

PROCEEDINGS OF SPIE

[SPIDigitalLibrary.org/conference-proceedings-of-spie](https://spiedigitallibrary.org/conference-proceedings-of-spie)

Remote observing with the Caltech millimeter-wave array

Stephen L. Scott

Stephen L. Scott, "Remote observing with the Caltech millimeter-wave array,"
Proc. SPIE 3351, Telescope Control Systems III, (26 May 1998); doi:
10.1117/12.308794

SPIE.

Event: Astronomical Telescopes and Instrumentation, 1998, Kona, HI, United States

Remote observing with the Caltech millimeter wave array

Stephen Scott

California Institute of Technology,
Owens Valley Radio Observatory, P.O. Box 968, Big Pine, Ca

ABSTRACT

Remote observing with Caltech's millimeter wave array at the Owens Valley Radio Observatory (OVRO) is being extended to use the graphical capabilities commonly available on computers today. To allow the instrument to be clearly presented to the user, a rich interface has been developed that combines the use of color highlights, graphical representation of data, and audio. Java and internet protocols are used to extend this interface across the Web to provide remote access. Compression techniques are used to enable use over low bandwidth links. This paper presents the design goals, implementation details, and current status of this effort with emphasis on the monitoring of the array.

Keywords: telescope control, user interface, remote observing, data compression, relational database, Java

1. INTRODUCTION

Modern astronomical instruments face the challenge of presenting complex and changing systems to observers, engineers and technicians. Caltech's six element millimeter wave aperture synthesis array at OVRO is in the process of implementing a new user interface to facilitate observing and troubleshooting of an increasingly complex instrument and to improve the remote observing capability. The requirements for this new interface originate in the details of how the instrument is operated.

Around the clock operation of the Caltech array is done by astronomers and students from a variety of institutions without the help of telescope operators. This style of operation benefits from a more accessible instrument than is typically required. Running an instrument efficiently without operators works well when there is easy and timely access to resources. These resources include expert observers, instrumentation designers and maintenance personnel. These people in turn need transparent access to the current state of the array to be able to provide assistance.

Remote observing can have many definitions. For the OVRO array, remote observing means there is no preferred location, or computer, for monitoring and control of the array. Trouble shooting the array requires control from the telescope receiver cabins and the central electronics area. Convenience in operating the array requires access from the control building and from the workstation near the dorm rooms and dining area. Even though the primary observer is usually on site at OVRO, there is a very active eavesdropping and coaching effort from Pasadena. During normal working hours, many engineers monitor the array from the lab to anticipate problems. To make efficient use of the array, engineering tests, program development and testing are frequently scheduled outside of normal working hours. It is convenient if this work can be carried out from home on a PC or Mac with an internet connection.

The monitoring portion of the remote observing problem for radio telescopes is in many ways more challenging than the control aspect, particularly if full diagnostic capability is required. Without auto-guiders or images to display, the data rates, screen layout and data presentation for monitoring present the major challenges. The approach to the new user interface for the array has been to concentrate on the monitoring aspects first and then follow with control. The current status is that the first set of the most commonly used monitoring screens are complete and operational and some limited control functions have been implemented. This paper covers this initial work, concentrating on the monitoring of the array and our plans for extending this to the control area. The user interface for the array may be directly accessed via the Web*.

Other author information - E-mail: scott@ovro.caltech.edu
*<http://www.ovro.caltech.edu/mm>

2. DESIGN GOALS

The primary design goal is to provide a rich graphical interface that makes it easy to control, monitor and trouble shoot the array. This includes the appropriate use of background color, time series graphical representation, time series file dumps and audio alarms. Screen space is important so information density must be high.

The secondary design goal is universal parallel access. It is desirable to be able to monitor and control the array from as many potential locations as possible. This implies access from different machine architectures and operating systems. Simultaneous monitoring from different locations is also necessary.

The third design goal is to limit resource use so that an implementation is possible with fairly common computer hardware. A potential user should be able to access the array with a modest PC over a modem. Specifically, a session with 6 windows, each containing 50 items of information updated every two seconds, should use less than a quarter of the cpu of a 150MHz Pentium and less than 25% of the bandwidth of a 28.8kbps modem. The centralized server is assumed to be of the UltraSparc class of machines and should be able to serve 50 windows using less than a quarter of the cpu. The estimates of the number of windows are intentionally large to make sure that the design is conservative.

