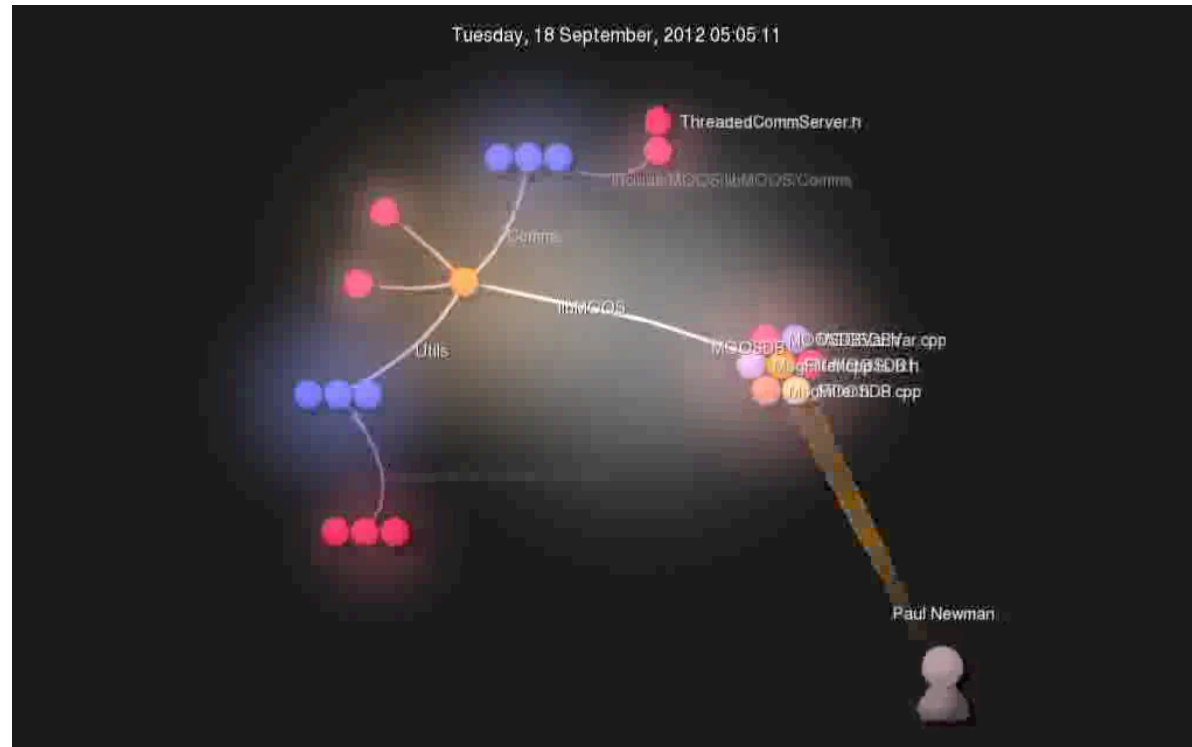
MOOS::V10

Paul Newman
BP Professor of Information Engineering
University of Oxford

www.themoos.org



MOOS::V10 is on github

It is still small

Language	files	blank	comment	code
C++	66	4733	3528	13382
C/C++ Header	122	2871	4935	7265
CMake	8	133	55	292
MATLAB	1	7	0	21
SUM:	197	7744	8518	20960

and has no non-system dependencies

What was wrong?

- DB was single threaded : dodgey comms held everyone up
- Clients' push and pull was coupled
- Registration was cumbersome
- No useful callback mechanisms on message reception
- Shocking directory structure

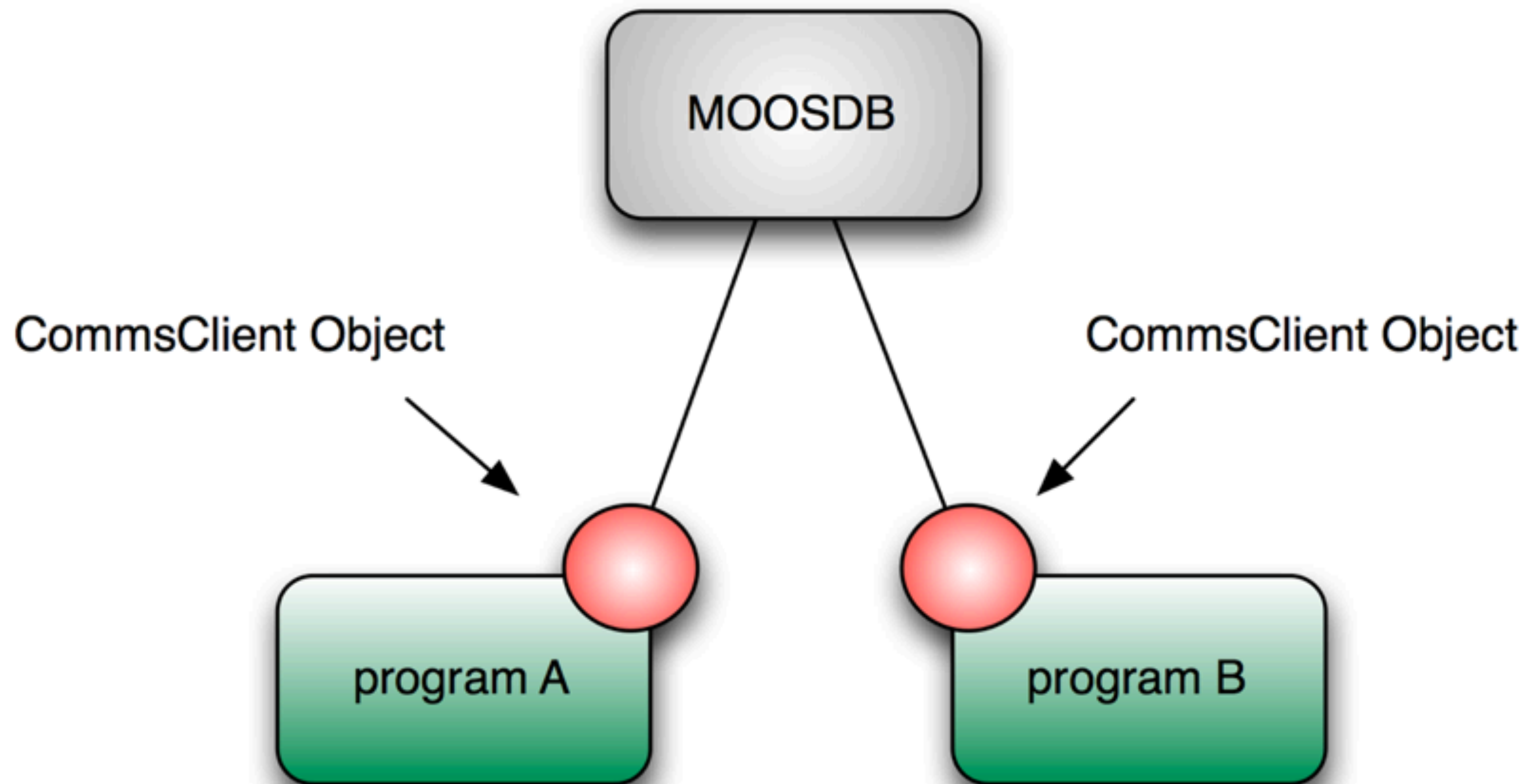
Part 1

Faster, more responsive communications

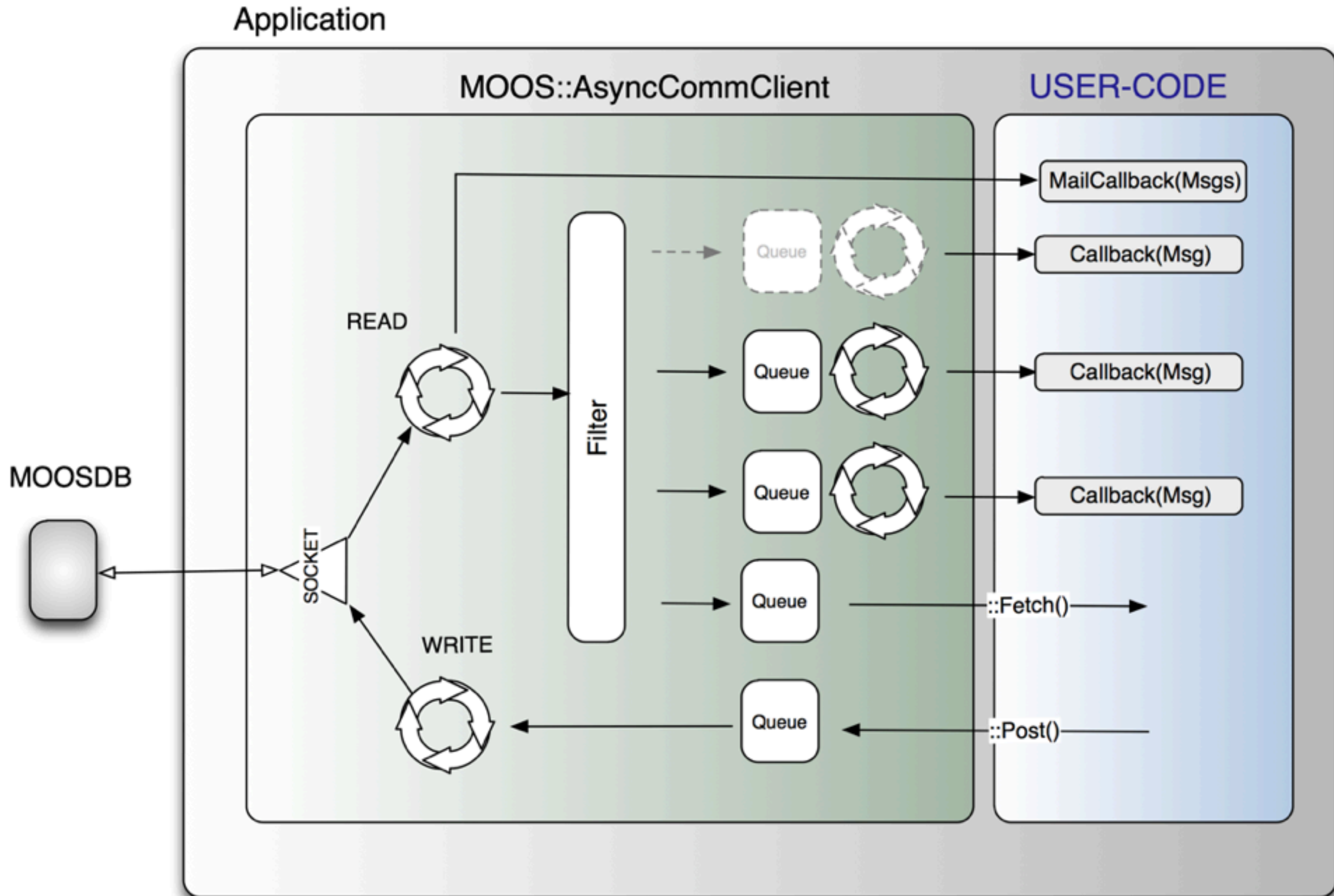
Most Common Problem's

- in bad networks large latency
- point to point comms was synchronous and a bit slow
- comms was not active - it was kind-a-passive

