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

Document Title:

mBrane Design

mBrane Version 1.0

Authors

Thor List, CMLabs
Eric Nivel, RU-CADIA
Kristinn R. Thórisson, RU-CADIA

PART 1 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	M12	M12	1
Remarks:					

Introduction

This document describes the design of the *mBrane* platform including the motivation to build it, the overall requirements, the details of the architecture, notes on operating system portability and how to make it as easy to use as possible for the users.

mBrane is a platform supporting large scale component-based architectures with large populations of interacting modules. It offers the possibility of distributed applications with coarse grained components using soft realtime communication via message passing.

Motivation

Other communication platforms exist which allow modules to communicate using messages. CORBA is one example of such a communication platform. Frameworks supporting dynamic publish-subscribe mechanisms exist too, such as the CMLabs Psyclone platform. The current version of Psyclone communicates using XML and although powerful in terms of flexibility does not offer the raw performance sought in this project.

The motivation is to create a communication platform which seamlessly can span multiple computers and operating systems allowing modules to register for messages by data type and allow modules to post messages without requiring them to know who will receive the data. The performance target is that 100 modules should be able to send and receive 10 messages in 10 milliseconds when distributed across a number of computers connected via a local network.

An additional requirement is that the platform should support Replicode VMs (also part of the Humanobs project).

Requirements

The following requirements are specified for the mBrane platform.

Scalability

The platform should scale well when moving from systems with tens of modules running on one computer to systems with thousands of modules running on tens of computers without requiring significant changes or upgrades to the hardware infrastructure.

Portability

The platform and the user modules should compile and run seamlessly on both 32 and 64 bit platforms supporting the major operating systems, specifically optimised for Microsoft Windows 7 and the CAOS distribution of the Linux system.

Modularity

The user should be offered a simple way of creating individual modules using the mBrane SDK without requiring them to know about the system or application as a whole, and be able to test each module both in isolation and in collaboration with parts of the rest of the system.

Reconfigurability

The system configuration, ie. which modules are running, which are active, which are subscribed to which data types, should be dynamically reconfigurable both at startup and during the runtime of the system, either manually by a human being or automatically by compiled code running in the system.

Context-based Subscriptions

The subscriptions which determine which data types trigger which modules should be context-based. This means that a particular subscription can either be active or inactive based on some larger context element. This specific requirement will be implemented using the notion of spaces where all subscriptions are part of one or more subscription spaces. If a space is currently active and the subscription level is above a given threshold then the subscription is active, otherwise it is inactive. One space can be part of another space using a similar activation level and a space is active if its activation level is above the parent space threshold. See *Figure 1* for a graphical representation. This offers a dynamic hierarchical context base which enables a powerful way of working with both global and local dynamic contexts.

