JMaCS: a Java monitoring and control system

Rob Dickens[*]
Latterfrosken Software Development Limited,
32 Bradford St, Walsall, West Midlands, UK, WS1 3QA

ABSTRACT

JMaCS is software intended to facilitate, in soft realtime, the local or remote interactive and programmatic monitoring and control of some distributed target, such as an astronomical telescope or telescope network. It is derived from experimental software written for a radar used for observing the Earth's ionosphere, and aims to bring to bear the remote polymorphism afforded by Java RMI (Remote Method Invocation). The core software does not provide all these facilities itself, but only a standard way to plug device interfaces into a third-party JMaCS implementation. It is presented here, together with the JMaCS implementation developed in parallel and a demonstration target, as a proof of concept.

Keywords: software, monitor, control, system, distributed, remote, GUI, Java, RMI, JMaCS

1. INTRODUCTION

This paper presents the JMaCS[1] monitoring and control software. It begins by explaining the motivation for the software's development, including the choice of platform, before going on to describe the software's design, including the assumptions upon which this was based. The implementation of the design is then discussed, where it is revealed what problems were encountered in practice, and how these were addressed. A demonstration target is then described, and the results of some load testing given. Finally, some conclusions are drawn from what has been demonstrated, and also from the experience of developing the software.

2. MOTIVATION

2.1 Software requirements

JMaCS is derived[2] from experimental software[1][2] originally written[3] to permit the monitoring and control of a radar used for observing the Earth's ionosphere[4][3]. Of particular interest when considering that software's requirements were the radar's (then novel) distributed design[4], together with its remote and sometimes inaccessible location. The scope of those requirements were subsequently to grow from being user interface-related only, to system-wide.

Although JMaCS is no longer radar-specific, its design goals are otherwise essentially the same: to facilitate the local or remote interactive and programmatic monitoring and control of a distributed target. This should be achieved in soft[5] realtime, which is to say that commands and status should be delivered in a timely manner, but with the built-in assumption that delays will sometimes occur. Where necessary, therefore, external underlying hard-realtime subsystems will be assumed.

2.2 Development platform

The Java[6] platform was chosen for its many features aimed at facilitating the development of distributed software. Of particular interest initially were those features which simplified the development of cross-platform

*rob.dickens@lafros.com +441922/01922 644 223

1 http://jmacs.org
2 with permission
3 by the author
4 http://www.eiscat.se:8080/ESR
5 for a definition, see for example [5]

GUI[6] apps. Later, it was to be the ability to send and receive executable objects, using efficient native[7] Java serialisation, and employ polymorphism[8] in a distributed setting.

3. DESIGN

3.1 Client-server design

It was decided that a given target should consist of a server together with some number of device-interface (DI) and user-interface (UI) clients. This was considered to be a suitable reflection of the distributed and possibly remote nature of the target, and also to have two advantages over a design involving DI servers: access to devices is more secure, since they may only be accessed via the target server chosen by the DI deployer; it is more easily extended, by the simple introduction of additional DI clients.

3.2 High-level vs low-level interactions

Assuming that the high-level network interactions (between client and server) afforded by Java RMI[9][7][8][9] would have the advantage of being more convenient, and low-level ones (using network sockets directly) would offer better performance, it was decided that the former should be used for submitting commands, and the latter for the dissemination of device status. Furthermore, in order to permit efficient dissemination on a local network, it was decided that the size[10] of all device status samples should be constrained by the payload limit[11] of a UDP[12] datagram (that could then be multicast to all recipients in one go).

3.3 DI plug-in supplies driver, optional GUIs

To have JMaCS talk to your hardware (or software), the idea is to supply a 'plug-in' for it, consisting of classes implementing particular Java interfaces. Thus, a driver class is required (for executing commands and generating status), together with optional classes for any monitor window and any control panel, plus any constants.

3.4 DIs register domain.like.names, resulting in logical hierarchy

Your plug-in must be supplied together with a domain.like.name, thereby defining the DI's position in a logical hierarchy representing the target as a whole. Users may then interactively navigate to a particular DI using a tree widget, and DIs have a way to address each other. Note that the hierarchy is only a logical one—the design requires that all interaction with a given DI (except for the dissemination of device status samples by multicasting) takes place via the target's server.

3.5 DI plug-in adapter supports 'Programmable Devices' (PDs)

Rather than have to write a new DI plug-in every time, it turns out to be useful to write 'adapter'[10] ones, which fill-in some or all of the 'blanks', and often introduce new ones of their own. Related DI plug-ins may then be created (with much less effort) by re-using the adapter, and supplying only that which is unique to each DI.

Using this technique, it is possible to write an adapter which accepts a reference to a definition of an actual named device (e.g. steerable antenna) having its own API[13], together with any components which are implementation-specific. Furthermore, the adapter can be endowed with the means to receive and run programs, exposing the device's API to those programs.

Such an adapter is called for by the design, which therefore refers to the definitions the adapter accepts as 'Programmable Devices' (PDs).

New PDs may be written in terms of existing ones using the object-oriented techniques of composition and inheritance[11], and catalogues then accumulated.

