The ALMA Common Software — Dispatch from the trenches

J. Schwarz[a], H. Sommer[a], B. Jeram[a], M. Sekoranja[b], G. Chiozzi[a], A. Grimstrup[c], A. Caproni[a,f], C. Paredes[e], E. Allaert[a], S. Harrington[d], S. Turolla[a] and R. Cirami[f]

[a]European Southern Observatory, Garching, Germany;

[b]CosyLAB, Teslova 30, SI-1000 Ljubljana, Slovenia;

[c]Centre for Radio Astronomy, University of Calgary, 2500 University Dr. NW, Calgary AB, Canada T2N 1N4;

[d]National Radio Astronomy Observatory, Socorro, New Mexico, USA;

[e]National Astronomical Observatory of Japan, Osawa 2-21-1, Mitaka, Tokyo 181-8588, Japan;

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

ABSTRACT

The ALMA Common Software (ACS) provides both an application framework and CORBA-based middleware for the distributed software system of the Atacama Large Millimeter Array. Building upon open-source tools such as the JacORB, TAO and OmniORB ORBs, ACS supports the development of component-based software in any of three languages: Java, C++ and Python. Now in its seventh major release, ACS has matured, both in its feature set as well as in its reliability and performance. However, it is only recently that the ALMA observatory's hardware and application software has reached a level at which it can exploit and challenge the infrastructure that ACS provides. In particular, the availability of an Antenna Test Facility(ATF) at the site of the Very Large Array in New Mexico has enabled us to exercise and test the still evolving end-to-end ALMA software under realistic conditions. The major focus of ACS, consequently, has shifted from the development of new features to consideration of how best to use those that already exist. Configuration details which could be neglected for the purpose of running unit tests or skeletal end-to-end simulations have turned out to be sensitive levers for achieving satisfactory performance in a real-world environment. Surprising behavior in some open-source tools has required us to choose between patching code that we did not write or addressing its deficiencies by implementing workarounds in our own software. We will discuss these and other aspects of our recent experience at the ATF and in simulation.

Keywords: software middleware, ALMA, radio astronomy, ACS

1. INTRODUCTION

The Atacama Large Millimeter Array (ALMA), whose construction is currently scheduled for completion in 2012, is a combined North American/European/Japanese project to build and operate a radio interferometer consisting of 66 antennas, sensitive in the wavelength range 0.3 - 10 mm, which will be located at an altitude of 5000m in the Atacama desert of Chile. Routine operation of the observatory will be managed from an Observation Support Facility (OSF) at 3000m, some 30 km from the observing site itself. Centers for data analysis and the support of the observing community will be located at regional centers in Chile, North America, Europe and Japan.

The software system under development for ALMA will support all phases of the operation of the observatory, beginning with preparation and evaluation of observing proposals, dynamic scheduling of approved observing programs, acquisition, on-line calibration and archiving of the interferometric data, through to post-observation

Further author information: (Send correspondence to J.S.)
J.S.: E-mail: jschwarz@eso.org, Telephone: 49 (089) 32006363

processing to produce data cubes of images and spectra and, finally, support for archival research. When ALMA reaches full operations, the software will need to support raw data rates of 6-60 MBytes/s, and thus to archive of order 200 TB/y.

The ALMA construction project is organized into Integrated Product Teams (IPTs) for each of the major elements in the observatory, *e.g.*, antennas, correlator, site. Development of the software and specification of the computing hardware fall under the aegis of the Computing IPT (CIPT). The overall architecture of the software (Schwarz[1]) in turn assigns responsibility for each of these processing phases to a separate subsystem team within the CIPT, and these teams are spread out geographically across institutes in all the countries taking part in the ALMA project.

Enabling such a widely distributed set of developers to produce the pieces of a software system that will run harmoniously in a distributed environment presents considerable technical and managerial challenges. The ALMA Common Software (ACS) addresses the two technical aspects, developmental and operational, by:

- Providing a common framework and programming model which developers at widely separated locations can exploit to build a unified software system;
- Providing a high-level container/component model for the development of distributed software objects, wrapping the underlying Common Object Request Broker Architecture (CORBA) middleware

The architecture and design of ACS have been described in detail in (Chiozzi[2]). Nevertheless, since understanding ACS's implementation of the container/component model and its relation to CORBA is key to following the case histories that follow, we expand on it here.

1.1 Container/Component model

Following Völter,[3] we define a component as a coarse-grained software element that exposes its services, explicitly declares its dependencies on other components and resources, can be deployed independently and requires a runtime environment. Components depend on this runtime environment, the *container*, for necessary services.

