An Introduction to Scalable Frameworks for Observatory Software Infrastructure

-

SC-644

G. Chiozzi – European Southern Observatory
gchiozzi@eso.org

SPIE 2006
Astronomical Telescopes and Instrumentation
May 29, 2006

Course Abstract

This course provides an analysis of the advantages and requirements for an integrated software infrastructure for observatories and similar scientific facilities. It provides a common framework for application software that can range from control to data analysis applications. Currently available and emerging technologies are evaluated and compared. The course concentrates on the architecture of an application framework necessary for such an infrastructure and on the impact on scalability, maintainability and reuse. Many practical examples will be given based on the ALMA Common Software, a CORBA-based, open source solution used by ALMA and other projects.

Learning outcomes:
This course will enable you to:

•identify the advantages and requirements of an observatory-level software infrastructure

•compare existing and emerging technologies

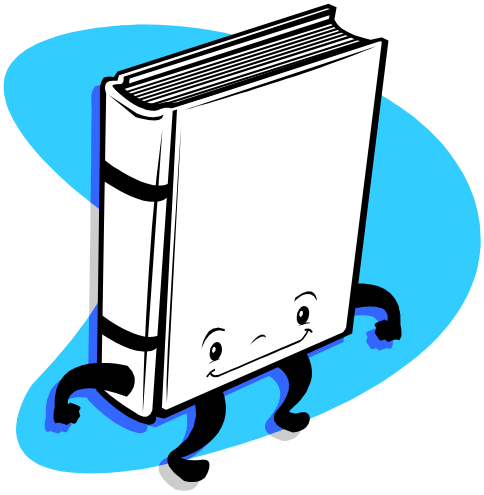
•estimate the impact of introducing a common software framework in a new or pre-existing project

•demonstrate applications implemented using the concepts described in the course

Intended audience:
This material is intended for anyone who is involved in the design and refurbishment of the software architecture of a scientific facility and in the selection of the middle-ware architecture to use. Those who develop applications integrated in the data flow and control infrastructure of an observatory will find this course valuable.

Instructor:
Gianluca Chiozzi currently works at the European Southern Observatory in Munich. For the last 10 years he has been heavily involved in the design and implementation of the VLT Common Software and Telescope Control Software. He is now responsible for the ALMA Common Software architecture and development. Before ESO he worked at the IBM Technical and Scientific Research Center in Milan.

1 - Introduction and Overview



Course Objectives

- Identify advantages and requirements of an observatory-level software infrastructure
- Compare technologies
- Estimate impact of a common software framework in a new or pre-existing project
- Demonstrate applications

This course provides an analysis of the advantages and requirements for an integrated software infrastructure for observatories and similar scientific facilities. It provides a common framework for application software that can range from control to data analysis applications. Currently available and emerging technologies are evaluated and compared. The course concentrates on the architecture of an application framework necessary for such an infrastructure and on the impact on scalability, maintainability and reuse. Many practical examples will be given based on the ALMA Common Software, a CORBA-based, open source solution used by ALMA and other projects.

Learning outcomes:
This course will enable you to:

•identify the advantages and requirements of an observatory-level software infrastructure

•compare existing and emerging technologies

•estimate the impact of introducing a common software framework in a new or pre-existing project

•demonstrate applications implemented using the concepts described in the course

I think it is important to adapt the course to the audience and follow up the questions.

Therefore these course note masters are not cast in the stone and, if necessary and useful, we can decide to go sometimes into deeper details or to skip parts that do not seem particularly interesting for the participants.

By experience, each course is different because of the different mix of participant's knowledge and experience.

Contents

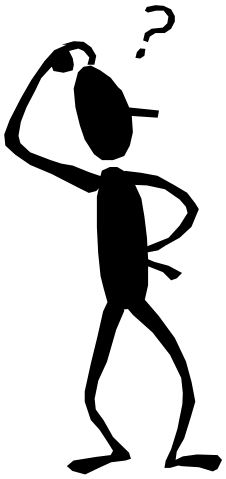
We will divide the course in 10 sections.

At the end of each section there will be time for questions, discussions and, in case, some extra detail.

You can interrupt me at any time for questions.

I will decide case by case if I have to reply immediately to the questions or if it is more efficient to reply later or to let the reply come out by itself from other parts of the presentation.

Let's introduce ourselves



Let's go around the classroom and introduce briefly ourselves:

•Who are we?

•What do we expect from this course?

•Any suggestions?

I am Gianluca Chiozzi and I am working at the European Southern Observatory in Munich.

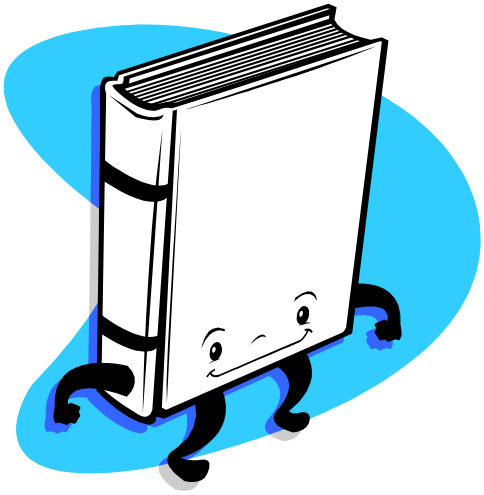
You can reach me at any time at the following email: gchiozzi@eso.org

I am now responsible for the ALMA Common Software (ACS) architecture and development, with a team of about 10 people (not all full time) distributed in various sites in Europe and North America. ACS is the software infrastructure for the ALMA project and is used also by other projects. I am working on this since about 6 years.

Before ALMA and ACS I have been heavily involved for another 6 years in the design and implementation of the VLT Common Software and Telescope Control Software. Here I have been responsible for introducing Object Oriented technology in the project, working on the architecture and design of some control subsystems and on the implementation of OO class libraries for the Common Software infrastructure.

Before ESO I have been employed at the IBM Technical and Scientific Research Center in Milan, working on image recognition systems and on user interfaces for utility management systems (like electrical or railways networks).

2 – Observatory Architecture



The “traditional” Observatory

In the traditional observatory:

- Parts/subsystems were independent, not integrated
- Complexity was low and there were no requirements of integration
- Astronomer was making observations and taking home the data

A modern Observatory (or any other experimental facility) has a complex integrated and distributed architecture.

In the past, the Astronomer (the main stakeholder for our systems) was traveling to the Observatory, make his observations, store data on tape and go back home to reduce it.

The telescope and, eventually, the instruments had a control system virtually independent from everything else.

Data reduction was done offline after the observation and there was no direct feedback from the observation data to the control system. An experienced observer was just driving the telescope based on his own feelings.

There was no observation data archive, nt quality of service measures and constraints, no facility engineering in the terms we think of now.

This has dramatically changed in the past 15 years, with the big observatories like VLT, Keck, Gemini, Hubble and so on.

Since the major observatories are now providing integrated facilities, astronomers expect the same also from smaller ones and the amount of integration required for the new projects like ALMA and the giant optical telescopes will be even more.

The Virtual Observatory is also contributing to this need for integration and quality control adding the intra-observatory dimension to the problem.

Modern Observatory architecture

- Integration of all parts (end to end data flow)
- Archive and virtual observatory
- Feedback

- Compare major projects and small projects.

Now all the systems in an Observatory are fully integrated.

Observation data, weather and telemetry are directly fed back in the control system to obtain optimized performance.

The astronomer is in many cases not even any more going to the observatory, but monitors the observation from his own institute and can ideally interact with the system remotely when the observation is taking place.

The astronomer expects to interact with the system using “standard” Web technology.

Data is archived to be usable by Virtual Observatories and therefore has to contain calibration and quality information.

Engineers and people responsible for the administration and maintenance of the observatory are another important stakeholder.

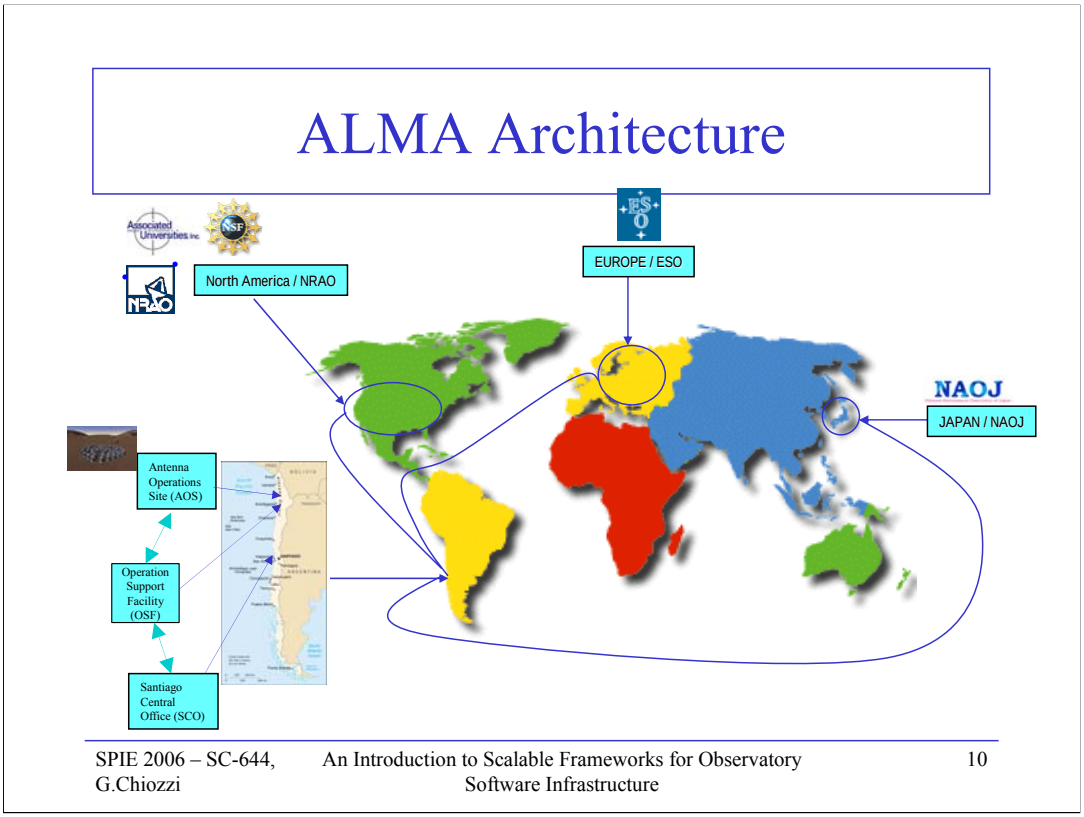
More and more performance is measured and telemetry information is analyzed daily to perform preventive maintenance and optimize the system or to measure performance trends over long time periods.

On the same path: common technical trends

- Common architectural concepts and terminology across many telescope projects
- Less reliance on 'home-grown' software, particularly in infrastructure
- Common software across entire observatories
- More distributed control using cheaper hardware: PCs dominate
- Real Time Linux or VxWorks
- More Java, less C/C++

Thanks to the open communication and interchanges between the major projects (for example collaborations and cross-participation to reviews), we are moving toward common concepts and a common terminology. The architecture of the various projects looks more and more similar and consistent. When we talk together we can easily understand each other and we can share design patterns and architectural solutions, if not actual code.

This is potentially a big advantage also for small projects, that can reuse this common architecture and solutions.



We can take as an example the Architecture of ALMA (but we could take the VLT, Hubble or any other major observatory).

We will start analyzing the overall architecture of ALMA and see how it impacts the software architecture.

First of all you can notice that ALMA will be a very geographically distributed system:

- AOS: Chajnantor (5000m)
  - Move antennas (scattered up to 14 km from correlator)
  - Swap in/out hardware modules
- OSF San Pedro (2800m, ~30 km from array)
  - Control & monitor array
  - Repair Hardware
- SOC: Santiago
  - Run pipeline
  - Maintain archive
- Regional Centers: USA, Europe, Japan
  - Accept proposals
  - Deliver data packages
  - Provide User support
- Astronomer's institutes:
  - Submit proposals
  - Monitor and interact with observing projects
  - Data reduction

Observatory Software Scope

- Proposal preparation and review
- Scheduling of observing programs
- Observation
- Calibration and Imaging
- Data delivery and archiving
- Data reduction
- Archival Research and Virtual Observatory compliance

Astronomers expect from the software in an Observatory a wide scope of services.

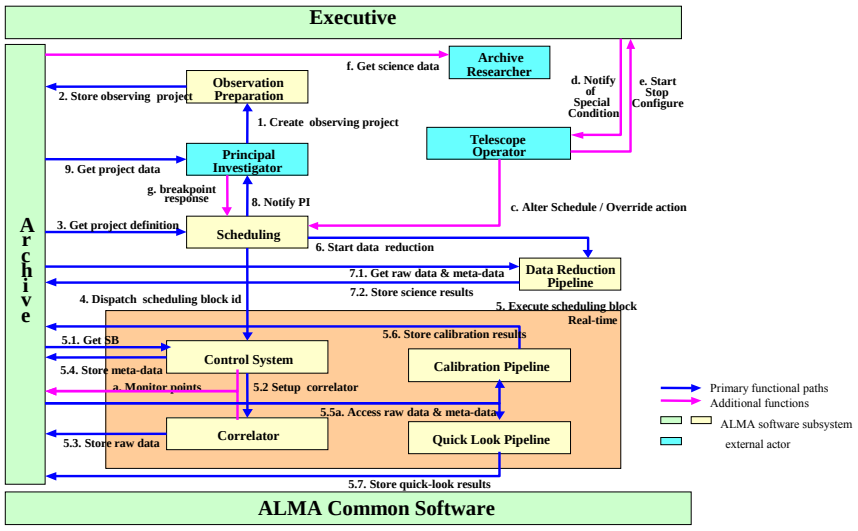
They need to be all integrated together.

Also, astronomical observation is not limited to experts in the field (like infrared or radio astronomers), but it shall be open to chemists, biologists and other multi disciplinary researchers.

The general user should be given standard observing modes to achieve project goals expressed in terms of science parameters, rather than technical quantities. But experts must be able to exercise at the same time full control.

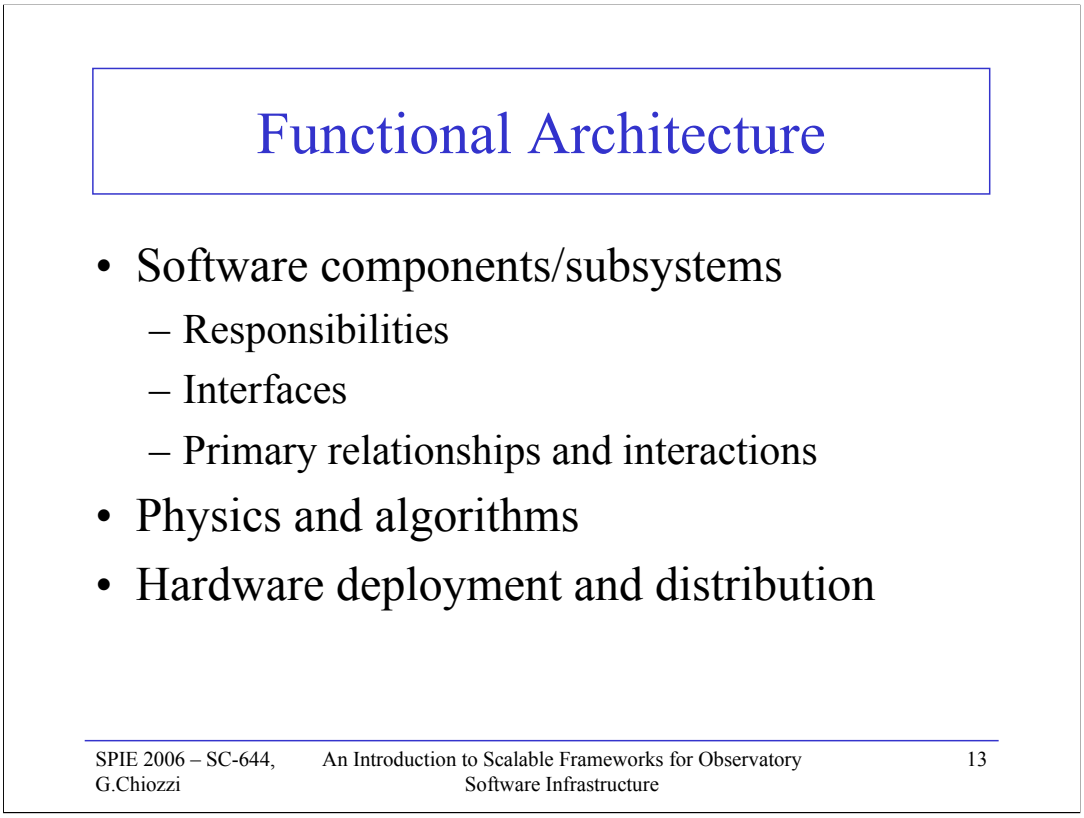
Making things easy and flexible for the astronomer adds up complexity to the software development

ALMA Software Architecture



Again we can take as an example a schematic architecture of the ALMA software (A.Farris, J.Schwarz ALMA HLA team).

This diagram shows the main subsystems in which the software has been divided and the main relationships among then in the form of a collaboration diagram describing the typical lifecycle of a project, starting with proposal submission and ending with a researcher looking for the data in the archive after the observation has been performed.



The previous diagram shows the "Functional Architecture" of the system.

The functional architecture is built based on the user requirements.

The functionality that needs to be implemented is assigned to components/subsystems and the architecture describes the responsibilities of each subsystem and the interfaces that are exposed to the other subsystems or to the external world.

Then the relationships between the subsystems (i.e. how these interfaces are used when asking reciprocally services) are described.

The functionality must be implemented according to the physics of the system and must implement specific algorithms that must be described in this architecture. For example scheduling algorithms, control algorithms, data reduction strategies are all part of the functional architecture.

Another essential driving factor is the actual deployment and distribution of the hardware that must be controlled by the software. For example, the physical deployment of motors and sensors and the physical connection of the electronics to the control computers affects the functional architecture of the system. Or the location of the data archives and of the CPU factories for data reduction.

Technical Architecture

- Communication mechanisms and networking
- Database technology
- Software deployment and activation
- Programming model

The “functional architecture” must be supported by a “technical architecture” that describes (and implements) the technical aspects of the software, like the communication protocols used, the threading model, the software deployment (process handling, distribution, activation and deactivation).

The requirements for the technical architecture are mostly derived requirements.

While the user requirements are the basis for the development of the functional architecture, we derive most of the technical requirements from the functional architecture itself: the technical architecture shall enable us to implement the functional architecture.

Separation of concerns

- Functional and technical architecture: two views
- Subsystem teams should concentrate on function
- Technical architecture provides them with simple and standard ways to, for example:
  - Access remote resources
  - Store and retrieve data
  - Manage security needs
- We want to keep the two concerns separate

Functional and technical architecture are two different views of the system.

Subsystem developers should concentrate on the functional aspects of the system, i.e. in the implementation of the physics and algorithms.

They have to be freed from the need of designing and implementing mechanisms for interfacing, communicating, deploying or handling security.

The detailed design of the Technical architecture must be mastered by infrastructure developers.

Application developers are required to understand just the *concepts* of both the technical and functional architecture.

Infrastructure Framework

An infrastructure framework is the key to this separation

- Programming model
- Satisfy performance, reliability and security requirements

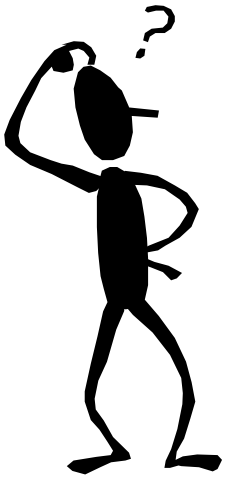
The key to reach this objective is to adopt a Software Framework that provides a consistent infrastructure for the whole observatory.

On one side the framework has to satisfy all the requirements of performance, reliability and security derived from the functional architecture.

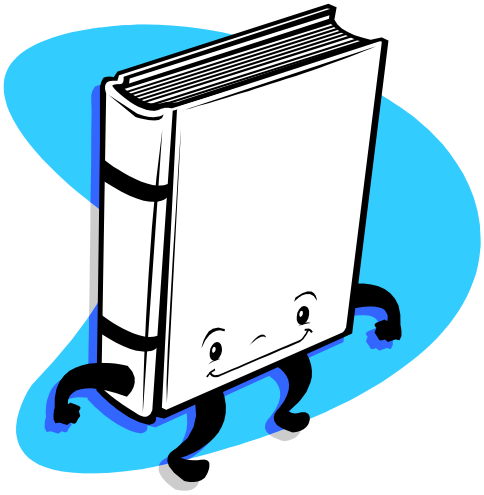
On the other side it must hide as much as possible its own internal complexity to the subsystem developers and provide them with a clear and streamlined programming model.

We will now begin to examine the requirements for such a framework in our domain and what options are available.

Questions (& Answers)



3 – Distributed systems and Middleware



Distributed System

- Requirement: the observatory is a distributed system.
  Servers and clients are distributed on different machines:
  - Possibly in different locations
  - With different purpose and functionality
  - With different requirements on performance and reliability

This actually implies an…..

As can be seen from the previous discussion, the architecture of the observatory is very distributed.

Servers and clients need to be distributed in different locations inside and outside the physical observatory where the telescope reside.

The different parts of the system (that we did not better specify yet) have different purpose and functionality and therefore have different requirements on performance and reliability.

If we take into account that parts of the system are dedicated to real time control of hardware, coordination, database management, data analysis up to the GUIs on the astronomer's desktop, we see that this distribution involves something more than a plain Distributed System.

Heterogeneous Distributed System

- Requirement: the observatory shall be an *heterogeneous* distributed system; servers and clients may use different:
  - Hardware
  - System software
  - Programming languages

What we really have is an *Heterogeneous* Distributed Systems, since the distribution involves different:

•Hardware platforms and architectures. From real time computers to PCs of any kind on the desktops, we can have very different hardware architecture (CPU, word size, alignment, memory available…)

•System software. Any of these machines can have a real time operating system, Linux or other variants of Unix, Microsoft operating systems, MacOS or even more exotic software platforms like PalmOS

•Programming Languages. Different programming languages are more suitable for different application domains. For example, C and C++ are most suitable for real time and CPU intensive applications, while Java fits well in coordination, high level or GUI developments. Astronomers will want to write their observation scripts and reduction procedures in high level scripting languages.

Transparent Heterogeneous Distribution

- Separation of Concerns:
  - Developers of clients and servers have to be unaware of the respective architecture
  - It shall be possible to change the architecture of a server transparently to the client
  - Ideally the developer of a client should not even know if a server is local or remote.

In order to achieve the "separation of concerns" objective, applications developers have to be unaware of the architecture (hardware, software, programming language, location) of the servers they interact with.

Having to deal explicitly with network communication protocols, byte order of message data, connection reliability and similar problem would be a major burden on the shoulders of the application developer.

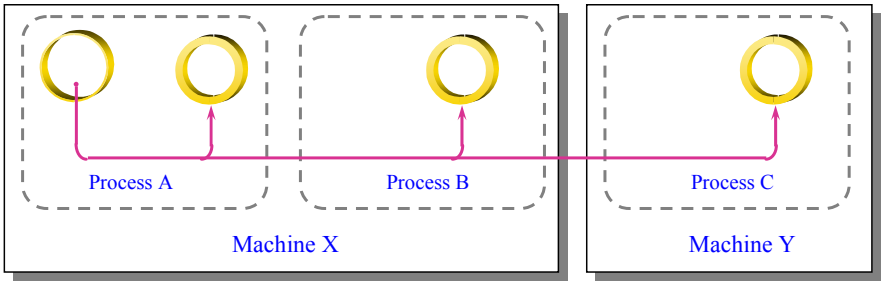
The technical framework has to take up this responsibility and hide all these problems to the functional developers.

It shall even be possible to fully replace the server with a different one without the client noticing.

We could (and this has been often the case in past projects) keep the heterogeneous domains separate. For example data analysis and control system could be implemented using different and independent software infrastructures, but this approach will lead to many problems in the interfaces. In the past, interfaces were limited and this was not an important issue. But the level of integration needed nowadays makes such a choice highly problematic.

The infrastructure Framework has to take care of these aspects of the system.

Location Transparency



Our framework should provide (re-)location transparency:
• an object can be local to your process, in another process on the same machine, or in another process on another machine
• but a client should not need to be aware of the real location of the server.

We keep our architecture easier and much more flexible if we can rely on a middleware that provides location and relocation transparency.

The developer of a client application should interact with a server as if it was a local object, not even knowing if it is local or remote. Remote function calls should resemble local function calls.

Using directly low-level network protocols from the application layer (for example using send and receive on socket based communication) does not allow to reach these objective, because the application software has to be fully aware of the network protocols and communication. This code is typically non highly scalable and hard to maintain and change.

This transparency makes it much easier to scale systems and optimise performance by re-deploying Servants on separate processes and hosts or repackaging together Clients and Servants that have frequent interactions.

Clearly there can be location constraints. For example two objects might HAVE to be colocated in the same process or host because they might need to access directly the same resources or for performance reasons. An example is a data reduction pipeline where two pipeline stages need access to a shared file with the best reachable performance. But in most case a proper design of the system's architecture allows to avoid or reduce such dependencies.

Middleware technology

Middleware is a software layer between the application layer and the network services and provides:

- Heterogeneous communication transparency
- Location transparency
- Message delivery and format integrity
- Dynamic invocation of server processes
- Load balancing
- Security

A special layer of software between client and server processes is needed to deliver the extra functionality. This software layer is called **middleware**. It hides the complexity of the extra functionality behind a common set of APIs that client and server processes can invoke.

Network protocol functionality only allows data exchange between client and server. More functionality is required for heterogeneous distributed systems, like:

•Location transparency
The application does not want to know the network address of the client it wants to use. A location (naming) service should allow to locate a service over the network by name and/or required functionality.

•Message delivery and format integrity
The system must warranty that messages are not lost or duplicated and that they are delivered uncorrupted.

•Dynamic invocation of server processes,
The client shall not be responsible for starting up the services it need, but the system shall be able to do it transparently

•Load balancing
If needed, the system shall be able to redistribute the services to allow load balancing over the distributed servers

•Security
If needed, the system must be able to handle security of communication using appropriate secure protocols. But without enforcing heavy communication protocols where there is no need and/or performance would be an issue.

Object Middleware

- Most used development technologies and programming languages are Object Oriented:
  - Implement objects
  - Call operations of objects
- Call local or remote objects transparently:
  - Client and object vs. client and server
- Object Reference:
  - first get one,
  - then use it.

Object Oriented concepts are pervasive in current software development: essentially all development methodologies and programming languages used nowadays are to some extent Object Oriented. Even scripting languages like Python provide support for Object Orientation.

Developers are used to think in terms of objects and object oriented programs are based on one side on the implementation of objects (define a class and implement it) and on the other side on clients invoking methods provided by instances of objects.

It is very natural to extend this concept outside the boundaries of a programming language or of a process on one host.

Object Middleware allows to call operations of objects that reside on other systems and are possibly implemented in other languages.

The objective is to make as transparent as possible to the developer calling a local or a remote object.

We replace the client/server model with a client/object model.

In order to be able to access and object, you first must get an “object reference” pointing to it. Calling a local object through a pointer or a remote object through a reference are made to look the same. Task of the Object Middleware is to provide mechanisms to implement and retrieve the references for remote objects and to locate the objects over the distributed system.

We will see how this can be done in the examples.

Object Interface and OO concepts

- Interfaces: strong emphasis
  - Clients only look at interfaces
  - Interface Definition Language
  - Decoupling from implementation
- Encapsulation, inheritance and polymorphism

A client is only concerned with the "interface" of the objects it interacts with.

Object Middleware puts a strong emphasis on the concept of "interface". Typically a language neutral Interface Definition Language is used to define interfaces and a strong decoupling between interfaces and implementation allows:

- One object to support many interfaces
- One interface to be implemented by many objects in different ways

The concept of interface provides a strong support for encapsulation.

Independent inheritance is typically supported on the side of interface definition and object implementation.

Polymorphism comes naturally from the separation between interfaces and implementation.

Actually, thinking in terms of separation between interfaces and implementation helps a lot in grasping the fundamental OO concepts. This is a major advantage of Java with respect to C++ from the language definition point of view (pure virtual declarations in C++ can be used to emulate interfaces, but are not conceptually equivalent).

Middleware: Buy, not build!

- Avoids need to write communication infrastructure in-house
  - Less effort
  - Less in-house expertise required
  - Access to outside expertise
  - Benefit from wide-spread use
- Often provides rich set of features
- Lots to choose from (both commercial and public domain)

The software infrastructure for an observatory shall be built on top of an Object Middleware.

But there is no point in developing a new one: there are various available on the market and it is just a matter of picking the right one.

Object Middleware Systems

- Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG)
- Internet Communications Engine (Ice) from ZeroC
- Enterprise Java Beans (EJB) and Java Remote Method Invocation (Java RMI) from Sun Microsystems
- .NET and Distributed Component Object Model (DCOM) from Microsoft

Four dominant examples of Object Middleware Systems are:
•CORBA: http://www.corba.org/
•ICE: http://www.zeroc.com/ice.html
•Enterprise Java Beans (EJB) and Java RMI: http://java.sun.com/products/jdk/rmi/
•.NET and DCOM: http://www.microsoft.com/com/default.mspx

We will base our examples on CORBA.

There are many parameters to drive the choice, depending on the requirements of the system under development:
•Open or closed standard
•Opens source versus commercial implementation
•Multiple vendors
•Market share
•Costs
•Number of architectures, operating systems, languages supported.

For ALMA we have decided to adopt CORBA because we think its characteristics make it the most suitable for the development of a software system for a large international and open collaboration in the scientific community:
•Very open standard, not controlled by specific vendors
•Wide availability of high quality open source implementations
•Intrinsically operating system, architecture and language independent.
•Vendor interoperable by design, i.e. applications from different vendors will work together

We are happy of this choice and we can state that CORBA maintains what promises.

