



MOOS ——Mission Orientating Suite

刘宏坤

- 1 Introduction
- 2 Foundation
- 3 Using the MOOS
- 4 Key MOOS Process



1

Introduction

- MOOS (pronounced “moose”) is an umbrella term applying to a set of communicating applications.
- “MOOS” refers to a suite of libraries and executables designed and proven to run a field robot in sub-sea and land domains.
- The heart of MOOS are its communications API and Library.





Foundation

2.1 Topology

MOOS has a star-like topology. Each application within a MOOS community has a connection to a single “MOOS Database” (called MOOSDB) that lies at the heart of the software suite.

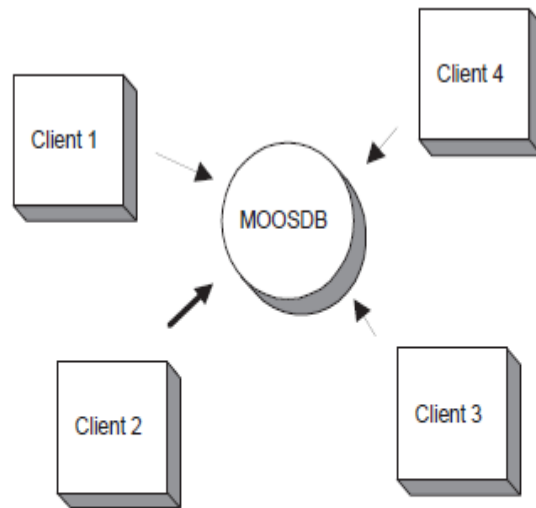


Figure 1: MOOS binds applications into a network with a star-shaped topology. Each client has a single communications channel to a server (MOOSDB).

2.2 Message Content

The communications API in MOOS allows data to be transmitted between MOOSDB and a client. MOOS only allows for data to be sent in string or double form. Data is packed into messages (CMOOSMsg class).

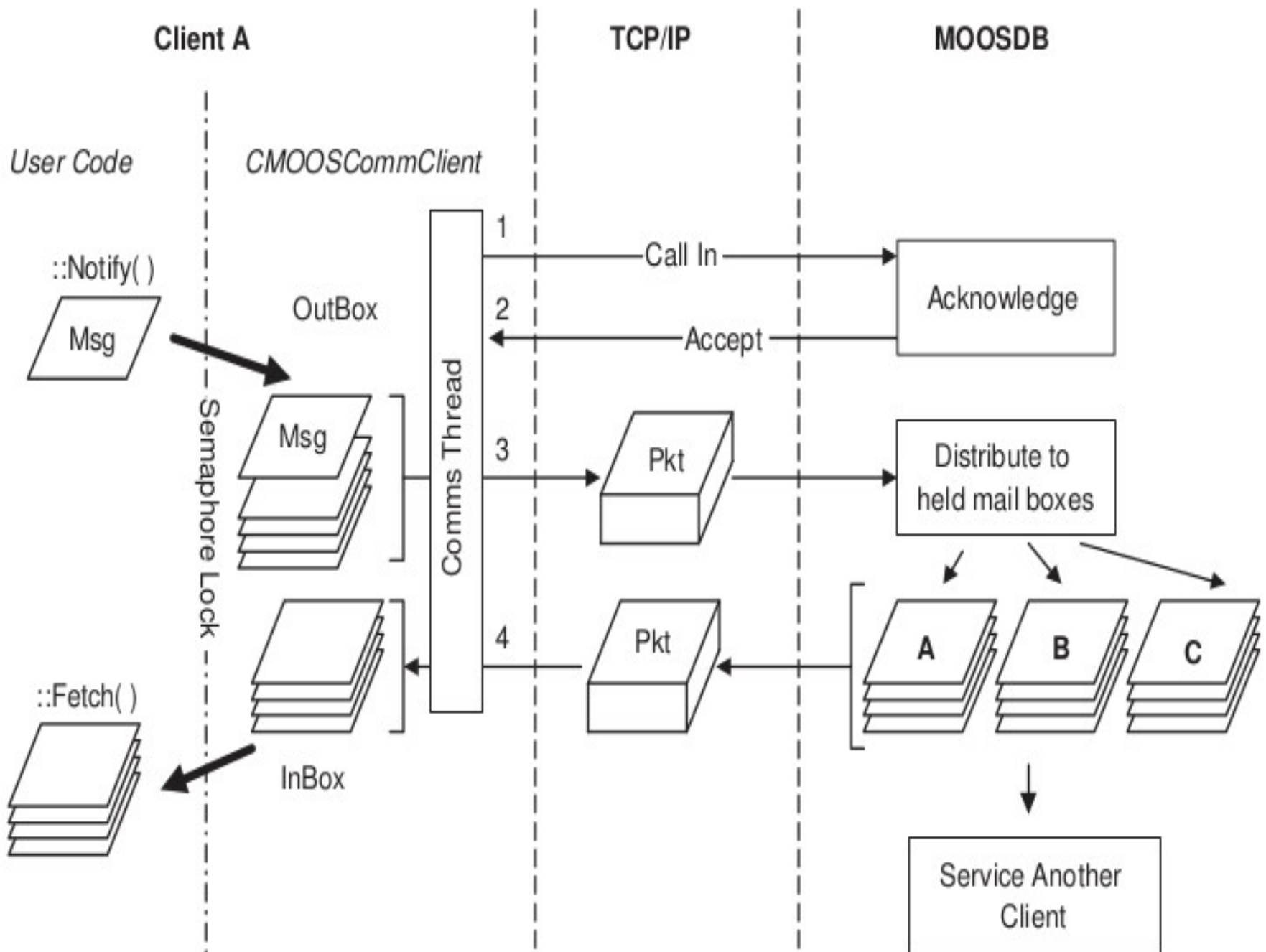
Variable	Meaning
Name	The name of the data
String Val	Data in string format
Double Val	Numeric double float data
Source	Name of client that sent this data to the MOOSDB
Time	Time at which the data was written
Data Type	Type of data (STRING or DOUBLE)
Message Type	Type of Message (usually NOTIFICATION)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

