A Distributed Software Correlator at the Rapid Prototyping Array

G. R. Harp*
Allen Telescope Array, SETI Institute, 2035 Landings Dr., Mountain View, CA 94043

ABSTRACT
The Rapid Prototyping Array (RPA) is a toy radio telescope located 30 miles from U. C. Berkeley in Lafayette, CA. It serves primarily as a software development test bed for the Allen Telescope Array (ATA). We have developed a minimally functional prototype of the ATA control system founded on C++, Java, and a CORBA-based distributed architecture. The system controls RPA pointing, electronics, and data processing, culminating in a real-time software correlator (i.e. an imaging system). This system has helped us characterize our preliminary design of the ATA control system. Overall, the distributed architecture provided successful, versatile control supporting a wide range of experiments from satellite tracking to beam characterization to celestial observation. However, some weaknesses in the CORBA communications layer were identified, and the synergies of mixing C++ and Java were balanced by paradigm mismatch between the languages. We learned that Java was as fast as C++ and supported more ready-made libraries. Based on these experiences, we changed our design to eliminate CORBA and build a pure Java system at the ATA, which is now under development.

Keywords: ATA, CORBA, C++, Java, software, distributed, architecture, radio astronomy, correlator

1. INTRODUCTION

The Allen Telescope Array (ATA) is a new radio interferometer under construction at the Hat Creek Radio Observatory in northern California, and is a joint venture of the SETI Institute and the U. C. Berkeley department of Radio Astronomy. It shall consist of 350 offset Gregorian dishes with 6 m diameter (collecting area ~10000 m²) operating over a frequency range of 0.5-11.2 GHz. The ATA shall instantaneously image a large field of view (2° diameter at 1 GHz) while simultaneously providing 15 independently tunable single-pixel “beams” pointed anywhere on the sky. As such, the ATA is explicitly designed to concurrently support 16 independent users at all times.

While beginning the control system design for the ATA, we noticed that there are a large number of devices that are functionally identical (e.g. 350 antennas, (350 * 16) = 5600 data processing channels, etc.). This situation suggests a distributed architecture that leverages parallel processing of many semi-autonomous computers. The philosophy of the ATA emphasizes a modular design, partly because the final number of elements is not fixed – construction may continue until there are 1000 elements or more. We therefore seek a design that gracefully absorbs new elements as they are added (or drops them in case of failure).

Indeed, the ATA is so complicated that we decided it would be best to build a prototype software control system to prove our design choices before attempting the real thing. For about 1 year, the ATA software team (R. F. Ackermann and myself) developed a simple radio interferometer using seven off the shelf (but customized) satellite antennas at the Rapid Prototyping Array (RPA) in Lafayette, California. This was also an opportunity for the software team members to become acquainted with the methods of radio astronomy and to gather requirements for the ATA. The time spent on this throwaway system was more than recovered in our gained experience and by proving our design choices.

With only a back of the envelope design process, we assembled a toy model of the ATA control system based on the RPA hardware. We used the same tools and architecture that were planned for the ATA. What follows is a description of our experiences and how they lead to significant modifications of the ATA software design. In section 2 we outline some characteristics of the RPA and our proposed software design.

* Email: gharp@seti.org.
Section 3 describes RPA software as it manifested, especially focusing on the most complex component: the correlator. Section 4 shows some data characterizing this system and section 5 gives a critical analysis. In section 6 we propose design modifications to overcome limitations mentioned in 5, followed by conclusions in 7.

2. INITIAL DESIGN

Our initial design for ATA control is schematically depicted in Figure 1. The system is composed of a large number of software applications or “components,” each of which is relatively self-contained. Many components (designated servers) provide services to other components in the system. These services are specified in published interfaces and are invoked by remote procedure calls on the servers. Other components (designated clients) call upon these services to make astronomical observations. Some components are both clients and servers, and we shoot for a rather flat hierarchy among components (to reduce coupling).

![Figure 1: Schematic of initial design for ATA control software.](image)

Most of the low-level servers, especially those that interact with hardware, are written in C++. Higher-level servers and clients are in Java. Messaging between components is managed by CORBA, which has a built-in mechanism for specifying interfaces and making remote procedure calls. The ATA has a requirement to allow remote operation by engineers, which is accomplished by a web interface (see Figure 1). System startup files and low volume observation data (e.g. weather station data) are stored in a database and accessed through a Java server via JDBC. All the component types depicted in Figure 1 are implemented at the RPA.

