Integrating the CERN LASER Alarm System with the Alma Common Software

A.Caproni[*ab], K.Sigerud[c], K.Zagar[d]

[a] European Southern Observatory, *Karl-Schwarzschildstr. 2, D-85748 Garching, Germany*

[b] INAF-OAT, Osservatorio Astronomico di Trieste, via G.B. Tiepolo 11, I-34131 Trieste, Italy

[c] European Organization for Nuclear Research, CERN CH-1211 Genève 23 Switzerland

[d] Control System Laboratory, Teslova ulica 30, SI-1000 Ljubljana, Slovenia

ABSTRACT

An alarm system is a cornerstone service in every computer controlled environment. Its purpose is the notification of exceptional conditions in the system requiring an intervention from the staff.

The specifications for the alarm system in the Alma Common Software (ACS) require not only that each alarm has to be shown to operators in a short time, but also that correlated alarms must be "reduced" and presented in compact form in such a way that operators are able to easily identify the root cause for an abnormal condition.

In the development of ACS we always investigate the availability of adequate implementations before writing a service from scratch. Such an implementation, the CERN Laser Alarm System, developed for the Large Hadron Collider, was fulfilling and exceeding our requirements.

We have therefore started a pilot collaboration project to verify the possibility of integrating Large Hadron Collider Alarm Service (LASER) into ACS. A test suite was developed to demonstrate that the full chain of events starting from the publication of new alarms from a set of sources to their representation in a GUI happened as expected. Particular attention was given to the reduction mechanism for its importance in helping the operators in finding the real cause of each problem in a short time.

The project showed that it is possible to integrate two different software systems if they are written with well defined interface and have a similar infrastructure. In this paper we describe the modifications we introduce to integrate CERN LASER into ACS.

Keywords: ALMA, alarm system, ACS, CERN, Laser, control software, common software.

1. INTRODUCTION

The Atacama Large Millimeter Array (ALMA) is an international collaboration between Europe, Japan and the North America to build a synthesis radio telescope that will operate at millimeter and submillimeter wavelengths. ALMA will be located in the Atacama desert of northern Chile on the high-altitude (5000m) Zona de Chajnantor, east of the village of San Pedro de Atacama, a site that provides excellent atmospheric transmission in the millimeter and submillimeter wavelength ranges.

The ALMA telescope is composed of two parts, the 12m Array consists of up to 64 meter antennas of 12 meters diameter that can be placed on 216 different stations for baselines up to 18 km and the Atacama Compact Array (ACA) that consists of twelve 7m antennas placed in compact configurations and four 12m antennas for measuring source total power[1]. The receivers will cover the range from 30 to 950 Ghz.

The ALMA Common Software (ACS) is a software infrastructure built on top of CORBA for the development of distributed systems based on the Component/Container paradigm[2]. ACS is widely used in ALMA: it extends from the high level applications down to the control software. From a system perspective, ACS provides the implementation of a coherent set of design patterns and services making the whole ALMA software uniform and maintainable; from the developer point of view, ACS provides a framework that hides the complexity of the CORBA middleware and other

* acaproni@eso.org; phone +49-89-32006629

libraries reducing considerably the programming effort.

ACS is a widely distributed project involving the European Southern Observatory in Garching, the Astronomical Observatory of Trieste (OAT) in Italy, the National Radio Astronomy Observatory (NRAO) in Socorro, USA, and the University of Calgary in Canada.

ACS extensively uses free software to minimize the programming effort and is freely distributed under LGPL license. It is also used by other projects, like the Atacama Pathfinder Experiment (APEX), in science operation, the Sardinia Radio Telescope (SRT), under development, and the ANKA synchrotron in Germany.

Following this philosophy, when we had to implement the alarm system we have looked at the available packages before starting writing our own implementation. We have found that the alarm service, developed for the Large Hadron Collider, at CERN was fulfilling and exceeding our requirements, although it was not available with an open source license. We have therefore decided to get in touch with the CERN staff to see if it was possible to integrate their alarm service into ACS and to make it available as LGPL to our users. The realization of a pilot project to verify the possibility of integrating LASER into ACS has been the beginning of a good and durable collaboration between ACS and CERN.

