

# MOOS Communications V10

November 19, 2012

## Contents

<b>1</b>	<b>Headlines</b>	<b>2</b>
<b>2</b>	<b>Structural Changes</b>	<b>2</b>
2.1	Seperate Projects . . . . .	2
2.2	Library Structure and Header Files . . . . .	2
<b>3</b>	<b>Developing with V10</b>	<b>3</b>
3.1	Ethos . . . . .	3
3.1.1	No code change . . . . .	3
3.1.2	Hetereogenous Systems . . . . .	4
3.2	Building V10 . . . . .	4
3.2.1	The quickest way . . . . .	4
3.2.2	Changing <code>#include</code> directives: . . . . .	6
3.3	Importing and Building Against MV10 . . . . .	6
3.3.1	How is MOOS found? . . . . .	7
3.3.2	Trouble Shooting . . . . .	7
<b>4</b>	<b>Leveraging V10</b>	<b>8</b>
4.1	Asynchronous Comms . . . . .	8
4.1.1	Low Latency Communications via <code>MOOSAsyncCommClient</code> . . . . .	8
4.1.2	Supporting Iterate Modes in <code>MOOSApp</code> . . . . .	8
4.2	Wildcard Subscriptions . . . . .	10
4.3	Common Command Line Interface . . . . .	11
4.3.1	New Command Line Related Functions to Overload . . . . .	11
4.4	MOOSDB Interface Improvements . . . . .	12
4.4.1	Command Line . . . . .	12
4.4.2	Live Network Audit . . . . .	12
4.5	Backwards Compatibility . . . . .	12
4.5.1	Retreating back to the known . . . . .	13
<b>5</b>	<b>Bench Marks</b>	<b>13</b>
5.1	Using <code>uDBAsyncTest</code> . . . . .	13
<b>6</b>	<b>Example Codes</b>	<b>17</b>
6.1	The simplest example using <code>MOOSAsyncCommClient</code> . . . . .	17
6.2	The Simplest Example using <code>CMOOSApp</code> . . . . .	18
6.3	Sharing Video Rate Data . . . . .	19
<b>7</b>	<b>Under the hood</b>	<b>22</b>

# 1 Headlines

MOOS Version 10 is a major new release of MOOS. It incorporates many changes that improve its performance and answer to requests made by the user community. This document sets out the features, changes and their implications on users of the communications API of MOOS.

## Functional Headlines

- **Multithreaded Design** The MOOSDB now offers improved resilience when working with clients which reside at the end of a high latency or low bandwidth link. Each client is now furnished with its own thread within the DB which allows each client to take as long as it needs to complete and `read` or `write` without holding up the interactions of other clients. This behaviour can be disabled from the command line in which case the behaviour of the DB reverts to pre V10 behaviour.
- **Wildcard Subscriptions** The MOOS communications API now supports wildcard subscriptions whereby clients can subscribe to variable which match variable name patterns and also source name patterns. For example a client could register for “\*:” meaning all variables from all sources or “J\*:K” meaning any variable beginning with “J” from a process ending in “K”
- **Reduced Latency** Aysnchronous communications are now supported via an upgraded MOOSDB and a new kind of communications client object `MOOSAsyncCommClient`. This allows data to be pushed to clients by the MOOSDB rather than clients having to fetch messages which match their subscriptions when the call into the MOOSDB. This affords a drastically decreased latency between data being published and clients receiving it.

## Structural Headlines

- **Fission into smaller, neater delineated code trees:** The communication layer, utility applications (like `pLogger`), and graphical tools all now live in seperate projects. For example, it is now possible to download only the communications layer and work just with that.
- **A single core library:** There is now just one core library `libMOOS` which is an amalgamation of what was previously `MOOSGenLib` and `MOOSLib`
- **A rationalised header file structure:** The header structure for commonly used files has changed to reflect this (but a backwards compatibility mode is provided so users can in the first instance carry on as if this is not the case)

# 2 Structural Changes

## 2.1 Seperate Projects

Pre version 10, the MOOS project was an ugly mash-up of source code which covered a strange span of functionality. This belied its maritime roots- it included things like `iINS` and `pNav` -applications which had clear heritage in marine autonomy and also the domain independent communications tools. In V10 these have been teased apart. There is now a standalone `core-moos` library which only contains communication and domain neutral basic utilities. This project is the sole focus of this document.

## 2.2 Library Structure and Header Files

The classes that implement the communications and application management (for example `CMOOSApp`) now reside in a single library called `libMOOS`. There is no `MOOSGenLib` anymore - the classes and function that lived in that lump of code now reside in `libMOOS` and the headers can be found in a

```
→ CoreMOOS git:(speedy) ✖ tree -d -L 3
.
├── Core
│   ├── MOOSDB
│   │   └── testing
│   ├── docs
│   └── libMOOS
│       ├── App
│       ├── Comms
│       ├── Thirdparty
│       ├── Utils
│       └── include
└── cmake
```

Figure 1: Top-level directory structure for MOOS V10

subdirectory called **Utils** within **core-moos**. There are in fact four key subdirectories in **libMOOS**. In figure 1 you can see the basic structure of the V10 code base.

