

msg: A Message-Passing Library

John Roll, Jacob Mandel

Smithsonian Astrophysical Observatory, 60 Garden Street, Cambridge, MA 02138

Abstract. As the foundation for the SAO MMT Instrument control software, we have developed an ASCII protocol for passing messages between client and server applications. Server interfaces are described as registered commands and published values. Clients may execute these commands and subscribe to published values. The mechanics of exchanging data between the client and server over a TCP/IP socket are handled by the library. The protocol and its implementing libraries have been designed with simplicity in mind. The simplest server can be written in three lines of code. However, we also have used the msg library to build complex systems with dozens of registered commands and hundreds of published values.

Introduction

Our message-passing library provides an elegant and practical solution to the problem of connecting together the functions of a data system. The API is built on a publish/subscribe metaphor that is easy to use. A server makes public (“publishes”) data and commands. Clients request notification (“subscribe”) about changes to those data. Clients may also execute the server’s commands and actively “set” or “get” the data a server makes public.

The table below represents the code schematic of a service interacting with a client. These are all of the routines that are needed.

Server	Client	
msg_server()	msg_client()	Server and client initialization.
msg_allow()		Server host-based authorization.
msg_register()	msg_command()	Server registers available commands. Clients may call commands.
msg_publish()	msg_subscribe()	Server publishes values, clients may subscribe to value(s) to allow passive notification of changes.
	msg_set()	Clients may actively set or get values that have been published by the server.
	msg_get()	
msg_loop()	msg_loop()	Servers and clients enter the event loop to allow messages to be processed.

With the exception of the routines for specifying the data types on “set/get” calls and some convenience routines for synchronous and asynchronous com-

mands, these routines cover 99% of the msg API. Simple server and client programs are thus very short and easy to code.

Protocol

A message in our protocol is a new-line terminated, variable length ASCII string of the form:

```
[#] command args\n
```

where:

- [#] The packet number signals the need for acknowledgment. If omitted or 0, no acknowledgement is expected. Server packets always have even numbers, while client packets have odd numbers.
- `command args` The message itself consists of all characters between the message number and the terminator. A command may not begin with a numeric character, to avoid conflict with packet numbers.
- `\n` – A new-line character terminates the message.

Our protocol implementation defines a basic command set to handle the passing of scalar values between clients and servers. Servers can easily add other application specific commands via the `msg_register()` call. In addition to passing data as command arguments and returned values, application specific commands may also define inter-packet data. Because such data are not necessarily terminated by a new-line, the sender and receiver of this data must each agree to write/read exactly the same number of bytes or they must define their own logical EOF mark.

Telnet Test

The protocol stream is made up of ASCII text. This simplification led directly to our “telnet test” requirement: any changes to the protocol had to preserve support for manual control in a telnet session. Meeting this test soon became an important design goal of our implementation, since it helped us focus on simplicity as a major aim. From a practical standpoint, telnet is an invaluable debugging tool.

```
john@snappy.harvard.edu : !tel
telnet snappy 2001
Trying 192.168.1.100...
Connected to snappy.
Escape character is '^]'.
1 set mode flat
1 ack
2 get camera
2 ack on
3 getcamerascale 1
3 ack 14 14
```

Implementation

We have implemented our message passing protocol in both C and Tcl. The C and Tcl libraries have slightly different API's in order to take advantage of the strengths of each language, but are very similar in overall design. The C version is written to be efficient in large multi-threaded instrument control services. The Tcl version is written to enable quick and simple GUI clients using Tcl/Tk widgets. The latter has also proven useful in creating simple servers, simulators, and protocol translators, where the speed of C is unnecessary and where Tcl's string capabilities are needed. Also, the Tcl implementation allows other Tcl/Tk programs to be used in close cooperation with our control systems. This includes the ds9 image display and control systems at the MMT Observatory.

Another important aspect of the msg implementation is that the library allows systems of clients and servers to recover automatically when an individual component is killed and restarted. This is a very valuable feature when working with complex hardware that is not yet robust itself.

Lab Experience

The msg library has been used in the Hectospec positioner lab over the last four years to develop and calibrate a 12-axis robotic positioner.

Its server systems include:

hecto The robotic positioner control server with several dozen message commands and several hundred status values.

snappy A multi-threaded frame grabber/controller connecting three intensified CCD cameras, which check positional accuracy for the robot in real-time.

stepper A server to control VME-based stepper motor controllers.

epbox A very simple server allowing access to the electronics peripheral box via an RS-232 serial line.

Several GUI interfaces are used to monitor the state of these complex systems by subscribing to the servers' published state values. Many of these values are displayed directly on a GUI as numeric values, while others are translated by the GUI to color and position indicators. The entire system can be run from command scripts or button presses on the GUI. The message protocol updates critical values 20 times a second. Other values are updated only once a second, or less if they are unchanged.

Each server has a companion simulator, written in Tcl. These simulators present the same message interface as the actual hardware control servers. High-level GUI and scripting software can be tested against the simulators without requiring the hardware to be available.

Telescope Experience

The msg library has been used to create an instrument control system for the MMT mini-cam instrument, which is the first of SAO's facility instruments for

the converted MMT Telescope. The mini-cam hardware was recently delivered to the MMT and will soon be available for use by astronomers.

Its server systems include:

topper The camera top box controller, including two shutter blades and two large filter wheels.

scanserv A simple barcode scanner, which determines the filters currently loaded into the topbox.

detector The science CCD array controller. The detector controller is multi-threaded, and handles the message passing protocol and the ICE protocol used by the IRAF ccdacq package.

keith The CCD dewar temperature monitor server.

guidecam The guide camera CCD array controller

guide The guide error computation server.

focus The MMT secondary control server.

telescope A protocol translator which talks to the MMT telescope control server.

As in the hectospec lab, the mini-cam system has GUIs to monitor the state of its servers and command scripts to control its operation. Some of the most important benefits of using the message based APIs for our systems came during recent systems integration at the MMT. The telescope protocol translator used to connect the several internal clients and servers to information and control of the telescope was written in a few hours after we arrived at the MMT. The focus server, used to control the telescope secondary with errors generated by the guide error computation server, was written in a few minutes. Here is the source that was added to the Tcl language secondary control program:

```
source msg.tcl

proc FOCUS.focerror { err args } {
    global focus

    set focus [expr $focus + $err]
}

msg_server    FOCUS
msg_allow     FOCUS cfaguide
msg_publish   FOCUS focus focus
msg_register  FOCUS focerror
msg_up        FOCUS
```