DRAMA 2 - AN EVOLUTIONARY LEAP FOR THE DRAMA ENVIRONMENT FOR INSTRUMENTATION SOFTWARE DEVELOPMENT

T. Farrell, K. Shortridge, Australian Astronomical Observatory, Australia

Abstract

The DRAMA Environment provides an API for distributed instrument software development. It originated at the Anglo-Australian Observatory (now Australian Astronomical Observatory) in the early 1990s, in response to the need for a software environment for large distributed and heterogeneous systems, with some components requiring real-time performance. It was first used for the AAOs 2dF fibre positioner project for the Anglo-Australian Telescope. DRAMA is used for most AAO systems and is or has been used at various other observatories looking for a similar solution. Whilst DRAMA has evolved and many features were added, the overall design has not changed. It was still a largely C language based system, with some C++ wrappers. It did not provide good support for threading or exceptions. Ideas for proper thread support within DRAMA have been in development for some years, but C++11 has provided many features that allow a high quality implementation. We have taken the opportunity provided by C++11 to make significant changes to the DRAMA API, producing a modern and more reliable interface to DRAMA, known as DRAMA2.

INTRODUCTION

The DRAMA API [1] remains the AAO's primary tool for constructing complex instrumentation systems and has been/is being used by various other observatories. With an approach based on the older Starlink ADAM Environment [2], it implements a tasking model; with each named task responding to named messages of a number of different types. In a DRAMA "System", tasks can run across different hosts in a heterogeneous environment. DRAMA was implemented from about 1992 and was designed to be highly portable at a time before ANSI C was available on all machines of interest. It has been run on many flavours of UNIX/Linux, VMS, VxWorks and MS Windows, and provided the ability to write soft[1] real-time applications and with good performance on, for example, 30Mhz 68020 CPUs. The flexibility allowed systems as complex as the AAO's 2dF system [3] to be implemented, making use of the most appropriate hardware for each job across a distributed system.

Most work is a DRAMA task is done in response to "Obey" messages – in effect, command messages; implementing "Actions". The design approach implements co-operative multi-tasking; multiple actions can be running at the same time but must deliberately return control to the DRAMA message reading loop between events to allow other actions to run and for the action itself to be "Kicked" – sent a message to change its behaviour in some way (typically, but not always, to cancel the action cleanly). The approach has worked well and a strongly objected-oriented task design approach was implementable for tasks written in C.

Attempts were made starting about 1994 to implement C++ interfaces for DRAMA, but the results were relatively poor and various different approaches were tried. One of the early issues was the poor portability of early C++ compilers, some features such as templates and exceptions were not reliably implemented and were not portable. Another was that we were still learning the best approaches to use.

Whilst DRAMA tasks using threads of various types have been implemented over the years, DRAMA itself has not supported using threads, with its own co-operative multi-tasking technique sufficient in most cases being more portable then threads were. In the C API, task authors must work around DRAMA when using threads; but in recent times, many libraries for component control have presumed threads are available and thread support has become widespread and is presumed to be available by most software engineers. We had been working on designs for proper thread support for DRAMA over some years, but had not yet implemented it.

C++11 [4] was a major revamp to the C++ language: Threads are now supported using a well thought out approach, by the compilers and standard libraries; Many new features are provided by C++11 that assist library implementers to construct quality interfaces; compilers of interest (GCC and Clang in particular) have implemented the full feature set on machines of interest (Linux and Mac OS X). We have taken advantage of the upgrade of C++ to implement DRAMA2, which will simplify writing and maintaining complex DRAMA tasks.

BASIC DRAMA

To understand DRAMA2, a quick introduction to DRAMA is needed. A DRAMA task executes a message receive loop that dispatches control to event/message handlers when messages arrive. A "Path" is a connection to another task, which is opened as required by application specific code and then used to send messages. There are various types of messages sent between tasks, which may be running across various machines on the network.

1 "Soft" Real-Time: Don't bet a life on an interrupt response, but good enough for ground based astronomy.