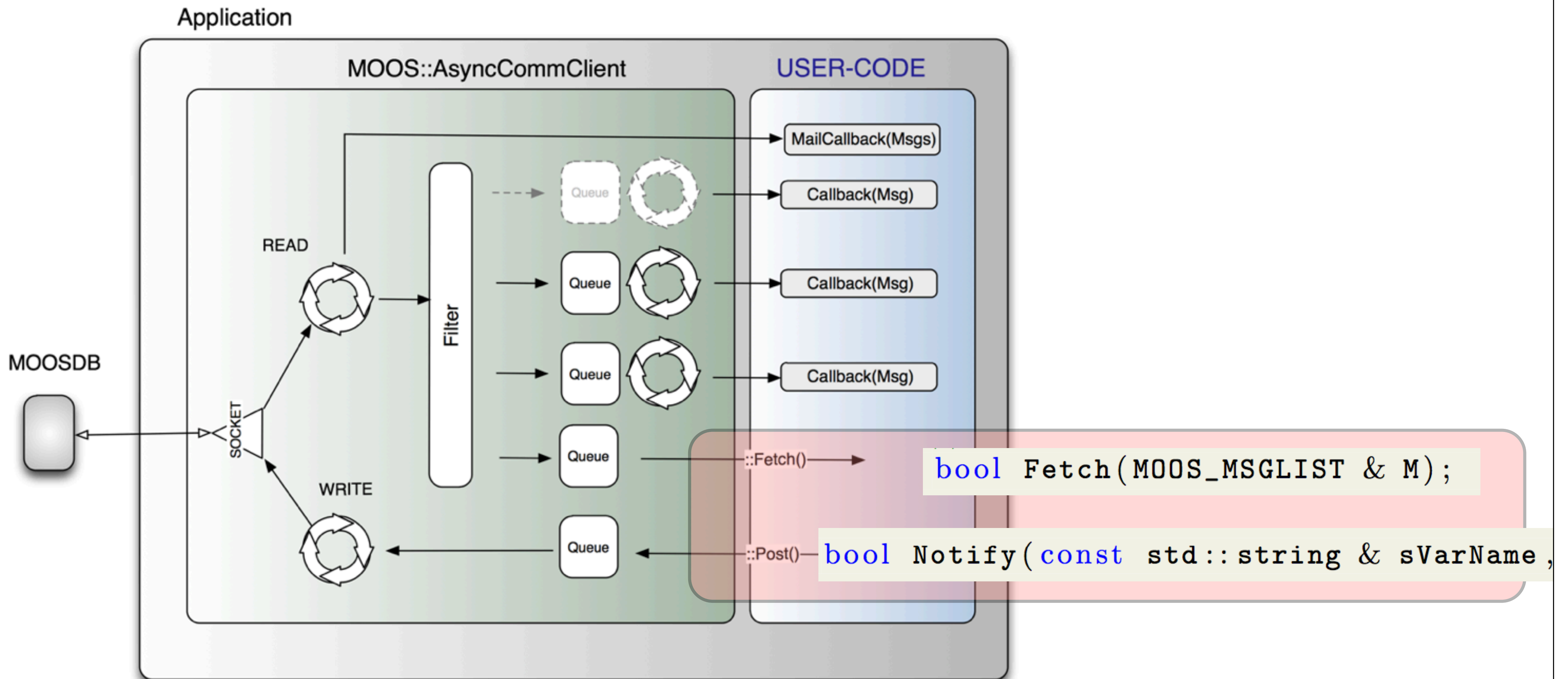
AsyncCommClient



A Threading Model

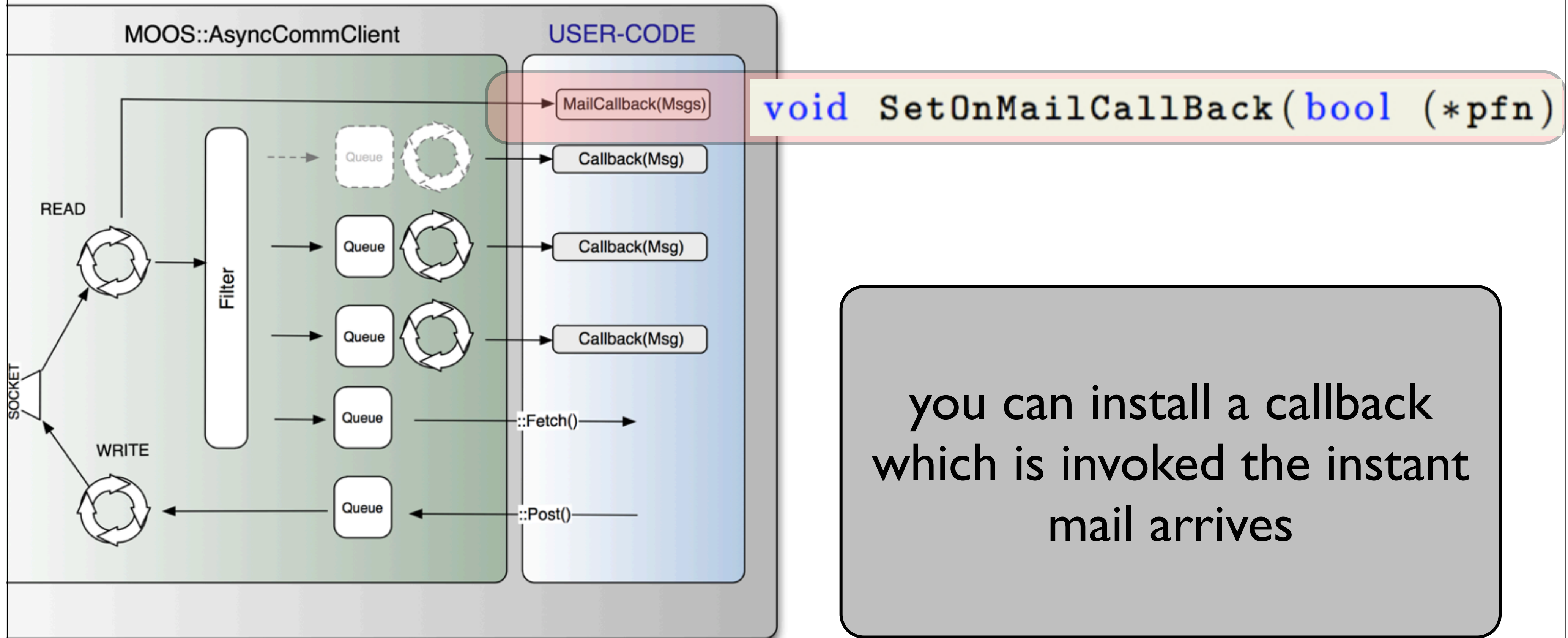


Old API Preserved



..but it is much more zippy....(behind the scenes)

Accessing Zippyness



you can install a callback which is invoked the instant mail arrives

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

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

//this is a mail callback - it is called as soon as mail arrives
bool OnMail(void *pParam){
    CMOOSCommClient* pC = reinterpret_cast<CMOOSCommClient*>(pParam);

    //grab all the held mail
    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();//print it
    }
    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;
}
```

called every time a message arrives

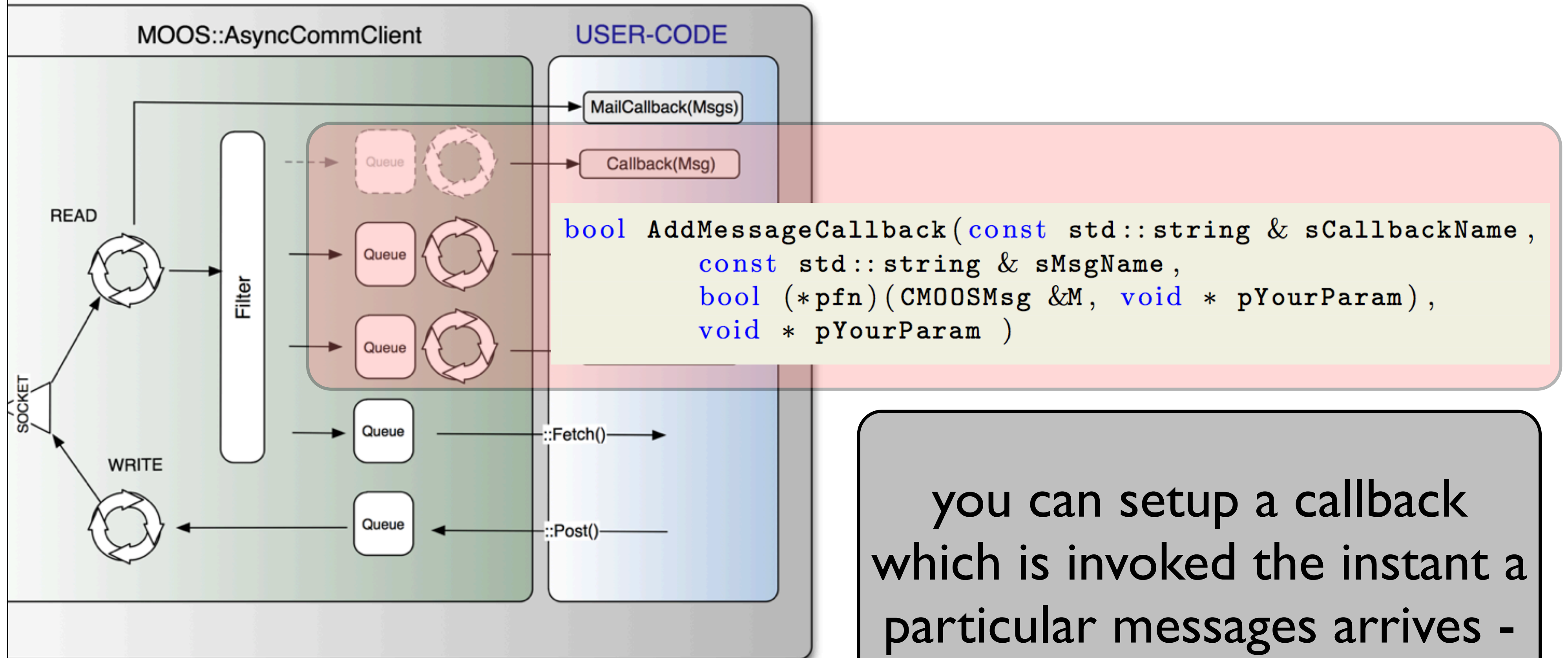
Install a callback

expect <0.1ms latency

OK, simple but crude

- All mail goes through same route
- we have “cloggable” pipleline
- callback in read() of client.

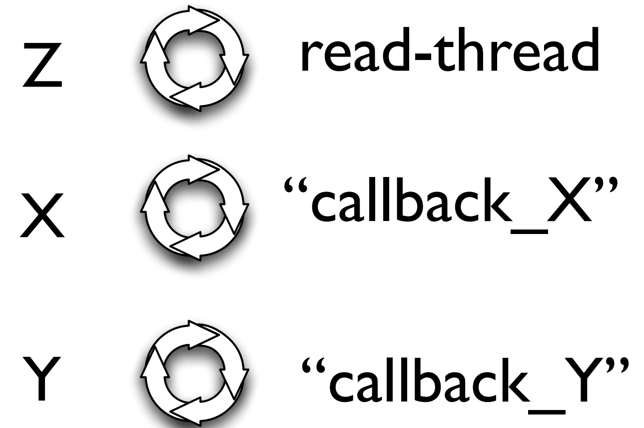
Active Queues




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

you can setup a callback which is invoked the instant a particular messages arrives - *in dedicated thread*

```
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){
     //  .....extra code here....
      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[]){
     //  .....extra code here....
      //configure the comms
      MOOS::MOOSAsyncCommClient Comms;
      Comms.SetOnMailCallBack(OnMail,&Comms);
      Comms.SetOnConnectCallBack(OnConnect,&Comms);

      //here we add per message callbacks
      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;
}
```



Active Queues are good and offer a very flexible mechanism. No clogging.