What is CORBA

- CORBA is a standard, not a product
- Common Object Request Broker Architecture (CORBA)
  - A family of specifications
  - OMG is the standards body
  - Over 800 companies
- CORBA defines *interfaces*, not *implementations*
- *CORBA core and services*

First of all we have to say what CORBA isn't: CORBA is not a product, but rather a standard specification. The OMG is the standard body for the specification of CORBA (and of UML) and is a consortium of the most important software vendors.

For this reason CORBA specifications define only interfaces and not implementation.

Any vendor should be able to provide an implementation based only on these OMG interface specifications and this implementation should be interoperable and usable together with the implementation of any other vendor, provided that both comply with the same specification level.

Reaching such an interoperability requires unambiguous and complete specifications. This typically requires various iterations. After some initial glitches (like the design of the Basic Object Adapter – BOA – later on superseded by the Portable Object Adapter) the core components of CORBA are now extremely stable and interoperable.

The latest additions and the higher level services still need some iterations to reach the same level of stability and interoperability.

In ALMA and the Alma Common Software we are currently using 5 different open source CORBA implementations:
•TAO for C++
•JacORB for java
•Omni ORB for Python
•Mico for its implementation of the Interface Repository service
•Open ORB for its implementation of an extensible IDL compiler used for code generation

We have also used in the past Orbacus and the native Java JDK ORB and replaced them with one of the ORBs previously listed and we have switched the implementation of some services from one ORB to another, to use the one better fulfilling our needs. And we can use the documentation, manuals, books relative to any of them to learn how to use best another one. This is a very good demonstration of interoperability.

When we discuss of CORBA, we should always keep in mind two levels:

•CORBA core functionality is the set of basic components (like the IDL language, the Object Request Broker and the IIOP communication protocol) that warranty interoperability. Every implementation has to support them.

•CORBA services are additional (but often fundamental) services that are built on top of CORBA by the vendors that want to provide them.

Summary: requirements and core CORBA

- No need to pre-determine:
  - The programming language
  - The hardware platform
  - The operating system
  - *The specific object request broker*
  - The degree of object distribution
- *Open Architecture*:
  - *Language-neutral* Interface Definition Language (IDL)
  - Language, platform and location transparent
  - *Interoperable*
- *Objects could act as clients, servers or both*
- *The CORBA infrastructure (the ORB) mediates the interaction between client and object*
- *Scalability designed in CORBA specifications*

Here I summarize once more the main design goals and characteristics of CORBA.

Many of these items have been listed already as common to all major object middleware technologies, but I have marked in colour (italics, for B&W print) the ones that are more specific of CORBA and that many "competitor" technologies do not take into account.

These goals map well with the requirements that have led us to identify the need for adopting a Middleware.

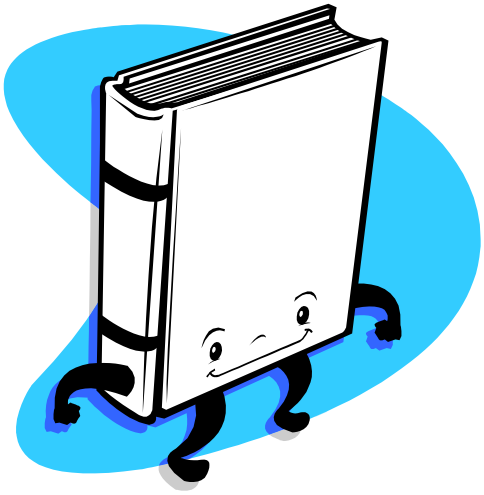
CORBA is designed for scalability and analyzing the architecture of the basic CORBA infrastructure allows to understand how this scalability can be reached.

See the slides in the "additional information" section for details.

Questions (& Answers)

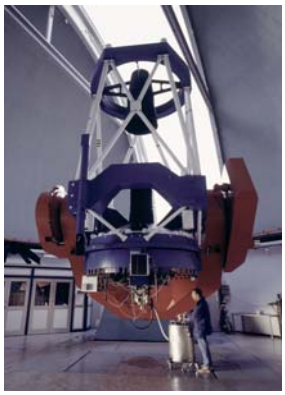
A questions and answers session is the best way to clarify this choice before using CORBA as a practical example to describe more in details the characteristics of an Object Middleware.

4 – Example application



The project: the BareBones Observatory

- We will analyze an hypothetical project to see how the concept of a software framework applies and to verify the benefits.
- We will show examples from the ALMA Common Software (ACS) and the projects using it.



Picture: ESO 2.2m telescope

In order to put the presentation that will follow into a practical context and to be able to discuss upon real examples, we will image to have a real project to analyze.

It is easier to make understandable examples if we consider a small project.

So, instead of imagining the project for a new giant installation we will take a small observatory.

We can eventually think to the refurbishing of an already existing telescope.

We will start analyzing the functional requirements, derive from that the technical requirements and then proceed through various steps of design showing how the software framework is being used. The examples will be based on the ALMA Common Software (ACS) as middleware and application framework.

These few slides provide just basic hook points for the discussion that will take place during the presentation.

Here the questions will be used to define the points to be analyzed with more details.

High level functional requirements

- End-to-end observatory operation
- Control of the telescope mount
- Control of the instrumentation. CCD camera
- Operators sit in a separate control room and monitor telescope status
- Astronomers sit at their institute and control the observation via the internet. They can remotely control the telescope and instruments.
- Data processing is done at the astronomer's institute offline, with data collected from the observatory.

Here is a list of few fundamental high level functional requirements for the software development of a modern telescope, even a small one.

•Software needs to be integrated end-to-end.
Stakeholders have decided that we are responsible for the implementation of the whole, end-to end-software for our observatory.
This means that we have to write not just the control software but also data archiving, processing and interfaces toward the astronomer. A bigger project would have also proposal management and scheduling issues.

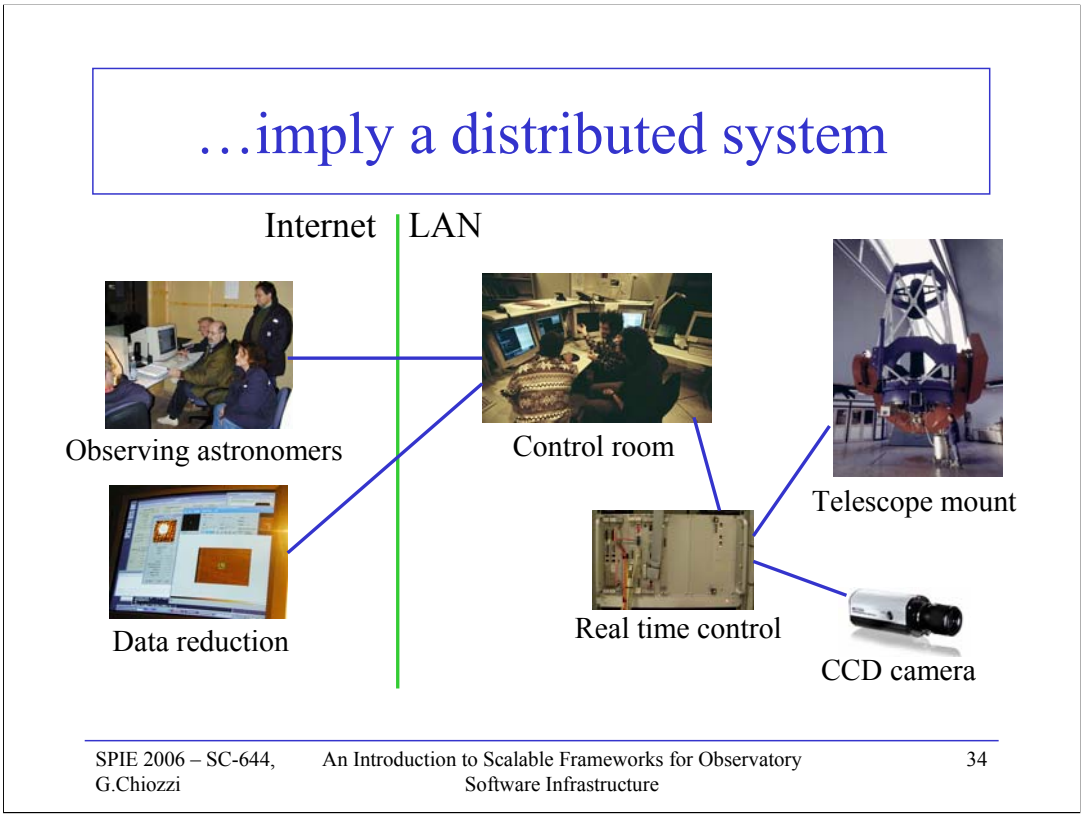
•We have to provide control of the mount, i.e. interface to the hardware on the low side and high level positioning and tracking commands on the higher level side.

•We are also responsible for the control of the instrumentation.
In particular we have a CCD camera to control and the data to be delivered.

•The system will be controlled from a nearby control room, where operators will seat.
Operators will have full control over the system from graphical consoles.
They should have the capability to collect and analyze telemetry from the telescope hardware.

•Astronomers will non have to come to the telescope site.
They will have to be able to observe from their home institute, possibly using a web-based application or in any case an application that does not require complex installation and easy to upgrade.

•Scientific data processing will not be done at the observatory.
The astronomer shall be able to retrieve raw scientific data and process it at the home institute.
But some data processing will be done at the observatory for quality control and for real time feedback during observation



The requirements described in the previous page, even if really just high level, immediately imply an (heterogeneous) distributed system.

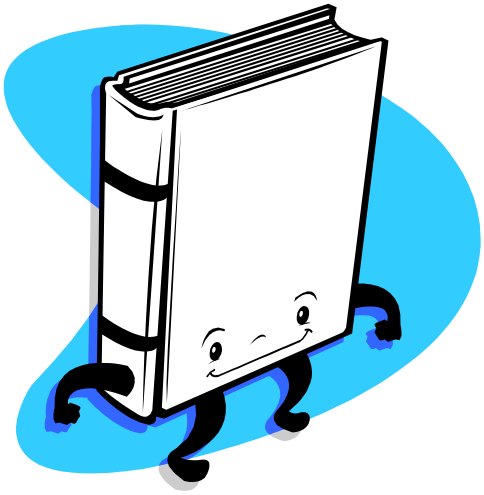
•The mount will have to be controlled by some kind of real time computer.
This computer will have to be interfaced with the hardware and electronics of the Mount.
If this is a refurbishing project, we will probably be able to choose the real time computer, but at least parts of the control and electronics hardware will remain, possibly imposing some constraints on the real time computer.

•The same or another real time computer will have to control instrumentation and CCD camera.
In general it is better to keep instrumentation control separated from the mount control, because we can buy cheaper hardware and have a more modular system

•In the control room we will have normally at least a workstation responsible for high level coordination and one or more console workstations.
We should also have some kind of archive to store raw data and telemetry. We should normally have the freedom to choose the platform we like more for this purpose.

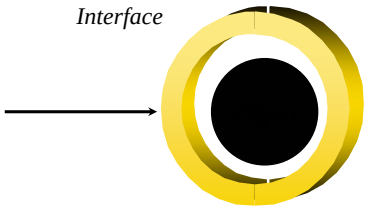
•Observing astronomers will have their own workstations, probably on heterogeneous platforms (Unix, Windows, Mac) and we cannot impose them the purchase of any specific hardware. The client software will have to be therefore platform independent. The same apply to data reduction.

5 – Interface definition



Interfaces vs. Implementations

- First step in design: Identify objects
  - Mount
  - CCD
  - Telescope
  - Observation
  - Exposure



- Second step: Define interfaces
  - We need a formal interface definition language
  - Implementation will come later and is independent from interface
  - Deployment is also independent from interface definitions
  - Interfaces shall be kept as stable as possible, but it must be possible to have them evolve when needed.

As a first step in the analysis and design of the system we have to identify the objects that will interact together.

Typically this will be done in layers.

Per each subsystem we will identify the outer layer of objects that will be used in the interactions between subsystems.

Going deeper in the analysis we will identify recursively internal layers of objects.

Once the objects have been identified, we will have to define their interfaces.

At this point we should not care about implementation and deployment.

It shall be possible to implement the interfaces later on using different programming languages and different architectures, as well as deploying the implementation in different ways.

The absolute separation between interface and implementation is essential to interoperability and scalability.

The best way to define interfaces if by using an implementation neutral but formal interface definition language that will be mapped in the implementation languages later on. Using a formal language is very important to avoid surprises and inconsistencies when integrating subsystems developed by different teams. Using just a textual Interface Control Document (ICD) can very easily lead to problems.

The clients of an object know and see only its interface and the interface shields completely the implementation underneath.

This makes it possible first of all to implement a servant in any language.

But it also means that it is possible:

•To have different implementations for the same interface, if needed in multiple languages.
For example one could provide a mock up implementation in Python for testing and an high performance servant in C++ for the final real time system.

•To have one implementation serve multiple interfaces.
For example, access to legacy system could be done defining the interfaces for each subsystem but implementing only one generic servant (for example a sort of protocol converter) able to implement all of them. Another example is a CORBA interface to access a object (or also relational) database. It is not necessary to provide the implementation for each object type (or table) in the database. One single implementation is able to “incarnate” dynamically all interfaces.

•To have one physical instance of a Servant to represent multiple logical instances. Or the other way around. Or any intermediate situation, based on scheduling and load balancing algorithms.

Interface Definition Language: IDL

- CORBA provides a formal interface definitions language called IDL

  IDL forms a 'contract' between client and servant.
- IDL reconciles diverse object models and programming languages
- Imposes the same object model on all supported languages
- Programming language independent means of describing data types and object interfaces
  - purely descriptive - no procedural components
  - provides abstraction from implementation
  - allows multiple language bindings to be defined
- A way to integrate and share objects from different object models and languages

CORBA specifies an Interface Definition Language. IDL is an ISO standard. Actually IDL is a general interface definition language used to specify interfaces also in environments that are not using CORBA at all.

Rather than inventing a new language, it is a good choice to take an already existing one that has been in use for long time.

Part of the IDL specification is also a formal mapping from IDL to any supported programming language and the other way around. There are for example mappings from and to IDL for C, C++, Java, Python, TCL and many others.

This has the great advantage of reconciling different object models and programming languages allowing them to easily inter operate.

But on the other hand forces some compromises, in particular since some of the supported languages are not object oriented.

For example,

•IDL supports interface inheritance (actually, multiple inheritance) but does not allow overloading, i.e. it does not allow multiple operations with the same name but a different signature. This would not be possible to implement in an easy and intuitive way in the C mapping.

•IDL supports Exceptions, but not inheritance between exceptions

IDL has also important limitations in the specification of interfaces. IDL 3 addresses a number of these limitations.

For example:

•IDL 2 only specifies the definition of published interfaces.
To build a model of the interactions between objects it is useful to be able to specify also:

- •Required interfaces
- •Published event
- •Subscribed events

•The specification of an interface consists only in the signatures of the methods, but not in the pre/post conditions.
For example, it is not possible to specify ranges for parameters

IDL Example: Mount

```
#ifndef _ACSCOURSE_MOUNT_IDL_
#define _ACSCOURSE_MOUNT_IDL_

#include <baci.idl>

#pragma prefix "alma"

module ACSCOURSE_MOUNT
{
   interface Mount : ACS::Component
                {
                void objfix (in double az,
                             in double elev);
                readonly attribute long status;
                readonly attribute ACS::RODouble cmdAz;
                readonly attribute ACS::RODouble cmdEl;
                readonly attribute ACS::RODouble actAz;
                readonly attribute ACS::RODouble actEl;
                };
};
#endif
```


This slide shows a small example of IDL file, representing an abstract (and extremely simplified) telescope mount.

This IDL file defines an interface called Mount that provides:

•one operation called objfix(az, el) to send the mount to a specified azimuth and elevation

•one attribute of type long for the status of the mount

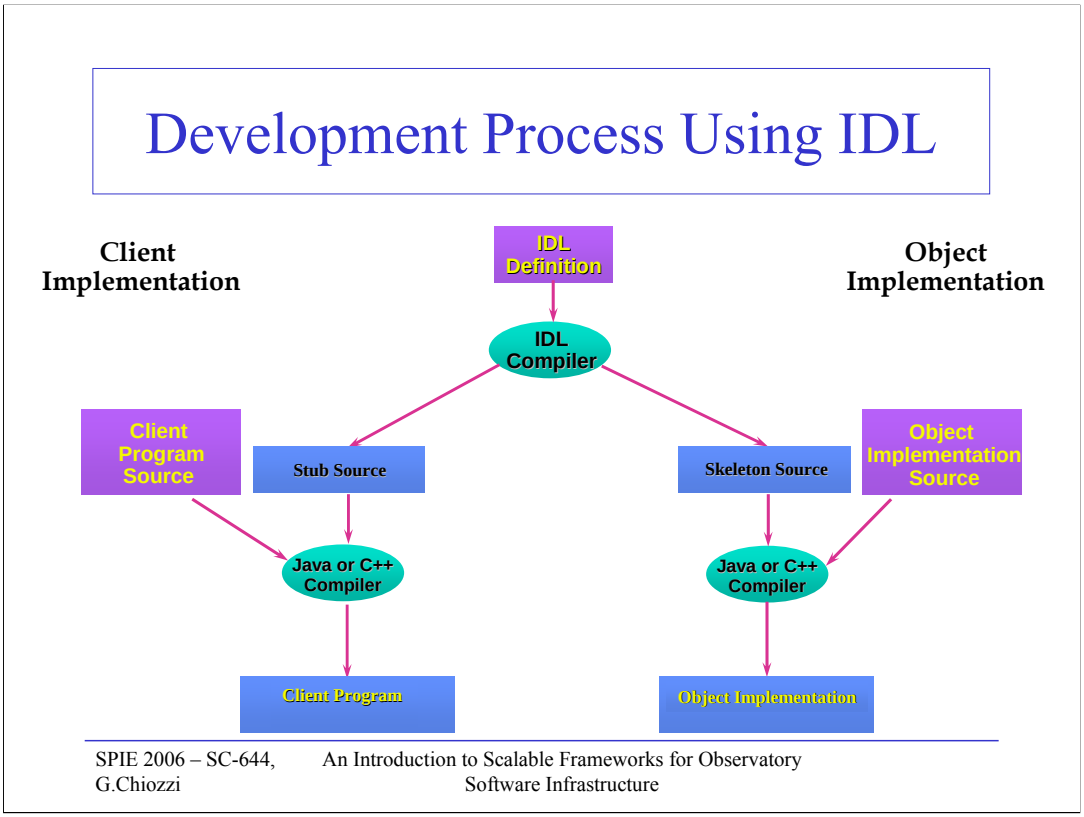
•a few attributes that are complex CORBA objects themselves

As you can see, the structure of IDL is very much taken from C++ include files and even relies on the standard C pre-processor (although some non C++ ORBs like JacORB only provide in practice limited support for pre-processing directives).

The IDL syntax only allows to formally define signatures for operations and attributes with types and names of parameters.

There is no formal provision for describing characteristics of a parameter like ranges or more in general a formal “design by contract” specification.

This would be actually very interesting and IDL could be augmented with comments describing such constraint using OCL (Object Constraint Language, still part of OMG “products” together with UML).



When thinking about the IDL language definition, it is important to think that IDL is a language to define INTERFACES and not IMPLEMENTATION. For example there is no point in implementing such things as:

•private operations or attributes
because what is private in Java or C++ is “hidden inside the implementation” and, therefore has no place in the interface

•operation overriding, i.e. redefining the same operation in a sub-class with a different behaviour, since the behaviour is again domain of the implementation.

It is quite common to confuse interface and implementation aspects.

This diagram shows how an IDL definition is used in the code development process.

•The IDL file is processed by one or more IDL compilers

•Each IDL compiler can produce:

- •Stub code
  The stub code is the code that a client has to call to invoke a remote CORBA object.
- •Skeleton code
  The skeleton is the code that has to be used as the bases for the implementation of the servant.

So the development process can be seen through the following steps:

•The IDL interface is defined in agreement between the developers of servants and clients and published as the contract between them.

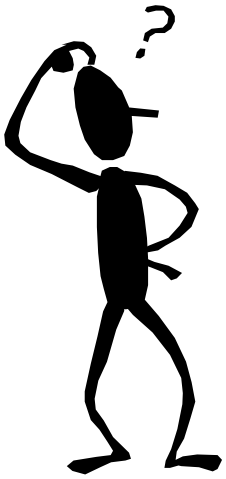
•The developer of the servant:

- •chooses an implementation language (for example C++)
- •selects a CORBA implementation for that language (for example TAO)
- •runs the agreed IDL through the IDL compiler and obtains a Skeleton. In the C++ case, this is an abstract C++ class mapping all operations and attributes of the class into abstract methods
- •subclasses the skeleton and implements all abstract methods

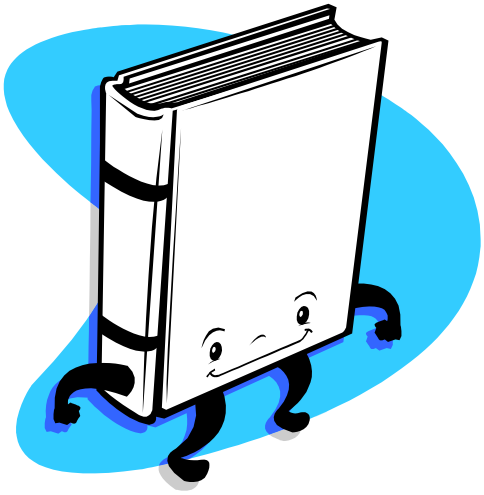
•The developer of the client:

- •chooses an implementation language (for example Python)
- •select a CORBA implementation for that language (for example OmniORB)
- •runs the agreed IDL through the IDL compiler and obtains a Stub. In the Python case, this is a python class mapping all operations and attributes of the class
- •simply calls the stub methods.

Questions (& Answers)



6 – From Object to Component Middleware



Object Middleware shortcomings

- Explicit programming of non-functional properties
- No standard configuration, packaging and deployment facilities

Weak “separation of concerns”

Steep learning curve

The Object Middleware model described in the previous pages has great advantages with respect to the previous approaches.

But experience has also shown some important shortcomings and alone it does not fulfil what it promises.

First of all, using directly CORBA still requires a lot of non-functional programming.

Middleware specific code appears in many places, so the developers need to take into account (and learn) the functioning of their Middleware.

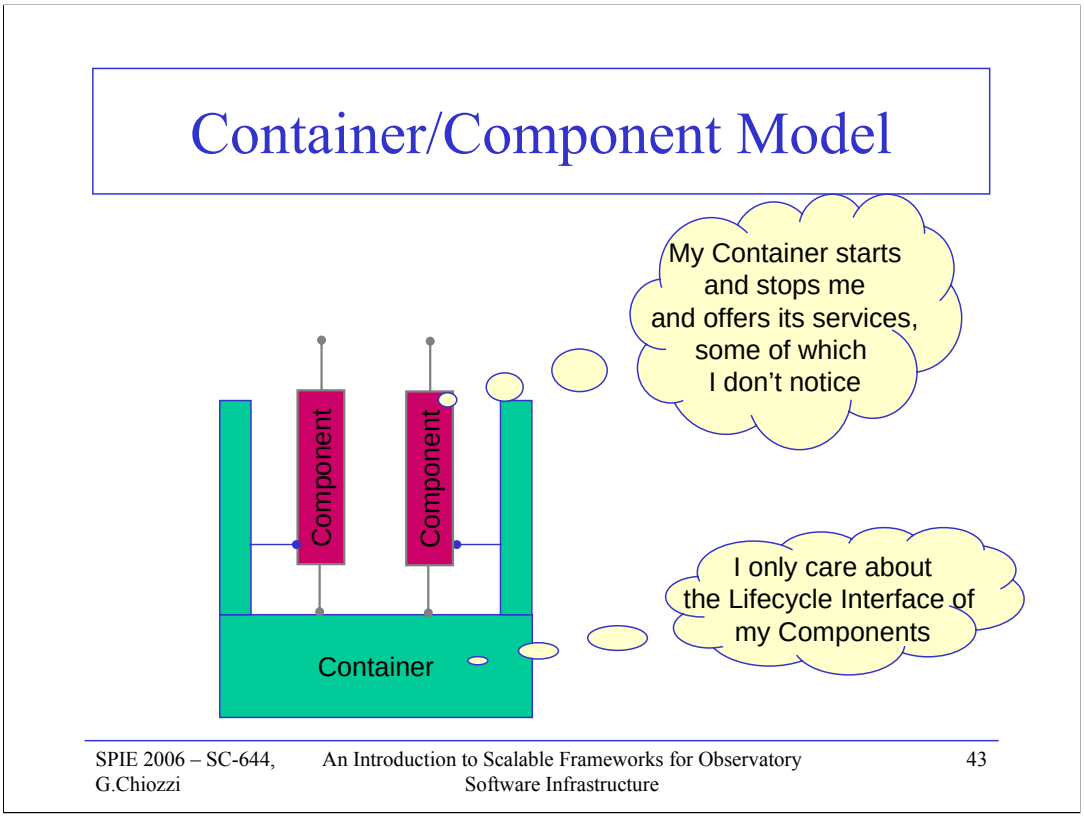
Also, the Object model focuses on the objects themselves and not on a more global view of the system, i.e. how objects are configured, packaged together and deployed. The developers of applications have to take care of this aspect, again mixing functional and technical aspects of the architecture.

The Object Middleware model leaves to the full responsibility of the developers the technical architecture of the system.

We have all the elementary building blocks to build our application, but we still need to find out what of these building blocks we really need in our specific case and HOW to put them together.

All this weakens our capabilities of keeping the desired “separation of concerns” between functional and technical aspects and leads to a steep learning curve for the developers of functional objects, because they still need to learn a lot of the technical aspects of the otherwise powerful and complete middleware.

What we really need is a ***framework***, i.e. a semi-complete application that based on a well specified architecture gives us a general skeleton to support our business logic. Clearly, the framework must be able to satisfy the requirements of our application domain, but at the same time be as close as possible to a “finite” system, since higher flexibility is almost always associated to higher complexity.



Our objectives of separation of functional and technical concerns can be reached "upgrading" from an "object" to a "component" Middleware and adopting a Component/Container Model.

A Component Middleware:

•Creates a standard "virtual boundary" around application component implementations. Functional developers are only concerned by the implementation of their Component code.

•Defines standard Container mechanisms needed to execute Components in generic Component Servers. The Container provides the whole execution environment and access to services for its Components.

•Specify the infrastructure needed to deploy and configure components in a distributed system. Components can be re-configured and moved around in the system without affecting the Component itself or other clients or servants. Configuration and deployment become a separate concern from Component development.

Consequences of this approach for functional developers are:

•Easier learning curve and reduced skill requirements: focus expertise on domain problems.

•Scalability taken care of by the Container and tuneable at deployment time

•Better adaptability and maintainability, with general reduced complexity.

Component/Container: buy vs. build

- Major Component models:

  .NET, EJB, CCM

  - .NET binds to Microsoft platform
  - EJB binds to Java programming language
  - CCM is still immature and there are just a few free implementations. Implementations are not interoperable.
- Off-the-shelf Component Container implementations require a wholesale commitment from developers to use the languages and tools supplied.
- Focus for these Component/Container implementations are big enterprise business systems
- For ACS we decided in 2000 to go for a custom Component/Container implementation. Recent investigations confirm this choice.
- We aim at staying as much a possible compatible with CMM concepts to allow adopting an implementation, when available.

Commercial implementations of the Component-Container model are now quite popular (EJB, .NET).

A vendor-independent specification, the CORBA Component Model (CCM), is part of CORBA 3 specification, but it is not yet widely used.

There are some free implementations (for example from TAO and MICO), but they are not interoperable and are limited to one single language. The specification still needs to be refined on the basis of the experience with the first implementations; interoperability will come as well.

These Component/Container models are rather comprehensive systems, and require a wholesale commitment from developers to use the languages and tools supplied. In particular,

•.NET binds you to develop in and for the Microsoft world

•EJB binds you to the Java programming language

Once more, only the CORBA CCM really promises vendor, platform and language independence.

At the same time, the focus of these models is on big enterprise business systems and they contain a lot of features that are not needed for our observatory and, more in general, experimental facility environment.

For these reasons when we started to develop ACS in 2000 we decided for a simple custom Component/Container model (that we actually inherited from the work done for the Control System of the ANKA Synchrotron). At that time, CCM was not even a complete specification and there were no implementations available.

Recent investigations done by other teams have confirmed that this decision, taken back in 2000, is still justified.

We keep in any case an eye on the evolution of the CCM and we try to keep as much as possible our system aligned with the CCM concepts, to be able to switch to an implementation at acceptable costs. As already mentioned, there are very interesting concepts introduced with CORBA 3 that we would like to adopt.

The choice of developing a custom Component Model is a typical example where the analysis of advantages and disadvantages between generic and custom implementations has made us decide for the custom solution.

While, in principle, general solutions should always be preferred, our custom implementation has the advantage of being:

•Interoperable

•Lighter and with a smoother learning curve

•Easier to customize to the specific needs of our application domain

In the next pages we will describe the ACS model, but most of what said is applicable to CCM and in abstract to the other Component Container models.

Component

- Developed by Application developers
- Deployable unit of software
- **Focus on functionality with little overhead for remote communication and deployment**
- 1…many Components per subsystem
- Functional interface defined in IDL
- Deployed inside a Container
- Well-defined lifecycle (initialization, finalization)

A Component is the basic deployable unit of software.

It encapsulates a coherent and consistent set of application "business" login functionalities, defined and exported to clients by means of IDL interfaces (and textual descriptions for semantic and constraints).

The designer of a subsystem identifies the functionalities to be implemented and partitions them in 1 or more Components.

This partitioning is based on a logical view of the system and leaves out to a large extent deployment considerations and technical issues.