ISBN 978-3-95450-148-9

Application specific code is provided to handle the "Obey" and "Kick" message types. The "Obey" message type causes application specific code implementing an "Action" of a specified "Name" to be run. "Kick" messages are used to communicate with a running Action of the "Name" in the message. The implementation of an action of a given "Name" is provided by application specific C language routines that are invoked in response to messages by the DRAMA event loop.

A Parameter system allows a task to publish information about itself, which can be used for enquiries and to display information on GUIs. Get/Set/Monitor type messages containing the parameter "Name" are used to work with parameters in other tasks.

Most messages can have an "Argument" attached, which provides a way of moving data across machines. These arguments are implemented using Self Defining Structures (SDS). These can be of any size and are designed to allow large amounts of data to be sent efficiently between tasks, both locally and across heterogeneous networks configurations. They are used for simple command arguments, complex structures and large image transfers.

Internally, a single global variable (a large structure) is used to maintain all the DRAMA details – in retrospect, possibly a mistake, but this allows a simpler API that did not need a DRAMA task pointer to be passed to all DRAMA API calls.

THE APPROACH TO DRAMA2

The basic approach used in DRAMA2 is based on work previously done to implement DRAMA GUIs with JAVA. In this work, the JAVA Native Interface (JNI) was used to allow JAVA to invoke and be invoked by the DRAMA API. The approach was directed to implementing GUIs but allowed many of the concepts that are required for DRAMA2, to be developed. Four particular areas drive the design:

Implementation as Wrapper around the C API

The DRAMA JAVA interface proved that dramatic changes to the C level interfaces to DRAMA are not required for thread and exception support. By implementing DRAMA2 as a set of wrappers around DRAMA C APIs, compatibility with the large set of existing tasks can be easily maintained and DRAMA2 could be implemented quickly.

Only One Thread Reading DRAMA Messages

There is one and only one thread that actually blocks for and reads DRAMA messages from the underlying message queue. Most of the "DRAMA" internal processing is done within this thread. Other threads can send DRAMA messages (and make other DRAMA API calls) but cannot actually read the messages directly. Much potential complexity is removed and no changes are required to the DRAMA C language internals when using this approach. If another thread needs to wait for a DRAMA message to occur, it must wait on a C++ condition, which is notified by the DRAMA thread when the message arrives.

Locking Access to DRAMA Structures

Locking is important to get right! Systems with multiple locks help to avoid lock wait delays, but significantly increase the complexity of the design required to ensure avoiding deadlocks. Since DRAMA2 is an API available to implement applications, it is much harder to avoid deadlocks when using multiple locks. As a result, only one lock is used and it must be taken by most methods that invoke the DRAMA C API. Use of the lock is normally internal to the DRAMA2 methods, but it can be used by application specific code to access any DRAMA C API not yet available or for application specific locking. Use of the DRAMA2 lock as the only lock in the application would avoid deadlock. The DRAMA2 lock is safe for recursive use – a thread that has already taken the lock will not deadlock if it attempts to take it a second time.

The DRAMA design allows the DRAMA2 message read thread to block waiting for a message without taking the lock. That thread only takes the lock when processing a message. Since the lock is free any time the DRAMA2 message read thread is waiting for a message, there is plenty of opportunity for application threads to lock access to DRAMA and send messages themselves.

Status and Error Reports vs. C++ Exceptions

The DRAMA C API uses an inherited status convention. Most functions have a "status" argument, which is a pointer to an integral type. Functions are expected to check the value pointed to is zero on entry. If it is not, they return immediately. If an error occurs, status is set to a non-zero value. The Inherited status convention neatly avoids the long series of nested if statements typical in the use C APIs that returns a value to indicate if they failed. The integer status value is passed as the result of DRAMA messages, allowing other tasks to determine if an Action has failed and to interpret the status value. An Error Reporting System (ERS) enables extra contextual information to be added when errors occur.

In C++, it is natural to replace the inherited status approach and ERS by exceptions. An exception class is provided which is a sub-class of std::exception. Any DRAMA2 method invoking a DRAMA C API must check the status returned and, if bad raise an exception. The DRAMA2 exception class stores the integer status value and information about the context and location of the exception.