```
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[]){

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

      //here we add per message callbacks
      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;
}
```

everything other than “V1” ends up here...

Use the “*” queue to have all mail not caught by other named active queues handled in a comms-independent thread

Final Notes on AQ's

- You don't need a queue per message. Any number can be sent to a given named queue
- you can send a message to multiple queues
- there is +1 copy per queue
- you can easily forget that you are in thread land.....

Part 2

Wildcard Registration

?attern Matching Registration

- ...give me anything from pHelm.
- ...give me all data from pHelm which has a name ending in "*jelly*"
- ...just give me *everything*
- ...send me all messages 4 char long with "t" as the 3rd character from any source with "Lionel" in its name

Server-side subscriptions

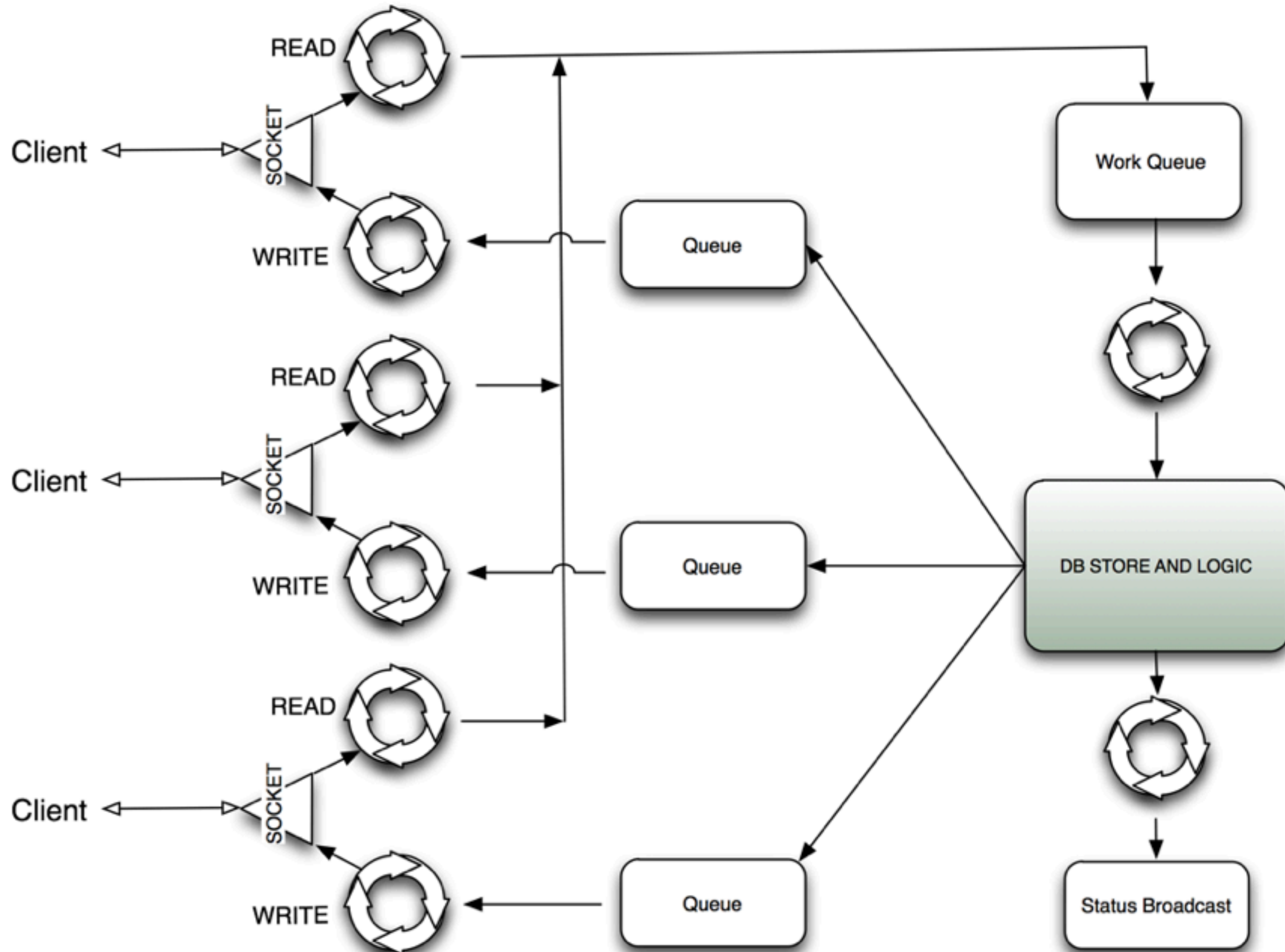
- previously clients had to do dynamic registration by looking for variable and client summaries.
- now the DB will do it for you.
- You register a pattern and as variables appear that match they will be pushed to you.

```
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);
}

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

    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");

}
```

Use wildcard registration when you don't know the detail of what you want upfront. Or if you are lazy.

upfront. Or if you are lazy.

Other Side of the Coin
DB-Threading


MOOSDB::V10
READ
Client
SOCKET
WRITE
READ
Client
SOCKET
WRITE
READ
Client
SOCKET
WRITE
Queue
Queue
Queue
Work Queue
DB STORE AND LOGIC
Status Broadcast

Preventing Excessive Zeal

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

write at no more than

- 50Hz to any client beginning with “camera” followed by two letters
- 100Hz to a client called “VisualOdometry”
- 50Hz for everyone else

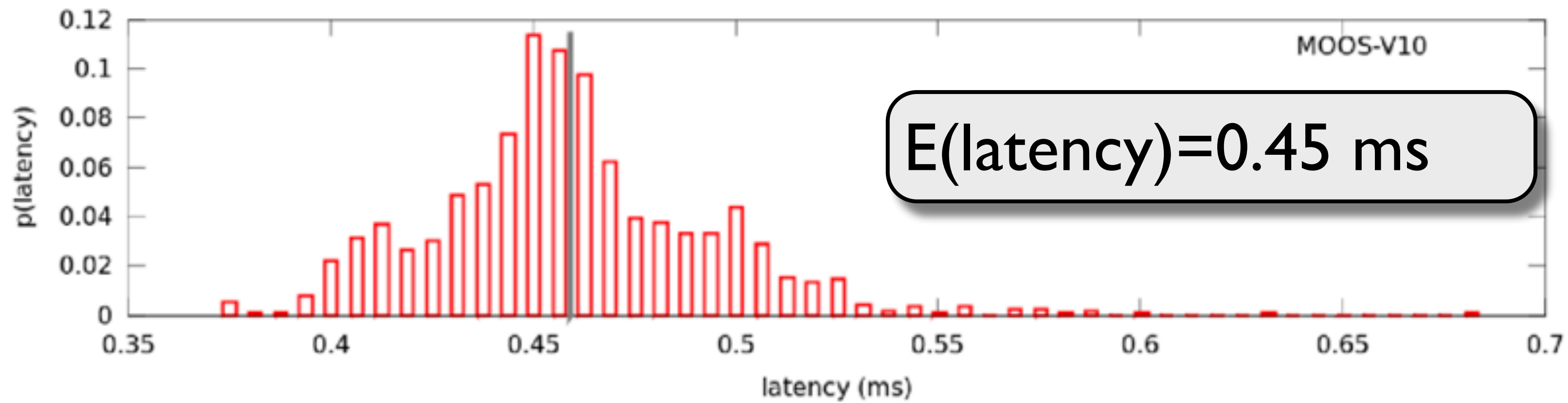
flexible way to limit IO bandwidth on a per client basis

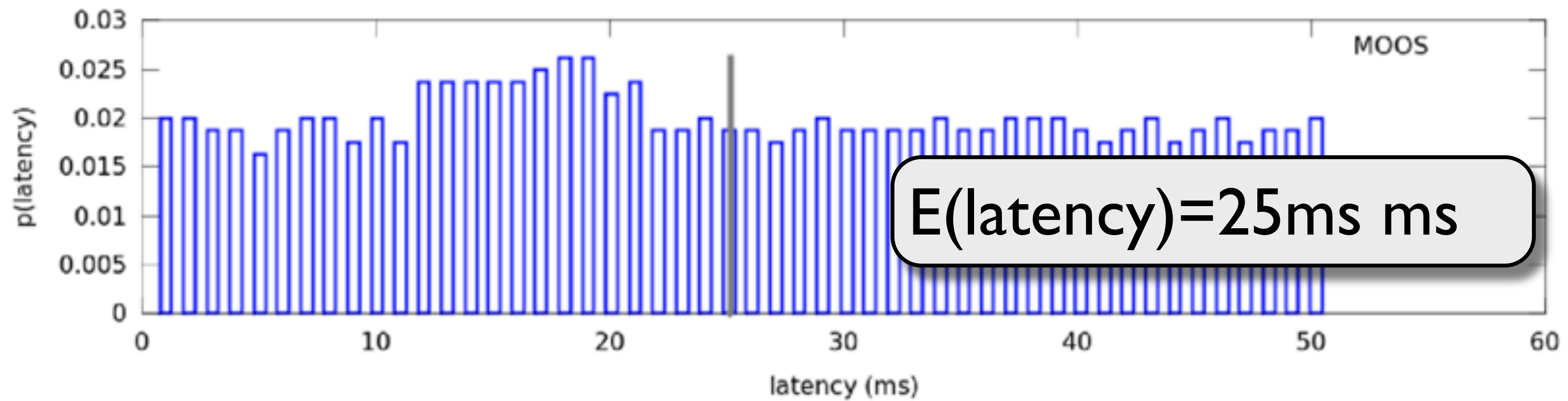
DB Control

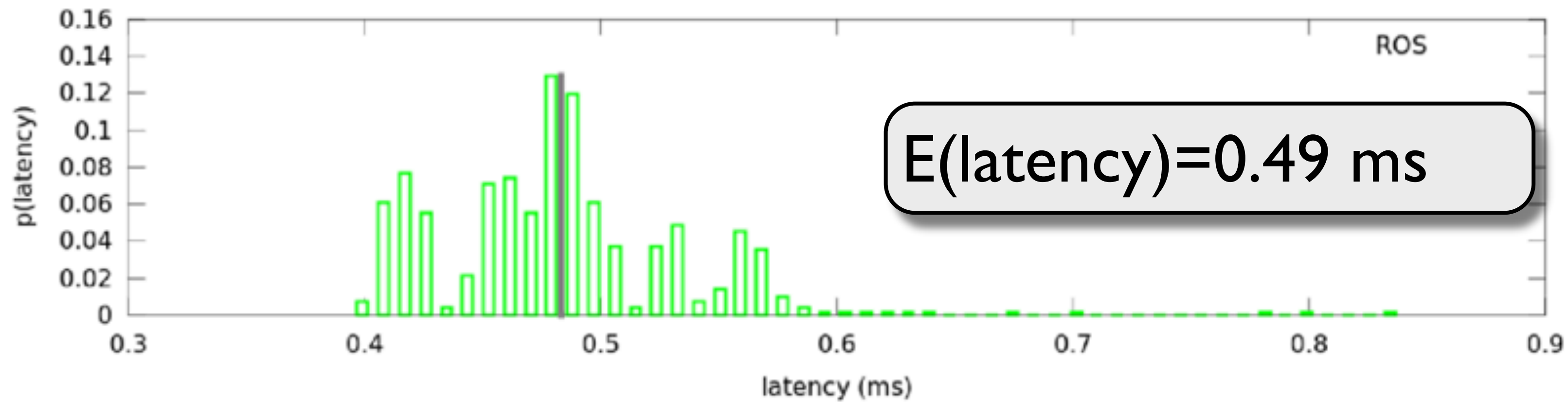
```
➜  bin  ./MOOSDB --help

MOOSDB command line help:

Common MOOS parameters:
--moos_file=<string>                specify mission file name (default mission.moos)
--moos_port=<positive_integer>      specify server port number (default 9000)
--moos_time_warp=<positive_float>   specify time warp
--moos_community=<string>           specify community name

