServerThread or to the IncomingThread.

Socket objects live on both sides until either Process 1 closes it because of timeout, or one of the processes crashes or one of the processes terminates in a correct way. Once the connection is established, Process 2 can write to socket as much as it needs and Process 2 receives and reads out all these messages. If the connection terminates, sockets are deleted on both sides.

The established socket is used only for one direction connection. To send a message in an opposite direction from Process 2 to Process 1, the new socket must be established. That means there are either two open sockets, or only one open socket or no open socket at all between two processes at one point.

V. DIALOG Online Monitoring

The behavior of complex distributed applications can be very difficult to understand without the help of a dedicated tool for online monitoring. The DIALOG GUI allows the visualization of the processes involved in the application as shown in Fig. 8.

The DIALOG GUI provides information on the processes connected to the Control Server. It shows which services and commands are provided and which services are subscribed for each process. The online monitoring offers the possibility to listen to services and commands in a real time. There is also the functionality of sending commands from the DIALOG GUI directly.

If the DIALOG system contains many processes, a user appreciates the filter in the DIALOG GUI for an easy searching.

VI. Tests

Several tests have been conducted to validate system components. The performance of the DIALOG and DIM library has been measured. The system consists of 8 Slave-control processes for 8 MUXs (8 FPGA), 1 Slave-control process for SWITCH (1 FPGA), 8 Slave-control processes for Spillbuffer, and 8 Slave-readout processes (total 25 slaves processes). All slaves are sending messages concerning their status. There is also one Master process incorporated in this test. The Master is receiving all messages from all slaves concerning their status. This setup simulates the behavior of the iFDAQ full setup.

The test measures how many messages can be delivered to one single process in 1 second. The test is conducted for different message sizes and for each message size is conducted five times to obtain the sufficient statistics. Firstly, the test is using the DIALOG library. Afterwards, the test is performed also with the DIM library.

Before the start of the test, we have to pay attention to spreading of slaves among machines. Since on machines operating with Linux, if the Master process and slave are running on the same machine, the message is sent directly from slave to the Master process and it is not running through the network at all. If we had not considered that fact, the test results would have been even above the network bandwidth. In the test, the Master process is running on its own machine and nothing else is running there.

For the test, the network bandwidth is 10 Gbps. Based on the bandwidth, we can expect the maximum data rate ~1.2 GB/s (throughput). Moreover, the network bandwidth is not saturated by anything else and is exclusively at test's disposal.

message size over 10 KB. even with smaller message size than the DIM saturation. The DIALOG saturate reaches the maximum network bandwidth already with message size 2 KB. In comparison to DIM, it is much sooner. The DIM library is starting to occupy the whole bandwidth with message size 10 KB. At this moment, it is worth of mentioning the most frequent message size is between 1 KB and 2 KB in the iFDAQ.

Both libraries stand at the maximum network bandwidth with message size from 10 KB to 100 KB. Then the data flow of both libraries changes again. While the DIALOG data flow stands at the level of the maximum network bandwidth and saturates it regardless the message size until the end, the DIM efficiency starts to decrease. At the level of 100 KB message size, the DIM data flow starts to decline until the end markedly.

From a global perspective, based on both plots in figures, we can draw a final conclusion. The performance of the DIALOG library is significantly better than the DIM library.

Message size [B]

Fig. 9 Number of messages

VII. Conclusion

The iFDAQ was successfully deployed and commissioned in 2014, allowed to successfully take data for nominal Drell-Yan conditions during the run 2015 and followed by a run dedicated to DVCS in 2016.

The DIALOG library is a new communication library for iFDAQ of the COMPASS experiment at CERN. It is a replacement for the DIM library. The DIALOG library provides efficient and reliable inter-process communications across different platforms. It's communication mechanism is based on the publish/subscribe method and allows for asynchronous communications, task parallelism and multiple destination

updates. It's characteristics of efficiency and reliability have considerably improved the performance and robustness of the complete iFDAQ. It was fully incorporated to all processes in the run 2016.

The DIALOG is responsible for basically all communications inside the iFDAQ, in this environment it makes available around 100 services provided by 30 servers.

services are available at a a given time.
bandwidth in a more efficient way. is
searching for light dark matter.
development continues.

