Introducing high performance distributed logging service for ACS

Jorge A. Avarias[a], Joao S. López[b], Cristían Maureira[b], Heiko Sommer[c] and Gianluca Chiozzi[c]

[a] National Radio Astronomy Observatory, Socorro, NM, USA
[b] Computer Systems Research Group, UTFSM, Valparaíso, Chile;
[c] European Souther Observatory, Garching bei München, Germany;

ABSTRACT

The ALMA Common Software (ACS) is a software framework that provides the infrastructure for the Atacama Large Millimeter Array and other projects. ACS, based on CORBA, offers basic services and common design patterns for distributed software.

Every properly built system needs to be able to log status and error information. Logging in a single computer scenario can be as easy as using `fprintf` statements. However, in a distributed system, it must provide a way to centralize all logging data in a single place without overloading the network nor complicating the applications.

ACS provides a complete logging service infrastructure in which every log has an associated priority and timestamp, allowing filtering at different levels of the system (application, service and clients). Currently the ACS logging service uses an implementation of the CORBA Telecom Log Service in a customized way, using only a minimal subset of the features provided by the standard.

The most relevant feature used by ACS is the ability to treat the logs as event data that gets distributed over the network in a publisher-subscriber paradigm. For this purpose the CORBA Notification Service, which is resource intensive, is used. On the other hand, the Data Distribution Service (DDS) provides an alternative standard for publisher-subscriber communication for real-time systems, offering better performance and featuring decentralized message processing.

The current document describes how the new high performance logging service of ACS has been modeled and developed using DDS, replacing the Telecom Log Service. Benefits and drawbacks are analyzed. A benchmark is presented comparing the differences between the implementations.

Keywords: ALMA Common Software, Data Distribution Service, Logging, Distributed Systems

1. INTRODUCTION

1.1 The ALMA Common Software

Atacama Large Millimeter/Sub-Millimeter Array (ALMA)[1] is a joint project between astronomical organizations of Europe (ESO), North America (NRAO), and Japan (NAOJ). ALMA is a large radio-astronomical project that will consist of at least 50 twelve meter antennas operating in the millimeter and sub-millimeter wavelength range, with baselines up to 10 [km]. It will be located at an altitude above 5000 [m] on the Chajnantor plateau in the middle of the Chilean Atacama desert. The science commissioning of ALMA will start in 2012, time for which the array will be fully operational for astronomical observations.

ALMA Common Software (ACS)[2–5] is an Object Oriented CORBA-based middleware software framework (for science facilities) that handles communication between distributed objects. ACS was designed and developed to support the complex control requirements of ALMA radio telescopes, but can be used to support control and data flow for any system with similar performance requirements.[6]

ACS provides a set of packages containing development tools, and common services and patterns needed to build and deploy object oriented and distributed systems. Most of the features provided by ACS are implemented using off the shelf components. ACS itself provides "glue layers" between these components, hiding all the details of the underlying mechanisms and complex CORBA features from the developer.

The ACS architecture is based on the *Component-Container model*.[7] *Containers* provide an environment for several portions of software called *Components*. They also provide several services to, and manage the lifecycle of, the Components. This way, Components developers can focus on domain problems rather than on software engineering issues.[6]

1.2 Telecom Log Service

The Telecom log service specification defines a set of IDL interfaces that are suitable for an implementation of any kind of a log service.[8] The telecom log service is modeled on the CORBA notification service and uses event-aware objects and an event channels to manage the logging of events to the persistent store. This implementation allows logs to generate events relating to the log and propagate them to their clients, filter events for logging, and forward events from suppliers to consumers.

