A MOOS-V10 Tutorial

Paul Newman, University of Oxford

July 21, 2013

....ten years on

Contents

1 How To Use This Tutorial? 3
1.1 What will I learn? . . . 3

2 Getting Started - Acquiring and Building MOOS 3
2.1 Before you start you will need . . . 3
2.2 Downloading and Building . . . 3
2.2.1 A note for exisiting `MOOS` Users . . . 4
2.3 Header, Source and Library Structure . . . 5
2.3.1 Header structure . . . 5
2.4 Importing and Building Against MOOS-V10 . . . 6
2.4.1 How is MOOS found? . . . 6
2.4.2 Trouble Shooting . . . 7

3 Basic MOOS Communications Concepts 7
3.1 The `MOOSDB` . . . 7
3.2 Data Types and `CMOOSMsg` . . . 7
3.2.1 How do I know what the payload of a CMOOSMsg is? . . . 8
3.3 Using the Comms Client Classes - `CMOOSCommClient` and `MOOS::AsyncCommClient` . . . 8
3.3.1 Sending Data with Notify . . . 9
3.3.2 Grabbing Mail with Fetch . . . 9
3.3.3 Configuring Connection Notification with `SetOnConnectCallBack` . . . 9
3.3.4 Configuring Mail Delivery with `Register` . . . 10
3.3.5 Wildcard Subscriptions with `Register` . . . 10
3.3.6 Starting communications with `Run` . . . 10
3.4 Working with Receive Callbacks . . . 11
3.4.1 Configuring Notfications with `SetOnMailCallBack` . . . 11
3.4.2 Adding Active Message Queues with `AddMessageCallback` . . . 11
3.4.3 The Wildcard Queue . . . 12
3.5 Application Writing with `CMOOSApp` . . . 13

4 Programming with MOOS Clients - Examples 13
4.1 Index of Example Codes . . . 13
4.2 The Hello World example - polling (Ex10). . . . 14
4.2.1 What is bad about this polling design? . . . 15
4.3 Installing and Using a Mail callback (ex20) . . . 15
4.3.1 An aside - Using a command line parser (ex30) . . . 16
4.3.2 What is bad about this responsive design? . . . 17
4.4 Adding Active Message Queues (ex40) . . . 18
4.4.1 Adding a Default Active Message Queue (ex50) . . . 19
4.5 Wildcard Registrations (ex60) . . . 21

5 Writing Applications with CMOOSApp 23

6 Configuring `MOOSDB` 24
6.1 Command Line Help . . . 24
6.2 Configuring Client Response Times . . . 24
6.3 Specifying When Clients are Assumed Dead . . . 25
6.4 Live Network Audit . . . 25

7 Further Examples 26
7.1 Sharing Video Rate Data . . . 26

1 How To Use This Tutorial?

I'm a complete newbie

You should start at section 1.1 the tutorial will take you through downloading, building, understanding and learning to program with the MOOS Library.

I have MOOS installed already - now what?

Maybe you have installed something like MOOS-IvP which packages MOOS for you or maybe you followed installation instructions on a website. In that case you can go straight to Section 2.3

I like to see example code straight away

Many folk find it easiest and most comforting to look directly an example source code to get a feel for what is going on. If this sounds like you go to Section 4

1.1 What will I learn?

This document is intended to help you get started in using the `MOOS` communications and application building API. It will take you through, in simple steps, the process of downloading, building and developing with the MOOS library. This will allow you to easily generate programs which can share data using the `MOOS` communication tools. These tools are all housed in a single, standalone, dependency-free project called `core-moos` so really this is a tutorial about core `MOOS` competencies and `core-moos` all in one.

2 Getting Started - Acquiring and Building MOOS

2.1 Before you start you will need

- a working compiler like `gcc` or `clang`
- `CMake` installed
- `git` installed (well actually this is optional as you can download the source code as .zip file and we won't make much use of git in this tutorial)

2.2 Downloading and Building

We shall begin where we should and check out a version of MOOS-V10 from a git repos. We will follow good practice and do an out of place build - the source code will go in "src" and we will build in "build". We will also, after fetching the source switch to the "devel" branch because here we are living on the edge [1].

```
pmn@mac:~$ mkdir core-moos-v10
pmn@mac:~$ cd core-moos-v10
pmn@mac:~$ git clone https://github.com/themoos/core-moos.git src
pmn@mac:~$ cd src
pmn@mac:~$ git checkout devel
pmn@mac:~$ cd ..
pmn@mac:~$ mkdir build
pmn@mac:~$ ccmake ../src
```

[1] if you want to know what branches are available type `git branch`

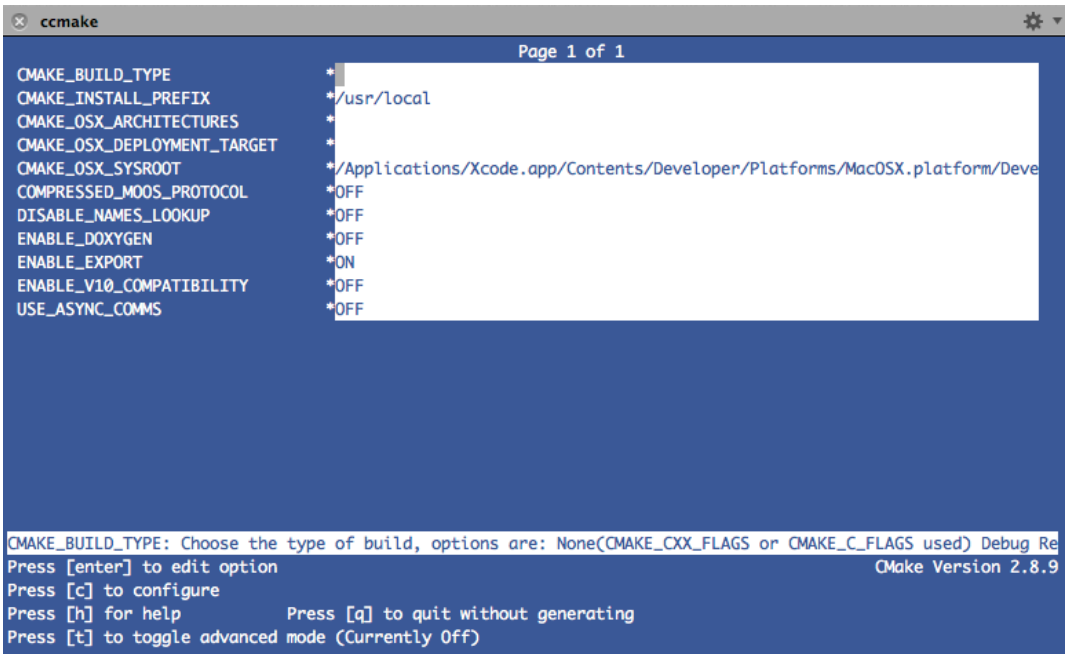

Figure 1: The default build screen for MOOS V10. Note that by default USE_ASYNC_COMMS is off. If you want really fast communications you should enable this.

At this point you should, after hitting 'c' a couple of times be presented with a CMake screen that looks like that shown in Figure 1 (note some of the entries are platform dependent so don't worry if what you see is not identical to this).

You are are now in a position to build the MOOS. So press 'c' until 'g' appears, then press 'g' and you are good to go. Then at the terminal prompt type 'make' to build the project. Two directories should have been created **bin** and **lib.** In lib you will see `libMOOS.a` and in `bin` you will find the newly created `MOOSDB`. If you run up the MOOSDB (by typing ./MOOSDB you should see output similar to that in Figure 2. You are now all set to begin developing with MOOS. Nice job.