**App** contains the classes like **CMOOSApp** and **CMOOSInstrument**

**Comms** contains everything to do with IPC communications

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

**Thirdparty** contains small pots 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)

You might be wondering where all the header files that used to be in **MOOSGenLib** have gone. They are now in “**MOOS/libMOOS/Utils/\*.h**”.

**Tip:** If you were previously including “**MOOSGenLibGlobalHelper.h**” then you now need to include “**MOOS/libMOOS/Utils/MOOSUtilityFunctions.h**” instead (or use the compatibility mode described in Section 3.2)

## 3 Developing with V10

### 3.1 Ethos

#### 3.1.1 No code change

A lot of effort has been taken to make the users transition to MOOS V10 painless. Indeed the goal was to make it possible to upgrade to V10 without having to change any source code. The only thing a user does need to do is link against the new library and this is made easy with the revamped **CMake** build system.

### 3.1.2 Heterogeneous Systems

No assumption is made that all components of a MOOS system will be upgraded. It is entirely possible to use a holy relic pre-V10 clients or an old trusted MOOSDB with new or rebuilt software which has linked against V10. The motivation here is to start using V10 you don't need to rebuild everything. You could for example simply run the new MOOSDB and you will still get improved performance. If your circumstances dictate, you can even run the new MOOSDB in safe mode in which it reverts to running the pre-V10 source code.

## 3.2 Building V10

### 3.2.1 The quickest way

Checkout a clean version of MOOS V10:

```
pmn@mac:~$ mkdir core-moos-v10;
pmn@mac:~$ cd core-moos-v10;
pmn@mac:~$ git clone path-to-git-repos src
pmn@mac:~$ mkdir build
pmn@mac:~$ cmake ../src
```

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 2 (note some of the entries are platform dependent so don't worry if what you see is not identical to this). If you simply want to link your existing code against MOOS V10 without needing to worry about the new header file structure 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 versions of MOOS. If you "include" one of these files they actually simply redirect to include header files residing in the new structure. 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.

**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

You 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** 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 3. You should be able to use this MOOSDB to manage all communications with any existing MOOS applications you have lying around - you should not have to upgrade them. MOOS V10 is backward compatible. You are probably wondering if just running this new DB by itself buys you anything. The answer is yes. Each client has its own thread so if you have dodgy comms between one client and the MOOSDB this ne'er-do-well client will not interfere with other client DB interactions - it won't be able to hold them up.

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

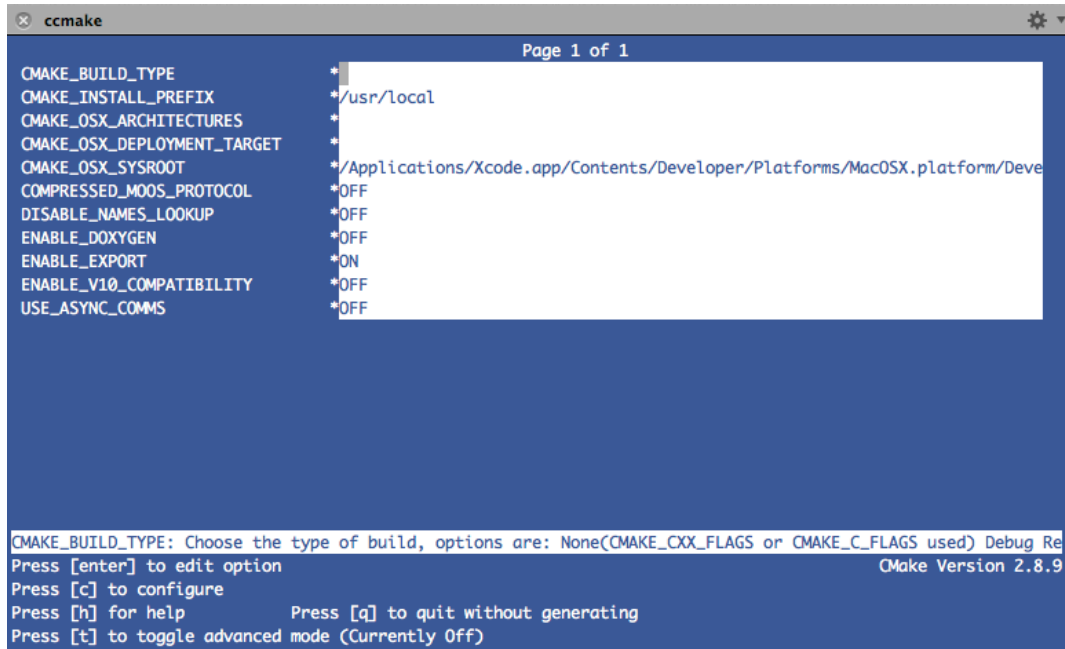


Figure 2: The default build screen for MOOS V10

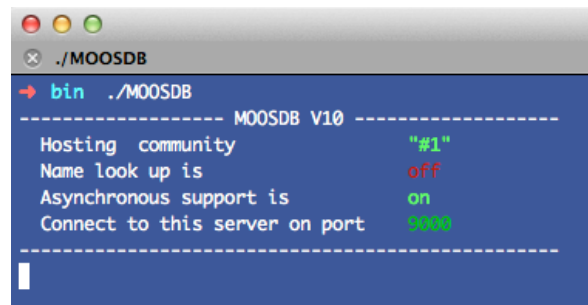


Figure 3: running the new MOOSDB

### 3.2.2 Changing #include directives:

If you are prepared to invest 30 minutes in committing to the new MOOS V10 project structure then this section tells you what to do. If you are as yet unsure if you want to upgrade, then don't bother simply use the `ENABLE_V10_COMPATIBILITY` option in CMake - this allows you to simply revert to old versions of MOOS without lifting a finger.