The interfaces between the Components of the subsystem and the external clients are defined by the IDL interfaces in a formal way.

Notice that the IDL interfaces can be used to implement generic or customized simulators effectively helping in decoupling the development/testing of Components inside the same subsystem or different subsystem.

Once a client has the agreed IDL interface of a component it needs to interact with, it can use a simulator or a mock-up to test its own component to a great extent without having to wait for the implementation of the counterpart.

The Mount IDL shown some slides before is already defined as a Component.

Components will be deployed inside Containers and therefore will have to satisfy a few specific conditions, in particular about the life cycle, imposed by the need of living inside the Container.

Container

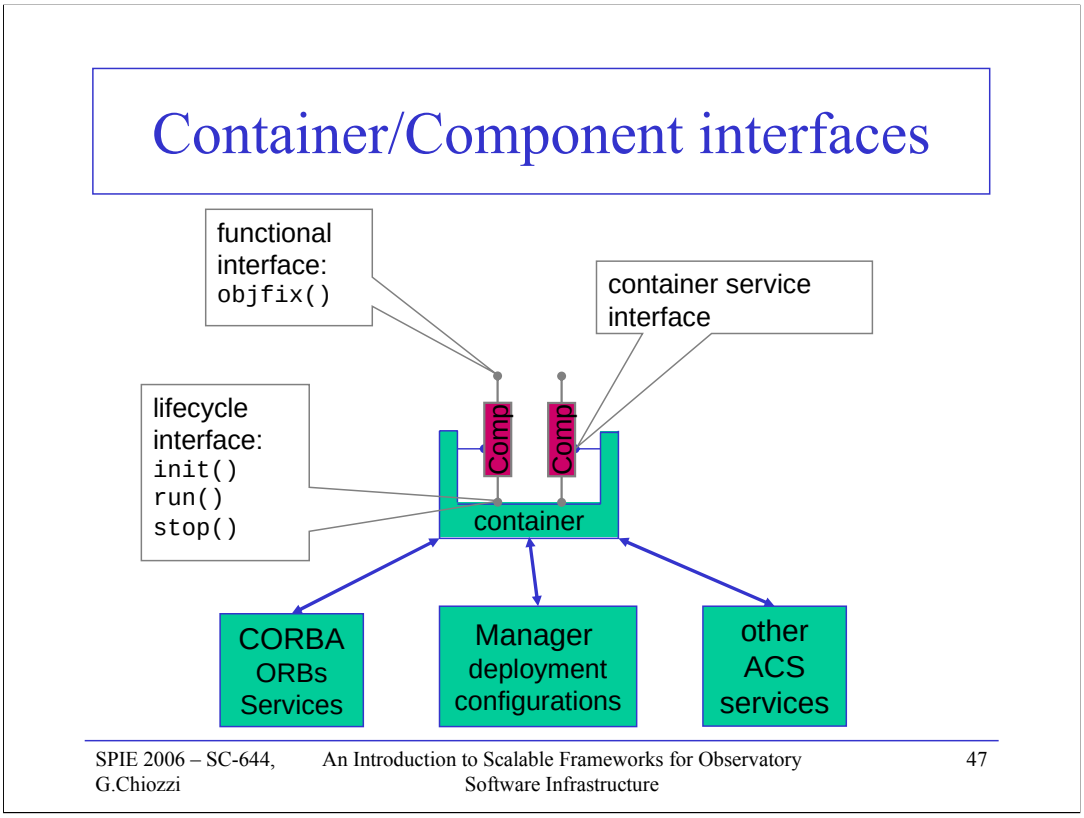
- Developed by the technical framework team
- Centrally handles technical concerns and hides them from application developers
  - Deployment, Start-up
  - Selection of Services, integration with other application specific Services (Error, Logging, configuration, …)
  - Convenient access to other Components and resources
  - Selection and configuration of various CORBA ORBs and policies; here CORBA alone is much too complicated.
- New aspects can be easily integrated in the future, w/o modifying application software

The Containers are generic applications that are implemented by the team responsible for the technical framework, i.e. for the implementation of the Component/Container Model.

They provide the execution environment for the Components and hide from the application developer issues related to deployment, start-up initialization of the run time environment and the services as well as convenient access to other Components and system resources.

In principle it would be possible to completely replace the core of the technical framework (for example replacing CORBA with some newer middleware) simply re-implementing the Container.

Also, new aspects (for example security or command parameter checking) can be easily integrated at the Container level without modifying the application software.



This diagram shows the relations between Components and Containers.

First of all, a Component provides a ***functional interface*** to other Components.

This is the part of the Component that the application developer needs to implement to satisfy its own application requirements.

In ACS the functional interface is specified through a standard CORBA 2 IDL interface.

The CORBA 3 CMM specification extends the IDL syntax to allow specifying also:

•What interfaces the component uses, providing therefore a bi-directional specification of the relationships between Components.

•What *events* a component *publishes*

•What events a Component *consumes*

With these very useful extensions with respect to CORBA 2 IDL specifications, Component specifications can explicitly express the connections that they offer to the outside world AND what connections they expect the outside world to offer them.

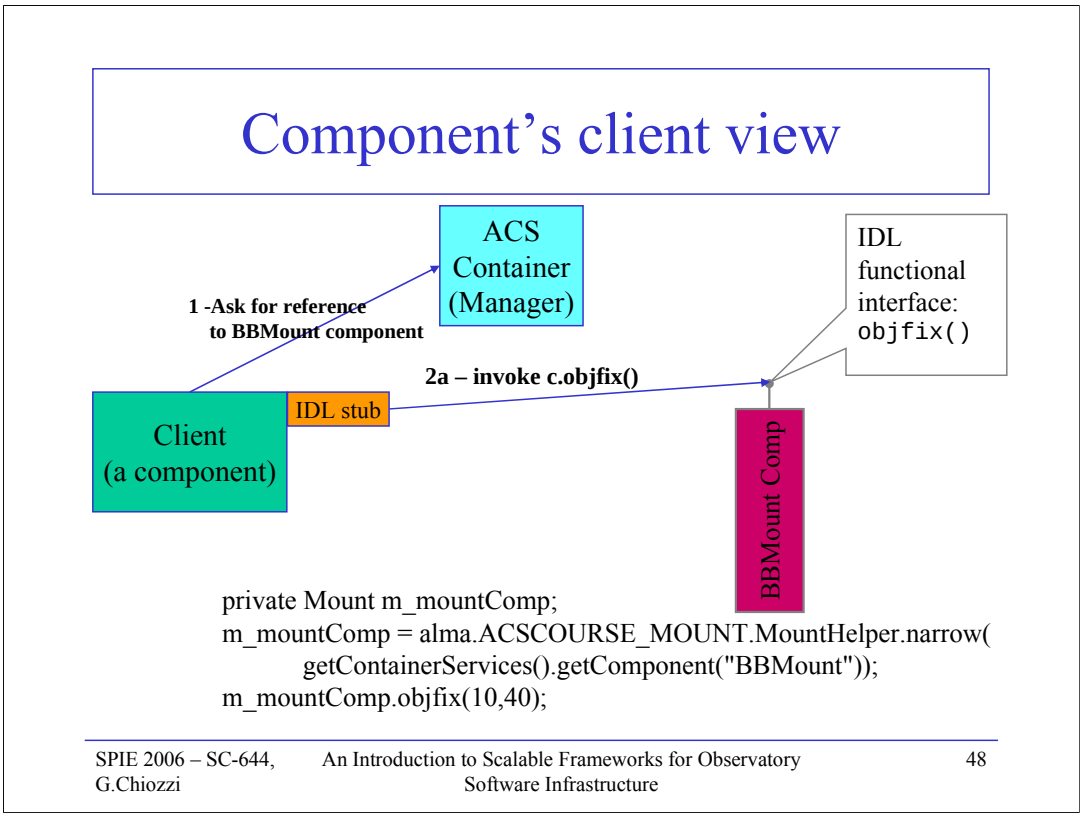
The Container hides as much as possible CORBA and the underlying architecture to the developers of Components, that can concentrate on the functional aspects of their specific Component.

We expect to extend the ACS Component/Container to handle CORBA 3 IDL specifications.

Then the Component is bound to implement a ***lifecycle interface***. In most cases the application developer can simply adopt a default lifecycle behaviour by inheritance or delegation from default Component implementations provided by the Framework.

The division of responsibilities between components and containers enables decisions about where and when individual components are deployed to be deferred until runtime, at which point configuration information is read by the container. If the container manages component security as well, authorization policies can be configured at run time in the same way.

Finally, Containers provide an environment for Components to run in, with support for basic services like logging system, configuration database, persistency and security. A ***container service interface*** is defined by the Container for the benefit of Components to access these services. Developers of Components can focus their work on the domain-specific "functional" concerns without having to worry about the "technical" concerns that arise from the computing environment in which their components run.



First of all let's see the system from the point of view of a Component that need to communicate with another Component.

A Component exposes its IDL interface to clients.

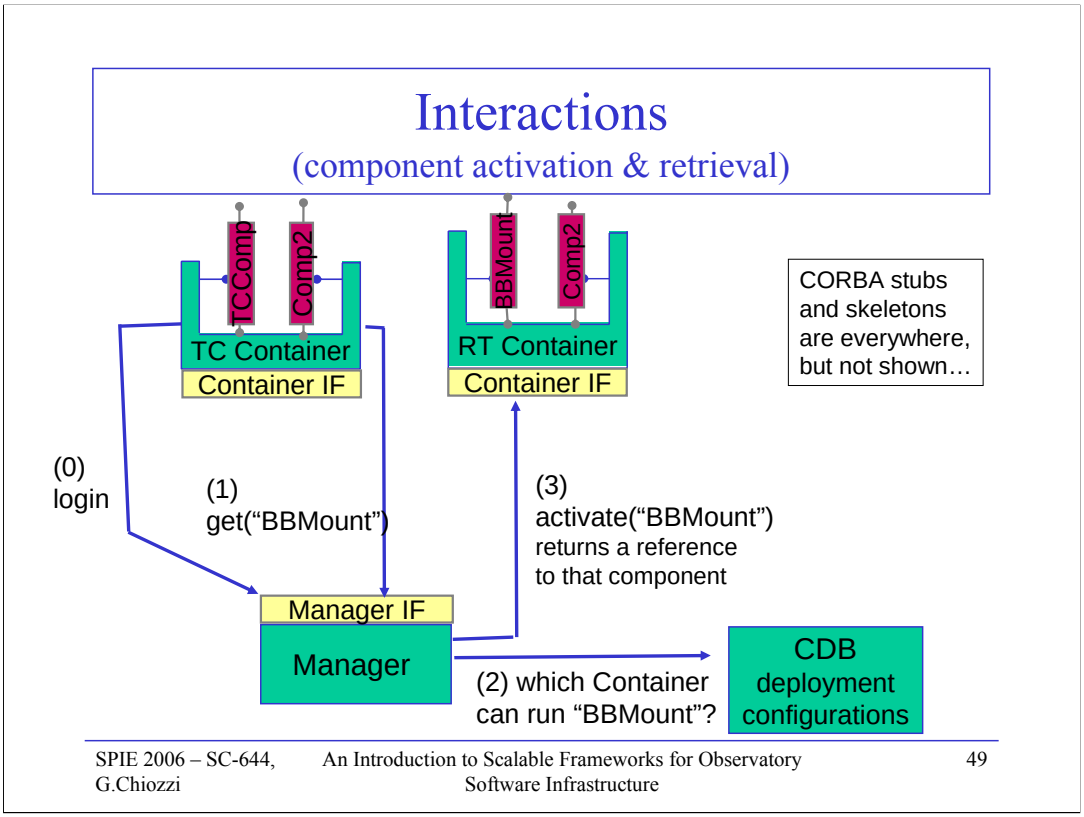
A client (possibly itself a Component) that wants to access the component, needs to ask for a reference. The request is done using the Container Services.

The client is completely unaware of any deployment and lifecycle issues for the Component it wants to talk to.

Once the client has a reference, can call directly the interface via the IDL stubs.

Few lines of code (Java in the example) are in principle sufficient to locate the needed Component, connect to it and call its methods.

Obviously error handling should be taken into account and this is making the code a bit more complex.



Let's see what is actually going to happen under the hood.

The ACS model includes also a **Manager** entity that centralizes deployment configuration, book keeping and system monitoring functionality.

Keeping these functions outside the Container helps significantly in making the model interoperable and language independent, since the Container themselves are simpler and can be therefore easily implemented when a new language or ORB needs to be integrated in the implementation of the Model.

This diagram shows how Components, Containers and the Manager interact.

When a Container becomes alive, it registers itself with the Manager.

The manager is responsible for keeping a runtime image of the system deployment and for monitoring that all entities are in an healthy condition.

The Manager is responsible of object location and therefore plays the role of the CORBA Naming Service (actually it is built on top of the Naming Service itself)

Manager and Container interfaces are also described through IDL interfaces and therefore their implementation language and platform are irrelevant.

Whenever a Component needs another Component, the request goes to the Manager that takes care (using standard CORBA services, like the Name Service) to locate where and if the Component needs to be deployed and, in case, dynamically deploys it and returns the reference to the caller.

The Manager takes care also of de-activating Components that are not needed any more in the system.

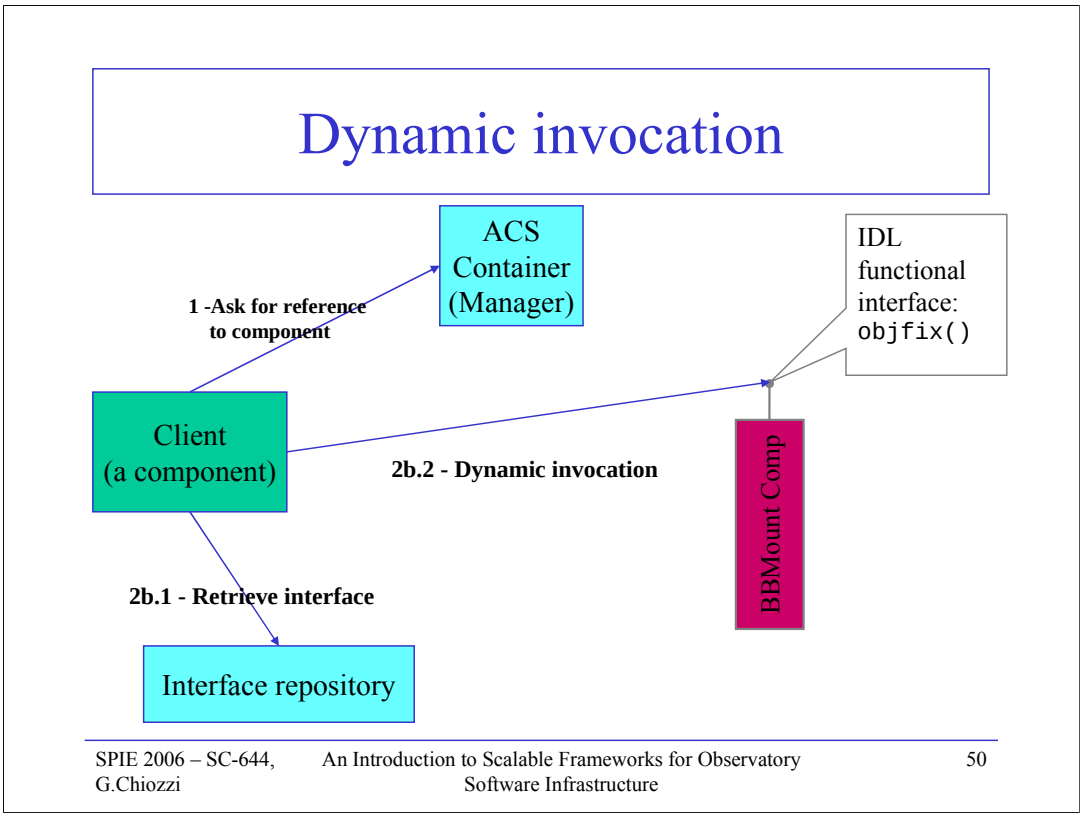
At this point the requesting Component can access its counterpart as needed and be notified by the Manager in case of problems.

The diagram shows how the communication between Components is set up and how it takes actually place.

But from the logical point of view, we can keep two separate views:

•The functional view or the application developer implementing the Component

•The technical view of the administrator responsible for configuration and deployment of the system



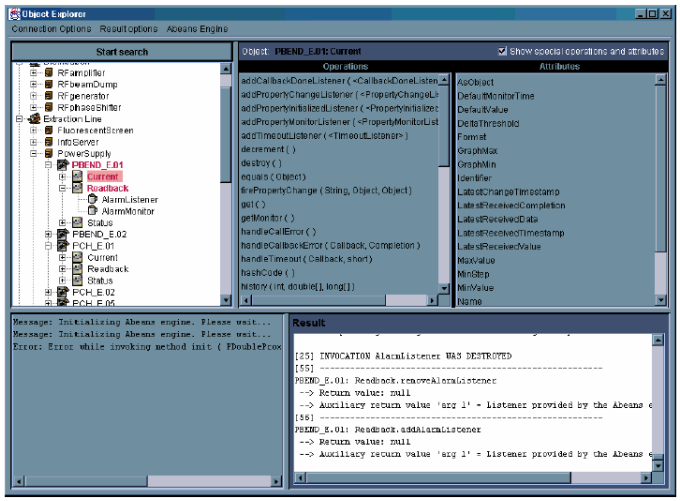
In the previous slides we have described how a Client can statically call methods of a Component.

Alternatively, a client can dynamically discover the interface of a Component by using the CORBA Interface Repository and using Dynamic Invocation

The Interface Repository is the CORBA way to provide language independent introspection facilities and can be extremely useful for the implementation of generic client that cannot or do not want to statically use the stubs generated from the IDL interfaces.

The ACS Object Explorer is such a generic client.

ACS Object Explorer



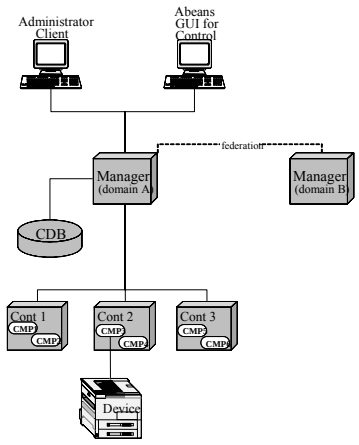
The **Object Explorer (OE):**

• Is a generic tool used for low-level inspection of Components in ACS. It can be used as a debugging or testing tool by the developers and maintainers of a system.

• Allows to interact with any Component or, more in general, any object whose reference can be retrieved from the Manager and whose IDL interface can be retrieved from the Interface Repository.

Component's Administrator View

- An administrator defines deployment by customizing the Configuration Database for the Manager
- Manager is responsible for managing and checking the lifecycle of Components
- Containers are directly responsible for the Components that are assigned to them



The administrator of the system has a different perspective.

He is interested mainly in the deployment:

•Where are Containers running?

•What Components are deployed and deployable in which Containers?

•What is the status and health of Components and Containers

•Who is using who?

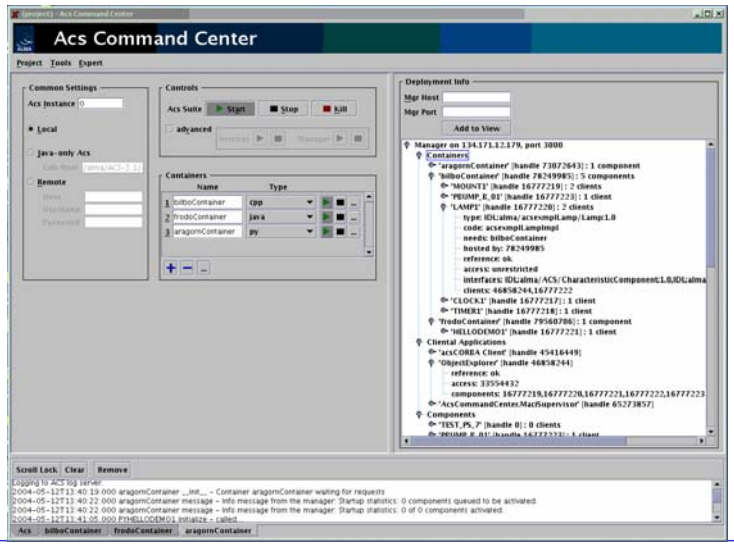
Based on this information, whose static part is kept in a configuration database, it is possible to evaluate and improve the performance of the system or to recover from error conditions. For example it is possible:

•To redeploy Components that have strong and continuous interaction on the same host or Container

•To deploy resource intensive Components on powerful or idle hosts

•Te deploy critical or unstable Components on a separate Container to reduce damage in case of crash.

ACS Command Center

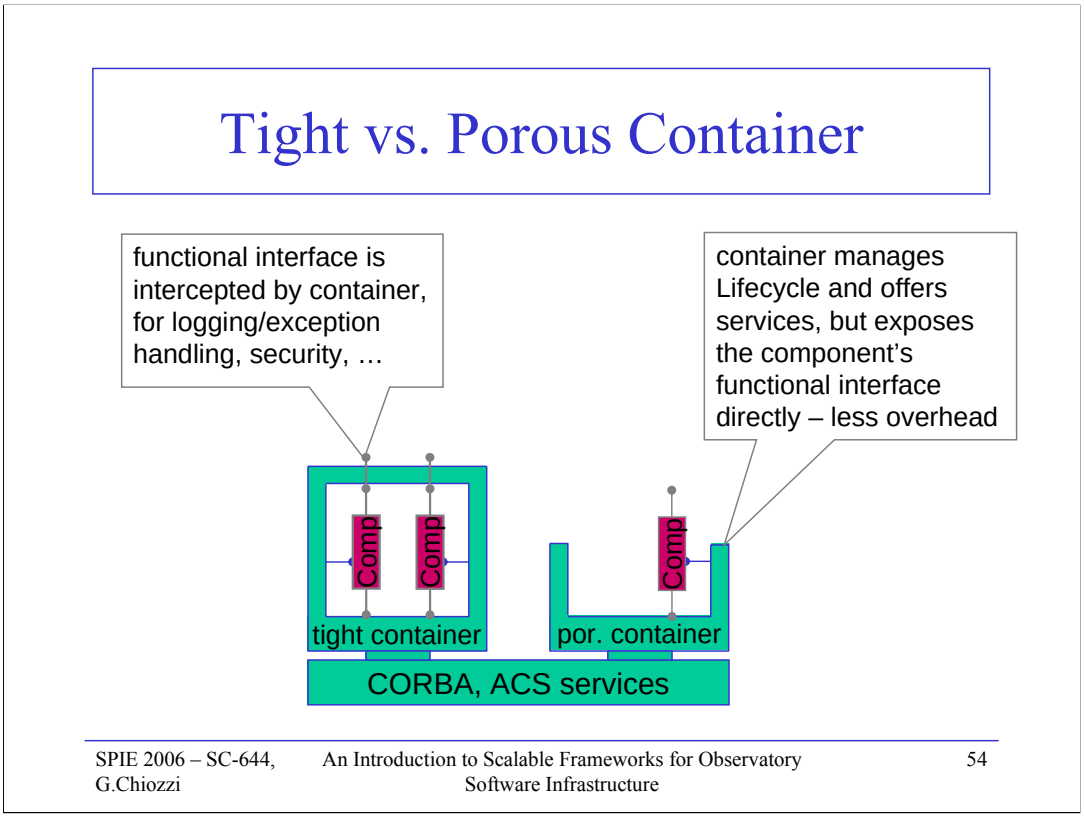


The ACS Command Center is an administrative application used to start and stop ACS services, manager and containers.

It allows to manage the system distributed on several hosts, start tools and inspect the deployment of the system.

The left side allows to control the startup and shutdown of the Services, Manager and Container on distributed hosts.

The tree on the right shows the run time system deployment and the relations between Components and Containers.



There can actually be two types of Containers:

•Porous Container
A Porous Container returns to clients directly the CORBA reference to the managed Components. Once they have received the reference, clients will communicate directly with the Component itself and the Container will only be responsible for the lifecycle management and the general services the Container provides to Components, like Logging.

•Tight Container
A tight Container returns to clients a reference to an internally handled proxy to the managed Component. In this way the communication between client and Component is decoupled and the Container has the capability of intercepting all calls to Components.
This allows us to implement transparently in the proxy layer, for example extra security and optimization functionality or additional debugging aids, at the expense of an additional layer of indirection, with some performance implication.

For example the Java Container in ACS is a tight container and interception is used to provide transparent XML serialisation of complex data structures.

Summary: Component Middleware

- Real Framework
- Component-Container based architecture emphasizes Separation of Concerns in two dimensions:
  - Technical and functional development
  - Implementation and deployment/administration
- Scalability
- Maintainability
- Reusability

The adoption of an Architecture founded on the Component-Container model is a big step forward in the direction of a real application framework:

the applications are framed inside a well defined technical architecture and a lot of the "do and redo differently" code is completely taken out of the hands of the developer.

The Component-Container emphasizes Separation of Concerns in two dimensions:

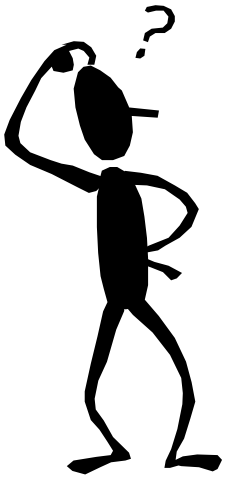
•Technical and functional development
It is possible to split the development team based on the skills in technical experts and domain experts and each developer has therefore a thinner profile.

•Implementation and deployment/administration
In this way it is easier to improve the performance of the system and make it scale up while it grows.

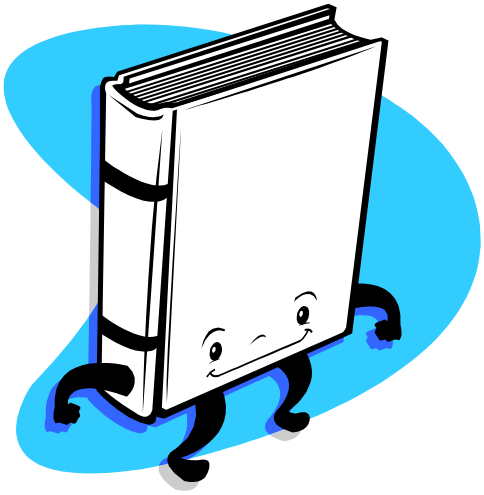
Components are also much easier to reuse and plug into the code, since the three contracts clearly specify what they provide and what they need.

It is also often easier to take legacy code and wrap it into Components that can be integrated into the system independently from the programming language.

Questions (& Answers)



7 – Core services and facilities



ACS Services

- Every application needs a set of basic services, like for example: logging, error management, alarms, events.
- CORBA provides a wide set of services
- For ACS we have identified the services essential for our application domain.
- These have been implemented mostly on top of standard CORBA Services
- The ACS work consists in wrapping the implementation to make it easier to use for our purposes.
- We will analyze here some of services provided by ACS

Until now we have defined an infrastructure to build distributed applications.

We can define interfaces, implement Components, deploy them on Containers and have them interact with each other.

But the are also a number of very important general services that almost every application will need.

For example distributed applications need:

•A centralized logging system

•A distributed alarm system

•An error system capable of propagate errors/exceptions across the network

CORBA provides the specification for a wide set of services, covering very generic requirements or domain specific requirements (see the appendix for details).

For ACS we have identified the that we think are essential for our application domain.

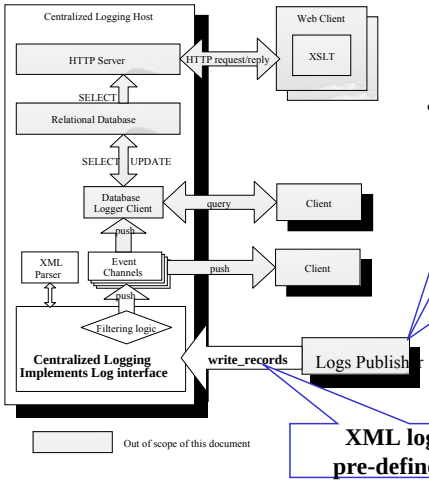
Whenever possible we have based our implementation on an existing equivalent CORBA service. Our work has been in this case to identify and implement on top of the standard services the usage design patterns most interesting for the development of our applications.

In this way we want to make as easy as possible the access to the services for the functional application developer.

New services or new ways of using already integrated services can be added whenever their need becomes apparent.

It is also very important to notice that many services are defined in CORBA in order to warranty vendor interoperability, so that we can easily replace the underlying implementation.

Logging System



- To publish any kind of status and diagnostic information for interested clients and fo archival.
- Based on CORBA Telecom Logging Service.

C++ API ⇒ ACE Logging

Java API -> java.util.logging

ACS Log Service -> IDL

XML logs follow pre-defined schema

A centralized logging system is the most essential service for the operation of a distributed system.

It is also probably the most important debugging tool for a distributed and concurrent system.

Using a source code debugger, it is in fact impossible to debug concurrent issues, because break points and function stepping heavily affect concurrency.

The standard CORBA Logging Service provides a very powerful and scalable logging infrastructure.

But this infrastructure is still too generic for our purpose.

In particular it does not provide any guideline on how to structure the contents of the messages.

In ACS we have therefore decided to structure messages using XML and we have defined an XML schema for the contents of logs.

Then we have implemented wrapper APIs in the supported languages and a generic server for other clients that make trivial to use the logging system and generate messages properly formatted according to the schema.

Doing this we have taken into account that Java has a native logging API and that therefore Java developers should have been very happy of being able to use this standard API to log transparently into the centralized logging system.

The driving forces in designing the ACS layer on top of the standard CORBA logging service have been:

•Define how the flexible and generic CORBA logging system shall be used: choose a path for the functional developer

•Make the usage as simple as possible

•Hide native CORBA and make it look like APIs the developers are already comfortable with.

