

Container-component model and XML in ALMA ACS

Heiko Sommer^a, Gianluca Chiozzi^a, Klemen Zagar^b and Markus Völter^c

^a European Southern Observatory, Karl-Schwarzschild-Straße 2,
D-85748 Garching b. München, Germany;

^b Cosylab, Teslova 30, SI-1000 Ljubljana, Slovenia

^c voelter – ingenieurbüro für softwaretechnologie, Ziegeläcker 11,
D-89520 Heidenheim, Germany

ABSTRACT

ALMA software, from high-level data flow applications down to instrument control, is built using the ACS framework. To meet the challenges of developing distributed software in distributed teams, ACS offers a container/component model that integrates the use of XML transfer objects. ACS containers are built on top of CORBA and are available for C++, Java, and Python, so that ALMA software can be written as components in any of these languages. The containers perform technical aspects of the software system, while components can focus on the implementation of functional requirements. Like Web services, components can use XML to exchange structured data by value. For Java components, the container seamlessly integrates the use of XML binding classes, which are Java classes that encapsulate access to XML data through type-safe methods. Binding classes are generated from XML schemas, allowing the Java compiler to enforce compliance of application code with the XML schemas. This presentation will explain the capabilities of the ACS container/component model, and how it relates to other middleware technologies that are popular in industry.

Keywords: Container, Component, CORBA, XML binding, Java

1. ACS, ALMA'S "ARCHITECTURAL CRUCIBLE SYSTEM"

Software multiculturalism, with geographically scattered developers at diverse levels of experience or interest in software design and technologies, is a major challenge for the ALMA software project.

The ALMA Common Software (ACS) was conceived early on as a framework not only to provide a software infrastructure suitable for the domain, but also to funnel into a uniform architecture the disparate styles and approaches naturally thriving within the over twenty teams at twelve participating institutes. To assure its acceptance within the project, the ACS architecture was subject to project-wide reviews, and has been shaped by the various styles of software development in this community. The result is a framework that unifies choices of application developers in a natural fashion, and thus helps join together software workers into a global development team.

To reduce the effort expended for development and maintenance, it is highly desirable to construct the framework as a layer on top of a proven middleware technology instead of building from scratch. Such an approach should also ensure easier integration with non-ALMA software if necessary, as well as facilitate collaboration between the ACS team and the application developers it is meant to serve.

The middleware selection process was performed in 1999 and 2000, at the very inception of the ACS project. The choice of middleware had to be made carefully, as it was to form the cornerstone of a software system that would have to stand the test of time – the several decades the ALMA facility is expected to be in operation. The following factors were considered:

- Amount of added value compared to direct socket programming.

Further author information:

H.S.: E-mail: hsommer@eso.org, phone: +49 89 3200 6341

- Complexity of the middleware and the steepness of the associated learning curve
- Additional services, such as naming, persistence, streaming, asynchronous communication, ...
- Availability for all target platforms of interest to ALMA.
- Seamless support of the full application gamut, from quasi real-time control system software, up to high-level data flow applications.
- Acceptable performance and no inherent limitations on scalability.
- Level of standardization.
- The amount, quality and longevity of support from vendors, academy and community.

These selection criteria left but one remaining middleware candidate¹: CORBA.

Since 2000, many things have happened in the middleware technology landscape. Microsoft launched the .NET initiative, Sun's Java 2 Enterprise Edition gained wide acceptance, and new promising middleware solutions have appeared, such as the Internet Communications Engine (ICE).² Also, there has been much ado about Web Services and their promise of integrating heterogeneous IT environments, especially in business applications.

Interestingly, if the middleware choice for ALMA were re-evaluated, it is very likely that CORBA would be chosen again:

- CORBA is still the only middleware that can be deployed on all target platforms currently in use by the ALMA project (Windows, Linux, VxWorks, Sun, ...).
- Only CORBA allows interoperability of all languages used by the ALMA development teams (Java, C++, Python).

