HUMANOBS – Humanoids That Learn Socio-Communicative Skills By Observation
EU 7th Framework Project #231453

Document Title:

mBrane User Guide

mBrane Version 1.0

Author

Thor List, CMLabs

PART 3 OF 6 OF DELIVERABLE **D5 Platform Software Release 1**					
BELONGS TO WP:	WP4 - Realtime Experimental Platform				
WP LEAD:	CMLabs				
WP PARTICIPANTS:	CMLabs, UNIPA-DINFO, SUPSI-IDSIA, RU-CADIA				
	Del. ID #	**WP #**	**Orig. Date**	**Actual Date**	**Docum. Vers.**
DATA	5	4	Mo12	Mo12	1
Remarks:					

Introduction

This document introduces the mBrane framework and describes to a user how to get started with creating, running and debugging mBrane modules, as well as configuring the mBrane system and nodes.

The mBrane Modules

An mBrane module is a piece of code which is triggered by the reception of a message that the module has subscribed to. This message could have been produced by any other module in the system or be a system generated message.

Usually, the user will want to start by defining their own custom message types. A message type is usually defined in a header file and could look like this:

```
class   Type1:public   payloads::Message<Type1,StaticData,Memory>{};
```

The class Type1 inherits from the generic Message using C++ Templates. Most often, message classes are related in functionality, so a generic base class can be defined

```
template<class U>      class   _Ball:  //      to allow derivation
public  payloads::Message<U,StaticData,Memory>{
public:
        uint32  d1;
};
```

from which subsequent classes can be derived from

```
class   Ball:public            _Ball<Ball>{
public:
        int32   id;
        Ball():_Ball<Ball>(),id(-1){}
        Ball(int32      id):_Ball<Ball>(),id(id){}
};
class   ReturnBall:public              _Ball<ReturnBall>{
public:
        int32   id;
        ReturnBall():_Ball<ReturnBall>(),id(-1){}
        ReturnBall(int32       id):_Ball<ReturnBall>(),id(id){}
};
```

Now that we have defined the classes we need to load them into the system:

```
MBRANE_MESSAGE_CLASS(Ball)
MBRANE_MESSAGE_CLASS(ReturnBall)
MBRANE_MESSAGE_CLASS(Type1)
MBRANE_MESSAGE_CLASS(Type2)
```

and now they are ready to be used by the modules we create.

An mBrane module using the message classes we have just defined could be defined like this:

```
MODULE_CLASS_BEGIN(ping,Module<ping>)
        void    start(){
        }
        void    stop(){
        }
        template<class T>       Decision        decide(T        *p){
                return  WAIT;
        }
        template<class T>       void    react(T *p){    //      to messages
                OUTPUT<<"ping "<<_id<<" got another message..."<<std::endl;
        }
        void    react(SystemReady       *p){
                OUTPUT<<"ping starting PingPong test, please wait..."<<std::endl;
                startTime = Time::Get();
                NODE->send(this,new Ball(0),N::LOCAL);
        }
        void    react(ReturnBall        *p){
                int32 counter = p->id;
                //OUTPUT<<"Test got to '"<<counter<<"' so far..."<<std::endl;
                if (counter == 500) {
                        runCount++;
                        if (runCount >= 200) {
                                endTime = Time::Get();
                                uint32 t = (uint32)(endTime-startTime);
                                uint32 c = (uint32)(counter*runCount);
                                OUTPUT<<"PingPong test took "<<t<<"us for "<<c<<
                                " msgs, "<<((double)t)/((double)c)<<"us per msg"<<std::endl;
                        }
                        else {
                                //OUTPUT<<"Test got to '"<<counter*runCount<<"' so far..."<<std::endl;
                                counter = 0;
                                NODE->send(this,new Ball(counter),N::LOCAL);
                        }
                }
                else {
                        NODE->send(this,new Ball(counter+1),N::LOCAL);
                }
        }
MODULE_CLASS_END(ping)
```

A module class must implement the start() and stop() methods, which can contain custom actions when the module is started and stopped, respectively. Secondly, the module should define a decide() method, which will tell the subsystem what to do in the event of another message trigger arriving while the first one is still being processed. The options are to wait or to pre-empt.