Log message

- XML formatted messages:
  - several different types: trace, debug, error,...
  - timestamp
  - runtime: components, thread, process, host, context
  - source: file, line routine
  - priority: 0-31
  - message (text)
  - others: identification, stack ID, stack level, ..

This is a list of the most important attributes of a log.

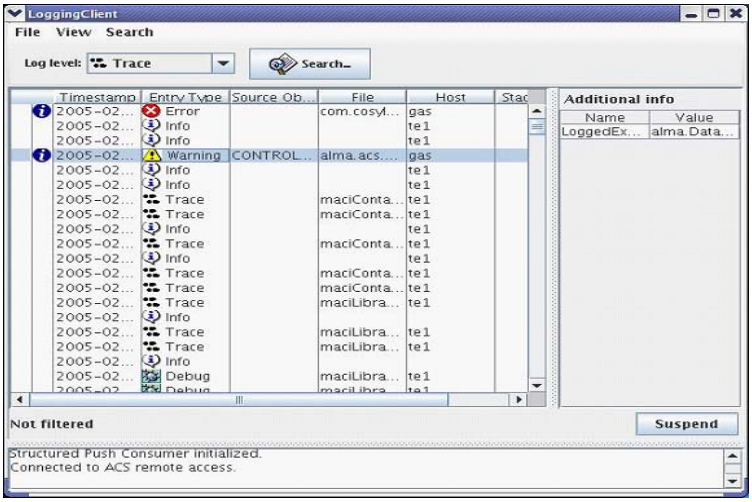
It shall be possible to uniquely identify the message, the place in the code and the runtime context for the message.

Example of a log message

```
<Debug
  TimeStamp="2002-10-7T13:44:16.530"
  Host="te1.hq.eso.org"
  Process="MountContainer"
  Thread="main"
  Context=""
  File="mountImpl.cpp"
  Line="205"
  Routine="mountImpl::~mountImpl
>
  Component destroyed
</Debug>
```


This is an example of log in the native XML format.

Logging client, jlog



An API allows to write clients of the logging system.

In this way any application can retrieve logs while they are published.

We have two clients of uttermost importance:

•A GUI client that allows operators to display and monitor logs.

•An Archive Client that receives logs with the purpose of storing them in a persistent archive for later analysis and retrieval.
ACS itself does not provide any archive, but the responsibility of providing one is left to applications.

Error System

- We need a unified way of dealing with errors through the system
- CORBA supports “distributed” exceptions
- We need more:
  - Error format standardisation
  - Error handling design patterns
  - Error trace
  - Error logging
  - Synchronous and asynchronous error handling
  - Error browsing and definition tools

The ACS Error System provides these features

It is extremely important to have a coherent and complete way of handling error conditions all over the system.

This involves handling errors:

•in different programming languages:
an error in a C++ Component has to be propagated and understood by a Java Component

•distributed over the network:
an error in a Component in one host hast to be propagated over the network to a client component on another host, possibly with a different operating system and architecture

CORBA allows defining exceptions in IDL (with some limitations due to the need of supporting non exception-aware programming languages) and throwing exceptions over the call to IDL operations. This means that a remote call can throw an exception that goes back over the wire to the caller and looks the same as a local exception. The exception’s data is handled by CORBA and marshalling is therefore transparent.

The possibility of treating local and remote exceptions in the same way is extremely important in order to build transparency in the distribution of Components, but it is not sufficient.

There are many other issues that we need to solve to allow treating efficiently error conditions in Components:

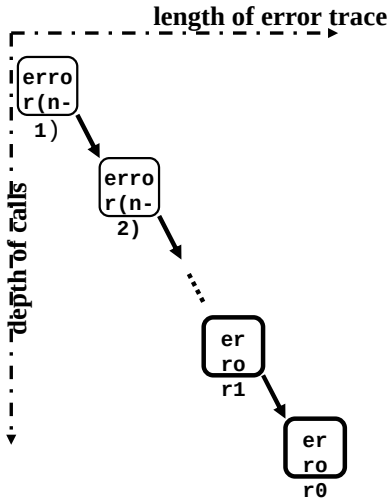
•Error format standardisation
A part from the exception “name”, we often profit significantly from additional context information in the data coming with the exception. But to be able to interpret this information the data structure shall be standardized in the format and contents.

•Error handling design patterns
There are a number of well proven error handling design patterns (see ARCUS Error Handling for Business Information Systems: http://www.eso.org/projects/alma/develop/acs/Releases/ACS_3_1/Docs/ARCUSErrorHandling.pdf). Providing a standard implementation for these patterns helps a lot in writing solid applications.

….continues

Error Trace

- linked list of objects/structures/exceptions
- chain of calls create an error trace: at each level error information can be added
- It is contained inside:
  - an exception
  - a completion



…continues

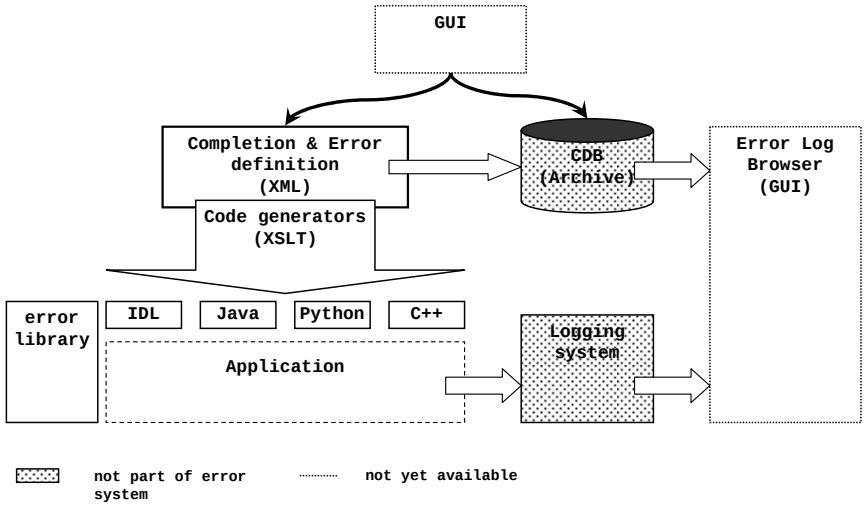
•Error trace
In a standalone application running in a single executable, a low level error is propagated up through the call chain until it reaches somebody that is capable of handling it or until the application is terminated. At each level useful context information can be added. Some languages like Java provide native support for retrieving and manipulating the call chain, but others like C++ do not.
This is the Backtrace design pattern and it is very useful to provide an implementation that works over CORBA network calls.

•Error logging
With a distributed system the Backtrace pattern allows to trace the chain of errors across distributed components, but the error traces end up all the times in different places, i.e. where the component that finally handles them resides.
It is important to have a centralized place where it is possible to browse and search for errors, with context information allowing to identify where each error occurred.
This can be done sending all error traces to the centralized logging system

•Synchronous and asynchronous error handling
The exception mechanism works for synchronous calls: the execution of an operation fails, an exception is thrown and it is caught by the caller.
But in highly distributed systems many actions have to take place asynchronously: an activity is started by a method call, but the method returns immediately and later on a callback is used to report the result. We need to have a standardised mechanism to report errors also in such asynchronous situations.

•Error browsing and definition tools
It is convenient to have friendly tools to browse the errors and to define the error structures used to report context specific information.

ACS Error System Architecture



Different middleware systems provide support at different levels for such issues, but they cannot provide a comprehensive solution, because they want to be fully general and here we often have some percolation from the application domain requirements.

ACS for example provides a solution on top of CORBA to these problems taking into account our Observatory/Scientific facility needs.

This diagram shows the architecture of the ACS error system.

Essentially we have:

•Defined a way produce and transport error traces with exceptions and propagate them consistently across languages in CORBA calls.

•Designed an XML schema for the definition of error conditions and for their storage in the logging system.

•Implemented code generators that from the XML error definitions produce IDL definitions for the exceptions and convenience support classes in the various programming languages, to overcome limitations in the CORBA support for exceptions.

•Implemented some standard error management design patterns

•Defined how to propagate errors in asynchronous calls

Alarm System

- Deals with abnormal situations
  - Fault states (FS)
  - Range from severe alarms to warning states
- Provides
  - FS collection, analysis and distribution, definition and archiving
  - FS reduction
  - Dedicated alarm consoles
- Porting of CERN LASER system to ACS technology

An alarm system is a cornerstone service in every computer controlled environment.

Its purpose is the notification of exceptional conditions (fault states) 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 collaboration with the CERN LASER team to integrate it with ACS.

The Laser implementation uses an architecture similar to ACS but a different set of underlying technologies.

The porting work has consisted therefore in identifying how to replace the original technologies with ACS and refactor when convenient the code to keep the biggest possible common code base, isolating the access to specific technologies in separate interfaces.

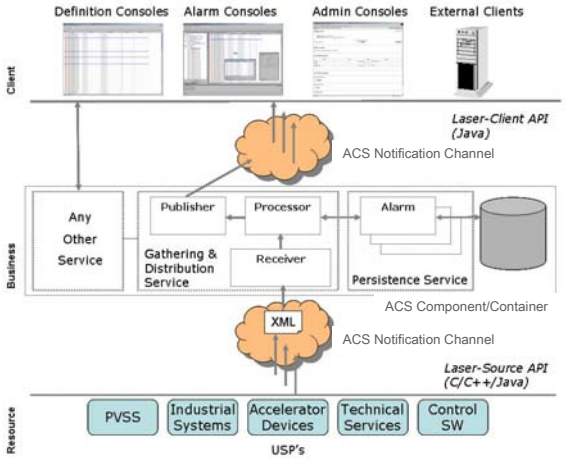
LASER is a messaging system; it collects, stores, manages and distributes information regarding abnormal situations called Fault States (FS). A FS is identified by a triplet: the Fault Family (FF), the Fault Member (FM) and the Fault Code (FC). The FF represents a set of elements with the same kind of 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.

System overview

- A distributed, layered system
- Layers communicate via well-defined interfaces
- Resource tier
  - Dispersed set of sources detecting FS changes
- Business tier
  - Implements business logic and services
- Client tier
  - Dedicated consoles and software clients

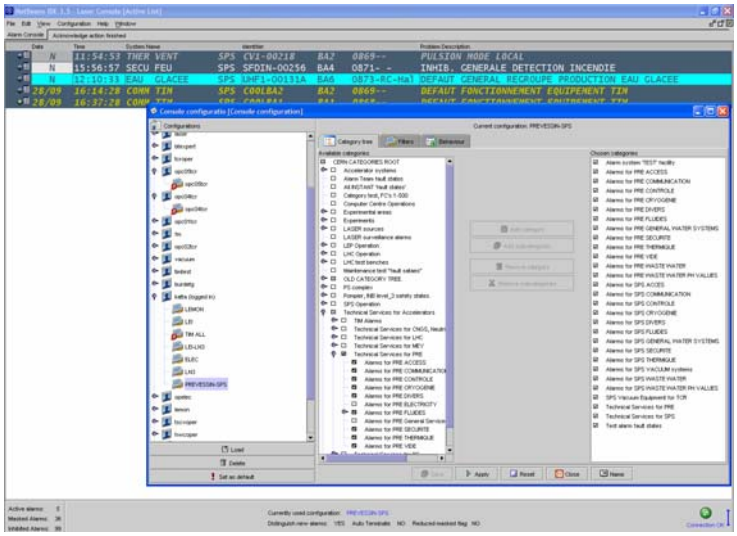


•Resource tier
- •Consists of a dispersed set of alarm sources
- •Communicates with business tier via the LASER Source API
  - •Triggers FS changes
  - •Sends 'Keep-alive'/Synch message
- •Implemented on a variety of platforms and OS

•Business tier
- •FS collection, analysis and distribution
  - •FS changes are asynchronously and sequentially collected from sources
  - •Different techniques are used to reduce the number of alarms distributed
  - •FS's are distributed into a hierarchy of domains of interest
- •USP monitoring
  - •'Watch-dog' mechanism based on USP's 'keep-alive' message
- •Alarm console user authentication & configuration
- •FS definition
  - •FS definition inserts, deletes, updates
  - •FS relationships, used for reduction
- •FS archiving
  - •FS changes
  - •FS definition changes
- •LASER implementation Relies on the Java 2 Enterprise Edition (J2EE) specifications, while ACS porting relies on CORBA and ACS Component/Container
  - •Java Messaging System (JMS) replaced with ACS Notification Channel wrapped inside a JMS interface
  - •Enterprise Java Beans (EJB) replaced by ACS Component/Container
  - •Hibernate/Spring replaced by ACS Configuration Database

•Client tier
- •Dedicated alarm consoles and software clients
- •Communicates with the business tier via
  - •The LASER Client API
    - •FS changes are sent asynchronously, based on the set of categories and filters passed to business tier
  - •The LASER Console API
    - •Login and configuration facilities for the dedicated alarm consoles

Alarm System console



Probably the most important element in an Alarm System is the operator's console.

This graphical user interface shall enable the operators to quickly and efficiently analyze the fault states occurring to identify the ultimate cause and take appropriate actions.

It is therefore necessary to allow a large degree of configurability and good filtering capabilities.

The development of such a GUI is very expensive and therefore the possibility of sharing one implementation across different projects is very appealing.

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.

Configuration Database

- The ACS Configuration Database addresses the problems related to defining, accessing and maintaining the configuration of a system.
- For each Component in the system, there might be a set of static (or quasi-static) configuration parameters that have to be configured in a persistent store and read when the Component is started up or re-initialized.
- This includes the "deployment structure" of the system, i.e. which statically deployed Components are part of the system and their inter-relationships.
  This information is used by the Component/Container infrastructure.

The ACS Configuration Database addresses the problems related to defining, accessing and maintaining the

configuration of a system based on ACS.

Typically, Components in the system have an associated set configuration parameters.

For example, Components representing devices need to define device characteristics, calibration parameters or limit values.

If we confider for example the Component representing a motor we normally need to be able to configure information like the brand, the serial number, the limit positions and so on.

There is clearly no point in hard coding this information in the code, because devices are replaced and recalibrated.

We need to put the configuration in a persistent store, keep it under configuration control and be able to and read it whenever

the Component needs it, for example at startup or initialization time.

The information that needs to be stored in this Configuration Database includes the *structure and deployment* of the system, i.e. which Components are part of the system and their inter-relationships. For what concerns system deployment, looking at the CDB only you should be able to see how the Components are distributed among the Containers and on what hosts the Containers are running.

For Components connected to HW, this would tell you as well what HW you are using and where it is located.

Changing the CDB you can move Components around and distribute them in a different way in the system.

CDB Issues

1. input of data by the user
   System configurators define the structure of the system and enter the configuration data. Easy and intuitive data entry methods are needed.
2. storage of the data
   The configuration data is kept into a database.
3. maintenance and management of the data (e.g. versioning)
   Configuration data changes because the system structure and/or the implementation of the system's components changes with time and has to be maintained under configuration control.
4. loading data into the ACS Containers
   At run-time, the data has to be retrieved and used to initialize and configure the Components.

There are 4 different issues related to this problem:

1. input of data by the user
System configurators define the structure of the system and enter the configuration data.

2. storage of the data
The configuration data is kept in a database.

3. maintenance and management of the data (e.g. versioning)
Configuration data changes because the system structure and/or the implementation of the system's components changes with time and has to be maintained under configuration control.

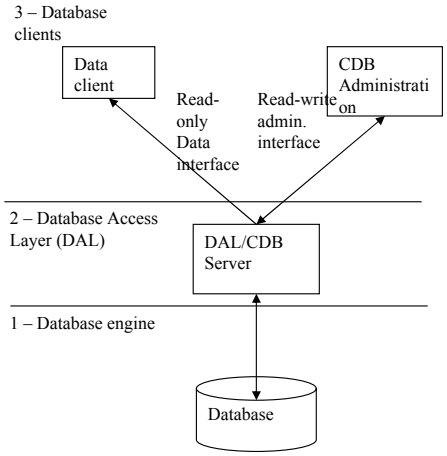
4. loading data into the ACS Components
At run-time, the data has to be retrieved and used to initialize and configure the DOs.

A CDB implementation has to take all these issue into account.

Three-tier database-access architecture

DB Engine independent Tiers:

- Database engine
- Database Access Layer (DAL).
- Database clients:
  - Data clients
  - CDB administrators



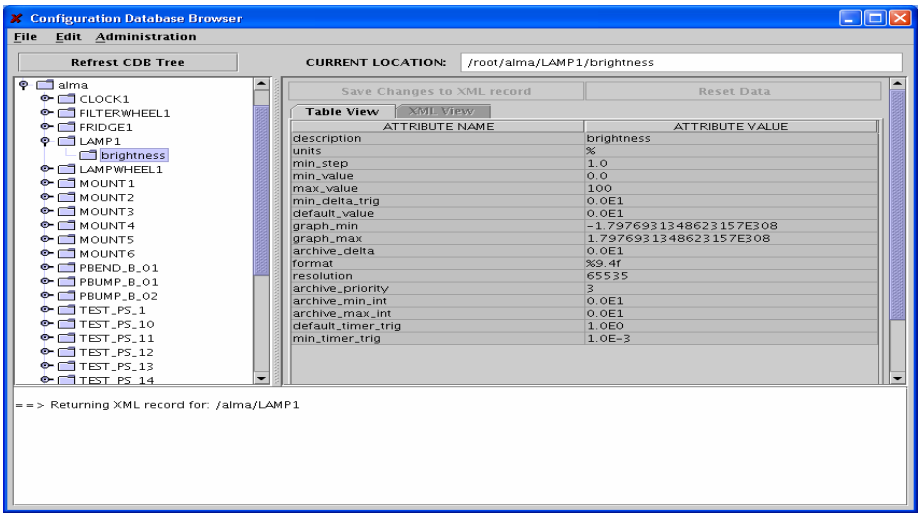
The architecture of the ACS configuration database is based on three layers:

1. The Database Itself. It is the database engine used to store and retrieve data. It may consist of a set of XML files in a hierarchical file system structure or it may be a relational database or another application specific database engine.

2. The Database Access Layer (DAL) hides the actual database implementation from applications, so that the same interfaces are used to access different database engines. For each database engine a specific DAL CORBA Service is implemented. The DAL is defined in terms of CORBA IDL interfaces and applications access data in the form of XML records or CORBA Property Sets.

3. The Database Clients access data from the database using only the interfaces provided by the DAL. Data Clients, like Components, Containers and Managers retrieve their configuration information from the Database using a simple read-only interface. On the other hand, CDB Administration applications are used to configure, maintain and load data in the database using other read-write interfaces provided by the DAL layer.

CDB Browser



The CDB browser is a client of the CDB that allows to:

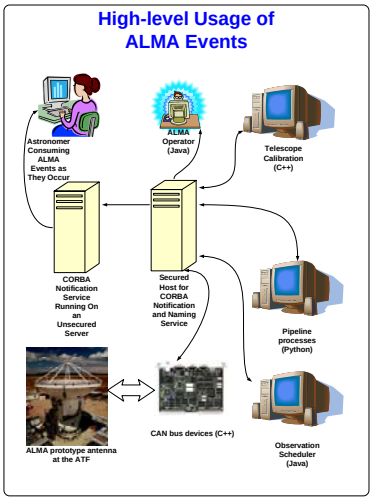
•navigate through the structure of the database

•browse the values

•Modify values and add new nodes

Notification Channel

- Events are **widely used** in ALMA for synchronization and asynchronous, *-to-* communication.
- Decoupling of Consumers and Suppliers
- Very easy interface:
  - Supplier classes
  - Consumer classes
- Contract based on IDL data structures.
- Strong naming conventions and checking tools
- Administrative interface
- Quality of service



ACS has provides four ways to communicate between components.

•Synchronous method calls.
A client calls a method defined in the IDL interface of a Component. The action is complete when the method returns.
We handle the special case of calls passing complex data structures using XML serialization techniques10

•Asynchronous method calls.
A client requests explicitly a service from a Component by calling an asynchronous method defined in the IDL interface of the Component and registers callbacks that directly "connect" Component and client. The method returns immediately and the Component will invoke the callback periodically to report back to the caller or when completed.

•Notification Channel
A Component publishes data over a CORBA Notification Channel and any interested client subscribes to the Notification Channel to be notified when data is available. Distinct from the asynchronous method call, there is no direct "connection" between publisher and subscriber, and the publisher is not aware of the subscribers collecting published data.

•The special case of publishing huge data volumes is handled in ACS by the Bulk Data Transfer package. This implements high efficiency transfer of high volumes of streaming data and is based on the TAO implementation of the CORBA Audio/Video streaming service. This is essential for the ALMA Correlator to send data at rates between 6-60 MB/s to the Archive.

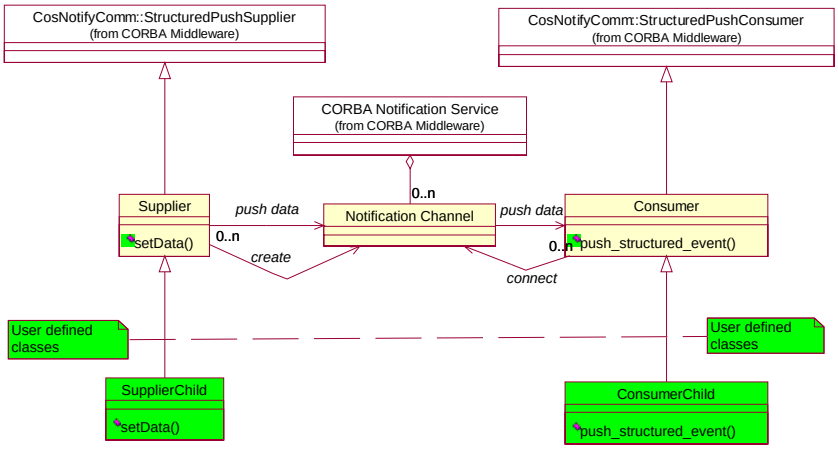
In recent years we have seen the Notification Channel used much more than expected, as a general mechanism to:

•Synchronize the activity of subsystems by means of the publication of synchronization events

•Publish data to be retrieved by one or many subscribers, not known a priori.

The event names and data structures to be transported are defined in the IDL interface specifications and can be used with few lines of code. The definition of events at the level of IDL interfaces has been very important, not only to simplify usage, but also to make it possible to infer the usage of events using source code and run time analysis tools that we have developed.

Notification Channel class diagram



The ACS classes that implement the Notification Channel have evolved together with increasing usage and are by now very easy to use, hiding in an effective way the underlying CORBA Notification Service.

Most of the use cases we have analyzed we have just the following pattern:

•A data structure is defined

•Whenever data is available the structure is filled in and published

•One or more subscribers receive the data they are waiting for.

Therefore we have provided in ACS an implementation for this pattern:

•The event data structure is defined in IDL

•A Supplier class allows to control easily when data is pushed on a channel:

- •Suppliers can create a notification channel
- •Suppliers know when a consumer has subscribed to an event type on the channel it publishes structured events to. A "smart" supplier will only publish events (thereby reducing network traffic) when consumers are subscribed. Only useful in a one-to-many model.
- •Suppliers can automatically execute a method if the connection is ever lost.
- •Suppliers can destroy a notification channel (coordinating with other suppliers when multiple suppliers publish on the same channel).

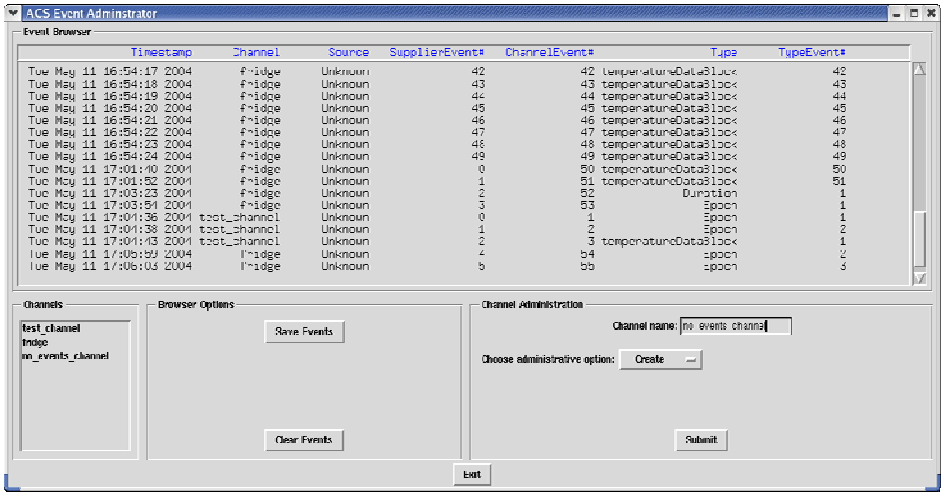
•A Consumer class allows to control easily when data is given to a client

- •Subscribe to and unsubscribe from all types of events.
- •Filter out structured events they don't want to process.
- •The consumer doesn't have to do anything with the event's data. Can literally be used as a notification mechanism.
- •Specify when they are ready to start receiving events.
- •Suspend and resume their connections to the channel at any time.
- •Notified when a Supplier begins publishing a new type of event and dynamically subscribe to it. The same holds true when subscriptions are no longer offered.
- •Automatically execute a method if the connection is ever lost (i.e., the channel is destroyed).

See the paper "A CORBA event system for ALMA common software", from D.Fugate for more details.

Our objective here has been to provide a very simple and standardized interface for the most common use case, while complex situations can still be dealt directly with the Notification Service APIs.

Event Browser



The Event Browser is a generic client for the Notification Channel.

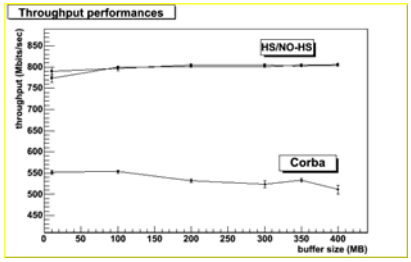
It is capable of subscribing to any publisher and display any kind of event published on the Notification Channel.

It is therefore an extremely valuable debugging tool.

Bulk Data Transfer

- Requirements from the correlator:
  - 64 MB (megabyte)/sec
- Based on CORBA A/V streaming service
- TAO C++ implementation
- Very easy interface, based on our use cases
- No CORBA A/V visible

See poster: PO1.032-6



Achieved Performance

- **Gigabit P2P Ethernet**
- **BD throughput around 800 Mbits/s (~100 MB/s) requirements fulfilled**
- **CORBA throughput around then 500 Mbit/s (~ 55 MB/s)**
- **Estimated gain in the throughput around 30%**

ALMA has very strong requirements for the amount of data that needs to be transported by software communication channels, in particular from the correlator to the archive (raw data from the antennas is luckily enough not under software responsibility).

A major development of the last year has been the bulk data system, devoted to the transport of huge amounts of data and based on the CORBA Audio/Video streaming service specification.

The bulk data system is described in details in [9] in this conference.

We have implemented very easy to use classes on top of the A/V streaming that implement the use cases we have identified for ALMA shielding completely CORBA and the details of the A/V itself.

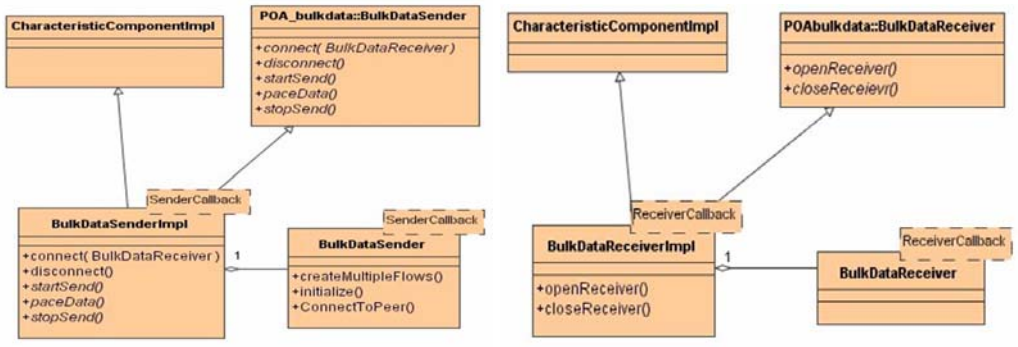
Using this system we avoid the performance penalty introduced by the CORBA communication protocol, transmitting data outbound directly in TCP or UDP format. On the other hand, we still use a well defined and standardized protocol for the handshaking and administration saving the effort of designing and implementing our own proprietary solution.

Unfortunately the only implementation we have available is the TAO C++ implementation. For the time being we do not have strong requirements to have the bulk data transfer available in Java or Python. We think anyway that it would be a reasonable effort to port to Java the basic components that would be needed to have our use cases working.

Bulk Data Transfer classes

Sender Component

Receiver Component



ACS Sender Component class diagram

The ACS Characteristic Component relative to the Sender is implemented as a C++ template class. The template parameter is a callback which can be used for sending asynchronous data. This callback class provides methods for sending data at predetermined user-configurable time intervals. To allow sending data in a synchronous way, a default callback class is provided, which disables the asynchronous mechanism.

ACS Receiver Component class diagram

The ACS Characteristic Component relative to the Receiver is implemented also as a template class. The template parameter in this case is a callback class, which has to be provided by the user and must be used to actually retrieve and manage the received parameters and data stream (see description in the next section).

Summary: services

- Every application needs some basic services
- Services as provided by generic middleware systems (like CORBA) are still too general
- It is very convenient to tailor the services to our application domain
  - by identifying typical usage patterns
  - providing very simple wrappers for application developers to use such patterns

Every distributed application in our domain needs several or all the services described in the previous slides.

Generic middleware infrastructures, like CORBA, provide typically all of them and often many more.

But what they provide is typically very generic because it shall be usable in many very different application domains and by projects with different objectives and programming standards.