Unfortunately, out-of-the-box CORBA does not satisfy all the requirements that ALMA Common Software is expected to meet, and ACS has had to provide additional features. ACS became a melting pot of selectively adopted architectural elements of CORBA, the CORBA Component Model³ (CCM), and Web Services.⁴ Let us look at some examples of how ACS enhances CORBA, borrowing ideas from mainstream solutions, sometimes even exceeding those:

Container-component model

ALMA software is structured as components which run inside containers.^{5,6} This enforces a uniform structure in all subsystems and enables developers to focus more on domain problems than on software engineering issues. The developed code is cleaner and smaller.

The component idea goes beyond the philosophy of CORBA, where applications must be started independently of the framework (and thus typically have a `main()` method), and can then *use* the CORBA software bus for remote object communication. Although more powerful, the CORBA approach is in this respect conceptually similar to using a socket library, and brings about code duplication as well as a lack of uniform structure in the system.

Confronted by successful component frameworks such as Sun's EJB or Microsoft's COM+, the Object Management Group (OMG, the custodian of CORBA) has recognized the advantages of a standardized container model, and produced the CCM specification. At the moment, not many free CCM implementations are available or are mature enough to be adopted by ACS, even though products like MICO or TAO's CMM addition, CIAO, look tempting.

The ACS component model tries to resemble that of CCM, e.g. by requiring functional component interfaces to be defined in (standard) CORBA IDL, which is a subset of CCM's IDL. Currently, we are investigating whether providing binary-level compatibility with CCM and J2EE is worthwhile.

Service-oriented architecture (SOA)

Many people find that CORBA's support for loosely coupled services still leaves a lot to be desired, and will frequently bring up requests for elements of service-oriented architectures, such as web services.

This naturally happens at the data-flow end of the application spectrum covered by ACS, where one of the requirements is to have ALMA subsystems coupled as loosely as possible. Yet web services go too far in this direction. They mandate exchange of data by value in textual format, which is not always desirable for performance reasons. Also, they make some communication mechanisms (e.g., callbacks) hard to employ. More importantly, unlike globally communicating services from different providers, ALMA software can always be centrally rebuilt at once, which eliminates the need for generic interfaces; by being generic, such interfaces can achieve better interoperability with previous software versions, but only at the expense of losing compile time checking. Another shortcoming of web services is the difficulties with stateful resources, which are currently tackled by related SOAs such as the Open Grid Services Infrastructure (OGSI) or the WS-Resource Framework.

In ALMA, almost all software will run behind the firewall; there will only be isolated, dedicated interactions with the outside world, e.g. project submissions, status reports, or data delivery. We felt that there was no need to integrate into ACS any specialized middleware for grid-like distributed computing, since the limited functionality required could be built separately, possibly using one of the solutions CORBA provides. Therefore ACS does not use any SOA framework directly, but instead adopts some of its concepts and implements them on top of CORBA in a way more suitable for ALMA, using "static glue" wherever possible. The most important such feature, type-safe exchange of XML data using binding classes, is the main topic of this paper and is described in detail in section 3.

For passing complex data objects by value, ACS uses XML as the on-the-wire representation. The concept is the same as found in the Simple Object Access Protocol (SOAP) specification for Web Services. The difference is that CORBA's IIOP is used as the underlying transport, which capitalizes on the features of the CORBA platform that are not available using web servers, for example the ability to efficiently push a complex data structure to several subscribers simultaneously via a publish-subscribe asynchronous communication infrastructure (the CORBA Notification Service).

CORBA Encapsulation

CORBA is complex, not just because it is powerful, but also because of its "Design by Committee"*. Very often, the large variety of choices offered by CORBA becomes overwhelming. ACS strives to encapsulate all interactions with CORBA in order to flatten the learning curve for application developers. Examples of such encapsulation are the container, which greatly simplifies remote communication, and the ACS notification channel⁷ wrapping the underlying CORBA Notification Service. ACS also transparently assembles together the ORBs and services from different ORB providers in an effort to get the best implementations; when left to the applications, this task is typically considered a formidable practical problem in a CORBA based system.

Centralized Administration

ACS uses a central Configuration Database (CDB), which allows for the central provision of configuration information for all components only once, independently of which container in the distributed system they will eventually be hosted in. Similarly, the configuration for all containers is also stored at a single location – containers only need to reach the central Configuration Database via the Manager; the Manager's network location is either specified when starting the container, or discovered dynamically afterward.