The actions needed to upgrade are pretty simple. Where previously you had something like `#include "MOOSLIB/MOOSCommClient.h"` you now use `#include "MOOS/libMOOS/Comms/MOOSCommClient.h"`. The following table will help you figure out how to include an particular header.

Header	include prefix
MOOSAsyncCommClient.h MOOSMsg.h XPCEndian.h XPCTcpSocket.h MOOSCommClient.h MOOSSkewFilter.h XPCException.h XPCUdpSocket.h MOOSCommObject.h MOOSVariable.h XPCGetHostInfo.h MOOSCommPkt.h ServerAudit.h XPCGetProtocol.h MOOSCommServer.h ThreadedCommServer.h XPCSocket.h	#include "MOOS/libMOOS/Comms/---.h"
ConsoleColours.h MOOSMemoryMapped.h MOOSTimeJournal.h IPV4Address.h MOOSNTSerialPort.h MOOSUtilityFunctions.h InterpBuffer.h MOOSPlaybackStatus.h MOOSUtils.h KeyboardCapture.h MOOSSafeList.h NTSerial.h MOOSAssert.h MOOSSafeList.h~ ProcessConfigReader.h MOOSException.h MOOSScopedLock.h SafeList.h MOOSFileReader.h MOOSSerialPort.h TMaxPair.h MOOSLinuxSerialPort.h MOOSThread.h TMinPair.h MOOSLock.h MOOSThreadedTimeJournal.h ThreadPrint.h	#include "MOOS/libMOOS/Utils/---.h"
MOOSApp.h MOOSInstrument.h	#include "MOOS/libMOOS/App/---.h"
MOOSLib.h	#include "MOOS/libMOOS/MOOSLib.h"