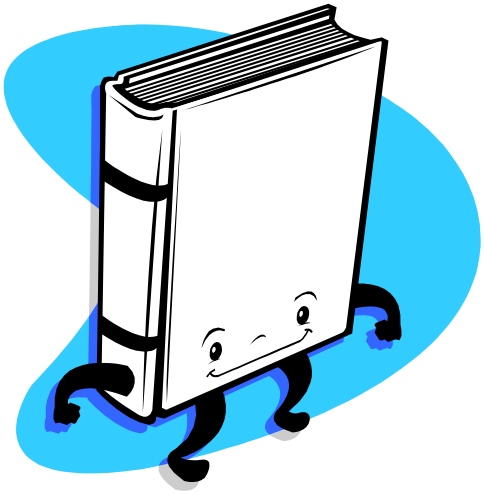
It is very convenient to select the “best way” to use the services according to our project constrains.

Once this is done, it is possible to provide higher level wrappers that make using the services in this way very easy.

The underlying middleware technology can be very well hidden behind simple APIs, so that functional application developers can use them forgetting about technical concerns.

In ACS we have been following this approach for all the services we are using.

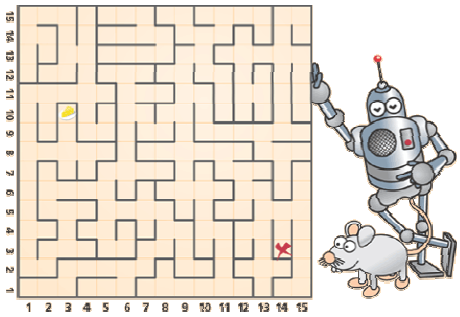
8 – High Level Framework



Even more high level?

- We have a framework but there are still too many ways to draw the architecture of the system.
- Take decisions, restrict the paths, show the "best way" for our domain
- Leave alternatives open

We will discuss examples from ACS



We have up to this point identified a very powerful and generic ***framework***, providing a number of basic services tailored to our need.

But still this is too generic: we can still put together these building blocks in many different ways and probably different developers in the team will take very different roads to the solution of similar problems.

To isolate as much as possible the application developers from the technical concerns, we need to provide also solutions to typical architectural problems in our domain and give them a framework closer to the "final" system. The technical framework team has to identify the "best way" among the possible solutions and provide high level framework elements that make very easy to use this now standard solution.

The "best way" always depend on the specific application domain and therefore the choices done at this level always depend on two opposite forces:

•Make it general, so that it applies to a wider application domain

•Make it very specific, so that it fits very well and easy into a problem

This always leads to necessary compromises.

In the design of ACS we have been and are driven by the following considerations:

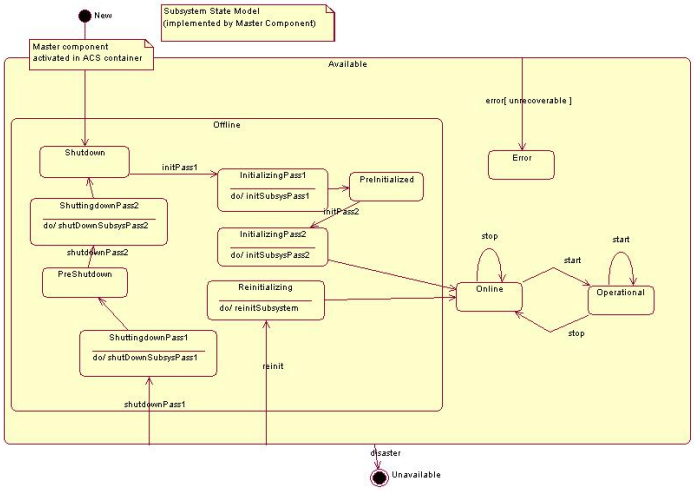
•Our domain is the whole Observatory software. Not just the Control System or the Data Reduction Pipeline. We need to satisfy the needs of all our stakeholders.

•Sometimes the requirements in the sub-domains are very different and there is no "one size fits all" solution. Then we have to provide alternative solutions, but mutually coherent and compatible.

•Some cases are really "special". We cannot completely close the door. We have to allow going via special paths when justified.

In the next pages we will discuss some packages in the ACS high level framework that allow developers to write in an easier way and with better integration and maintainability applications for our "observatory domain". Depending on the time available we can analyze more or less of these examples and discuss them.

Master Component and State Machines



The ALMA architecture is based on subsystems that are “running” independently.

This is a very common architecture and appears in many other scientific (and industrial) facilities.

The subsystems are managed (started, stopped, checked for health) by an high level coordination application.

This applications does not want to know about the peculiarities of each subsystem and want to be able to tread them all in the same way (well, there are always exceptions, but forget about them for the time being).

It is therefore reasonable to define a standard interface that each subsystem has to implement and expose to the administrator.

The most natural type of interface for such purpose is a state machine and therefore we have specified subsystem level state machine and implemented it in a Master Component.

This is a big help in getting a system that is easy to integrate even if the subsystems are developed by completely independent teams, as it is the case for large international collaborations.

The overall Technical Architecture is specified:

•The system is divided in sub-systems

•Each sub-system has a Master Component implementing a standard State Machine

•This Master Component coordinates the activity of the other Components making up the subsystem

•The Administrator Component deals in a standard way with all the subsystems

Components implementing state machines are very useful also in many other places, but strangely enough we have not been able to find any general state machine implementation in the free source world that we could use for our implementation.

Therefore we have implemented a general solution based on code generation directly from UML models.

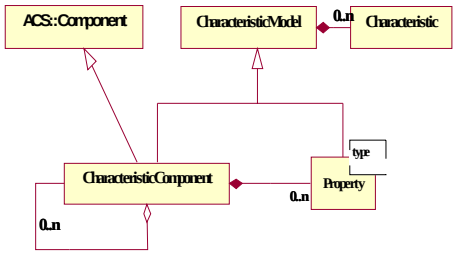
Actually, we have realized in the last year that there are very good reasons and even good tools to generate code from UML Models.

This would even more strengthen the separation between technical and functional concerns by letting application developers design their component and data entities in UML using standard commercial tools and have all the code generated up to filling in the body of the functional operations.

Task of the technical team is then the implementation of suitable code generators.

Devices:
Component-Property-Characteristics pattern

- **(Characteristic) Component**: base class for any physical/logical Device (e.g. temperature sensor, motor)
- Each Component has **Properties** (e.g. status value, position - control/monitor points)
- **Characteristics** of Components and Properties (Static data in **Configuration DB**, e.g. units, ranges, default values)
- **ABeans/GUIs**



On the Control System side the concept of a Software Device representing physical or logical devices of the system such as antenna mount, antenna control unit, correlator, etc is very common.

It is therefore useful to take a formal design pattern for this model and implement it.

We have taken the *Characteristic Component-Property-Characteristics* pattern.

The Device itself is mapped on a Component in our Component/Container model.

Each *Characteristic Component* implements operations and is further composed of *Properties (representing monitor and control pointes).*

A *Characteristic Component* can also contain references to other *Components* to build hierarchies.

Both *Characteristic Components* and *Properties* have specific *Characteristics*, e.g. a *Property* has a minimum, a maximum, units…. The common behaviour of *Characteristic Component* and *Property* has been factorized in the *Characteristic Model* common base class.

Values of the *Properties* are updated asynchronously by means of monitor objects.

While there are in principle an infinite number of Component types, for example one for each physical controlled device, there are very few different *Property* types

Underneath this high level pattern, design patterns for synchronous and asynchronous value retrieval/setting, monitoring and archiving or alarms are part of the Property definition

The implementation of this pattern provides once more a clear path for the Technical Architecture of the Control System.
The developers responsible for the implementation of the control system have to:

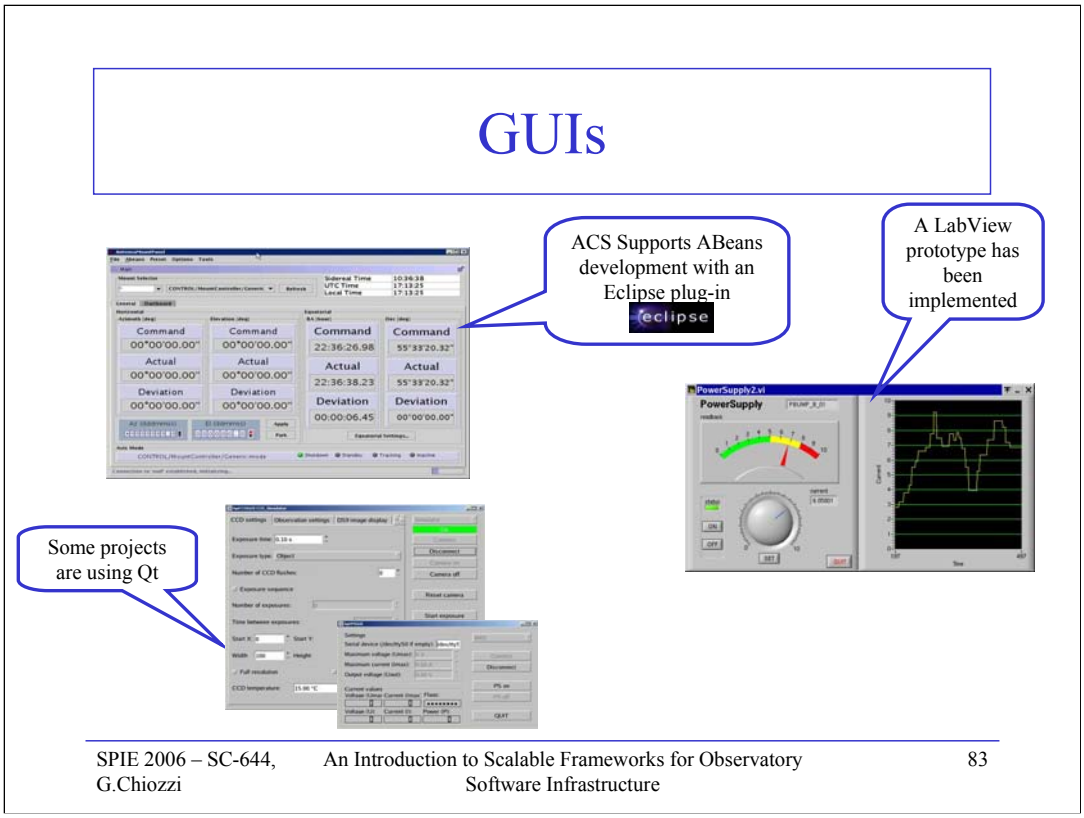
•Identify the hierarchy of logical and physical devices

•Identify the operations allowed on each device and the monitor and control points. This is the IDL interface of the devices.

•Implement the code for the operations.

•Implement the hardware access layer (using the Bridge pattern) to connect the properties with the actual hardware

The framework provides them with standardized configuration and deployment means, automatic monitoring for telemetry and many other facilities.



The Component/Property/Characteristic pattern enables for example to use standard GUI builders to develop application and engineering panels in a comparably easier way.

An example is the generic ABeans GUI building framework that has been adopted by ACS and by other Control System frameworks.

Abeans are Java Beans that are aware of the Component-Property-Characteristic paradigm.

Using Abeans it is possible to use any Java Visual Builder (like Sun Java Netbeans, or upcoming Eclipse extensions) to visually build user interfaces for Components.

A set of graphical Java Beans implements the most useful widgets for the development of Control System applications, aware of the concepts of Components, Properties and Characteristics.

At the same time a code generator produces Java Beans based on the IDL interface of ACS Components. These Beans are therefore automatically integrated in any Visual Builder.

For example, a Gauge widget can be associated to an ACS Property to display the value, draw trend plots and configure automatically itself based on the Characteristics stored in the Configuration Database.

Other projects are using the Qt libraries from C++ or Python code, preferring them to Java libraries.

A prototype of interface between ACS and LabVIEW has been also implemented by two projects using ACS (see appendix).

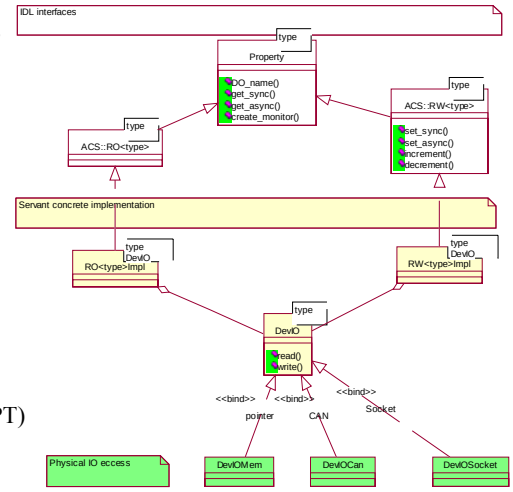
This is a very interesting approach, because it allows the hardware and electronics engineers (often accustomed to use LabVIEW) to build they own control panels.

Property Servant implementation

The DevIO bridge pattern decouples Properties from HW.

DevIO implementations available:

- Memory location (ACS defaults implementation)
- CAN bus access (ALMA)
- Socket generic interface (APEX)
- RS232 (OAN)
- PC Joystick (HPT)
- Webcam (HPT)
- CCD cameras (FBIG, Finger Lake) (HPT)
- Heidenan Encoder board IK220 (HPT)
- Motor Control Board (HPT)
- CCS Real time database (VLT)



This slide shows the decoupling between the high level concept of Property and the access to the actual hardware.

While the implementation of Properties is completely general, access to hardware is delegated to a simple DevIO class according to the Bridge design pattern.

The DevIO class needs to implement read and write functions to access the hardware.

When a property is instantiated, it receives a proper DevIO implentation that enable it to retrieve and store (if writable) values in the hardware.

There are already many DevIO implementation available, some developed for ALMA and some developed from other projects:

•Memory location (ACS defaults implementation)

•CAN bus access (ALMA)

•Socket generic interface (APEX)

•RS232 (OAN)

•PC Joystick (HPT)

•Webcam (HPT)

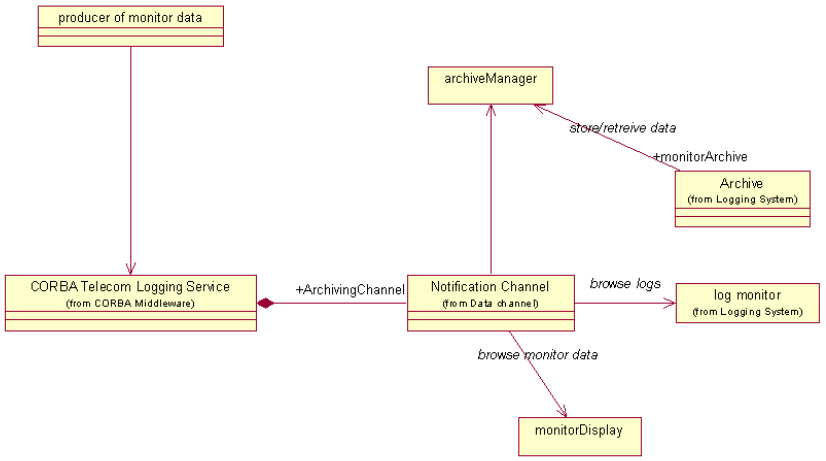
•CCD cameras (FBIG, Finger Lake) (HPT)

•Heidenan Encoder board IK220 (HPT)

•Motor Control Board (HPT)

•CCS Real time database (VLT)

ACS Monitor Archiving system



All Control Systems need to provide telemetry data to monitoring clients and send it to an archive for offline analysis.

Since we map monitor points into Properties, we can implement a generic monitoring system in the properties as a standard service for all developers.

Archiving is enabled/disabled and configured on per-property basis. ACS Properties publish their value on a specific ArchivingChannel notification channel as structured events, by using the ACS Logging System.

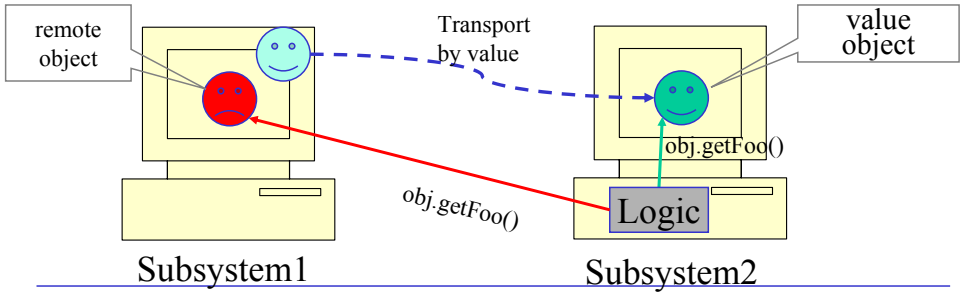
The parameters for data publishing are defined in the Configuration Database and it is possible to specify, on a per–Property base:

- Archive priority
- Max time interval between two archive submissions
- Min time interval between two archive submissions
- Min value change that forces an archive submission

Entity data: XML value objects

Why Value Objects?

- Less remote calls -> Better performance
- Run-time independence between subsystems increases reliability



The Acs Architecture requires the ability to send Entity Data as Value Objects from one subsystem to another or to retrieve Entity Data from the Archive Subsystem and use it locally, until it is time to commit the changes in the archive. Thie applies, for example, to Persistent Objects, like “User”, “ObservingProject”, “CorrelatorConfig”

XML as the Format for Value Objects

We have chosen to use XML as the format to be used for the serialization of Value Objects.

Using CORBA and different programming languages, the only alternative would have been **CORBA valuetype**.

XML serialization has the following advantages over CORBA valuetype:

• XML is suitable also for Data Persistence

• XML is usable also on transport protocols different from CORBA, like http or email.

• XML Schema allows stronger typed declarations with respect to IDL and allows to use versatile automatic validation tools

• CORBA valuetype is not supported by many ORBs

• XML can be easily manipulated “by hand” or using many publicly available tools. This is particularly important for a step-by-step development of the software, where advanced manipulation tools will be developed in later phases of the project.

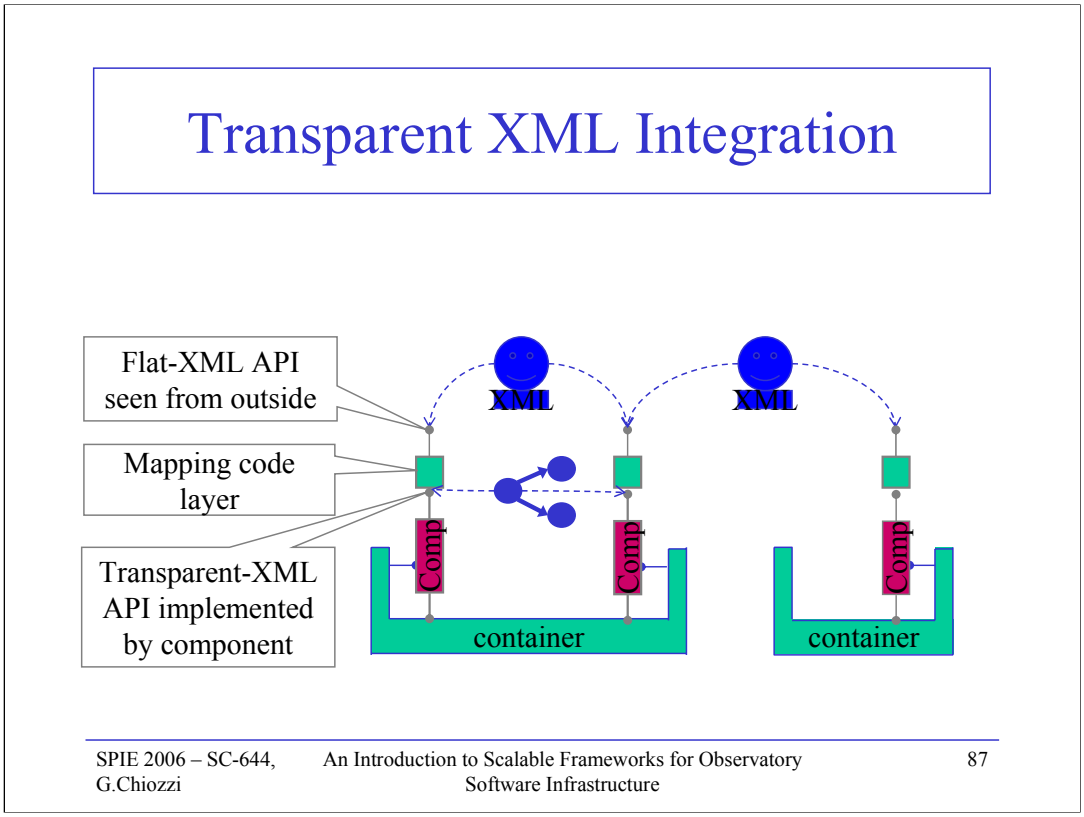
With the advantages of XML data added to CORBA's data types, ACS lets the developer make the best choice for every parameter in every component method in the IDL:

•To send simple data by value, the built-in data types of CORBA can be used, with the advantage of efficient binary transport;

•For more complex, usually hierarchical data, the data definition can be provided outside of the IDL in an XML schema file, and has to be referenced in the IDL. This option is expected to be chosen for nested structures such as an Observing Project and its Scheduling Blocks, where the size is of the order of a few 100 kB.

XML transport is realized in IDL using an ACS-defined CORBA struct as a vehicle; it contains a string field for the serialized XML, plus complementary administration meta data, such as a unique ID.

As a rule of thumb, large data structures should be broken up into smaller groups, each described by its own XML schema. For example, ALMA's Observing Project, the Proposal, and the Scheduling Blocks are each modeled separately. A balance must be found between quickly accessing large parts of the data tree in one call, and not transporting too much data at a time when only a part of it is needed.



XML transport is realized in IDL using an ACS-defined CORBA struct as a vehicle; it contains a string field for the serialized XML, plus complementary administration meta data, such as a unique ID.

XML binding frameworks are used to generate native language binding classes from XML schemas.

Binding class instances can form in-memory representations of any XML document that complies with the schemas used for the code generation. Binding classes offer static methods to instantiate objects from XML, and methods to serialize binding objects to XML. They also allow validation against the schema.

Applications are written against the type-safe accessor and manipulator methods of the binding classes.

Every component implements one interface that is defined in CORBA IDL; the methods of that interface may use XML data as string parameters or return values.

However, without additional support, both the client and the component implementation would send or receive XML data as strings rather than as

trees of binding class instances, even if they use type-safe binding classes inside their implementations. ACS and the ORB could only guarantee that a valid string is returned. At both ends of a remote call, the applications would be taking on the burden of performing their own marshaling and unmarshaling.

ACS resolves this problem by integrating transparent marshaling and unmarshaling of XML binding classes in the container:

• The “XML" component interface is the IDL interface seen from outside the container. XML data used as IDL method parameters appears as plain CORBA strings. The Java container provides an implementation of this interface.

•The “transparent-XML" interface is a Java interface which ACS generates using a custom IDL compiler. It resembles the XML interface, except that Java binding classes are substituted for XML-strings. The component implements this interface and receives an incoming XML transfer object as a tree of Java binding classes; likewise, it returns binding classes wherever XML is expected on the IDL level.

•The mapper class is part of the container and unmarshals parameters from XML strings to binding classes and back.

•A component that uses another component can retrieve from the container a transparent-XML view of the other component. Thus both in implementing its own interface and using other components, a component is provided with the “illusion" of sending around Java binding classes. In fact, for calls between collocated components, the container is free to shortcut XML serialization.

Code generation

- code generation in ACS widely used:
  - uniform framework throughout different modules
  - reduction of routine tasks
  - avoiding typing errors
  - better focusing on functionality of application
- Model Driven Architecture?
  - creation of an application (skeleton) from IDL-file or, better, UML

In ACS many repetitive tasks are handled with the help of code generation tools.

For example:

• error system interfaces and implementation classes are generated from an XML specification

•XML binding classes

•documentation

Code generation has many advantages recently finally became available powerful tools to implement efficiently code generation solutions.

In particular we rely a lot on the openArchitectureware code generation framework:

http://www.openarchitectureware.org/

Together with the ALMA High Level Analysis team we think that code generation from UML will be able to relieve the programmers from a lot of code editing, since a big part of the Component's code can be easily generated. This is an important step toward Model Drive Architecture.

acsGenerator

- acsGenerator: let's start form IDL specifications:
  - application skeleton
  - Configuration Database
  - creation of automatic tests (TAT)
  - creation of an engineering GUI (optional)
- contributed to ACS by the HPT team
- written in Python
- use of templates in editable textfiles
- easy to adapt to different styles
- easy to customize (resource files, templates)
- improved maintenance and flexibility
- separation of parsing and code-generation
- can be extended for creation of Python and Java skeleton
- command-line and GUI version

A first step in this direction has been the implementation of an ACS component generation framework that starts from IDL component specifications.

The acsGenerator has been implemented by the HPT team and contributed to the ACS code base and is a very good example of the advantages of using the same software infrastructure in multiple projects. Now the code acsGenerator is used by various projects and is being evaluated to be used by ALMA as well.

Component simulation framework

- Why simulation?
  - Distributed development
  - Features or entire subsystems not yet available
  - Test a subsystem in isolation
- Simulation of Components from IDL interface spec.
- Dumb default or "intelligent" simulation

The functional entities collaborating in an ACS application are the Components.

The interaction between components is based on the IDL interfaces and/or on the notification channel.

No client is ever aware of the actual implementation of a component it interacts with.

There are very good reasons to be able to simulate a complete component:

•In a distributed development the code base cannot be all the time synchronized. Therefore a subsystem team can desire to have a stable simulation of components developed by other subsystems.

•At any intermediate time between releases, some pieces of code contributed by the various subsystems are only partially implemented. Another subsystem might need functionality that is not available yet.

•It also happens that the intermediate code does not perform according to specifications. This might confuse the developer of a subsystem using it. When running tests or when implementing modular regression tests, who is at fault in case of problems?

It is therefore very difficult to get the integrated (but partial) system working.

It is also very difficult to identify the subsystems responsible for bugs and work around them to proceed with the integration tests.

It is therefore much quicker to get the complete system exercised if the capability to fake the missing software functionality is available.

If possible, modular regression tests should only rely on internal code and simulation for external Components.

Due to the fact that only IDL interfaces can be seen by clients of Components and not the actual implementations, the most effective means of simulation for ALMA is at the Component level. That is, it should be possible to specify to the Container that the implementation for a given Component is a simulated Component factory. Also, because of the very nature of CORBA and IDL interfaces, clients using the Component will never know they are not using the real deal. Component implementations are hot-swappable within the ACS framework.

The ACS Component simulator allows developers to configure the behavior of simulated Components in four different ways – completely self-implementing components, configuration files found in the ACS CDB, a GUI, and an API.

The ACS Component Simulator has the following characteristics:

• It is implemented in Python

•Uses the CORBA IDL Interface Repository (IFR), a CORBA service which stores and retrieves IDL, it is possible to accurately create method return values for the developer without their input.

•Can be executed from an interactive Python session. This implies the developer can swap out entire method/attribute implementations with ease.

•Instead of simulating components at the interface level where all component instances of a given IDL type behave identically, we simulate at the named component instance level. This means that each simulated component of a given type can be configured to behave uniquely which is different from the three proposals.

•Using native Python methods, it is possible to dynamically create the implementation of any IDL interface. This implies simulation could indeed occur at the component level without making modifications to the container.

•Using native Python methods, it's possible to read method/attribute return values in the form of XML strings from the ACS CDB.

Summary: High Level Framework

- Identify and implement:
  - High level Technical Architecture blocks
  - Standard domain design patterns
  - Standard paths when many alternatives are available
- Do not rule out special cases

The developers of the system should be confronted only with the choices connected to the functional aspects of the system.

But most frameworks leave still too much freedom of choice because are thought for “any kind of application”.

Developers then risk to get distracted by technical choices.

In a big project, different subsystems might take different paths leading to waste of development resources, duplication of effort and interface problems.

It makes therefore sense to identify and implement, on top of the all purpose framework, solutions that are still general for the domain of application and for the whole observatory, although give a clear path to the developers.

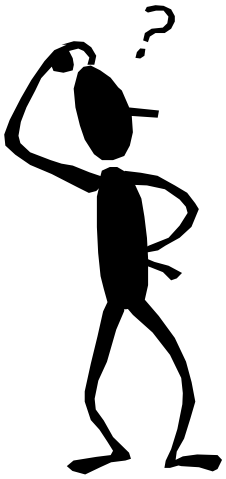
What to do (or to reuse) at this level is really a matter of choice, but there are plenty of examples.

This is also an area where “functional developers” can feel to be strongly limited in their freedom of choice by the “technical team”.

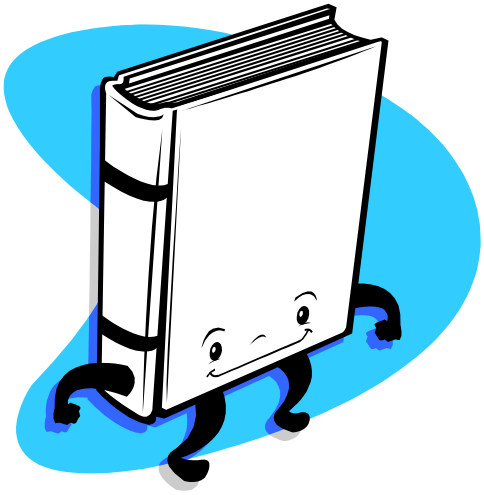
This is often done with a good will and at the advantage of the global project: often it is better a sub-optimal solution working for everybody than many optimal but different solutions that will cost immediately for duplication of effort and in the future for maintenance.

Nevertheless, there are really cases where searching for a “special solution” is not avoidable.

Questions (& Answers)



9 – Development support



Dealing with big projects

- Big projects, big troubles:
  - Ensure that installations are “identical”
  - Integration issues
  - System deployment issues
  - Maintenance issues
  - Personnel turnover

When there is a big project that spans a whole observatory, is developed across many development sites and spans over many years (or combinations of these characteristics) there are many sources of problems:

•The many development machines have to be aligned with the same software at the same level. The framework we have described consist of very many pieces from many sources and it is very easy to encounter incompatibilities between these pieces and the underlying operating system. It is therefore necessary to centralize the definition of the “mix that works” and ensure that everybody gets the right cocktail (possibly being able to check if a configuration is clean or not)

•All the pieces developed in the different sites and by different developers have to come together and be integrated. There should be standard ways of testing the functionality of each single component autonomously and automatically and of integrating them and test them as a unit. If there is an integration team, very often it does not have the knowledge needed to thoroughly test and debug the single components.