At any point where the DRAMA C API must invoke a DRAMA2 method, there must be an interface function. That has an inherited status argument. This function must catch any exception thrown by DRAMA2. If the exception is the DRAMA2 exception, the original status value will be available and can be returned to the C API as the status of the call, otherwise another status value

ISBN 978-3-95450-148-9

will be returned. Any extra context available in the exception will be reported using ERS.

TASK STRUCTURE

A Simple Task

Example 1, below, shows "Hello World" in DRAMA2. This program implements a task named "TASK1", which has just one Action – named "HELLO". Sending an Obey message with the name "HELLO" will result in the message "Hello World" being output and the task then exiting. The action is implemented by sub-classing the abstract class "MessageHandler" providing an implementation of "MessageReceived()". Any number of actions can be added in a similar way and they don't normally cause the task to exit, and may be invoked multiple times in sequence.

```
#include "drama.hh"
using namespace drama;
// Action definition.
class Action1 : public MessageHandler{
private:
  Request MessageReceived() override {
    MessageUser("Hello World");
    return  RequestCode::Exit;
  }
};
// Task Definition
class ExTask : public Task {
private:
  // actions
  Action1 Action1Obj;
public:
  ExTask(const std::string &taskName)
:
    Task(taskName) {
    Add("HELLO",
        MessageHandlerPtr(&Action1Obj,
                          nodel()));
  }
};
// Main program.
int main() {
  CreateRunDramaTask<ExTask>("TASK1");
  return 0;
}
```

Example 1: "Hello World" in DRAMA2.

Threaded Actions

In Example 1, the "HELLO" action is running in the main DRAMA2 thread. Whilst it can "Reschedule", in the traditional DRAMA way to return control to the message thread, the intent of DRAMA2 is to support running actions in threads. The class "thread::TAction" is an abstract sub-class of "MessageHandler". The user of this class must provide the method "ActionThread", which is invoked within a thread when an Obey message of the specified name is received. When the thread completes, DRAMA2 is informed and the action is marked as completed. Importantly, all details of thread creation; joining the thread etc. is hidden by DRAMA2. Any exception thrown by the thread is reported via DRAMA2 as an action failure – the task does not abort. Example 2 below shows a simple implementation of a thread action.

```
// Action definition.
class Action1 : public
thread::TAction{
public:
  Action1(std::weak_ptr<Task>
theTask):
    TAction(theTask) {}
private:
  void ActionThread(const sds::Id &)
override {
    MessageUser("Hello World - from a
thread");
  }
};
```

Example 2: Threaded "Hello World" in DRAMA2.

Action threads can create their own sub-threads, which can interact with DRAMA, but the implementer is then responsible for handling creation, joining the thread, dealing with exceptions in the thread, etc.

An important requirement for implementing access to DRAMA APIs from threads is to get the DRAMA "Context" right. In a normal DRAMA task, with only one thread runing, the DRAMA API has been able to presume that certain components of the DRAMA Global structure indicate which action is running, and other information about that action. When a threaded action is started by DRAMA2, this information is captured just before the thread is started. Later, any call to a DRAMA C API from the threaded action must first lock access to DRAMA, save the current DRAMA context and then enable its own DRAMA context. This must be undone when the call to the DRAMA C API is complete. An object of a particular DRAMA2 class is used to wrap this up using RAII (Resource Allocation Is Initialisation) to ensure it is undone correctly even is an exception is thrown. This is done transparently by the DRAMA2 API, but is available to application code if access to other DRAMA C API's is required.

Kicking Threaded Actions

A DRAMA Action can be "Kicked", which provides a method for other tasks to communicate with a running action. Often used for Action cancellation, Kick messages are flexible and a system design may use them to update a running action. The "WaitForKick()" method and related methods allow a thread to wait for a kick message to be received.

Alternatively, a "KickNotifier" object may be created before say entering a CPU intensive loop. These objects create a thread that waits for a kick message. The caller can ask the object if a kick was received and respond correctly.