2.2.1 A note for exisiting `MOOS` Users

Skip this section if you are new to MOOS. If you are already an old hand at MOOS and simply want to link your existing code against MOOS V10 without needing to worry about the new header, rationalised file structure introduced in MOOS V10 then you will need to turn on `ENABLE_V10_COMPATIBILITY`. This switch adds an additional set of include path to those exported by the project, which have the same structure as those present in previous (now legacy) versions of MOOS. If you "include" one of these files they actually simply redirect to include header files residing in the new structure. But be advised that this is not a happy long term policy - you should think, if possible, about updating your code - but there is much to be said for not *having* to change your code simply to use V10. Hence the introduction of this switch.

Tip: Turn on `ENABLE_V10_COMPATIBILITY` to make V10 appear to have the header structure of earlier versions. This allows you to use V10 without needing to change any of your source code

Tip: you can use the V10 `MOOSDB` with old MOOS applications - you don't *have* to recompile them. V10 is backwards compatible.

2.3 Header, Source and Library Structure

The classes that implement the communications and application management (for example `CMOOSApp`) reside in a single library called `libMOOS`. There are in fact four key subdirectories in `libMOOS`. In figure **??**you can see the basic structure of the code base.

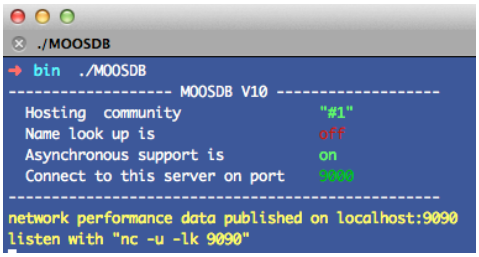

Figure 2: running the MOOSDB

App contains the classes like CMOOSApp and CMOOSInstrument - you use thse to make application writing very easy

Comms contains everything to do with MOOS IPC communications

Utils contains everything that used to be in MOOSGenLib (with some nice additions)

Thirdparty contains small lumps of thirdparty code which is being leveraged in V10 (all licenses included)

include contains some high level include directories that make using libMOOS easy (and backwards compatible)

The directory called `MOOSDB` contains the source-code of the `MOOSDB` and has a subdirectory containing various small testing programs. The `MOOSDB` program has a dependency on core-moos but nothing else. The only other directory of interest is `tools which is home to 'umm'` the swiss army knife of MOOS.

2.3.1 Header structure

It is important to understand where the header files are found in the file structure of the MOOS project - they typically do not live along side the corresponding .cpp files. Take for example CMOOSApp.cpp which lives at Core/libMOOS/Apps/CMOOSApp.cpp - the actual location of CMOOSApp.h is libMOOS/Apps/include/-MOOS/libMOOS/Apps/CMOOSApp.h. This may seem convoluted but it eases many things when it comes to developing in various IDE's and a constant way to reference headerfiles in during development and when installed. In this case CMOOSApp.h is included by writing #include "MOOS/libMOOS/Apps/CMOOSApp.h" whether or not the headers are installed or whether or not you are tinkering with MOOS source itself. So it helps to have a rule. If the source file is in libMOOS/X/file.cpp then the header is included as #include "MOOS/libMOOS/X/file.h" - simple.

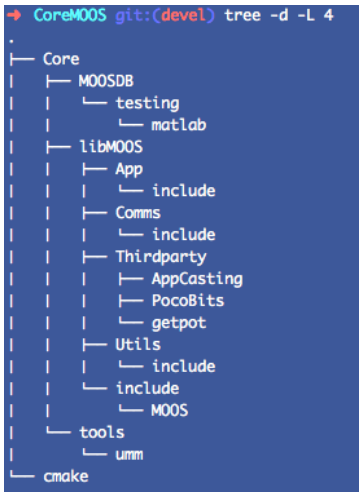

Figure 3: Top-level directory structure for MOOS V10

2.4 Importing and Building Against MOOS-V10

So now you have built the new MOOS. Next questions is "how do you link against it". If you use `CMake` then this is trivial you just need to insert the line `find_package(MOOS 10)` in your `CMakeList.txt` script. This goes and finds the latest build you made of MOOS V10 (and only V10) and collects the correct include paths, library names and library paths and puts them in the following `CMake` variables:

`MOOS_INCLUDE_DIRS` This contains the list of include directories you need to include to find `MOOS V10` header files.

`MOOS_DEPEND_INCLUDE_DIRS` This contains the list of include directories which MOOS needs to find teh headers it depends on (should be empty)

`MOOS_LIBRARIES` This contains the precise library name (absolute path) for `libMOOS`

`MOOS_DEPEND_LIBRARIES` This contains the absolute paths for the libraries MOOS depends on (should be empty)

These variables can be used to import all you need to know about MOOS into an external project. You can see how to do this in some the example `CMakeLists.txt` file given below. Here we make an executable called `example_moos` , explicitly search for MOOS-V10, set up include paths, set up an executable and finally indicate how to link.

```
#this builds some code using MOOS
set(EXECNAME example_moos)

#find MOOS version 10 be explicit about version 10 so we don't
#find another old version
find_package(MOOS 10)

#what source files are needed to make this exectutable?
set(SRCS  example_moos.cpp)

#where should one look to find headers?
include_directories( ${MOOS_INCLUDE_DIRS} ${MOOS_DEPEND_INCLUDE_DIRS})

#state we wish to make a computer program
add_executable(${EXECNAME} ${SRCS} )

#and state what libraries said program needs to link against
target_link_libraries(${EXECNAME} ${MOOS_LIBRARIES} ${MOOS_DEPEND_LIBRARIES})
```

2.4.1 How is MOOS found?

You have probably noticed that you do not need to install MOOS V10 for `find_package(MOOS V10)` to work. CMake simply appears to automagically find the latest build directory. It is worth understanding how this is done. CMake provides support for `find_package` by writing at build time to a file in ~/.cmake/modules. In this case because we are talking about `MOOS` there is a file in ~/.cmake/modules/MOOS (who's name is a whole load of crazy letters) inside of which is the location to a file called `MOOSConfig.cmake`. This file is created in the build directory when MOOS is configured. The find_package directive imports `MOOSConfig.cmake` (and from there `UseMOOS.cmake`) and this tells the importing CMake instance how to use MOOS.

2.4.2 Trouble Shooting

All the above should go smoothly but there have been instances reported in which things go wrong - this is always due to previous installations of MOOS and old configuration files hanging around. Executing the following steps should help if you get into trouble

- clean down the `MOOS-V10` project (why not remove the whole build directory?)
- remove all contents of `~/.cmake/modules/MOOS`
- remove any old copies of MOOSConfig.cmake you may have hanging around in you build tree. Note that once upon a time, long ago there was a `MOOSConfig.cmake` file checked into the source tree of MOOS-IvP. This can cause all kinds of trouble.....
- If header files are not being found by your project:
  - if your code previously worked with older versions of MOOS did you change your source code to reflect the new locations of headers? Or, if you really don't want to change you code, did you enable `V10_COMPATIBILITY` when you built MOOS-V10?

3 Basic MOOS Communications Concepts

Before we start writing some code, we need to cover some basic concepts. However If you are already a MOOS user you can skip to the next section - similarly if you like to look at working example code to learn new software libraries then you should jump (temporally at least) to Section 4.

