Implementation of a Simple Multi-threaded Messaging System with an Asynchronous Mailbox and Associated Classes

Janna McLaughlin, Matt Szaro, Tim H.

CS 475, Spring 2012.
George Mason University.

Table of Contents

Introduction	*3*
Problem Definition	*4*
Program Design, Conceptual	*4*
Program Design, Technical	*5*
Test Application	*8*
References	*8*
Program Listings	*9*

Introduction

In the course of choosing a topic, it became apparent rather early on that building a multithreaded, distributed communication system would be especially interesting and relevant to our anticipated careers; for that reason, we began looking into development and applications of inter-thread/inter-process communication using message passing. Our team ultimately settled on a messaging system. We originally contemplated building something akin to an e-mail client and server, though through the course of discussions, we settled on a system that more resembles a chat server, somewhere between relay chat and instant messaging.

While a practical implementation would be distributed, and communicate over sockets, our research implementation is in fact entirely local. All communication happens between threads using asynchronous mailboxes. Aside from this, no deviations were taken from what we believed to be an appropriate distributed system. We believe that if one were to replace parts of code with sockets, rather than inter-thread communication, our application logic would still be valid.

Our system was designed as a Windows application, and makes use of the Win32 API. However, it could be ported somewhat easily to pthreads and POSIX. Our design and findings follow.

Definition of the Problem

Our goal was to implement a multithreaded, asynchronous messaging system. To that end, we required a server thread to dispatch client messages without running into synchronization issues. The defining challenges were message passing, and by extension of that, routing. How should one thread communicate with another through a server thread? How should the server know where to route inbound messages? We also encountered challenges in server-to-client communication - it is important that each client only be able to receive messages intended for it. These challenges were all overcome.

Program Design, Conceptual

Conceptually, we have the following program design: There is a single server thread, which all client threads will connect to. The server will have its own mailbox for receiving messages, which it then dispatches to the appropriate client thread. Client threads each have their own mailboxes, which they check for messages.

The server serves as the main point of contact for all client threads. Clients cannot send messages to other clients without passing the message through the server first. The server may log messages that pass through it.

Client messages contain information about origin and destination, a timestamp set

at the origin, and the message to be sent.

Program Design, Technical

Our implementation language was C++. We studied some Java code provided with *Modern Multithreading* as a reference, however our implementations vary, and in most cases are not "simply" ports.[1]

We implemented a sample program (test driver) as well as nine classes: Asynchronous Mailbox, Binary Semaphore, Client, ChatMessage, Counting Semaphore, Mutex, Server, Synchronous Mailbox (*Unused*), and Thread. The classes are described as follows:

Asynchronous Mailbox - Counting Semaphores for MessageAvailable and SlotAvailable, plus a Sender Mutex and Receiver Mutex. We used a rotating bounded buffer for message data, stored in an array. The mailbox sends and passes void-pointers, which clients cast to and from when working with the mailbox. (This functions in a similar way to Java implementations, which pass Objects around.[1]) The mailbox has a blocking receive operation, meaning that incoming messages must wait while another message is being received, a concept taken from *Modern Multithreading*.[1] The mailbox has a buffer-blocking send operation, meaning that multiple messages may be sent out from the mailbox as long as the buffer in the mailbox is not already full of message data.[1] Also

has a resend method for re-sending messages, if necessary.

Binary Semaphore - Simple wrapper around a Counting Semaphore of queue size one.

Client - A thread which contains a mailbox and a username. Registers its mailbox and username with the server upon being created. Sends messages to the server's mailbox, and checks for new messages in its own mailbox.

ChatMessage - A structure containing the origin client, destination client, a timestamp, and a message.

Counting Semaphore - Implementation of a counting semaphore using Win32 thread functions to block. May be initialized to any value equal to or greater than 0 to designate how many "permits" the semaphore currently holds.[1] Standard methods for counting semaphores, taken from examples outlined in *Modern Multithreading*, include methods P( ) and V( ).[1] Method P( ) takes from the number of available permits or waits, or suspends a thread, until a permit is available. Method V( ) releases a permit and wakes up a suspended thread. Implementation contains a thread queue to handle unblocking of threads when signalled.

Mutex - A wrapper around Binary Semaphore, though without ownership information.

Server - A single, central thread which contains its own mailbox and controls 1 mailbox

per client. Clients must register their mailbox and username with the server before they can pass messages. The server dispatches messages once they are received.

Synchronous Mailbox (*Unused*) - Counting Semaphores for received and sent, and mutexes to control receiving and sending. Operation of send and receive methods are synchronous, so a thread calling send( ) will block until another thread calls receive( ), or vice versa. The mailbox passes void pointers, which threads cast to and from when working with the mailbox. (This functions in a similar way to Java implementations, which pass Objects around. Examples of such may be found in *Modern Multithreading*.[1])

Thread - Thread controller with run, suspend, resume and kill functionality. The Run( ) method sets the thread state to "running," meaning the thread is executing code. The Suspend( ) method sets the thread state to "waiting", meaning the thread halts execution to free up the CPU. The Resume( ) method wakes up a suspended thread. The Kill( ) method sets the thread state to "stopped" so it can no longer execute code. This class calls appropriate Win32 API methods to handle threads at an OS level. Classes which are intended to be run as threads do not extend this class, as in Java. Instead, the Thread class accepts a function pointer from an appropriate method (a *run*, *main* or something of that nature). The function pointer must be set on a thread's *func* parameter before the thread is started via thread->Run( ). Functions must be of the following declaration: void threadFunc(void*).

Also included is our joke C++ Class **Unary Semaphore**, which was written for entertainment purposes while taking a break. Our implementation uses QuantumPV( ) to work on quantum computers, but may change when you go to call it or see the result. This is of course not used anywhere in the code, but was included with the program listing as we thought Dr. Carver might get a laugh out of it.

Test Application

The sample application is fairly simple and straightforward. It creates a server thread, then spawns a user-defined number of client threads. The client threads have a testing #define which causes their Run( ) method to send random strings to other clients at random. If this is in fact defined, the user can watch as the threads send messages to one another through the server. Clients will std::cout messages they are sending, as well as the messages they receive.

In more detail, the server thread has its own Asynchronous Mailbox, and storage for pointers to each client's Asynchronous Mailbox and username. The server checks its mailbox for client messages, and sends them to the appropriate destination client mailbox. Furthermore, each client thread also has its own Asynchronous Mailbox. Once a client is created, it registers its mailbox and username with the server. It then may send a message, or block until a message is received. There are also Sleep( ) calls in the test program to make std::out more readable, and also to simulate a real small message system (there wouldn't be constant message spam).

References

[1] Carver, Richard H., and Kuo-Chung Tai. *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*. Hoboken, NJ: Wiley, 2006. Print.