## 3.3 Importing and Building Against MV10

So now you have built the new MOOS. Next question 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 the 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})

```

### 3.3.1 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. Its 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.

### 3.3.2 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 you 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?

## 4 Leveraging V10

This section will explain how programmers and users of MOOS can leverage some of the important and hopefully helpful new functionality in MOOS V10.

### 4.1 Asynchronous Comms

This section explains new functionality offered in the V10 communications classes which allow for very low latency communications between components. This new facility allows one clients write of data to instigate a read on all other clients who have previously expressed interest in that data. This then is a departure from the model of a client having to call in to the DB, deliver its post and while “on the line”, pick up and mail the DB has waiting for it. Of course we must stress that the user has to opt in for this new functionality - there is no imperative to run new code. To enable Aysnchronous Comms you need to enable `USE_ASYNC_COMMS` in a configure time in CMake.

**Tip:** To enable fast asynchronous comms you have to turn `USE_ASYNC_COMMS` to ON when configuring MOOS using CMake

#### 4.1.1 Low Latency Communications via `MOOSAsyncCommClient`

The key class is `MOOS::MOOSAsyncCommClient` which is a derivate of the tried and test `CMOOSCommClient`. Its interface is identical to `CMOOSCommClient` and so the user should notice no programmatical difference in using this client.<sup>1</sup>

When a `MOOS::AsyncCommClient` connects to a V10 MOOSDB it instigates some quite different behaviour. Firstly the DB spawns two additional threads -one to handle reading from the client and one to handle writing to the client. These threads are not synchronised - they operate independently pulling and pushing data from work queues from withing the DB itself. The client itself also has distinct read and write threads - when a user posts some data it is added to a work queue on which the read thread is waiting. The read thread pushes this data to the DB and simply waits for another chunk of work to appear on the queue. Similarly the clients read thread sits in a blocking read on the socket linking it to the DB. When data arrives it is placed into the clients “mailbox” and optionally a user callback is invoked. Importantly, this architecture of each client having a read and write thread at the client and MOODB ends allows for data to be pushed to clients at any time. Take for example the case of 50 clients all having subscribed for variable “X” when the 51st client publishes “X” this data can be instantly placed on the outgoing queue of all 50 interested clients. Because the read threads on the clients are in a blocking read they two an respond immediately leading to some very responsive behaviour. Note also that if one of those clients has a dodgey communications link to the DB this has not effect on the other 49 clients. This then is in stark contrast to the pre V10 releases of MOOS. In section 5 some performance metrics are given which highlight the difference in behaviour between V10 and previous incantations of the MOOS communications API.

#### 4.1.2 Supporting Iterate Modes in `MOOSApp`

If MOOS is compiled with `USE_ASYNC_COMMS` then the `m_Comms` member of `CMOOSApp` becomes a `MOOSAsyncCommClient` and so all communications will be using this new faster functionality. `CMOOSApp` is designed to provide an easy to use framework in which to write applications which leverage the MOOS communications API. The ability for `MOOSAsyncCommClients` to have data pushed to them and invoke an asynchronous callback affords the opportunity to augment the behaviour of `CMOOSApp`

<sup>1</sup>Indeed many users have little direct interaction with the communications object preferring instead to operate withing the comfort of classes derived from `CMOOSApp` which wraps the low level communications API.



to provide application developed with greater flexibility and develop apps which respond quickly to communication events.

MOOS V10 offers three new configuration modes which are described in the tabel below. The mode in which the application operates can be set either in the applications configuration block (e.g by having a line like `IterateMode = 2` or programatically by calling `SetIterateMode(REGULAR_ITERATE_AND_COMMS_DRIVEN_MAIL)`). These modes are supported by an additional configuration parameter called `MaxAppTick` who's function is described in the table. This new parameter can be set in the configuration file `MaxAppTick=100` or passed as second parameter in `CMOOSApp::SetAppFreq(AppTick,MaxAppTick)`.

	REGULAR_ITERATE_AND_MAIL
Summary	This mode is the default just as in pre-V10 releases <code>Iterate()</code> and <code>OnNewMail()</code> are called regularly and if mail is available, in lock step.
Configuration Block	<code>IterateMode=0</code>
OnNewMail	called at most every <code>1/AppTick</code> seconds. If mail has arrived <code>OnNewMail()</code> will be called just before <code>Iterate()</code>
Iterate	called every <code>1/AppTick</code> seconds. So if <code>AppTick=10</code> <code>Iterate()</code> will be called at 10Hz.
Role of AppTick	sets the speed of <code>Iterate()</code> in calls per second
Role of MaxAppTick	not used
Role of CommsTick	not used as communications are asynchronous

	COMMS_DRIVEN_ITERATE_AND_MAIL
Summary	The rate at which <code>Iterate</code> is called is coupled to the reception of mail. As soon as mail becomes available <code>OnNewMail</code> is called and is then followed by <code>Iterate()</code> . If no mail arrives for <code>1/AppTick</code> seconds then <code>iterate</code> is called by itself. When mail is arriving <code>Iterate()</code> and <code>OnNewMail()</code> are synchronous - if <code>OnNewMail()</code> is called it will always be followed by a called to <code>Iterate()</code>
Configuration Block	<code>IterateMode=1</code>
OnNewMail	Called at up to <code>MaxAppTick</code> times per second. So if <code>MaxAppTick=100</code> <code>OnNewMail()</code> will be called in response to the reception of new mail at up to 100Hz.
Iterate	called at least <code>AppTick</code> times per second (if no mail) and up to <code>MaxAppTick</code> times per second
Role of AppTick	sets a lower bound on the frequency at which <code>Iterate()</code> is called. So if <code>AppTick = 10</code> then <code>Iterate</code> will be called at at least 10Hz
Role of MaxAppTick	sets an upper limit on the rate at which <code>Iterate</code> (and <code>OnNewMail</code> ) can me called. If <code>MaxAppTick=0</code> both the speed is unlimited.
Role of CommsTick	not used as communications are asynchronous

	REGULAR_ITERATE_AND_COMMS_DRIVEN_MAIL
Summary	Iterate is called regularly and OnNewMail is called when new mail arrives. Iterate will not always be called after OnNewMail unless it is scheduled to do so. In this way OnNewMail and Iterate are decoupled.
Configuration Block	IterateMode=2
OnNewMail	Called as soon as mail is delivered at up to MaxAppTick times per second.
Iterate	called every AppTick times per second
Role of AppTick	sets the speed of Iterate() in calls per second as in REGULAR_ITERATE_AND_MAIL
Role of MaxAppTick	limits the rate at which OnNewMail is called. If MaxAppTick=0 both the speed is unlimited. With a slight abuse of notation in this mode MaxAppTick does not control Iterate() speed at all - it simply limits the rate at which new mail can be responded to
Role of CommsTick	not used as communications are asynchronous

## 4.2 Wildcard Subscriptions

MOOS-V10 extends the way in which clients can subscribe for data by allowing “wildcard subscriptions”. A client can register its interest in variable whose name and source matches a simple regex pattern. Currently only patterns containing \* and ? wildcards are supported with their usual meanings so ? means any single character and \* means any number of characters. An example will make this whole thing clear and we will be using the new CMOOSApp::Register( sVarPattern, sAppPattern,dfInterval) interface.

```
bool MyApp::OnConnectToServer()
{
    //register for all variables ending with "image"
    //from any process with an name beginning with "camera_"
    Register("*image","camera_*, 0.0);

    //register for every single variable coming from a process
    //called "system_control"
    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
    Register("error_*","process_0?", 2.0);
    return true;
}
```

The logic which supports this new functionality is implemented at the MOOSB and turns out to be a pretty useful and compact way to define some fine granularity on what MOOSApp (or indeed CommsClient because that is the fundamental communications object) receives. 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.

### 4.3 Common Command Line Interface

MOOSApp now supports a whole set of command line options which, by making a very small change to your code will make all your programs which use MOOSApp respond in the same way <sup>2</sup>. The upgrade is effected by invoking a new version of `CMOOSApp::Run` which takes `argc` and `argv` as parameters:

```
bool Run(const std::string & sName, const std::string & sMissionFile, int argc←
, char * argv[]);
bool Run( const std::string &, int argc, char * argv[]);
```

This in turn populates a member variable within `CMOOSApp` called `m_CommandLineParser` which is used internally to parse the following commandline variables and flags.