3.1 The `MOOSDB`

This is a program which coordinates all the communications beween any and all programs using the MOOS communication facility. You typically run MOODSB[2] from the command line. Having started it you can safely leave it running for ever - you don't need to interact with it in any way. Its not a bad idea to set it up as a daemon. The MOOSDB does have some command line switches and you can read about them in Section 6 - but for now simply running `./MOOSDB` will start it running with a very useable set of defaults.

You should think of the `MOOSDB` as a program containing a list of named variables which, in concert, represent the state of your system. As a user of MOOS your applications can push data to the `MOOSDB` and have data sent to them in response to some other application pushing data. You can request to be told about every push or limit it to no more than once every τ seconds where τ is a value or your choosing.

3.2 Data Types and `CMOOSMsg`

The data which MOOS sends between processes is wrapped in a `CMOOSMsg.` You will ultimately, pehaps behind the scenes in an API call, package your data, be that string, double or a chunk of binary data, in a CMOOSMsg. Sometimes we refer to the delivery or transmission of one or more CMOOSMsg as getting or sending "Mail". Maybe not the best noun to have choosed with hindsight as in the UK at least in real life mail often gets lost and is often late. Luckily the opposite is true in MOOS.

You should think of a CMOOSMsg as a communique about a named lump of data. This data could be a double floating point value, a string or a binary chunk - it all depends on client who perfomed the first push of this named data to the MOOSDB - after that its type is set in stone.

[2] I wish I had not called it MOOSDB - of the DB because that brings with it a whole load of connotations of heavyweight databases. But this is a case of horse stable and bolted.

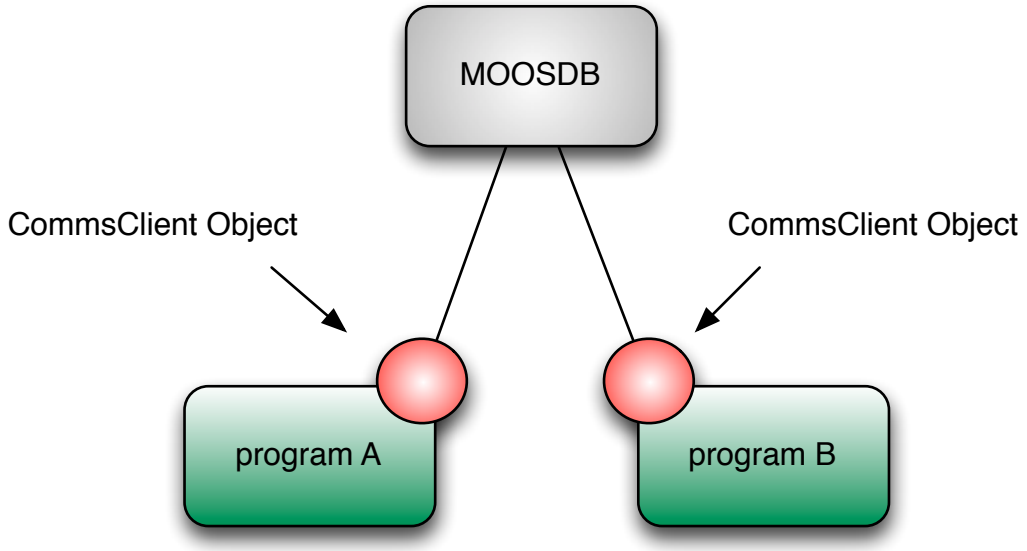

Figure 4: the simplest of MOOS communities - a DB and two programs which communicate with each other (share data). The red circles represent an instance of a CommsClient object. Note how the DB acts as a communications hub. We often refer to program A and program B as "clients". There is no restriction of the number of clients a community can have and they can live on as many different computers as there are clients.

3.2.1 How do I know what the payload of a CMOOSMsg is?

Good question. If you are processing a CMOOSMsg in your code it is because you have requested to be informed when that data has been updated (you do this by calling ::register from your code - see Section 3.3.4) . So this means you must have had a conversation with the author of the program that is doing the pushing (maybe even in your own head) so you are likely to know for example that a variable called "LeftImage" is a binary lump, or "battery_percentage" is a double. However if you are not sure you can use the methods `IsDouble() IsString() IsBinary()`.

3.3 Using the Comms Client Classes - `CMOOSCommClient` and `MOOS::AsyncCommClient`

The term comms client is used to refer to an c++ object which you as a developer can use to send and receive data via the MOOSDB. The object handles all of the details of managing the connection to the DB all you have to do is push data into it and using one of more of the API's get one or more (always in a std::list) of CMOOSMsgs out of it. There are some key methods offered by the comms clients which you need to know about and these will be covered in upcoming sections. But before we do that you should know that there are two kinds of comms clients - one old one new:

MOOS::AsyncCommClient This is the one you should use and was introduced in MOOS-V10 in 2013. It offers the fastest (lowest latency) way of getting data between applications. It manages two queues - one for outgoing messages and one for incomming messages and they run independently. Of course you as a user don't get to see this. As far as you are concerned a comms client is a portal into which you pour outgoing messages and receive them from

CMOOSCommClient This is the orginal client written in 2003 when MOOS was in its infancy. You can use it of course and it is after all the base class of MOOS::AsyncCommClient but if you do, you will be missing out on many good things. This client has a single thread managing communications in the background - input is coupled to output.

The following few sub sections will introduce you to small set of methods (functions) which you will need to know about to use MOOS. After that we'll bring them all together in some simple examples. The thinking is its a good idea to get the right nouns installed before getting going. Of course if you prefer you can jump straight to the examples in Section 4

Basic Operations

Again, you might at this point want to jump ahead for some complete example code - if so go to Section 4. What comes next in this section is a highlevel introduction to some methods which goven key methods and competencies of MOOS

3.3.1 Sending Data with Notify

use this method and its overides to send either double, std::string or binary data of any size. The overloaded versions

```
bool Notify(const std::string & sVarName, const std::string & sVal, double ↩
    dfTime=-1)
```

```
bool Notify(const std::string & sVarName, double dfVal, double dfTime=-1)
```

```
bool Notify(const std::string & sVarName, const std::vector<unsigned char>& ↩
    vData,double dfTime=-1)
```

which send a string a double and a vector of bytes (use this for binary data) respectively under the variable name `sVarName`

3.3.2 Grabbing Mail with Fetch

Use Fetch() to retrieve mail being held by your comms client ready for you to read. Note this does not go and fetch data from the MOOSDB - it simply returns to you what has already been collected but is currerntly being held for you by the worker threads in the client. Typically people use fetch if they want to poll the comms client to see if there is any fresh communications - they might for example out it in a while(1) loop continually looking for mail and processing it if and when it arrives.

```
//where M is a std::list<CMOOSMsg> typedefed to be a MOOS_MSGLIST
bool Fetch(MOOS_MSGLIST & M);
```

3.3.3 Configuring Connection Notification with `SetOnConnectCallBack`

```
    void SetOnConnectCallBack(bool (*pfn)(void * pParamCaller),
void * pCallerParam);
```

3.3.4 Configuring Mail Delivery with `Register`

The `Register` method is used to state what data you want to receive. If you register for "X" and some client posts "X" to the the `MOOSDB` then you will receive a "X" in your mail. The `dfInterval` parameter allows you to specify how often you wish to be told about changes to the variable in question. For example Register("X",2.0) means "tell me about X but only at a maximum rate of twice a second" so even if somebody is writing "X" at 500Hz you won;t be flooded by it. The special case of `dfInterval=0` means "tell me about every change". In other words, if you register for a variable with a zero interval every time any client writes to the DB with that variable you will receive a corresponding message.

```
bool Register(const std::string & sVar,double dfInterval=0);

bool Register(const std::string & sVarPattern,
              const std::string & sAppPattern,
              double dfInterval);
```