2.3 Communications Mechanics

Each client has a connection to the DB. This connection is made on the client side by instantiating a class provided in the core MOOSLIB library called `CMOOSCommClient`. Using the `CMOOSCommClient` each application can:

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





A large, stylized orange number '3' that serves as a logo. It has a thick, rounded design with a central white circle. The '3' is positioned on the left side of the slide, with its right edge overlapping the text 'Using The MOOS'.

3

Using The MOOS

3.1 MOOSLib

The primary role of MOOSLib is to contain all the communications components used both by the MOOSDBitself and CMOOSCommClient objects owned and used by client applications.



3.1.1.1 CMOOSApp

Perhaps the most important class exposed from the library is CMOOSApp. This class should be used as a base class for all MOOS applications. It provides along with other things:



3.1.1.1 CMOOSApp

1. Management and configurations of a CMOOSCommClient object
- 2.Tools for reading configuration parameters (using a file reading tool exported from MOOSGenLib)
- 3.Timing control of the main thread of the application and an additional communications thread.



3.1.1.1 CMOOSApp

4.Virtual functions that can be overloaded to perform specific actions when:

- a. New mail (notifications) arrives
- b. The default work of the application should occur
- c. The client connects to the MOOSDB
- d. The client disconnects from the server
- e. The application is about to start