1.3 Data Distribution Service

Data Distribution Service (DDS) is a formal specification from Object Management Group (OMG) for Publish-Subscribe data in distributed systems.[9] The specification describes the service using UML models providing a platform independent model that can be implemented in several programming languages.[10]

This service is not specific to CORBA, but there are DDS implementations that use the CORBA platform like OpenDDS.[11] DDS targets real-time performance specifying Quality of Service (QoS) on each entity described in the model to assure predictable behavior and performance.[10,12]

The DDS specification defines several entities to accomplish the communication: *DomainParticipant*, *DataWriter*, *DataReader*, *Publisher*, *Subscriber* and *Topic*. Each one can be configured through QoS policies defined specifically for each entity type.

1.3.1 OpenDDS: An open source DDS implementations

OpenDDS is an open source, C++ implementation of the OMG Data Distribution Service type DDS3 specification, aiming at the Minimum Compliance Profile defined by the DDS specification.[9] OpenDDS is built on top of ACE[13] and uses the TAO[14] CORBA implementation to provide functionality like the IDL processor, the pluggable transports and some service initializations.[11] OpenDDS does not fully implement the DDS standard and currently only provides a partial implementation of the DCPS layer.

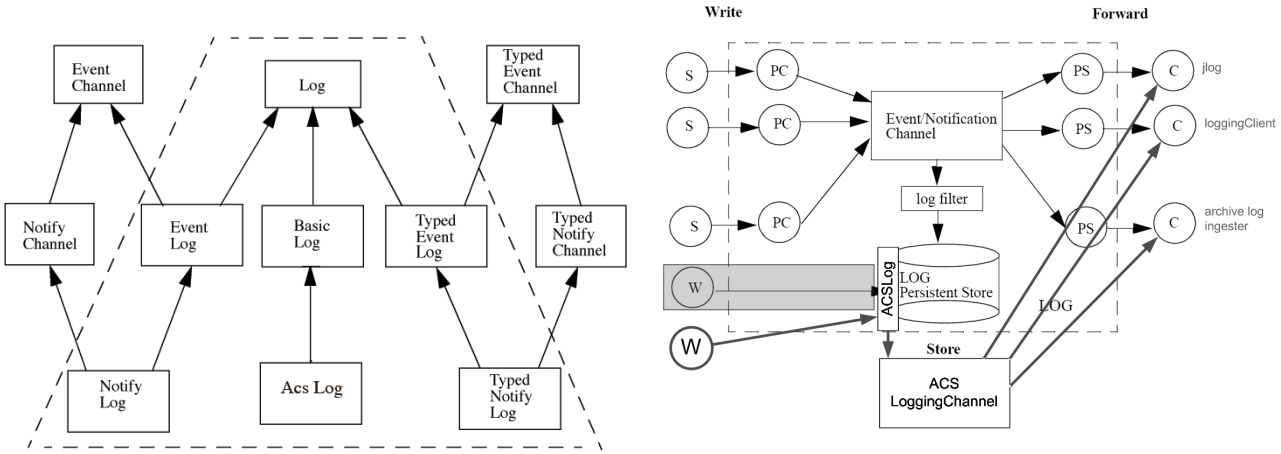
OpenDDS uses CORBA interfaces as defined by the DDS specification to initialize and control service usage. Data transmission is accomplished via a OpenDDS specific Pluggable Transport layer that allows the service to be used with a variety of transport protocols (conceptually, the architecture borrows from TAO's Pluggable Protocols Framework). OpenDDS currently supports TCP and UDP point-to-point transports as well as unreliable and reliable multicast transports.

2. ACS LOGGING SERVICE

The ACS logging subsystem[15] leverages three systems that have been either already built, or at least well designed and specified. These are:

- CORBA Telecom Log Service for centralizing all log entries generated throughout the system.
- CORBA Notification Service for distributing log entries to interested clients (logs consumers) when the entries are submitted to the centralized logger.
- The mechanism for generating, formatting, filtering and caching log entries by clients, components and containers.
  - C++ logs suppliers use the ACE Logging framework with its C++ API;
  - Java logs suppliers use the standard Java Logging API;
  - Other log suppliers, e.g. a Python application, can use the stand-alone ACS Log Server which provides the generic functionality.