In EJB and .NET Enterprise Services (COM+), central administration is generally not supported in the sense of the previous paragraph (with a few exceptions such as the WebSphere application server). In these frameworks, every node of the distributed system must be configured separately. This can lead to tedious work if the same

*The competition from the proprietary non-CORBA ORB ICE, which the well-known CORBA author Michi Henning has defected to, explains at <http://www.zeroc.com/iceVsCorba.html>: "Often, the only way to reach agreement during the submission process for a CORBA specification is to take the grand union of the feature sets of all the pre-existing proprietary implementations and to somehow shoe-horn them into a standard. This results in specifications that are far larger and far more complex than necessary. This means that [it is] much harder to use the resulting complex APIs."

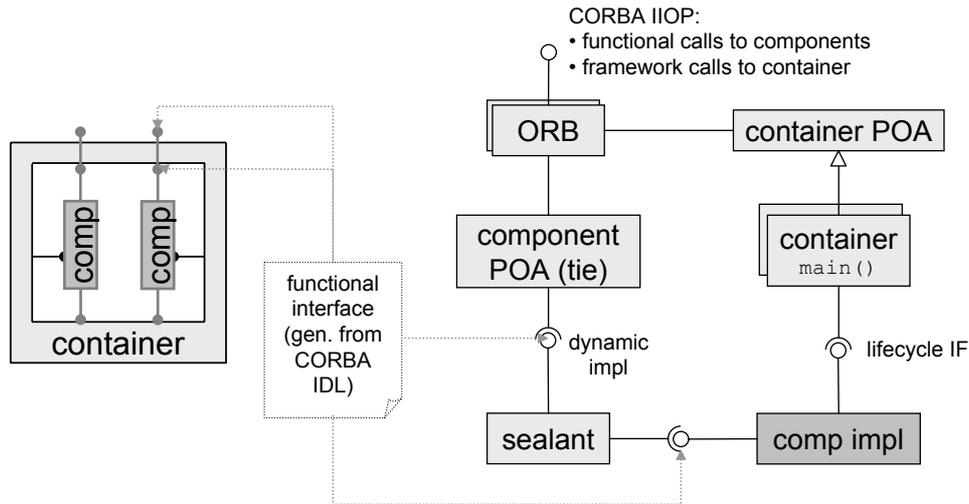


Figure 1. ACS Java container. Left: conceptual view with two components inside a container; right: CORBA view, with the container and a component both registered with the ORB as CORBA services.

property has to be set for all of the containers. Also common operations, such as moving a component from one container to another, require both containers to be reconfigured. Other distributed systems, e.g., EPICS, do not provide a solution to this problem at the architectural level either; instead, they share configuration data using underlying IT infrastructure, such as the Network File System.

2. ACS CONTAINER/COMPONENT MODEL OVER CORBA

There are feature variations among the ACS containers for Java, C++, and Python which are motivated by different requirements in the typical usage areas, as well as limited availability of third party software and ACS development resources. Currently ACS Java containers (which can host components implemented in Java) offer the broadest range of features. The most obvious advantages are the transparent integration of XML binding classes, and the interception of functional calls which constitutes a “tight container”.⁵ Java containers also minimize the infiltration of CORBA into the component code. Future requirements may lead ACS to provide these features also in C++ and Python containers. For the following discussions, we examine the ACS Java container, without pointing out all variations with respect to the other languages.

To illustrate how ACS containers are implemented on top of the CORBA middleware, Fig. 1 shows the conceptual view of a Java container next to a more detailed class diagram. To write application components, the developer can work with the conceptual view in mind, and remain fairly unaware of CORBA working underneath; one exception is that the functional component interface must be specified in CORBA IDL, so that the corresponding generated Java interface can be implemented by the component.[†]

Under the hood, the container has manifold interactions with CORBA:

- When the container is started, it instantiates the CORBA ORB and registers itself with the ORB’s Portable Object Adapter (POA), making the container accessible as a CORBA service via an IDL-defined container interface. This container interface is common to all ACS containers for all programming languages, and allows the ACS framework to centrally administer containers.