•The deployment on the operational system involves many hosts and possibly many sites. Downtime due to deployment problems is very expensive and therefore it is important to have precise deployment and rollback procedures.

•Debugging an maintaining the system can be very expensive.

•One should get the architecture, design and implementation right in the first place. Therefore it is very important to have means to evaluate (or, better, measure) the quality of the work done and to test it.

•When problems will come out or changes will be needed it will become very important to have a system that is homogeneous and understandable. If the same patterns and tools are reused over and over, everybody in the team knows where to puts its hands.

•Factorizing common code in a single place (the framework) allow to “fix one and cure all”

…. Continues on next page

Software Engineering practices

Software Engineering and Quality Assurance activities:

- •Software Process
- •Document Reviews, Format, Templates
- •Development Environment
- •Integration Procedure
- •Coding Standards
- •Code Inspection
- •Configuration Management
- •Testing framework and assessment
- •Change Management

All these troubles can be mitigated by adopting Software Engineering practices.

Balancing the cost of the overhead introduced with the complexity of the project and the benefits that can be reached will dictate up to which point it makes sense to push for formal practices.

But in any case it is counter productive to simply state rules on paper and ask people to follow them.

It is essential to provide tools and support so that the adoption of the practices and their verification is transparent or becomes second nature.

…. continues from previous page

•For a system that will take many years to implement and that will be operated for many years by different people than the developers it is important not to underestimate the problem of personnel turnover. Personal ownership of the code shall be avoided, because if the person disappear the knowledge will be lost and intervening on the code will require expensive reverse engineering. Founding the architecture on standardized patterns and pushing for factorization and reuse (coupled with code review and team rotation) is of great help.

All these considerations typically appear as requirements for the system to be built.

Tools to support SW lifecycle and QA

1. Tools to build and integrate the software (make, ant)
2. Code configuration control tools (CVS)
3. Regression testing infrastructure
4. Problem tracking system to track faults and change requests (Action Remedy)
5. Tools to produce documentation of software (both inline and online – *doxygen*, TWiki)
6. Automated night reporting infrastructure to run regression tests and check standards compliance (ESO NRI) on the complete codebase.
7. CASE tools (UML Modeling, Editors, Quality Control)
8. Tailored standards for most of the process phases and deliverables

And what they translate into for practical usage

Can the Framework help?

- Drive technical architecture choices
- Accretion point for factorization
- Distribution vehicle for:
  - Upgrades
  - Controlled versions of the software
  - SE support tools
  - Development and deployment environment configuration

Can the adoption of an observatory wide software framework help with these issues?

Clearly yes.

As we have said already, first of all the whole system structure becomes much more uniform and consistent, because everybody is pushed to use the same architectural and design pattern.

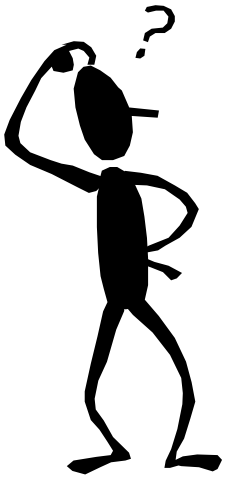
Then new solutions of general usage can be integrated in the framework to be reused by other groups.

But it is also a good idea to integrate in the distribution of the framework also all the tools for software engineering we have described in the previous pages and the configuration of the development environment.

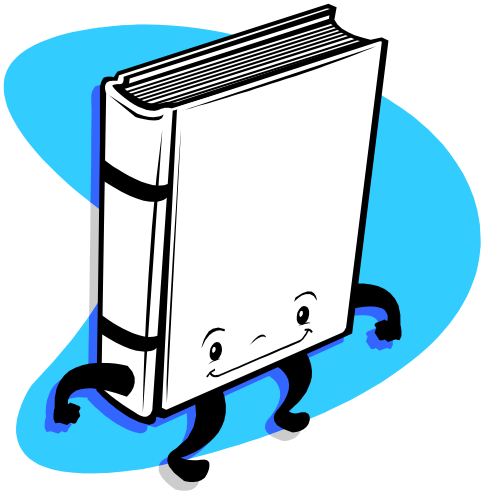
If the installation of the framework on a freshly installed system produces a working environment for development, testing or deployment it is much easy to have reproducible installations.

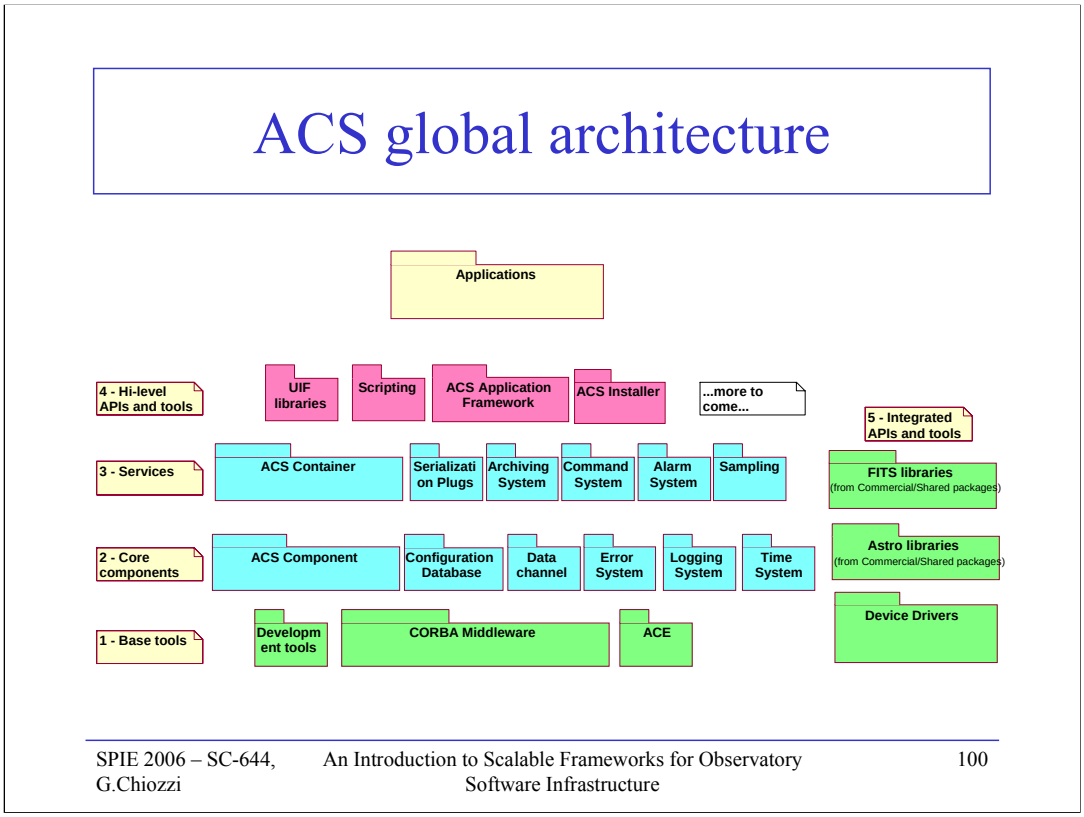
This approach is again a “global gain” paid at the expenses of personal freedom for the developers and therefore it is important to find the right balance based on the characteristics of the team and of the project.

Questions (& Answers)



10 – Wrapping up





The ALMA Common Software is an example of the approach described in the previous pages.

This package diagram is a simplified version of the complete ACS Package Diagram from the Architecture document

The architecture is divided in layers and each layer can use the packages in the same layer and in the layers below.

This allows us to keep under control the dependencies between packages.

An important aspect is that the “base tools” layer is a thick and reliable foundation based on CORBA and other “off the shelf” publicly available tools and software packages.

This includes or defines as pre-requisite for ACS installation:

- A standardized set of development tools (like compilers, Makefile extensions, installation procedures and tools, JUnit and other test support tools, emacs configuration and so on)
- CORBA implementation and services for the different languages
- ACE and other public domain libraries used by ACS and available for application developers.

It is a main objective to use whenever possible readily available packages and not to re-implement services that already exist.

But for each service/package, ACS provides and “interpretation” of the way we want it to be used in the terms of design patterns and support code implementing the design patterns to makes it easy to use our “interpretations”. This reduces the learning curve and makes the code more uniform across the distributed development sites.

In some case there is really no ready made implementation that we can use and therefore we provide our own implementation, but keeping an eye at the OMG specifications.

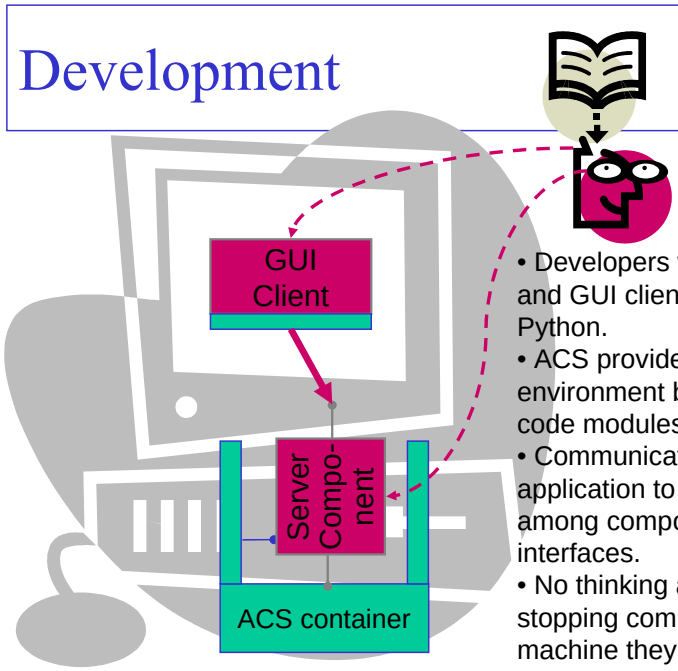
We also recognize that this approach heavily constrains the freedom of the developers to choose between the different possibilities of using a service; therefore we allow to “drill a hole” in the upper ACS layers and use directly the underlying layers when this is justified by a real need.

Typically such holes are later on closed again by incorporating the new solution into ACS itself.

An example of this is the ACS “Data channel” that wraps the CORBA Notification Channel to provide very easy access to the push-push model.

The ACS distribution is used also to package and distribute other APIs and tools that are not part of ACS and that are not used by ACS, but are used by a number of ALMA subsystems and that is therefore convenient to distribute as one single entity together with ACS. This can include for example FITS libraries, Astronomical Calculation libraries or device drivers.

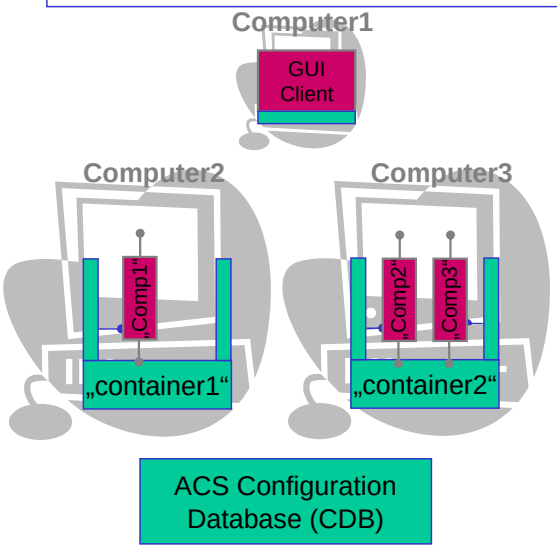
Development



- Developers write Components and GUI clients in Java, C++, or Python.
- ACS provides an integrated build environment based on application code modules.
- Communication from an application to a component, and among components, uses ACS interfaces.
- No thinking about starting and stopping components, or on which machine they should run later.

.

Deployment



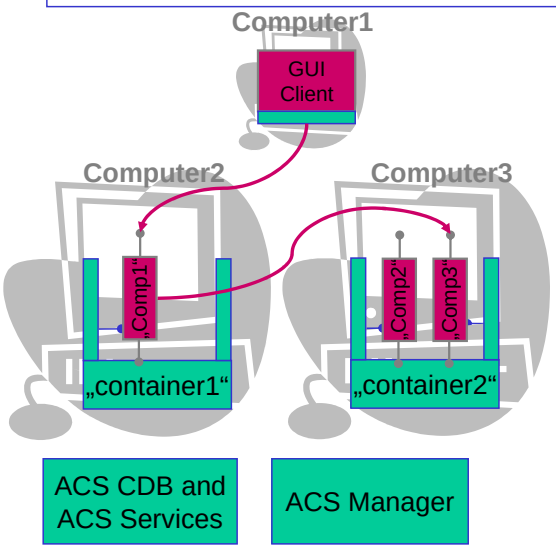
- One or more containers get assigned to each computer.
- Components get assigned to containers.
  - This location information is stored centrally in the Configuration Database (CDB).
- Other configuration data for containers and components is also stored in the CDB.
- Different deployments for unit tests, system tests, and various stages of the productive system.

Details on container location information and container startup:

• for the system to work, it is good enough to start containers by hand on any machine. They dynamically add themselves. This is only done for tests though.

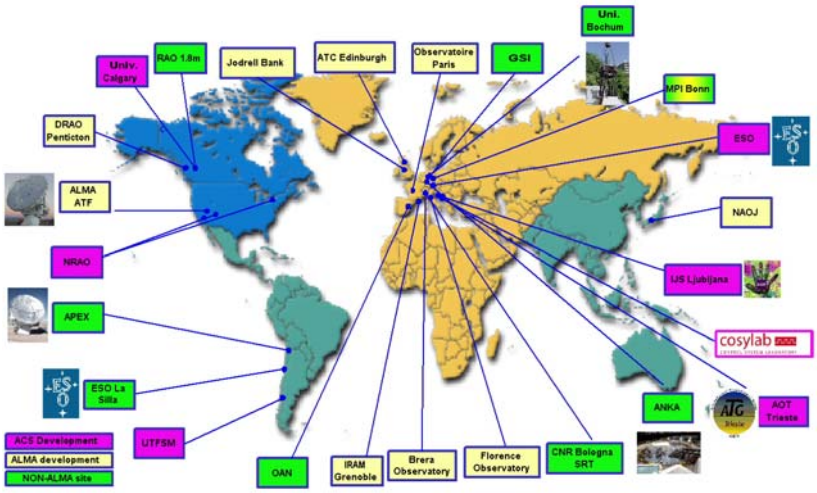
• In the real ALMA, the central starter application “Executive” starts containers on various machines, before any application software gets run. Exec maintains a configuration file that assigns containers to machines.

Runtime



- ACS containers start and stop components (“lifecycle management”)
- containers provide components and clients with references to other components.
- the “manager” is the central intelligence point that keeps the system together. Components never see it directly.

ACS installations and projects



ACS is fully based on public domain software and is in an advanced development phase.

The development is backed by a big project (ALMA) but there is interest in a wider community of users.

Therefore it has a good potential for being adopted by other projects.

ACS is installed in all ALMA development sites, but it is also used or under evaluation by a number of other projects.

Quantifying the advantages

- The feeling is good
- Can we measure the impact of adopting such a framework?
  - Very difficult: all numbers are debatable
  - But we have some good example

All the arguments that have been given in favor of adopting a common software infrastructure for the whole observatory produce the good feeling that it is an investment worth to do.

But can we really have some measure of the advantages?

This is very difficult and numbers here are debatable.

The only real way to get a reliable measure would be to have a big project already done and measured and re-do it from scratch adopting a framework.

For sure there are some successful project that have been done to a great extent in this way: the VLT is an example of well performing observatory whose software has been developed within schedule based on a framework common to a big part of the observatory.

But we cannot really measure:

- Technology is advancing so fast that we cannot really reliably compare any two software works done at 10 years distance
- Nobody has the resources and the interest in developing two parallel systems with and without such a framework.

But there are some partial examples that can be measured and can be extrapolated.

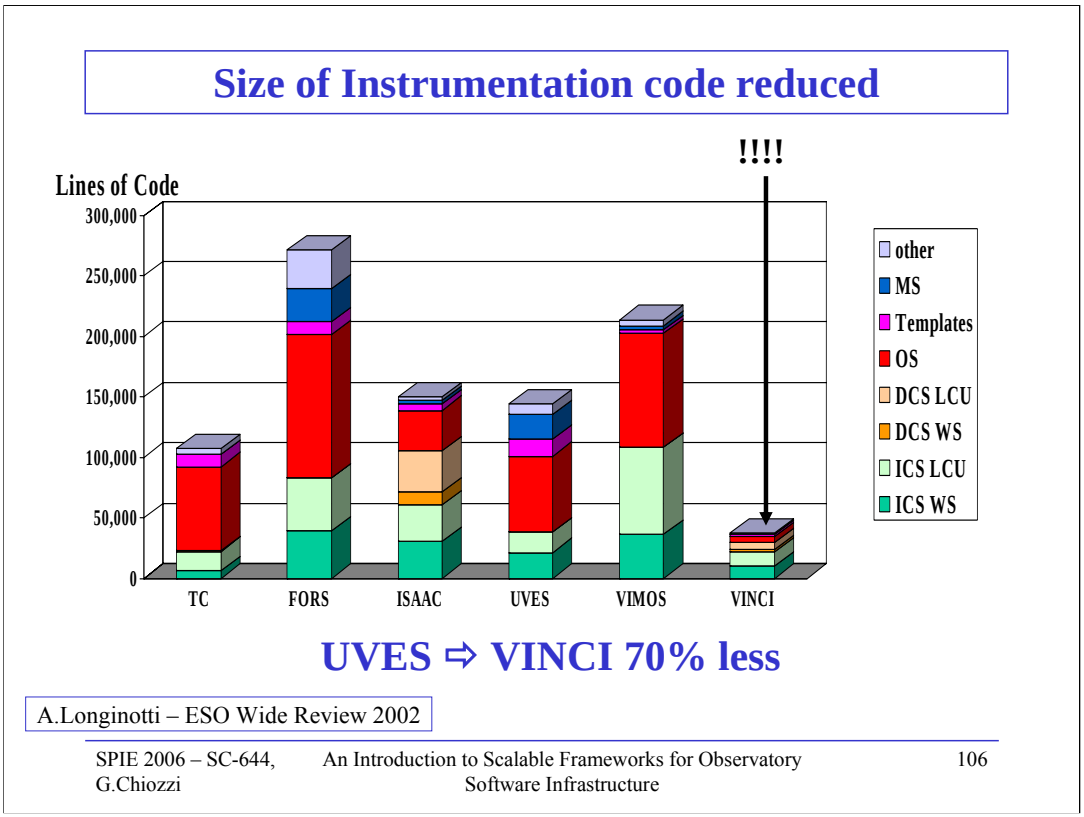
One of these is shown in the next slides and comes from the VLT.

The Control System of the VLT and of its instruments has been developed based on a Common Software.

After having developed a number of instruments it has been realized that that there was a big potential for factorizing major parts of the architecture of the instrument in a Common Instrumentation Software.

This has been done extending the VLT Common Software with Instrumentation Software and new instruments have been implemented using this extension.

Therefore we have the possibility of quantitatively compare instruments developed with and without such a common infrastructure.



The VLT INS Common Software has been implemented as a result of the analysis of the first instruments.

In particular, a number of these instruments has been implemented directly at ESO or with the direct participation of ESO developer.
In any case, for every instrument there is a least an ESO software engineer, from the instrumentation group, assigned to follow up the consortia.
More the software for the first instrument (the TC, test camera) has been always given around as a template/example.

This means that we have always taken care of making sure there were always a lot of similarities, in particular in the architecture, between the various instruments.

The development of the INS has made available an "implementation" for these similarities that has given immediate benefits.
More over, there has been a continuous feedback between the INS and VINCI developers, so that "missing things" have been developed immediately and put directly in the INS distribution.

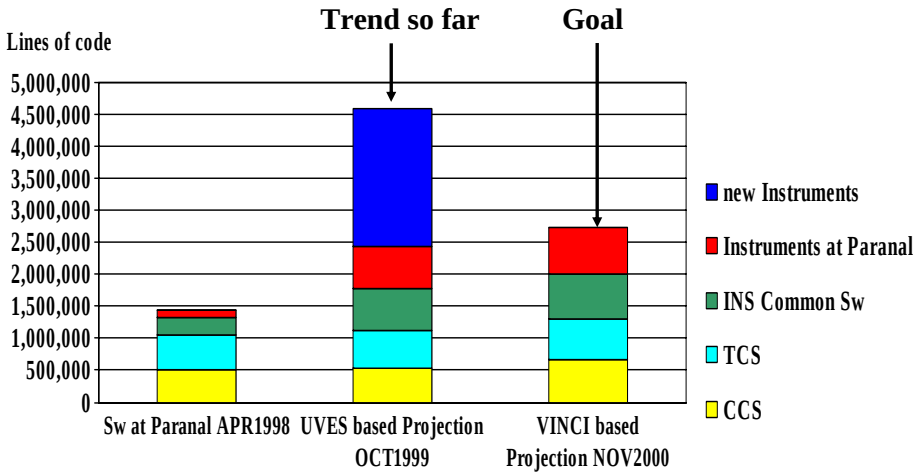
If you add up the INS code and the VINCI code, you will therefore probably get in total more of the UVES lines of code (I do not have the numbers).
But the INS code is then available for reuse for all instruments and its cost gets amortized back with time.

I think that the best way to realize how it works is to look at the set of functionalities described in the documents.
All documents abouts the INS Common Software are available at this link:

http://www.eso.org/projects/vlt/sw-dev/wwwdoc/JAN2006/dockit.html

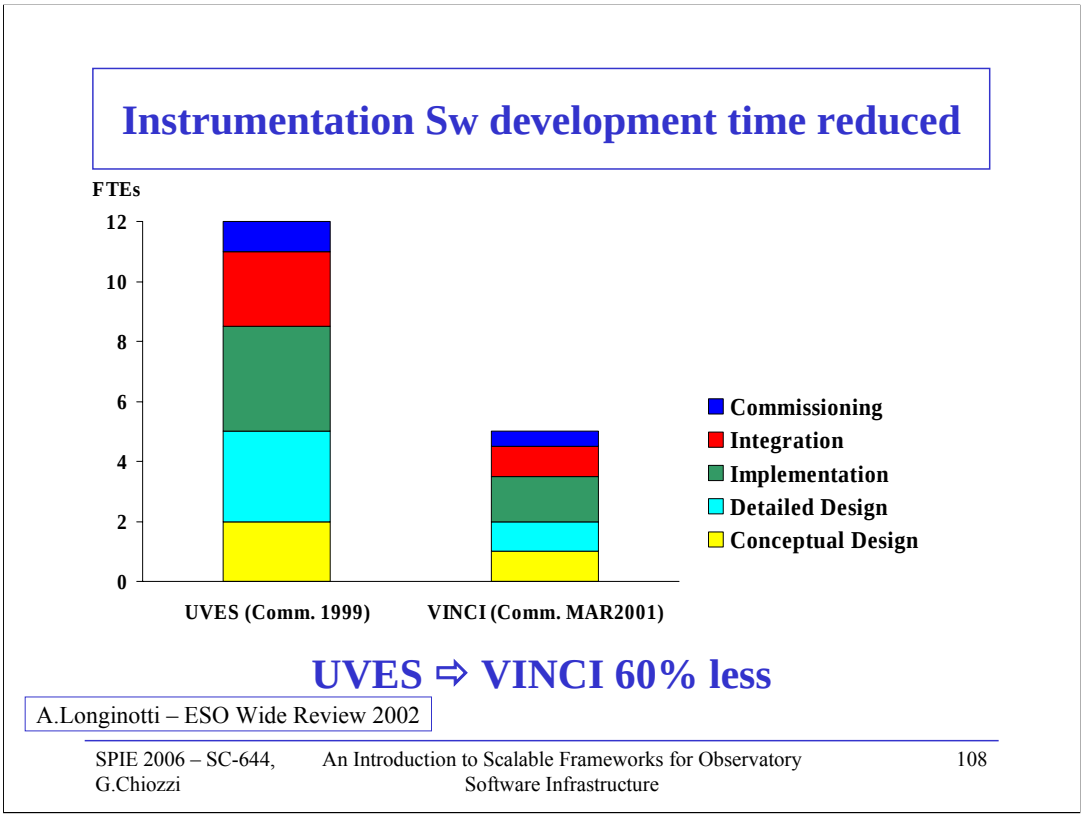
This link contains the whole documentation for the VLT Common Software and includes the Instrumentation Common Software.
Look at volumes 5a, 5b and 5c.

Instrumentation code more maintainable



OCT1999 ⇨ NOV2000 Projection: 60% less

A.Longinotti – ESO Wide Review 2002



As of today, the number of instruments that are using or will be using the Instrument Common Software is 23.

Comparing the data for UVES (that does not use the INS Common Software) and VINCI (that uses it) we can estimate a gain of 7 FTEs in the development phase of the project.

If we extrapolate over the development of the 23 instruments and take into account the cost of developing the INS Common Software itself (~25 FTEs), we get a total gain for the astronomical community of:

23*7 - 25 = 136 FTEs

Then we have to count the gain in maintenance.

As shown, the INS Common Software allows to reduce of two thirds the lines of application code.

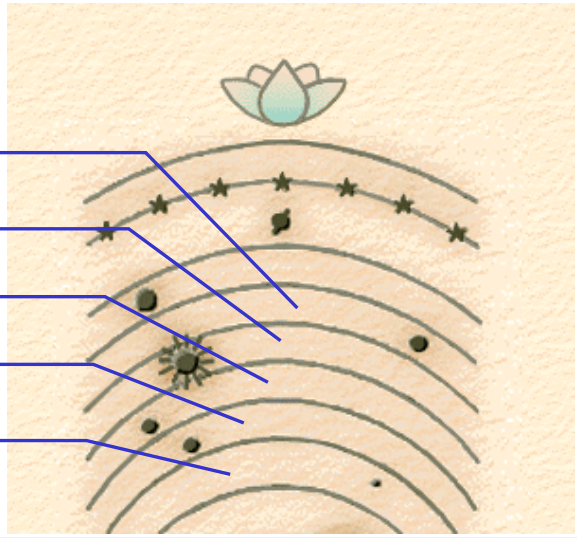
Based on the current situation at Paranal with a mix of instruments with and without INS Common Software and on the resources allocated to software maintenance for the instruments, it is reasonable to estimate a gain of 2.2 FTEs/year.

The maintenance of the INS itself is currently of 1.2 FTEs/year.

Therefore we can round the total gain to something like:

- 135 + (1 * observatory life) FTEs

The path to Heaven?



Let's summarize the steps that have led us to the definition of the elements needed by a framework that could be used for the whole software infrastructure of an observatory.

The elements are listed down up in the order in which we have encountered them following our logical thread.

Who should try this path?

- New big projects:
  - Select a core technology
  - Assign a technical team to the development of the high level framework
- New small projects
  - Select a recent but stable solution (ACS like?)
  - Collaborations
- Upgrade projects
  - Select a recent but stable solution
  - Introduce it in selected areas and let it percolate

Who should try the path of adopting an observatory wide framework like the one described?

I think that every project would find big gains but using different approaches.

A new big project cannot probably avoid to adopt such a solution.

To get the better results, a technical architecture team has to be established.

The team has to select a core technology among the palette of currently available recent but mature choices (avoiding risky cutting hedge technology). Starting from scratch does not make sense.

Then the technical team has to work on defining and implementing the high level framework, trying mainly to do a good work of integration of existing solutions and implementing only what is really necessary.

A new small project should fully adopt an existing solution built by a big project or by a collaboration, trying to behave somehow as a subsystem of the collaboration.

This would allow a very fast startup, with extremely rapid progress on the solid ground of proven solutions.

A small team should concentrate the effort on the functional aspects and not on the technical framework. Nevertheless some specific technical development will be very likely necessary, because each project has some very specific requirements. This work could be done in the form of collaboration with the big project.

What about already existing systems that need to be refurbished and upgraded?

There are many around and in most cases the owners cannot afford to put the resources for developing a completely new system.

But hardware becomes obsolete and cannot be replaced, software maintenance becomes expensive and localized interventions are unavoidable.

In such a case it will be probably most cost effective to take a recent but stable complete solution, just like in the case of small projects. Then apply the solution to the critical parts, for example replacing subsystems whose hardware must be replaced. Or to the parts that have proven weaker and harder to maintain. Then build bridges between the old and new system to allow them to interoperate.

If the system to be upgraded is big the development team can probably give an important contribution in form of ideas and collaborations for the development of new high level framework features to the team developing the adopted framework.

Contour conditions

- It is not easy to adopt a framework in a project.
  - Different background, cultures, experience.
- What contour conditions are necessary?
  - Small motivated team

  Or
  - Strong management and control procedures

Experience shows that it is not easy to get accepted by the developers the introduction of a framework like the one described in this course.

The problem is that each developer or group has it own different background, experience and culture.

As said, the framework has the purpose of driving the developers toward narrow but safe technical paths.

Many developers would see this a limitation in their freedom, and this is certainly partially true.

Other would say that they can do the job much better for what they specifically need. And this is also often true.

The problem is that the advantages can be seen much better from above rather than from the perspective of the single developer:

Non optimal solutions traded for uniformity and coherence

Freedom traded for maintainability

Focus on functional work

Therefore the success is bound to one of two contour conditions:

•The project is done by a small motivated development team that is convinced of the advantages of the framework solution and can push it up

•A strong management imposes the solution to the whole team and establishes control procedures until the project is sufficiently advanced that the gains have become clear to everybody.