This rest of the description will be focused in the C++ part of the ACS logging system.



(a) ACS Log relationship with Telecom Log classes

(b) Parts of the Telecom Log Service used by ACS

Figure 1. ACS Centralized log service representation

2.1 Parts of the logging system

2.1.1 Centralized Log

The Centralized Log that implements the CORBA Telecom Log Service is the facility that receives log entries from the entire system and dispatches them to interested clients. Particularly, it implements the Log interface of the Telecom Log Service shown by figure 1(a). Currently ACS is using Telecom Log Service implementation provided by The ACE ORB (TAO), but in a very custom way, using just a minimal subset of the functionality provided by the standard and the implementation.

The Centralized Log uses CORBA for getting the routed logs from the logs suppliers (publishers), applications that use the formatting, buffering and pushing capabilities of ACE API, Java Logging API, the ACS Log Service, etc. as they send log records to it. Thus, the Centralized Logging Service receives an XML string as the "any" parameters in a call to `write_records` from a supplier. The XML string is formatted according to the description described in 2.1.2. Since the XML string already contains the timestamp information, it should take precedence over the actual current timestamp of the reception of the log entry.

The parsed log entry is then forwarded to the appropriate event channel of the CORBA Notification Service, which further distributes the log entries. The choice of the event channel depends on the log entry's type and content. For example, log entries related to debugging could be forwarded to a different event channel than those related to archiving. The logs consumers (clients) can access log entries by subscribing to the event channel of interest as shown in figure 1(b).

2.1.2 Log Entry

Every log entry is represented as an XML document node. The representation is shown in listing 1.

The elements of the XML Log Entry are briefly described below:

- **TimeStamp:** The timestamp is a mandatory attribute of every log entry. It specifies the exact time when the log entry was submitted. The time is encoded in ISO 8601 format with a precision to one millisecond.
- **Source Code Information:** The element representing a log entry is equipped with these attributes that convey the location in the source code from which the log entry was generated: *File*, *Line* and *Routine*. These three attributes are optional since they cannot be provided for log entries in each language due to grammar or implementation restrictions.

Listing 1. A generic log entry representation in XML

```
<LogEntryType TimeStamp="yyyy-MM-ddThh:mm:ss.fff"
        File="filename" Line="lineno"
        Routine="routine"
        Host="hostname" Process="procname" Thread="threadname" Context="context"
        StackId="stackid" StackLevel="stacklevel"
        LogId="id" Uri="uri"
        Priority="p">
    <Data Name="name">value</Data>
    log entry message
</LogEntryType>
```

- **Runtime Context Information:** The log element has six attributes that give more information regarding the runtime context in which the log entry is submitted: *Host*, *Process*, *Thread*, *Context*, *StackId* and *StackLevel*
- **Priority:** The log type implies default priority of a log message. However, if the priority is explicitly specified, then the default is overridden. Priority is measured as an integer number ranging from 1 (TRACE) to 11 (EMERGENCY), where 1 is lowest and 11 highest priority.
- **Log Entry Message:** The optional log entry message is a string of characters. The message can be either an XML formatted string, or a CDATA section. The only rule it must obey is not to contain a sub-string]] > or characters such as '<', '>' or '&', since they would terminates a CDATA section.

2.1.3 ACS C++ Logging

The current ACS C++ Logging API for generating, formatting and filtering log entries is based on the ACE Logging API and is provided by a collection of operating system wrappers and common design pattern implementations. The deployment for ACS C++ logging system is shown in figure 2. The figure shows that an instance of the `LoggingProxy` class is created by every Container. This instance is created and destroyed by the container life cycle and configured using the ACS Configuration Database (CDB). This proxy has the responsibility to send the log entries to the centralized logger (`ACSLog`) and to filter the logs at component level.