6 Graphical User Interface
7 as opposed to using XML (eXtensible Markup Language)
8 here, object-oriented programming technique whereby instances of different subclasses may be supplied where an instance of the superclass is specified, thereby enabling different behaviour in each case
9 http://java.sun.com/javase/technologies/core/basic/rmi
10 following (native Java) serialisation
11 a little under 64 KiB
12 User Datagram Protocol
13 Application Programming Interface

3.6 Device status sampling configurable dynamically

So as to produce regular snapshots of the overall target, all DIs should sample the status of their respective devices at the same time and at regular intervals.

Given that they may be added or removed dynamically, and that those which interface to PDs may run different programs, the design calls for the parameters which control when this sampling occurs, namely the reference boundary and period, to be configurable dynamically.

3.7 Server maintains per-client control-privilege flag

Each command originating from a given client will be tagged with that client's control-privilege flag, before being relayed to the destination DI. It will then be up to that DI to decide whether or not to execute it, depending on the value of the flag, and on whether or not the command makes any changes (as opposed to being just a query).

4. IMPLEMENTATION

4.1 Layering of the software

It soon became apparent that DI plug-ins needed to be insulated from most of the changes being made to the monitoring and control system itself. This was addressed by partitioning the latter, resulting in a JMaCS API middle layer (org.jmacs

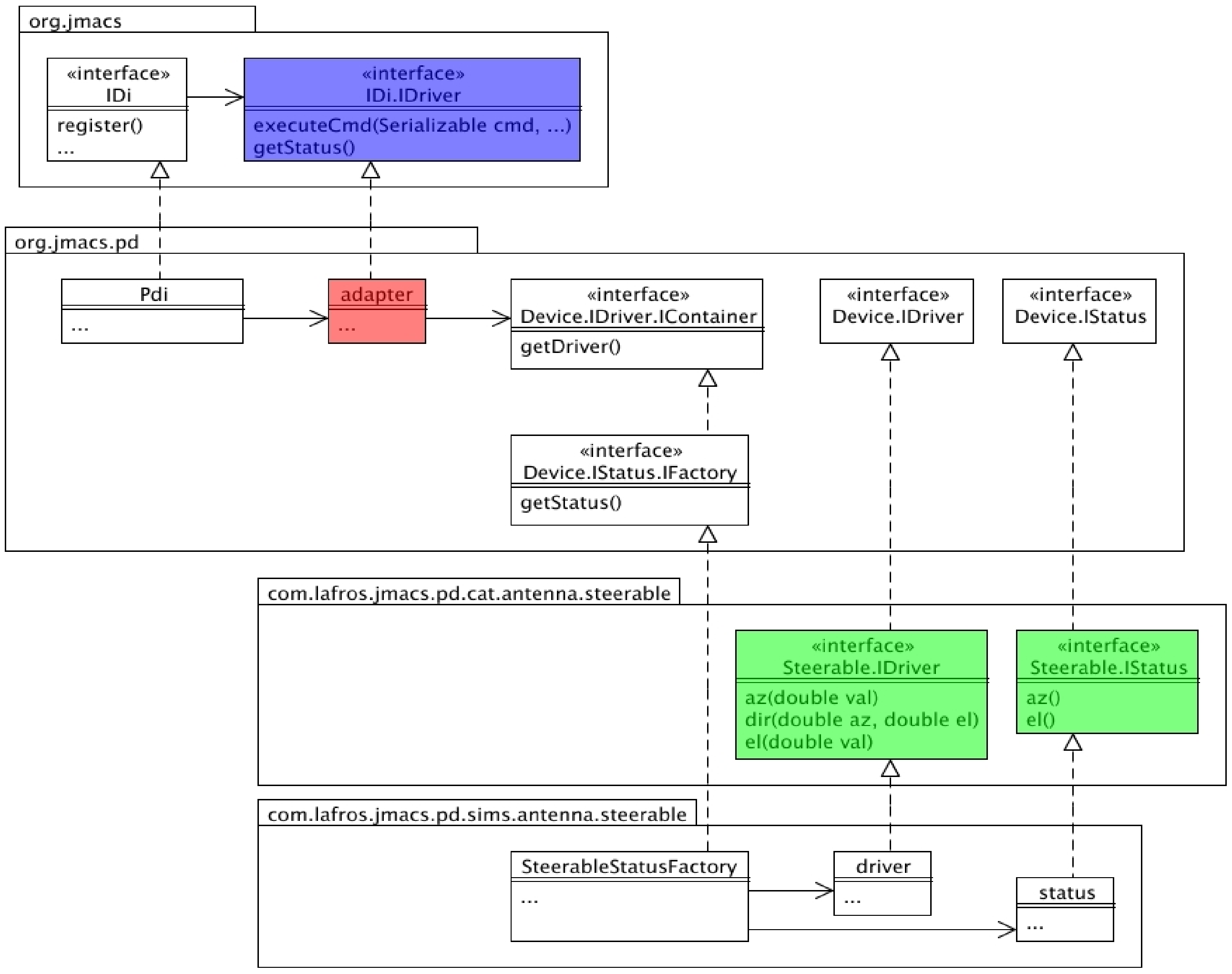


Fig. 1. UML (unified modelling language) class diagram showing how the PD framework's DI plug-in adapter works in the case of a simulator implementation of an abstract steerable antenna PD

package) and JMaCS implementation upper layer (to be supplied by a third party), the lower layer consisting of the DI plug-ins themselves.