3.1.1.2 A First Worked Example

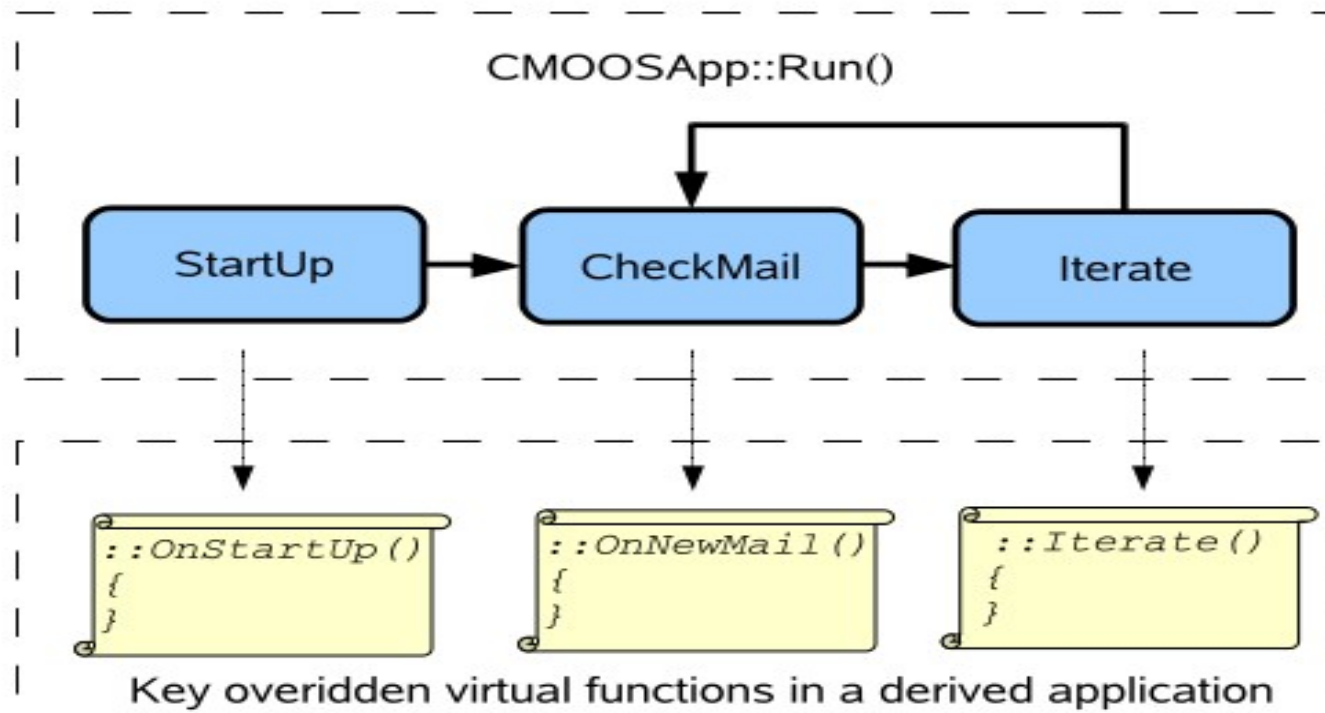
So let us use CMOOSApp procedure is as follow to build an new applications:

1. Make a new "main.cpp".
2. Make a new class derived class from CMOOSApp.
3. In main() make an instance of this class.
4. Call ::Run() on this object.



3.1.1.2 A First Worked Example

As needs dictate overload the following virtual functions:
`::Iterate()` , `::OnNewMail()`,
`::OnConnectToServer()`, `::OnStartup()`



3.1.1.2 A First Worked Example

OnStartUp

This function is called by CMOOSApp::Run just before it enters into its own “forever-loop”. This is the spot that you would populate with initialisation code, and read configuration parameters (including those that modify the default behaviour of the CMOOSApp base class) from file.



3.1.1.2 A First Worked Example

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. 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.



3.1.1.2 A First Worked Example

OnNewMail


This function is called when some other process has posted data that you have previously declared an interest in. The mail arrives in the form of 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 or numerical data and to act / process the data accordingly.



3.1.1.2 A First Worked Example

OnConnectToServer

It is actually a callback from a thread in the instance CMOOSCommsObject m_Comms possessed by CMOOSApp. The callback occurs whenever contact has been made with the MOOSDB. This is one of two places where the programmer is advised to call m_Comms.Register to tell the MOOSDB that we want to be sent mail if any other process posts data relating to a particular variable.

A decorative orange graphic consisting of a thick line and a circle, resembling a stylized 'C' or a part of a network diagram, located in the bottom right corner of the slide.