The C++ API provided by ACS consists several macros, who help the end user to fill a structure which contains all the data described in 2.1.2. This structure is then passed to the `LoggingProxy` to be formatted in the XML format defined in 2.1.2. Finally `LoggingProxy` sends the data to the centralized logger using a synchronous CORBA `write_records` call.

2.2 Logging service approach shortcomings

The ACS logging system implementation as was described is functional and it meets the performance according to the requirements as long certain conditions are met. These performance issues are inherited by the Telecom Log Service and the Notification Service usage, the problems which most affect to this service are:

- The decreasing performance when there are too much consumers attached to the outgoing Notification Channel (e.g. many developers running their own jlog instances)
- The presence of a slow consumer in the log event transfer, this is a common scenario when a consumer (e.g. jlog) is executed in a machine with very high load, or when the consumer takes very long time in process the log event. This behavior will do slower log event transfers.
- Every log record is wrapped by a CORBA Any, CORBA un-/marshalling of Anys is very slow compared to fixed data types. The impact of this is noticed especially when trying to change the log record format from XML strings to an IDL-defined `struct`. The gain from not having to construct and parse XML is overcompensated by the additional work of un-/marshalling a complex `struct` into an `Any` object.

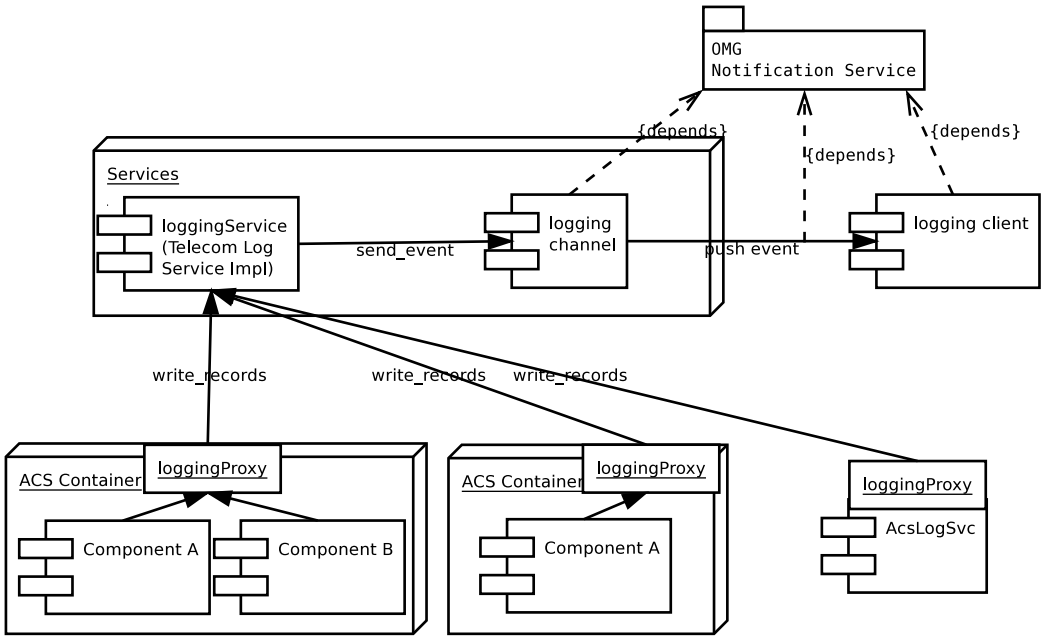


Figure 2. ACS logging service deployment

There are also particular issues coming from the implementation of the ACS centralized logger: the service delivers the Log events to the consumers as individual log records, which is usually slower than send an event with a sequence of Log Events.

2.3 Improving the logging service

The improvements that can be done over the logging system are bound to the current technologies already used as part of it. Therefore, it is not possible to avoid all the issues related to the Notification Service. So problems related to the numbers of clients connected to a Notification Channel and the presence of a slow consumer as part of the clients cannot be avoided at all, and they still represent a huge problem when the service is deployed in real scenarios. Then, the only way to improve the throughput and the performance of the current logging system implementation is doing some changes to avoid as much as possible the un-/marshalling of the CORBA `Anys`.