Consequently, only the org.jmacs package is required for DI plug-in (`org.jmacs.IDi.IDriver` implementation) development. JMaCS implementation packages are only required for deployment, and must include one containing a class implementing `org.jmacs.IDi.AbstractFactory`, to be instantiated by the deployment code with a call to `org.jmacs.IDi.AbstractFactory.getInstance()`.

4.2 PD framework

The DI plug-in adapter called for by the design is provided by the PD framework (org.jmacs.pd package). The way this works is illustrated in Fig. 1.

Thus, all PD definitions must include a Java interface which extends `org.jmacs.pd.Device.IDriver`. An implementation of this (which might or might not be part of the definition) may then be instantiated and made available locally to any command interpreter (`org.jmacs.pd.IInterpreter`) which is provided, and to any programs (`org.jmacs.pd.IProgram`) run, and also remotely via a dynamic proxy[14].

The methods that must be supplied when writing a PD program are shown in Fig. 2.

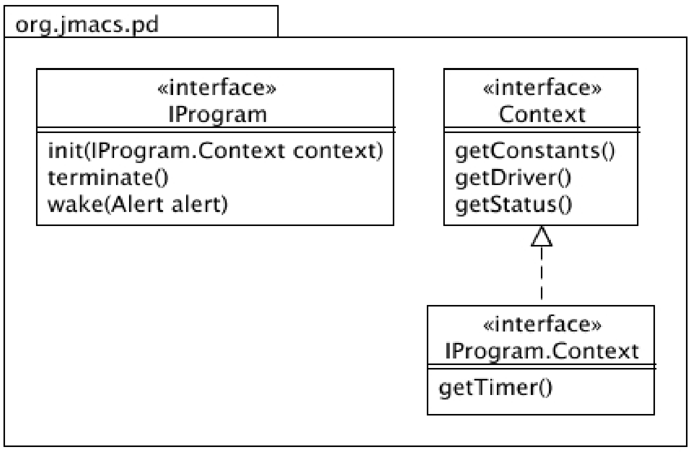


Fig. 2. UML class diagram showing the Java interface to be implemented by PD programs. The wake() method is called periodically, as controlled by the timer.

The PD framework thus introduces a further software layer, as shown in Fig. 3.

JMaCS implementation packages	
org.jmacs	
DI plug-ins	**org.jmacs.pd**
	Fully implemented PDs

Fig. 3. Layering of the software, including PD framework

4.3 Negotiating the Internet

As mentioned in §3.2, all device status samples are required to fit into the payload of a UDP datagram, and initially, connectionless (network) transport[12] was used throughout.

It was later discovered that there was effectively a limit of only a few kilobytes to the size of UDP datagrams which could be sent over the internet. This necessitated significant changes to the JMaCS implementation, to enable connection-oriented transport[12] to be used for sending samples over the Internet. Their dissemination from local and remote DIs involves the stages depicted in Fig. 4.

