SCAL

Simple C++ Actor Library

Design Notes

Publication History

Date	Who	What Changes
26 October 2015	Brenton Ross	Initial version.

Table of Contents

1 Introduction....4
  1.1 Scope....4
  1.2 Overview....4
  1.3 Audience....4
2 Actor Classes....5
  2.1 Actor....5
  2.2 Impl Template....5
  2.3 Proxy Template....5
  2.4 User Classes....5
  2.5 Messages....5
  2.6 Callbacks....5
  2.7 Supporting Classes....6
    2.7.1 Scheduler....6
    2.7.2 ThreadPool....6
    2.7.3 Printer....6
  2.8 Thread Classes....7
    2.8.1 Clock....7
    2.8.2 Signaller....7
    2.8.3 Selector....7
3 Activities....8
  3.1 Activity Classes....8
    3.1.1 Activity....8
    3.1.2 Predicates and PredicateTable....8
    3.1.3 Rule Engine....8
    3.1.4 Activity Manager....9
  3.2 Notes and Issues....9
    3.2.1 Future Work....9
Appendix A....10

1 Introduction

The Actor software model uses objects that effectively run in their own threads. The public methods of the actor objects are queued and run sequentially.

This model allows us to develop multithreaded applications without having to worry about mutexes and other thread management features in the application level code (they are all handled by the supporting library).

This form of programming has some implications which are discussed in this document.

1.1 Scope

The document is concerned with the implications of using the actor model in C++ programs.

1.2 Overview

Its messy.

1.3 Audience

Anyone interested in using an actor model for C++ programs.

2 Actor Classes

2.1 Actor

The core function is the Actor class which defines an abstract Message class (which is a function object) and which manages a queue of these messages.

The Actor class implements the ThreadFnOwner interface which is used by the Scheduler to notify when thread functions start and stop.

2.2 Impl Template

This is derived from the Actor class and the user class which is its template parameter.

This provides methods for constructing callback objects (see below).

2.3 Proxy Template

This is derived from the user class which is its template parameter. It provides a method for enqueing the messages.

2.4 User Classes

In order to use this library to create an actor object the user must create an abstract base class and classes derived from the Impl and Proxy templates.

Calls to the actor are made to the proxy object. This queues up the calls onto the message queue owned by the corresponding implementation object.

The base class should have a static method which returns a pointer to a new instance of the proxy class. The constructor for the proxy takes a pointer to a new instance of the implementation object.

2.5 Messages

The msg template class uses some of the recent additions to the C++ standard library, such as varadic parameter lists and tuples, to allow us to automatically convert a function call into a message which can be queued and executed at a later point.

Naturally it is important that the parameters are either objects which can be copied or references to objects that will continue to exist.

2.6 Callbacks

Some messages require the called method to reply to the originator of the message. One way of handling this is for the originator to provide a “self addressed envelope” for the reply. A pointer, using the shared pointer template, is passed in the calling method and executed by the callee. A shared pointer is used so that the callback object is automatically deleted once it is no longer needed.

2.7 Supporting Classes

2.7.1 Scheduler

A singleton class called Scheduler is responsible for assigning actors to threads. Its main method, run(), should be called by main() once the actors have been created and the initiating messages sent.

References to the actors that are ready, i.e. those that have a message on their queue, are held on a ready queue. The scheduler waits on a condition variable while its queue is empty.

When the scheduler is notified that the ready queue is no longer empty it pops off the actor reference and dequeues its waiting message. These are then combined into an ActorFunction object and passed to the thread pool for execution.

The scheduler includes flags that allow the program to continue when there are no messages pending. This is useful for programs the use timers or which wait on client processes.

2.7.2 ThreadPool

The scheduler maintains a thread pool which has a list of up to THREADS_MAX threads.

The exec() method accepts an ActorFunction object and passes it to the next available thread. Additional threads may be created if we have not reached the maximum. This method will block until a thread becomes available.

The runner() method implements each thread. It waits until a function object is available and then executes it. The owner of the function object is notified when the function is about to start and when it concludes.

2.7.3 Printer

Getting messages out of programs is usually done with some sort of “print” statement. Unfortunately this is a problem in an actor based program as the messages from different actors can end up being interleaved in the output stream.

The library includes a Printer actor that is passed entire lines as messages by a helper Print class which provides the usual C++ stream operators.

2.8 Thread Classes

There are a few cases where a typical program would block while it waited for some event. The actor functions should be written so that they never block. To accommodate this problem the library introduces some thread based classes.

2.8.1 Clock

Since it is unwise to call functions like sleep() from within an actor's methods there is a clock object that can be scheduled to make a call back to the actor at a future time, or on a regular basis.