A component encapsulates a well-defined piece of the overall application functionality. An application is assembled from collaborating components accessing each other through a well-defined component interface. This interface is technically separate from the component implementation, which may be exchanged without affecting clients.

A container may host one or more components, and where the container runs and which components it will host can be decided at run-time. A container and its components execute in a single process (which, in the case of Java, is a single instance of the Java Virtual Machine) and a single address space. ACS provides container implementations for C++, Java and Python, as well as a Manager that coordinates their requests for access to remote components and services. CORBA ensures that communication between components in the same container occurs via native language invocations, while accesses to components in other containers (either on the same node or on a remote node) are mediated by CORBA's remote invocation facilities.

Components publish a required *lifecycle* interface used by the container to administer them, and a *functional* interface, which specifies the methods offered to clients, including (but not restricted to), other components. All interfaces are written in OMG IDL, the language-independent Interface Definition Language, which supplies the basis for language-independent method invocation by clients. Using their lifecycle interfaces,the container configures components, activates and passivates them over time, checks their state (running, standby, error,...) and can restart one in case of severe problems. The container/component environment is shown in figure 1; further details are given in Sommer.[4]

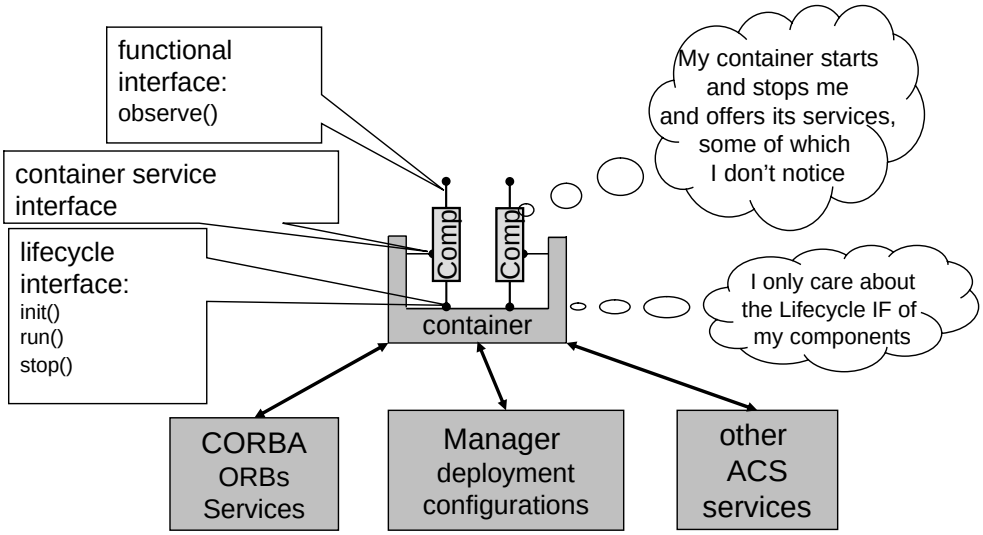

Figure 1. The Container/Component environment (from Schwarz[1]).

1.2 CORBA services provided by ACS

In addition to the Manager and container implementations, ACS provides those additional CORBA services that are needed for the ALMA system.

- Synchronous and asynchronous remote invocation
- Centralized logging system using the CORBA Notify Service and Telecom Logging Service
- Publish/subscribe facility based on the CORBA Notify Service
- CORBA Naming Service
- CORBA Interface Repository (particularly useful for component method discovery by introspection, *e.g.*, by the ACS Object Explorer utility.
- CORBA Audio/Video Streaming Service for the transmission of bulk data to multiple consumers

Open-source implementations of CORBA ORBs and Services have been chosen: ACE/TAO for the C++ ORB and the Naming, Notify, Telecom Logging and A/V Streaming Services, JacORB for the Java ORB, OmniORB for Python, and Mico for the Interface Repository.

1.3 Status

ACS, now at version 7.0.1, has delivered one major release and one or more patch releases each year since 2000. It is largely feature-complete, and is in daily use at the Operations Support Facility in Chile and at the ALMA Test Facility in New Mexico. Most of the current development of ACS, including the setting of its priorities, is driven by the reports of users from these sites and from the Integration and Test group within the CIPT itself. Since the architecture and design of ACS have been treated in numerous papers and presentations in the past, we will use the remainder of this paper to recount some of the more significant events in the front line of ACS development and use.