[†]It is reasonable for ACS to not try to encapsulate CORBA at this point, because some neutral mechanism for defining interfaces is required anyway, to ensure interoperability in an environment with multiple programming languages. Exposing CORBA IDL to the developer frees ACS from providing its own solution to the same problem.

- The container then uses CORBA to contact the ACS manager and provides it with the container's object reference for future interactions. Invisible to application components, the manager is also implemented as a CORBA service. It dynamically handles requests for access to components by evaluating the deployment information in the configuration database (CDB), and instructing an available container to instantiate and run the requested component. More information on the ACS manager architecture (MACI) and how it provides for flexible deployment can be found in Ref. 1.

Once the container has set up the ORB and done the handshaking with the manager, it turns over the thread of control to the ORB and waits for incoming calls to the container interface. As yet it is an empty container, without components.

- When somewhere else in the system a component or other client needs access to a component that should run in our container, the manager will forward this request to the container. This call contains meta information such as type, unique component name, and implementation class. The container then instantiates the component. To connect the component with CORBA, the container also instantiates the associated POA skeleton class. We use the POA tie approach, which means that the component implementation is independent of CORBA related classes. It is particularly important in Java to not use up the single inheritance option just to fit the component into the framework. Instead, the component only implements the Java interface with the functional methods, which the IDL compiler has generated out of the component specification as part of the build process.

The standard approach using CORBA would be to connect directly the component implementation with the POA object, so that all incoming calls to the component are delegated by the ORB straight to the component. However, the container must be able to intercept functional calls, e.g. to enforce security rules, or simply to log all calls in a standardized fashion. Rather than using CORBA's interceptor architecture, our container inserts a framework object ("sealant") into the calling chain between the ORB and the component. This object uses Java's dynamic proxy mechanism to spontaneously mimic the component's functional interface toward the ORB. It will receive all calls and after having carried out its interception tasks, will usually delegate the call to the real component implementation object. The sealant class is also a convenient place to intercept exceptions thrown inside the component, and deal with them more appropriately than the CORBA ORB would do by default.

Before the container returns the component's CORBA object reference to the manager, thus making it available for functional calls, it gives the component the opportunity to initialize itself, possibly setting up connections to other resources. Every component must implement the Lifecycle Interface, which is all that the container sees of a component. This interface contains methods to notify the component about initialization and shutdown events.

- When a component needs a reference to another component, it does not have to deal with CORBA or the ACS manager. As part of the component initialization process, the container has provided the component with a callback object which implements the so-called Container Services Interface. (This interface is shown only in the conceptual view in Fig. 1 as the "stick" in the components' sides.) The component uses the Container Services Interface to search for and retrieve other components. The container forwards these calls over CORBA to the ACS manager.

Even though the access mechanism is independent of CORBA, the component reference returned to the calling component is a generic CORBA object reference and must be cast ("narrowed") to the correct component interface in order to be useful. It seems likely that in the future ACS will generate type-safe access code in the style of the CCM or EJB Home Interfaces, so that components can retrieve other components without performing CORBA casts.

3. XML EXCHANGE

XML is a well-established format for transporting hierarchical data, and has been adopted by the ACS architecture, as mentioned before in section 1. A common problem with applications processing XML data using the standard techniques for parsing and data access is the resulting reliance on runtime string matching, which in turn circumvents compile time type checking. The result is fragile software.

XML binding frameworks can help restore type safety in such a system. They allow the generation of custom Java classes whose instances form a tree that represents an XML document in memory. All element and attribute data is then accessed through type-safe methods. Application code will thus be coerced by the compiler to follow any changes in the data model, which is defined in XML schemas. Even with binding classes being used inside the implementation, an application will still normally encounter serialized XML at the outside interface level; for example, Java web service implementations must receive XML data in generic string format, even if they subsequently turn these strings into a tree of XML binding objects.

As we will see, ACS removes even this top-level type ambiguity and allows Java components to work solely with binding classes, without ever having to parse or serialize XML. The component developer does not even need to know that XML is the underlying data interchange format.