3.1.1.2 A First Worked Example

```
#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[]){

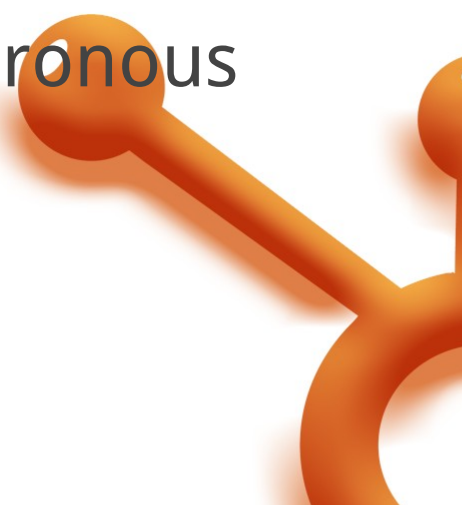
    //configure the comms
    MOOS::MOOSAsyncCommClient Comms;
    Comms.SetOnConnectCallBack(OnConnect,&Comms);

    //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;
}
```

3.1.2 CMOOSInstrument

The class CMOOSInstrument is another important base class. It is intended to simplify the writing of applications interacting with hardware via a single serial port. The class extends CMOOSApp with utilities to manage and set up a platform-independent serial port . The serial port can be configured to be asynchronous and receive unsolicited streaming data or to be synchronous and perform blocking read and writes.



3.2 MOOSGenLib

The library MOOSGenLib is a tool chest. It contains utilities and classes used throughout MOOS. In particular it provides:

- Platform-independent serial ports
- Thread safe configuration reading tool – CMOOSMissionFileReader.
- String manipulation/parsing tools.
- Geodesy tools.
- debug statement tools - MOOS equivalent of printf



3.2.1 Utility Functions

MOOSGenLib contains a host of utility functions that are described below. These functions are ubiquitous within MOOS and should not be substituted with local version producing the same functionality.



3.2.1 Utility Functions

MOOSFormat

The MOOS version of sprintf. It returns a formatted std::string object.

MOOSTrace

The MOOS equivalent of printf printing a formatted string to the console.

MOOSGetTimeStampString

Returns a time/date string formatted in the MOOS convention - useful for naming temporary local files for development purposes etc.

MOOS_ANGLE_WRAP

Wraps All angles (in radians) to be with $\pm\pi$ - forgetting to wrap angles can cause woe.



3.2.1 Utility Functions

MOOSGetValFromString

Extracts named token=val pairs from a string. For example: name=AUV1,pose=[3x1]{2,3,4}

MOOSTime

Returns the current time in decimal seconds (a double) for the current process. All connected processes will show the same time even if their respective machine clocks differ. This is achieved by deducing a client correction during initial handshaking with the MOOSDB.

MOOSPause

Pauses the current thread (not process) for a specified number of milliseconds.



A large, stylized orange number '4' graphic. It features a central circular ring with a thick orange border. A vertical stem extends upwards from the top of the ring, ending in a small orange sphere with a white dot. A diagonal stem extends from the left side of the ring, also ending in a small orange sphere with a white dot. The entire graphic has a soft orange glow or shadow around it.

4

Key MOOS Process

4.1 Naming Conventions

Process Naming


TABLE 2. THE NAMING CONVENTION FOR MOOS APPLICATIONS

Name	Description	Example
i[Name]	Interface applications. Interacts (has I/O) with an external device, for example via a keyboard or serial port	iGPS, iCompass, iRemote
p[Name]	Pure applications. Only interacts with other MOOS applications	pHelm, pLogger, pNav
u[Name]	Utility applications. Not used at run time but a useful at other times	uPlayBack, uGeodesy

4.1 Naming Conventions

Variable Naming

If a sensor, managed by a process called iSensor, measures one of these quantities then the name under which the data should be published has the format `SENSOR_CATEGORY`. This is best highlighted with a few examples:

- iGPS measures X and Y position. It publishes `GPS_X` and `GPS_Y`.
 - iDepth measures depth. It publishes `DEPTH_DEPTH`.
 - iLBL measures range and depth. It publishes `LBL_DEPTH` and `LBL_TOF` (time of flight).
- 
- A decorative orange graphic consisting of a thick line and a sphere, resembling a stylized letter 'C' or a part of a molecular structure, located in the bottom right corner of the slide.