This particular system design is not particularly original for a distributed computing system, nor indeed even for telescope control software. For example, the SOLIS solar telescope\(^1\) employs a very similar architecture. More generally, distributed architectures substantially similar to this one have been employed at the Green Bank Telescope\(^2\), and are under development for the Atacama Large Millimeter Array\(^3\) and the Gran Telescopio CANARIAS\(^4\), to name a few.

3. IMPLEMENTATION

Like the ATA, the RPA is jointly operated by U. C. Berkeley and the SETI Institute. It consists of seven 3.6 m, prime focus radio dishes. The antennas are equipped with two orthogonal linearly polarized receivers
sensitive between 1400-1630 MHz with a system temperature of about 290 K. Electronics perform a two-stage down conversion resulting in a net 10 MHz nominal bandwidth. Whereas astronomical data on the ATA is processed by custom hardware, on the RPA they are digitized and processed by general-purpose computers. The baseband signal (0-15 MHz, two polarizations) is digitized at 30 MS/s/pol by a Sun Ultra-60 computer (UltraView analog to digital converter (ADC) installed) – with one computer per antenna. These computers are interconnected on a 100 Mb/s local area network comprising several other computers. The ADC computers are labeled RPA1 – RPA7 (number associated with respective antenna), and communicate via Ethernet with an eighth Ultra-60 labeled RPA0, a Linux/Intel computer named RPAWEB and a Windows/Intel computer named RPACAM.

![RPA Prototype Software](image)

**Figure 2** Schematic of how two RPA clients (software spectrum analyzer and satellite tracker) interact with RPA server hierarchy.

As an example of how the system works, consider Figure 2. Here two clients (software spectrum analyzer, satellite tracker) interact with the server hierarchy to perform their functions. Each box indicates a separate application (process) running on various machines. The black arrows indicate CORBA connections between components. Servers colored gray must be run on a specific machine because they talk to hardware. The satellite tracker interacts with two servers to acquire a satellite’s position and convert it to coordinates suitable for the hardware. This position is then sent to the antennas, causing them to track the satellite. At the same time, the spectrum analyzer client controls electronics and data acquisition hardware to acquire frequency spectra from the satellite currently being tracked.
Software Correlator

In another example, we examine in detail one particular client application called the correlator. The correlator acquires time series data from each antenna and calculates the cross correlation of this data between all pairs of antennas. The cross correlation for a given pair is associated with a real space vector starting at the first antenna and ending at the second. A spatial fourier transform of these cross correlations results in an image of the object under evaluation. Spectral analysis can be achieved by fourier transforming the time series data before cross correlation (called FX architecture). Given the existing hardware, we decided on an FX approach. Since the data from each antenna passes through a dedicated computer, it makes sense to have that computer perform the FFT before shipping data out for correlation.

It was immediately recognized that the maximal processing speed of the computers and network fall far below the acquisition rate of the ADC’s. The total acquisition rate is 30 MS/s × 8 bits/Sample × 2 polarizations × 7 antennas = 3360 Mb/s. By comparison, one expects of order 50 Mb/s net transfer rate from our network. Likewise, the Ultra-60’s (300 MHz) cannot perform significant processing on the 60 MS/s data stream from the ADC (only 5 instructions per sample!). The time series fast fourier transform (FFT) dominates the processing time, and benchmarks using the FFTW library of FFT routines showed that the Ultra-60’s could process ~ 200 kS/s/channel (using 512 sample FFT’s), where we use “channel” as shorthand for the data stream from a single antenna and polarization.

Both the network and FFT limit the amount of data that can be processed. The data processing is performed in a component called “Data Server” and is accomplished as in Figure 3. For each channel, N consecutive samples (typically N = 512) are captured. Approximately 1000 × N samples are then skipped, after which another N samples are captured (typical capture rate = 40 kS/s/channel).

![Figure 3 Functional diagram of data server, including data reduction approach.](image)

In a separate thread each 512 sample “minibuffer” is Fourier transformed to complex amplitude over the frequency range 0-15 MHz. The analog bandpass filter is such that the lowest and highest 2.5 MHz are corrupted by aliasing, so we ship only half the Fourier transformed data, the range 3.75-11.25 MHz. In total this program reduces the 30 MS/s × 8-bit data stream from a single channel to 20 kS/s × 32-bit floating point. This data are time-tagged and shipped across the network in blocks comprising 32 mini-buffers or 16384 samples.
Data blocks are captured by the Data Queue program running on RPA0. This program assembles one data block from each antenna into a data structure. Because TCP/IP transmission is non-deterministic, it is necessary to time-align the blocks from different antennas. When a data structure is complete with data from all 7 antennas, it is shipped to various Data Subscribers, including the Correlator and Data Monitor programs. The Correlator performs the cross correlations and stores them to disk. The final image formation step is performed off line.