2. USING THE RIGHT TOOLS

Developers, and particularly developers engaged in work on infrastructure such as ACS, are often admonished not to become fixated on the latest technology fad, but to concentrate instead on just getting the job done: "We already have a tool for that" or "Better is the enemy of good enough". To forestall such criticism, developers may continue to struggle to use what they have, instead of advocating the use of better tools that might make a qualitative productivity improvement. When a tool proves just too inefficient or cumbersome in actual use, however, it may be time to give up on it. Three examples:

2.1 Software configuration control

The ALMA CIPT began work in 2000 using a software configuration control system that had been developed in-house at ESO for the Very Large Telescope project. This system was built around the concept of high-level modules, which were treated as a unit. The system had several drawbacks: tracking different revisions of a single file was difficult or impossible, the system ran only on Linux, was not supported by any of the publicly available Integrated Development Environments (IDEs) or other development tools and, of course, required ESO staff for its maintenance. Moving to the Concurrent Versions System (CVS), as we did in 2002, resolved all these issues in one stroke. Copious documentation and help is available for CVS, and many open-source projects are backed by CVS repositories, making their latest source code available to us should we need it. Not least among the advantages of using CVS is the ample tool support that is available for it. IDEs such a Eclipse and NetBeans and UML tools like MagicDraw all have nearly seamless integration with CVS built into them. Looking back, it is hard to remember why we ever hesitated to make the change.

2.2 The TWiki

After well over a year of frustration in trying to fit freewheeling tecnical discussions into ALMA's Electronic Document System (EDM), which was never designed with this purpose in mind, we set up a TWiki*, a Web-based collaboration platform, for a single subsystem to try out. Within a month, other subsystems were clamoring for their own TWiki areas, and one more month later, CIPT management declared that "resistance is useless," and added its own TWiki. In its nearly 5 years of operation, the TWiki for ACS alone has seen average rates of over 250 topic updates/month and over 10000 topic accesses/month. It is universally recognized within the CIPT as much more suitable for the coordination of distributed teams than the EDM it superseded. As of this writing, there are over 6700 topics on all the CIPT TWiki webs, and the TWiki has become our principal e-collaboration platform. This was a classic case of having only a hammer (the EDM system) and using it to treat everything else as a nail.

2.3 A problem reporting system

The ALMA CIPT also inherited a problem reporting system from an earlier project at ESO. This system needed specialized consultants to make even the most trivial customization and had a Web interface that depended on particular versions of particular browsers. This tool was so widely disliked by those whom it was meant to benefit, that only a small percentage of the project team actually used it to report software issues, most preferring to use e-mail to report and discuss problems. This made defect tracking difficult, and meant that only a small portion of the development staff was kept apprised of the existence of any particular bug. Nevertheless, the view in the ALMA Computing group was that replacement of this system with a more flexible, user-friendly alternative would require more time and effort than the software project could afford.

A decision by the ALMA project to adopt the JIRA Issue Tracking System† project-wide forced the issue. It was compatible with all common browsers, and allowed the individual user considerable freedom to customize the view of those issues he/she found relevant. The software development team made the transition at the end of March 2007, and in the 14 months since, 1456 new issues (comprising bug reports, enhancement requests and task assignments) have been created. During the preceding six years in which the old problem reporting system was used, 1160 issues were created. Even if we concede that the level of software development effort was ramping

*`http://TWiki.org`

†`http://www.atlassian.com/software/jira`

up in the first few years of the ALMA project, the increased use of the problem reporting mechanism since the JIRA system was introduced is eloquent testimony to the difference between a tool that barely does the job and one that is appreciated by its users.

3. TEST ENVIRONMENTS

3.1 Unit tests

3.1.1 Extending the ACS unit test suite

There are over 500 separate tests in the existing suite of unit tests for ACS. These tests are run nightly to verify the continued validity of the code in the Concurrent Versions System (CVS) source code repository. A successful run of all tests is a necessary, though not sufficient, condition for the release of a new version. When an enhancement or bug fix is added to ACS, a test of the fix or new feature is added to the suite; consequently, the test suite is evolving along with the ACS code itself. A recent addition to the collection is described below.

For ACS to support development in C++, Java and Python, it is important to ensure that applications (components) behave the same way, and that they log messages with identical content and format under the same conditions. The design and implementation guidelines for the ACS logging infrastructure had been discussed and agreed over time among the various groups of developers responsible for the implementation in a particular language. However, a detailed analysis still revealed a number of discrepancies between the various languages. That typically originated either from the software authors' individual interpretation of use cases that were not fully specified, or from some corner cases that had not been considered at all. Once these discrepancies were identified and clarified, it became a matter of exposing them in a test-suite.

