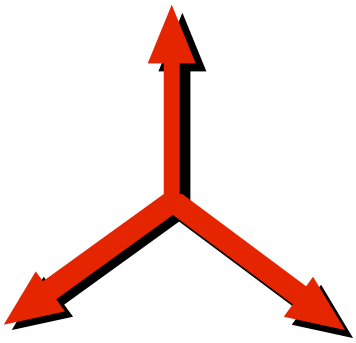
EXPANDED VERY LARGE ARRAY



A Loosely Coupled, Distributed Processor Antenna Monitor & Control Subsystem

Kevin Ryan

July 10, 2003

NATIONAL RADIO ASTRONOMY OBSERVATORY
P.O. Box 0, Socorro, New Mexico 87801

Operated by Associated Universities, Inc.
Under Contract with the National Science Foundation

Abstract:

Most distributed processing systems today rely on some form of remote procedure call (RPC) where one processor calls functions (or methods) on another processor over the network.

It is the author's belief that a system based on RPC's cannot be truly modular and, in the case of the EVLA, has real-time determinacy issues.

The purpose of this paper is to present the design of a distributed system as applied to the EVLA Antenna Monitor and Control System (AMCS) that does not implement RPC's.

1 The EVLA Antenna Control System

1.1 The AMCS must be Highly Modular

The EVLA AMCS is a real-time system of many processors distributed over a non-real-time network. Unlike most radio-telescope arrays, the EVLA must be designed to accommodate changes as new and different types of antennas are added during its ongoing future development.

The ability to operate a real-time system distributed over a non-real-time network and the ability to accommodate different hardware and future additions necessitates a modular system consisting of highly independent modules. Independent modules are characterized by the fact that they can:

1. Perform their jobs without depending on control from other modules or on the network, and
2. Be modified without causing changes to other modules in the system.

Two factors contribute to the 'independent-ness' of a module:

1. Cohesion – its 'single-mindedness', its focus on its own 'job',
2. Coupling – a measure of its interaction with other modules in the system.

A module is more independent if it has high (tight) cohesion and low (loose) coupling.

1.2 RPC Based Systems are not as Modular as Possible

1.2.1 Strict Interfaces Mean Tighter Coupling

Many distributed systems today rely on a means where a local process controls a remote process by calling functions or methods contained in the remote process. This is sometimes referred to as Remote Procedure Calls (RPC), Remote Method Invocation (RMI), object brokering, etc. They require that the local machine maintain a well-defined interface to the remote functions to be called. In the case of Object Oriented systems, the local process maintains a 'copy' of the remote object and operates on it as if it were a local object.

If a change is made to the remote module that causes a change to the interface, than users of that interface must also change. This implies that the system is not as modular as it could be; the reason being that the coupling between the modules is tighter than it ought to be. This extra coupling is a result of the formal interface.

1.2.2 Inter-Module Operation Means Looser Cohesion

If a module relies on another module to perform its task, then that module is not wholly independent. In non-real-time systems or distributed systems with real-time

communications between modules, this does not present as large a problem as is the case in the AMCS. The AMCS uses standard Ethernet for its network; Ethernet and the associated communications protocols are not time-deterministic. This means that the controlling module cannot be certain that a remote function will execute 'on time'. This has further disadvantages in that it causes the need for the system to be designed so that remote function calls are asynchronous to the local process and, in turn, adds complexity.

1.3 A Different Approach

RPC implies that *verbs* are being communicated between modules. The calling module actually invokes the functions of the remote process, such as calling 'setPosition' for a servo module.

A different approach would be to communicate *nouns* between modules. In other words, simply tell the remote module what is wanted and let him invoke the necessary methods himself.

Now the calling module no longer has to know about the remote functions needed to configure the remote module; the interface between the modules can be relaxed and the coupling loosened.

If a 'noun-based' communications method can be established between the modules of the system, then they can utilize a single simple interface – that of text passing.

1.4 A Simple Observation Example

An observe script should not attempt to control the antennas in the system; control is the job of the AMCS. The observe script should simply tell the AMCS what is to be done and when to do it. The observe script might simply be XML text of the following (overly-simplified) format. The example is for an observation consisting of two scans; one at 15:08:09 the other at 15:14:27 VLA LST on L-Band.

The following 'script' is received from e2e:

```
<observation>
  <configuration
     time       = "2002y09m05d15:08:09:LST"
     subarray   = "1, 3, 7, 9, 13-23"
     ra         = "14:00:28.6526"
     dec        = "+62:10:38.526"
     band       = "L"
  />
  <configuration
     time       = "2002y09m05d15:14:27:LST"
     subarray   = "1, 3, 7, 9, 13-23"
     ra         = "15:49:17.4686"
     dec        = "+50:38:05.788"
     band       = "L"
  />
</observation>
```

This would tell the system that antennas 1, 3, 7, 9, 13-23 are to be used to observe the specified source on L-Band starting at 15:08:09 and continuing until it is time to start moving so as to be in position for the next configuration at 15:14:27.