3. IMPLEMENTATION

The interface to the user is presented as a set of independent windows. The windows that a user selects will depend on their interests and the amount of screen real estate available to them. The local window manager can be used to resize and place the windows to customize the layout. The windows are active and receiving data even when iconified.

A classic client/server architecture is used to separate the data source from the presentation of the data. The fundamental data source is a shared memory area containing all of the data points within the array. The client/server connection is a TCP/IP socket and thus suitable for locating the client arbitrarily on the internet. The details of the architecture are based on issues of performance and programming ease.

3.1. Java Clients

All of the client programs are written in pure Java to achieve a reasonable measure of platform independence and portability. A single Java client program dominates the suite of clients. This client is run to bring up any of the different realtime monitoring windows (RMW) that receive continuous realtime updates. Currently there are 16 different types of these windows and we anticipate having around 50 within the next year. Figures 1 and 2 show two different RMWs with the general launch menu in the upper right corner of Figure 1.

When the generic RMW client starts it connects to the master server program and requests to be served with a specific type of display. The full layout is then sent to the client and it configures itself accordingly. As the figures illustrate, the individual RMWs can have layouts that are significantly different even though all are running the same Java client. The common feature of an RMW is that it is made up of individual realtime data cells that are displayed as text and continuously updated. The rest of the RMW consists of labeling and layout information. By concentrating all the unique program content in the server and none in the client, a new RMW can be implemented by programming in only one place.

The other client programs are for specialized windows that require constructs not available in the RMW model. The client/server implementation is maintained to feed update information to the client, but in this case each client is a different Java program. The realtime observing log window falls into this category and is illustrated in Figure 3. The initial historical information for this window is gotten directly from the Sybase relational database for the array using the Java Database Connectivity (JDBC) calls. Subsequent updates are sent by the server program using data gotten from shared memory. It is the first of the specialized windows to be written but others are planned.

There are several tradeoffs that are made when running the clients as applets from a browser or as Java applications. The main function of the browser is to supply the hypertext link to run the program and of course the Java Virtual Machine (JVM) itself. Since all of the array clients run in their own window, the browser window itself is superfluous after starting the program. Running the client as an applet insures that the latest version of the code will be executed at the expense of download time for the class files over the net. Currently the client class files are about 100KB in size, doubling if database access is required. Neither of these sizes is an issue over a good connection but over a modem it will take 30 to 60 seconds for the download. Browsers have also been slow to incorporate the