For that purpose the container log-test suite was implemented. It was built on top of the generic ACS test automation framework, which permits one to execute processes and compare their output with a reference file, after applying some filters to remove *e.g.*, timestamps or host/user dependent pathnames. On the basis of the observed discrepancies, a set of about 25 individual tests for particular behaviour/response has been implemented, which is applied for every language. That way, it becomes straightforward to check for the deviation from the desired behaviour, and to verify whether some item has been fixed.

In the case of logging comparisons, for example, the behaviour can be influenced by *e.g.*, environment variables or configuration database settings prevailing at start-up of the containers. To ensure that all tests are run under the same conditions, each test normally is wrapped by a complete start-up/shut-down cycle of ACS and those parts, typically containers, under test.

3.1.2 Monitoring of unit tests

Until recently, monitoring of resource usage was not part of routine ACS test procedures. It was difficult enough to get one's code debugged and through a steadily expanding gauntlet of unit tests successfully. The tools available for such monitoring required time to learn, and taking this time never quite rose to the top of anyone's todo list. Most such tools also required that the application code be instrumented via the use of special compilation options, which could cause the the target application to behave differently under monitoring than without it. The release of the JMX monitoring tools that accompanied Java 1.6 lowered the entry-level barrier significantly. Entering `jconsole` at the command line allowed one to monitor any and all local java processes with a simple but powerful GUI (monitoring remote JVMs is slightly more complicated). No additional code instrumentation was required. Even the simplest of unit tests, combined with monitoring, could gain value. One such test executed a loop that would load and unload a single component in a container an arbitrary number of times. Monitoring this test for a few hours showed that the container's heap was growing steadily at a rate of 10 MB/h. A heap dump was obtained with `jconsole` and analyzed using `jhat`; it showed two classes with large instance counts, which enabled us to find and eliminate the memory leak. JMX suite, `jstack`, to find a problem in JacORB that was causing the ACS Manager to hang after a connection failure, a problem for which we We used a related tool from the implemented a successful workaround.

These experiences encouraged us to take another look at the memory error detector of `valgrind`, a tool aimed primarily at C and C++ programs, and one that had long been available to us, but which we had scarcely used. When we used `valgrind` to find a memory leak in the ACS alarm system, it helped us uncover a problem in the ACS threading code and resolve another that was causing a container to crash at shutdown in some cases.

3.2 The Standard Test Environment (STE)

The platform on which the ALMA software is integrated and tested for the first time as an end-to-end system is the "Standard Test Environment" (STE), an isolated subnetwork of computers managed as a unit. It provides a configuration of Linux customized to the needs of ALMA, network infrastructure, and user environment to test and run ALMA software. There are several STEs currently in use at the various ALMA development and operations sites.

Lacking access to the observatory's instruments, tests on the STE are necessarily run with some kind of simulation of that hardware. While this deprives these tests of a "real-world" character, it does have the advantage that attention can be dedicated to the higher-level, end-to-end operation of the system, rather than to the myriad problems that arise the first time an attempt is made to merge software and hardware. Interactions among components can be tested and debugged here, as can the configuration and scalability of the various ACS/CORBA services. The use of simulated data allows data formats but also content (*e.g.*, metadata) to be checked and validated long before the software is used in anger.

3.2.1 Mystery of the hung notification service

One relatively straightforward end-to-end system test involves the execution of Scheduling Blocks (SB) for holography, an operation that is used on an antenna to measure misalignments in its optics. An SB is a finite, self-contained set of observing instructions that can, to a certain extent, run on its own. An astronomer prepares a holography project containing a set of SBs that are then processed through the system. As an SB moves through the system, different software subsystems publish events as they start and finish their particular task on ths SB, and other subsystems subscribe to these events to track the SB's status.

We pick up the story at the point at which the Integration and Test team had already successfully run holography projects with a few SBs through the system. An important test of the robustness of the system (which will be required to run constantly round-the-clock during routine operations) would be: how large a project, *i.e.*, how many SBs, could be run through the system successfully? The first time that this was tried, the system appeared to hang during the execution of the 54th SB. Cursory examination of the status of all components, containers and CORBA services in the system did not reveal any obvious status errors. No events, however, were being detected on any of the channels of the CORBA Notify Service. The service itself showed up as a live process, but did not respond to any invocation requests. When we looked at the process with the GNU debugger, the only thing that seemed at all odd to us was the number of threads in the process, 289; before the first SB had been executed, there were only 76 threads in the process.

Two questions required answers:

- Why did the Notify Service fail when it reached 289 threads?
- Were the extra 223 threads normal, or did they represent an anomalous condition? If the latter, where did the excess threads come from?

