MSOTCS: A New Telescope Control System for the Australian National University's 2.3m Telescope at Siding Spring Observatory

Jon Nielsen and Gary Hovey
Research School of Astronomy and Astrophysics
Australian National University, Canberra ACT 2611, Australia

Introduction

The Mount Stromlo Observatory Telescope Control System (MSOTCS) is software written in C++ running on the real-time operating system QNX6. It has been written to accommodate new remote observing software, TAROS. MSOTCS is responsible for the control and safety of the hardware which constitutes the optical support structure of the telescope and of associated plant which controls the observing environment such as ventilation, air-conditioning, building/dome rotation, shutters, windscreens, safety interlocks etc. Most importantly, the TCS implements or controls the servo-mechanisms which effect telescope axis motion, instrument rotation, focusing and secondary mirror positioning. For MSOTCS we chose to employ a proprietary astrometric kernel called TCSpk (developed by Pat Wallace) because (i) it is computationally rigorous but reasonably efficient, (ii) it shares code with the TPOINT telescope pointing analysis package ensuring consistency of pointing correction implementation, (iii) it is linked with the widely used SLALIB astrometry and conversion routines, and finally (iv) because it is a library of well-documented C-language routines it offers a kernel which can be cleanly incorporated into our chosen system architecture. Figure 1 shows the architecture of MSOTCS.

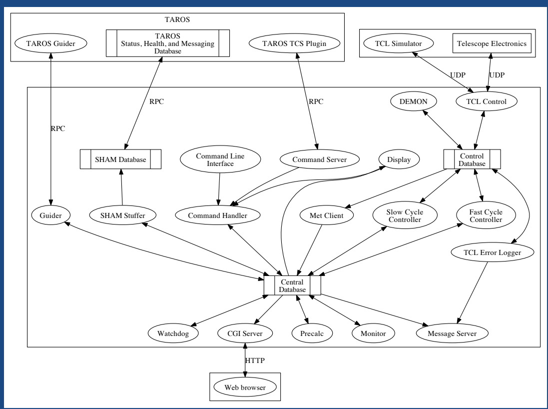

Figure 1. MSOTCS architecture showing processes and shared memory databases

Planning for Reliability

Providing Information to the Observer

MSOTCS has been designed with robustness and reliability in mind right from the start. It has also been designed to provide as much information about the operating environment as possible, including helpful suggestions to the observer. For example if the telescope is tracking when the primary mirror cover is closed, a warning is generated. The system will very rarely refuse to perform an action because of the configuration or the state of the hardware, but it will issue such warnings as it thinks will be useful to the observer. For example it is still possible to operate the telescope if the instrument rotator or the telescope focus are reporting an error. MSOTCS does not attempt to second guess whether or not it is sensible to operate without rotator or focus control – it assumes that the observer is capable of making that decision. However, when it comes to situations that are potentially damaging to the telescope, such as when humidity or wind speed is excessive, MSOTCS will close the building shutter to protect the telescope. Note that in the case where the observer is physically at the telescope (rather than operating it remotely) it is possible to override this behavior (as a side-effect of the "remote observer lockout" switch).

Handling Errors and Restarting Processes

Each process that is a part of MSOTCS has been designed to be highly cohesive and loosely coupled to other processes via the central database, which resides in shared memory. This design means that most processes are individually restartable. From this then flows the strategy that whenever a process encounters a serious error it simply logs the details and exits. If the process is restartable, the watchdog process will restart it and the operation of the system will continue with relatively minimal impact. In particular, as there are only two processes required for the telescope to track (these being the tcl_control process that interfaces to hardware, and the fast_cycle_control process that provides the main telescope loop), as long as these two processes continue to run, the telescope will continue to track despite other processes dying and restarting.

The contents of the *main()* function of each process is wrapped in a try-catch block, so that any exceptions that are thrown and not caught lower down in the code will always be caught at the top level and result in an error message and a graceful exit.

Perhaps the most serious error that can ruin this strategy is for a process to be holding a resource such as a shared lock when it exits. If this were allowed to happen then the process could not be restarted, as the lock can now no longer be relinquished. To avoid this, we use the RAII (Resource Acquisition Is Initialization) strategy. Locks (in our case mutexes in shared memory) are held by objects whose scope is exactly the required scope of the lock. Thus when the object holding the lock goes out of scope (either through the normal path of execution or because of the throwing and catching of an exception) the lock is released automatically. We made use of a class that we had already written for another project to provide this functionality. Others may wish to investigate the Boost C++ libraries (http://www.boost.org), which provide a scoped lock model.

Keeping the whole system running

As MSOTCS is designed to run as an embedded system it is necessary for it to start automatically when the machine it is hosted on is started. This is easy enough to achieve using the standard *rc.local* startup script. Keeping the system running at all times is another requirement. In order to achieve this, the single watchdog process is not sufficient, as it will shut down the system if any of the critical processes exit. We implemented a very simple "safe start" process, the sole purpose of which is to start the MSOTCS watchdog process at system boot time, and to restart it if it ever quits. Because this process is so simple it is much easier to verify through static inspection that it is free from bugs.

Avoiding memory leaks

In order to avoid memory leaks we ensure that every dynamically allocated object is stored in an auto_ptr, where possible. In some cases (for example objects created via the RPC clnt_create function) this is not possible. A wrapper class to implement the RAII strategy would mitigate against this. We did not do this and in fact encountered an RPC-related memory leak because of it. There were two other memory leaks that required debugging effort. The first was in a system gui library and could be worked around once it was found. The second appeared to be a compiler bug that occurred when passing an auto_ptr object as a parameter to a particular function. Reordering of the affected section of code eliminated the leak.