According to the Telecom Log Service standard, the event is represented by a `struct` that contains fields with the relevant data of the log described is section 2.1.2, but also the XML string is included in the same event, which means that the information is included twice although the logging system just cares about the XML data. To avoid extra processing in the `Any` transformation, a new Log record `struct` was defined, containing only the priority of the log to be used in the filtering of the centralized logger and the XML data to be used by the clients consuming the events. This new `struct` is shown in the listing 2. This log record description is not part of the current standard of the Telecom Log Service, but ACS is extending the current standard to use its relevant features and to provide a system with more throughput.

Also, as part of this change, and to avoid extra CPU usage in CORBA un-/marshalling, each event sent through the Logging Notification Channel will contains a sequence of log records, and each log record will be put in the sequence in the same order as they arrive to the centralized log record. For cases when it is necessary to send log records with a very high priority (like EMERGENCY priority), the original functions still exists, allowing to send one log record at a time in the event. The responsible of doing this distinction is the Logging Proxy. With this approach the system ensures that the very important log events will arrive as soon as possible, avoiding all the extra waiting time gained if a sequence is filled.

Listing 2. The definition of the new Log record struct

```
struct XmlLogRecord{
    // xml string that represents the Log
    string xml;

    // Log level of the log record
    AcsLogLevels::logLevelValue logLevel;
};
```

To avoid conversion among different data structures, the `XMLLogRecord` is used across the entire logging system. Therefore the Centralized Log service interface exposes the new `void writeRecords(in XmlLogRecordSeq xmlLogRecords)` method. This method is used by the LoggingProxy to send a sequence of XML log records to the centralized logger, replacing the old `write_records` method defined by the Telecom Log Service.

3. DATA DISTRIBUTION SERVICE BASED LOG SERVICE

As described in section 2.1.1, the remote centralized logger uses the Notification Service to send the log events to the logging clients. Currently ACS ships with two clients: `jlog`, a Java-based GUI that shows and filters log records, and `loggingClient`, a command line interface tool that is able to connect to the logging channel and show the arriving log records. The ACS Centralized logger service uses the Telecom Log Service in a very customized fashion, and actually the only essential component in use is the communication through the logging notification channel; the rest of the standard is not used at all. Because this, the Telecom Log Service can be easily replaced by another publisher-subscriber mechanism to propagate the logs in an asynchronous way. In this aspect, DDS has proven to have very good performance and throughput results in the past, and it is already integrated and tested in the ACS environment as part of a prototype to replace the current implementation of the ACS Notification Channel,[16] also based in OMG Notification Service standard.

The replacement of the Notification Service by DDS in the Centralized logger is a very straightforward task. The process consists only in changing the logging push supplier used in the classes that extend the Telecom Log Service in ACS by a DDS Publisher. In the client side, the change is also trivial, replacing the logging push consumer by a DDS subscriber.

The usage of DDS should help to increase the throughput of the centralized remote logger because of the following reasons:

- It removes the need to use a generic type in the event transmission. DDS doesn't use CORBA `Any`s and the marshaling/unmarshaling will no longer be used, saving CPU time
- The size of DDS event is less than a Notification Service Event (using a CDR representation of a CORBA `Any`). This saves network bandwidth.
- It removes the problem of a slow consumer attached to the logging channel because there are QoS settings available in DDS that can be set to assure a maximum blocking time.[16]
- It decreases the penalty of the performance of the entire system when there are multiples clients attached to logging channel according to tests and the results already done in.[16]

On the other hand, there are some drawbacks in the usage of DDS (OpenDDS). The following are the most important issues:

- There is no native DDS implementation for Java or Python, both programming languages supported by ACS. The only way to use a DDS Publisher or DDS Subscriber in Java is using JNI, but in the past JNI has been the cause of several containers crashes, and is always avoided when possible. For Python, the situation should be easier than in Java, and it should be enough to provide a wrapper. Alas, at the moment of writing this solution doesn't exist.