By monitoring the the Notify Service, we saw that 4 threads had been added to the process following the completion of each SB, reaching 288 after the 53rd SB had been successfully executed. Clearly, some resources were not being released at the end of each processing cycle, and we decided to address this application-level issue before tackling the job of understanding the meaning of the "magic number," 289.

First, we saw that 53 instances of a component that was intended to be created and destroyed with each execution of an SB were still in the system, and that a single client still maintained references to all of them. Since a component may not be destroyed while a client still holds a valid reference to it, these components, though inactive, still held onto system resources including, in our case, subscriptions to events on the channels of the Notify Service. When this error had been corrected, the test completed 70 SBs before failing with the same symptoms as before. This time, 3 threads per cycle were being added to the Notify Service, which showed that the fix had removed one of the event subscriptions that had been created for each cycle.

At this point, the task of determining what part of the system was still tying down these unused resources became much more difficult, because we immediately found ourselves confronted by the fact that the API of

the CORBA Notify Service provides no way to determine the subscribers to an event. While this may be in harmony with the general philosopy of the publish/subscribe paradigm (publishers don't need to know who their subscribers are), it makes finding a rogue subscriber devilishly difficult. Should a component terminate without explicitly cancelling its subscriptions, those subscriptions will continue to live as threads for the life of the Notify Service itself. With no direct way to identify the source of the problem, we had to resort to inspection of the source code of all participating subsystems. After managing to find and fix one such error, we were able to move 104 SBs through the system before the Notify Service hung with 289 threads.

Understanding the particular significance of the number 289 necessitated examining *both* the number of threads running in a process *and* its memory usage at the same time. In the case of the Notify Service, 289 threads corresponded to a use of about 3 GBytes of memory, essentially the limit on a 32-bit Linux process; checking the Linux system documentation, we saw that the default stack size assigned to a thread was 10 MB – this made the process hit its memory limit. This stack size is much larger than is needed for a thread that only needs to manage a consumer subscription and could probably be safely reduced by a factor of 10. Had we started off with a 1 MB/thread stack size, however, we would have taken ten times as long (over 500 rather than 50 SBs) to discover the underlying application problem.

Whichever subsystem component(s) would turn out to be holding on to event subscriptions unnecessarily, it was clear to us that a system infrastructure that collapses when one of its components misbehaves cannot be considered robust. It would have been better for ACS to anticipate and prevent problems rather than to trust that standard procedures or best practices would always be followed by application developers. We needed, at the level of ACS, to:

- Record, at the level of container or ACS manager, the identities of all event publishers and subscribers.
- Use this information to clean up any connections remaining when the subscribing or publishing component terminated.
- Monitor the health of the Notify Service (and, by analogy, any other critical services), alerting the system operator well in advance of any approaching resource roadblocks.

Design work for the first two tasks is currently underway, and we have already gone through two implementation cycles in the development of daemon processes, discussed below, to deal with the need for robust monitoring of CORBA services in general. For monitoring of the details of the Notify Service, the TAO implementation offers an extension that is, however, not part of the CORBA standard. By wrapping this extension in an implementation-neutral interface, we can use it in ACS but leave open the possibility of replacing it with something else in the future.

3.3 The Antenna Test Facility (ATF)

The ALMA Test Facility (ATF) at the site of the Very Large Array on the Plains of San Agustin, New Mexico, was created to serve as the test bed upon which the two ALMA prototype antennas would be evaluated. After this process concluded in May 2004, it became a platform for ALMA prototype instrumentation and software testing. Since September 2007 the ATF has been managed and operated by the Computing IPT, with the principal goal being the testing of the software under development on production or production-like antenna, receiver and correlator hardware. The ATF is run very much like a normal radio astronomical observatory, with a staff of operators, engineers, scientists and (occasionally) software developers.

In spite of the significant number of flaws remaining in the system, it functions well enough to make the ALMA scientific staff eager to use it for observations of astronomical sources, to see what the state-of-the-art ALMA instrument hardware is capable of delivering. There is therefore a certain constructive tension between the demands for system time of developers and astronomers, even though the overall goal of both groups is the same: to ensure that a robust, reliable system is delivered for the commissioning of the observatory.