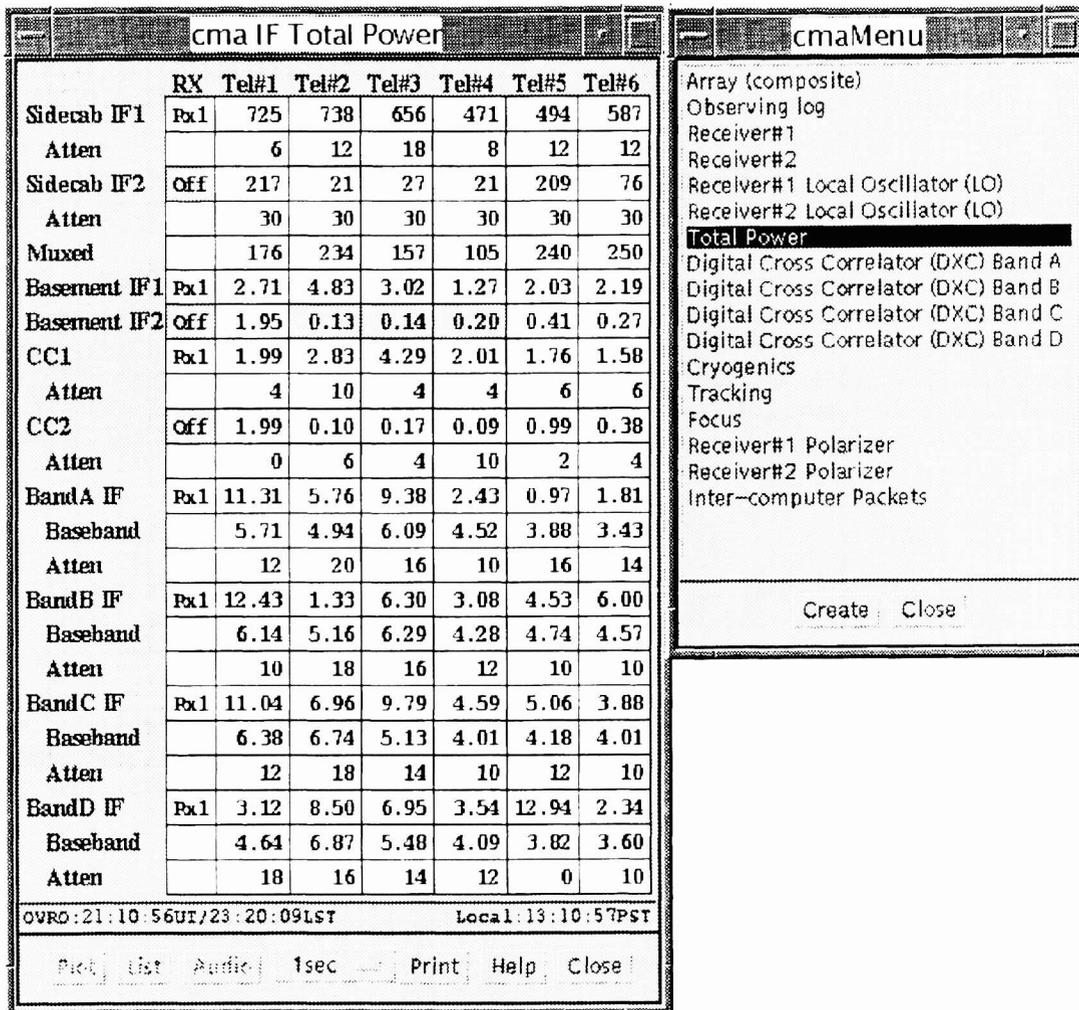


Figure 1. A realtime monitor windows using tabular layout to follow the total power through the signal path. The general window launch menu is on the right.

latest versions of Java so making use of the most recent features of Java in the client may break applets run through the browser. However the browser does provide a simple, universal method of finding programs and running them so the power of the model cannot be ignored. For security reasons, most browsers don't allow printing or writing files on the local disk. Newer versions of browsers allow you to customize the security policy so these restriction probably won't be an issue in the future.

The alternative is to run the client as an application. To do this, the client class files are downloaded to the local hard drive and then a local Java Virtual Machine (JVM) is invoked with the correct arguments to start the client. Sun distributes the Java Runtime Environment (JRE) for free for this purpose. The application approach requires more motivation and expertise on the part of the user but is probably the preferred option for frequent modem users. There are several tricks to make these steps easier including making a pre-packaged zip or tar file available for ftp or commercial solutions with windows install type packaging or push technology to keep class files up to date. To avoid evolutionary client/server incompatibilities with the distributed clients, a version checking scheme is implemented. When the client attempts to start, it checks the version number of the server with its version number and if a mismatch is detected an error message indicates that a new client should be downloaded.

All of the current Java clients are compatible with the Java Development Kit (JDK) version 1.1. Most browsers are now capable of running JDK1.1 programs so the current interface is available both through the browser and as an application. The Java source code for all the clients is about 4800 lines.