```
variables:
--moos_app_name=<string>      : name of application
--moos_name=<string>          : name with which to register with MOOSDB
--moos_file=<string>          : name of configuration file
--moos_host=<string>           : address of machine hosting MOOSDB
--moos_port=<number>           : port on which DB is listening
--moos_apptick=<number>        : frequency of application (if relevant)
--moos_commstick=<number>      : frequency of comms (if relevant)
--moos_iterate_Mode=<0,1,2>    : set app iterate mode
flags:
--moos_iterate_no_comms       : enable iterate without comms
--moos_filter_command         : enable command message filtering
--moos_no_sort_mail           : don't sort mail by time
--moos_no_comms               : don't start communications
--moos_quit_on_iterate_fail   : quit if iterate fails
help:
--moos_print_example          : print an example configuration block
--moos_print_interface        : describe the interface (subscriptions/pubs)
--moos_print_version          : print the version of moos in play
--moos_help                   : print this message
--help                        : print this message
```

**Tip:** To enable common command line parsing call `CMOOSApp::Run(moos_name,argc,argv)` or a variant when you start a `MOOSApp`

#### 4.3.1 New Command Line Related Functions to Overload

MOOS-V10 adds new virtual functions to `CMOOSApp` which can be overloaded to process additional command line parameters if `argc`, and `argv` have been passed to your `CMOOSApp` derived class.

- `OnProcessCommandLine()` is called so you can do additional command line parsing
- `OnPrintExampleAndExit()` is called when `--moos_print_example` is present on the command line. The intent is you use this function to print out an example configuration file block.
- `OnPrintInterfaceAndExit()` is called when `--moos_print_interface` is present on the command line. The intent is you use this function to print out details of the processes subscriptions and publications.

<sup>2</sup>sadly this does require a code change as there is no other way to get command line parameters in `CMOOSApp`.

- OnPrintHelpAndExit() is called when -moos\_help or -help is present on the command line. The intent here is you print out help for additional command line parameters here.

## 4.4 MOOSDB Interface Improvements

### 4.4.1 Command Line

Its pretty dull to only be able to configure processes from a script. MOOSDB now supports a better command line interface which allows you to set the port it is serving on and various other configurations. All accessed via ./MOOSDB --help

```
>>pmn@mac ./MOOSDB --help
MOOSDB command line help:
-m      (--mission_file) <string>          specify mission file name
-p      (--server_port)  <positive_integer> specify server port number
-s      (--single_threaded)          run as a single thread
-w      (--time_warp)    <positive_float>  specify time warp
-d      (--dns)           run with dns lookup
-c      (--community)    <string>          specify community name
-W      (--webserver)    <positive_integer> run webserver on given port
-h      (--help)          print help and exit
```

### 4.4.2 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	B/s in	B/s out
	0	20	20	1207	1227
	1	19	19	1216	2177
	2	19	29	171	1241
	3	19	19	1216	2177
	4	19	29	171	1241
	5	19	19	1216	2177
	6	19	29	171	1241
	7	19	19	1216	2177
	total	153	183	6584	13658

## 4.5 Backwards Compatibility

MOOS V10 was designed to offer complete backwards compatibility between all versions of MOOS. You should be able to run legacy code with modern clients and old DB's alike. You should be able

to run heterogenous communities with any combination of pre V10 and V10 applications. The table below shows the options available for different combinations.

Client	MOOSDB	OK	Async. Comms	Synch. Comms	Multithreading DB	Single Threaded DB
Pre10	Pre10	✓	✗	✓	✗	✓
Pre10	V10	✓	✗	✓	✓	✓
V10	Pre V10	✓	✗	✓	✗	✓
V10	V10	✓	✓	✓	✓	✓

#### 4.5.1 Retreating back to the known

It is also possible to force the MOOSDB to behave (ie run almost exactly the same code) as previous versions did. So if you are using the V10 code base but want to return to the good old days the recipe is:

1. Run MOOSDB with the single threaded switch `./MOOSDB -s`
2. Make sure you compile V10 with `USE_ASYNC_COMMS=OFF` and `ENABLE_V10_COMPATIBILITY=ON`

## 5 Bench Marks

### 5.1 Using uDBAsyncTest

It is useful to have some statistical sense of performance. Its nice, when you have built V10 to have some sense of latencies and throughputs. In the directory build directory `bin` you will find an executable called `uDBAsyncTest` which can be used to do some benchmarking. `uDBAsyncTest` instantiates a configurable number  $2N$  of Comms Clients  $N$  of which are AsyncCommsClients and the other  $N$  of which are pre-V10 clients. Client  $C_j$  is asynchronous if  $j$  is even. All clients register with the MOOSDB under the numerical name  $j$  (so “1”, “2” etc) Client  $C_0$  is Asynchronous and is charged with regularly posting a variable “X” or configurable size to the MOOSDB. All clients including  $C_0$  subscribe to X. The time in milliseconds between  $C_0$  sending and client  $C_j$  receiving the data is logged to file `asynctest.log` as a plain text matrix :

$$\begin{array}{cccccc}
 0 & \tau_1^0 & \tau_2^0 & \tau_3^0 & \dots \\
 1 & \tau_1^1 & \tau_1^1 & \tau_1^1 & \dots \\
 2 & \tau_1^0 & \tau_1^0 & \tau_1^0 & \dots \\
 \vdots & \vdots & \vdots & \vdots & \ddots
 \end{array}$$

where  $^j\tau_i$  is the  $i^{th}$  latency for client  $C_j$  and each row begins with  $j$ . The test can be run for a configurable number of seconds after which histograms of performance for each kind of client can be produced. The options for `uDBAsyncTest` are below.