The advent of the ATF as a quasi-dedicated test bench for software has been a game-changing event in the life of the CIPT in general and the ACS team in particular. Every day software is tested against the expectations of

real users (both operators and astronomers) on real hardware, and, in general, without the presence of those who wrote it. Problems get reported, therefore, from the point of view and in the language of users, not developers. Add to this the fact that the system has been debugged enough by this time so that many problems that do surface are hard to reproduce and appear intermittently. By the time the ACS team sees the software trouble report, the system has probably been restarted at least once, and the conditions that led to the problem are no longer present. The operators and users at the ATF cannot be expected to carry out complicated debugging procedures using diagnostic tools intended for the expert developer, and even less so when there is pressure to get the system back up again so that astronomers can proceed with their observations. Extended visits to the ATF by the ACS team, and the development of canned debugging procedures and scripts have turned out to be the most effective way to address issues of this type. We have also experimented with remote debugging via secure login and/or screen-sharing, but at this stage in its development, the security of being able to look out the window and see that the antennas are behaving is essential; as the system becomes more solid in the future, we expect to use these techniques much more.

3.3.1 Stalking the zombie container

Each antenna at the ATF consists of a hierarchy of hardware devices, each represented as a software component, and all controlled by a single on-board Linux-based computer running a real-time kernel. An ACS C++ container provides lifecycle management, logging, monitoring and remote access services to the components. When the container crashes or becomes unresponsive, any observing in progress comes to a sudden and ignoble halt. Almost from the start of ATF operation in September, we began seeing reports variously described as "hangs," "crashes" or simply "excessive time" to load a component by this container. The frequency with which the problem occurred varied between a few times a day and once a week; without a clear characterization of the problem – was it really a crash or a hang? were we seeing one problem or several? – even this was hard to pin down. What *was* clear, however, was that all logging output from the container and its components suddenly stopped at about the time the hang/crash/timeout (pick one) was noticed.

We could not reproduce this problem on the STE, and even our attempts to do so at the ATF via remote debugging were unsuccessful. Our only hope was to get more information from the operations staff.

We asked them questions: is the container seen by the operating system? (It was.) Could any of the components within the container be accessed via the "Object Explorer" ACS tool. (They couldn't be.) It became clear that most of the incidents reported had the defining characteristic that the container process was alive but somehow blocked. We asked that upon recurrence, the operators use gdb to produce a thread backtrace to the screen, copy it and mail it to us. Unfortunately, we fell victim to not following our own advice we give our users: read the manual! Had we done so, we would have quickly found the simple commands needed to obtain the needed backtrace as a text file and could have then packaged them into a shell script that the ATF operators could use. We paid dearly in lost time for this neglect. By this time, the patience of our scientific users, as well as their confidence that we knew what we were doing, were wearing thin.

Meanwhile, at ESO in Garching, one of us was on the trail of another, apparently unrelated, problem reported from the ATF. It would occasionally take an excessively long time, 10 minutes, to load a single component in the antenna container (load time should normally be less than one second). This problem, however, *could* be reproduced on the STE, and we were able to find its source in short order: every time a log message was generated in the real-time application code, a new static Logger was created, and with it, a new mutex. (Normally a Logger is created when a component is activated and used by the component for all its logging needs; in addition, the container maintains a logger of its own.) Soon the container was swamped with Loggers and mutexes. There were two ways to resolve this issue, and both were implemented: the application code was changed to use non-static logging methods, and ACS was changed to supply the unique instance of a single static Logger, the container's "global logger," for all static logging invocations.

As this fix was being checked into CVS, two of us were on-site at the ATF, where it seemed that our presence was all that was needed to make the antenna container hang again. The backtrace of all container threads showed that many of them were waiting to get access to a Logger instance, which was itself protected by a mutex. The summary of our first day at the ATF was:

- There is a problem with logging
- A deadlock condition blocks the logging methods so they do not return any more.
- The container is alive, it receives commands, but each method blocks as soon at it tries to send a log, which makes it seem that the container is dead.
- Finally, when all CORBA threads in the ORB's pool are used up, the container really dies, does not respond to pings from the manager and everything collapses.

Studying the code did not bring Enlightenment: we could find no way that the mutex in question could be held for more than a fraction of a second. We did see, however, that a surprisingly large number of logger instances existed in the container. Our colleague in Garching suggested that his changes for the long component load time problem might have a significant influence on our results; when we applied his modifications to the code we were using at the ATF, the container obligingly stopped hanging. Although we still do not understand exactly how the deadlock could have developed, several months of intensive use at the ATF have gone by since these changes were introduced and the container hangs have not reappeared. We suspect that some interactions with the real-time kernel of the operating system brought on by the rapid creation of an excessive number of Logger objects and mutexes might have been the proximate cause.

3.3.2 Capacity and performance of the logging system

ACS received a rather succinct list of requirements for the logging system it was supposed to provide:

- Applications must be able to log timestamped records of: execution of actions, status, anomalous conditions.
- Short term persistence of logs shall be provided by having the total size and/or time span of such logs adjustable. Long term persistence shall be provided by a backup policy into persistent store.
- It must be possible for log messages to be searched and filtered (both in input, *e.g.*, to avoid flooding of identical messages and in output).
- Use the implementation language's standard loggin API if one exists.
- The user must be able to query the persistent store by time range, message, type, program, etc.
- The logging system must be distributed and platform independent.