The Subarray Device will parse the script and refine the information needed to tell its antenna Devices what to do and then re-issue the refined XML to each of its antennas:

```
<observation>
  <configuration
     time       = "2002y09m05d15:08:09:LST"
     ra         = "14:00:28.6526"
     dec        = "+62:10:38.526"
     band       = "L"
  />
  <configuration
     time       = "2002y09m05d15:14:27:LST"
     ra         = "15:49:17.4686"
     dec        = "+50:38:05.788"
```

```
     band       = "L"
  />
</observation>
```

In this case, the <subarray> information is removed since each Antenna requires antenna-specific instructions only. Each antenna converts RA and DEC to AZ and EL and sends the following to its servos:

To AzimuthServo:

```
<configuration
     time       = "2002y09m05d15:08:09:LST"
     position   = “344.270”
/>
<configuration
     time       = "2002y09m05d15:14:27:LST"
     position   = “18.382”
/>
```

To ElevationServo:

```
<configuration
     time       = "2002y09m05d15:08:09:LST"
     position   = “59.929”
/>
<configuration
     time       = "2002y09m05d15:14:27:LST"
     position   = “72.266”
/>
```

Notice that position is no longer AZ/EL or RA/DEC Position, it is simply ‘position’; this is because a servo only knows about its own position, the fact that it is the AZ or EL servo is irrelevant to the servo itself.

The Antenna Device also knows about its own switches and which ones must be positioned to receive on L-Band and creates the following further refinement that it sends to the appropriate Switch Devices:

To XBandConverterSwitch:

```
<configuration
     time       = "2002y09m05d15:08:09:LST"
     position   = “L”
/>
<configuration
     time       = "2002y09m05d15:14:27:LST"
```

```
     position   = “L”
/>
```

To XBandIFAmplifierSwitch:

```
<configuration
     time       = "2002y09m05d15:08:09:LST"
     position   = “X”
/>
<configuration
     time       = "2002y09m05d15:14:27:LST"
     position   = “X”
/>
```

To FirstLOSwitch:

```
<configuration
     time       = "2002y09m05d15:08:09:LST"
     position   = “LCS”
/>
<configuration
     time       = "2002y09m05d15:14:27:LST"
     position   = “LCS”
/>
```

At the specified times, the various components call their own (local) methods to configure themselves to the specified states. Real-time operations do not occur over the network, e2e does not have to know that the antenna he is using has a FirstLOSwitch and the internal functions and methods needed to implement the configuration changes are kept encapsulated within their respective modules.

1.5 System Operation is Orthogonal

A CPU is said to have an orthogonal instruction set if each of the instructions in the set performs it operation on all data-types in all addressing modes. The AMCS is designed to operate orthogonally as a result of early requirements that each module be independently operable. This means that a script from e2e could be designed and sent to operate *any* module in the system not just the highest (sub)array level. Likewise, a technician or operator could use a text-passing user interface to operate any Device, from the whole array to a single switch in a similar manner. It also means that a switch can be operated without being connected to its antenna as would be the case if were being serviced on the test-bench.

All devices in the AMCS will have the ability to be operated both from their parent device and directly from a local interface.

2 Implementation of a Text-Based Interface

The EVLA AMCS consists of software *Devices* and *DeviceServers*. Devices represent components of the physical system. A block diagram an EVLA Antenna might have a one-to-one correspondence between each block in the diagram and software a Device. DeviceServers, on the other hand would not be present in the block diagram of the system.

2.1 DeviceServers Provide the Information Path

DeviceServers have the following characteristics:

- Exactly one DeviceServer per processor.
- Serves zero to many Devices.
- C*ontains* the component Device objects that it serves so it may manipulate them directly without having to 'broker' them as would be case over a network.
- Provides a single network interface shared by all of its Devices.
- Provides an operationally transparent interface to all onboard Devices. The DeviceServer must not be 'seen' by the rest of the system in terms of operation. A block diagram of an antenna depicts the subcomponents of the antenna; things such as servos, synthesizers, switches, etc. The operational software must see these same items in the same way; this means that it should not view them hanging from a server object. The DevicesServer must present the Devices to the system in a way that the system 'thinks' it is communicating with the Device directly

In the CMP the DeviceServer is called the CMPServer; for the MIB, it is sometimes referred to as the MIBObject but should more concisely be named the MIBServer.

The DeviceServer provides the network communications mechanism. It receives the text-based command data and passes it on to the specified Device. It must do this however so that the client application addresses the target Device itself and not the Device's DeviceServer.

2.2 Devices Use the Information

The specific Devices (i.e. Elevation Servos) provide the real functionality of the AMCS. Each specific Device will be responsible for the overall 'modularness' of the system. It must provide the functionality needed to make the represented physical device a stand-alone entity to the point that it can operate independently of the rest of the system. This is natural when one considers that a servo does not have to be in a radio-telescope in order to be a servo.

This functionality includes self-monitor and control. AMCS Devices will perform 'checker' services on themselves. This is what distributed-processing is about. Some advantages of this are:

- No other system component knows more about a device than the device himself.
- A central checker system will be a cause for undue amounts of raw data flowing across module boundaries. Self-checking will only send high-level (and meaningful) messages when appropriate.
- A Device will be able to accept commands of a generic nature and know exactly how to break it down into the format needed by its own specific control-points..