```

V10 performance and compatibility testing
-p (--test_period)           : test period in seconds (20 seconds ←
  default)
-m (--message_period)        : send test data every m milliseconds (←
  default 100 ms)
-c (--num_clients)           : number of clients to instantiate (←
  default 40)
-s (--payload_size)          : size of data to send default (default ←
  1024 bytes)

```

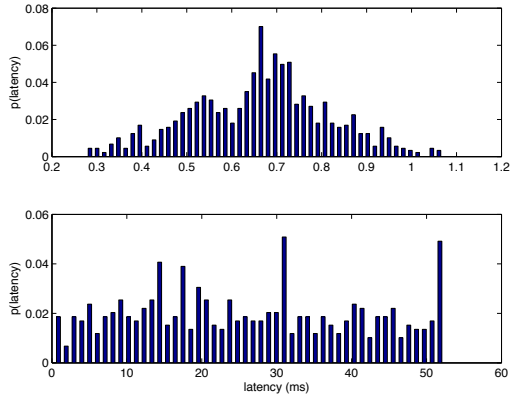
```
Example Usage: test for 15 seconds with 20 clients and sending 100K every 50 ↵
ms

./uDBAysncTest -p 15 -c 20 -m 50 -s 100000
```

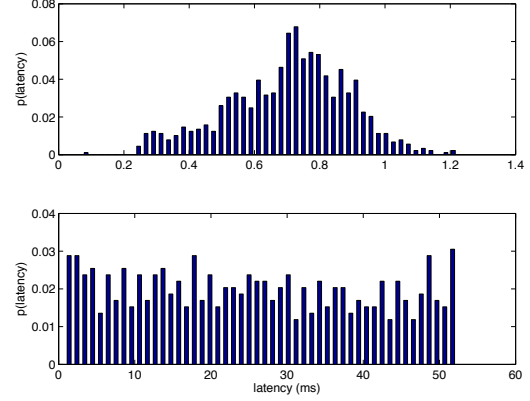
The figures on the following pages give the probability distributions over message latencies under different conditions (specified in caption). For example Figure 4 was generated with