Resetting hardware

One of the biggest hurdles to successful deployment of a remote or automated system is dealing with problematic hardware. If a device needs resetting during the night it is a simple matter for a local observer to do it, and the cost to them is merely that they have to leave the warmth of the control room for a brief period of time. For a remote observer however a troublesome piece of hardware that cannot be reset remotely can mean the end of their night. MSOTCS is able to reset a majority of telescope systems remotely but there will always be some things outside of its control. Whether or not observers are willing to take the risk probably depends on the reliability (both actual and perceived) of the hardware involved.

Messages

Transient or persistent

There are two types of messages in MSOTCS, transient and persistent. Transient messages are typically informational and provide feedback to the observer on their actions. For example, the message that the telescope has acquired the desired object and begun tracking is a transient message. These messages are displayed to the observer in a scrolling messages window. Persistent messages represent the state of the system and are typically warnings and errors. They remain active until revoked by the system. For example, the message that the telescope is tracking but that the mirror cover is closed is a persistent message. This message will remain displayed on the screen until the observer either opens the mirror cover or stops the telescope tracking.

Mnemonics

Every message, both transient and persistent, has a unique mnemonic, which is a string of at most 10 characters. This has greatly assisted development (all message strings are kept in a single header file – only the mnemonics are used in the code), debugging (if the system is displaying a particular error you can very easily find the section of code that is generating it by searching for the mnemonic), and support (when requesting assistance or logging a fault users need only report the mnemonic, not the entire message string). This simple but effective technique was also used in the previous TCS.

Providing data to external clients

Mapping parameter names to database variables

The entire state of the system is encapsulated by the central database. Because this is just a C structure stored in shared memory it is effectively available for any other process on the machine to access. To facilitate external access to this database, a mapping of parameter names to variables in the database is maintained. This is achieved through an STL map keyed on the variable name. The value type is a pointer to the abstract base class Base_StringIO. The derived template class StringIO stores a reference to a variable of any type as well as implementing the virtual functions in Base_StringIO. This approach is necessary because it is not possible to store pointers to a template class directly in an STL map. The map itself is encapsulated in a class called StringIO_Map that provides convenience functions for adding, getting, and setting parameters.

This approach provides a mechanism to store a reference to each variable in the central database, together with the units for that data and simple linear scaling parameters so that, for example, data that is stored in radians can be displayed in arcseconds. We also provide a function that can convert the stored data into a string for display, and a function that can convert a string into the appropriate type for storage in the central database.

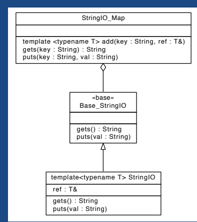

Figure 2. Class diagram for StringIO

Different mechanisms for data retrieval

Four mechanisms are provided by MSOTCS for the retrieval of data by external users. The first and simplest is an engineering tool that simply dumps the contents of the database to standard output. This is possible because every structure in the database has a corresponding *operator<<* defined. Secondly the "view" command allows the retrieval of any named variable from the database via the StringIO_Map class. Thirdly the sham_stuffer process uses the TAROS Status Health and Messaging (SHAM) mechanism to publish a subset of the database to an external machine. Finally the cgi_server process provides a mechanism for a FastCGI compatible web server to query the database via the StringIO_Map class. The regular cgi mechanism requires that the web server start a new cgi process for each new incoming connection. This is very inefficient in our case because the start up cost is high, as it has to instantiate a StringIO_Map object with its hundreds of map entries. FastCGI (http://www.fastcgi.com) allows a cgi process to remain running and process each new incoming connection without restarting. This is ideal for our cgi_server process.

Logging data

In addition to being able to retrieve the instantaneous values of variables, MSOTCS also allows the logging of any variable in the database to a file at 20Hz, which is the rate of the main loop. This is particularly useful for tuning servos and diagnosing encoder faults, amongst other things. The logged data is also available via a web interface that uses the jQuery flot JavaScript library (http://code.google.com/p/flot/) to present the data graphically.

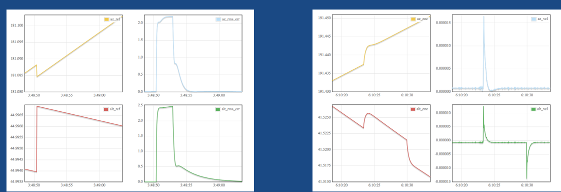

Figure 3. Examples of logged data presented graphically.

Why QNX?

QNX (http://www.qnx.com) provides a POSIX environment that is very like traditional Unixes. Our existing C++ codebase was originally developed under Solaris but the modules we chose to reuse for MSOTCS were easily ported to QNX. The TCSpk codebase also compiles without warnings. However the main advantage of QNX for a TCS is that it is a realtime system with fast and efficient interprocess communication. Apart from communicating through shared memory, processes in MSOTCS also communicate via QNX's native messages and pulses. In QNX this communication mechanism is fast and efficient: indeed, the QNX operating system itself is just a set of cooperating processes communicating in this way (on top of a small microkernel).

QNX has a strong history in realtime, embedded, and instrument control applications and is used extensively in industry (particularly these days in the automotive industry). Free licenses are available for non-commercial use. Academic licenses are also available for eligible institutions.

Although hardware support in QNX is more limited than in some other operating systems (Linux for example) this is not in practice a problem when new hardware is being purchased specifically for the project. We have found QNX to be an excellent platform on which to develop and would recommend it for any project where realtime performance is important.

Contact

Jon Nielsen
Research School of Astronomy and Astrophysics
Mt Stromlo Observatory
Cotter Rd, Weston Creek, ACT 2611, Australia

Email: jon@mso.anu.edu.au
Phone: +61-2-6125-0230
Web: http://www.mso.anu.edu.au/

THE AUSTRALIAN NATIONAL UNIVERSITY