REFERENCES

[1] P. Abbon, et al.(the COMPASS collaboration): *The COMPASS experiment at CERN*. In: Nucl. Instrum. Methods Phys. Res., A 577, 3 (2007) pp. 455518.

[2] V. Y. Alexakhin, et al. (the COMPASS Collaboration): *COMPASS-II Proposal*. CERN-SPSC-2010-014, SPSC-P-340. May 2010.

[3] C. Gaspar, M. Donszelmann, Ph. Charpentier:¨ *DIM, a Portable, Light Weight Package for Information Publishing, Data Transfer and Inter-process Communication*. International Conference on Computing
in High Energy and Nuclear Physics, Padova, Italy, 1-11th February 2000.

[4] C. Gaspar, M. Donszelmann:¨ *DIM – A Distributed Information Management System for the DELPHI Experiment at CERN*. Proceedings of the 8th Conference on Real-Time Computer applications in Nuclear, Particle and Plasma Physics, Vancouver, Canada, June 1993.

[5] C. Gaspar, J. J. Schwarz: *A Highly Distributed Control System for a Large Scale Experiment*. 13th IFAC workshop on Distributed Computer Control Systems – DCCS'95, Toulouse, France, 27-29th September 1995.

[6] M. Bodlak, et al.: *Development of new data acquisition system for COMPASS experiment*. Nuclear and Particle Physics Proceedings, 37th International Conference on High Energy Physics (ICHEP). AprilJune 2016, vol. 273275, pp. 976981. Available at: http://dx.doi.org/10.1016/ j.nuclphysbps.2015.09.153.

[7] M. Bodlak, et al.: *FPGA based data acquisition system for COMPASS experiment*. Journal of Physics: Conference Series. 2014-06-11, vol. 513, issue 1, s. 012029-. DOI: 10.1088/1742-6596/513/1/012029. Available at: http://stacks.iop.org/1742-6596/513/i=1/a=012029?key= crossref.78788d23de2b4a6a34d127c361123b8c.

[8] M. Bodlak, et al.: *New data acquisition system for the COMPASS experiment*. Journal of Instrumentation. 2013-02-01, vol. 8, issue 02, C02009-C02009. DOI: 10.1088/1748-0221/8/02/C02009. Available at: http://stacks.iop.org/1748-0221/8/i=02/a=C02009?key=crossref. a76044facdf29d0fb21f9eefe3305aa5.

[9] M. Bodlak, et al.: *Developing Control and Monitoring Software for the Data Acquisition System of the COMPASS Experiment at CERN*. Acta polytechnica: Scientific Journal of the Czech Technical University in Prague. Prague, CTU, 2013, issue 4. Available at: http://ctn.cvut.cz/ap/.

[10] T. Anticic, et al. (ALICE DAQ Project): *ALICE DAQ and ECS User's Guide* CERN, EDMS 616039, January 2006.

[11] C. Ghabrous Larrea, et al.: *IPbus: a flexible Ethernet-based control system for xTCA hardware*, 2015 JINST 10 C02019. doi:10.1088/1748-0221/10/02/C02019.

[12] CASTOR – CERN Advanced Storage manager. Available at: http: //castor.web.cern.ch. (Accessed: 2017-05-01).

[13] Electronic developments for COMPASS at Freiburg. Available at: http: //hpfr02.physik.uni-freiburg.de/projects/compass/electronics/catch.html. (Accessed: 2017-05-01).

[14] The GANDALF Module. (online). Available at: http://hpfr03.physik.uni-freiburg.de/gandalf/pages/information/about-gandalf.php?lang=EN. (Accessed: 2017-05-01).

[15] iMUX/HGESICA module. (online). Available at: https://twiki.cern.ch/ twiki/pub/Compass/Detectors/FrontEndElectronics/imux\ manual.pdf.

(Accessed: 2017-05-01).

[16] Linux at CERN. (online). Available at: http://linux.web.cern.ch/linux/ scientific6/. (Accessed: 2017-05-01).

[17] S-Link – High Speed Interconnect. (online). Available at: http://hsi.web. cern.ch/HSI/s-link/. (Accessed: 2017-05-01).