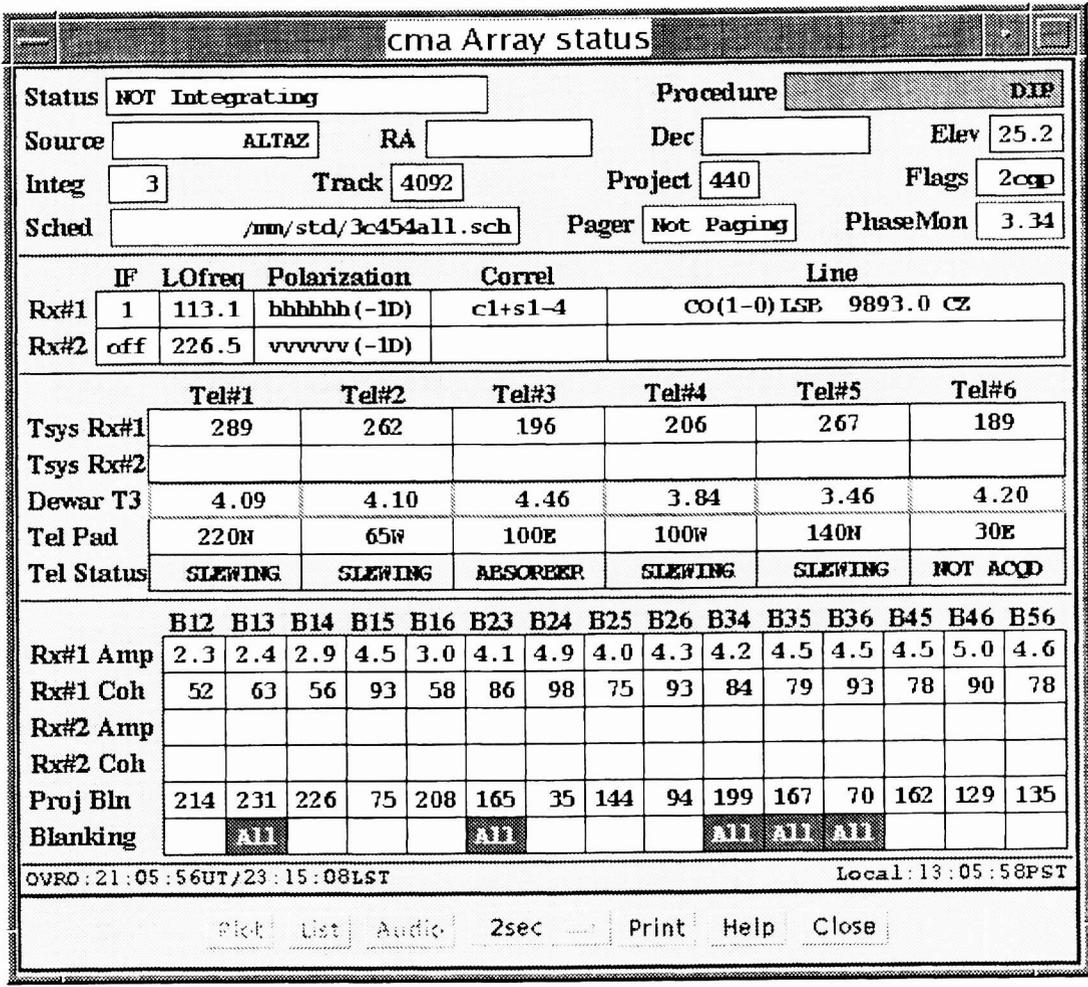


Figure 2. The general array status monitor window illustrates the flexibility of the layouts available.

3.2. C++ Servers

The server programs are written in C++ and run on the Unix machine that is the host for the array control. There is a one to one correspondence between a server program and a specific instance of a client display. This model of an individual server per client increases overhead but guarantees robustness. If a single server program goes down, only one display is affected, not all. There are several reasons C++ was chosen over Java. The first is efficiency of execution. The goal is to support many realtime windows simultaneously from a single server so cpu usage is an important consideration. Current JVM implementations are a factor of 3 to 10 times slower than C++. The second reason is trivial access to the shared memory which is used as the source for all of the realtime data. The shared memory is modeled as a C structure and so it just appears as another class under C++ with the member data available for direct access. The third is program size. A bare JVM with no program requires about 4 MB of memory while a typical C++ server with program ranges from 0.9 MB to 1.3 MB. The Java server solution would probably use over 4 times the amount of memory.