DB Control:
-d     (--dns)                      run with dns lookup
-s     (--single_threaded)          run as a single thread (legacy mode)
-b     (--moos_boost)               boost priority of communications
--moos_timeout=<positive_float>     specify client timeout
--response=<string-list>            specify tolerable client latencies in ms
--warning_latency=<positive_float>     specify latency above which warning is issued in ms
--tcpnodelay                        disable nagle algorithm
--webserver_port=<positive_integer> run webserver on given port
--help                              print help and exit

example:
  ./MOOSDB --moos_port=9001
  ./MOOSDB --moos_port=9001 --response=x_app:20,y_app:100,*_instrument:0
➜  bin
```

Part 3

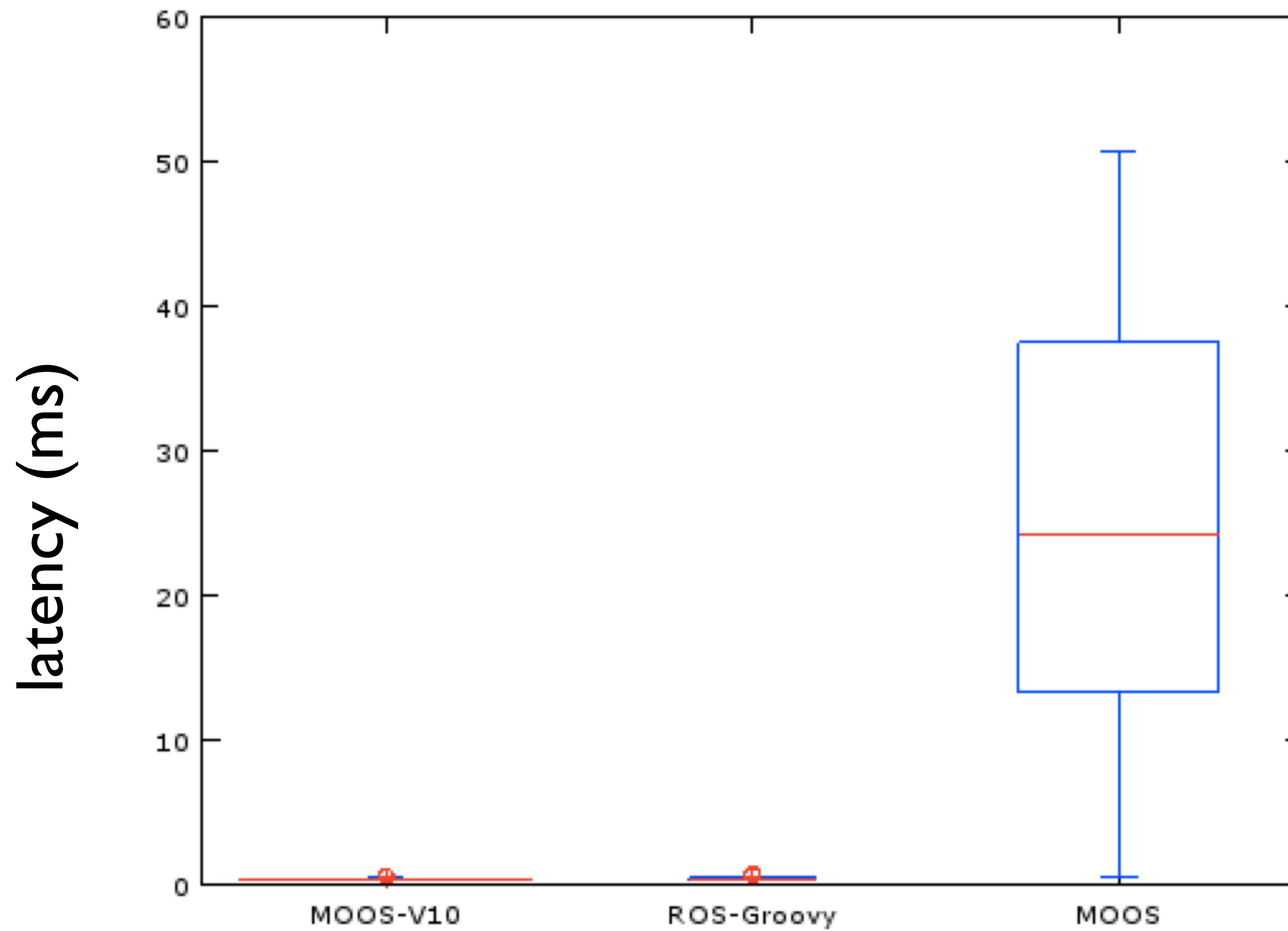
Performance
Does it Make a Difference?



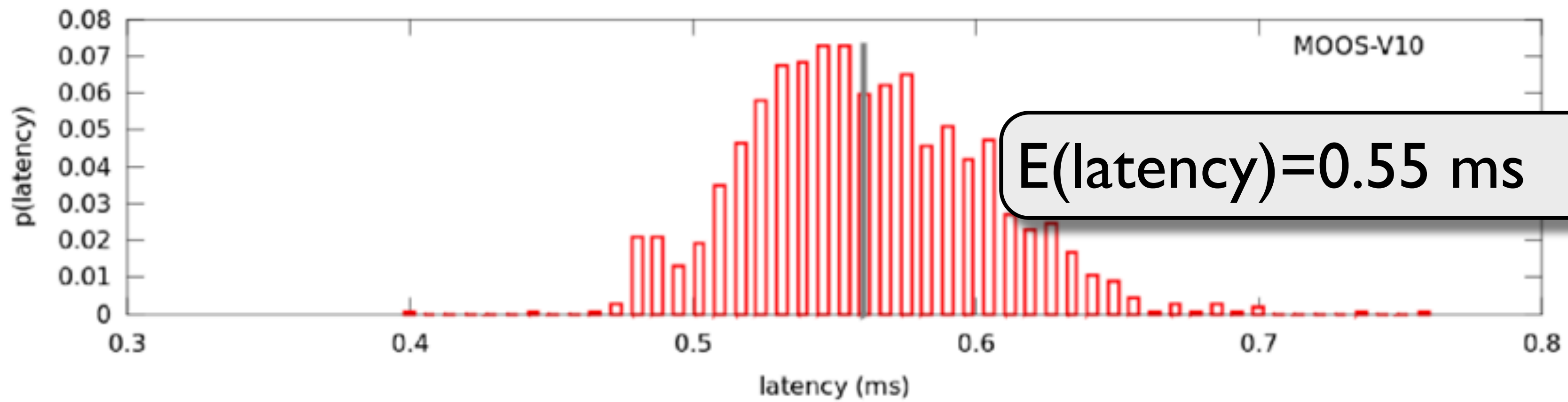


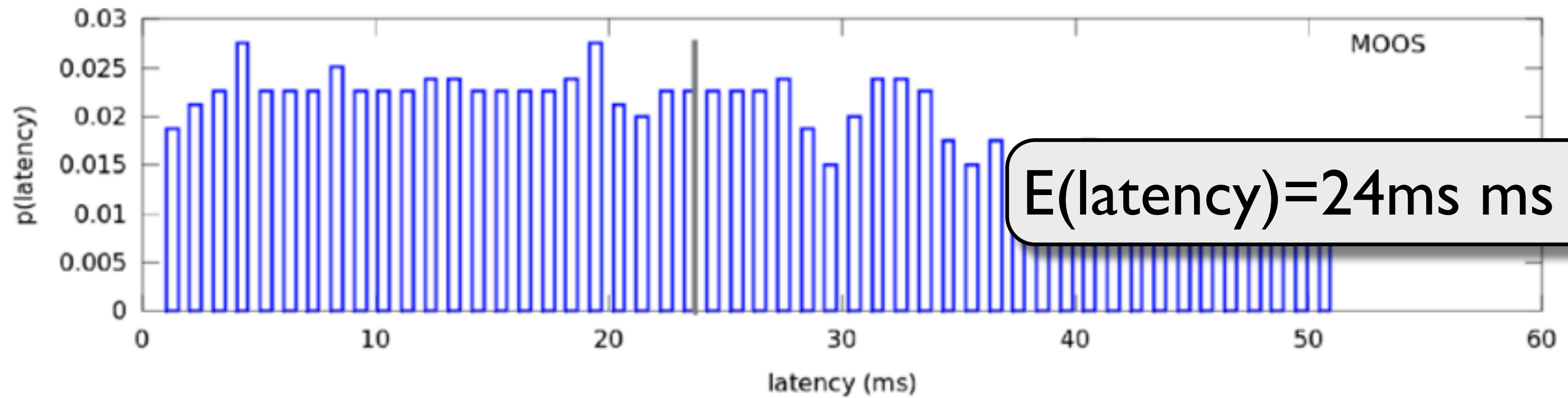


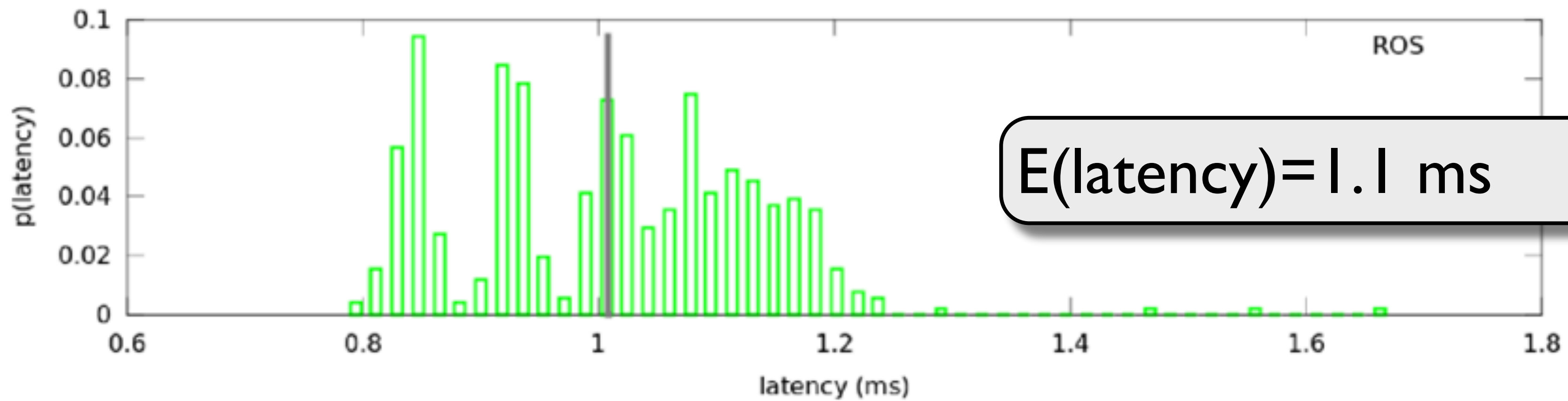
1KB@20Hz sent to 5 Clients



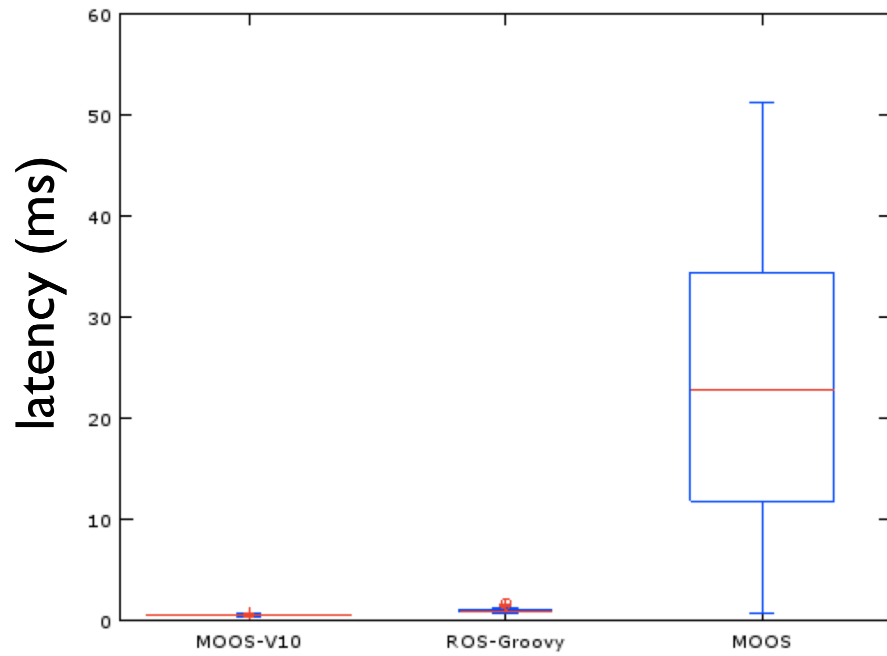
1KB@20Hz sent to 5 Clients

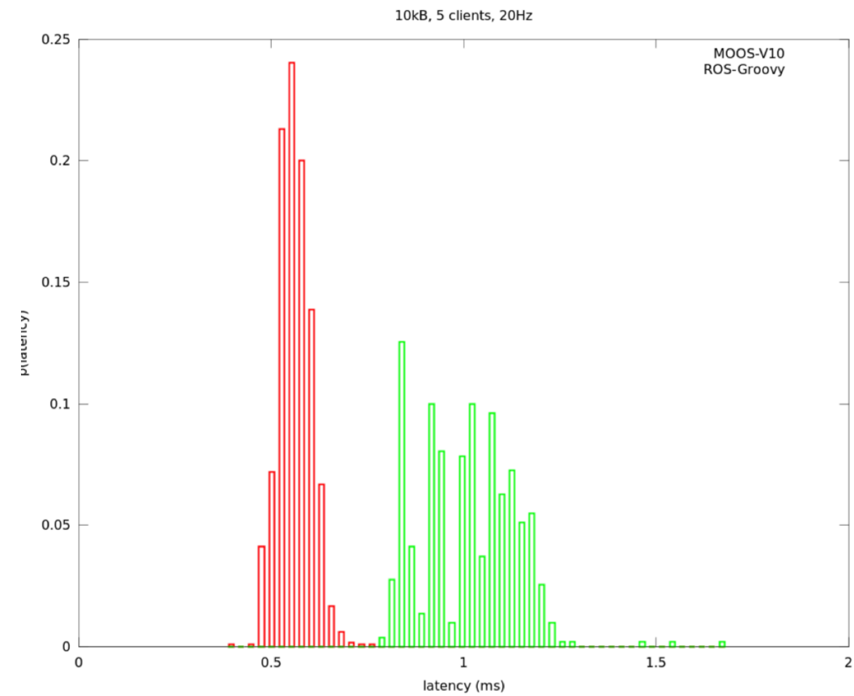




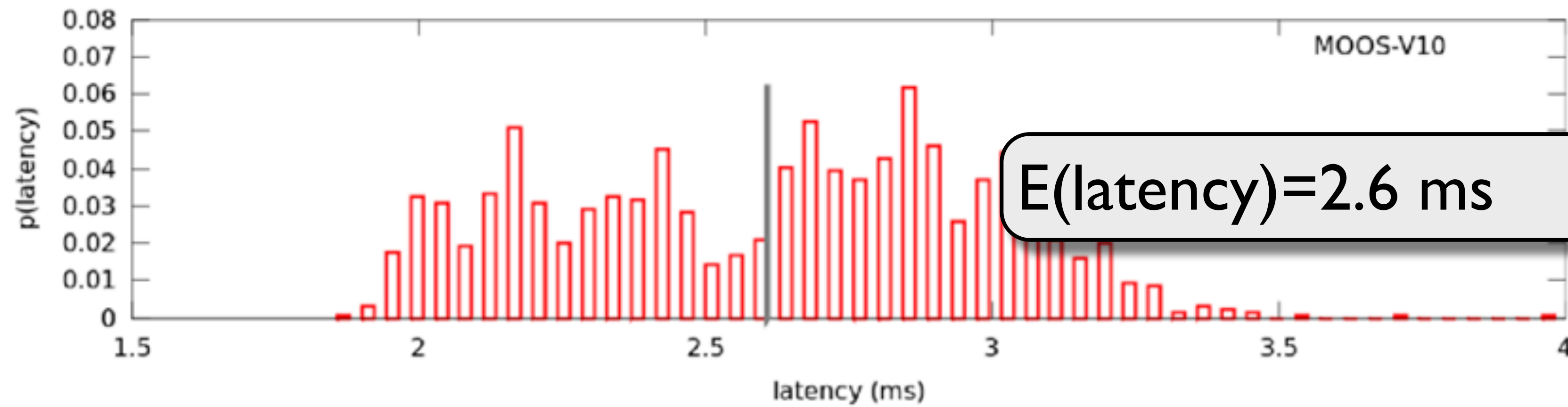


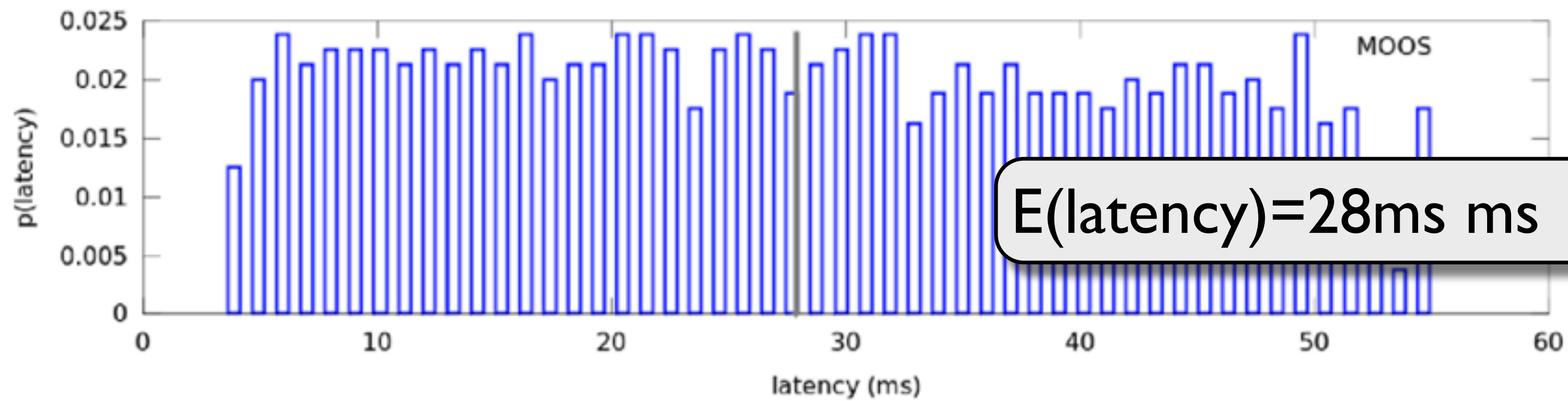
10KB@20Hz sent to 5 Clients

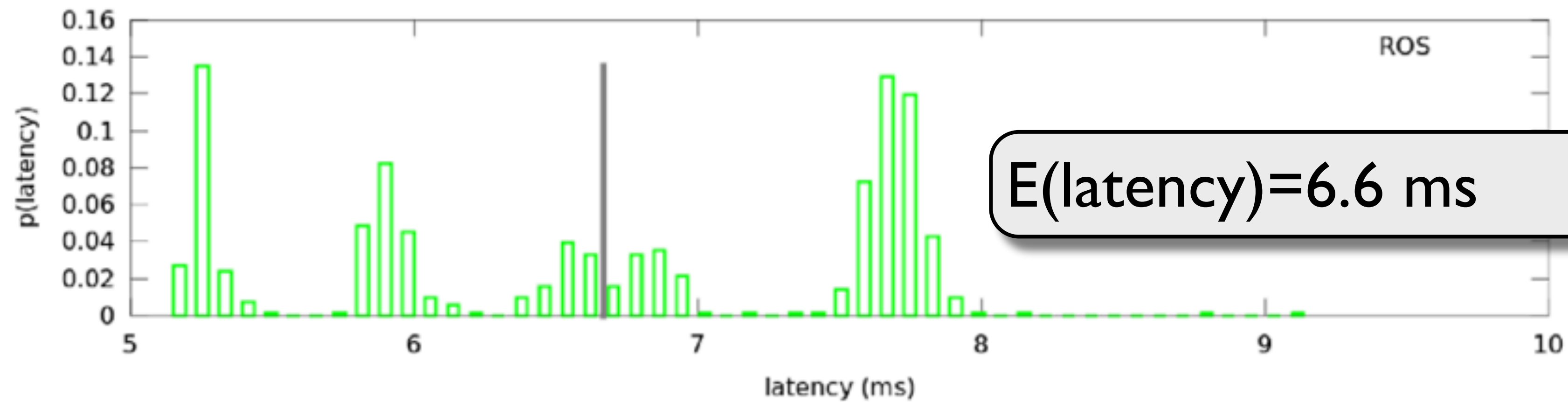




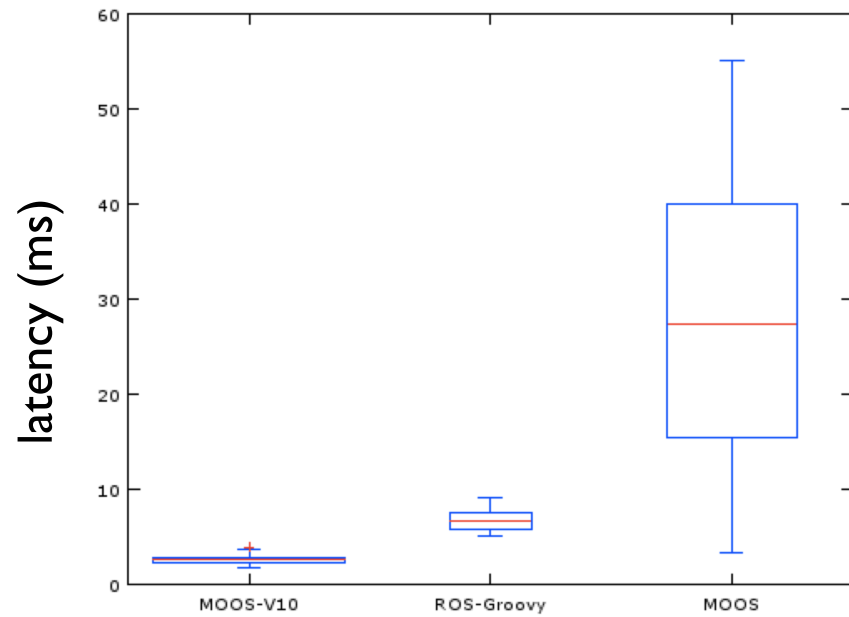
10KB@20Hz sent to 5 Clients

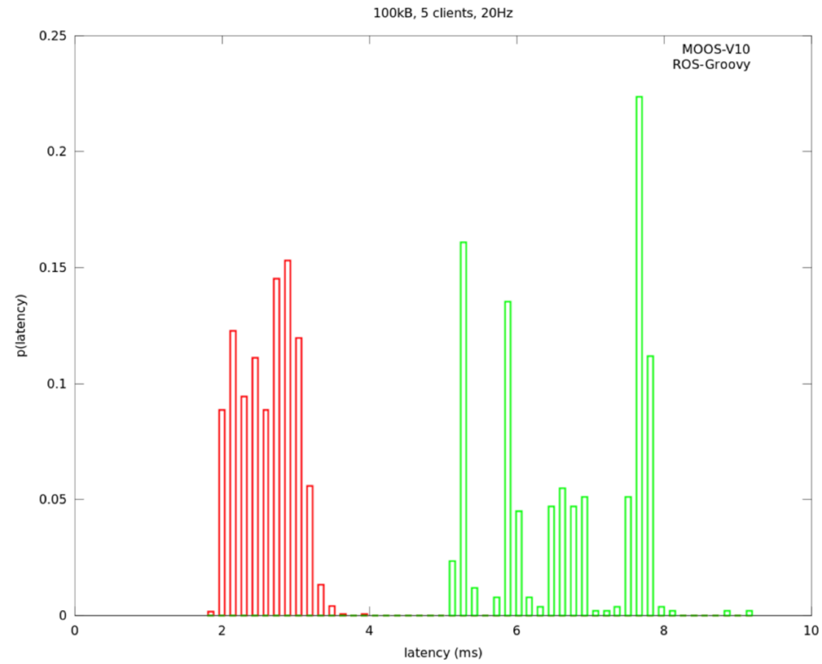