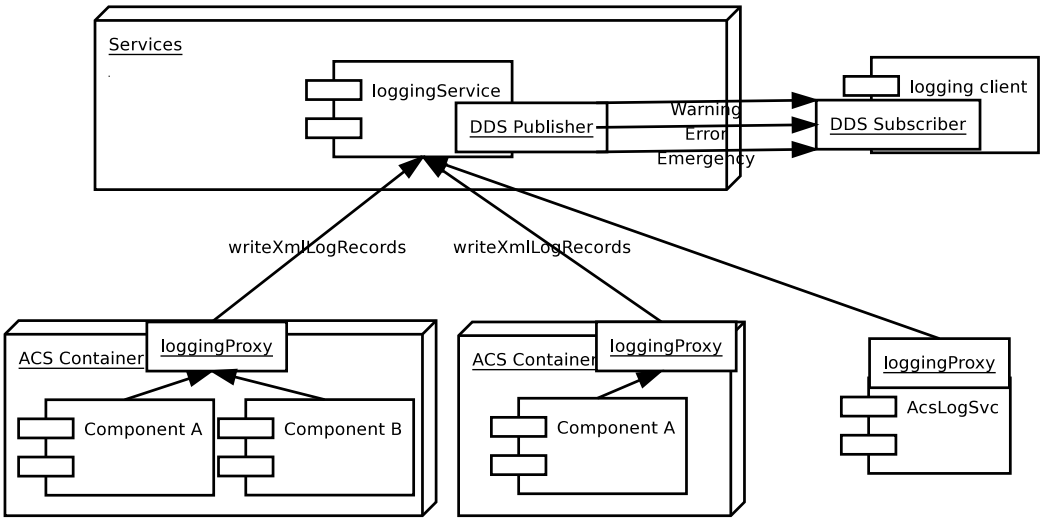


Figure 3. ACS DDS logging service deployment, the subscriber is connected to the Warning, Error and Emergency channels. Eventually a consumer could be connected to the 10 channels at the same time.

- The performance penalty of the system when there are several clients connected still exists, but it is lower than in the case when the Notification Service is used.[16]

3.1 Designing the DDS logging service

As a fundamental concept, the data event used in the DDS-based implementation of the logging system will be represented by the same `struct` already defined in listing 2. The principal change is introduced with the priority mapping of the log events into the logging channels available. In the implementation that uses Notification Service there is just one channel to send all the logs events. This approach makes the priority of the log irrelevant, and the Centralized Logging Service is used only as an intermediary between the log records suppliers and the log records consumers (i.e., there is no extra processing or filtering on it). The logging channel is a NotifyService process running in the service machine, every log event or sequence of log events is pushed by the ACS Centralized logger to the logger Notification Channel, then the logging channel process pushes the events to the interested clients. In the DDS implementation the channels are a logical entities called *topics* which are not processes like in the Notification Service implementation. Topics are just a direct connection between the publisher and the subscriber. In practice, a publisher can have multiples channels available, but the subscriber can be connected just to a subset of them.

In the DDS implementation of the Logging Service, topics are used as priority channels for the different log events. There is a channel for each priority level defined in the logging service (10 different log levels in total). The main ideas behind this concept are: the separation of concerns inside the logging service allowing to set special QoS parameters to each channel according to the priority level, and to avoid the excessive filtering done by the clients allowing to them to connect only to the "interesting" channels. For example, if some bug occurs during the normal operation of the system, the system should be halted, then the log filter level must be changed to avoid the discarding of the debug log records and finally, in the worst case, the entire system must be restarted. With this idea it is necessary just to attach a "debug" client to the debug channel to start receiving the debug log records.

A common deployment of the DDS logging service is shown in figure 3. The main changes are done in the services node, where the Logging Channel process disappears and the connection and events transfer and connection between the centralized logger and the clients is direct.