4.2 Scheduling Communications ——pScheduler

SEQUENCES A looping sequence of messages can be created and published by pScheduler.

Each element of a sequence is specified in the configuration block with a line:

SEQUENCE = PUBLISH NAME @ VALUE : TIME OFFSET .

```
ProcessConfig = pScheduler
{
    SEQUENCE = LIGHT_CONTROL @ ON : 2
    SEQUENCE = LIGHT_CONTROL @ OFF : 4
}
```



4.2 Scheduling Communications ——pScheduler

TIMERS A “timer” allows a variable to be written to the database repetitively. A timer can be started and stopped by publication (by some other application) of user specified variables. The scheduler can also be told to derive the value of the periodic variable from another MOOS variable, which, if arrives in the Scheduler’s mail box, overrides the initial value.



4.2 Scheduling Communications

——pScheduler

syntax is as follows


```
TIMER = PUBLISH_NAME @ TIME , START_VARIABLE, STOP_VARIABLE, VALUE_VARIABLE  
-> VALUE
```

Figure 2 shows a typical configuration block. In this case the variable

```
ProcessConfig = pScheduler  
{  
    TIMER = CAMERA_CONTROL @ 4.0 , MISSION_START,  
MISSION_END, DESIRED_CAMERA_ANGLE -> 0.0  
    TIMER = CAMERA_GRAB @ 4.0 , MISSION_START -> GRAB  
    TIMER = CAMERA_GRAB @ 4.0 , -> GRAB  
}
```


4.2 Scheduling Communications ——pScheduler

RESPONSES The last competency is one of responding to the publication of one variable with the publication of one or more different variables. The syntax is obvious: `RESPONSE = STIMULUS VARIABLE :RESPONSE.VARIABLE@ VALUE,RESPONSE VARIABLE2 @ VALUE,.....` Here `STIMULUS VARIABLE` is the name of the variable we wish pScheduler to respond to, and after the colon comes a command-separated list of response variables with the values they should contain.

A decorative orange graphic consisting of several interconnected circles and lines, resembling a stylized molecular structure or a network diagram, located in the bottom right corner of the slide.

4.2 Scheduling Communications

—pScheduler

```
ProcessConfig = pScheduler
{
    //generate a sequence 6 seconds long ....
    // VAR1 will fire after 1 second
    // VAR2 fire after 3 seconds
    // VAR3 fire after 6 seconds
    // one second later VAR1 will fire again..repeat...
    SEQUENCE = VAR1 : RED @ 1 SEQUENCE = VAR2 : ORANGE @ 3 SEQUENCE =
    VAR3 : GREEN @ 6

    //generate a timer that writes "VAR_T1" with value "TimerData"
    TIMER = VAR_T1 @ 3.0, -> TimerData

    // generate a timer that writes "VAR_T2" with a string version
    // of the current value of DB_TIME (published by the DB) TIMER =
    VAR_T2 @ 2.0 ,,,DB_TIME -> TimerData

    // generate a timer that writes "VAR_T3" with the current
    // value of DB_TIME (published by the DB)
    // which only starts when "GO_T3" is published and stops
    // when "STOP_T3" is published
    TIMER = VAR_T3 @ 4.0,GO_T3,STOP_T3,DB_TIME -> TimerData

    //generate a response to "SURPRISE_ME". The variable "BOO"
    // takes on string value "HOO" and variable R9 has value
    //"get_a_grip"
    RESPONSE = SURPRISE_ME : BOO @ HOO, R9 @ get_a_grip

    //generate a response to "DB_TIME".
    RESPONSE = DB_TIME : ACKNOWLEDGMENT @ I_GOT_THE_TIME
}
```



THANK
YOU