Telescopes produce one or more stream of data whose validity depends on the proper operation of various hardware and software components. Last generation telescopes have a very big number of sensors used to verify that each component is operating properly. As the number of components in the system grows, some method to collect, store and present abnormal situation in a quick and clear way to the user is needed. Accelerators and the Large Hadron Collider (LHC) in particular, have a huge amount of components and sensors, several order of magnitude bigger than modern telescopes.

2. LASER ALARM SERVICE

The Large Hadron Collider Alarm System (LHC-AS) also known as LASER, is a second generation alarm service developed at CERN in Geneve, that has recently replaced the alarm service built for the Large Electron Positron collider (LEP)[6].

LASER is a messaging system; it collects, stores, manages and distributes information regarding abnormal situations called Fault States (FS). A FS is identified by a triplet: the Fault Family (FF), the Fault Member (FM) and the Fault Code (FC). The FF represents a set of elements with the same kind of problems, like power supplies. The FM specifies the particular instance of an object in the FF, for example a specific power supply. The FC is a code representing a particular problem occurring in the FM.

From an operational point of view, the alarm service receives FS from alarm sources. Each FS is made persistent and correlated with other FS previously received. Each FS is defined in the database together with other information, like for example the exact position of the failing component as well as the name and telephone number of the responsible person for that particular alarm. The alarm system uses the FS as a key to retrieve such information from the database and build a new, more complete snapshot of the specific FS. Finally, the alarm service sends this snapshot to the clients, one of which is the operator GUI that presents the alarms to the operators that can take the more appropriate action to fix the problem.

When the system is complex, the cascade of alarms produced as the consequence of just a single failure can be huge. LASER correlates active alarms in order to show to the operators only the root cause of a specific alarm. We call this phase *reduction* and it is a key process to help users in finding and fixing quickly each problem.

2.1 LASER system overview

The LASER system is a distributed, layered application. Each layer depends on the layer below and provides a set of services for the layer above. The definition of a clear set of interfaces drives this hierarchy.

The software is composed of three tiers: the *resource tier*, composed of a set of sources, detecting and sending FS changes; the *business tier*, that implements the system logic and its services having a knowledge base of the specific domain; the *client tier* composed of clients consuming the business services.

LASER is based on Java 2 Enterprise Edition (J2EE) paradigm: Enterprise Java Bean (EJB), for the component/container model, and the Java Message Service (JMS) for asynchronous communications between components. Persistency is managed by Oracle9i with the support of Hibernate, while SonicQM is used to exchange

JMS enterprise messages between the components. Other packages are used to help reducing the programming effort.

FS between the sources and the service are sent in XML form because the sources can be written in different programming languages and run on different platforms: JMS connectivity has been written in C++ based on the HTTP protocol.