To implement the text-based interface, a Device must be able to interpret the data by mapping textual keyword/value pairs to local methods. This will be done with an XML parser. In the case of higher level Devices such as Antennas and (Sub)Arrays, they must also be able to form new textual keyword/value pairs to send to their children devices.

2.3 Information Flow

The following code output shows how information would flow in the system for the example script given in section 1.

```
[Mercury:evla/amcs/test] kevdev% java TestServer
DeviceInterfaceParser: [{configuration={subarray=1, 3, 7, 9, 13-
23, time=2002y09m05d15:08:09:LST, band=L, ra=14:00:28.6526,
dec=+62:10:38.526}}, {configuration={subarray=1, 3, 7, 9, 13-23,
time=2002y09m05d15:14:27:LST, band=L, ra=15:49:17.4686,
dec=+50:38:05.788}}]
TestAntenna : configureBand : property  = L
DeviceInterfaceParser: [{configuration={position=X,
time=2002y09m05d15:08:09:LST}}]
XBandConverterSwitch    : Queuing Switch Position = 'X' For time
= 2002y09m05d15:08:09:LST
DeviceInterfaceParser: [{configuration={position=L,
time=2002y09m05d15:08:09:LST}}]
XBandIFAmplifierSwitch  : Queuing Switch Position = 'L' For time
= 2002y09m05d15:08:09:LST
DeviceInterfaceParser: [{configuration={position=LCS,
time=2002y09m05d15:08:09:LST}}]
FirstLOSwitch           : Queuing Switch Position = 'LCS' For
time : 2002y09m05d15:08:09:LST
TestAntenna : configureRa : property  = 14:00:28.6526
DeviceInterfaceParser: [{configuration={position=344.270,
time=2002y09m05d15:08:09:LST}}]
AzimuthServo            : Queuing Servo Position = '344.270' For
time = 2002y09m05d15:08:09:LST
TestAntenna : configureDec : property  = +62:10:38.526
```

```
DeviceInterfaceParser: [{configuration={position=72.266,
time=2002y09m05d15:08:09:LST}}]
ElevationServo          : Queuing Servo Position = '72.266' For
time = 2002y09m05d15:08:09:LST
TestAntenna:administerConfiguration : WARNING: Unrecognized
Configuration Properties : {subarray=1, 3, 7, 9, 13-23}
TestAntenna : configureBand : property  = L
DeviceInterfaceParser: [{configuration={position=X,
time=2002y09m05d15:14:27:LST}}]
XBandConverterSwitch    : Queuing Switch Position = 'X' For time
= 2002y09m05d15:14:27:LST
DeviceInterfaceParser: [{configuration={position=L,
time=2002y09m05d15:14:27:LST}}]
XBandIFAmplifierSwitch  : Queuing Switch Position = 'L' For time
= 2002y09m05d15:14:27:LST
DeviceInterfaceParser: [{configuration={position=LCS,
time=2002y09m05d15:14:27:LST}}]
FirstLOSwitch           : Queuing Switch Position = 'LCS' For
time : 2002y09m05d15:14:27:LST
TestAntenna : configureRa : property  = 15:49:17.4686
DeviceInterfaceParser: [{configuration={position=18.382,
time=2002y09m05d15:14:27:LST}}]
AzimuthServo            : Queuing Servo Position = '18.382' For
time = 2002y09m05d15:14:27:LST
TestAntenna : configureDec : property  = +50:38:05.788
DeviceInterfaceParser: [{configuration={position=72.266,
time=2002y09m05d15:14:27:LST}}]
ElevationServo          : Queuing Servo Position = '72.266' For
time = 2002y09m05d15:14:27:LST
TestAntenna:administerConfiguration : WARNING: Unrecognized
Configuration Properties : {subarray=1, 3, 7, 9, 13-23}
[Mercury:evla/amcs/test] kevdev%
```

3 Benefits of XML Text-Based Interfacing

3.1 Inter-Module

Perhaps the biggest advantage that the XML model affords is that inter-module communications are greatly simplified. Modules would simply have to know how to pass strings.

Just as a parent device tells its children what to do, so too can a human via a very simple GUI or even command shell if all that is needed to operate the Devices are text-based command/value pairs.

A technician could telnet into a servo on his workbench and type: “<position><123.45>”.

String based protocols are being used extensively in industry. They make extremely 'thin' clients such as web browsers possible. Because of this, it is possible that any device supporting a web browser might be able to fully operate an AMCS Device.

3.2 Intra-Module

If the Device understands XML for communications across the network, it can use this same understanding for operations within itself.

3.2.1 Device Initialization

XML can be used to define the state or mode that the Device should configure itself too on initialization. This XML block would be a part of the same descriptor file that defines a Device's monitor and control points and other parameters.

Since it is text based and file-resident the contents of the descriptor file can be changed and the changes made effective without code recompilation.

3.2.2 During Operation

XML provides a persistent, ready-made (it was already used for the communications interface) 'list' of the states that a Device is or will be in. This list can be used by the Device as a queue of 'what to do, when' and by their parent Devices (or even human users) for validation purposes.