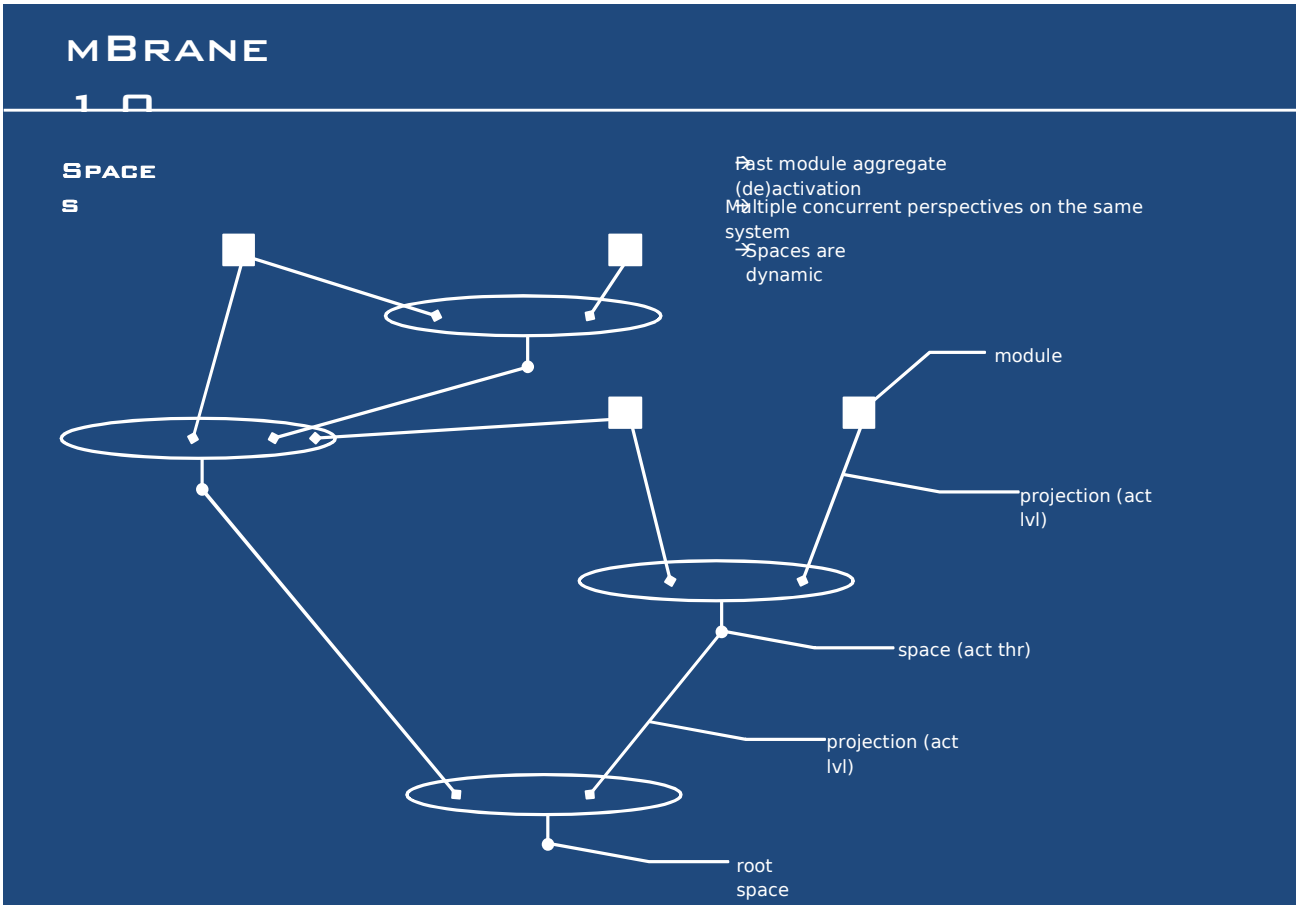

Figure 1: Graphical representation of hierarchical spaces and activation thresholds

Architecture

The mBrane architecture is divided into the mBrane Core, the mBrane Node, a number of network transport libraries (for TCP, UDP, etc.) and the user modules libraries.

mBrane Core

The mBrane core provides objects and functionalities for pipes, queues, threading, networking, payloads, memory management and dynamic library loading. This is the place where all the operating system dependent code resides as well, shielding both the mBrane Node and the users from having to deal with this part.

mBrane Node

The mBrane node is responsible for managing and running user modules, managing message queues, taking care of all network traffic and making sure that all the nodes in the system are present and time synchronised. One node will always be the reference node off which all other nodes synchronise their clocks. If the reference node disappears another node will be designated the reference node and take over its responsibilities.

Network Transport Libraries

The mBrane nodes can make use of a number of network transport protocols when communicating with each other. Communication is split up into discovery, control, data and streaming data and can be divided between a primary and a secondary network. Each communication channel can be broadcast based (like UDP) or packet based (like TCP), although the discovery channel always has to be broadcast based, of course. This means that up to eight different network channels can be specified for each node and a separate network transport library can be loaded for each. Initially, TCP and UDP communication will be supported and later one or more Infiniband protocols will be implemented.

User Modules Libraries

When a user wants to implement new modules and message types they put these into dynamically loadable libraries which are loaded into mBrane at runtime. A custom message type is defined as a class inheriting from the Message class and the user can add any number and type of content variables to their messages.

Modules can then be created by implementing a number of methods and registering these as modules using predefined macros. Modules can be programmed to receive both specific and generic types of data (messages and streams) and respond by creating new messages to be posted as output. Modules can also decide if and how they wish to be pre-empted when new triggermessages arrive and perform custom actions upon startup and shutdown.

Processing Model

The processing model is based on message pipelines with execution threads pop'ing jobs containing messages for processing by modules. See *Figure 2* for a graphical representation of the processing model.