![Diagram](image)

**Figure 4** The Data Queue time-aligns the data streams coming from each Data Server and assembles data structures containing signals from all antennas. These structures are then shipped to all subscribing clients, such as the correlator or a monitoring client.

This system was effective in gathering data from satellites and various celestial objects including the sun and Cassiopeia A. For example, Figure 5 shows stills from a movie created as the sun drifted through the primary beam pattern of the RPA antennas (held fixed). The sun’s position is indicated by a white arrow in each image. The aliased features which surround the sun in a hexagonal pattern result from the fact that the RPA antennas are laid out in a regular array. These features can be removed by a well-known data processing method called “CLEAN.” This is discussed in greater detail in ref. 1, and both clean and dirty movies of the sun can be found at the same website.

![Images](image)

**Figure 5** Radio images of the sun taken at the RPA. The sun’s image position is indicated by arrows. The hexagon of bright features surrounding the sun is due to aliasing caused by redundancy in the layout of the RPA antennas. The gray circle indicates the RPA primary beam diameter.
4. CHARACTERIZATION

A few tests were performed to identify the bottlenecks in this correlator and estimate its ultimate speed. Beginning with the FFT, a simple program benchmarked the time to process 1 MS using the FFTW floating point library. The samples were broken up into mini-buffers of length $2^N$ and each mini-buffer was fourier transformed independently. In this simple benchmark, the spectral resolution of the result is $N$-dependant; in practice, $N$ would be a user-controlled parameter. This test was performed on one of the antenna computers (Sparc 366 MHz) and on an Intel laptop (PIII 600 MHz). As seen in Figure 6, the calculation time for small FFT lengths is dominated by setup for the FFT; hence the time is reduced with increasing length. For large FFT lengths, we observe the logarithmic increase predicted by the asymptotic limit that the time for a single FFT goes like $N \log (N)$. Over most of the range of measurement, the processing time is under 2 seconds. Because each computer must process two channels (and performs other tasks), we concluded that a sustained processing rate of about 200 kS/s/channel is possible with these computers.

![Time to Process 1 MS vs. FFT Length](image)

Figure 6 Time to process one million samples using FFT’s of various lengths on two different computers (using FFTW library).

Switching to an integer FFT, we expect this number could be improved by a factor of a few. Likewise the ADC cards fit a standard PCI bus, so they could be moved to 2 GHz Linux / Intel boxes at modest cost. This too should improve throughput by a factor of a few. Thus it seems reasonable to suppose that 1 MS/s/channel would be achievable with modest investment.

A more serious bottleneck is the network. A benchmark was created that tests the maximum data transfer rate that can be achieved using CORBA over Ethernet in the current system (Figure 7). In this test, a server on one antenna computer sends a block of data to a client RPA0 repeatedly, measuring the time required for transfer as a function of the block size. For large block sizes, this rate saturates at a level determined by the network protocol and hardware. Notice that this is substantially lower than the absolute hardware limit (100 Mb/s).
Figure 7 indicates that no more than 30 Mb/s can be reliably transferred with the current architecture. After FFT, each sample comprises 32 bits, so we’re left with about 1 MS/s. Considering that each computer processes 2 channels and all 7 antenna computers must communicate data over the same network, we estimate that no more than 70 kS/s/channel can be achieved. This is in line with experimental results, as we successfully ran the software correlator at speeds up to 50 kS/s/channel.

![Data Rate vs. Block Size](image)

**Figure 7** CORBA data transfer rate (between machines, over Ethernet LAN) as a function of the size of the data block being exchanged. The maximum, average, and minimum rates are shown for 100 attempts to send data.

Some of the Figure 7 data is plotted differently in Figure 8, which shows the average transfer time as a function of block size. Extrapolating this data to zero block length gives us an estimate of the time required to set up a CORBA transfer on this system (about 10 ms). Note that the CORBA connection was held open throughout the test. This figure emphasizes the importance of combining multiple small data blocks into large packets for transfer, especially over a CORBA network.

To achieve the maximal data processing rate with the current architecture, the network bandwidth could be enhanced in three ways:

1. Instead of transmitting 32-bit floating-point values, we could gain a factor of 2 by transmitting in a 16-bit representation.
2. Upgrade the 100 Mb/s network to 1 Gb/s, gaining a factor of 10.
3. Set up multiple networks, potentially having exactly one network per antenna, gaining a factor of 7.