Bibliography

- ALMA/ACS Papers in this conference
- ACS Web Page: http://www.eso.org/projects/alma/develop/acs/
- IT Architectures and Middleware, C.Britton, Addison Wesley
- CORBA/OMG web page: http://www.corba.org/
- D.C.Schmidt and TAO web page: http://www.cs.wustl.edu/~schmidt/TAO.html

•This conference contains a number of papers with more details on ACS. You can look in particular at the following papers:

- •Application development using the ALMA Common Software [6274-06]
- •Integrating the CERN Laser Alarm System with the Alma Common Software [6274-07]
- •Bulk Data Transfer Distributer: a high performance multicast model in ALMA ACS [6274-54]

•ACS Web Page: http://www.eso.org/projects/alma/develop/acs/
The ACS Web Page contains a lot of documentation, a detailed architecture description and references to other papers and documents.

• C.Britton, *IT Architectures and Middleware*, Addison Wesley, 2001 ISBN 0-201-70907-4
This is a very interesting (and reasonably thin!) book focusing on requirements and principles of distributed systems, offering an overview of middleware technology alternatives.

•CORBA/OMG web page: http://www.omg.org/
The OMG web page is the starting point to find the CORBA specifications, although what can be found there is too superficial or too detailed for a useful introduction and startup. Better to look in other pages or books.

•D.C.Schmidt and TAO web page: http://www.cs.wustl.edu/~schmidt/TAO.html
Page full of papers on distributed design patterns, CORBA design, high performance and real time distributed systems.D.C.Schmidt is one of the real gurus of the field

•M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns - Patterns for Enterprise, Realtime and Internet Middleware*, Wiley & Sons, to be published in 2004
This very good book describes the most important patterns used in Object Middleware and compares CORBA, .NET and WebServices.

•M.Henning, S.Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, ISBN: 0-201-37927-9
Like the Bible: very old, but still the essential one.

•Communications of the ACM, October 1998. Special issue on CORBA
Old but very interesting collection of introductory papers on CORBA

Acknowledgments

- The material for the course comes from the experience of:
  - ALMA project and in particular the whole ACS team
  - VLT project
  - ESO Software engineering team, that works across projects.
- Collaboration and lots of discussions with other projects and in particular S.Wampler, B.Goodrich, K.Gillies
- Many ideas for the middleware presentation come from web presentations and in particular from the ACE/TAO web page

Thanks to everybody for ideas, slides and discussions

The material for this course comes from more than 10 years of experience in the development of Common Software but, most important, of discussions with all people involved in developing and using this software.

All this people has therefore given an important contribution and many have also provided slides or ideas for slides.

First came the VLT project, where in particular K.Wirenstrand, R.Karban, A.Longinotti have to be thanked for the past work together and for the discussions we have all the time to compare VLT, ALMA and to see how the software world is evolving.

In the ALMA project everybody has shaped a piece of ACS, but a particular thank for the discussions and slides goes to H.Sommer, J.Schwarz, A.Farris, M.Voelter and the other members of the ACS team. Many slides about ACS come from ACS presentations, papers and courseware prepared by many ACS team members. All these presentations are available form the ACS web page.

The collaboration with M.Plesko and the Joseph Stephan Institute in Ljubljiana for the design and development of ACS is also of great importance.

The definition of processes and standards is essential for the success in the usage of a software infrastructure. Therefore I value as very important the collaboration with the Software Engineering team in ESO (in particular M.Zamparelli and G.Filippi).

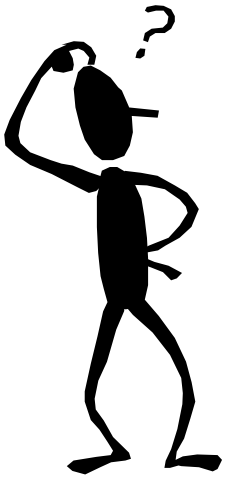
In the last few year the contribution for the other projects has been extremely important, both in terms of feedback and active contribution. Some slides are also derived from presentations given at the ACS workshops by the teams using ACS and contributing “from outside” to the ACS development. Also in this case the original presentations can be found in the ACS web page.

The architecture of the software for observatories is converging toward a common model. This is also thanks to the many discussions and collaboration between the people involved. I want to mention here in particular Steve Wampler (ATST), Bret Goodrich (ATST), Kim Gillies (Gemini, TMT) and Al Conrad (Keck).

Many ideas for the middleware slides come from presentations on the web (cited in the bibliography).

In particular the ACE/TAO web page provides plenty of excellent material and good inspiration.

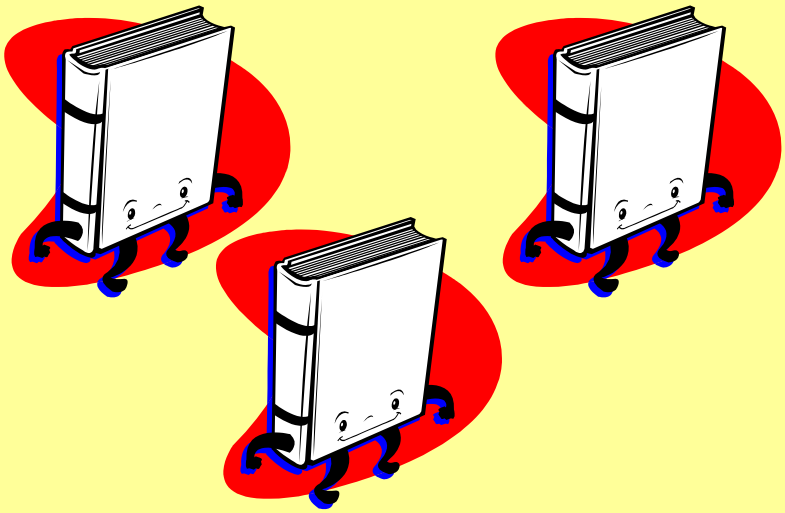
Questions (& Answers)



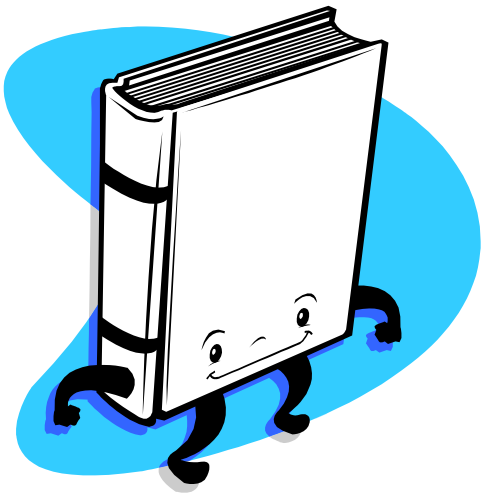
Course Evaluation

Take your time to properly fill in the course evaluation

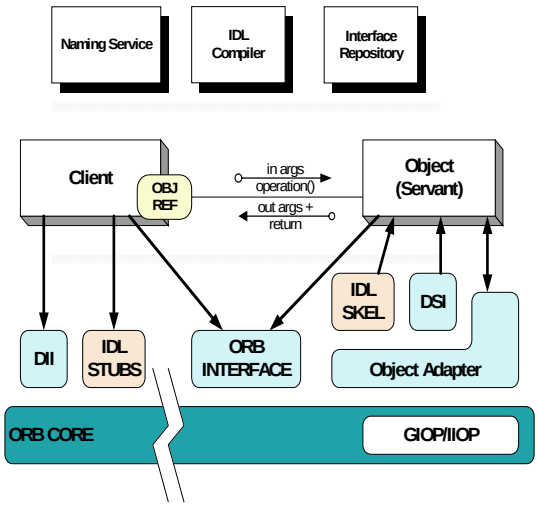
Appendix - Additional information



CORBA details



Overview of CORBA



- It simplifies development of distributed applications by automating/encapsulating
  - Object location
  - Connection & memory mgmt
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency
  - Security

This slide summarizes the overall architecture of the core CORBA.

From the logical point of view, a ***client*** get hold of the ***object reference*** of an Object it wants to talk to, for example using the ***Naming Service***, and then can invoke its operations.

The interface of the object is known to the Client via the ***IDL interface*** published by the object.

Under the hoods, the ***Object Request Broker (ORB)*** transports a client request to a remote object an returns the result. It is typically implemented as a set of client and server side libraries.

Interoperability is warranted by the ***General Inter-ORB Protocol (GIOP)*** and by its TCP/IP incarnation called ***IIOP***.

All vendors are bound to support IIOP but can also implement their own protocol, for example for performance optimization or to exploit specific hardware like ATM networks. There is also a standard secure protocol based on SSH. This allows to exploit network capabilities transparently do the application developers.

On the ***servant*** side the Object Adapter provides the environment in which servants live. In particular it takes care for:

•Mapping of object references into implementation

•Object life cycle

•Threading policy

Compiled interfaces are provided by the ***stubs*** and ***skeletons*** generated by the ***IDL compilers*** (more on this later).

Interpretative interfaces are handled through:

•***Interface Repository***. Repository of the IDL interfaces known to the system. Used for language independent introspection.

•***Dynamic Invocation Interface (DII)*** used on the client side to dynamically generate calls to object operations. Necessary for generic applications and for the implementation of CORBA inside interpreted languages.

•***Dynamic Skeleton Interface (DSI)*** used on the servant side to dynamically implement objects that incarnate a given IDL interface. Necessary for example to implement generic servants like protocol converters or object-to-relational database interfaces.

CORBA server characteristics

- When we say "server" we usually mean server process, not server machine
- One or more CORBA server processes may be running on a machine
- Each CORBA server process may contain one or more CORBA object instances, i.e. "servants", of one or more CORBA interfaces
- A CORBA server process does not have to be "heavyweight"
  - e.g., a Java applet can be a CORBA server
- Clients always talk explicitly to *servants*, and not to *servers.*

When we consider the traditional client-server model, we think of a "client process" requesting a service from a "server process" (or, sometimes, a server machine).

Obviously, also with CORBA the communicating entities are processes running on distributed hosts, but the communication abstraction is higher level.

A CORBA process providing a service to a client contains one or more CORBA object instances, called "servants".

Each servant implement one or more CORBA IDL interfaces and clients do address and communicate explicitly with the servants.

The "server" is only the process inside which the "servant" lives and the client is not aware of that.

Clients always talk explicitly to the servants using the object reference in a fully object oriented model.

Deployment of "servants" in "servers" can be dealt with in a way completely transparent both to clients and servants themselves, as we will see later on.

As we have seen and we will see with more details later, CORBA also uses the term "service" to denote fundamental, almost system-level services to OO applications and their components. Services are specified by means of interfaces, implemented by "servants" and deployed within "servers".

Therefore, make sure to keep always in mind the difference between:

•Servant

•Server

•Service

Interoperable Object Reference (IOR)

- An Interoperable Object Reference is the distributed computing equivalent of a C++ pointer:
  - An IOR uniquely identifies one object instance
  - IORs can be uniquely mapped into a string and back for easy and portable storage.
- An IOR contains:
  - A fixed object key with the fully qualified interface name and an instance identifier
  - Transient information such as the host and port of its server
- An IOR can be persistent
  - Some CORBA objects are transient, short-lived and used by only one client
  - But CORBA objects can be shared and long-lived
- CORBA objects can be relocated
  - The fixed object key of an object reference does not include the object's location
  - CORBA objects may be relocated at admin time or runtime
  - ORB implementations may support the relocation transparently
- CORBA supports replicated objects

Servants are addressed by means of their object references or, more specifically, by their Interoperable Object Reference (IOR).

An IOR uniquely identifies one object instance, I.e. it allows to locate the object in the network and identify what interface it implements.

Interoperability is warranted by representing IORs as strings that can be easily transported and used on heterogeneous systems.

CORBA object references can be persistent.

Some CORBA objects are transient, short-lived and used by only one client.

But CORBA objects can be shared and long-lived

business rules and policies decide when to "destroy" an object

IORs can outlive client and even server process life spans.This means that once a client has obtained the IOR for an object, it can continue to use it also after a restart of the server, unlike a normal C++ pointer.

CORBA objects can be relocated

The fixed object key of an object reference does not include the object's location

CORBA objects may be relocated at admin time or runtime

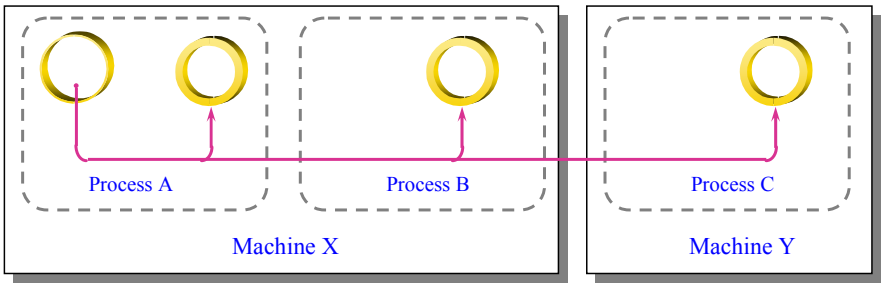
ORB implementations may support the relocation transparently

CORBA supports replicated objects

IORs with the same object key but different locations are considered replicas. The same IOR can contain "alternative solutions" for getting in contact with the desired servant.

The flexibility of the IOR specification is one of the keys to CORBA interoperability and scalability.

CORBA Location Transparency



A CORBA Object can be local to your process, in another process on the same machine, or in another process on another machine

In order to invokes operations on a servant, a client uses the CORBA reference to obtain a local Stub object (I.e. an object in its own language and instantiated in its own process).

Then it makes native language calls to the Stub.

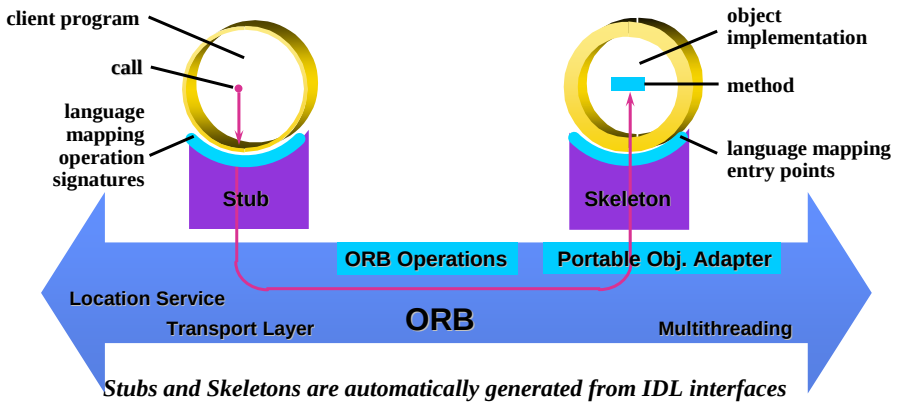
The Stub and the underlying ORB map these calls into calls to the real Servant, but the client is not aware of where the Servant resides.

It can be a local object as well, an object in another process on the same machine or an object in another host.

There is some overhead in this mapping, but good ORB implementations make this overhead minimal and calls to local Servants can be reduced to a few levels of indirection, avoiding any real inter-process communication.

But this transparency makes it much easier to scale systems and optimise performance by re-deploying Servants on separate processes and hosts or repackaging together Clients and Servants that have frequent interactions.

Stubs & Skeletons



Stubs and Skeletons are automatically generated from IDL interfaces

The Stubs and the Skeletons contain all the code needed to interface the user code with underlying ORB and CORBA machinery.

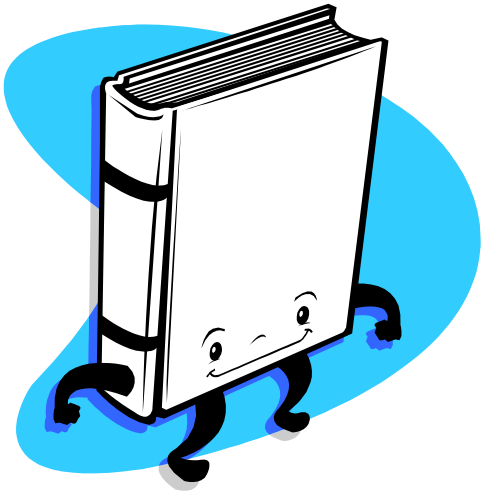
Often the code in the Stubs and Skeletons is ORB dependent and you cannot normally use the code produced by the IDL compiler of one CORBA implementation with the ORB libraries of another one, but this is not important because:

•the interfaces of Stub and Skeleton are based on the formal IDL to language mapping and therefore the user code does not change changing ORB (unless you use vendor extensions)

•the communication between ORBs is also interoperable (unless you use vendor extensions)

This allows to mix and match CORBA implementations based on your needs and to replace them with others.

IDL details



IDL simple data types

- Basic data types similar to C, C++ or Java
  - long, long long, unsigned long, unsigned long long
  - short, unsigned short
  - float, double, long double
  - char, wchar (ISO Unicode)
  - boolean
  - octet (raw data without conversion)
  - any (self-describing variable)

IDL complex data types

- string - sequence of characters - bounded or unbounded
  - string<256> msg // bounded
  - string msg // unbounded
- wstring - sequence of Unicode characters - bounded or unbounded
- sequence - one dimensional array whose members are all of the same type - bounded or unbounded
  - sequence<float, 100> mySeq // bounded
  - sequence<float> mySeq // unbounded

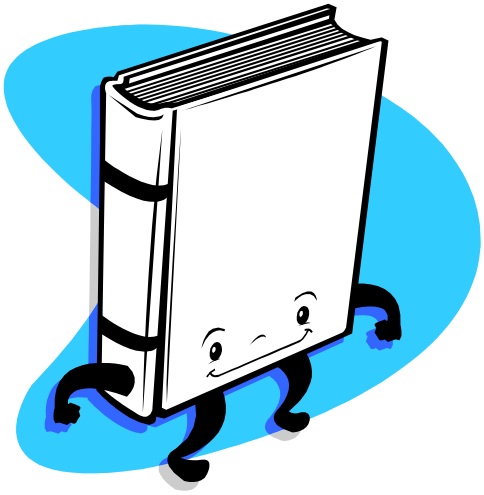
IDL user defined data types

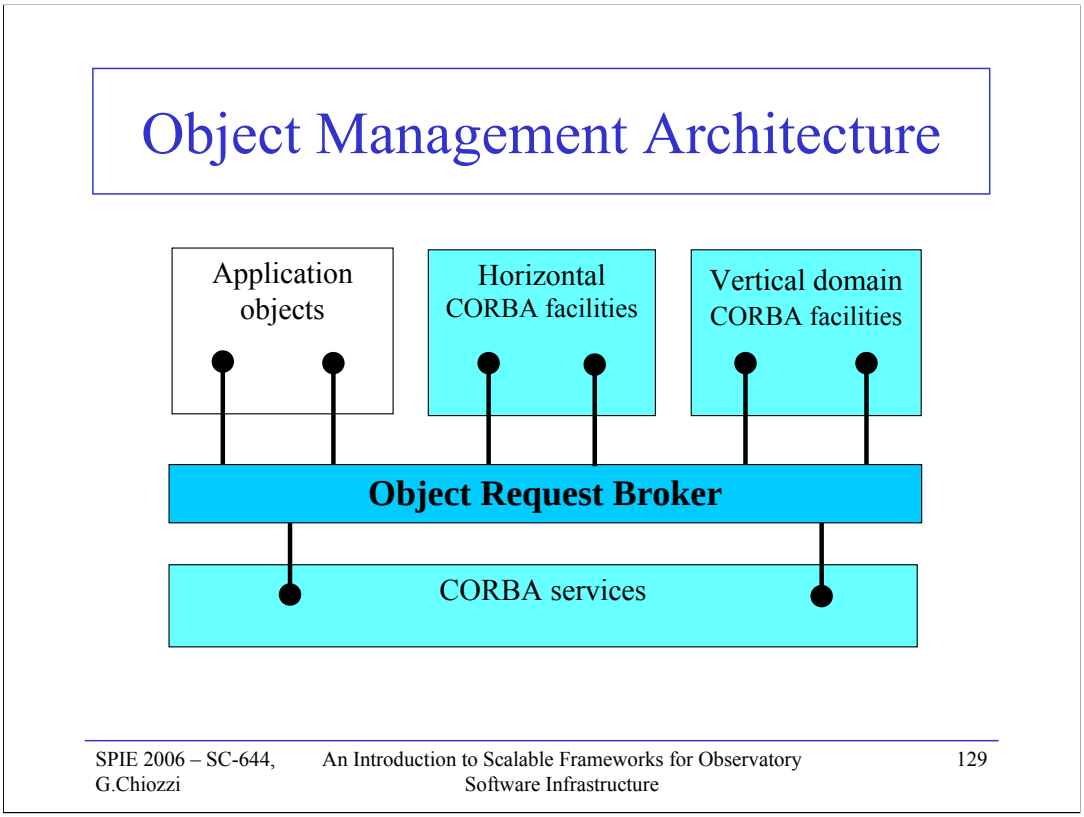
- Facilities for creating your own types:
  - typedef
  - enum
  - const
  - struct
  - union
  - arrays
  - exception
- preprocessor directives:
  #include, #define, macros

Operations and parameters

- Return type of operations can be any IDL type (but **in ACS we do not** allow references to other components)
- each parameter has a direction (in, out, inout) and a name
- similar to C/C++ function declarations

CORBA Services





The OMG has defined on top of the core CORBA architecture an Object Management Architecture (OMA) with the purpose of providing an architecture and interoperability foundation to allow the development of plug-and-play software.

The basic idea is that when applications provide basic functionality, they shall provide it via standard interfaces.

In this way:

•Multiple, interchangeable implementations of the same functionality can be interoperable but still be characterized by differences in performance, price or adaptation to run on specific platforms.

•Specialized high level components, developed independently and for different purposes, can still be made interoperable because they use the same palette of basic building blocks (interfaces).

Applications - even if they perform totally different business tasks - share a lot of common functionality: objects notify other objects when something happens; object instances are created and destroyed and new objects' references are passed around; operation must be made secure and transactional. Beyond this, applications within a business domain (telecommunication, transportation,…) share even more functionality. The OMA abstracts out this common functionality from CORBA applications into a set of standard objects that perform standard, clearly-defined functions.

As it has been discovered at a high price in the past years, it is not sufficient to write software using object oriented techniques or in any case specific languages to make it reusable and interoperable with other software. Two pieces of software can work together only if they expectations on the environment they want to live in are compatible. Just like two IC chips can live on the same motherboard only if they expect the same kind of power supply.

The OMA defines:

•CORBA Services (COS)
Specify basic services that almost every object needs. This part of the OMA started first and is quite well developed and supported

•Horizontal facilities
Provide intermediate level services common to all applications. They can substantially help to develop applications in any domain but are not strictly necessary.

•Vertical domain facilities
Are specifications for services useful in specific application domains and are defined by Domain Task Forces inside the OMG with focus on a particular application domain, such as telecommunication, Internet, manufacturing and so on. There is an OMG interest group on real time control and there could be one on Astronomy or, more in general, experimental facilities. Some of the facilities developed here have found a widespread usage very well outside the original application domain.

This distinction is useful to clarify who inside the OMG is responsible for the specification of a service or of a facility.

But from the point of view of users of services and facilities it is not really important and there are now vertical domain facilities (like the Telecom Notification Service) that can be actually considered for any purpose of usage plain CORBA services.

Therefore in the coming pages I will not distinguish and only talk in general terms of CORBA services.

CORBA services

- Defined on top of the core CORBA
- OMG defines IDL interfaces and semantic to services in order to ensure interoperability
- To be implemented as standard CORBA objects by vendors
- Vendors choose what services to implement

The services are defined on top of the ORB.

They are defined by means of formal specification documents that include IDL interfaces and semantic description in English text. They shall be implemented as CORBA Objects (or appear as internal CORBA Objects, I.e. CORBA objects that are not accessible from outside the process but only to the local objects).

The vendors or CORBA implementations are free to choose what services they want to implement. But if they implement a service, they are bound to implement it according to the specifications. Some widespread services are implemented by every vendor, but some other are extremely specific and seldom implemented.

But it is important to notice that it is in many cases possible to select any implementation of a service and use it with another ORB, thanks to the fact that interfaces are through IDL and the interoperable CORBA communication bus.

CORBA services

- Naming Service
- Event Service
- Notification Service
- Logging Service
- Audio/Video Streaming Service
- Life Cycle Service
- Concurrency Control Service
- Time Service
- Property Service
- Persistent State Service
- Security Service
- Trading Service
- Transaction Service
- Query Service
- Relationship Service
- Externalization Service

What follows is a list of CORBA services with a brief description:

Naming Service -- Supports both persistent and non-persistent hierarchical mappings between sequences of strings and object references. In addition, the Interoperable Naming Service defines a standard way for clients and servers to locate the Naming Service, as well as any other CORBA service.

Event Service -- Supports decoupled communication among multiple suppliers and consumers using the standard GIOP/IIOP protocol.

Notification Service -- Is a more powerful form of the Event Service that supports filtering and correlation.

Logging Service -- Allows applications to send logging records to a centralized logging server.

Audio/Video Streaming Service – Defines a model for implementing an open distributed multimedia streaming framework.

Lifecycle Service -- Provides a standard means to locate, move, copy, and remove objects.

Concurrency Service -- Provides a mechanism that allows clients to acquire and release various types of locks in a distributed system.

Time Service -- Provides globally synchronized time to distributed clients.

Property Service -- Supports the association of name-value pairs with CORBA objects.

Persistent State Service -- Provides a way to make a service persistent. PSS presents persistent information as storage objects that reside in storage homes.

Security Service – Provides identification and authentication of users and objects, authorization and access control, security auditing, security of communication between objects.

Trading Service -- Implements a mapping between attribute constraints and sequences of object references that match those constraints. Therefore it supports the finding of CORBA objects based on properties describing the service offered by the object

Transactions -- Coordinates atomic access to CORBA objects

Query -- Supports queries on objects

Relationships -- Provides arbitrary typed n-ary relationships between CORBA objects

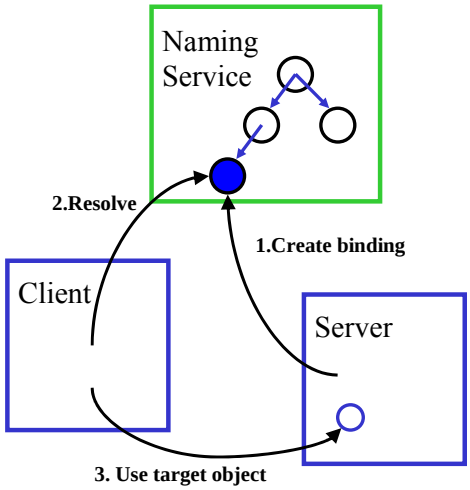
Externalization -- Coordinates the transformation of CORBA objects to and from external media

TAO for example implements most of these services, but other vendors implement only a subset.

In the following pages we will look at some examples with more details, with the purpose of understanding what Services are and what they can bring to the developers.

Naming Service

- maps logical names to server objects
- references may be hierarchical, chained
- returns object reference to requesting client
- allows federation



The naming service is a simple locating service that allows clients to look up an object location using a name as a key. The name can be specified in a human-readable stringified name format or in a raw name format. Typically, a tree-like directory for object references is used, much like a file system provides a directory structure for files.

Before a client can look up an object, the association between the object location and its name must be created. This association is known as an *object binding*, and it is normally made by a CORBA server.

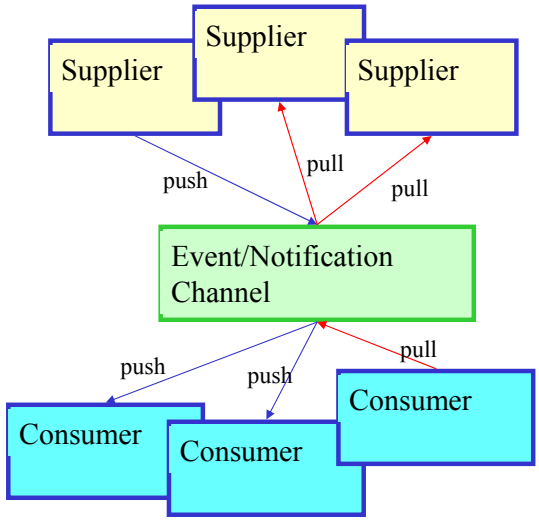
Then a client can *resolve* the name asking the Naming Service by name and receiving back the reference to be used.

The Interoperable Naming Service (INS) is a URL-based naming system on top of the CORBA Naming Service, as well as a common bootstrap mechanism that lets applications share a common initial naming context.

Naming Services can be federated. A federated service provides a single logical service to clients, but consists of a number of physical servers. This allows scalability and redundancy of the system.

Events and Notification services

- Asynchronous messaging
- *Publish/subscribe* paradigm
- Decouple suppliers and consumers of information
- Push and pull models
- Notification adds to Event Service:
  - Filtering
  - Structured Events
  - Sharing subscription information
  - QoS properties



The standard CORBA operation invocation results in synchronous execution:

•Both client and servant must be active

•The client blocks until the operation returns

•Communication is point-to-point

For many application it is required to have asynchronous communication, eventually with multiple suppliers and consumers.

The Event Service provides a model for asynchronous communication based on the “publish/subscribe” paradigm with an *Event Channel* that plays the role of a mediator between suppliers and consumers of events and encapsulates the queuing and propagation semantics.

Some examples are:

•A telemetry system where telemetry data is published and displayed on many consoles, on top of being archived in a central database.

•An alarm system, where alarm conditions can be published by many objects and need to be collected in a central service and dispatched again to many clients.

•Synchronisation events emitted by one object and used to synchronise the action of many other objects. For example a “target reached” event used to start exposure and data collection.

The Notification Service is mostly an extension of the Event Service, but provides very important features.

Filtering is extremely important, because without that the Event Service is actually a broadcast mechanism: all subscribers receive all events published on the channel and have to select themselves the ones they are really interested in.

Logging service

- Logging is fundamental for distributed systems, but complex:
  - Persistent log records
  - Log record filtering
  - Log forwarding for scalability and federation
  - Support for QoS associated with the Notification Service.
  - Administration interfaces
- Based on Notification Service
- Implements CCITT X.735 recommendation

A centralized logging system is essential for the development, monitoring and administration of a distributed system.