3.3.5 Wildcard Subscriptions with `Register`

MOOS-V10 offers a great deal of flexibilty in which clients can subscribe for data by allowing so called "wildcard subscriptions". This is the second version of the function list above. A client can register its interest in variable whose name and source (the name of the client that send it) matches a simple regex pattern. Only patterns containing * and ? wildcards are supported with their usual meanings i.e. '?' means any single character and '*' means any number of characters. An example will make this whole thing clear and we will be using the `Register( sVarPattern, sAppPattern,dfInterval)` interface. Imagine we have a Comm Client object called `CommsObject` - here are some ways we could configure some fancy wildcard subsriptions:

```
//register for all variables ending with "image"
//from any process with an name beginning with "camera_"
CommsObject.Register("*image","camera_*", 0.0);

//register for every single variable coming from a process
//called "system_control"
CommsObject.Register("*","sytem_control",0.0);

//register for any variable beginning with "error_" and
//produced by a process with a nine letter name beginning
//with "process_0" but please, only tell us at most twice
//a second
CommsObject.Register("error_*","process_0?", 2.0);
return true;
```

The logic which supports this new functionality is implemented at the MOOSDB and turns out to be a pretty useful and compact way to define some fine granularity on what data is received. Of course it can also be used to achieve blunderbuss subscriptions by subscribing to all variables from a given process - `Register(''*'',ProcessName)` - or even all variables from all processes - `Register(''*'',''*'')` the ultimate wildcard.

3.3.6 Starting communications with `Run`

Before you can use a comms client to send and receive mail you need to start its threads and this is done with the `::Run()` method. You need to tell the client the name or ip address of the machine

which is running the MOOSDB and the port on which it is running (this is often port 9000). You also need to give you client an name - this is the name with which this node will appear in the community so its a good idea to give it something semantically relevent. It also needs to be unique in the community - this is important. The final parameter is only revelant to old re MOOS-V10 clients (ie not MOOS::AsyncCommsClients) and it specifies how many times each second the client will talk to the MOOSDB.

```
bool Run(  const std::string & sServer,
           int Port,
           const std::string & sMyName,
           unsigned int nFundamentalFrequency=5);
```

3.4 Working with Receive Callbacks

Section 3.3.2 explained the simplest way to read mail - you rely on the comms client to hold a deep and meaningful conversation with the MOOSDB and when you are ready you simply pick up all as yet un processed messages (which we call "mail") from with a call to "Fetch". From the user's perspective this is a form of polling[3] and we are well served by considering a more responsive paradigm. The next few subsubsections will introduce some of the mecahnisms available to process mail as soon as it comes in with very very low latency.

3.4.1 Configuring Notfications with SetOnMailCallBack

```
void SetOnMailCallBack(bool (*pfn)(void * pParamCaller),
    void * pCallerParam);
```

Use SetOnMailCallBack when the simple polling method does not quite fit with your needs and you want really rapid response communications. This function allows you to install a callback which is called as soon as mail arrives. It is important to note that it is called from a thread in the comms client who's raison-d'etre is to manage communications with the DB and not run user code. The expectation is you would call Fetch from inside this function to actually retrieve the mail. The implication here is that you need to be careful about what you run in the callback - if you do a lot of work it will obviously impact the comms as no new mail will be processed or read from the DB until your work is done. At least if you are using a AsyncCommClient writing to the MOOSDB will not be affected because at least they have seperate threads for reading and writing. There is of course a better way to proceed and that is by using the AddMessageCallback API described in Section 3.4.2

3.4.2 Adding Active Message Queues with AddMessageCallback

It is easy to think of situations in which you want some sophistication and flexibiliy in the way you process mail. We've already mentioned in Section 3.4.1 that there are risks in processing all mail in a single callback. Imagine you had a need to always process message "X" as quickly as possible but also process message "Y". Difficulties arrive if "Y" takes a while to process - while thats happening we have no way of getting "X" which is next in the pipeline. A solution to this is provided by the AddMessageCallback method. This allows you to define a per-message callback and what is more each callback is run fom its own thread. You can also install as many threaded callbacks per message as you wish. For example imagine you were receiving images from a camera published under "Image" and there are 8 different inage processing tasks you wish to run on those images in which case you would install 8 different callbacks. When you install a callback with AddMessageCallback you get

[3]even though in V10 behind the scenes the comms client is not polling -it responds to data availability very quickly

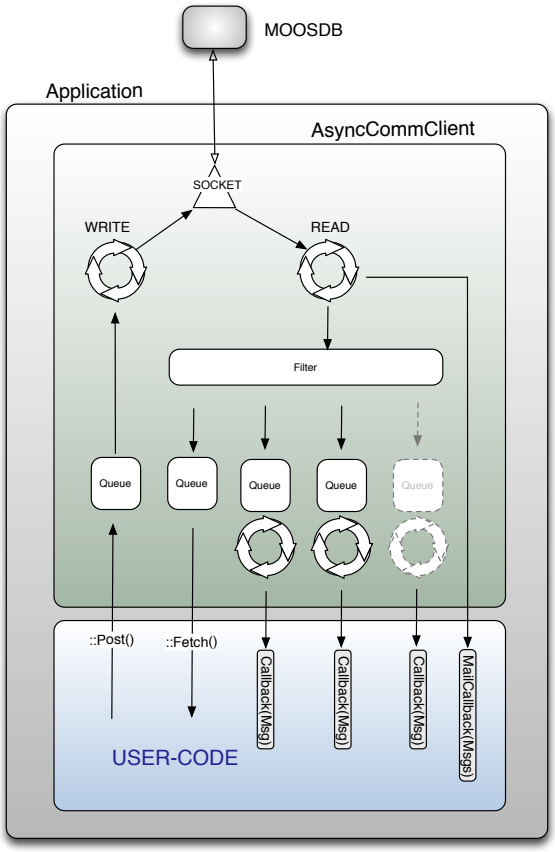

Figure 5: The threading model of the AsyncCommClient. Note that send and receive are decoupled and that the receive side supports multiple active queues each of which can invoke a specialised callback. Alternatively you can simply call Fetch() from time to time and get all unread mail. Then again you could install a single mail callback to handle all mail. You have several options.

to specifiy a nickname for the callback (which in this instance is synonymous with what we call an ActiveQueue) and a parameter of your choosing which you wish to be handed along with the `MOOSMsg` in the callback.

```
bool AddMessageCallback(const std::string & sCallbackName,
        const std::string & sMsgName,
        bool (*pfn)(CMOOSMsg &M, void * pYourParam),
        void * pYourParam )
```

3.4.3 The Wildcard Queue

Section 3.4.2 explained, in part, how to decouple mail processing from comms management by installing an active message queue via `AddMessageCallback` but that does result in running a thread for every message for which a callback is added. Although this is a common paradigm with many advantages som users may prefer to have the advantages of decoupling mail processing from mail reception and yet have all messages processed in the same thread. This is what the Wildcard Queue can be used for. Again the user uses `AddMessageCallback` but instead of specifying a particular message name as

the second parameter, they specify "*". This message name is interpreted specially inside the comms code and it means that any message that is not collected by other active queues (instantiated by calls to `AddMessageCallback`) will be placed in this queue. Note that you can install as many wildcard queues as you like (each with its own distinct nickname) and so you can have each and every message processed by any number of callbacks - its up to you - although most folk will be happy with one message handling thread per message type

3.5 Application Writing with `CMOOSApp`

There is one other major class which you really should know about and that is `CMOOSApp.` It is a convenience class which makes writing applications very simple. If you use it as a base class then your derived class inherits a processing loop, a CommsClient and mechanisms to read configuration files. All you have to do as a user is overload some key functions. These are

`Iterate()` By overriding the `CMOOSApp::Iterate` function in a new derived class, the author creates a function from which he or she can orchestrate the work that the application is tasked with doing. As an example, and without prejudice, imagine the new application was designed to control a mobile robot. The iterate function is automatically called by the base class periodically and so it makes sense to execute one cycle of the controller code from this "Iterate " function.