Sending Messages

A "Path" class is provided to enable sending DRAMA messages to other tasks. In traditional C DRAMA,

ISBN 978-3-95450-148-9

message sending does not block and an action must explicitly reschedule to message handle replies. In DRAMA2, message sending is only possible from a threaded action. The thread, but not the task, is blocked to await replies. As a threaded action can have child threads, they may have any number of messages outstanding at any time. Example 3 shows how to send an Obey message to a server. In this case, the action name is "HELLO".

```
Path server(…)
…
server.Obey(this, HELLO);
```

Example 3: Sending an Obey message.

There are various message sending methods, including the ability to monitor for changes to the values of parameters in other tasks. By default, the methods will block until the subsidiary action completes, but there are features allowing overriding of the default processing of the various possible replies to a message.

OTHER FEATURES

SDS

The sds::Id class provides access to DRAMA's SDS objects, used to send data between tasks. Action implementations can access any structure sent to them and can send such structures as arguments in any message they send. Some complexities of the underlying SDS system made writing a clean C++ interface hard prior to C++11. In C++11, the move assignment and move copy operators proved liberating, allowing an effective and relatively clean interface to be constructed.

The "sds::Id" class is extensive, providing methods allowing the building and accessing of complex structures, as well as providing simple ways of building and accessing typical command line arguments.

The sds::Id class allows SDS structures to be written to and read from files, buffers of various forms (e.g. for sending in messages) and for details of such structures to be listed to streams and other locations, for debugging purposes. Simple and highly efficient access to large arrays is provided.

Accessing Command Arguments

All arguments to actions are sent in an SDS structure, but there is a standard approach to command arguments, which allows simple generic programs to be used send to obey messages to any task. Various simple methods are provided by the sds::Id class to construct such arguments. In the action receiving the message, sds::Id class methods can be used, but there is also an alterative interface – via the "gitarg" namespace. These are a series of classes that create sub-classes of standard types initialised from an SDS structure. For example, a gitarg::Bool uses an SDS structure to initialise a Boolean type, accepting for example string values "YES" and "NO" to indicate the value.

GUIs

DRAMA provides a number of GUI toolkits, Java and Tcl/Tk based GUIs being commonly used at this point. These will continue to work with DRAMA2 tasks. Additionally, new toolkits will be constructed using DRAMA2, with Python likely to be the first.

The support for working with threaded systems easily ensures DRAMA2 can be used with many other modern systems. The first DRAMA2 task implemented outside the package itself was a GUI using the Qt widget set, an extensively threaded environment.

Documentation and Regression Testing

An important part of the implementation of DRAMA2 was to ensure the documentation was created with the package, rather then the tradition of being tacked on later. The "doxygen" tool was chosen as the interface documentation tool and all interfaces have been documented as the code was written.

A 130-page manual has been generated, working through all the many features of DRAMA2. The manual includes a large number of code examples, all of which is available as compilable code. Generation of the detailed manual and the required examples quickly highlighted various flaws or unnecessary complexities in the initial interfaces, allowing them to be revamped before release.

As example programs were generated to demonstrate and test features they have been added to our regression test facilities. As a result, any change to DRAMA2, or the underlying DRAMA software, is automatically subject to extensive testing.

CONCLUSION

DRAMA2 has quickly modernized the development of DRAMA tasks. It is well documented and allows reliable tasks to be written quickly. It allows sequenced code to be written for sequenced jobs, but with all the efficient non-blocking DRAMA messaging facilities available. Much of the (potentially risky) complexity of creating threaded distributed applications is hidden from programmers using DRAMA2, in the most common cases.

REFERENCES

[1] T.J. Farrell, K. Shortridge, J.A. Bailey, "DRAMA: An Environment for Instrumentation Software," Bulletin of the American Astronomical Society, Volume 25, No 2 (1993).

[2] Allan P. M., "The ADAM software environment," Astronomical Data Analysis Software and Systems I, 126 (1992).

[3] Taylor K., et al., "Anglo-Australian Telescope's 2dF Facility", Proc. SPIE 2871 (1997).

[4] ISO/IEC 14882:2011 "Information technology -- Programming languages -- C++" (2011).

ISBN 978-3-95450-148-9