14 http://java.sun.com/javase/6/docs/technotes/guides/reflection/proxy.html

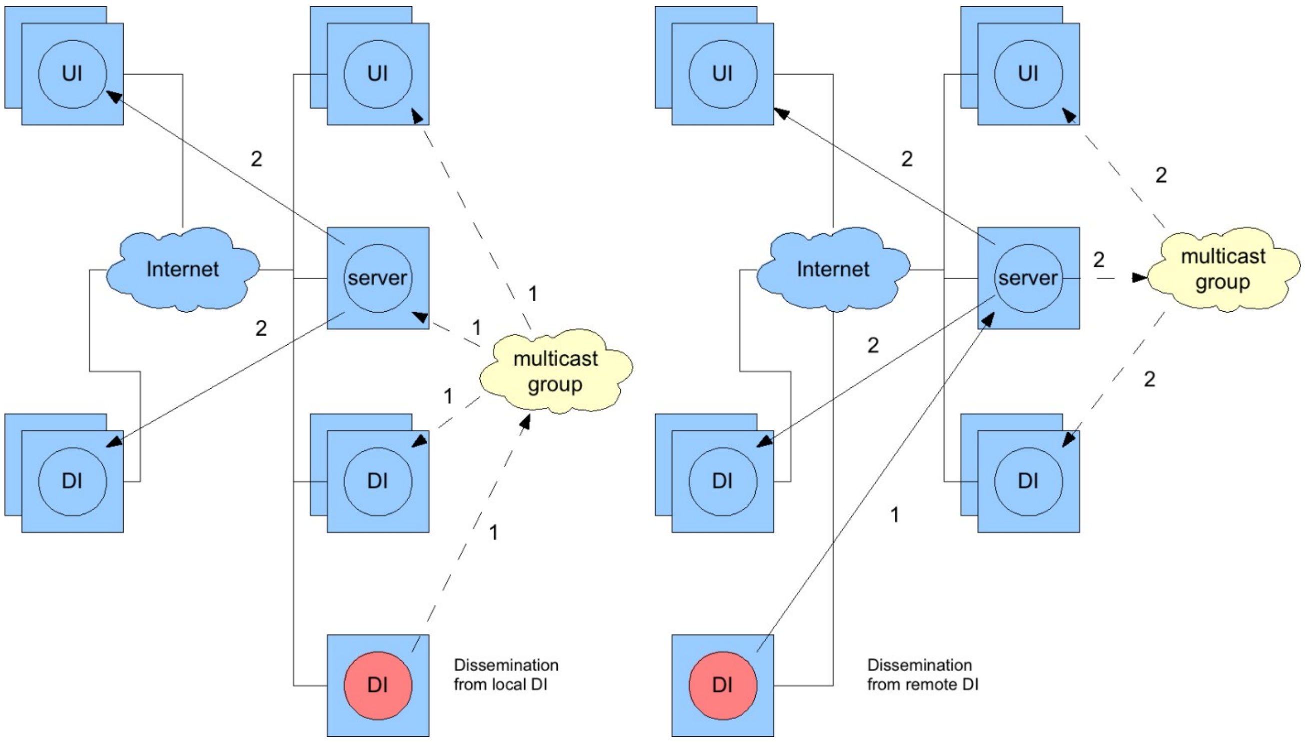


Fig. 4. Dissemination of device status samples from local and remote DIs. Arrows with dashed and continuous lines indicate the use of connectionless and connection-oriented transport respectively.

Certain problems were also encountered relating to security measures such as network address translation (when running clients from certain remote locations), and changes were required in order to accommodate cases where this results in changes to port numbers.

4.4 Enabling more numerous remote subscriptions

Tests in which some UI clients were (connected and) subscribing locally—having device status samples multicast to them directly—and others remotely—having the samples relayed to them individually by the server—revealed that this relaying of the samples in the latter case was resulting in a severe bottleneck.

The following two changes to the implementation were made, to help alleviate this. Instead of making only a single server thread responsible for sending all samples to all remote subscribers, and having only a single subscription connection to each remote subscriber, there is now a separate thread dedicated to each DI being subscribed to, and a separate connection per remote subscription. This allows different samples to be sent (by different threads, over different connections) concurrently.

One further change which it is anticipated will also help, but which has not yet been made, is to have the server store each device status sample in a special 'direct' buffer, thus avoiding having to copy the data back and forth between the native and Java memory spaces.

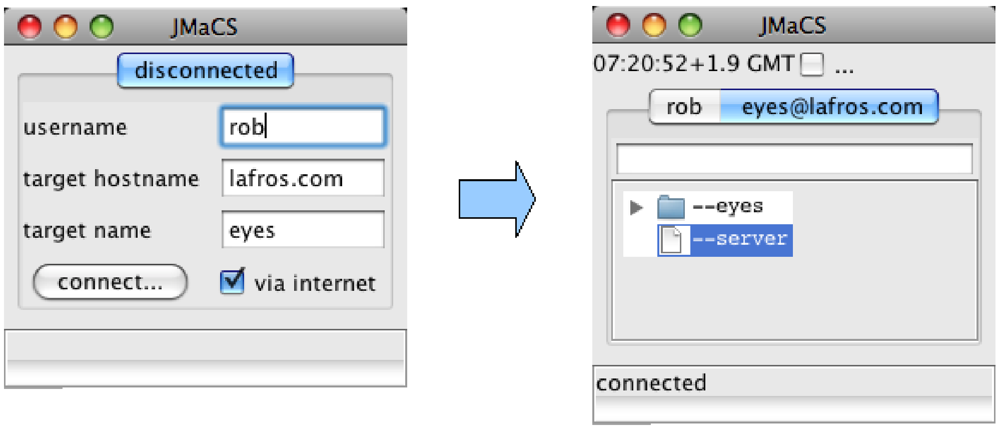
4.5 User interface

The AWT[15]-based GUI of the original experimental software has evolved into a Swing[16]-based one utilising a set of JUICe[17] libraries which have been developed in parallel with JMaCS.

Among these is the JUICe.app application framework, which allows the UI client to be run either as a stand-alone application (using Java Web Start[18]) or as an applet (embedded in the web-browser window).

15 Java's Abstract Window Toolkit
16 standard component set built on top of the AWT
17 http://lafros.com/juice
18 http://java.sun.com/developer/technicalArticles/WebServices/JWS_2/JWS_White_Paper.pdf

As shown in Fig. 5, connecting to the target creates a viewer pane, containing a DI navigator (as suggested in §3.4) and scrollable area in which to display the monitor windows and control panels (optionally) supplied by each DI, together with a command line (whose commands will be sent to the selected DI).



username: rob
target hostname: lafros.com
target name: eyes
[✓ via internet]

07:20:52+1.9 GMT ...
connected

Fig. 5. Connecting to the demonstration target

The following features are implemented:

- compensation for any clock lag or drift (relative to server);
- synchronised refreshing of monitor windows (after a set delay), where the titles of any which could not be refreshed are highlighted;
- intrusive alerts (issued by the UI client itself, together with any monitor windows written using the JUICe.alerts library);
- 'hot deployment'[19] of downloaded classes where appropriate;
- interactive configuration of PD program properties.