```
./uDBAysncTest -p 20 -c 5 -m 50 -s 1000
./uDBAysncTest -p 20 -c 5 -m 50 -s 10000
./uDBAysncTest -p 20 -c 5 -m 50 -s 100000
./uDBAysncTest -p 20 -c 5 -m 50 -s 1000000
```

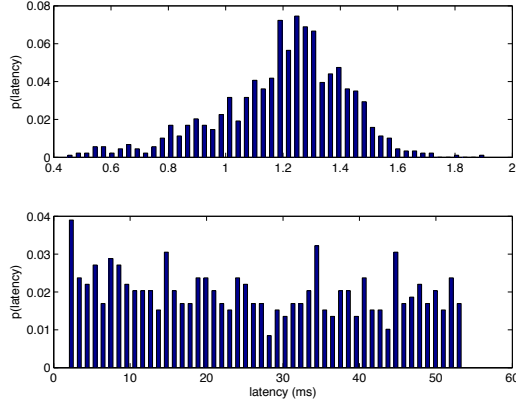
Which fires 1K,10K 100K and 1MB messages at 20Hz (every 50ms) to the DB which are immediately routed to 5 clients. This sort of testing not only gives a sense of latencies but also a sense of the total through-put of the system. For the results given here the MOOSDB was on the same machine as ./uDBAysncTest which means we are testing logic speed not network speed. The results are gathered on a 2.2GHz Intel Core i7 Mac book Pro running OS X 10.8 and all code was built in Debug (so these results are conservative). Its pretty clear that V10 clients offer between one and two orders of magnitude improvement in performance interms of latency.



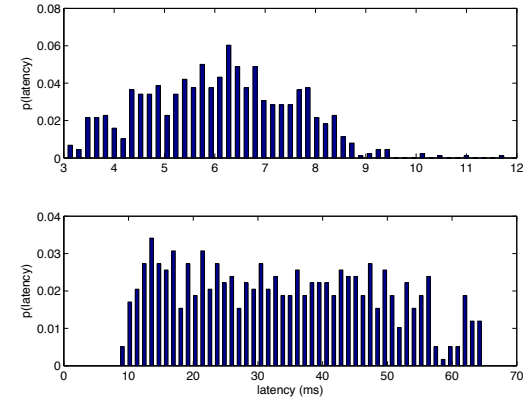
(a) Sending and receiving 1KB messages.



(b) Sending and receiving 10KB messages.

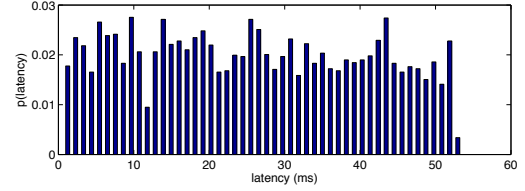
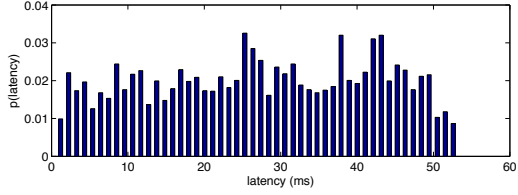
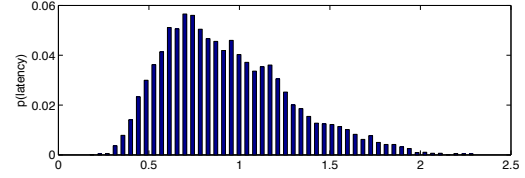
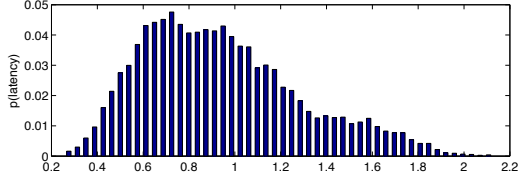


(c) Sending and receiving 100KB messages.



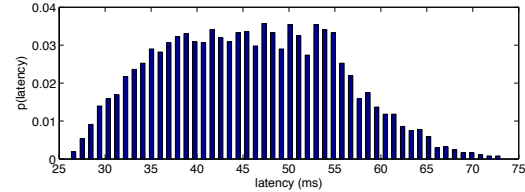
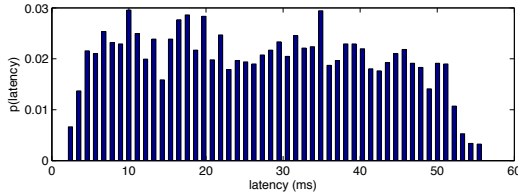
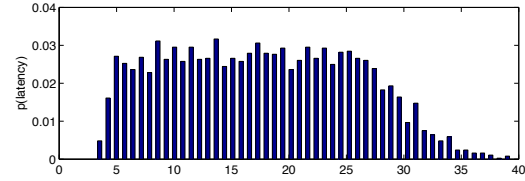
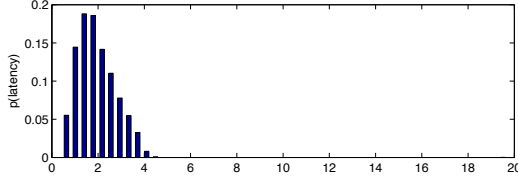
(d) Sending and receiving 1MB messages

Figure 4: Histograms of latencies between sending and receiving 1KB,10KB, 100KB and 1MB messages. Messages are sent to 5 clients at 20Hz. Top figures are for asynchronous clients lower figures are for pre V10 clients with a comms-tick to 20Hz. All cases are using the V10 MOOSDB. As an example of total through put take the example of sending 100KB messages to 5 clients 20 times a second so  $100K \cdot 5 \cdot 20 = 10MB/s$ .



(a) Sending and receiving 1KB messages.

(b) Sending and receiving 10KB messages.



(c) Sending and receiving 100KB messages.

(d) Sending and receiving 1MB messages at 10Hz

Figure 5: Histograms of latencies between sending and receiving 1KB, 10KB, 100KB and 1MB messages. Messages are sent to 50 clients at 20Hz unless otherwise stated. Top figures are for asynchronous clients lower figures are for pre V10 clients with a comms-tick to 20Hz. All cases are using the V10 MOOSDB. As an example of total throughput take the example of sending 100KB messages to 50 clients 20 times a second so  $100K \times 50 \times 20 = 100MB/s$ .



## 6 Example Codes

### 6.1 The simplest example using MOOSAsyncCommClient

The simplest (in terms of its proximity to the core communication classes) example of using MOOS-V10 communications is given in figure ???. Here a MOOS::MOOSAsyncCommClient is instantiated in its rawest form. It is configured with a Mail and OnConnect callback and set free with a call to Run. Note that 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. The CMakeLists.txt file for this example is also given below.

Listing 1: A simple example using MOOSAsyncCommClient

```
/*
 * ExampleA.cpp
 *
 * Created on: Nov 10, 2012
 * Author: pnwman
 */
#include "MOOS/libMOOS/Comms/MOOSAsyncCommClient.h"
#include "MOOS/libMOOS/Thirdparty/getpot/getpot.h"

bool OnConnect(void * pParam)
{
    CM00SCommClient* pC = reinterpret_cast<CM00SCommClient*> (pParam);
    pC->Register("X",0.0);
    return true;
}

bool OnMail(void *pParam)
{
    CM00SCommClient* pC = reinterpret_cast<CM00SCommClient*>(pParam);

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

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

    return true;
}

int main(int argc, char * argv[])
{
    //understand the command line
    GetPot cl(argc,argv);
    std::string db_host = cl.follow("localhost",2,"-s","--server");
    int db_port = cl.follow(9000,2,"-p","--port");
    std::string my_name = cl.follow("exampleA",2,"-n","--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;
}

```

Listing 2: CMakeLists.txt to the simple example above

```

#this builds an example program
set(EXECNAME comms_example)

find_package(MOOS 10)

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

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

```

## 6.2 The Simplest Example using CMOOSApp

We can of course achieve the same thing by subclassing CMOOSApp. The code listing below shows how.

Listing 3: A simple example using MOOSAsyncCommClient