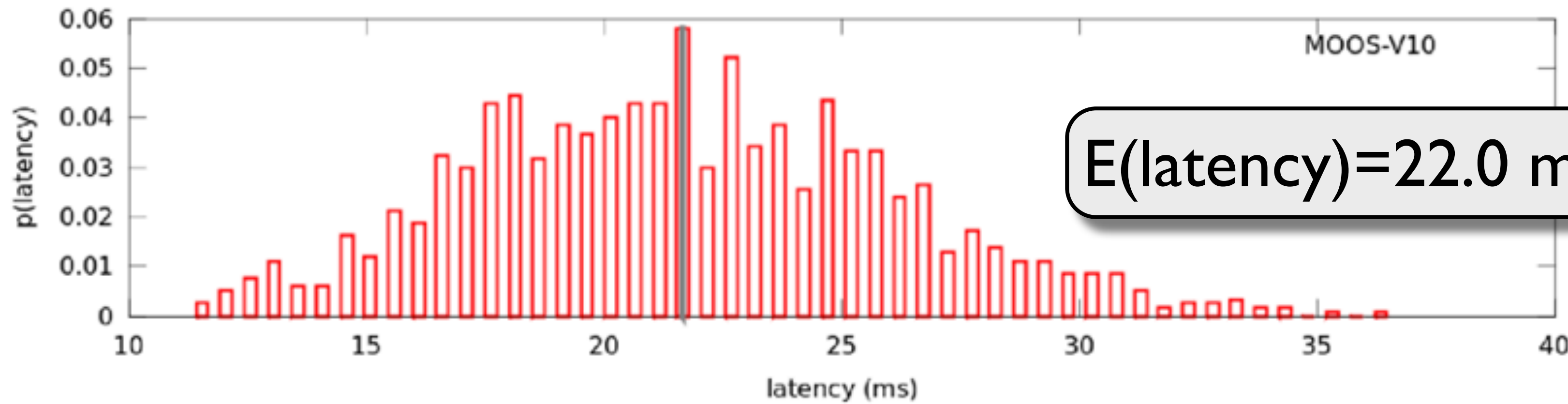


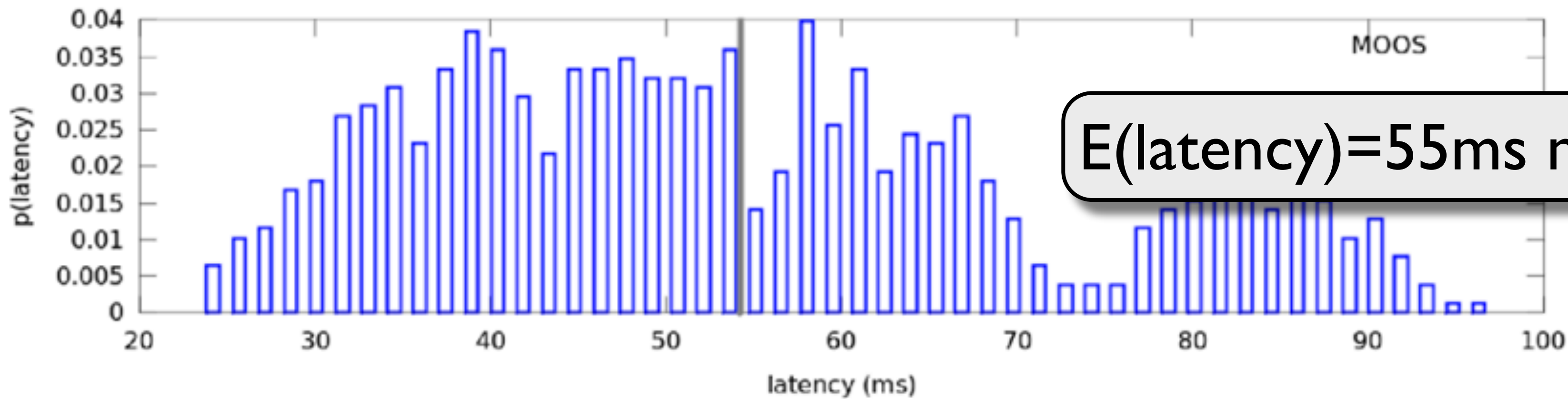
100KB@20Hz sent to 5

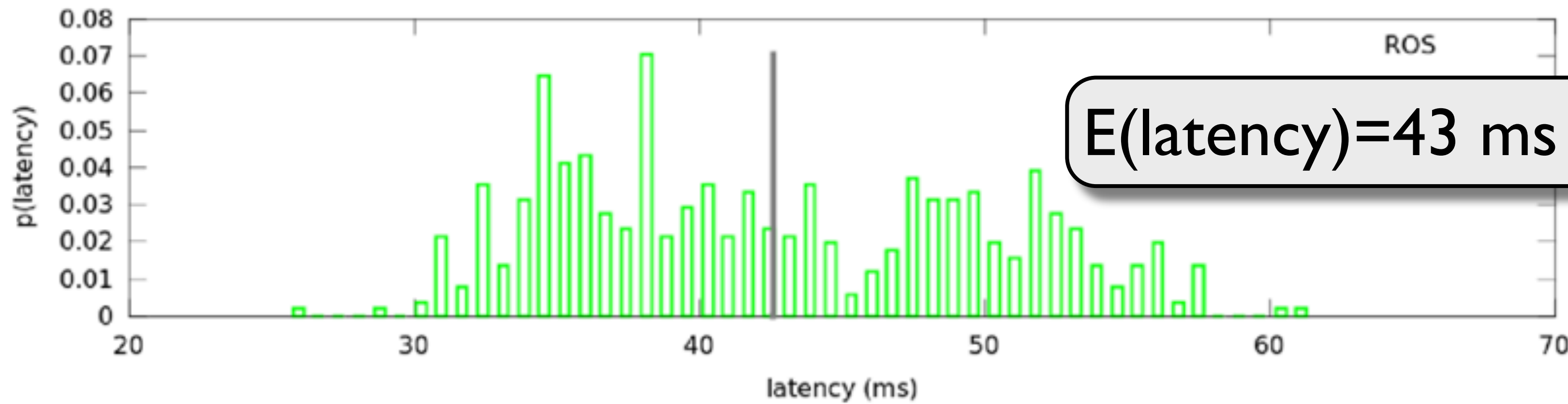




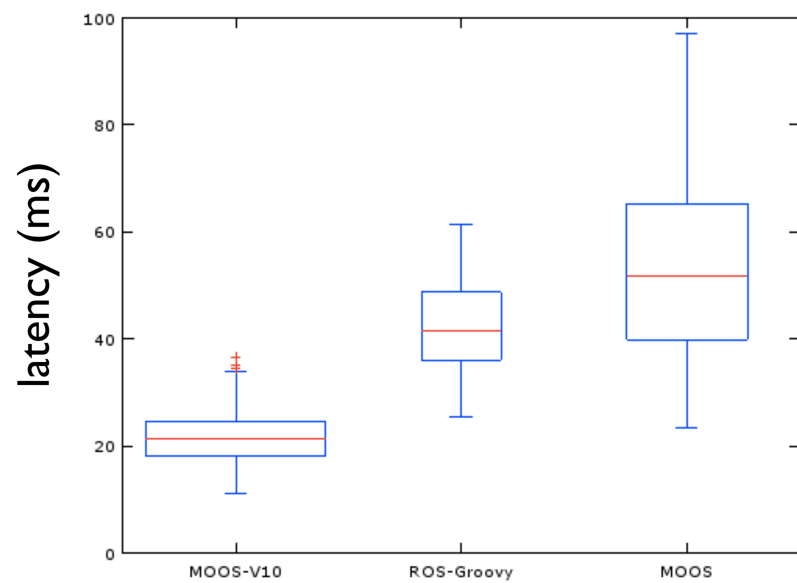
100KB@20Hz sent to 5 Clients

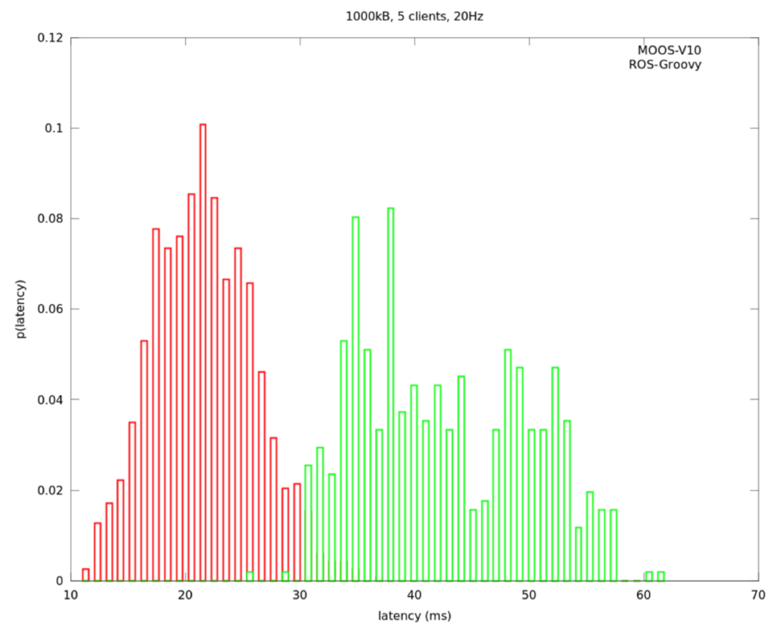






1MB@20Hz sent to 5 Clients





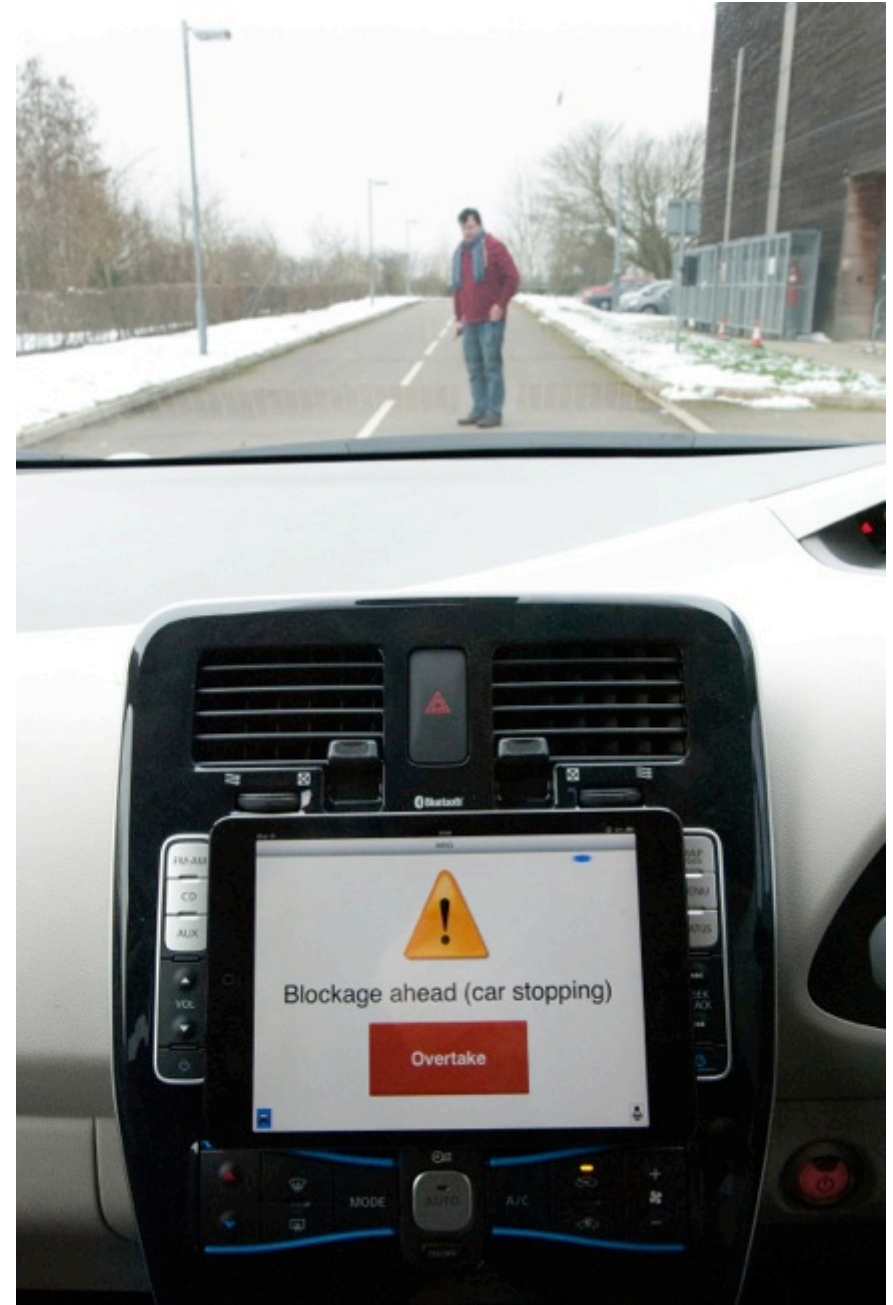
1MB@20Hz sent to 5 Clients

Part 4

Is It Reliable ?

Considerations

- There is a good deal of new code (but you can revert to old code with switches!)
- Performance not formally verified
- Unit tests are multiplying
- And we rely on it to run some pretty demanding projects....



Part5

Structure, Building and Using

core-moos Structure