The individual server programs define the type of windows available. By building up a useful set of base classes, a new window can be defined quickly and compactly. A typical server program using the existing base classes takes only 1 or 2 pages of code. A more complex server that creates new data types by extending the base classes might take 5 pages of source code. The source code for the base classes is about 1500 lines.

```

cma Observing Log
-----
waitime Find lines Stop
Results Realtime updates active...
20Feb98/20:46:43 RADIO_OFFSET3 AZ=0.118 EL=-0.142
20Feb98/20:46:44 RADIO_OFFSET4 AZ=0.169 EL=-0.238
20Feb98/20:46:44 RADIO_OFFSET5 AZ=0.036 EL=-0.229
20Feb98/20:46:44 RADIO_OFFSET6 AZ=0.194 EL=-0.230
20Feb98/20:46:44 -SCH: INT INTEG_MIN=5 REPS=3 LOW_EL=SKIP
20Feb98/20:50:05 ACQLIM LIMIT=14
20Feb98/20:52:14 ----- Integration 1 completed; rep 1 out of 3
20Feb98/20:57:33 ----- Integration 2 completed; rep 2 out of 3
20Feb98/21:02:52 ----- Integration 3 completed; rep 3 out of 3
20Feb98/21:02:52 -SCH: DIP
20Feb98/21:08:04 DIP1/rx1: Tau0=0.184 +/- 7% Tsys0= 173
20Feb98/21:08:04 DIP2/rx1: Tau0=0.125 +/-27% Tsys0= 220 **Hots unstable( 3%)**
20Feb98/21:08:04 DIP3/rx1: Tau0=0.207 +/- 1% Tsys0= 92
20Feb98/21:08:04 DIP4/rx1: Tau0=0.273 +/-25% Tsys0= 201 **Hots unstable( 4%)**
20Feb98/21:08:04 DIP5/rx1: Tau0=0.205 +/- 3% Tsys0= 134
20Feb98/21:08:04 DIP6/rx1: Tau0=0.197 +/- 5% Tsys0= 104 **Hots unstable( 2%)**
20Feb98/21:08:06 DIP complete: Tau0=0.205 @113.08GHz
20Feb98/21:08:06 -SCH: FLAGS P
20Feb98/21:08:06 Flags: passband Tuning:v01
20Feb98/21:08:06 -SCH: INT/NOISE 0.5
20Feb98/21:08:51 ----- Integration 4 completed
20Feb98/21:08:51 -Schedule done, reps expended: /mm/std/3c454all.sch
20Feb98/21:08:52 SCH: SCHEDULE/FILE/LST/NOWAIT/RUN FILE= /mm/std/uranus.sch START_TIME=2
20Feb98/21:08:52 -SCH: OBS URANUS LOW_EL=SKIP
20Feb98/21:08:53 -SCH: FLAGS 1CP T=1
20Feb98/21:08:53 Flags: primary std. coherence, passband Tuning:v01
20Feb98/21:08:53 -SCH: POINT/CHANGE/P3 INT=10 CYCLES=100 REPS=1 ACCURACY=0.03 TIME_LIM
20Feb98/21:08:53 Low visibility (<0.20) baselines not used: B13 B16 B34 B35

```

Figure 3. The observing log window is a specialized window that directly accesses the Sybase relational database.

3.3. Shared Memory

An efficient programming environment for the C++ servers is based on the shared memory containing all of the data of interest about the array. The primary source of data for the shared memory is eight different realtime microcomputers embedded in the array hardware that send UDP datagrams twice a second. This data is received and put directly into the shared memory. The server programs then simply map the shared memory into their address space to have access to the live data.

3.4. Communication Protocol

Most of the cell data are integers or doubles but some are straight text. When an update is sent by the server, all of the numerical data transformed to ascii text. This relieves the less efficient Java client from having to format the output. It also allows numerical data to be replaced with text strings. For example, if the shared memory is not being updated, the cells are filled with question marks rather than the stale erroneous data. If a numerical value overflows its field width, it is replace by a string of asterisks. These transformations could be done in the client by sending more status information but then the amount of transmitted data is increased. The background color of each cell is sent as a single character code following the cell text data.

To improve throughput on the data update, the data stream is compressed at the server by transmitting only changes to the existing screen and escape codes that encode the number of unchanged characters to skip. This algorithm matches the data quite well, where often the screen appearance shows just the twinkling of the least significant digits. This is another reason that ascii encoding is used. Typical compression rates range from 5 to 10. This compression step is essential for displaying multiple windows over a modem.