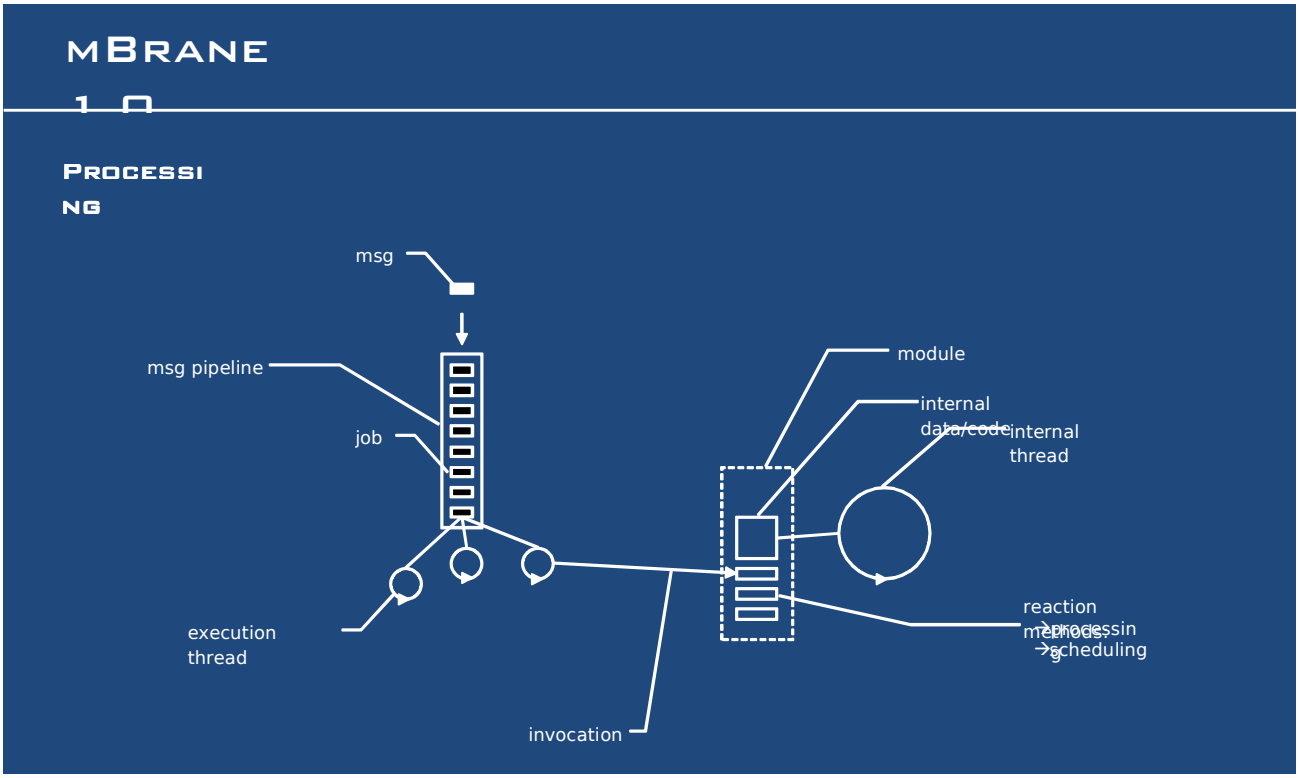

Figure 2: The mBrane processing model

Portability

The mBrane system is specifically designed to support 100% seamless integration on multiple different platforms including Windows and Linux, 32 and 64 bit. This offers the users piece of mind when developing their modules, but introduces a number of challenges for the design of mBrane.

Threading

Threading is handled somewhat differently on different operating systems. On Windows threads are implemented as part of the Win32/64 subsystem and UNIX systems has PThreads as well as several other libraries. For this project we chose to support the Windows and PThreads systems, which will enable portability to most Microsoft Windows versions and Linux systems. Thread initialisation, signalling and termination are handled very differently, so separate code will be implemented to handle this. Thread monitoring is not only handled differently, but is actually implemented differently especially between different flavours of PThreads, so the more advanced bits of this may not be available equally on all systems.

Networking

TCP and UDP networking are implemented on most platforms using Berkley sockets, but the subsystems does sometimes offer challenges on some systems. The receive socket buffering on Linux is best implemented in software using non-blocking calls and on Windows the identity of each network interface is done using a very long descriptor that includes the model name of the network interface card, making it troublesome to specify the use of a specific network interface in code. Error codes are also different on Windows and Linux and even the causes of these vary and will need to be handled differently.

Time

High resolution time is a topic of much debate. On Windows the processor high resolution timers offer a convenient way of measuring time differences with very high precision (100 nanoseconds), but standard time is only offered down to the nearest millisecond. On Linux access to the processor high resolution timers are only available via assembler code, but standard time can be read down to the nearest microsecond. On some kernels, however, the standard system ‘tick’ is 16 milliseconds and time resolution is sometimes limited because of this.

Atomic Operations

Atomic operations such as read or test and increment an integer number are easily available on Windows both for 32 and 64 bit integers. On Linux, these are actually available, but only using undocumented calls to the GCC subsystem. Luckily, Google can help find this information. Semaphores and mutexes are also implemented differently, especially when using named versions to make them available across processes and significant custom code will need to be implemented to handle this.

Dynamic Libraries

Loading of a dynamic library of code is offered on all platforms and although the actual code varies widely between systems the base functionality are available. Some differences will affect the user as their code will need to be compiled differently and global static variable are instantiated differently, but this should be easily overcome by providing example projects and readme files.

Compiler Differences

The biggest challenge of cross platform development comes from the different ways the compilers implement the C++ standards. The handling of inline methods in classes and the implementation of standard macros like the __COUNTER__ macro put significant strain on any development project and as such limitations on which compilers and versions that can be used. In this project the Microsoft Visual Studio 2008 compiler is used with 2005 and 2010 expected to work as well and on Linux the GCC compiler version 4.3 and above will be supported. Both 32 and 64 bit versions of these compilers will be supported, although 64 is recommended as the code is optimised for it by implementation design.

Usability

The mBrane framework is designed to make it as easy for the user as possible to create and manage their modules. They merely need to program how each module will handle each received message type, compile and then load them into a running mBrane system. Two files exist for managing the mBrane system, one for the physical configuration of nodes including network channels and synchronisation and another for managing modules including subscriptions and on which node to run, as well as the subscription spaces.

The node configuration file is static and will not need to change after the system has been started up. The application configuration file, however, is merely an initial configuration which can be changed later on at any time to match updated requirements, contexts or module configurations.