4.6 Access control

Users are required to log in to the target they specify (by host name and target name). Authorisation and control-privilege is granted by the target's administrator. Any controlling user may assume the role of 'principal user', in which case all other users will be prompted for confirmation whenever they submit a command.

No access-control has yet been implemented for DIs: registration is not yet password-protected, and all have control privilege. However, the DI plug-in API does supply the name of each command's originating DI (or null in the case of a user), so the recipient may decide not to execute a command from an unknown source.

5. DEMONSTRATION

5.1 The demonstration target

This is[20] an easily understood 'installation', intended to demonstrate, amongst other things, how one DI may control others, in this case simply the root[21] controlling its two children. Thus, we have a PD representing an eyes 'device', together with two steerable antenna ones named 'eyes.left' and 'eyes.right'. The user first submits a command to the root, via its control panel (or the UI's built-in command line), and the root then sends corresponding commands to its children. Two 'eyes' commands are available—one to 'look' at a particular point in space (Fig. 6) and another to track a projectile from a specified launch site to a specified target one (Fig. 7).

19 technique for ensuring that the latest versions are always loaded
20 publicly accessible to account holders at http://lafros.com/jmacs/impln at time of writing
21 the DI having the same name as the target

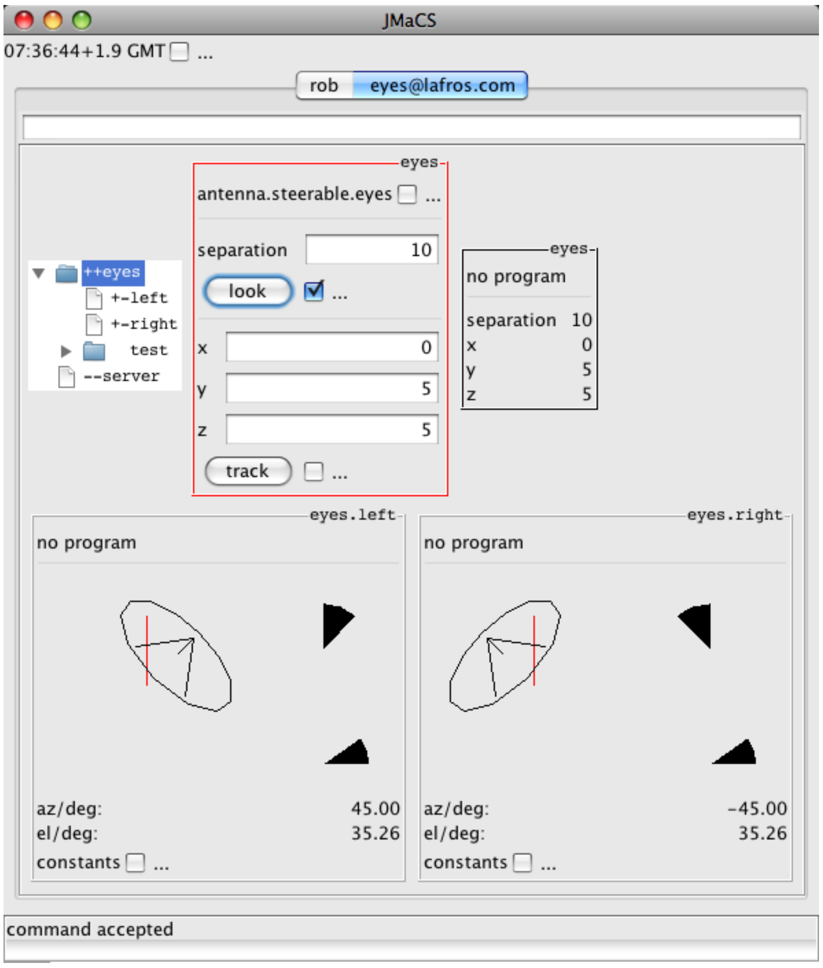


Fig. 6. Command to 'look' at a point in space

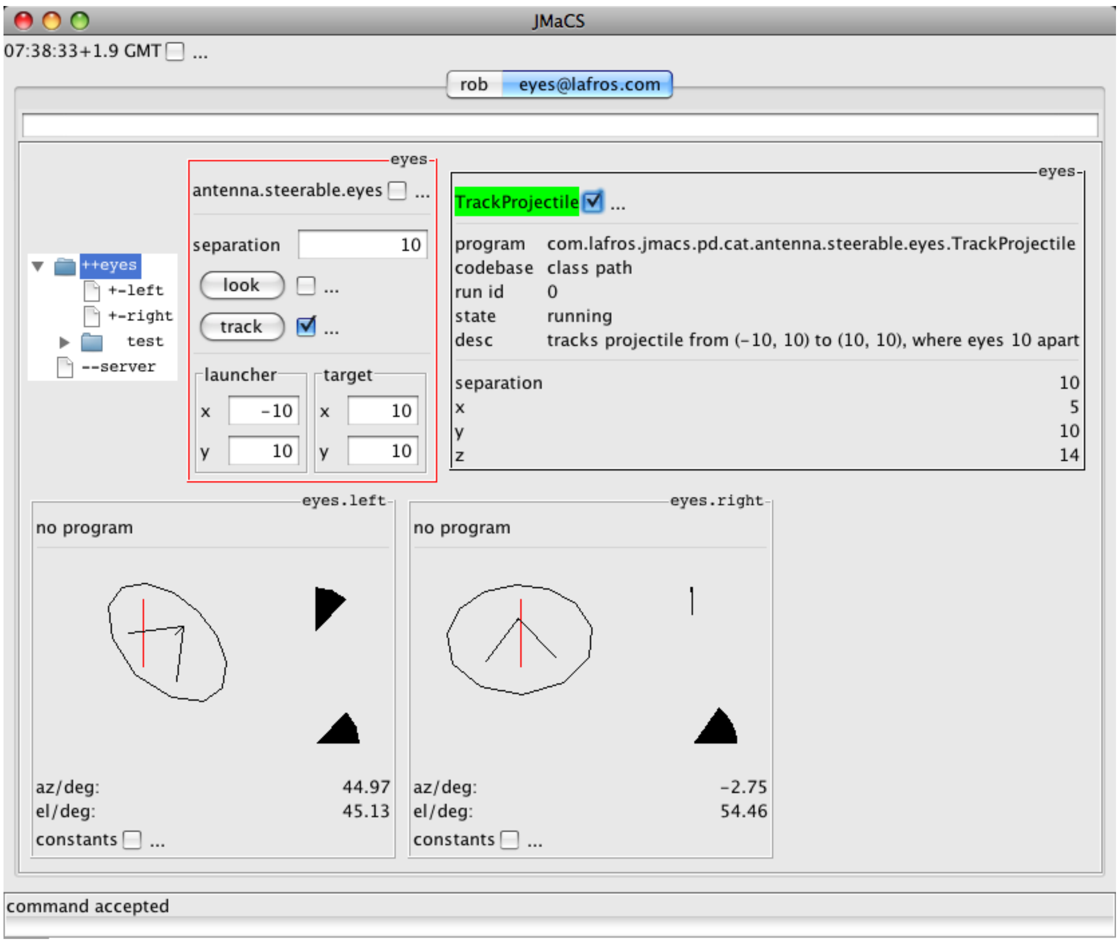
Note that the track command also demonstrates running a PD program. Here, the program is one instantiated by the interpreter implementation supplied by the eyes PD, and is run by the eyes DI itself. It periodically calculates the next position of the projectile, and from this, the corresponding azimuth and elevation angles of each antenna, then sends the appropriate commands to the antenna DIs. It is also possible for programs to be sent as commands themselves, and even (as mentioned in §4.5) for their properties to be configured interactively via the PD's control panel, as shown in Fig. 8.