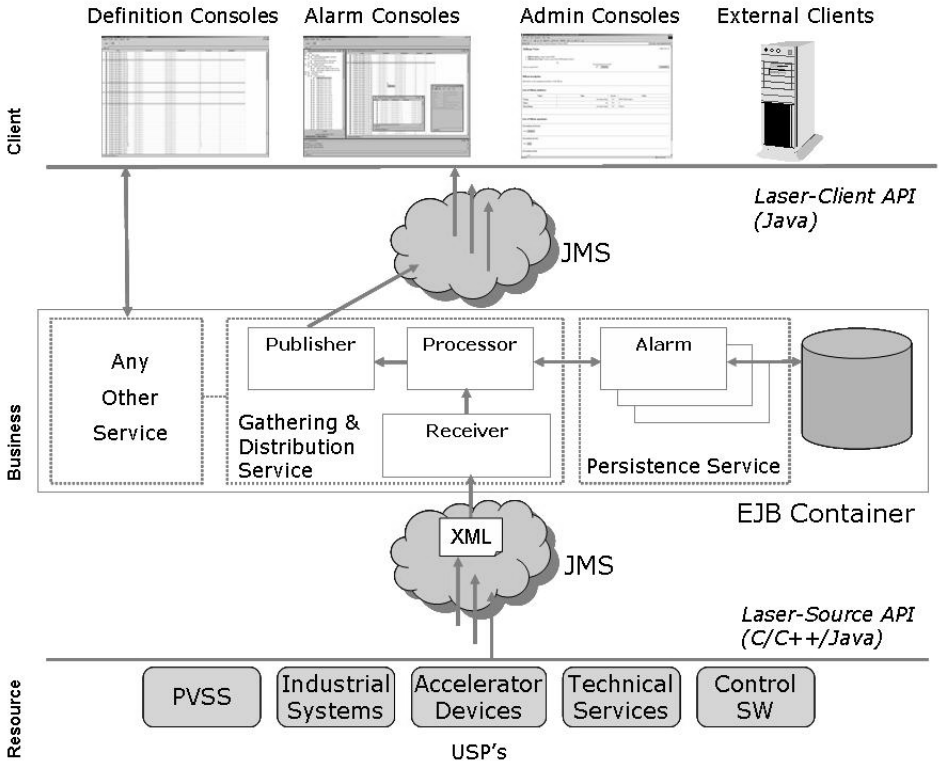

Fig. 1. The structure of LASER.

LASER comes with a Graphical User Interface (GUI) developed on top of Netbeans and recently replaced by a new implementation written in pure SWING. In that interface the alarms are presented to the operator that have to take the appropriate action to solve the problem. The alarm system does not take any action on itself: it is always the operator that fixes the problem and acknowledges the alarm when the problem has been fixed.

2.2 The resource tier

The resource tier is composed of the sources of alarms, i.e. applications that monitor the hardware and the software to detect malfunctioning. Each alarm source has a definite set of FS whose state can change from active to inactive. The sources can be written using different programming languages and run on different platforms.

The *Laser-source API* has been written to connect the sources to the business tier and is very small in order to be as simple as possible for the user. The API is written in java and in C++ and runs in all software environments used at CERN like embedded and real time system, different operating systems or hardware platforms and so on. Table 1 shows the a java example to send an alarm.

The sources build a message containing the FS and an action, like active or terminate. The API embeds the message into a structure and publishes the message in a JMS topic to the business tier.

Each source periodically sends a heartbeat to the alarm service to notify that it is in a healthy state.

Table 1. The java procedure to send an alarm.

```
public void send_alarm(String faultFamily, String faultMember, int faultCode, String faultState)
{
    AlarmSystemInterface alarmSource AlarmSystemInterfaceFactory.createSource(this.name());
    FaultState fs = AlarmSystemInterfaceFactory.createFaultState(faultFamily, faultMember, faultCode);
    fs.setDescriptor(faultState);
    fs.setUserTimestamp(new Timestamp(System.currentTimeMillis()));
    Properties props = new Properties();
    ...
    props.setProperty(...);
    fs.setUserProperties(props);
    alarmSource.push(fs);
}
```

2.3 The business tier

The business tier is the core of the alarm service:

- listens for FS changes and heartbeats from the sources
- reads the further data of a received alarm from the database
- reduces or masks the FS depending on the knowledge of the environment and the current status of the system
- persists the FS
- traces and archives the changes of the FS
- allows management changing and definitions of FS without stopping the alarm service
- authenticate users on the client GUIs

All these services are realized by EBJ and the communications between the upper and the bottom layers happen through a definite API.

In order to maintain easy and short the *Laser-source API*, the sources send to the business layer only the triplet describing an alarm with the time of its creation. For each alarm received, the business tier reads its complete definition from the database in order to present to the operators a complete snapshot of the situation, its possible solution and consequences. Table 2 shows some of the information stored in the database for each alarm.

One of the most relevant parts of the business tier is the reduction of the alarms. In a complex environment where a failure can cause a cascade of secondary alarms, it is very important to show to the operators the root cause of a problem. Operators are also confused when the operators GUI shows a great number of repeated alarms of the same type. Alarm reduction addresses both these problems.

To perform the reduction, the alarm system reads from the database a set of dependency rules between alarms describing their correlation. Whenever the service receives a FS change, it applies that set of rules and eventually marks some alarms as reduced.

All the alarms, both reduced and not reduced, will be sent to the client because some clients can be interested in receiving all the alarms regardless their reduction status: it is the GUI that hides the reduced alarms to the operators depending on the specific configuration.