`OnNewMail()` This function is called when mail has arrived. The mail arrives in the form of a `std::list<CMOOSMsg>` — a list of CMOOSMsg objects. The programmer is free to iterate over this collection examining who sent the data, what it pertains to, how old it is, whether or not it is string, binary or numerical data and to act / process the data accordingly.

`OnConnectToServer()` is a callback from a thread in the CommsClient object that handles all the IPC communications 3. The callback occurs whenever contact has been made with the MOOSDB server and is a good place in which to make calls to `Register()` to subscribe for mail.

`OnStartUp()` This function is called just before the base class enters into its own "forever-loop" calling `Iterate` at regular intervals. This is the spot that you would populate with initialisation code, and in particular use the functionality provided by the `m_MissionReader` member object to read configuration parameters (including those that modify the default behaviour of the CMOOSApp base class from file.

4 Programming with MOOS Clients - Examples

This section exists to provide full examples of the ideas described in Section 3. If you are already pretty familiar with MOOS or the kind of programmer who learns by example or by imbibing deeply off full listings this is the place to start.

These examples are intended to be run standalone - so you just start a DB and simply run the example program. This in someways is unusual because more often than not interesting cases will involve two or more clients (for example a producer and a subscriber). Here the examples will be both subscriber and producer - it makes them compact. Just dont let this confuse you.

4.1 Index of Example Codes

All of the examples given in this document can be found in the examples subdirectory. Here is a table describing what each example illustrates.

Example	Description	New Detail
ex10	Simplest standalone comms client configuration. Uses polling to retrieve mail	using a raw comms clien
ex20	Use of a callback for mail reception instead of user polling for mail	Use of Mail callback
ex30	as ex20 only with command line parsing	command line parsing
ex40	Standalone comms client with active message queues	active queues
ex50	Installing a default active queue	default queue
ex60	Using Wildcard Registrations	wildcard registrations

4.2 The Hello World example - polling (Ex10).

In example 1 we can see the simplest of ways to use a CommsClient object. In `main()` we instantiate a comms client and install a single callback - one that will be invoked when a client succesfully connects to a `MOOSDB`. Then we start up the client asking it to name itself as "EX10" and we tell it that the `MOOSDB` it should connect to is running on localhost (the same machine) and on port 9000.

What follows is a simple loop which runs once a second (because of the 1000 millisecond `MOOSPause`). It posts a message under the name of `''Greeting''` which contains the string `''Hello''`. Then the loop fetches any and all incoming messages (since the last loop iteration) by calling Fetch. All mesages are now contained in a MOOSMSG_LIST which is simply a typedef for a `std::list` of `std::strings`. We then iterate over each member printing its contents. The only other part t pay attention to is what happens in the `OnConnect` callback. Here we told the client (and by implication the DB) that we want to receive "Greeting" messages. In this particular case then we are subscribing to a message we are sendind ourselves. This is not exactly common in practice but it makes for a compact first example. Note finally that allthough we are polling to pick up the mail, we are not polling to contact the MOOSDB - that is happening behind the scenes on our behalf in a separate mechanism.

Listing 1: Ex10: A simple example using MOOSAsyncCommClient and polling for mail

```
#include "MOOS/libMOOS/Comms/MOOSAsyncCommClient.h"

bool OnConnect(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);
    pC->Register("Greeting",0.0);
    return true;
}



int main(int argc, char * argv[]){

    //configure the comms
    MOOS::MOOSAsyncCommClient Comms;
    Comms.SetOnConnectCallBack(OnConnect,&Comms);

    //start the comms running
    Comms.Run("localhost",9000,"EX10");

    MOOSMSG_LIST M;
    for(;;){
        MOOSPause(1000);
        Comms.Notify("Greeting","Hello");
        Comms.Fetch(M);
        MOOSMSG_LIST::iterator q;
        for(q = M.begin();q!=M.end();q++)
```

```
        {
            q->Trace();
        }
    }
    return 0;
}
```

4.2.1 What is bad about this polling design?

This mechanism of having a communications thread (handled by the comms object) interact with the `MOOSDB` and to build a list of messages which you as a user is pretty much how MOOS used to work before V10 came along - although the mechanism was oftend wrapped up in a `MOOSApp` instance (Section3.5). Its main disadvantage is that you have to actively look to see if mail has arrived. This might be easy to accommodate, it might even be convenient, but sometimes you may want to respond to a message super fast with low latency. V10 can help here.

4.3 Installing and Using a Mail callback (ex20)

Another simple (in terms of its proximity to the core communication classes) example of using MOOS-V10 communications is given in Listing 2 below. Here a `MOOS::MOOSAsyncCommClient` is instantiated in its rawest form and one that does not need you to poll to pick up held mail. It is configured with a *Mail* and *Connect* callback and set free running with a call to `Run`(). Note that once again, in the Connect callback it registers for the data that is being posted once a second in the main() forever loop. Many MOOS users will be used to using `CMOOSApp` which manages the interaction with the Comms Client Objects however it is instructive to look at the most fundamental example.

Listing 2: Ex20: A simple example using MOOSAsyncCommClient

```
#include "MOOS/libMOOS/Comms/MOOSAsyncCommClient.h"

bool OnConnect(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);
    pC->Register("Greeting",0.0);
    return true;
}

bool OnMail(void *pParam){
    CMOOSCommClient* pC = reinterpret_cast<CMOOSCommClient*>(pParam);
    MOOSMSG_LIST M;
    pC->Fetch(M); //get the mail
    MOOSMSG_LIST::iterator q; //process it
    for(q=M.begin();q!=M.end();q++){
        q->Trace();
    }
    return true;
}

int main(int argc, char * argv[]){

    //configure the comms
    MOOS::MOOSAsyncCommClient Comms;
    Comms.SetOnMailCallBack(OnMail,&Comms);
    Comms.SetOnConnectCallBack(OnConnect,&Comms);
```

```
    //start the comms running
    Comms.Run("localhost",9000,"EX20");

    for(;;){
        MOOSPause(1000);
        Comms.Notify("Greeting","Hello");
    }
    return 0;
}
```

4.3.1 An aside - Using a command line parser (ex30)

This example is certainly raw, it assumes the `MOOSDB` is on localhost and port 9000. We could do a lot better by using the `MOOS::CommandLineParser` and using it to discover options provided on the command line as shown in listing 3:

Listing 3: Ex30: A fuller example using MOOSAsyncCommClient

```
/*
 * A simple example showing how to use a comms client
 */
#include "MOOS/libMOOS/Comms/MOOSAsyncCommClient.h"
#include "MOOS/libMOOS/Utils/CommandLineParser.h"

bool OnConnect(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);
    pC->Register("X",0.0);
    return true;
}

bool OnMail(void *pParam){
    CMOOSCommClient* pC = reinterpret_cast<CMOOSCommClient*>(pParam);

    MOOSMSG_LIST M; //get the mail
    pC->Fetch(M);

    MOOSMSG_LIST::iterator q; //process it
    for(q=M.begin();q!=M.end();q++){
        q->Trace();
    }
    return true;
}

int main(int argc, char * argv[]){

    //understand the command line
    MOOS::CommandLineParser P(argc,argv);

    std::string db_host="localhost";
    P.GetVariable("--moos_host",db_host);

    int db_port=9000;
    P.GetVariable("--moos_port",db_port);

    std::string my_name ="ex30";
```

```
    P.GetVariable("--moos_name",my_name);

    //configure the comms
    MOOS::MOOSAsyncCommClient Comms;
    Comms.SetOnMailCallBack(OnMail,&Comms);
    Comms.SetOnConnectCallBack(OnConnect,&Comms);

    //start the comms running
    Comms.Run(db_host,db_port,my_name);

    //for ever loop sending data
    std::vector<unsigned char> X(100);
    for(;;){
        MOOSPause(1000);
        Comms.Notify("X",X);
    }
    return 0;
}
```

To be complete, Listing 4 shows the complete `CMakeLists.txt` file for this example is given in listing 4

Listing 4: CMakeLists.txt for the simple example above

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)

if(COMMAND cmake_policy)
   cmake_policy(SET CMP0003 NEW)
endif(COMMAND cmake_policy)


#this builds an example program
set(EXECNAME ex30)

find_package(MOOS 10)

#what files are needed?
SET(SRCS ex30.cpp)

include_directories(  ${MOOS_INCLUDE_DIRS} ${MOOS_DEPEND_INCLUDE_DIRS})
add_executable(${EXECNAME} ${SRCS} )
target_link_libraries(${EXECNAME} ${MOOS_LIBRARIES} ${MOOS_DEPEND_LIBRARIES})
```

4.3.2 What is bad about this responsive design?

While this is a simple design it is not the best plan - it opens the door for doing an unbounded amount of work in the callback which is invoked by one of the threads which is used in handling communication with the `MOOSDB`. Now V10 DB's can handle this (each client is serviced by an indepedent thread) but its a better plan to use "Active Message Queues" as discussed in Section 3.4.2. The Section 4.4 provides an example of how to do that.

4.4 Adding Active Message Queues (ex40)

This example shows you how to add active queues for some messages. Figure 5 gives a pictorial view of how the threads involved in Active Message Queuing interact. The point to note and understand when looking at this example is that "X" and "Y" will be handled in different callbacks invoked from independent threads. This means you can take as long as you like to handle "X" and it won't interfere with the processing of "Y".

Listing 5: Ex40: Installing a per-message callback with an active message queue

```
/*
 * A simple example showing how to use a comms client
 */
#include "MOOS/libMOOS/Comms/MOOSAsyncCommClient.h"
#include "MOOS/libMOOS/Utils/CommandLineParser.h"
#include "MOOS/libMOOS/Utils/ConsoleColours.h"
#include "MOOS/libMOOS/Utils/ThreadPrint.h"

MOOS::ThreadPrint gPrinter(std::cout);

bool OnConnect(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);
    pC->Register("X",0.0);
    pC->Register("Y",0.0);
    pC->Register("Z",0.0);

    return true;
}

bool OnMail(void *pParam){
    CMOOSCommClient* pC = reinterpret_cast<CMOOSCommClient*>(pParam);

    MOOSMSG_LIST M; //get the mail
    pC->Fetch(M);

    MOOSMSG_LIST::iterator q; //process it
    for(q=M.begin();q!=M.end();q++){
        gPrinter.SimplyPrintTimeAndMessage("mail:"+q->GetSource(), MOOS::↩
            ThreadPrint::GREEN);
    }
    return true;
}

bool funcX(CMOOSMsg & M, void * TheParameterYouSaidtoPassOnToCallback)
{
    gPrinter.SimplyPrintTimeAndMessage("call back for X", MOOS::ThreadPrint::↩
        CYAN);
    return true;
}

bool funcY(CMOOSMsg & M, void * TheParameterYouSaidtoPassOnToCallback)
{
    gPrinter.SimplyPrintTimeAndMessage("call back for Y", MOOS::ThreadPrint::↩
        MAGENTA);
    return true;
}
```

```
int main(int argc, char * argv[]){

    //understand the command line
    MOOS::CommandLineParser P(argc,argv);

    std::string db_host="localhost";
    P.GetVariable("--moos_host",db_host);

    int db_port=9000;
    P.GetVariable("--moos_port",db_port);

    std::string my_name ="ex40";
    P.GetVariable("--moos_name",my_name);

    //configure the comms
    MOOS::MOOSAsyncCommClient Comms;
    Comms.SetOnMailCallBack(OnMail,&Comms);
    Comms.SetOnConnectCallBack(OnConnect,&Comms);

    //here we add per message callbacks
    //first parameter is the channel nick-name, then the function
    //to call, then a parameter we want passed when callback is
    //invoked
    Comms.AddMessageCallback("callback_X","X",funcX,NULL);
    Comms.AddMessageCallback("callback_Y","Y",funcY,NULL);

    //start the comms running
    Comms.Run(db_host,db_port,my_name);

    //for ever loop sending data
    std::vector<unsigned char> X(1000);
    for(;;){
        MOOSPause(10);
        Comms.Notify("X",X); //for callback_X
        Comms.Notify("Y","This is Y"); //for callback_Y
        Comms.Notify("Z",7.0); //no callback
    }
    return 0;
}
```

4.4.1 Adding a Default Active Message Queue (ex50)

We can extend this example by installing a default message queue - this will mean all messages that are not trapped by existing active queues will be thrown onto this default queue and its associated callback function will be invoked for each of these homeless and bereft messages.

Listing 6: Ex50: Installing a default active message queue callback

```
/*
 * A simple example showing how to use a comms client
 */
#include "MOOS/libMOOS/Comms/MOOSAsyncCommClient.h"
#include "MOOS/libMOOS/Utils/CommandLineParser.h"
#include "MOOS/libMOOS/Utils/ConsoleColours.h"
```

```
#include "MOOS/libMOOS/Utils/ThreadPrint.h"

MOOS::ThreadPrint gPrinter(std::cout);

bool OnConnect(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);

    //wildcard registration any two character name beginning with V
    pC->Register("V?","*",0.0);

    return true;
}

bool DefaultMail(CMOOSMsg & M, void * TheParameterYouSaidtoPassOnToCallback)
{
    gPrinter.SimplyPrintTimeAndMessage("default handler "+M.GetKey(), MOOS::↩
        ThreadPrint::CYAN);
    return true;
}

bool funcA(CMOOSMsg & M, void * TheParameterYouSaidtoPassOnToCallback)
{
    gPrinter.SimplyPrintTimeAndMessage("funcA "+M.GetKey(), MOOS::ThreadPrint↩
        ::CYAN);
    return true;
}


int main(int argc, char * argv[]){

    //understand the command line
    MOOS::CommandLineParser P(argc,argv);

    std::string db_host="localhost";
    P.GetVariable("--moos_host",db_host);

    int db_port=9000;
    P.GetVariable("--moos_port",db_port);

    std::string my_name ="ex50";
    P.GetVariable("--moos_name",my_name);

    //configure the comms
    MOOS::MOOSAsyncCommClient Comms;
    Comms.SetOnConnectCallBack(OnConnect,&Comms);

    //here we add per message callbacks
    //first parameter is the channel nick-name, then the function
    //to call, then a parameter we want passed when callback is
    //invoked
    Comms.AddMessageCallback("callbackA","V1",funcA,NULL);