It should be noted that none of these DIs interfaces to any hardware device. The steerable antenna ones are identical instances of a simulator implementation of an abstract PD named com.lafros.jmacs.pd.cat.antenna.steerable, defined for use with the radar mentioned in §2.1 (where the DI did interface to the hardware).

It should also be pointed out that the demonstration target is not, at the time of writing, strictly distributed, since all of the DIs are running on the same host as the server.

5.2 Load testing

The demonstration target also has an additional ten DIs named 'eyes.test.load 0-9', as shown in Fig. 9. These, again, are software-only, and simply allow the size of their status samples to be adjusted, purely for the purpose of load testing. Thus, various numbers of users, both local and remote, were logged in, various numbers of load-testing DI monitor windows opened, and various sample sizes and sampling periods configured. In each case, the number of monitor windows which JMaCS failed to refresh on each sampling boundary (after a suitable synchronisation delay), as determined by counting the number whose titles were highlighted, was noted. It was found that the bottleneck referred to in §4.4 was still present, although less severe, and now thought to be limited by the external factor of Internet


JMaCS

07:38:33+1.9 GMT ☐ ...

rob | eyes@lafros.com

++eyes
+-left
+-right
test
--server

eyes

antenna.steerable.eyes ☐ ...

separation	10

look ☐ ...

track ☑ ...

launcher		target	
x	-10	x	10
y	10	y	10

eyes

TrackProjectile ☑ ...

program	com.lafros.jmacs.pd.cat.antenna.steerable.eyes.TrackProjectile
codebase	class path
run id	0
state	running
desc	tracks projectile from (-10, 10) to (10, 10), where eyes 10 apart

separation	10
x	5
y	10
z	14

eyes.left

no program

az/deg:	44.97
el/deg:	45.13

constants ☐ ...

