Under the Hood of the MOOS Communications API

Paul Newman

March 17, 2009

Abstract

This document will give you all the gory details about the mechanisms and protocols lying at the heart of the MOOS Communications API. The casual or novice user need not worry about many of these details in the first instance.

1 Introduction

Much of the text that follows is taken from the original "MOOS document" found on the project website. I have carved this section out as I think it stands alone as a description of the low-level how and why of the MOOS communications architecture.

1.1 Topology

MOOS has a star-like topology. Each application within a MOOS community (a `MOOSApp`) has a connection to a single "MOOS Database" (called `MOOSDB`) that lies at the heart of the software suite. All communication happens via this central "server" application. The network has the following properties:

- No Peer-to-Peer communication.
- All communication between the client and server is instigated by the client. (i.e. the `MOOSDB` **never** makes a unsolicited attempt to contact a `MOOSApp` .)
- Each client has a unique name.
- A given client need have no knowledge of what other clients exist.
- A client has no way of transmitting data to a given client — it can only be sent to the `MOOSDB` .

- The network can be distributed over any number of machines running any combination of supported operating systems.

This centralized topology is obviously vulnerable to "bottle-necking" at the server, regardless of how well written the server is. However, the advantages of such a design are perhaps greater than its disadvantages. Firstly, the network remains simple regardless of the number of participating clients. The server has complete knowledge of all active connections and can take responsibility for the allocation of communication resources. The clients operate independently with inter-connections. This prevents rogue clients (badly written or hung) from directly interfering with other clients.

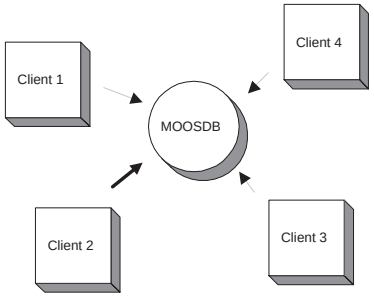

Figure 1: MOOS binds applications into a network with a star-shaped topology. Each client has a single communications channel to a server (`MOOSDB`).

2 MOOS Communities (and multiples of)

Post-2002 releases of MOOS have included explicit support for multiple communities. The idea is that it is sometimes advantageous to have groups of processes (bound together by mutual connection to a MOOSDB hub) communicating with each other. To this end, MOOSMsgs have been extended to include a "Source Community" field, which names the community from whence the message came.

Under the "one mission one mission file" paradigm all processes running in a community read from the same mission file (see the Programming with MOOS Document for more on this paradigm). The community name is specified by the line `Community = <Name>` at the top of the mission file. [1]

The question remains — how does this information get used? Well, when the MOOSDB serving each community starts up (see Section 6) the community name will be read and all messages sent from the DB will be tagged with this community name (see Section 3). However this tagging only happens if the data content originated from the `MOOSDB` 's own local community — if it did not (i.e. it came from another community) then the community name of the originating community is preserved. To understand how data from an external community could appear in one particular `MOOSDB` , see the document on `MOOSBridge` for more information.

[1] Perhaps near the `ServerPort=` and `ServerHost=` global definitions.

Variable	Meaning
Name	The name of the data
String Val	Data in string format
Double Val	Numeric double float data
Source	Name of client that sent this data to the MOOSDB
Time	Time at which the data was written
Data Type	Type of data (STRING or DOUBLE)
Message Type	Type of Message (usually NOTIFICATION)
Source Community	The community to which the source process belongs — see Section 2

Table 1: Contents of MOOS Message

3 Message Content

The communications API in MOOS allows data to be transmitted between MOOSDB and a client. The meaning of that data is dependent on the role of the client. However the form of that data is constrained by MOOS. Somewhat unusually, MOOS only allows for data to be sent in string or double form. Data is packed into messages (CMOOSMsg class) which contain other salient information, as shown in Table 1. The fact that data is commonly sent in string format is often seen as a strange and inefficient aspect of MOOS. For example, the string Type=EST,Name=AUV,Pos=[3x1]{3.4,6.3,-0.23} might describe the position estimate of a vehicle called "AUV" as a 3x1 column vector[2]. It is true that using custom binary data formats does decrease the number of bytes sent. However binary data is unreadable to humans and requires structure declarations to decode it, and header file dependencies are to be avoided where possible. The communications efficiency argument is not as compelling as one may initially think. The CPU cost invoked in sending a TCP/IP packet is largely independent of size, up to about 1000 bytes. So it is as costly to send two bytes as it is 1000. In this light there is basically no penalty in using strings. There is however a additional cost incurred in parsing string data which is far in excess of that incurred when simply casting binary data. Irrespective of this, experience has shown that the benefits of using strings far outweighs the difficulties. In particular:

- Strings are human readable — debugging is trivial, especially using a tool like MOOSScope. (See the document on Graphical Tools for more on MOOSScope (uMS) .)
- All data becomes the same type.
- Logging files are human readable (they can be compressed for storage).
- Replaying a log file is simply a case of reading strings from a file and "throwing" them back at the MOOSDB in time order.
- The contents and internal order of strings transmitted by an application can be changed without the need to recompile consumers (subscribers to

[2]Typically string data in MOOS is a concatenation of comma-separated "name = value" pairs.

that data) – users simply would not understand new data fields but they would not crash.

Of course, scalar data need not be transmitted in string format – for example the depth of a sub-sea vehicle. In this case the data would be sent while setting the data type to `MOOS_DOUBLE` and writing the numeric value in the double data field of the message.

4 Threading and Process Models

The choice of processes over threads was made on two counts, firstly that of stability – a rogue process cannot corrupt the program/data space of another process (in a sane OS that is). Secondly, on the basis of swift and pain-free development by several programmers with diverse backgrounds. Building a single monolithic executable by several people requires, at a minimum, adherence to programming guidelines and styles that may not be native to all those included – especially in an academic environment. The use of small-footprint, independent processes implies that developers can use whatever means they see fit to accomplish the job. Linking with the communications library integrates them seamlessly with all other processes but denies a process the means of interfering with others.

5 Communications API Mechanics

Each client has a connection to the DB. This connection is made on the client side by instantiating a class provided in the core `MOOSLIB` library called `CMOOSCommClient` . This class manages a private thread that coordinates the communication with the `MOOSDB` . The `CMOOSCommClient` object completely hides the intricacies and timings of the communications from the rest of the application and provides a small, well defined set of methods to handle data transfer. Using the `CMOOSCommClient` each application can:

1. Publish data – issue a notification on named data.
2. Register for notifications on named data.
3. Collect notifications on named data.

5.1 Publishing Data – notification

Assume that as a result of some computation or input, a process `A` has a result that is likely to be of use to other undisclosed (remember MOOS applications do not know about each other) processes `B` and `C` – for example a position estimate. The simplest way in which `A` can transmit its new data is to simply call the `Notify` method on its local `CMOOSCommClient` object specifying the data, its name and the time at which it is valid. Behind the scenes this method then creates a suitable `CMOOSMsg` filling in the relevant fields. The action of publishing data in this way should be viewed as a *notification* on a named data variable. Crucially this does not imply a change in the *value* – the outcome of the computations resulting in the need to publish data may remain

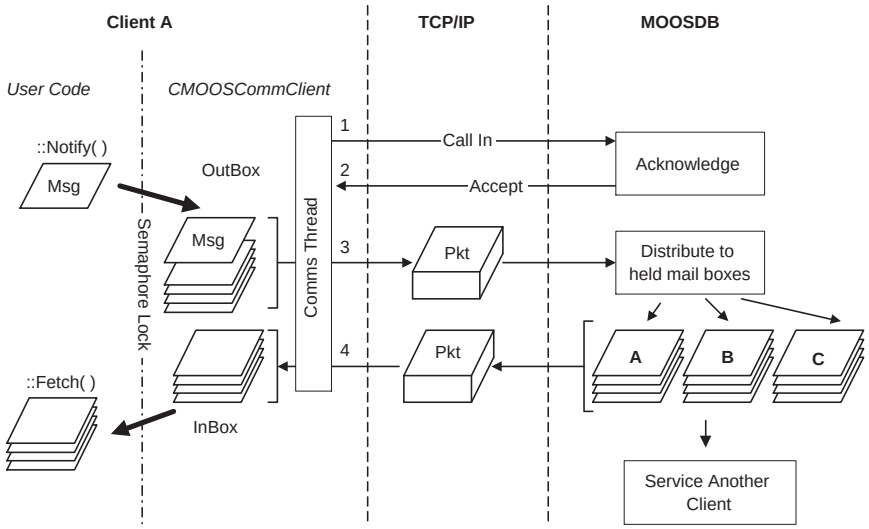

Figure 2: The mechanics of the client server interaction in MOOS. The user code calls the `Notify` method to transmit data. This method simply adds a message to the "outbox". Some time later (1) the communications thread calls into the database. When the database is not busy it accepts the client's call (2). The client then packs the entire outbox into a single large transmission which is sent to and read by the server (3). The server unpacks the packet into its constituent messages and places copies (according to subscriptions and timing) in the mailboxes of other connected clients. The server then compresses the mailbox of the current client into a packet and sends it back to the client (4). At this point the transaction is then complete and the server terminates the conversation and looks to begin the same process with a different client. Upon receiving the reply packet, the client communications thread unpacks it and places the resulting messages in the "inbox" of the client. The user code can retrieve this list of messages at *any time* by calling the `Fetch` method.

unchanged. For example, two sequential estimates of location of a vehicle may remain numerically the same but the *time* at which they are valid changes – this constitutes a data notification.