```

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

/*
 * ExampleB.cpp
 *
 * Created on: Nov 11, 2012
 * Author: pnwman
 */

class ExampleBApp : 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[])
{
    //understand the command line
    GetPot cl(argc,argv);
    std::string db_host = cl.follow("localhost",2,"-s","--server");
    int db_port = cl.follow(9000,2,"-p","--port");
    std::string my_name = cl.follow("exampleA",2,"-n","--name");

    ExampleBApp App;

    App.Run(my_name.c_str(),"Mission.moos");

    return 0;
}

```

### 6.3 Sharing Video Rate Data

Here is a simple example code for sharing video data using the package OpenCV <sup>3</sup>. 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 :

<sup>3</sup>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 cmake gui as Cmake does not look in /opt by default.

– ./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 your camera output appearing in two windows with very little lag.

Listing 4: Example code to build a camera sharing example

```
#include "opencv2/opencv.hpp"
#include "MOOS/libMOOS/App/MOOSApp.h"
#include "MOOS/libMOOS/Thirdparty/getpot/getpot.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()
    {
        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;
    }
}
```

```

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())*1e3<<" 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[])
{
    GetPot cl(argc,argv);
    CameraApp App;
    App.SetCommandLineParameters(argc,argv);
    App.Run("CameraTest");
    return 0;
}

```

Listing 5: 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 camera_example)

#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.
2. The way in which in this example `SetIterateMode` is used to make the application respond quickly to the reception of mail.

## 7 Under the hood

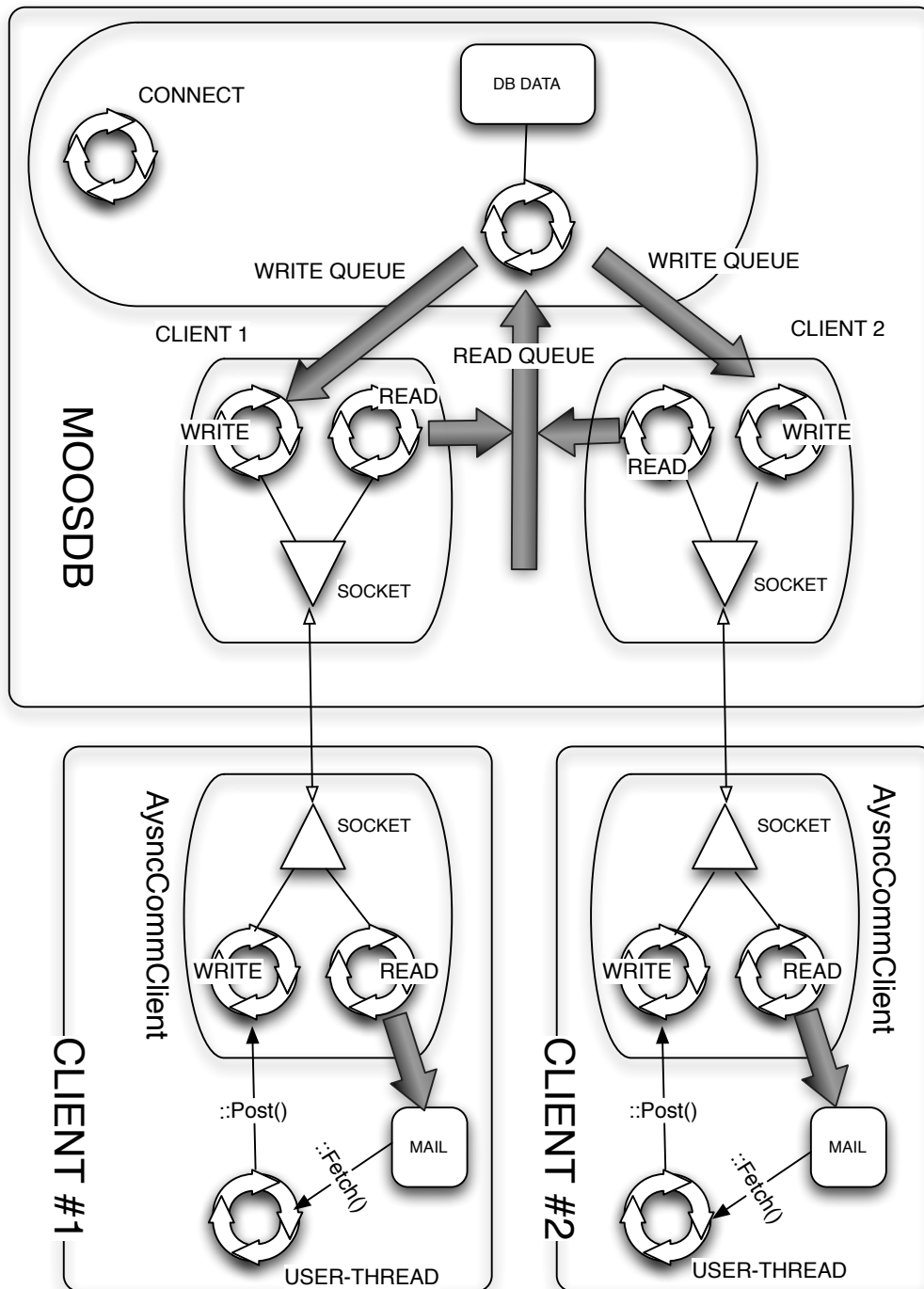


Figure 6