eyes.right

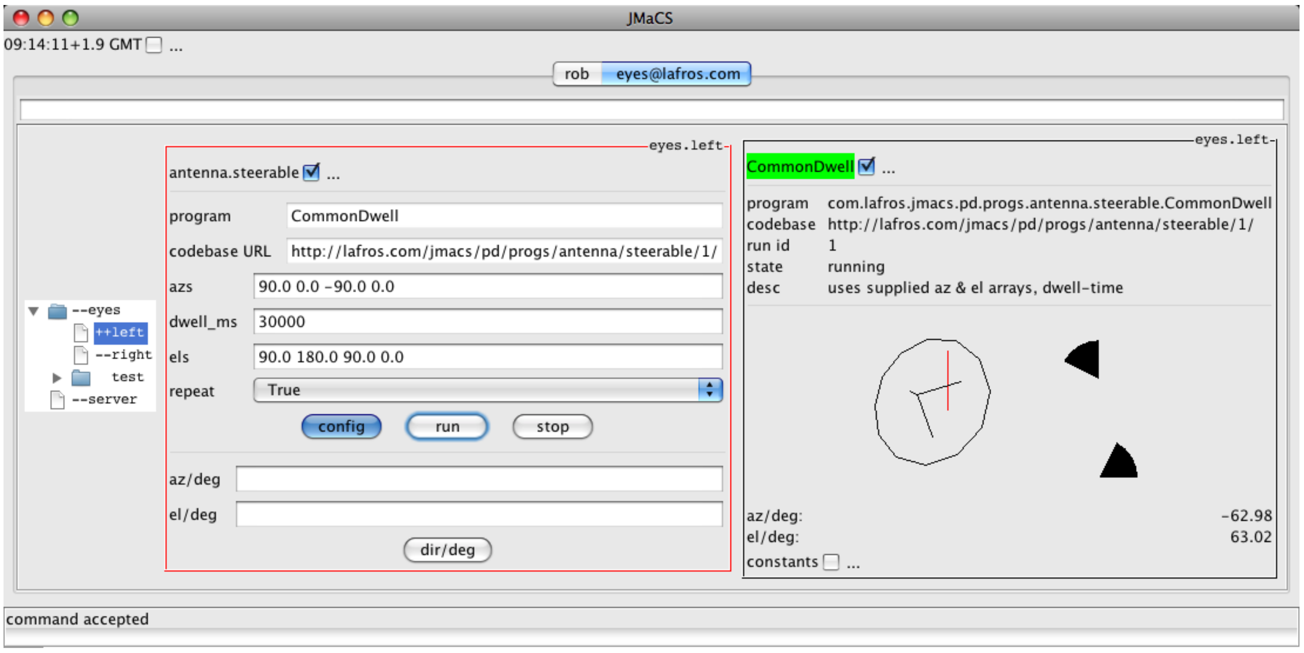
no program

az/deg:	-2.75
el/deg:	54.46

constants ☐ ...

command accepted

Fig. 7. Command to track a projectile


JMaCS

09:14:11+1.9 GMT ☐ ...

rob | eyes@lafros.com

--eyes
++left
--right
test
--server

eyes.left

antenna.steerable ☑ ...

program	CommonDwell
codebase URL	http://lafros.com/jmacs/pd/progs/antenna/steerable/1/
azs	90.0 0.0 -90.0 0.0
dwell_ms	30000
els	90.0 180.0 90.0 0.0
repeat	True

config | run | stop

az/deg	
el/deg	

dir/deg

eyes.left

CommonDwell ☑ ...

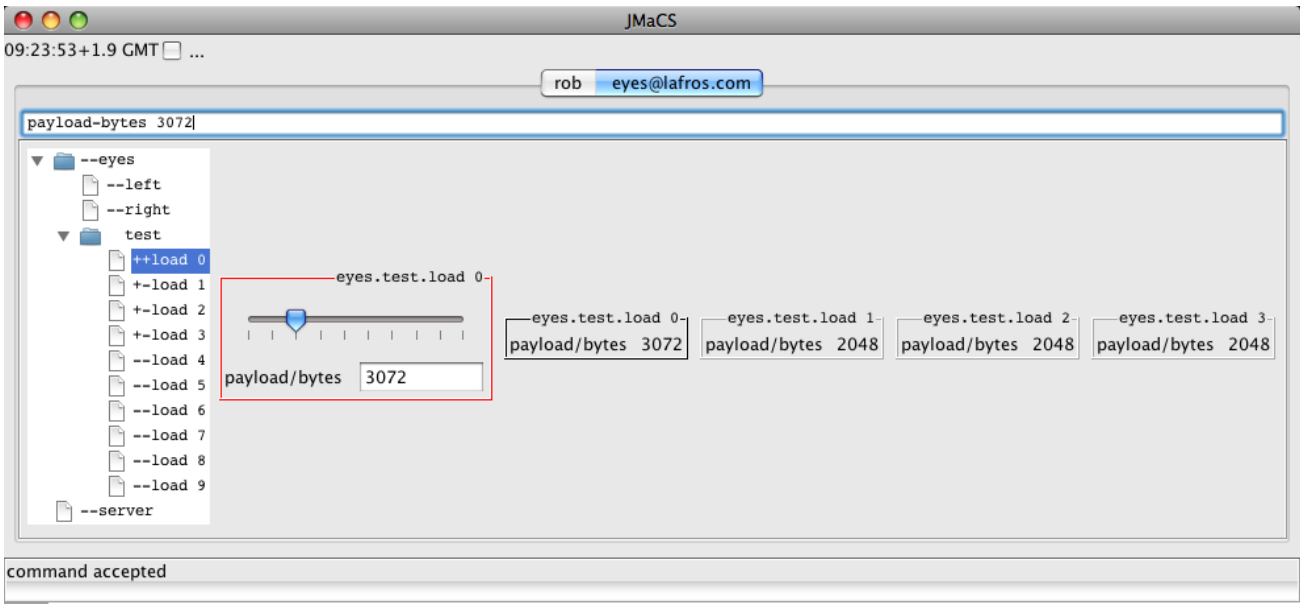
program	com.lafros.jmacs.pd.progs.antenna.steerable.CommonDwell
codebase	http://lafros.com/jmacs/pd/progs/antenna/steerable/1/
run id	1
state	running
desc	uses supplied az & el arrays, dwell-time

az/deg:	-62.98
el/deg:	63.02

constants ☐ ...