3.5. Realtime Monitoring Window Life Cycle

To fully illustrate the implementation it is useful to examine the life cycle of a single RMW. Life begins when a Java RMW client is initiated and passed the name of a window type as an argument. At this time there is no graphical representation of the client. The client then opens a TCP/IP socket connection to a pre-assigned port on the array control computer with a request for a specific type of realtime window. The Unix system then runs a simple "internet service" program attached to this port that execs a copy of the specific server program for the type of window requested. This establishes a one to one mapping between the client and a window specific server program. The client now requests its layout from the server, and after reacting to the reply it makes itself visible. It then begins a loop of requesting an update of the live data from the server, displaying the data, and then sleeping for the update interval. The actual sleep time is manipulated based on the short term average of the actual update rate to remove any systematic latency. This tuning is particularly useful over a modem line and keeps the actual update rate close to the requested rate even at one second updates. The multi-threaded nature of the Java environment allows reaction to user initiated events, such as initiating a plot, while in this loop. Driving the update cycle from the client is appropriate for monitoring data as it is then very robust to unforeseen delays. These delays only cause a gradual degradation of response rather than total failure.

4. REALTIME DATA CELLS

The realtime data cell is the fundamental building block of the realtime monitoring windows. It is the inherent capabilities of these cells that make it easy to build RMWs with a high level of functionality. A RMW and all its children are shown in Figure 4. The rectangles containing text on a light background are instances of the cells. The values within the cells are updated on a user selectable time scale that can be set from half a second to thirty minutes using the choice menu at the bottom of the window. A realtime cell can optionally have the cell background color change as a function of the value in the cell. Typical color assignments are yellow for warning and red for alert. An audio chime can also be associated with the cell by the user to call attention to the alert. When the window is initiated, each cell begins to cache a history that can contain the most recent 200 samples. This history is then available for display by other parts of the program. The cells are also used as part of a compact menu for selecting the advanced features. When a cell is clicked with the mouse, its border is enhanced and it becomes the "selected" cell. In Figure 4 the "Tot Power" for telescope #4 is the selected cell. Clicking on the plot or list button at the bottom of the window brings up the plot or list window for that cell.

The plot and list features coupled with the history are an important part of the user interface for the cells and are shown in the figure. When a plot or list window is initiated it is seeded with the values in the cell history. This allows the user to see an event occur in the main window and then to initiate a plot to look back in time before the event. Both plot and list contain print options. Additionally, the contents of the list window can be written to disk in a two column ascii format. The plotting is intentionally simple. The ordinate is auto scaled but the abscissa simply uses one pixel per sample to allow a maximal number of samples to be plotted and avoid unnecessary rescaling. The ability of the list window to write to disk allows more complex analysis using a spread sheet or other tools. Because the histories for all of the cells within a window are synchronized, plots and listings are time aligned for cross comparison of cells. Each plot or listing contains an extension of the history buffer that can contain 1000 points.

Control functionality can also be added to cells. Using the same cell selection metaphor and the control button at the bottom of the window, a control widget can be launched. Cells with control capability are indicated by a purple outline rather than the standard black. An example of a control widget can be seen in the upper right hand corner of Figure 4. Security for control features is currently based on the internet address of the client but will be changed to a login based mechanism using our Unix password files. If control is not authorized then the control button does not appear on the window.

Many of the cells are arranged in tabular form to provide a dense display and to emphasize the relationship of the data. It is also possible to lay out the cells in a more free format where there is a label associated with each cell. These two styles can then be stacked vertically to provide a window with multiple sections. Figures 1 and 2 illustrate some of the possibilities.