A separate filtering rule used by the logging service is the *Audience* field. Currently, the audience is filtered in the client side after parsing the XML string. DDS provides a logical partition of the topic called *Partition*. This feature can be used by the centralized logger to target the log record to the right audience (partition) with

the right priority (topic). The partition can be used to target several audiences at the same time. It is even possible to send "broadcast" events to all the audiences, leaving the partition QoS as an empty sequence.

4. RESULTS AND COMPARISON

In order to quantify the benefits of using DDS over the current logging system, several tests were run. Two aspects are considered:

- Throughput (logs/s)
- CPU usage of the logging service

A detailed test suite can be found in.[16]

4.1 Tests configuration

The software configuration used to run the tests is the following:

- ACS 9 Pre-release running on RHEL 5.4 32bits (Intel Core 2 Duo @2.54 Ghz).
- ACE version 5.6.5, provided as part of ACS 9 Pre-release.
- TAO version 1.6.5, provided as part of ACS 9 Pre-release.
- OpenDDS version 1.1.

Different versions of the ACS Centralized logger were tested for comparison purposes:

- Logging before ACS 8.1: This is the plain ACS centralized logger without any changes.
- Logging after ACS 8.1: This is the ACS Centralized logger that includes the changes explained in section 2.3.
- Logging DDS: This is the ACS Centralized logger using DDS and including the improvements done in the logging after ACS 8.1, in this case the events are sent one-by-one over the logging channels.
- Logging DDS w/ Sequence: This is the ACS Centralized logger using DDS and including the improvements done in the logging after ACS 8.1, but in this case the log events are sent in sequences with length equals to the size of log records sent by the logging proxy.

4.1.1 Test 1: Logging without clients

The first test consists of an ACS Component that in a specified time frame logs messages, as fast it can, without clients listening. In this test it was also verified that the message size was irrelevant in the performance. The following table shows the results and compares it with the current logging system:

Logging system used	Throughput $[log/s]$	logging service CPU use [%]	NotifyService CPU use [%]
Logging before ACS 8.1	~ 5.000	~ 60	~ 50
Logging after ACS 8.1	~ 24.000	~ 35	~ 15
Logging DDS	~ 31.000	~ 55	N/A
Logging DDS w/ Sequence	~ 27.000	~ 10	N/A

4.1.2 Test 2: Logging with one client

The second test is similar to the fist one, but it has a client, specifically the loggingClient provided by ACS. The following table shows the results and comparison:

Logging system used	Throughput $[log/s]$	logging service CPU use [%]	NotifyService CPU use [%]
Logging before ACS 8.1	~ 4.000	~ 55	~ 65
Logging after ACS 8.1	~ 17.000	~ 30	~ 25
Logging DDS	~ 25.000	~ 50	N/A
Logging DDS w/ Sequence	~ 23.000	~ 10	N/A

5. CONCLUSIONS

The tests and their results verified that the plain implementation used by years in ACS Logging system was very inefficient, it used a considerably amount of CPU, and it had a throughput of around 5.000 logs/s, that is very slow in comparison with the logging service with improvements. The improvements described in section 2.3 allow to the system improve the throughput in around five times and also to reduce considerably the usage of the CPU of the logging service process and the Notification Service process, this is translated in the deployment as less load of the system which runs the ACS services, in the practice is possible to notice less CORBA Timeouts exceptions because the logging processes leave more CPU time for other services to be executed in the same machine.

The DDS implementation increases even further the throughput of the already improved ACS Centralized logger. In the CPU usage aspect, when the DDS implementation handled the events one by one, there is an increment in the CPU usage, but when the DDS implementation that uses sequences to sent the log events the CPU usage is very low because there are less operations to do when the service has to move an entire block of data, in this same implementation there is a decrement of the throughput in comparison with the DDS implementation that handles the events one by one, this is explained because the memory used to represent the sequence has to be copied entirely when it is processed by the DDS DataWriters and the memory operations are slow in comparison with the CPU.