There are two types of reduction rules:

- *node reduction*: when it is known that a failure in an equipment A triggers a failure also in the equipment B then the latter alarm is reduced, with the effect that only A, the root cause of the FS, is shown;
- *multiplicity reduction*: when there is a great number of alarms of the same type then these alarms are reduced and a new alarm is shown with the effect to reduce the number of alarms shown in the client GUI.

Table 2. The definition of an alarm.

system-name	The name of the system
Identifier	The identifier of the system
problem-description	The description of the problem
instant	Specify if the alarm is instantaneous
cause	The cause of the problem
action	The action the operator must follow to fix the problem
consequence	A description of the consequences of the alarm
priority	The priority of the alarm: there are four priority levels
responsible-id	The ID of the responsible person
piquetGSM	A GSM number to call to notify about the problem
help-url	An url that point to the documentation of the problem
source-name	The name of the source
location	The location of the source
piquetEmail	An email address to send a message about the alarm

2.4 The client tier

The client tier is composed of java applications that consume the data published by the business tier. The client connects to the business tier by means of the *Laser-console API*. The business tier supports both login and configuration facilities.

Once connected, the clients can access services of the business tier by means of the *Laser-client API*. This API allows the clients to access active FS after sending a message to the service with a definition of which kind of messages the client is interested in. At this point the communication between the core service and the client proceeds asynchronously with the alarm service sending the alarms selected in the first message.

Three GUIs developed with Netbeans are part of the client tier: the definition console, the alarm console and the admin console. The definition console and the admin console allow the user to define alarms, sources and categories as well as create accounts and configurations for the operators of the alarm console.

The Alarm console, was initially developed with Netbeans and has been recently replaced by a SWING implementation. It shows the alarms to the operators: when the operator starts the GUI, he has to log into the system then the GUI loads his specific configuration and connects to the alarm service. In the configuration it is possible to select the categories of the alarms to show to each operator showing to him only the alarms relevant for his area of interest. In the configuration is also possible to define if the operator is interested in receiving all the alarms or only the reduced alarms. An apposite configuration editor panel allows the user to change its configuration at run-time.

Fig.2 Shows the Netbeans version of the operator GUI. The main window, in background, shows several alarms; the front panel shows the configuration editor. Alarms with different priorities are shown with different colors.

Whenever the alarm consol receives an alarm, it shows a line in the table with the label N that means “new”. When the operator presses the mouse button over the alarm, the N changes to the date when the alarm was issued by the source. If an active alarm becomes terminate, its entry remains in the main panel until the operator explicitly acknowledges the alarm by adding a comment.