Notable for its absence in the above list was any specification of the logging rates that the system would be required to support. In practice, this left individual developers free to use logging as liberally as they liked; in application unit testing, logging did not appear to be a significant perturbation. When all subsystems started contributing to the logging load, however, logging was revealed as anything but cost-free. The ACS team now looked at the possibilities for performance optimization. Since log messages are passed around the system as XML-encoded strings, we tried to improve performance by using IDL structs; to our surprise, we found that logging throughput was degraded by a factor of three. Using the Wireshark network protocol analyzer (http://www.wireshark.org/), we were able to see that the struct-encoded log messages actually required *more* space than their XML counterparts. The reason for this is that the CORBA Notify Service on which the logging system is based encodes the IDL structs as CORBA Any types, which include more metadata than the XML strings; consequently, this attempt to improve logging performance had to be abandoned.

While we were experimenting with ways to speed up the logging system, operators and astronomers at the ATF were looking for ways to slow it down. They reported that *too many* logs were flooding the operator's console and making the logging system useless for them. Although log messages could be filtered by severity (*e.g.*, TRACE, DEBUG, INFO, WARNING, ERROR, EMERGENCY), an operator's "trace" message might be an application developer's "warning," and it was the developer's decision that determined what went into the source code.

To solve this problem, ACS defined an "Audience" field in the log record, intended to indicate whether a message was meant for the operator, and therefore should be displayed on the screen, or was, instead, only relevant for developers and should be saved, but not displayed. For this facility to be used effectively, however, it would be necessary that all log messages destined for the operator be retrofit with this new field, a process which is still underway. In any case, we were convinced that the logging system was not the answer to the operator's needs but rather, like a ship's log or an airline flight recorder, was meant for *post facto* troubleshooting and analysis. Another mechanism had been provided by ACS to alert operators to conditions or events that require immediate attention: the alarm system (Caproni[5]), based on the Laser alarm system developed for the Large Hadron Collider at CERN. Although log messages still scroll by on the operator's console, more and more out-of-normal conditions are being signaled to the operator via alarms.

Support of after-the-fact troubleshooting was, however, the main task of the logging system, and it was becoming critical at the ATF. An antenna would occasionally slew off in a direction not at all related to the one intended. The control software group believed it was a hardware problem; the antenna hardware group thought that the software had miscommanded the antenna. To determine which hypothesis was correct, the control software group wanted to log many of the commands sent to the antenna every 21ms on the real-time bus. The plan would do this on up to 10 antennas at a time, at a rate per antenna of 250-500 logs/s. With a typical log record length of 400 bytes, this would yield a log rate from the antenna of about 200KB/s, and from 10 antennas, of about 2 MB/s, with the other 50 or so antennas remaining logging at about 10% of this rate, adding another 1 MB/s to the total of 3 MB, which would be one half of the 6 MB/s average rate for *science* data. Writing this data to a database, while technically possible, would not be cost-effective, particularly since this high-rate logging data is only useful for debugging for a few days after it is acquired. Our ORACLE DBMS installation at the ATF, for example, could only parse and ingest about 1000 such log messages per second when that was its *only* task.

Should it consume a quantity of resources comparable to those used by the processes that it is monitoring, logging itself would become a driving factor in the performance of the system. We concluded that sustaining such a high logging rate would distort the project's priorities and resources. It was mutually agreed among all parties that rates of about 250 logs/antenna/s would be supported at the ATF, and that once in Chile, we would not attempt to do this on all antennas at once. In collaboration with the subsystem team responsible for the ALMA Archive, who did the implementation, and following discussions with the control software team, we came up with a two-pronged approach to handling high-rate logging. Those logs above a certain severity level would be routinely stored in the Archive. Those below that severity level, including those generated from low-level monitoring, would be written to a circular buffer of files on a central disk, and would be available for inspection for an amount of time limited by the number of files in the buffer and the size of each file, both configurable at runtime. This solution allowed developers access to very detailed logging information, and at the same time, ensured that the logging rate would not be allowed to limit overall system performance.

When the logging system cannot keep up with the rate of log message input, either it will eventually run out of memory as its buffers grow too large, or it must discard lower-priority messages in order to stay alive. Likewise, a slow client can prevent the logging system from emptying its queue, also causing out of memory errors sooner or later. Part of our current activity is dedicated to configuring the logging system so that it can recognize when its queue is growing dangerously large, alert the operator, and discard messages until a stable equilibrium has been re-established. This feature will be available with the next major release of ACS.