When there are clients attached to the logging channel the throughput decreases in all the cases, but the logging service implementations using the Notification Service suffers a higher throughput decrease. Also the CPU usage decreased a little in all the cases explained by the less log event sent through the channel.

6. FUTURE WORK

DDS provide a wide range the QoS settings, all of them have different impacts in the performance of the Topics and should be studied and tested to measure to know if they can be used by the DDS Logging Service described in this document.

It should be studied the usage of resources not considered during the execution of the tests presented. It is important to know the real impact of the network bandwidth with the different implementations and see if DDS, specifically OpenDDS, provide a better usage of this resource. Also the memory usage is big concern, it should be verified the memory footprint for each one of the implementations presented here.

Finally the current design of the ACS Logging Service could be improved removing all the intermediaries between the logging proxy and the logging clients. With DDS this should be an easy task that can boost even more the performance of logging system.

REFERENCES

[1] Hibbard, J. E., Corder, S., and ALMA Project Team, "Status of the Atacama Large Millimeter/Submillimeter Array," in [*Bulletin of the American Astronomical Society*], *Bulletin of the American Astronomical Society* **41**, 567–+ (Jan. 2010).

[2] Chiozzi, G. et al., "CORBA-based common software for the ALMA project," in [*Proceedings of SPIE 2002*], (2002).

[3] Chiozzi, G. et al., "The ALMA Common Software: A developer friendly CORBA-based framework," in [*Proceedings of SPIE 2004*], (2004).

[4] Raffi, G., Chiozzi, G., and Glendenning, B., "ALMA Common Software (ACS) as a basis for a distributed software development," in [*Proceedings of the 11th Astronomical Data Analysis Software & Systems Conference*], (2001).

[5] Chiozzi, G., Sekoranja, M., Caproni, A., Jeram, B., Sommer, H., Schwarz, J., Cirami, R., Yatagai, H., Avarias, J. A., Hoffstadt, A. A., López, J. S., Grimstrup, A., and Troncoso, N., "ALMA Common Software (ACS), status and development," in [*Proceedings of ICALEPS*], (Oct. 2009).

[6] Chiozzi, G., Gustafsson, B., Jeram, B., Sivera, P., Pleško, M., Šekoranja, M., Tkačik, G., Žagar, K., and Fugate, D., "ACS, a CORBA-based Common Software for ALMA and other projects." `http://www.esrf.eu/conferences/Corba_Controls/PAPERS/chiozzi2.pdf` (2002).

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M., [*Design Patterns: Elements of Reusable Object-Oriented Software*], Addison Wesley Professional (1994).

[8] Object Management Group, "Telecom Log Service Specification - formal/03-07-xx," (2003).

[9] Object Management Group, "Data Distribution Service (DDS) for Real-time Systems," (January 2007).

[10] Gerardo Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in [*Proceedings of the 23rd International Conference on Distributed Computing System Workshops (ICDCSW'03)*], (2003).

[11] Object Computing, Inc., "OpenDDS Developer's Guide, TAO Developer's Guide Excerpt Chapter 31," (2007).

[12] Joshi, R., "Building an effective real-time distributed publish-subscribe framework, part 1." `http://www.embedded.com/columns/showArticle.jhtml?articleID=191800680` (2006).

[13] Schmidt, D., "The ACE library webpage." `http://www.cs.wustl.edu/~schmidt/ACE.html`.

[14] Schmidt, D., "The ACE ORB webpage." `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[15] Žagar, K. and Georgieva, R., *Logging and Archiving*. ALMA, 1.36 ed. (2007).

[16] Avarias Alfaro, J. A., "Data Distribution Service as an alternative to the CORBA Notification Service for ALMA Common Software." Memoria para optar al Título de Ingeniero Civil Informático, Universidad Técnica Federico Santa María (December 2008).