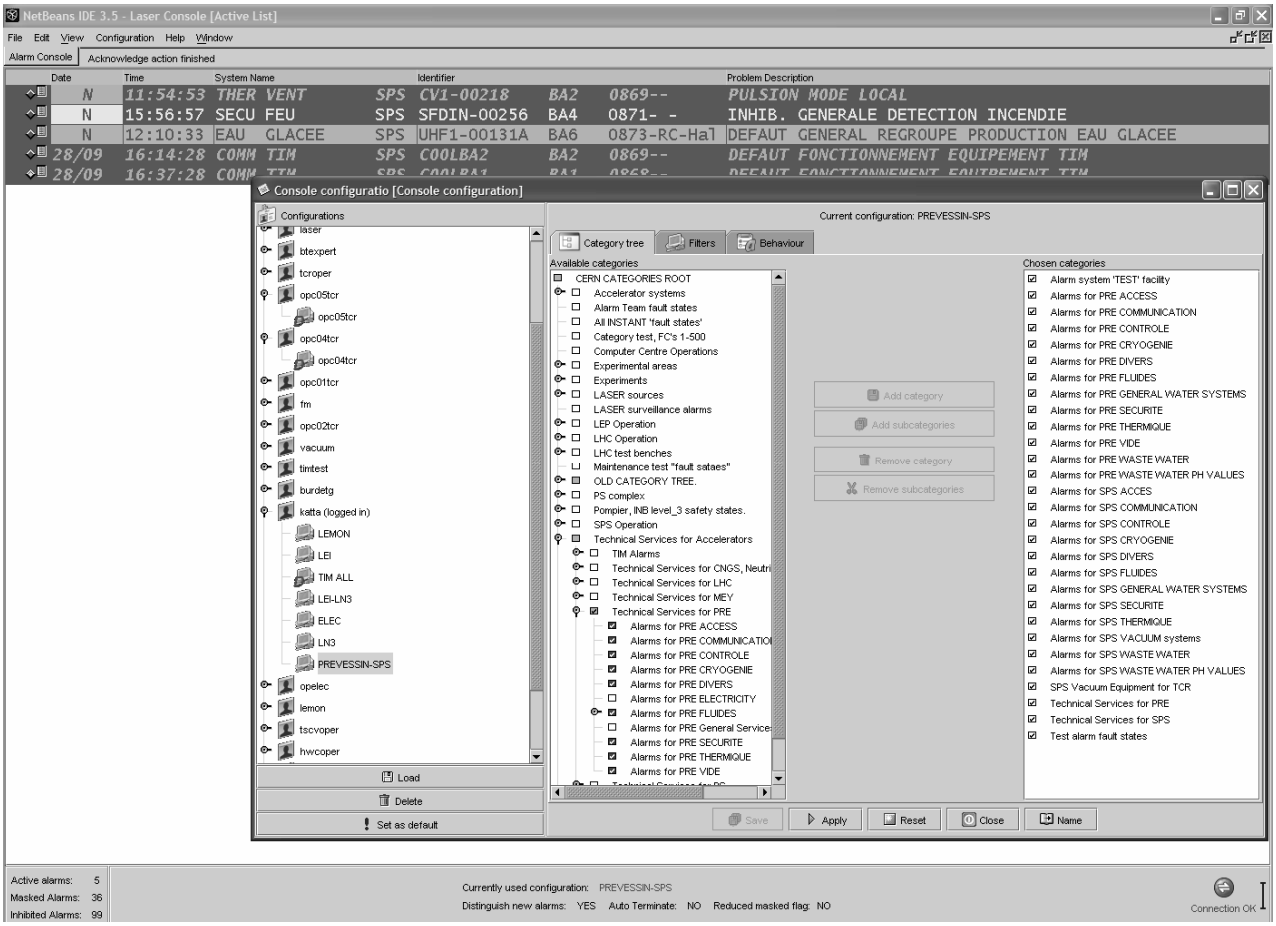

Fig. 2. The alarm console and the user configuration panel.

3. THE INTEGRATION PROCESS

The integration of LASER into ACS started with the definition of a pilot project aiming at understanding if such integration was at all possible. The pilot project culminated with the setup of a test suite showing that several FS sent by different sources arrived to the operator GUI passing through the service and eventually after having been reduced. This test suite was first shown in occasion of the ACS workshop at the ICALEPCS conference, held in Geneva in October 2005.

3.1 Evaluation

When we started evaluating LASER, we found that the two projects have many architectural similarities, but are also very different for the kind of technologies adopted: LASER is based on J2EE together with several commercial tools; ACS is based on CORBA, has its own container/component model and only uses freely available tools.

In this phase we have analyzed the general structure of LASER identifying the compatibility issue of the external packages with ACS. For each of these packages we have evaluated an alternative suitable replacement minimizing the impact on the original code. Table 3 shows a summary of these changes.

LASER is developed in J2EE while ACS has its own component/container model that extends from java to C++ and python, the three languages officially supported by ACS. We prefer to integrate LASER into ACS instead of writing an interface between ACS components and J2EE components. In fact, by examining the alarm service core we have found that the main logic is implemented by Plain Old Java Object (POJO) whose methods are called by J2EE interfaces: it was sufficient to rewrite J2EE interfaces as IDL interfaces and connect the IDL methods to the POJO methods.

Table 3. Comparison of ACS and LASER technologies.