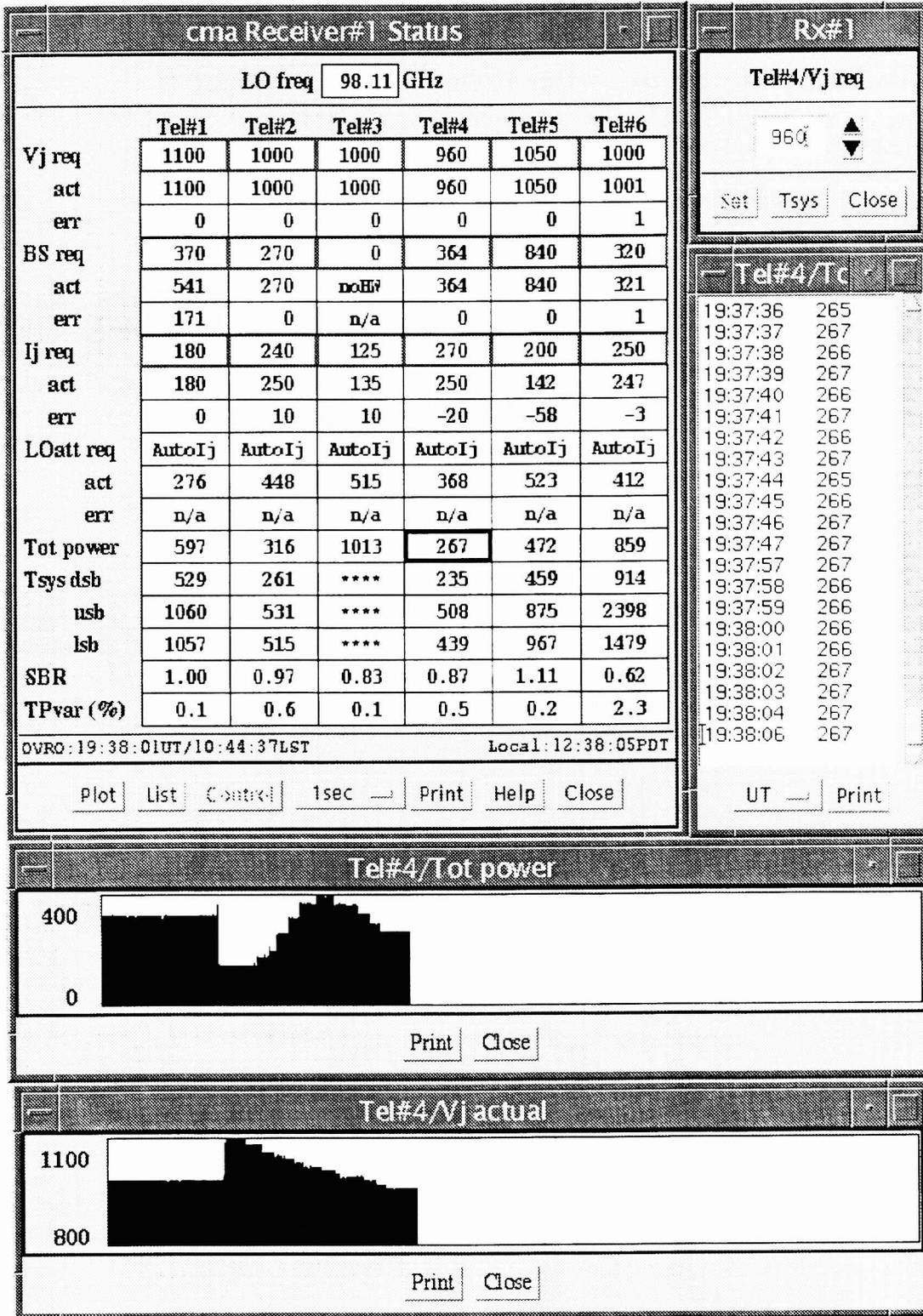


Figure 4. A realtime monitor windows surrounded by all of its children. The window in the upper right is a control widget.

5. RESOURCE USE

The estimates given below on resource use assume a realtime monitoring window with 90 to 100 cells and a two second update rate.

5.1. Client

The basic JVM with the graphics and all client code loaded uses about 8.5 MB of memory. Each instance of a window uses approximately an additional 1.6 MB by the time the cache is full. Usually all of the windows on a given computer are launched from the interactive menu and run under the same JVM. This makes the total memory use for 6 window (a very full screen) run about 18 MB which is acceptable on a 64 MB machine that is the norm these days. CPU usage per window on a Pentium 150 or a Sun UltraSparc 1/140 runs 3% to 5% giving a total of 20% to 30% for a 6 window load. At least a factor of 2 speedup is expected as the Java compilers mature and "just in time" native compiler technology is refined.