3.1. XML Data Transfer Objects

In any distributed software system, it is vital to reduce the number of fine-grained calls accessing remote data. The usual strategy is to pack together some cohesive set of data items and transmit them by value over the network at once, c.f. Ref. 8, 9. The gained performance has to be traded in for a somewhat compromised OO design though, introducing issues with stale data.

Transporting groups of data by value in one call not only helps avoid network congestion and improve overall system performance; migrating such data from one machine to another also decouples the two computers, which adds to the robustness of the system. If the computer which originally supplied the data fails later, the other machine will not be affected, since it has retrieved all required data by value beforehand.

Since ACS uses CORBA for remote communication, any data transfer object mechanism that ACS offers to application software must be able to go through the CORBA layer. The requirement for supporting multiple programming languages rules out any solutions based on native language serialization formats, which would otherwise be a possible choice for languages like Java. We indeed set an even stricter design goal, that it be possible for a client to work with a CORBA remote object reference directly, without really demanding any ACS layer on the client side to enable reception of data transfer objects.

There are several ways to transport data by value using CORBA. In the past, for the sake of language independence, CORBA architecture focused on remote invocations, reserving by-value transport to *simple* data types, i.e. primitive data or **structs** of primitive data. For hierarchical data, the **struct** mechanism is very awkward to use, with the lack of **null** value support being one of the nuisances. Only recently has CORBA started to offer a more advanced means for transporting complex data structures by-value, known as CORBA value types.¹⁰ Still CORBA primarily focuses on system functionality rather than system data, a shortcoming that invites system architects to blend in other technologies.

Instead of supporting CORBA value types as data transfer objects, ALMA architecture favored XML for the following reasons:

- XML avoids some difficulties with CORBA's built-in types, as described in Ref. 11.
- XML as a serialization format can be used not only to send data by value between applications, but also to store that data persistently in a file or a database, which would not be possible with CORBA value types. A completely different persistence mechanism would be required, with all the known difficulties of mapping object data to storage systems.
- XML schema allows for data declaration (constraints etc.) more powerful than that which is possible using CORBA value types;
- XML data has a natural string representation, which facilitates generating trace messages; as a string, XML can easily be injected manually into the software system, for example for running unit tests, or to mimic an application that has not yet been built.

The decision to use XML schema to define transfer objects that can be sent as CORBA operation parameters does not necessarily determine the representation of an XML document as a CORBA type. The most obvious choice is the `string` type, which is universally available across middleware solutions, and is generally one of the most popular representations of intrinsically textual XML data. Another interesting, though hypothetical option, would be to use the recently specified CORBA DOM/Value mapping.¹² Not fully serialized, XML data would be sent over CORBA as a Document Object Model (DOM) tree composed of CORBA valuetypes generated from the XML schemas. Unfortunately, this DOM mapping is still not available in any ORB we use, and we did not further investigate this option.

ACS thus uses plain strings as the transportation format of XML data. Most of the problems with the XML string transport, which are discussed in Ref. 11, are luckily resolved in ACS through the integration of XML binding classes (for the time being only offered for Java).

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 file:

- 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. The resulting separate pieces of XML data reference each other using their unique IDs. It is of course possible to send lists of independent XML transfer objects in a single method call; for example, a complete ALMA Observing Project definition, with the associated Proposal and all Scheduling Blocks, can be sent around together at once.

3.2. XML-Java Binding Classes

XML binding frameworks are used to generate native language binding classes from XML schemas. This step is typically integrated into the build process, such that XML schemas are treated as source input, and the (Java-) compiled binding classes are the final output.

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 ("unmarshaling", a form of parsing), and methods to serialize binding objects to XML ("marshaling"). They also allow validation against the schema. In Fig. 2 we see some of the classes compiled from the Scheduling Block XML schema. This schema defines an element `SchedBlock`, which has child elements such as `ObsTarget` or `PhaseCalTarget`. The binding classes match these schema types and elements.