Events happen in many different hosts and processes and need to be correlated to be able to understand the inter-relationships between things occurred in different places.

Therefore developers want to be able to log actions and events and collect them in a central place.

It must also be possible to store this information persistently for later analysis.

Also “telemetry” information about the behaviour of the system has the same typical life cycle.

A logging system is really a common service needed by any application and is also very complex if we take into account the requirements for scalability and reliability.

The OMG Telecom working group has defined a logging service supporting the CCITT X.735 recommendation and base on the CORBA Notification Services that is now widely used also outside the Telecom vertical domain and has been implemented by various vendors.

Scalability is based on forwarding specifications that allow log objects to forward messages one to the other building hierarchies and redundant nets.

Quality of Service specifications and a thoroughly defined Administration Interface take care of the reliability requirements.

Summary: CORBA Services

- OMG defines standard specification for common services that go across domain and application borders
- Vendors provide implementation of the service
- These two steps guarantee to the user:
  - Scalability
  - Reliability
  - Interoperability
- Pick the right service for your requirements:
  - Asynchronous communication
  - Security
  - Persistency
  - Load balancing
  - ……

Here I summarize once more the main design goals of CORBA Services.

These goals map well with the requirements that have led us to identify the need for adopting a Middleware.

By picking the right services from the palette of available specifications we can find a ready made solution for wide sets of application requirements.

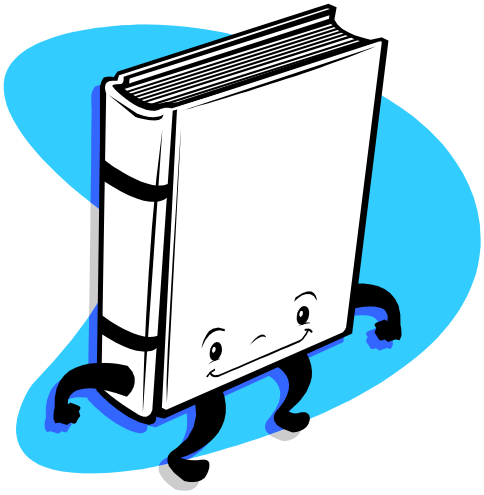
The fact that the services have been specified at OMG consortium level guarantee that they have been thoroughly thought and are coherent and consistent.

Although often the specification appears complicated and over killing, implemented in house what provided by a Service results very often in over simplification of the problem and under estimation of the requirements with the result that it is often necessary to radically extend and change the architecture of the “home brewed” service during development with inconsistent and often not scalable and unreliable results.

The general (but not absolute) interoperability allows to select the implementation that better satisfies the requirements and there are often available “light implementations” that are simpler and thinner than the full blown implementation at the expense of features (sometimes unneeded in the application domain).

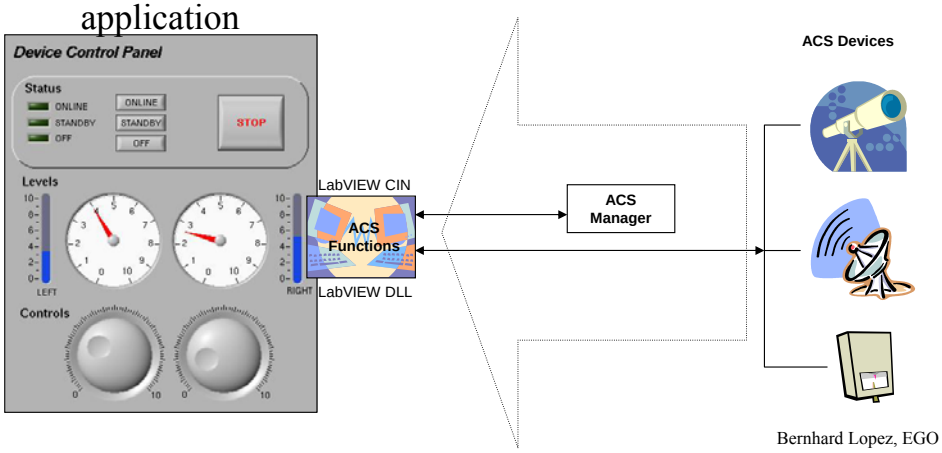
Even in the case where it is not possible to use an existing implementation, it is often very productive to start from the OMG specification and take it at the basis for a home made partial implementation.

Interfacing ACS and LabVIEW



ACS to LabVIEW Interface

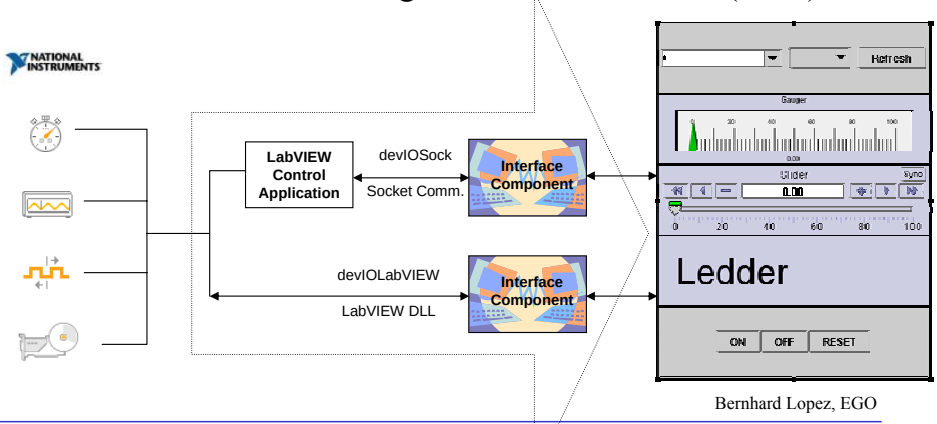
- Invoking ACS functions directly from the LabVIEW application



Bernhard Lopez, EGO

LabVIEW to ACS Interface

- Using an Interface Component based on socket communication or invoking LabVIEW functions (DLL)



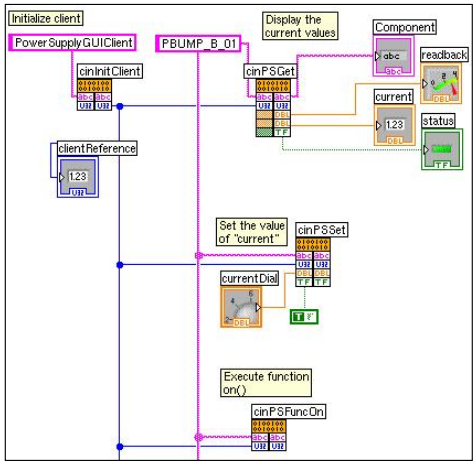
Bernhard Lopez, EGO

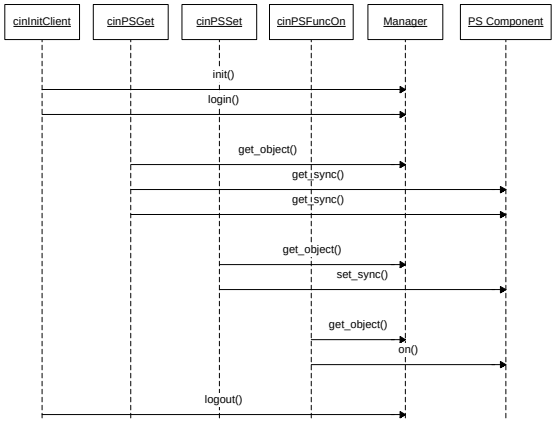
ACS to LabVIEW Implementation

- ACS functions are incorporated at LabVIEW applications via Code Interface Nodes (CIN)
- Why CIN and not DLL?
  - Global variables
  - Standard hook-functions
- Using several CINs makes the development straight-forward:
  - cinInitClient
  - cin<*device*>Get
  - cin<*device*>Set
  - cin<*device*>Func<*function*>

Bernhard Lopez, EGO

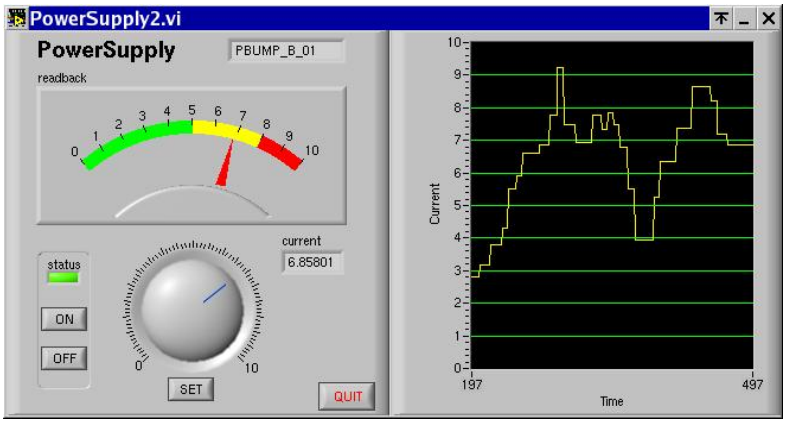
ACS to LabVIEW Implementation (cont.)





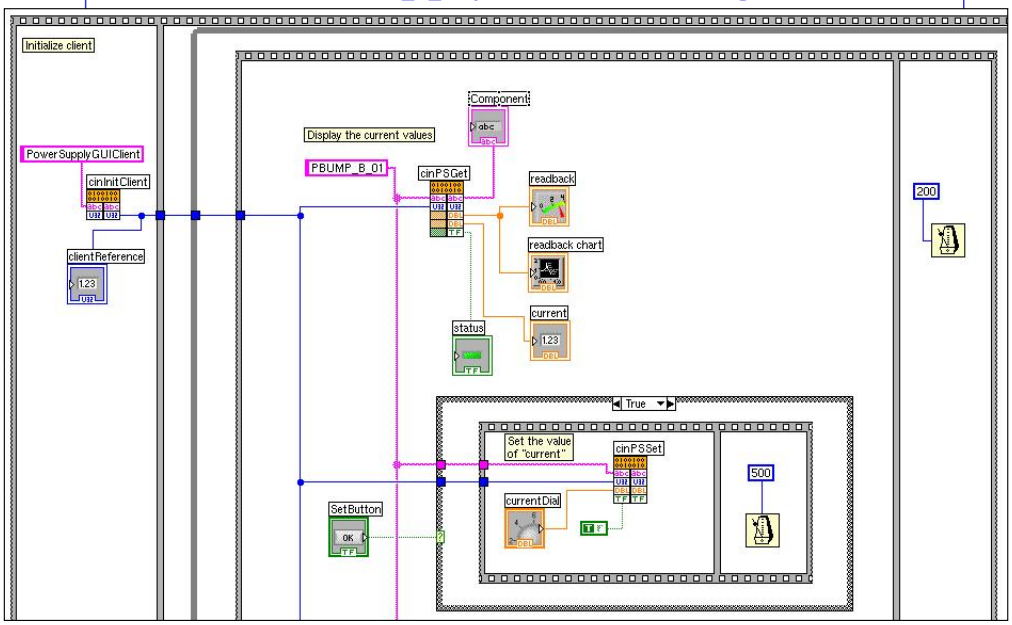
Bernhard Lopez, EGO

PowerSupply GUI



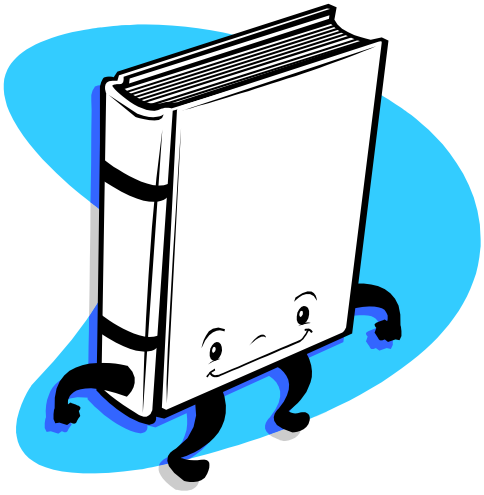
Bernhard Lopez, EGO

PowerSupply Block Diagram



Bernhard Lopez, EGO

Alarm system details

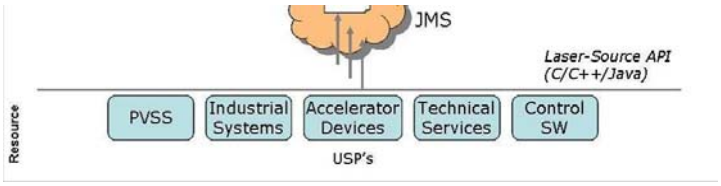


ACS Alarm system technology

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

Resource tier

- Consists of a dispersed set of sources
- Communicates with business tier via the Source API
  - Triggers FS changes
  - Sends 'Keep-alive'/Synch message
- Implemented on a variety of platforms and OS



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.

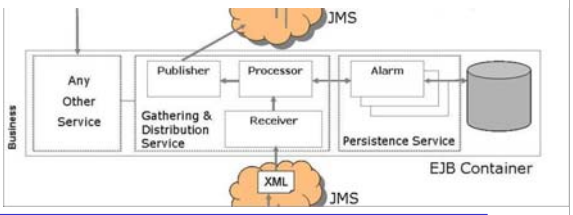
The *Laser-source API* has been written to connect the sources to the business tier and is very small in order to be as simple as possible for the user. The API is written in java and in C++ and runs in all software environments used at CERN like embedded and real time system, different operating systems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

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.

Business tier services

- FS collection, analysis and distribution
  - FS changes are asynchronously and sequentially collected from sources
  - Different techniques are used to reduce the number of alarms distributed
  - FS's are distributed into a hierarchy of domains of interest
- Source monitoring
  - 'Watch-dog' mechanism based on source's 'keep-alive' message
- Alarm console client configuration
- FS definition
  - FS definition inserts, deletes, updates
  - FS relationships, used for reduction
- FS archiving
  - FS and FS definition changes



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.

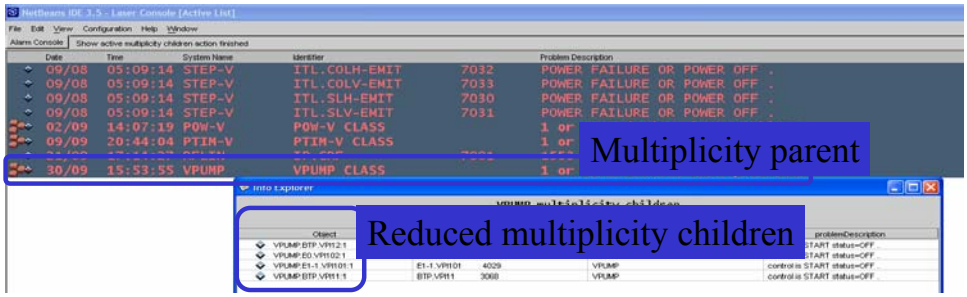
In order to maintain easy and short the *Laser-source API*, the sources send to the business layer only the triplet describing an alarm with the time of its creation. For each alarm received, the business tier reads its complete definition from the database in order to present to the operators a complete snapshot of the situation, its possible solution and consequences. Table 21 shows some of the information stored in the database for each alarm.

Close up on FS definition

- FS static information
  - Id
    - Fault family (System name)
    - Fault member (Identifier)
    - Fault code (Problem description)
  - Priority
  - Information
    - Cause
    - Action
    - Consequence
  - Piquet information
    - Help URL
    - Piquet GSM
    - Piquet email
  - Definition responsible
  - Location
- FS relationships
  - Source
    - Unique name
    - Brief description
    - Connection timeout
    - Definition responsible
  - Categories
    - Connect alarms to nodes and/or leaves in the category tree
  - Multiplicity reduction
    - Create the multiplicity parent and set the threshold value
    - Connect children alarms to the parent
  - Node reduction
    - Select the node parent
    - Connect children alarms to the parent

Close up on FS reduction

- Multiplicity reduction
  - A <*threshold*> number of multiplicity children FS are activated ➔ the multiplicity parent FS is activated ➔ The active multiplicity children are reduced
  - The multiplicity parent FS is a 'dummy' FS
- Node reduction
  - The node parent FS is activated ➔ the active node children FS are reduced
  - The node parent FS is a real FS



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.

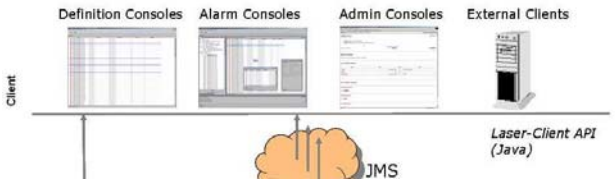
All the alarms, both reduced and not reduced, will be sent to the client because some clients can be interested in receiving all the alarms regardless their reduction status: it is the GUI that hides the reduced alarms to the operators depending on the specific configuration.There are two types of reduction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

Client tier

- Dedicated alarm consoles and software clients
- Communicates with the business tier via
  - The LASER Client API
    - FS changes are sent asynchronously, based on the set of categories and filters passed to business tier
  - The LASER Console API
    - Login and configuration facilities for the dedicated alarm consoles



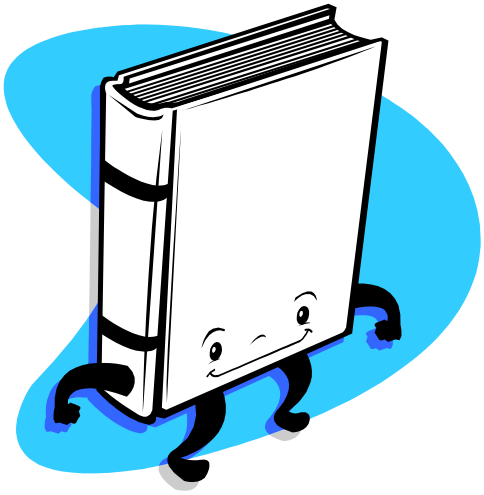
The client tier is composed of java applications that consume the data published by the business tier. The client connects to the business tier by means of the *Laser-console API*. The business tier supports both login and configuration facilities.

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.

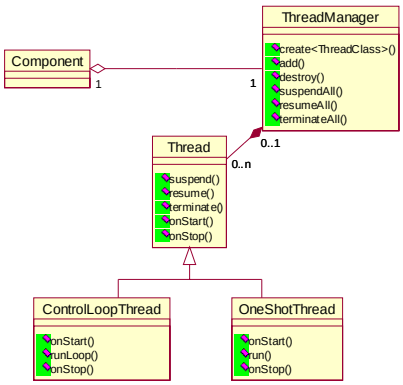
Whenever the alarm consol receives an alarm, it shows a line in the table with the label N that means “new”. When the operator presses the mouse button over the alarm, the N changes to the date when the alarm was issued by the source. If an active alarm becomes terminate, its entry remains in the main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

Other services and packages



Threading Support

- Many Components have a multi-threaded structure
- Management of threads was a source of problems
- Developed easy-to-use threading classes:
  - Override a *run()* method
  - Use the thread manager
- Based on *ACE Threads* in C++, *concurrent* library in Java



Many Components, in particular in the area of the Control Software, have a multithreading structure. This means that there are threads of execution, like control or monitoring loops, that are intrinsically associated with the Component, i.e. are started when the Component is initialised and stopped when the Component is taken down.

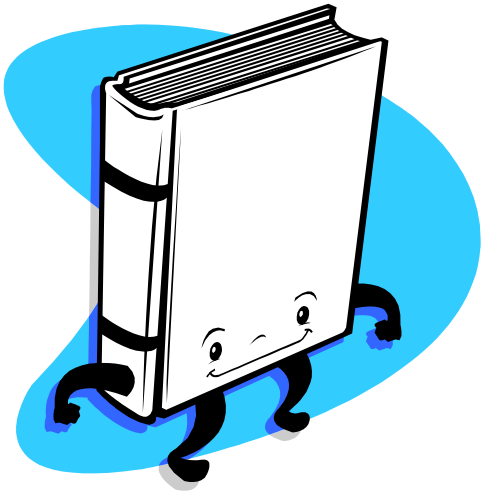
We have seen that the management of such threading Components was a source of problems in the application code, with threads left hanging after Component destructions and other misbehaviour.

We have therefore decided to provide support for well behaving thread design patterns, in particular for C++ and Java.

Each Component now has an associated pool of threads. The ContainerServices provides Components with a Thread Manager object that can be used to get hold of Component-specific threads. This makes it possible to tie the lifecycle of the threads to the lifetime of the Component. It is actually very important to make sure that when a Component is de-activated all related threads are cleanly terminated. Failure to handle this situation might introduce large instabilities in the system, often difficult to diagnose. Problems in this respect can come from the integration in Components of functionality coming from 3rd party packages: in this case we cannot rely on the Thread Manager to handle threads spawned by the external libraries.

In C++ we have built threading classes based on top of the very good APIs provided by the ACE framework. Sub-classing and overriding one method is sufficient to have a thread function executed once (in order to have one-shot asynchronous action) or in a repeated loop (as in the implementation of a control loop). Complete management of the thread (start, stop, resume, etc) is possible.

ALMA Software Engineering details



Integration Layers :

I1	Compiles and links successfully.
I2	Adoption of approved Coding Standards.
I3	Unit or Integration test passed.
I4	Test Coverage is sufficient.
I5	Run-time memory checks ok.
I6	Computation of Complexity and other metrics.

This nomenclature is only introduced to explain the next slide

Quality Assurance Tools

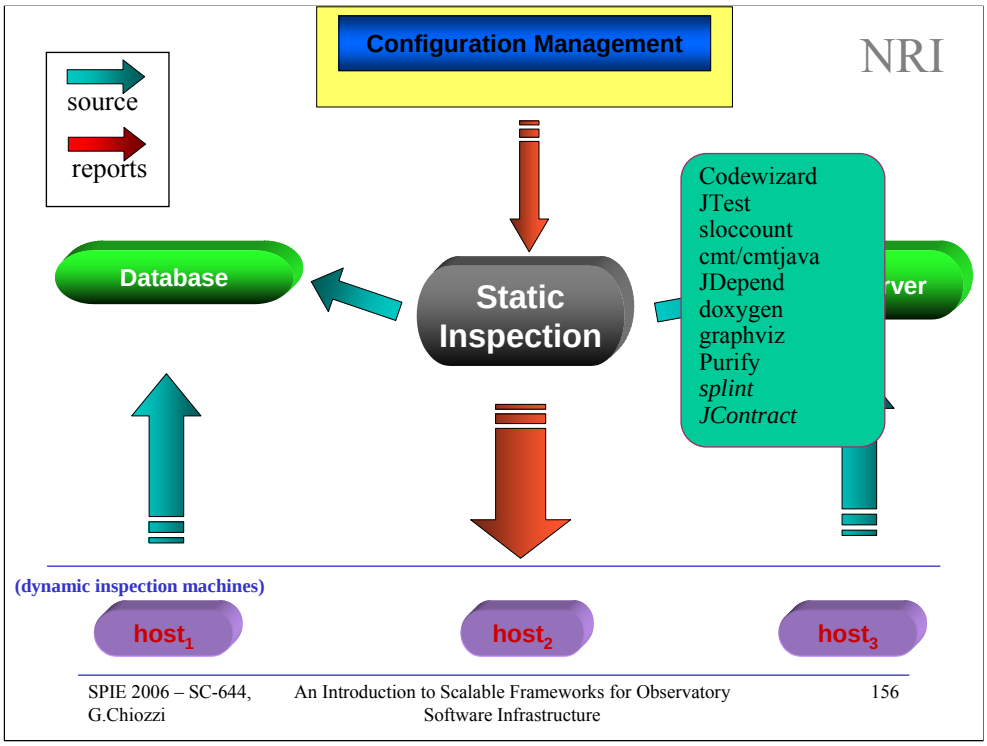
	Linux		
	C/C++	Java	Python
I1	acsMakefile	acsMakefile	acsMakefile
I2	Codewizard / Splint	JTest	PyLint
I3	TAT, CppUnit	TAT, JUnit	PyUnit
I4	Purify	JProbe	NA
I5	Purify/Valgrind	JProbe	NA
I6	CMT++	CMTJava	NA

For each category and programming language the corresponding tool is shown

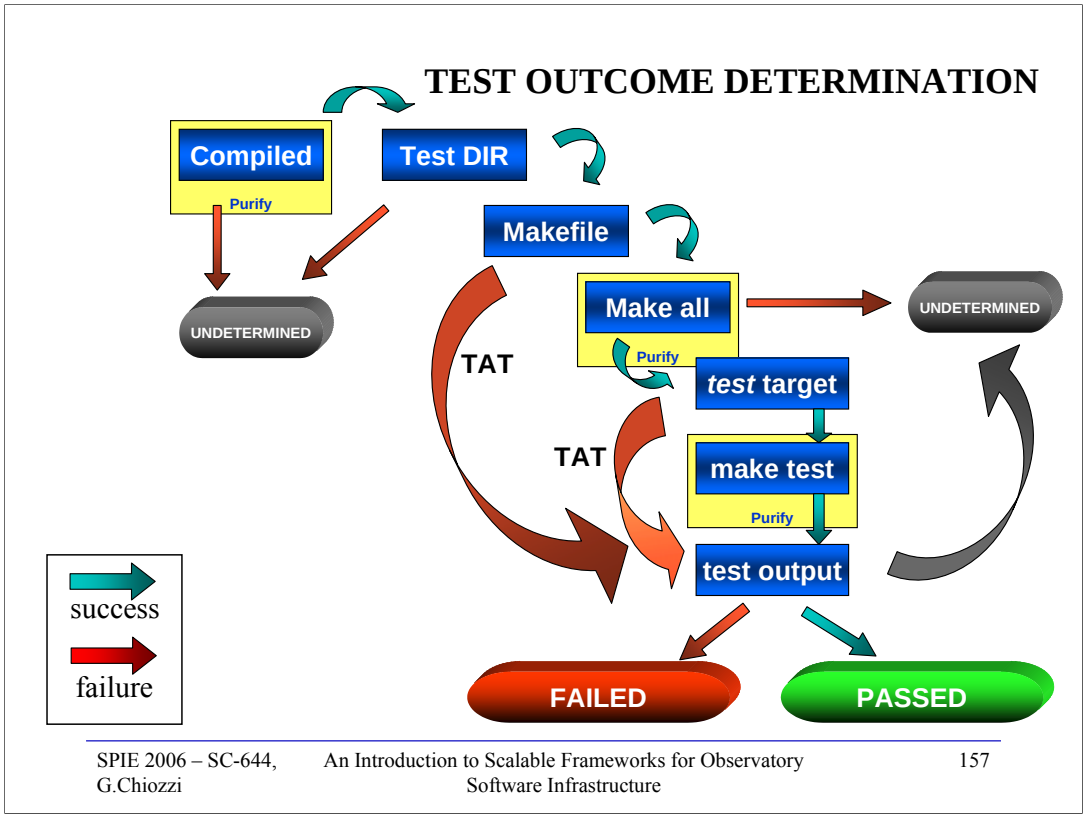
Automated Inspections (NRI)

- Build reports (GO/NO-GO)
- Tests (availability, execution results GO/NO-GO, coverage, memory behaviour)
- Coding Standards (ALMA, MISRA, Motorola, Scott Meyers, Sun)
- Inline documentation sufficiency
- lines of code (total, per language, per module, production vs test code)
- Algorithmic Complexity (McCabe)
- module dependency diagrams
- SPR statistics, number of commits, number of unused files
- Java duplicate classes verification
- Events and Channels in use
- metrics on design quality (Robert C. Martin, for Java)

Introducing NRI: what does it bring



Generic layout of NRI. Dynamic NRI in particular is to be considered as an early warning system. NRI is also the *glue* which holds together all the tools mentioned above (and probably more)



For each and every blue box there may be an outcome of success or failure. The specific path followed determines whether the TEST will be considered FAILED, PASSED or UNDETERMINED.

Data archival for trend analysis

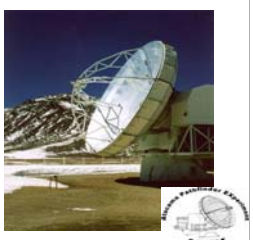
http://websqa.hq.eso.org/alma/snapshot/

	ACS	ARCHIVE	CONTROL	CORR	EXEC	ICD	OBSPREP	PIPELINE	SCHEDULING	TELCAL
Total Modules	**56**	**6**	**34**	**49**	**3**	**8**	**3**	**2**	**2**	**9**
Build FAILED	**1**	0	**13**	**8**	0	0	0	0	0	0
Test FAILED	**2**	**3**	0	**3**	0	0	**2**	**2**	**1**	0
Instrumentation Failed	**11**	**1**	**2**	**1**	0	0	**1**	0	**1**	0
Test UNDETERMINED	**7**	**1**	**16**	**12**	0	0	**1**	0	0	0
No Makefile	0	0	0	0	0	0	0	0	0	0
Missing Test Directory	**6**	**1**	**14**	**30**	0	**8**	0	0	0	**3**
Test TIMED OUT	0	0	0	0	0	0	**2**	0	0	0
Test CORE DUMPED	0	0	0	0	0	0	0	0	0	0
Test PASSED	**41**	**1**	0	**3**	**3**	0	0	0	**1**	**6**

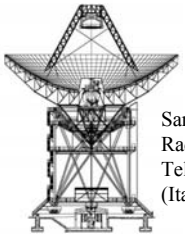
To make the point clearer, this is a snapshot from a moment in time last year.

The ACS community

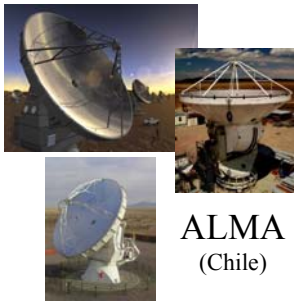
OAN 30m (Spain)



APEX (Chile)



Sardinian Radio Telescope (Italy)



ALMA
(Chile)

HPT
Hexapod
Telescope
(Germany → Chile)

ANKA (Germany)