```
➜ CoreMOOS git:(devel) tree -d -L 4
.
├── Core
|   ├── MOOSDB
|   |   └── testing
|   |       └── matlab
|   ├── libMOOS
|   |   ├── App
|   |   |   └── include
|   |   ├── Comms
|   |   |   └── include
|   |   ├── Thirdparty
|   |   |   ├── AppCasting
|   |   |   ├── PocoBits
|   |   |   └── getpot
|   |   ├── Utils
|   |   |   └── include
|   |   └── include
|   |       └── MOOS
|   └── tools
|       └── umm
└── cmake
```

#include “MOOS/libMOOS/<MODULE>/<header>.h”

Comms
Utils
Apps
Thirdparty

Now one single library: libMOOS.a

Compiling and Using

```
#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})
```

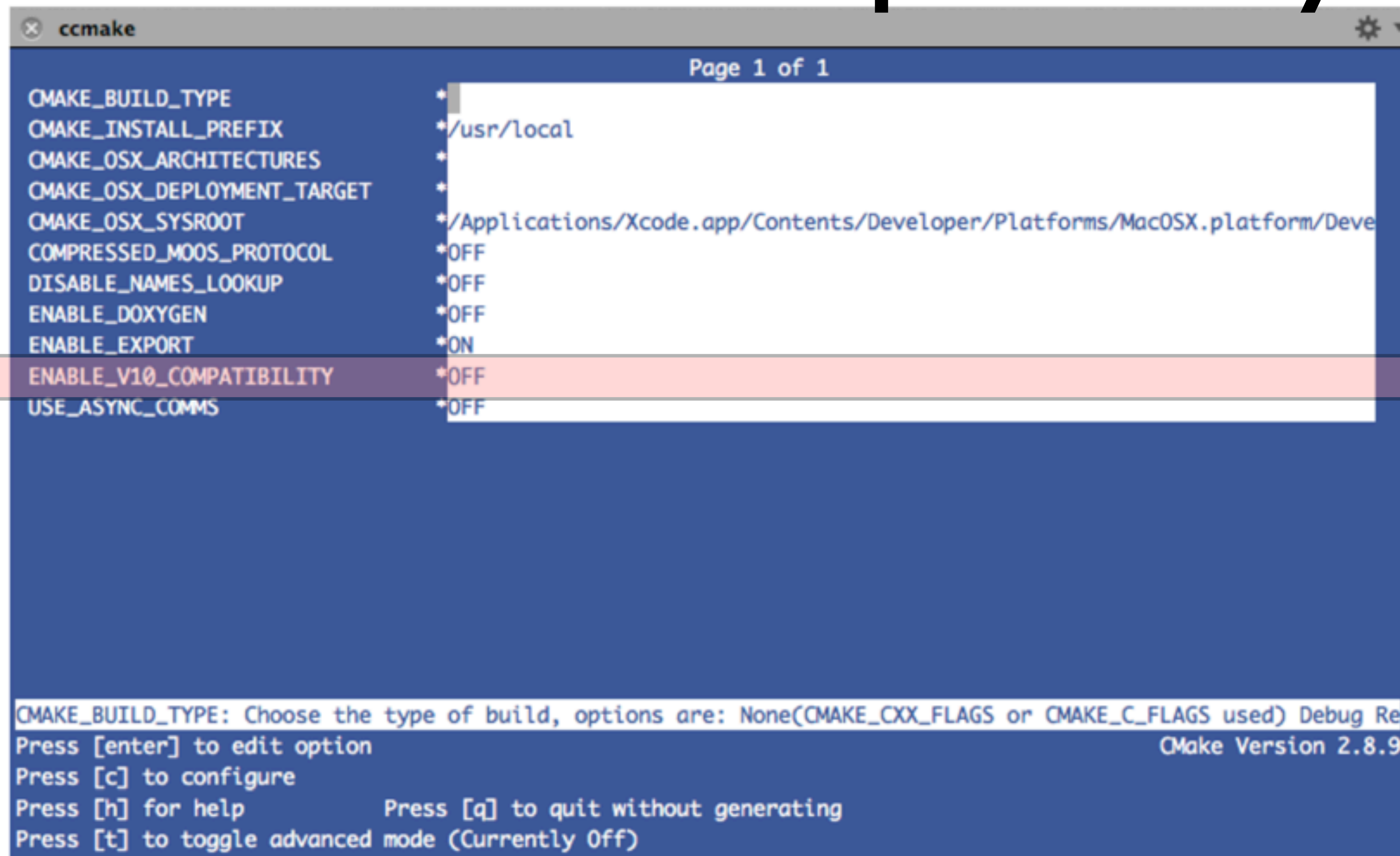
build properties and location discovered automagically

Binary Compatibility

Client	MOODB	OK	Async. Comms	Synch. Comms	Multithreading DB	Single Threaded DB
Pre10	Pre10	✔	✗	✔	✗	✔
Pre10	V10	✔	✗	✔	✔	✔
V10	Pre V10	✔	✗	✔	✗	✔
V10	V10	✔	✔	✔	✔	✔

Designed so legacy binaries can work with upgraded binaries

Source Compatibility



Designed so legacy source can leverage V10 with zero code change. But this is a lazy thing.....

Part 6

MOOSApp++

CMOOSApp Revisited

All the communications upgrades are available...and more

App::OnMessage

```
/** called by a separate thread if a callback
 * has been installed by calling AddMessageCallback()*/
virtual bool OnMessage(CMOOSMsg & M);


/**
 * Add a callback to ::OnMessage() for a particular message. Th
 * as soon as a message named as sMsgName arrives. You do need
 * remember OnMessage could be called simultaneously by N threa
 * @param sMsgName
 * @return true on success
 */
bool AddMessageCallback(const std::string & sMsgName);
```

Have ::OnMessage called for each registered message in a seperate thread. Simple AsyncComms in CMOOSApp

And Also Your Own CB

```
/**
 * Register a custom call back for a particular message. This call back
 * will be called from its own thread.
 * @param sMsgName name of message to watch for
 * @param pfn  pointer to your function should be type
 * bool func(CMOOSMsg &M, void *pParam)
 * @param pYourParam a void * pointer to the thing we want passed as pParam above
 * @return true on success
 */
bool AddCustomMessageCallback(const std::string & sCallbackName,
        const std::string & sMsgName,
        bool (*pfn)(CMOOSMsg &M, void * pYourParam),
        void * pYourParam );
```

Active Queues are thus exposed to CMOOSApp

Controlling App Flow

```
//enumeration of ways application can iterate
enum IterateMode
{
    REGULAR_ITERATE_AND_MAIL=0,
    COMMS_DRIVEN_ITERATE_AND_MAIL,
    REGULAR_ITERATE_AND_COMMS_DRIVEN_MAIL,
}m_IterationMode;

//set up the iteration mode of the app
bool SetIterateMode(IterateMode Mode);
```

The old way....

Three ways to control *OnNewMail* and *Iterate* behaviour

Event Driven + LockStep

```
//enumeration of ways application can iterate
enum IterateMode
{
    REGULAR_ITERATE_AND_MAIL=0,
    COMMS_DRIVEN_ITERATE_AND_MAIL,
    REGULAR_ITERATE_AND_COMMS_DRIVEN_MAIL,
}m_IterationMode;

//set up the iteration mode of the app
bool SetIterateMode(IterateMode Mode);
```

Event Driven
  ::*OnNewMail()*
then always cal
  ::*Iterate()*

Independent Event Driven Mail

```
//enumeration of ways application can iterate
enum IterateMode
{
    REGULAR_ITERATE_AND_MAIL=0,
    COMMS_DRIVEN_ITERATE_AND_MAIL,
    REGULAR_ITERATE_AND_COMMS_DRIVEN_MAIL,
}m_IterationMode;

//set up the iteration mode of the app
bool SetIterateMode(IterateMode Mode);
```

Event Mail
    ::OnNewMail()
only call
    ::Iterate()
when scheduled

New Niceties

```
/** called just before OnStartUp is called ....
 virtual bool OnStartUpPrepare(){return true;};

/** called just after OnStartUp has finished ...
 virtual bool OnStartUpComplete(){return true;};
```

More granularity in execution

```
/** make a status string - overload this in a ....
 virtual std::string MakeStatusString();

/** called before OnStartUp and before ....
virtual bool OnProcessCommandLine();

/** called when command line is asking ....
virtual void OnPrintHelpAndExit();

/** called when command line is asking ....
virtual void OnPrintExampleAndExit();

/** called when command line is asking ....
virtual void OnPrintInterfaceAndExit();

/** called when command line is asking ....
virtual void OnPrintVersionAndExit();
```

New Niceties

```
/** called just before OnStartUp is called ....
 virtual bool OnStartUpPrepare(){return true;};

/** called just after OnStartUp has finished ...
 virtual bool OnStartUpComplete(){return true;};