Applications are written against the type-safe accessor and manipulator methods of the binding classes. If someone modifies the XML schema that defines the data model for the XML transfer objects, new binding classes will be generated; the compiler will then enforce valid usage of the XML data, e.g. denying access to a data field which was removed as part of the schema modification. Note that with the standard representations of XML (such as DOM), application code would contain generic calls like `addChildNode` instead of `addPhaseCalTarget`, thus defying compile time checking.



Figure 2. Java XML binding classes generated by the Castor framework, as seen in the Eclipse IDE.

ACS harnesses Castor[‡] for XML binding. Another potential candidate, SUN’s JAXB[§], unfortunately lacks the ability to serialize and parse incomplete XML data. (We believe that it is essential to be able to start constructing XML data in one component, and while the data is not yet valid against the schema, transfer it to another component.)

For C++ and Python, we have not yet found satisfactory binding frameworks. Fortunately, the way ACS handles XML transfer objects leaves the usage of binding frameworks optional, with the decision whether to use them made separately for every component and every remote interaction. Missing support for binding classes for certain languages does therefore not compromise system interoperability. Especially in the case of C++ components, there is a much lesser demand for binding classes than in Java, since these components are typically used in software layers close to the hardware, where data is either of fine granularity or bulky size, but rarely of the kind that lends itself to the usage of XML transfer objects.

3.3. Transparent Serialization

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 (see section 3.1). 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, but could not examine the contents of the string on behalf of the application to guarantee that the string contents are valid XML; much less could they guarantee that the XML data is of the type the receiver expects. At both ends of a remote call, the applications would be taking on the burden of performing their own marshaling and unmarshaling above the ORB level.

ACS resolves this problem by integrating the marshaling and unmarshaling of XML binding classes into the container, unnoticed by the components. The idea is illustrated in Fig. 3:

[‡]<http://www.castor.org/xml-framework.html>

[§]<http://java.sun.com/xml/jaxb/>

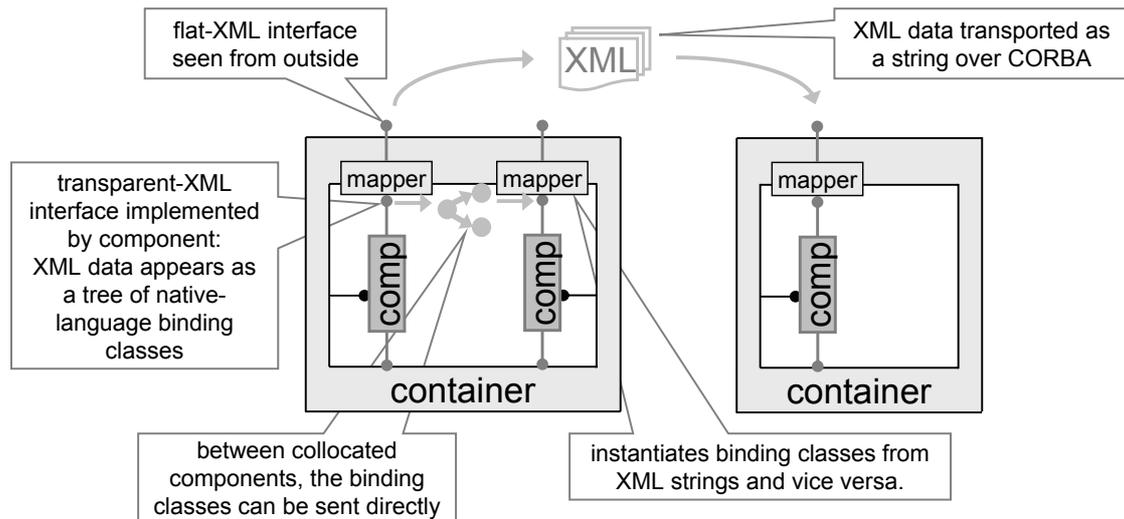


Figure 3. Conceptual view of ACS containers providing XML binding class objects (depicted as gray filled circles) to components, while using a CORBA compliant string serialization format outside the container.