	ACS	**LASER**
Remote invocation	CORBA	Java RMI (via J2EE)
Asynchronous messaging	CORBA Notification Service (via ACS Notification Channel)	SonicMQ (via JMS)
Persistence of configuration	XML (via ACS Configuration Database) and/or ALMA archive	RDBMS (Oracle)
Temporary state storage	In memory Hash table (prototype)	Oracle object cache
Persistent state storage	In-memory transactional database (via Prevayler)	RDBMS (Oracle)
Marshalling/unmarshalling for on-the-wire presentation	XML (via Castor)	XML (via Castor)
Marshalling/unmarshalling for database persistence	StringBuffer/XML DOM	Hibernate
Server container	ACS container	J2EE application server (Oracle Application Server, via Spring Framework)
GUI framework for alarm console	Very likely NetBeans, although ABeans is the usual choice	NetBeans

At the present stage, ACS does not have a commercial database and the storage is implemented as a set of XML files constituting what we call the Configuration Database (CDB). This will change in future but from the time being we had to replace all the Java Database Connectivity (JBDC) calls with CDB calls. To speed up the writing of the prototype, we have decided to store in the database only the static configurations of the fault states and the reduction rules leaving out of the process the FS changes, received at run time from the sources, as well as the users configuration of the GUI.

For the asynchronous messaging, LASER uses SonicQM via JMS. This is a commercial tool: to get rid of this package, we decided therefore to implement a JMS interface for ACS based on CORBA Notification Channel (NC).

Regarding the GUI, ACS has supported in the past Netbeans just as an Integrated Development Environment (IDE) for building GUIs based on abeans[7], a set of GUI widgets and API developed by Cosylab. In recent time ACS has replaced Netbeans with the Eclipse, but still using abeans. For the pilot project we have decided to adopt directly the LASER GUI built using the Netbeans libraries.

3.2 Porting of LASER

We have written a procedure to import each LASER software module into ACS. Even if ACS modules support C/C++ for real time systems, and C++, java and python for not real time environment, the structures of the two modules was surprising similar and the changes were almost limited to the replacement of ant with the ACS makefile.

Analyzing the original code we identified the dependencies between each LASER module and implemented the build procedure for the entire software.

To make the modules compile after the removal of the undesired tools and packages, we have replaced all the relevant methods with stub implementations that only throw an exception. The general idea was to be able to detect the exceptions and replace the empty methods with the proper implementation. Unfortunately that was not the best choice, because the original code catches and masks a great number of exceptions making the debugging effort very complicate and frustrating. A better but less elegant approach would have been that to log a message and exit the process: the missing implementation would have been very easy to identify. The time dependency of the execution of the unimplemented methods would also have been clearer. In fact, it happened sometimes to identify and fix an exception and to discover that this exception was instead triggered by a previous one masked somewhere.

The effort of the developers was, on one side, the replacement of the packages and tools not adopted by ACS as explained upon. On the other side the debugging of the code took a very long time especially at the beginning when the functionality of the original system was almost unknown and a great number of exceptions were masked and re-thrown with different types and messages.

The LASER GUI, instead, ran almost untouched apart some foreseen changes needed to adapt the tool to the ACS environment and to the missing storage of the users' information in the configuration database.

As a conclusion of the pilot project, a test suite has been implemented. Three java ACS components simulated a chain of failures where two FS are all consequence of the failure of the first component: these dependencies have been depicted in the CDB. The test suite showed that the alarm service reads the reduction rules, receives and reduces the alarms and finally sends the alarms to the client GUI. The GUI is able to show the reduced and the not reduced alarm depending on the configuration chosen at startup: if the reduction is active only the root cause of the cascade of the alarms is effectively shown otherwise all the alarms appear to the operator. This was enough to demonstrate that the whole chain from the source to the operator GUI passing through the alarm service is working properly.

3.3 The map between EJB and ACS Components

The usage of POJO to implement the core logic of the system was of great help in understanding the functioning of the system in one side and the porting into ACS on the other: the mapping between EJB interfaces and POJO methods has been replaced by the mapping from IDL interfaces to POJO methods.

Each EJB is defined by an home interface, a remote interface and the actual business logic; the deployment is described by an XML file[9]. The remote interface is a java interface containing the definitions of the methods whose implementation is written in the business logic. The business logic contains the methods in the remote interface but does not implement the interface.

The methods in the home interface depend on the type of EJB, entity or session bean, and represent its life cycle methods. The methods defined in this interface are implemented in the business logic that again does not implement this interface but instead uses a well defined name pattern. These methods implement the life cycle for each particular kind of EJB.

The implementation of the methods defined in the two interfaces is contained in the business logic class. Each of these classes owns a POJO and each method calls another method, with the same name, in the POJO.