Implementing these changes has the potential of raising the bandwidth to 7 MS/s/channel, which is higher than the FFT data processing limit. Thus, with the current ADC’s, an upgrade to the computer hardware...
and networks, and a finite software development, it is reasonable to expect that a correlator with 1 MS/s/channel bandwidth can be built using the present architecture.

**Figure 8** Average time to transfer a single block of data via CORBA / Ethernet in the present system, as a function of block size.

It is difficult to put this number into perspective, but consider that eight, 2 GHz Linux boxes alone would cost about $16,000 at current prices. This can be compared to the ATA correlator currently being designed at U. C. Berkeley based on custom hardware. The Berkeley correlator processes 100 MS/s/channel from 350 antennas, or 5000 times as much data\(^7\), but its expected cost is only a few million dollars (~200 times more money). Based on this analysis the RPA correlator is not very cost effective. But as a prototype system for the ATA control software it has been quite valuable, as described in the next section.

5. CRITICAL ANALYSIS

Now we review some enduring impressions from our prototyping experiences and feed these back into design decisions for the ATA control system. On the whole, we believe the RPA prototype software was very successful. With a time investment of two men, working part time over about a year, we succeeded in creating a functional radio observatory at the RPA. The antennas move, track satellites and celestial objects. The RF and IF electronics are under software control, as is data acquisition, and a real time interferometric imager was created. Other characterization experiments were performed including antenna beam pattern measurements and a survey of radio frequency interference at the RPA.\(^8\)

We believe these successes are attributable to the choice of a distributed computing architecture and object-oriented programming styles. By limiting interconnections between software components, each component remains self-contained and relatively simple. Thus each software component can be debugged in isolation.
from the rest of the system. There were a few inter-component dependencies that required debugging, especially in the software correlator, but by holding such dependencies to a minimum it is possible to produce effective software in a relatively short time.

Another advantage held by our group is that we are a very small team (two members). Thus very little time was spent in meetings discussing design options. We worked together in the RPA control room about twice a week, and this provided enough interaction time to keep our work synchronized and prevent redundancies.

Our development tools have mostly worked well. We use a variety of programming IDE’s on various platforms but frequently choose Source Navigator because of its cross platform availability and capability for supporting both C++ and Java. At the beginning we strongly considered using a UML tool such as Rational Rose for code generation and reverse engineering, but ultimately rejected this approach because the tools were too hard to use. For CORBA ORB we chose Orbacus, and were very happy with this choice. Orbacus is a high quality product, freely available for non-commercial use, with excellent documentation and support. Source code revision and backup was handled by CVS, which has worked very well. Finally, our initial choice of database was the open source version of Oracle’s product for Linux. Although powerful, this product was difficult to administer, so at the ATA we will probably use MySQL.

The biggest breakdowns in the RPA software were all associated with CORBA in one way or another. First of all, CORBA is typically used to make a long term, point-to-point connection between two software components. In order for this connection to succeed, both components must be up and running. In a hierarchical system such as ours, this introduces order dependence into how various software components must be started up. At the same time, if one software component goes down, it is typically required to restart the entire system because CORBA does not support the ability to automatically reconnect.

Another inconvenience is that CORBA interfaces are fairly brittle. If one adds a method to an interface, all parts of the system that use that interface must be recompiled even if they don’t use that method. This limitation is really attributed to C++ more than CORBA, but exists nonetheless.

The big advantage provided by CORBA is that it provides a programming language independent messaging system. We found that Java and C++ components connected seamlessly in CORBA, with all data marshalling automatically supported. However, this wasn’t as beneficial as expected. Initially we expected that it would be easier to write code that interacts with hardware in C++ because most hardware drivers are written in C. Yet nowadays most devices provide higher-level interfaces. Parallel and COM port devices can be directly controlled Java using its Communications API. Many electronic devices support a GPIB interface. These are conveniently controlled by a GPIB interface box that is controlled in Java via a COM port. The only RPA device requiring a C driver was the ADC card. Recently we wrote a JNI interface to this device so that now, every RPA device can be controlled directly in Java.

In all benchmarks we have performed, we have never noticed a significant difference in processing efficiency between C++ and Java. This is due to just-in-time (JIT) compilation technology, which compiles frequently used sections of Java bytecode down to native instructions, on the fly. In several instances, the Java implementation of numerical processing routines was actually faster than an equivalent implementation in C++, probably because the compiler is better at optimization than we are.