3.3.3 Bringing the system up (and down) reliably

With a system of hardware and software still under development, it is important to be able to bring the system down and up when a problem cannot be diagnosed and fixed on the spot. It is extremely frustrating for users if they can't just "press the reset button" when something goes wrong; such reset problems occurred often enough to become a problem during large-scale testing at the ATF. The ACS infrastructure of services and containers gets started on many computers upon request from a central machine that runs the Operator Master Client (OMC), the primary user interface to the running system. The original mechanism for this was to spawn remote ssh processes to run a number of start and stop shell scripts on the remote machines. Unfortunately, this tied the remote processes to the fate of the OMC: when the latter was brought down for any reason, the remote

processes would hang. Moreover, accessing the output and monitoring the status of these processes from the OMC was a tedious job in itself. The use of ssh turned out in the end to be so unreliable that we came up with a completely different design for system startup: on every machine, a CORBA-aware daemon process is started at system boot time, and the OMC connects to this daemon and can command it to start or stop the services and containers as needed. The first version of the daemons simply acted as a remote script runner, but we have been enhancing the daemons to use more direct start/stop calls for ACS executables, with no change to the daemon API. This work will be complete by the next major release of ACS in the Fall of this year. Use of the daemons has eliminated the problems that were associated with ssh: the daemons exist independently of the state of the OMC, have direct access to the output of the scripts they run and have a well-defined interface to the OMC. The entire ALMA software system can be brought up and down repeatedly. Beyond managing the lifecycle of ACS infrastructure processes, the daemons turned out to be very useful for monitoring the processes they manage, and to raise alarms that the ACS alarm system then forwards so that they get displayed to the operators. Both the smart monitoring that can employ special knowledge of the ACS processes that the daemon handles, as well as interfaces tailored to the OMC and the alarm system are advantages of using custom daemons instead of off-the-shelf solutions such as SNMP.

The ACS daemons, with their custom checking and error reporting, have also helped us to sort out problems with asynchronous execution, where the calling OMC application wrongly assumed that a particular ACS process had been started or stopped successfully after the call returned from the daemon. When the unloading of real-time kernel modules fails on antenna or correlator computers, the ACS containers may enter an uninterruptible state; when this happens, the only remedy is to reboot the process's host computer. The daemons can be programmed to recognize this situation and to initiate the reboot.

4. CONCLUSIONS

The lessons that we have learned from the experiences described in this paper are not particularly new or surprising. However, hours of reading books about or listening to lectures on best practices can't compare with experience in producing what Freud called "insight with affect." What we won't soon forget in the next, critical years of ACS testing and improvement are that we must:

- Include resource monitoring routinely in our unit and integrated tests.
- Learn how to use our tools properly; even developers need to read the manual.
- Make our infrastructure resilient in the face of client errors; clean up after others who don't.

ACKNOWLEDGMENTS

We are grateful for the help and collaboration of our colleagues in the ALMA Computing IPT's Integration and Test team, particularly P. Sivera, M. Pasquato and T. Powers.

REFERENCES

[1] Schwarz, J., Farris, A., and Sommer, H., "The ALMA Software architecture," in [*Advanced Software, Control, and Communication Systems for Astronomy*], Lewis, H. and Raffi, G., eds., *Proc. SPIE* **5496**, 190–204 (2004).

[2] Chiozzi, G., Jeram, B., Sommer, H., Caproni, A., Plesko, M., Sekoranja, M., Zagar, K., Fugate, D., Marcantonio, P. D., and Cirami, R., "The ALMA Common Software: a developer friendly CORBA-based framework," in [*Advanced Software, Control, and Communication Systems for Astronomy*], Lewis, H. and Raffi, G., eds., *Proc. SPIE* **5496**, 205–218 (2004).

[3] Voelter, M., [*Server Component Patterns*], John Wiley and Sons, New York (2003).

[4] Sommer, H., Chiozzi, G., , Zagar, K., and Voelter, M., "Container-component model and XML in ALMA ACS," in [*Advanced Software, Control, and Communication Systems for Astronomy*], Lewis, H. and Raffi, G., eds., *Proc. SPIE* **5496**, 219–229 (2004).

[5] Caproni, A., Sigerud, K., and Zagar, K., "Integrating the CERN LASER Alarm System with the Alma Common Software," in [*Advanced Software and Control for Astronomy*], Lewis, H. and Bridger, A., eds., *Proc. SPIE* **6274** (2006).