Each EJB becomes an ACS java Component described by an IDL interface whose method signatures are obtained translating the signature of the methods from the remote interface. We had to define a data structure in IDL for each particular type of java structure that appears in the EJB interface as method parameter or return value.

The link with the LASER code is done by instantiating in the ACS component the same POJO owned by the EJB and executing the POJO method every time an IDL method is called.

The EJB life cycle methods have been replaced by ACS life cycle methods.

The deployment of the EJB is replaced by the deployment of an ACS component, described by an entry in the CDB.

The procedure described above can be summarized by the following steps, where we have omitted all the CORBA and ACS specific steps:

1. identify each EJB to create an ACS component with its IDL file to describe the interface and its entry in the CDB to describe the deployment
2. build an IDL structure for each parameter and return type present in the EJB remote interface of the EJB
3. scan the EJB remote interface and translate each method signature in the equivalent IDL
4. instantiate in the ACS component the same POJO instantiated in the business logic of the EJB
5. call in the implementation of the methods of the ACS component the method with the same name in the POJO

By looking at this description, it is clear that the definition of clear interfaces between the three tiers as well as between the different components of the LASER system was of great help in the whole process.

3.4 Communication between the tiers

LASER manages the communications between the tiers with an implementation of JMS based on the HTTP protocol. The persistency of the messages is performed by a Sonic tool, a commercial package that can't be part of the ACS distribution. We have therefore decided to implement the JMS specifications in ACS by means of the CORBA Notification Channel. The reason for this choice is not only because NCs are widely used in ACS, as discussed in other papers in this conference[3]. but, as we will see later, because NCs are very convenient for this purpose.

In the ACS implementation of JMS it is possible to publish two kinds of messages: text and object messages. Object messages contain a serializable java Object; text messages encapsulate a String. In the Alarm System we have extensively used the text version.

Whenever a source sends an alarm to the alarm service, the alarm is translated into its XML representation using castor[10]. The XML string is embedded into the JMS text message and finally published in NC. Whenever an alarm is received by the alarm service, it rebuilds the alarm object from the XML string contained in the message, again by using castor. At this level, the XML representing the alarm is the same string sen in the LASER implementation.

The same procedure is followed when an alarm is sent from the Alarm Service to the GUI, i.e. from the business tier to the client tier but the objects translated into XML strins are different. LASER code does not send XML string at this level.

The ACS implementation of JMS specifications is independent of the LASER code. It is instead very generic because each message can encapsulate a serializable java Object or a String. In the case of the Alarm System the String is the XML representation of an object but it can be another entity depending on the intercommunicating applications. Another advantage is that the developer that uses this implementation is able to send messages through a NC without dealing with CORBA stuff.

By implementing the JMS specifications, the changes in the original code were limited to the instantiations of the proper objects but we maintained the same software structure of LASER and the same topic used in LASER.

As we said above, LASER is basically a messaging system that collects, manages and redistributes alarms to the clients. It does much more than that of course, but what the alarm system never does is taking an action to fix a problem: it is always the operator that interprets the alarm in the GUI and takes the appropriate action. The only things the AS can do on its own, is to send notifications, like for example emails, when receives an alarm.

The ACS porting of LASER allows going further in this direction. NCs have the feature that more clients can listen at the same time to the massages passing through the wires. This allows the developer to write clients that listen to the NC and takes some actions when detect one specific alarm, or an alarm of a particular class. This action can be taken close to the sources if the client listens at the communication between the source tier and the business tier. Or it can be taken at a higher level monitoring the messages from the business tier to the clients. Having this opportunity was a major reason to implement JMS with CORBA NC.

3.5 Considerations

The pilot project shows that it is possible to inject a third party system into ACS without introducing big changes in any of the two packages. We analyzed the amount of changes we introduced in the original code with the help of a python utility by comparing the original LASER code with the modified version running in ACS.

The number of modules remained almost the same apart of the implementation of JMS in ACS. The number of original java files changed is less the 10% of the entire original code while the number of java files in the version running on ACS results about 10% greater then the original version.

The first result was obtained because, even if the two systems are very different, their structure is very similar.