As far as the application-specific code in `A` is concerned, the invocation of the `Notify` method results in the data being sent to the `MOOSDB` . What is really happening, however, is that the `CMOOSCommClient` object is placing the data in an "outbox" of `CMOOSMsg` s that need to be sent to the `MOOSDB` at the next available opportunity. The next obvious question is "when does the data reach the `MOOSDB` ?". The `CMOOSCommClient` object thread has a "Comms Tick" – essentially a timer that contacts the `MOOSDB` at a configurable rate (typically 10Hz but up to 50Hz). All the messages in the outbox are packed into a single packet or "super message" called a `CMOOSPkt` . Eventually the `MOOSDB` will accept the incoming call from the client `A` and receive the packet. At this point the data flow is reversed and the `MOOSDB` replies with another `CMOOSPkt` containing notifications (issued by other processes like `B` and `C`) that are relevant to `A` . How a process declares what constitutes a relevant

notification is discussed in Section 5.2. If at the time the thread calls into the MOOSDB the outbox is empty (i.e. there is nothing to notify) a NULL message is created and sent. Similarly the MOOSDB may reply with a CMOOSPkt containing only a NULL message if nothing of interest to the client has happened since its last call in. This policy preserves the strict symmetry of "one packet sent, one packet received" for all occasions.

5.2 Registration

Assume that a list of names of data published has been provided by the author of a particular MOOS application. For example, an application that interfaces to a GPS sensor may publish data called GPS_X and GPS_Y . A different application may register its interest in this data by *subscribing* or *registering* for it. An application can register for notifications using a single method Register specifying both the name of the data and the maximum rate at which the client would like to be informed that the data has been changed. The latter parameter is specified in terms of the minimum possible time between notifications for a named variable. For example, setting it to zero would result in the client receiving in the CMOOSPkt (that is the MOOSDB 's reply) a collection of messages describing each and every change notification issued on that variable.

5.3 Collecting Notifications

At any time the owner of a CMOOSCommClient object can enquire whether it has received any new notifications from the MOOSDB by invoking the Fetch method. The function fills in a list of notification CMOOSMsg s describing what has changed, what client made the change, when it was done and what the data type is (see Table 1 in Section 3). Note that a single call to Fetch may result in the presentation of several notifications corresponding to the same named data. This implies that several changes were made to the data since the last client-server conversation. However, the time difference between these similar messages will never be less than that specified in the Register function described in Section 5.2.

6 The MOOSDB

The MOOSDB is the heart of the system. It serves as the hub through which all communication occurs. It is tempting to think of the MOOSDB as simply a blackboard – an entity which stores the current state (represented by values of named variables) of the system. Typically, a blackboard allows clients shared access to a centralized repository of information. Although the MOOSDB does maintain the most recently set value of all variables and in that way is similar to a blackboard, the way in which data is retrieved is very different. The MOOSDB records, on behalf of its connected clients, the *history* of changes to data. Assume a client A has subscribed to variables called p and q with a minimum notification period of τ_p and τ_q seconds respectively. The time is now t and the last call-in was at time t',$\Delta = t - t'$ seconds ago. When A calls in, it is not simply presented with the most recent values of p and q but rather all the *changes* that have occurred to them between t' and t'. Imagine

that some other process `B` is publishing changes (value and/or time-stamp) to p and q every τ_B seconds. The call-in will result in n_p and n_q notification messages being returned to the client where

$$n_p = \begin{cases} floor(\frac{\Delta}{\tau_p}) & \tau_p \geq \tau_B, \\ floor(\frac{\Delta}{\tau_B}) & \text{otherwise.} \end{cases}$$

$$n_q = \begin{cases} floor(\frac{\Delta}{\tau_q}) & \tau_q \geq \tau_B, \\ floor(\frac{\Delta}{\tau_B}) & \text{otherwise.} \end{cases}$$

If no clients have issued relevant notifications since t' then there will be no notification messages stored at the `MOOSDB` for collection by the client. The only exception to this rule is the first time a client calls in after registering for a notification on a variable. In this case the value of the variable is returned in a notification message with a time-stamp specifying when the data was last set.