5.2. Server

Each server program uses about 0.9 MB of memory if linked to a dynamic shared library and about 1.3 MB when linked statically. Processor usage per server is about 0.3% of the cpu of an UltraSparc 1/170. The cpu requirements are minimal so hosting 50 windows should have little impact as long as the machine has at least 128 MB of memory. This should not be difficult at today's memory prices.

5.3. Bandwidth

A single window requires about 0.08 kilobytes per second (KBPS) after compression for update traffic. Six windows would require about 0.5 KBPS which is about 20% of the 2.5 KBPS that is typically achieved with a 28.8 modem. Again, the compression factor of 5 to 10 that is typical is important in maximizing the throughput.

6. PORTABILITY

Because the implementation concentrates all knowledge of screen layout and data in the server, the system has proved to be very portable. The Java client can run unmodified for any new realtime display that is generated, only requiring that the new server address and port number be passed in on the command line. The main requirement for a port to a new instrument is the existence of a shared memory area containing live data from the instrument on a Unix machine that is to run the server programs. If a shared memory exists, then the cell generation classes can be used without modification and realtime display generation is very easy. The creation of a new realtime window involves writing a simple one or two page C++ program that maps in the shared memory and then sets up labels and realtime data cells using locations in the shared memory. The software was installed at the Sub-Millimeter Array (under development by the Smithsonian Astrophysical Observatory) and a simple window created in about 4 hours. Other methods of retrieving the realtime data besides shared memory can be used but some coding would have to be done to build up classes using these methods.

7. FUTURE PLANS

The realtime windows that have been implemented include all of the ones most commonly needed to run the array. The next wave of realtime windows will include some convenience windows, such as weather, a realtime data plot, and a talk window. It will also include and the next layer of diagnostic windows. These diagnostic windows will include a tree structured fault analysis as well as display all of the monitoring points in the array.

On the control front, the next step is to implement a proper login for the clients based on the Unix password files. It will be possible to "browse" the array without login but to have access to control features a login will be required. The authentication mechanism will be implemented on the Unix host so it is also possible to do host or domain based filtering as well.

Further control plans are to provide graphical control of some of the more commonly changed parameters that are used to optimize performance of the instrument. A specialized widget for automatic receiver optimization is also needed. The present command line interface will be redone into a Java client to give full functionality in the new style. A replacement of the command line control interface with a graphical interface is a future goal.

The actual arrangement and selection of windows can take some time so it would be desirable to save the size and position of all of the array windows so they could all be invoked at once. The design for this is to have a single pane that contains all of the array windows and their state. This pane can then be manipulated, for example iconified, as a whole. The state of the pane and its contents could be saved to a file and restored as desired.

8. LESSONS LEARNED

The model of parallel, universal access to an instrument is very powerful. It enables collaborative use of the array, allows teaching via eavesdropping, and helps with troubleshooting and maintenance. Because the array is easily accessible and visible, it empowers the users, builders and maintainers and they become much more interested in the instrument.

The use of the cells in the monitoring windows as part of the menu for selecting advanced functions saves screen space and complexity. Having plot capability instantly available for arbitrary data items is very useful. Hard copy capability is essential.

Java is a new programming language and much has been learned about it. It is a well structured, well thought out language that provides a very pleasant programming environment. One of the most powerful aspects of Java are the built-in classes and functions that come with the language. These can be used instead of building many things from scratch. On the other hand, the widget set available for creating graphical applications, at least at the JDK1.1 version, is somewhat limited. The language is still evolving quite rapidly so version matching of code and JVM is an important issue. The newer features take a while to migrate into the browsers and some of the less common JVMs (like MacOS and OS-2), with a 6 to 12 month delay not uncommon. The cross platform graphical features are very convenient but not without inconsistencies so graphical debugging must be done on all target platforms. Large programs can take a long time to download at modem speeds so as Java programs become large, the applet model becomes less attractive. Performance is an issue with Java so it is not the language of choice where raw speed is essential. However, for many applications this is not the case and Java is a very good choice.

ACKNOWLEDGMENTS

The author thanks Ray Finch and Hemant Shukla for development and support of the shared memory areas and for program installation. This work is supported in part by NSF Grant AST 93-14079.