A number of react() methods are then defined to handle different types of trigger messages. Each method can then deal with the specific type of message in a separate way, or if a joint method should handle multiple message types, a base class (such as T *p) can be specified as the message type.

When the module wishes to post a new message that it has created, it can use the NODE->send() method. NODE is actually defined as `#define NODE module::Node::Get()` to make the code a bit easier to read.

To see more examples, the built-in pingpong or Perf libraries can be examined.

The mBrane Configuration Files

To run the modules defined in code one has to first setup the mBrane nodes and system. An example config file is as follows:

```
<NodeConfiguration application_configuration_file="../App/test_application.xml">
        <Network sync_period="10000" bcast_timeout="1000" boot_callback="NULL">
                <!--sync_period:   in ms; period at which sync signals are broadcast
                from the reference node-->
                <!--bcast_timeout: in ms; delay after which a node believes it is the reference node;
                if 0, the node will be the time reference-->
                <!--boot_callback: path to shared library defining a function to be called when the
                reference node is ready (use it to boot the other nodes)-->
                <Interfaces>
                        <TCP shared_library="tcp"/>
                        <UDP shared_library="udp"/>
                        <RM     shared_library="mBrane_rm"/><!--Reliable Multicast-->
                        <IB shared_library="mBrane_mpi_o_ib"/><!--MPI over infiniband, Linux only-->
                </Interfaces>
    <Discovery interface="UDP" nic="Intel(R) 82567LM Gigabit Network Connection - Packet Scheduler
Miniport" port="10000"/>
    <Primary>
      <Control interface="TCP" nic="Intel(R) 82567LM Gigabit Network Connection - Packet Scheduler
Miniport" port="10001"/>
      <Data interface="TCP" nic="Intel(R) 82567LM Gigabit Network Connection - Packet Scheduler
Miniport" port="10002"/>
      <Stream interface="TCP" nic="Intel(R) 82567LM Gigabit Network Connection - Packet Scheduler
Miniport" port="10003"/>
    </Primary>
    <Secondary>
                        <!--same form as primary's-->
                </Secondary>
        </Network>
        <Threads thread_count="16"/><!--in [1,512]-->
</NodeConfiguration>
```

Here the network interfaces and protocols can be specified for communication, split into a primary and secondary network. Each network has three types of traffic: Control, Messages and Streams. The network interface names (the 'nic') has to match the nic name that the OS provides.

If one wishes to setup a multi node system, a <nodes> entry can be added to the above, listing all other nodes in the system.

```
        <Nodes>
                <!--ex: <Node hostname=""/>-->
        </Nodes>
```

The first line of the configuration file contains a parameter called application_configuration_file. This points to the xml file containing the configuration of the modules in the system (independently from the physical node setup).

An example application configuration file could be:

```
<ApplicationConfiguration user_library="Perf">
        <!--first, space definitions-->
        <Space name="space1" activation_threshold="0.5"/><!--name is mandatory-->
        <Space name="space2" activation_threshold="0.5">
                <Projection space="space1" activation_level="1"/><!--space1 projected on space0-->
        </Space>
        <!--second, module instantiations-->
        <Module class="ping" name="first_module" host="local"><!--name optional-->
                <Projection space="space1" activation_level="1">
```

```
                    <Subscription message_class="SystemReady"/>
                    <Subscription message_class="ReturnBall"/>
                    <Subscription stream="0"/>
                </Projection>
        </Module>
    <Module class="pong" name="second_module" host="local">
      <Projection space="space1" activation_level="1">
        <Subscription message_class="SystemReady"/>
        <Subscription message_class="Ball"/>
        <Subscription stream="0"/>
      </Projection>
    </Module>
</ApplicationConfiguration>
```

Here we see two modules being defined, but not until we have defined the spaces in which their subscriptions live. In this example, space2 has a projection onto space1, which means that space2 is only active when space1 is active and the activation level is high enough.

The two modules are then defined each with projections onto one or more spaces and each projection contains a one or more subscriptions for message types.

For more information on using modules, spaces and subscriptions, please refer to the mBrane design document.