- The “flat-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 (i.e. an implementation of the Java interface generated by the standard IDL compiler), in which XML data comes in Java **Strings**. This implementation is nothing more than what has been described in section 2 as the sealant interceptor class.
- The “transparent-XML” interface is a Java interface which ACS generates using a custom IDL compiler. It resembles the flat-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. It compares the two interfaces and maps the parameters of incoming calls to the corresponding parameters in the transparent-XML interface.
- 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, whereas for calls between components in different containers, the container transparently marshals and unmarshals all XML binding class objects.

As an example, the IDL definition of a component that selects a Scheduling Block could look similar to

```
typedef xmlentity::XmlEntityStruct SchedBlock;
interface MyXmlComponent : ACS::ACSCOMPONENT
{
    SchedBlock getBestSchedBlock();
};
```

where `XmlEntityStruct` is the CORBA struct that contains the serialized XML as a `string`. The standard IDL compiler generates the flat-XML Java interface following the IDL-to-Java mapping rules, unwinding the typedef'd `SchedBlock`:

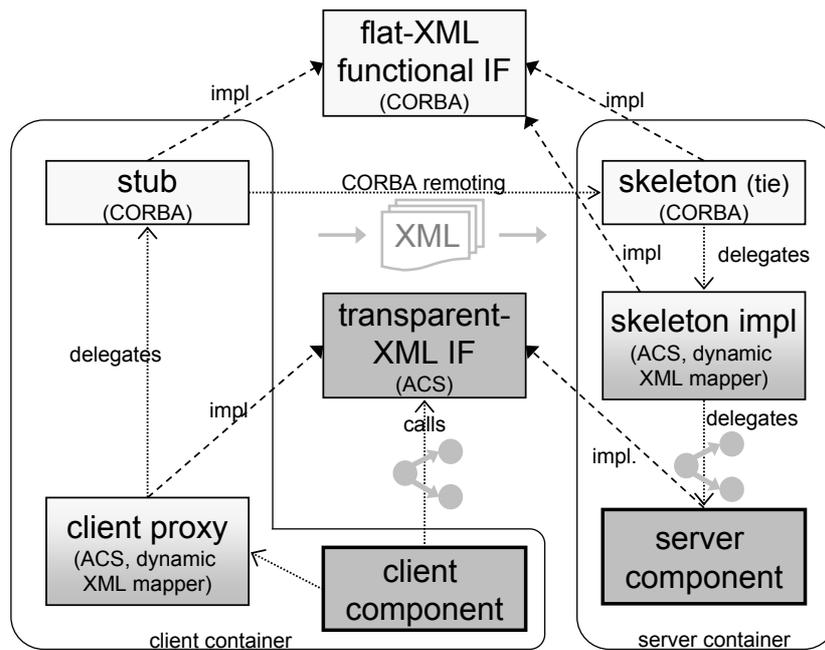


Figure 4. Framework classes and flow of XML data from one component to another. To the darker shaded objects and interfaces, XML data appears as binding classes, to others as serialized **strings**. Note that only the client and server component classes must be written by the developer; the rest is provided by ACS or CORBA.

```
public interface MyXmlComponentOperations extends ACSComponentOperations
{
    alma.xmlentity.XmlEntityStruct getBestSchedBlock();
}
```

The ACS IDL compiler exploits the `typedef` for the `SchedBlock` and generates the transparent-XML Java interface, using the correct binding class:

```
public interface MyXmlComponentJ extends ACSComponentOperations
{
    public xyz.SchedBlock getBestSchedBlock();
}
```

The component implementation can therefore create or retrieve or otherwise manipulate an instance of the Java class `xyz.SchedBlock`, enjoying type-safe methods for navigation and data access, and then simply return this object, oblivious to the subsequent marshaling performed by the container. The tedious task of XML parameter conversion is removed from the application code, and the developers can trust the compiler that at runtime no XML data of unexpected format will be received.