/** make a status string - overload this in a ....
 virtual std::string MakeStatusString();

/** called before OnStartUp and before ....
virtual bool OnProcessCommandLine();

/** called when command line is asking ....
virtual void OnPrintHelpAndExit();

/** called when command line is asking ....
virtual void OnPrintExampleAndExit();

/** called when command line is asking ....
virtual void OnPrintInterfaceAndExit();

/** called when command line is asking ....
virtual void OnPrintVersionAndExit();
```

Status now automatically includes CPU load information

Command Line Processing

```
/** called just before OnStartUp is called ....
 virtual bool OnStartUpPrepare(){return true;};

/** called just after OnStartUp has finished ...
 virtual bool OnStartUpComplete(){return true;};

/** make a status string - overload this in a ....
 virtual std::string MakeStatusString();

/** called before OnStartUp and before ....
virtual bool OnProcessCommandLine();

/** called when command line is asking ....
virtual void OnPrintHelpAndExit();

/** called when command line is asking ....
virtual void OnPrintExampleAndExit();

/** called when command line is asking ....
virtual void OnPrintInterfaceAndExit();

/** called when command line is asking ....
virtual void OnPrintVersionAndExit();
```

opportunity to capture command line options using MOOS::CommandLineParser

Using the Parser

```
class DBTestClient : public CMOOSApp
{
public:
    DBTestClient(){};

    bool OnProcessCommandLine()
    {

        _bShowLatency =  m_CommandLineParser.GetFlag("-l","--latency");
        _bVerbose = m_CommandLineParser.GetFlag("-v","--verbose");
        _bShowBandwidth =   m_CommandLineParser.GetFlag("-b","--bandwidth");

        if(m_CommandLineParser.GetFlag("--moos_boost"))
        {
            m_Comms.BoostIOPriority(true);
        }

        std::string temp;

        if(m_CommandLineParser.GetVariable("-s",temp))
        {
            _vSubscribe = MOOS::StringListToVector(temp);
        }

        std::vector<std::string> vPublish;
        if(m_CommandLineParser.GetVariable("-p",temp))
        {
            vPublish = MOOS::StringListToVector(temp);
        }
```

Built-in Options

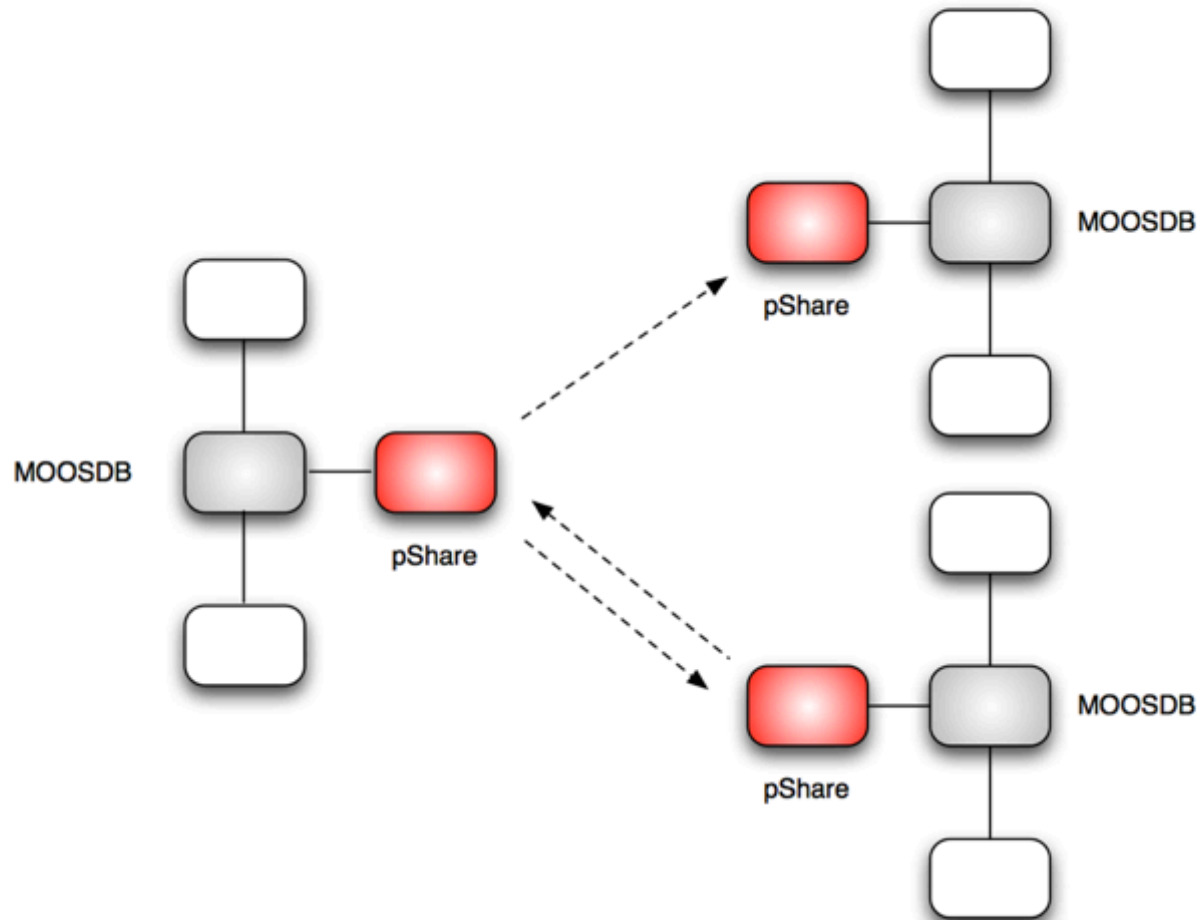
```
variables:
--moos_app_name=<string>        : name of application
--moos_name=<string>            : name with which to register with MOOSDB
--moos_file=<string>            : name of configuration file
--moos_host=<string>            : address of machine hosting MOOSDB
--moos_port=<number>            : port on which DB is listening
--moos_app_tick=<number>        : frequency of application (if relevant)
--moos_max_app_tick=<number>    : max frequency of application (if relevant)
--moos_comms_tick=<number>      : frequency of comms (if relevant)
--moos_iterate_Mode=<0,1,2>     : set app iterate mode
--moos_time_warp=<number>       : set moos time warp

flags:
--moos_iterate_no_comms         : enable iterate without comms
--moos_filter_command           : enable command message filtering
--moos_no_sort_mail             : do not sort mail by time
--moos_no_comms                 : do not start communications
--moos_quiet                    : do not print banner information
--moos_quit_on_iterate_fail     : quit if iterate fails
--moos_no_colour                : disable colour printing

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 help on moos switches
--help                          : print help on moos messages and custom help
```

All Apps inherit and handle these switches

Part 7

Bridging Communities

Sharing with pShare



UDP (inc. multicast) data sharing between communities

Wildcard Aware

```
ProcessConfig = pShare
{
   Output = src_name = X?, route = fancymachine:9021
   Output = src_name = Q?:procA, route = 192.168.4.10:9021
   Output = src_name = W*:*A, route = multicast_7
}
```

Forward any two letter variable beginning with "X" to port 9021 on "*fancymachine*"

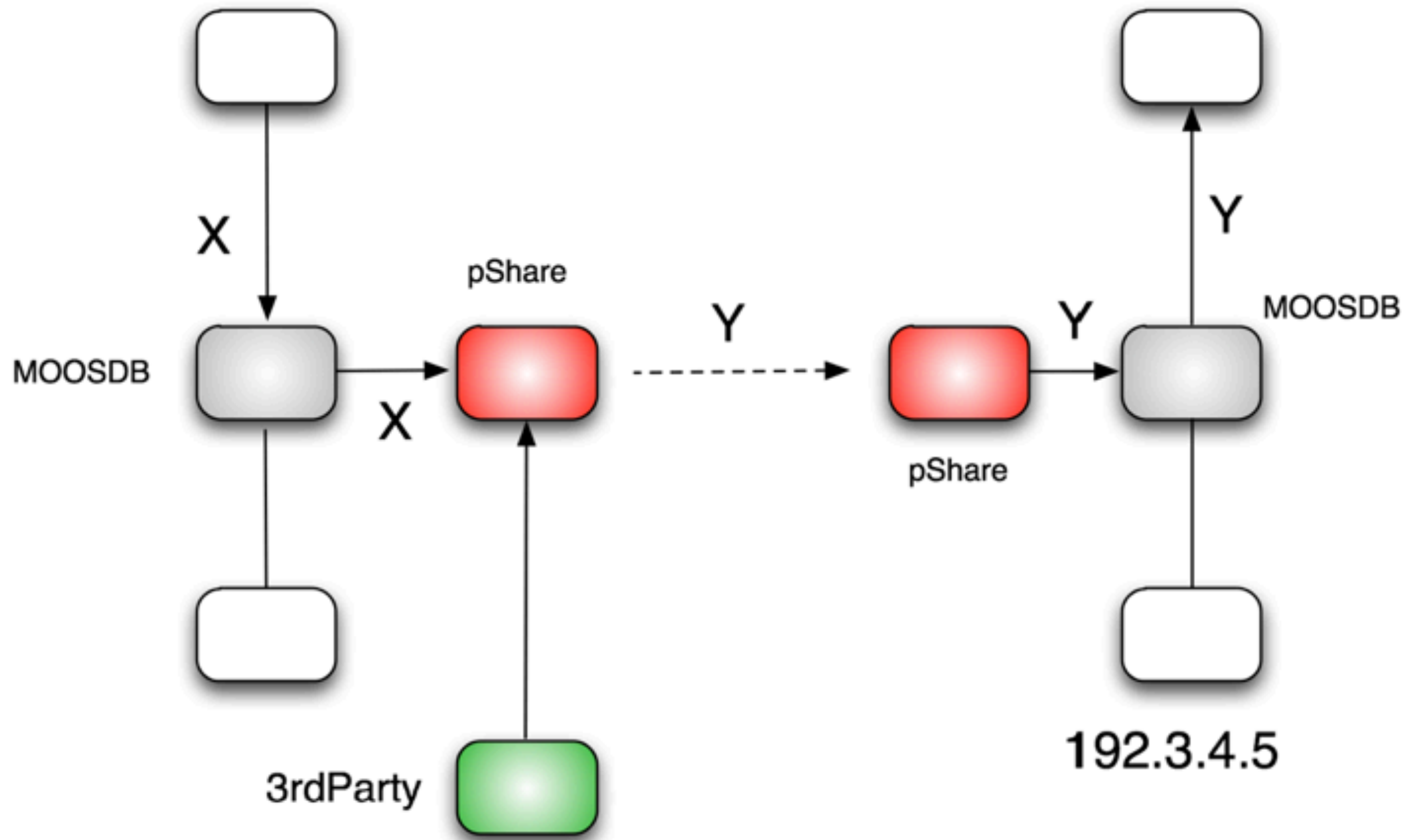
Multicast Forwarding

```
ProcessConfig = pShare
{
    Output = src_name = X?, route = fancymachine:9021
    Output = src_name = Q?:procA, route =multicast_8
    Output = src_name = W∗:∗A, route = multicast_7
}
```

Forward any variable beginning with "*W*" from a client ending in "*A*" to channel multicast_7

channel multicast_7

Dynamic Forwarding



On the fly configuration of sharing

Part 8

And what remains?

What am I working on?

- Unit testing suite
- Application level testing suite
- pAntler revisited
- IOS / Android interfaces
- Rich documentation

Big Thanks To

- Battelle for sponsorship (Rob Carnes)
- Mike Benjamin for making everything happen
- POCO community and F Schaefer (getpot)
- Alon Yaari and Josh Leighton for Beta Testing

Hope it is helpful

www.themoos.org