command accepted

Fig. 8. Interactive program configuration (in this case, to execute a 'figure of eight')



```
09:23:53+1.9 GMT ...
payload-bytes 3072
```

```
command accepted
```

Fig. 9. Load-testing DIs, showing how textual commands generated by control panels may be recalled to the command line

connection upload bandwidth. This was found[22] to be about 32KiB/s, corresponding to e.g. four remote users, each having four monitor windows open, where each monitor is sent a 2KiB device status sample (on each sampling boundary). This configuration was easily achieved using sampling periods down to 2s, even with all four UI clients running on the same host.

6. CONCLUSIONS

The JMaCS software has been presented. Testing using the demonstration target has shown that it works well within the confines of the test. More extensive testing is therefore now called for, after making the following changes:

- running the JMaCS server on a host with more Internet upload bandwidth;
- running the DIs on one or more separate hosts, preferably including remote ones.

It would also be of interest to run the system for a target whose DIs interfaced to hardware devices (rather than simulators). Based on the above demonstration, and subject to the suggested further tests having favourable outcomes, it is tentatively claimed that,

- monitoring and control using JMaCS is now viable;
- the benefits of employing Java's remote polymorphism have been demonstrated.

Finally, two conclusions are drawn from the experience of developing the software.

- Only very clearly defined Java programming projects, such as implementing a well-defined plug-in, should be attempted on a casual basis. The Java platform does indeed put facilities within reach that previously have required programmer specialisation. However, a depth of understanding is still required in order to make effective use of them.
- Where there is a need to incorporate native code, interfacing to hardware devices can sometimes be quite challenging. However, it is anticipated that more recent versions of the Java platform have reduced this need. Indeed, with the advent of a realtime edition[23] of the platform, it should be possible to write the hard-realtime part of the DI in Java as well.

22 using http://www.broadbandwatchdog.co.uk/speed-test.php
23 http://java.sun.com/javase/technologies/realtime

ACKNOWLEDGEMENTS

Work began on the original experimental software in January, 1998, in connection with a three-month Java feasibility study[13] funded by the EISCAT group at RAL in the UK. This continued up until the end of 2001, during which time the author was employed by EISCAT (at the Svalbard Radar), and under the tutelage of the then science director, Tony van Eyken.

The author is grateful to the following, for useful conversations and help on various occasions subsequently: Thor Bendiksen, Math Smith, Ian McCrea, Steve Crothers, John Hynes, Michelle Henderson, James Wadsworth, Jon Smith, Tony van Eyken, Noora Partamies, Paul Eglitis, Patrick Guio, Chris Thomas and Nigel Wade.

REFERENCES

[1] Rob Dickens, Secure remote monitoring-and-control for the EISCAT Svalbard Radar: a case study in Java object-oriented design, 9th International EISCAT Workshop talk (Aug 1999).
[2] Rob Dickens, Monitoring and control of the 'radar.eiscat.esr' device, 10th International EISCAT Workshop poster (Jul 2001).
[3] Röttger, J., U.G. Wannberg and A.P. van Eyken, The EISCAT Scientific Association and the EISCAT Svalbard Radar Project, J. Geomag. Geoelectr., 47, 669-679 (1995).
[4] Bjørnå, N., B. Hultqvist, W. Kofman, J. Roettger, K. Schlegel, T. Turunen and D.M. Willis, The EISCAT Polar Cap Radar: Report on the design specification for an incoherent scatter radar facility based on the archipelago of Svalbard, prepared by the Polar Cap Radar Working Group established by the EISCAT Council, with financial support from the Rutherford Appleton Lab, UK (Nov 1990).
[5] Peter C. Dibble, Real-Time Java Platform Programming, Sun Microsystems Press, pages 7-9 (2002).
[6] James Gosling and Henry McGilton, The Java Language Environment, http://java.sun.com/docs/white/langenv (May, 1996).
[7] Daniel J. Berg and J. Steven Fritzinger, Advanced Techniques for Java Developers, Wiley, Chapter 7 Mastering Java Remote Method Invocation Techniques (1997).
[8] Rickard Öberg, Mastering RMI, Wiley (2001).
[9] Esmond Pitt and Kathleen McNiff, java.rmi, Addison Wesley (2001).
[10] For example, Steven John Metsker and William C. Wake, Design Patterns in Java, Addison Wesley, Chapter 3 (2006).
[11] For example, Peter Coad and Mark Mayfield, Java Design, Yourdon Press (1997).
[12] Douglas E. Comer and David L. Stevens, Internetworking with TCP/IP Volume III - Client-Server Programming and Applications, Prentice Hall, §2.3.5 (1994).
[13] Rob Dickens, On the advisability of adopting Java for development in the future, Report prepared for EISCAT group, RAL, UK (Apr 1998).