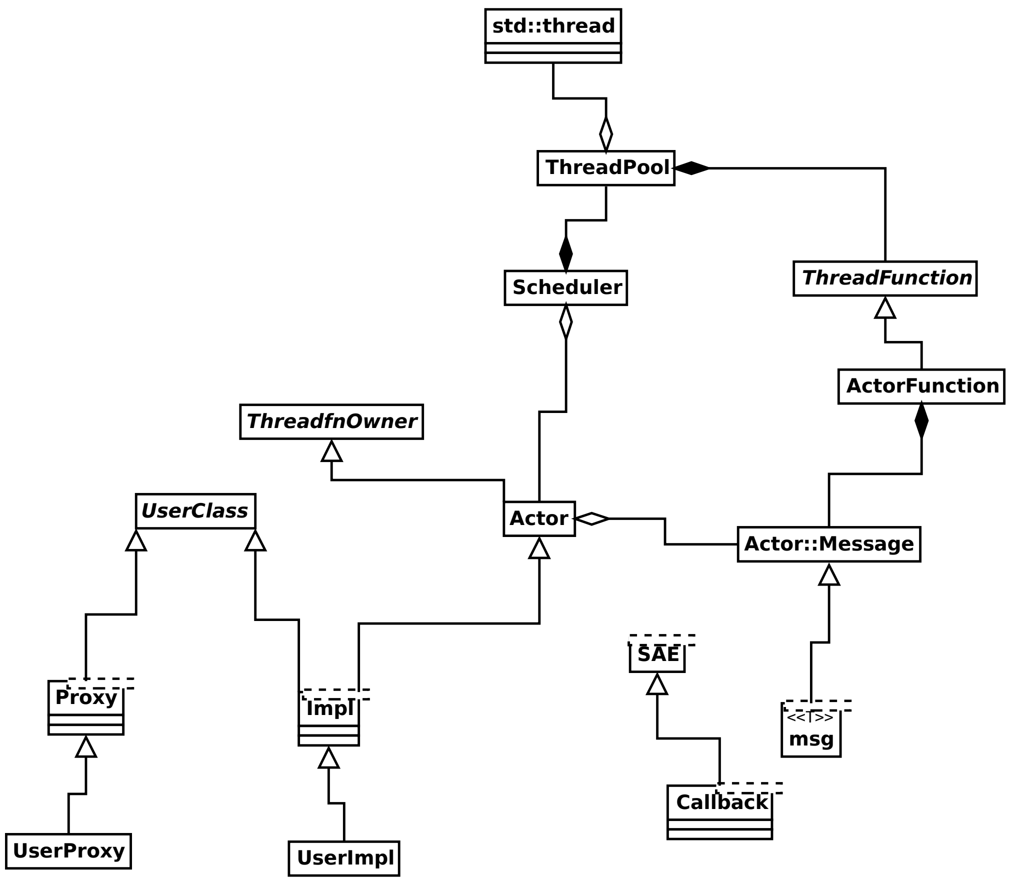
It includes a static method for converting a time point into a string.

2.8.2 Signaller

This object uses the actor's callback method to notify an actor of an operating system signal.

2.8.3 Selector

This allows an actor to be notified when there is activity on a file descriptor. It is used instead of the select() system call, or where a read or write might block.



3 Activities

This functionality is not currently part of SCAL

The problem that confronts one when trying to develop an application using the actor model is that there is a rather severe case of “inversion of control”. Each actor has to maintain its state in an explicit set of attributes and the program is reduced to a large set of event handlers, predicate functions and action functions.

In the more general case an actor might be having separate conversations with several other actor objects and needs to be able to keep the state of each conversation.

The model developed for this experiment was based on the Expert System pattern. The event handlers would make changes to the state variables and then trigger a rule engine. The rule engine would apply predicate functions stored in a table until one returned true. Each predicate function has an associated action function which can make further changes to the state and make calls to other objects.

3.1 Activity Classes

3.1.1 Activity

This is a base class that provides the basic operations for a collection of state variables. Each activity class will have a string defining its type so that the system can apply different sets of predicate functions according to the type of activity. Each activity object has a unique serial number so that messages can be delivered to the state variables for a specific conversation.

The derived classes need to implement an install method that sets up a table of predicates that is installed into the rule engine.

3.1.2 Predicates and PredicateTable

The Predicates class defines a common type for the PredicateTable template. The template parameter is the class derived from Activity.

The table contains pointers to functions that return true if state variables in the activity match its conditions. Each predicate function has a corresponding integer which is used to look up the action function to run.

The table is stored into the rule engine, indexed by the type of activity. It is therefore only necessary to install the predicates once for each type of activity, for each actor.

3.1.3 Rule Engine

The engine holds tables of predicate functions, indexed by the type of activity, and a table of action function pointers indexed by the integer associated with each predicate function.

When triggered the engine applies the predicates associated with the activity until one returns true. The corresponding action function is then executed.

3.1.4 Activity Manager

This is a base class that actor implementations can inherit in order to use the activity part of this library.

It includes an abstract method for creating activities. These are installed into the rule engine when the first of each type is constructed.

3.2 Notes and Issues

For someone used to the usual imperative programming style it is quite difficult to design a program using the actor model. The myriad of functions and state variables quickly overwhelms the mind.

One approach is to start with the action functions, then for each one work out what combination of state variables should trigger it. Then you can design the predicate functions. The event handlers are done last and modify the state variables.

Without careful design it is easy to create an event storm, or for the program to stall in a state that generates no actions, or for the actors to perform unexpected actions.

A tool or code generator would be a useful addition as it could verify that the above problems were unlikely. However, we need to be careful not to simply define a new language – the aim is to create actors in C++, not some other language.

3.2.1 Future Work

The next round of work on this project should aim to develop a more complex example program and to develop some tools for designing the activity functions.

Appendix A