Politecnico di Milano - University of Illinois at Chicago
Networked Operating System exam
Prof. Piuri

A networked mailbox inter-process communication mechanism

Introductory notes

By Daniele Paolo Scarpazza

1 Foreword

In this paper I describe the features of the *mailboxd* distributed mailbox server implementation which I realized as an assignment for the "networked operating systems" exam.

2 Overall structure

The "distributed mailbox kit" is composed of:

- an executable-format **server**, whose name is `mailboxd`, compiled for the desired architecture and ready to run by the user;
- a C++ object-format **library**, whose name is `remotemailbox.a`, that the user should link inside its own projects;
- a C++ **header** file, whose name is `remotemailbox.h`, containing required definitions and exports, that the user should include inside its projects;
- two **example** programs, respectively called `reader` and `writer`, provided together with their C++ source code, whose purpose is to test the proper operation of the server and to show how to use the library;
- the full set of **source files** for both the executable server and the library.

3 Features

The are the features which have been implemented in the server and in the local objects:

3.1 Full-duplex two-way, asynchronous, buffered communication.

The number of messages that a process can send to another process's mailbox is not limited; moreover the sender can write an unlimited number of messages without going into blocking condition, even if the recipient did not read any of them yet. Furthermore, the recipient can in turn write an unlimited number of messages to the original sender or to another mailbox without the need to empty its incoming message queue before.

3.2 Guaranteed, acknowledged, connection-oriented message delivery

The message passing mechanism uses a *connection-oriented* TCP/IP data stream, in which data is guaranteed to be delivered without errors, without data loss, in the right order and without duplication. An additional mechanism of acknowledgement has been added to provide a feedback mechanism and signal errors. Since the communication is connection-oriented, it is sufficient for a process to disconnect from the server in order to have its mailbox automatically unregistered; thus instantly avoiding all the consistency problem that could arise with *UDP datagram*-based implementation.

3.3 Strong, robust, single-threaded high-performance architecture

There is no overhead due to a multi-process architecture, in which the multiple instances have to communicate with each other; there is also no overhead due to synchronization of shared resources between the threads of a multi-threading architecture: the architecture has a single process with a single execution thread, which nevertheless guarantees full concurrency by an extremely careful choice of the call parameter to the socket functions.

3.4 Full event logging and monitoring

The mailbox server contains an embedded facility to completely log what happens, in particular:

- whenever a new process registers a itself into the mailbox server, the server logs the event with its timestamp and the name of the new mailbox;
- whenever a registered process leaves the mailbox server, the server detects the closing of the connection (since the link is connection- oriented)
- whenever data is sent to a mailbox, the process detects the event, logs the timestamp, reports possible problems and the reult of the delivery;

3.5 Multicasting support

More than one process can register to own a mailbox with the same name. In this case all these processes can send data with the same identity, and every message that is send to their common mailbox is automatically replicated internally by the mailbox server for every owner. This replication is completely transparent for the senders, which have only to send messages to a single mailbox, with a single name, exactly as if it were owned by a single process.

3.6 Broadcasting support

It is possible to broadcast messages, that is to say, a process can send a message and have it delivered to all the registered mailboxes inside the server. This feature has been implemented to allow possible future higher-level protocols that could stack upon this service; this higher-level protocols could implement particular features such as some sort of "network directory service", host/server/program activity monitoring, and so on.

3.7 Unlimited number of mailbox

There is no limit in the number of mailboxes that can be instantiated into the server, save the available memory space, of course.

3.8 Smart memory allocation policy

Both mailboxes and their data queues are arranged into lineary-addressable vector structures, which guarantee a high-speed constant-time access cost; but since complete unboundedness in the number and size of data structure is desired, this vector lies in a dynamically allocated area, which can grow and be resized according to the current activity of the server. I have decided not to use a very high-level data structure (such as the STL container classes) and instead strictly cooperate, perhaps at a lower level with the operating system, via the *realloc(...)* function calls and the other flavours of *malloc*s. The only data structure which present the risk of being reallocated to a larger block when needs to grow is simply a little array of pointer, nevertheless, in order to avoid frequent reallocation of this data structure, a granularity block has been added in order to allocate more than one slot at a time, causing reallocation to be invoked only on 1 case out of n.

3.9 Immediate incoming message process

There is no way for a process to cause some blocking situation in the server or influence in some other way the communication with another mailbox or process. A process can, at any time, start the sending of a new message or read its incoming message queue, receiving immediately attention from the server independently from what the other processes are doing on their mailboxes, and this is achieved without using datagram-based communication.

3.10 Unlimited non-blocking outgoing queue

There is no way to go into blocking condition on a message send operation. This is because of the way the server has been internally implemented.

3.11 Unlimited number of mailboxes per process

Every process can transparently instantiate an unlimited number of objects from the *CRemoteMailbox* class, and connect them to the same server or to different servers, with no limitations. This way it is also possible to realize a sort of bridge between different remotemailbox servers.

3.12 Hardware independence

All exchanged data by the protocol itself is automatically serialized in order to avoid byte-order problems between big-endian and little-endian machines.

3.13 Graceful termination

If a server is killed without prior closing of its sockets, the sockets are automatically closed, but a new server cannot bind to the same socket and port for a security timeout, in order to sharply separate the old activities from the broken-down ones. This timeout period is, under Linux kernel 2.2.14, exactly 60 seconds.

In order to properly and gracefully close sockets from the server side without causing such block, a signal handler has been programmed and installed into the server.