The difference in the number of lines of code between the two systems is mostly due to the implementation of the JMS specifications for ACS and the storage with CDB to replace JDBC and Oracle.

The preliminary study of the original code was very important to isolate potential problems and to find a replacement strategy for critical packages and tools. Unfortunately the choice of using exceptions made the life of the debugger more complicated. Having time to port and run the LASER tests in ACS would have been very useful in isolating and fixing

bugs before testing the entire system.

The pilot project was driven by time constraints more then by establishing a stable cooperation between ACS and the LASER team. The immediate consequence is that the changes needed to make the original software running properly were made directly in the original code. Thereafter the pilot project represents the base for a future collaboration and helped a lot in identifying critical areas as well as part of the code that remained mostly unchanged in the process.

The collaboration between CERN and ACS culminated with an agreement for future development of LASER. A version of LASER modified to run together with ACS will be released with LGPL license in a future release of ACS.

The pilot project demonstrated that it is possible to adopt LASER in ACS having the core of the original software running almost unchanged. It also helped in finding similarities and differences between the two projects in such a way the most critical parts of the code have been identified.

To make this collaboration between LASER and ACS really effective we need to decouple the original LASER code from its ACS version: the core will be almost the same in the two versions, and a set of interfaces must be defined for this purpose.

We will work to refactor the original LASER code in order to isolate the portions of the code in common between ACS and LASER. The glue between the two systems will be a well defined set of interfaces and the effort of the two teams will be focused on the shared part of the code leaving out of the process the details specific to each subsystem. In this way each change, bug fix or improvement introduced to the code by one of the two teams will be immediately and transparently available to the other.

A procedure to release LASER to our users will be found. LASER is archived in CVS at CERN and accessible only by authorized personnel. We will produce a distribution of the alarm system by providing a snapshot of LASER into ACS CVS. This snapshot will contain the sources of LASER and possibly some patches that might be needed but the majority of the code will remain untouched.

4. CONCLUSIONS

We have imported the LASER code into ACS and written a prototype to show that it was working as expected. Changes have been done mainly in well defined interface classes for a total of less then 10% of the original LASER code. We have replaced the commercial tools used by LASER and J2EE beans with the equivalent ACS tools. The core elements of LASER run mostly unchanged.

The preliminary study of the system helped in solving a great majority of the incompatibilities between the two packages. On the other hand, some strategy adopted while making the code compile and the lack of time to run the LASER tests in ACS slow down the project.

The pilot project demonstrates that independent services can be integrated and used in heterogeneous software infrastructures if the logical architecture is sufficiently similar and if attention is given to isolate the interfaces toward the infrastructure itself. It also highlighted the portions of code that need major attention with respect to those that remained untouched.

The CERN LASER team and ACS have subscribed an agreement for developing together the alarm system in future and as an immediate consequence LASER will be distributed in a future release of ACS under the LGPL license.

A refactoring of the original code is foreseen to decouple the two packages in such a way it will be easy and fast to import changes and upgrades in the two systems in both directions.

REFERENCES

1. T.L. Wilson, A.J. Beasley, H.A. Wootten, *Status of the Atacama Large Millimiter Array*, http://www.eso.org/projects/alma/publications/
2. G. Chiozzi et al., *The ALMA common software: a developer friendly CORBA-based framework*, Proc. SPIE Vol.5496-23, Astronomical Telescopes and Instrumentation, Glasgow, June 2004.
3. G. Chiozzi et al., *Application development using the ALMA common software*, these proceedings.

4. F. Calderini, B. Pawloski, N. Stapley, *Moving towards a common alarm service for the LHC era*, Proc. ICALEPCS, Gyeongju, Korea, October 2003.
5. R.P. Finch, S.L. Scott, *Fault detection and handling in the Caltech Millimiter Array*, Pubblication of the Astronomical Society of the Pacific, 108, August 1996.
6. Alarm team, *LHC Alarm System, User requirements document*, European Laboratory for particle physics, CERN, Geneve, 2001.
7. GNU Lesser General Public License, http://www.fsf.org/licensing/licenses/lgpl.html
8. Abeans,http://abeans.cosylab.com
9. Enterprise Java Beans, http://java.sun.com/products/ejb/
10. The Castor project, http://www.castor.org/