The ACS IDL compiler is built as an extension to the OpenORB IDL compiler, using the convenient parse graph of the IDL file it provides. To produce the transparent-XML Java interface, the ACS IDL compiler does not simply replace the `typedef`'d occurrences of XML data with the corresponding Java binding classes, but instead evaluates the entire IDL parse graph to capture all data types that must be replaced with corresponding binding class aware types. For instance, if a component IDL file declares a data `struct ObsProjectTree` which has an XML `SchedBlock` as one of its fields, the ACS IDL compiler will generate a variation of the Java class that represents the `ObsProjectTree` (i.e. one with a Java `SchedBlock` binding class field) and will substitute all occurrences of `ObsProjectTree` with the new class, possibly leading to further replacements. While the ACS IDL compiler fully propagates type substitutions, it keeps the number of custom generated Java interfaces and

classes at a minimum. That is to say, it does not redundantly produce peer classes for those generated by the standard IDL compiler as long as the transparent-XML mechanism does not require them.

The (un-)marshaling mapper is built similarly to the container sealant, implementing the flat-XML interface dynamically. Fig. 4 shows the interaction of the various framework classes and two components which exchange an XML transfer object.

A Java component whose IDL interface contains XML transfer objects can either implement the flat-XML interface or the transparent-XML interface. While the latter is generally recommended, there are cases in which a component developer will not want to make a static choice between these two options; for example, if the XML transfer objects received as parameters of one method will always be routed through to another component or storage system without being inspected, the container would produce an unfortunate performance impact if it would first instantiate binding classes from XML, pass them to the component, and then serialize them again. For such cases, the container offers a hook mechanism that allows components to obtain unparsed XML strings for certain methods, while still providing transparently created binding classes for the other methods of that component.

4. CONCLUSION

We have described a new way of integrating type-safe usage of XML in a distributed system, which has to our knowledge a unique advantage over other middleware frameworks. The usage of XML binding classes is catching on, so we are curious if some similar integration into a middleware framework will be seen in the future.

Despite its hybrid architectural and cultural origin and its ties with ALMA, ACS is flexible and powerful enough to be attractive as a general application framework for a growing number of scientific facilities.⁶ We are looking forward to new collaborations!

REFERENCES

1. G. Chiozzi, B. Gustafsson, B. Jeram, M. Plesko, M. Sekoranja, G. Tkacik, and K. Zagar, "CORBA-based Common Software for the ALMA project," in *Advanced Telescope and Instrumentation Control Software II*, H. Lewis, ed., *Proc. SPIE* **4848**, pp. 43–54, 2002.
2. ZeroC, Inc. *Internet Communications Engine (ICE)*, <http://zeroc.com>, 2004.
3. Object Management Group, *CORBA Components, v3.0 – Full Specification*, <http://www.omg.org/cgi-bin/doc?formal/02-06-65>, 2002.
4. World Wide Web Consortium, *Web Services*, <http://www.w3.org/2002/ws/>, 2004.
5. J. Schwarz, A. Farris, and H. Sommer, "The ALMA Software Architecture." These proceedings (5496-22), 2004.
6. G. Chiozzi, B. Jeram, H. Sommer, A. Caproni, M. Plesko, M. Sekoranja, K. Zagar, D. Fugate, P. D. Marcantonio, and R. Cirami, "The ALMA Common Software: a developer friendly CORBA-based framework." These proceedings (5496-23), 2004.
7. D. W. Fugate, "A CORBA event system for ALMA common software." These proceedings (5496-68), 2004.
8. M. Völter, A. Schmid, and E. Wolff, *Server Component Patterns*, Wiley, 2002.
9. Sun Microsystems, *Core J2EE Patterns: Transfer Object*, <http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html>, 2002.
10. Object Management Group, *CORBA 3.0 – Value Type Semantics chapter*, <http://www.omg.org/cgi-bin/doc?formal/02-06-41>, 2002.
11. D. C. Schmidt and S. Vinoski, "CORBA and XML, part 1, 2, 3," *C/C++ Users Journal*, 2001. <http://www.cuj.com/documents/s=7995/cujcexp1905vinoski/>, <http://www.cuj.com/documents/s=7993/cujcexp1907vinoski/>, <http://www.cuj.com/documents/s=7990/cujcexp1910vinoski/>.
12. Object Management Group, *XMLDOM: DOM/Value Mapping Specification*, <http://www.omg.org/docs/ptc/01-04-04.pdf>, 2001.