6. DESIGN MODIFICATIONS

In light of the points mentioned above, we altered the design of the ATA control system. To begin with, we choose to minimize the use of C++ in our development effort and write the entire system in Java. Where Java needs to interact with hardware through native calls or with legacy C code, JNI interfaces are created and C++ is used as the glue between JNI and native code. As an example of the latter, we recently created a JNI interface to the FFTW library.
By eliminating pure C++ components from the architecture, the main reason for choosing CORBA is abolished. Given the weakness of CORBA in supporting robust connections described above, we re-opened our examination of messaging systems. Ultimately we found that no pre-existing messaging system has the features of order independence and self-healing that we desired. At the same time, the standard Java API’s including Object Serialization, Self Reflection and Discovery greatly simplify the task of writing your own messaging system. For example, Object Serialization takes care of all marshalling and demarshalling code in a pure Java system. This lead to the creation of the Java Simple Distributed Architecture (JSDA), a hand rolled solution for messaging adopted at the ATA. A future paper will focus on a detailed description of JSDA.

Another element missing from CORBA is the concept of resource ownership. Any component with access to the CORBA network has the capability of calling upon any other one. By developing our own distributed computing network, we have the opportunity for introducing “security” directly into the infrastructure, which will prevent e.g. multiple clients from attempting to take control of the same device.

As JSDA provides a self-healing network for distributed computing, a pure Java system has other advantages. Because Java is not statically linked, small changes to code (such as adding one method to an interface) typically require compilation of only that single class. Because only one programming language is involved, program maintenance is simplified. It is no longer necessary to remember the slight but critical syntax differences between C++ and Java. Only one set of container API’s need be remembered, and so forth.

The features of C++ that we miss most are 1) multiple class inheritance and 2) templates. These two features can greatly reduce code redundancy. In fact, by using multiple inheritance it would be possibly (though not simple) to introduce a self-healing reconnection layer between CORBA and all C++ software components. This would be another solution to the problems described in section 5 if one chose to implement a pure C++ system. Unfortunately, introduction of an abstraction layer between CORBA and Java is not easy because Java does not support multiple inheritance (or by extension, mix-in classes). While C++ provides many powerful API’s (such as the Standard Template Library), Java provides more. Also Java does not carry the baggage of being backward compatible with C, as does C++. For these and other reasons, the choice between a pure Java and pure C++ system was fairly easy to make.

7. CONCLUSION

When we began the ATA development effort, we were faced with the task of developing a highly complex control system that is in many ways unique. Thanks to the foresightedness of the ATA steering committee and management, we were also lucky to have the opportunity to experiment with another, pre-existing radio array (the RPA) that was almost totally unutilized by any other parties. Although ATA software development is only just beginning, the RPA experience has been extremely helpful in testing ideas and proving the tools we plan to use. Consider that even at the time of writing there does not yet exist a movable ATA antenna to which our software can connect. Because of the RPA, we already have concrete experience controlling a radio interferometer.

Not only that, but we have already passed through a complete development cycle with the control software. After implementing the RPA, we discovered weaknesses in our original architecture and now have the opportunity to correct them. Additionally, much of the software developed at the RPA can be adapted or re-implemented to operate in our new environment with little effort. Thus the first sections of ATA software are being assembled rapidly.

The other important lesson learned is that mixing programming languages is not as effective as it sounds. CORBA provides the capability to mix C++ and Java in a fairly convenient way, and is generally successful. However, human beings have more difficulty switching paradigms. In the long run, code maintainability is strongly enhanced by restricting the languages to a set of one. For our needs, either C++ or Java would have sufficed. However, it is well known that Java software is developed more quickly and requires less effort to maintain than equivalent software written in C++. This principle is borne out by our experience. Consequently we choose to implement ATA control as a pure Java system.
Acknowledgements: As with any significant project, many people contributed to building and implementing the RPA. For example, D. C.-J. Bock, D. D. DeBoer, G. C. Bower, J. W. Dreher and many others designed and installed the RPA hardware and electronics, and created simple software that demonstrated its capabilities. Special acknowledgement is owed to R. F. Ackermann, the other member of the ATA software development team. Ackermann and I shared equally in the design and development of the software described here.

8. REFERENCES

3. Many documents describing the ALMA software may be found at http://www.alma.nrao.edu/development/computing/.
6. The FFTW library of floating point FFT routines can be found at www.fftw.org.
7. The number of antenna pairs rises quadratically with the number of antennas. So for the correlation part of the data processing the Berkeley correlator must perform 250,000 times more calculations than the present correlator.
15. Admittedly, the JNI interface was significantly more difficult to write than the C++/CORBA interface program.