    //add a default handler
    Comms.AddMessageCallback("default","*",DefaultMail,NULL);

    //start the comms running
```

```
    Comms.Run(db_host,db_port,my_name);

    //for ever loop sending data
    std::vector<unsigned char> data(1000);
    for(;;){
        MOOSPause(10);
        Comms.Notify("V1",data); //for funcA
        Comms.Notify("V2","This is stuff"); //will be caught by default
    }
    return 0;
}
```

Note also in this example we are using a **wildcard registration** (we do not specificy the complete message name but simply say we are interested in anything beginning with "V" followed by any character. We will look at some more wildcard examples in the next section

4.5 Wildcard Registrations (ex60)

A major improvement in MOOSV10 is the abiity to make wildcard registrations. In the next example we instantiate three clients Comms1, Comms2 and Comms3. All three are told to run a default message queue (so any and all mail callbacks are run in a single dedicated thread). However they are each given a different OnConnect callback withi which they make different registrations.

Comms1 registers for any message which comes from a client whose name begins with the letter "C"

Comms2 registers for any message which begins with the letter "V" followed by a single character but only if it comes from a client whose name ends in a 2.

Comms3 registers for any message from anyone, what a flibbertigibbet!

Only Comms1 and Comms2 publish data - they each send V1 and V2 the former being binary and the latter a string. So what shoud be printed by the mail callbacks?

- Comms1 should print each V1 and V2 from Comms1 and Comms2
- Comms2 should print one in two V1 and V2 - only if they come from Comms2
- Comms3 should also print all V1 and V2's but it will occasionally also print summaries from the MOOSDB like DB_TIME as it receives everything in the community.

Listing 7: Ex60: Using Wildcard registrations

```
/*
 * A simple example showing how to use a comms client
 */
#include "MOOS/libMOOS/Comms/MOOSAsyncCommClient.h"
#include "MOOS/libMOOS/Utils/CommandLineParser.h"
#include "MOOS/libMOOS/Utils/ConsoleColours.h"
#include "MOOS/libMOOS/Utils/ThreadPrint.h"

MOOS::ThreadPrint gPrinter(std::cout);


bool OnConnect1(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);
```

```
    //wildcard registration for any variable from a client who's name begins ↩
        with C
    return pC->Register("*","C*",0.0);
}

bool OnConnect2(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);

    //wildcard registration any two character name beginning with V
    //from a client who's name ends in "2"
    return pC->Register("V?","*2",0.0);
}

bool OnConnect3(void * pParam){
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (pParam);

    //wildcard registration for everything
    return pC->Register("*","*",0.0);
}


bool DefaultMail(CMOOSMsg & M, void * TheParameterYouSaidtoPassOnToCallback)
{
    CMOOSCommClient* pC =  reinterpret_cast<CMOOSCommClient*> (↩
        TheParameterYouSaidtoPassOnToCallback);

    gPrinter.SimplyPrintTimeAndMessage(pC->GetMOOSName()+ " : Rx : "+M.GetKey↩
        ()+
            " from "+ M.GetSource(),
            MOOS::ThreadPrint::GREEN);

    return true;
}


int main(int argc, char * argv[]){

    //understand the command line
    MOOS::CommandLineParser P(argc,argv);

    std::string db_host="localhost";
    P.GetVariable("--moos_host",db_host);

    int db_port=9000;
    P.GetVariable("--moos_port",db_port);

    std::string my_name ="ex60";
    P.GetVariable("--moos_name",my_name);

    //configure the comms
    MOOS::MOOSAsyncCommClient Comms1;
    Comms1.SetOnConnectCallBack(OnConnect1,&Comms1);
    Comms1.AddMessageCallback("default","*",DefaultMail,&Comms1);
    Comms1.Run(db_host,db_port,"C-"+my_name+"-1");

    MOOS::MOOSAsyncCommClient Comms2;
```

```
    Comms2.SetOnConnectCallBack(OnConnect2,&Comms2);
    Comms2.AddMessageCallback("default","*",DefaultMail,&Comms2);
    Comms2.Run(db_host,db_port,"C-"+my_name+"-2");

    MOOS::MOOSAsyncCommClient Comms3;
    Comms3.SetOnConnectCallBack(OnConnect3,&Comms3);
    Comms3.AddMessageCallback("default","*",DefaultMail,&Comms3);
    Comms3.Run(db_host,db_port,"C-"+my_name+"-3");

    //for ever loop sending data
    std::vector<unsigned char> data(1000);
    for(;;){
        MOOSPause(10);
        Comms1.Notify("V1",data);
        Comms1.Notify("V1",data);

        Comms1.Notify("V2","This is stuff");
        Comms2.Notify("V2","This is stuff");
    }
    return 0;
}
```

5 Writing Applications with CMOOSApp

We can of course achieve the same thing by subclassing CMOOSApp. The code listing below shows how. This section is not finished.....maybe broken into a new document...

Listing 8: A simple example using MOOSAsyncCommClient

```
/*
 * simple MOOSApp example
 */

#include "MOOS/libMOOS/App/MOOSApp.h"

class ExampleApp : public CMOOSApp
{
    bool OnNewMail(MOOSMSG_LIST & Mail)
    {
        //process it
        MOOSMSG_LIST::iterator q;
        for(q=Mail.begin();q!=Mail.end();q++){
            //q->Trace();
        }
        return true;
    }
    bool OnConnectToServer()
    {
        return Register("X",0.0);
    }
    bool Iterate()
    {
        std::vector<unsigned char> X(100);
        Notify("X",X);
```

```
        return true;
    }
};

int main(int argc, char * argv[])
{
    //here we do some command line parsing...
    MOOS::CommandLineParser P(argc,argv);
    //mission file could be first free parameter
    std::string mission_file = P.GetFreeParameter(0, "Mission.moos");

    //app name can be the  second free parameter
    std::string app_name = P.GetFreeParameter(1, "ExampleApp");

    ExampleApp App;

    App.Run(app_name,mission_file,argc,argv);

    return 0;
}
```

6 Configuring MOOSDB

6.1 Command Line Help

MOOSDB offers a command line interface which allows you to set the port it is serving on and various other configurations. All are accessed via `./MOOSDB --help`

```
>>pmn@mac  ./MOOSDB --help
MOOSDB command line help:
--moos_file=<string>                    specify mission file name
--moos_port=<positive_integer>          specify server port number
--moos_timewarp= <positive_float>       specify time warp
--moos_community=<string>               specify community name
--moos_timeout=<positive_float>         specify client timeout
--response=<string-list>                specify client response times <name:↩
    response_ms,... >
--warn_latency=<positive_float>         specify latency above which warning is ↩
    issued in ms
--webserver_port=<positive_integer>     run webserver on given port

--tcpnodelay                            disable nagle algorithm
-s (--single_threaded)                  run as a single thread
-d (--dns)                              run with dns lookup
-b (--moos_boost)                       boost priority of communications
-h (--help)                             print help and exit
```

6.2 Configuring Client Response Times

The MOOSDB has some inbuilt security controls that are designed to prevent a rogue, ill-mannered client to hog resources. It seems improper that a random client joining a community can decide to send 10 million messages per second and because of that, reduce the performance of other clients. On

the other hand it seems inappropriate to disallow all clients for all time very rapid performance simply because of a percieved risk. The solution offered in MOOS-V10 is that the MOOSDB by default offers premiums service to all comers [4] - in other words every client will be serviced as soon as possible and all clients will be have data pushed to them as soon as possible. However the launcher of the `MOOSDB` may choose to restrict response times for clients- this has the effect of having each transaction with the DB contain more indvidual messages and prevents rogue clients being disruptive. Even introducing a repsonse time of 10ms can have a marked increase in performance for a very heavily loaded system. It is also possible to control which clients should be throttled and which should not.

```
pmn@mac:~$ ./MOOSDB --response=*:20
pmn@mac:~$ ./MOOSDB --response=VisualOdometry:10
pmn@mac:~$ ./MOOSDB --response=Camera??:10,VisualOdometry:10,*:20
```

In the above, the first example sets all clients to have a minimum reposnse time of 20ms. The second example expicitly sets a client called `VisualOdometry` to have a 10ms response while all others have the default of 0ms (instant response). The final example has any client whos name begins with "Camera" followed by two characters set to 10ms and `VisualOdometry` at 10ms and every other client at 20ms.

6.3 Specifying When Clients are Assumed Dead

`MOOSDB` has alway been suspicious of clients that unexpectedly go quiet (the comms thread, which operates behind the scenes, stops working) and it will disconnect them. However its pretty annoying if you are debugging an application and because you could not solve you problem in 5 seconds, the DB disconnects your application and so differnt behaviour is invoked while debugging (the app will try to reconnect as soon the debugger sets teh application free). In V10, `MOOSDB` has the `--moos_timeout` option which allows you to specify the time in seconds the DB should tolerate a silent client. Set this to a big number when you are debugging.

6.4 Live Network Audit

Sometimes its nice to quickly get a summary of the network performance of the MOOSDB and the clients it supports. The MOOS V10 DB supports a very lightweight way to see how things are going. When the DB starts you'll see it print out something like `'network performance data published on localhost:9090 listen with "nc -u -lk 9090" '`. So if you follow this advice and in a terminal start `netcat (which is the 'nc' command)` listening on port 9090 it will receive UDP packets which contain performance data. Here is an example output - don't be put off by the fact that the client names are actually numbers in this case - that just happens to be the naming scheme this community was running. The network summary packet is sent once a second and contains valid statistics for that last second.

```
client name   pkts in  pkts out  msgs in  msgs out   B/s in  B/s out
       0        20        17       20     20          1207      1227
       1        19        19       19     19          1216      2177
   total        39        36       39       39        2423     3404
```

[4] if they are using the AsyncComms

7 Further Examples

7.1 Sharing Video Rate Data

Here is a simple example code for sharing video data using the package OpenCV [5]. The program can be started in one of two ways - once as a server which opens a camera and starts streaming images and as a client which displays them in a window. Note this is not an elegant program - it fixes the images size and does a fairly ugly bit of memory management. It is presented here as a quick and dirty exposition of using MOOS to send data at a moderate rate - its not an example of good use of OpenCV.

- Start a MOOSDB
- To start a server in a terminal window from the command line whilst in the directory containing the binary type :
  - `./camera_example -s --moos_name SERVER`
- To start a client from a similar terminal to that above type :
  - `./camera_example --moos_name A`
- To start another client, you guess it, open another terminal and try
  - `./camera_example --moos_name B`

If you do the above you should see you camera output appearing in two windows with very little lag.

Listing 9: Example code to build a camera sharing example

```
#include "opencv2/opencv.hpp"
#include "MOOS/libMOOS/App/MOOSApp.h"

class CameraApp : public CMOOSApp
{
public:
    bool Iterate()
    {
        if(server_){
            vc_>>capture_frame_;
            cv::cvtColor(capture_frame_, bw_image_, CV_BGR2GRAY);
            cv::resize(bw_image_, image_, image_.size(), 0, 0, cv::↩
                INTER_NEAREST);
            Notify("Image",(void*)image_.data,image_.size().area(),↩
                MOOSLocalTime());
        }
        else{
            cv::imshow("display", image_);
            cv::waitKey(10);
        }
        return true;
    }
    bool OnStartUp()
```

[5] so you will need OpenCV installed on your machine. The CMakeLists.txt file should find this installation and handle everything for you but if you are using mac ports you may need to specify the location of OpenCV in the ccmake gui as Cmake does not look in /opt by default.

```
{
    SetAppFreq(20,400);
    SetIterateMode(COMMS_DRIVEN_ITERATE_AND_MAIL);

    image_ = cv::Mat(378,512,CV_8UC1);

    if(server_){
        if(!vc_.open(0))
            return false;
    }
    else{
        cv::namedWindow("display",1);
    }

    return true;
}
void OnPrintHelpAndExit()
{
    PrintDefaultCommandLineSwitches();
    std::cout<<"\napplication specific help:\n";
    std::cout<<"  -s             : be a video server grabs and sends images↩
        (no window)\n";
    exit(0);
}
void OnPrintExampleAndExit()
{
    std::cout<<" ./video_share -s    \n";
    std::cout<<" and on another terminal..\n";
    std::cout<<" ./video_share       \n";
    exit(0);
}

bool OnProcessCommandLine()
{
    server_=m_CommandLineParser.GetFlag("-s");

    return true;
}
bool OnNewMail(MOOSMSG_LIST & mail)
{
    MOOSMSG_LIST::iterator q;
    for(q = mail.begin();q!=mail.end();q++){
        if(q->IsName("Image")){
            std::cerr<<"bytes: "<<q->GetBinaryDataSize()<<" latency "<<
                    std::setprecision(3)<<(MOOSLocalTime()-q->GetTime())*1↩
                        e3<<" ms\r";

            memcpy(image_.data,q->GetBinaryData(),
                    q->GetBinaryDataSize());
        }
    }
    return true;
}
bool OnConnectToServer()
{
    if(!server_)
```

```
            Register("Image",0.0);
        return true;
    }
protected:
    cv::VideoCapture vc_;
    cv::Mat capture_frame_,bw_image_,image_;
    bool server_;


};
int main(int argc, char* argv[])
{
    //here we do some command line parsing...
    MOOS::CommandLineParser P(argc,argv);
    //mission file could be first free parameter
    std::string mission_file = P.GetFreeParameter(0, "Mission.moos");
    //app name can be the  second free parameter
    std::string app_name = P.GetFreeParameter(1, "CameraTest");

    CameraApp App;
    App.Run(app_name,mission_file,argc,argv);

    return 0;
}
```

Listing 10: CMakeLists.txt to build the camera sharing example above

```
#this builds an example program
project(camera_example)
if(COMMAND cmake_policy)
       cmake_policy(SET CMP0003 NEW)
endif(COMMAND cmake_policy)

cmake_minimum_required(VERSION 2.8)

#find MOOS version 10 or later
find_package(MOOS 10)

find_package( OpenCV )

set(EXECNAME video_share)

#what files are needed?
set(SRCS CameraExample.cpp)

#what include directives?
include_directories(  ${MOOS_INCLUDE_DIRS} ${MOOS_DEPEND_INCLUDE_DIRS} ${↩
    OpenCV_INCLUDE_DIRS})

#make a program!
add_executable(${EXECNAME} ${SRCS} )

#and link thus...
target_link_libraries(${EXECNAME} ${MOOS_LIBRARIES} ${MOOS_DEPEND_LIBRARIES} $↩
    {OpenCV_LIBS})
```

There are several things to note about this example which are worth spotting:

1. The way in which MOOS-V10 can handle command line argument parsing for you using the `OnParseCommandLine()` virtual function in `CMOOSApp`. Also note that the switches like `--moos_name` are handled automatically for you. If this is a surprise read section **??**.
2. The way in which in this example `SetIterateMode` is used to make the application respond quickly to the reception of mail.