Justina's Blackboard Architecture

Mauricio Matamoros Jaime Márquez Jesús Savage

March 18, 2013

Abstract

To control a general purpose mobile service robot a complex software is required, and a complex software must be divided into several modules in order to make it maintainable, allow people to work on it simultaneously, and reduce the complexity of it's development; eliminating the need to update or change the existing modules due to the integration of new modules/functions or the modification of a current module. The current document describes Justina's Blackboard Architecture, an architecture for the communication layer of the several modules which operates the robot Justina.

1 Introduction

A general purpose mobile service robot can be seen as a complex system that responds to stimuli with a predetermined behavior. Those stimuli may be produced as a change in the robot's environment or as a change in the robot's internal status. In an higher level of abstraction, those stimuli are a reaction to a direct interaction of the environment of the robot (like in a human-robot interaction where a human gives an order to the robot), i.e. by command; or because the robot decides to act due to an indirect interaction (the robot notices that a human falls and decides to act before the human request for help, or is smoke is detected in the environment), i.e. by awareness.

This reaction by command or awareness can be mapped on how the modules which comprise the software system of the robots interact: imperatively by a synchronous command-response or by event firing due to a change in the status of the robot. Both methods are equivalent and which can be achieved by the first method can also be achieved by the second one, and each has its pros and cons, but still the imperative method is the best for commands and the event firing method if the best for awareness so, if the robot is going to react in both ways it is desirable that the software supports both methods.

In general, a software system can be developed under several architectures, depending on the characteristics to be analyzed, like the control flow, module dependency, hierarchical structure, communication between modules, etc. Regardless the approach used to design the software of the robot, a constant is that all the modules need to communicate with at least one other module. This can be achieved in two general ways: through a central module (Centralized) or by direct connection (Peer-to-Peer). Using a Peer-to-Peer architecture makes a tightly-coupled system with (relatively) high performance, but changes done to one module impact on its dependants increasing development time; while the use of a Centralized architecture makes a loose-coupled system with (again relatively) low performance, but the changes done to one module.

The development of the software which controls a general purpose mobile service robot is a process which requires the collaboration of several persons with a single field of research/expertise called Experts. Those experts are commonly bachelor or master students whose participation in the project is between one and four years, and commonly they are inexperienced programmers or beginner programmers. It is desirable that the Experts spend their time developing, testing and enhancing their algorithms; instead of wasting time working on the communications layer of the application to make it work together with the whole system.

Another problem occurs when an Expert abandons the team without leaving a fully functional module without source code, or the source code is not clean and understandable. In both cases the module cannot be modified, so it is desirable that the whole system can grow up without modifying some of its components.

With all this considerations, a software architecture for the communication between the modules which reduces the changes to be made to the dependants of a modified module has been chosen and it is described in the next sections of this document.

2 Background

In the current section, the basic concepts and definitions are shown.

The development of a computer program has always been an obscure topic, considering trivial the step between the algorithm and it's execution over a data set. The development of software system has been considered an art, where the artist (the developer) uses all his knowledge about the programming language and his experience to implement the algorithm, including all the auxiliary functions and the necessary heuristics.

This has led to the well-known traditional organization (or development method) Main-program/Subroutines [1, 2], where a function or procedure describes the main algorithm's execution, aided by a collection of functions or subroutines called secondary procedures that perform specific tasks. Such secondary tasks, but may correspond to other algorithms, are not specified in the main algorithm (considered trivial or given), so it is up to the programmer to wits and propose a solution that does not compromise the performance of the main algorithm being implemented.

During the execution of a program that controls the operation of a robot using a particular algorithm, it is important that data is available in time. However, it is irrelevant for the algorithm if the data was requested to the sensor when the algorithm requires it, or if it was requested in advance by an external agent and stored in a buffer to be available when the algorithm requires it. This leads to think in concurrence, or even in parallel and distributed computing for robots which requires more processing power.

2.1 Peer-to-Peer Architecture

The term Peer-to-Peer refers to those systems and applications which use distributed resources to perform a function in a decentralized way [3]. In a distributed system, the peers (or equals) are processes that communicate with each other, and unlike the client-server architectures, can act as a client or server at convenience [4]. These decentralized computing systems can be done at any node in the network, without a clear distinction between clients and servers [5].

Sommerville defines a Peer-to-Peer architecture in [5] as:

> "Peer-to-Peer systems are decentralized systems where computations may be carried out by any node on the network and, in principle at least, no distinctions are made between clients and servers".

According to Sommerville [5], applications and Peer-to-Peer networks are designed to take advantage of computing power and storage available on a potentially large computer network, while Milojicic in [3] focuses Peer-to-Peer systems not in the computing power, but in sharing resources in a community of peers.

In a decentralized architecture, the nodes allow to enroute packets from one node to another, offering multiple paths, and several nodes which perform the same task can be added leading to high redundancy and fault tolerance [5]. In addition, it increases scalability, as direct communication among peers and facilitates the incorporation of new resources, being the decentralization applicable to algorithms, programs and data [3].

However some advantages of the Peer-to-Peer architectures are not suitable for a General Purpose Mobile Service Robot whose software is made of a relatively small number of modules (nodes) and has not enough computing power to allow redundancy.

2.2 Blackboard Architecture

The Blackboard architecture was originally designed as a method to handle complex and poorly defined problems. The first famous example is the speech recognition system Hearsay II. A more recent example is the PLAN component of the mission control system for the RADSAT-1 [6]. Other examples of use include signal processing, pattern recognition and speech recognition [1].

In a Blackboard architecture, several specialized subsystems work together to create a partial or approximate solution to the problem [4]. A Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure called *Blackboard*: the central data store, the other components read and write coordinated by a control component [4].

2.2.1 Problem

The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures; problems that, when decomposed into subproblems, span several fields of expertise; problems which solution involves the solution of partial problems using different representations and paradigms, and problems which deal with uncertain or approximate knowledge during its solution and each transformation step can also generate several alternative solutions [4].

2.2.2 Forces

A Blackboard is useful in the following cases [4]:

- A complete search in the solution space is not feasible in a reasonable time.
- Since the domain is inmature it is needed to experiment with different algorithms for the same task. For this reason, the individual modules should be easily interchangeable.
- There are different algorithms to solve partial problems.
- Input as well as intermediate and final results have different representations, and algorithms are implemented according to different paradigms.
- An algorithm usually works on the results of other algorithms.
- Uncertain data and approximate solutions are involved
- Employing disjoint algorithms induces potential parallelism. If possible, strictly sequential solution shall be avoided.

2.2.3 Solution

The idea behind the Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure. Each program is specialized for solving a particular part of the overall task, and all programs work together on the solution. [4].

2.3 The *Publisher-Subscriber* design pattern

The *Publisher-Subscriber* design pattern helps to keep the state of cooperating components synchronized. To achieve this it enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state [4].

2.3.1 Problem

A situation often arises in which data change in one place, but many other components depend on this data [4].

2.3.2 Forces

The solution should balance the following forces [4]:

- One or more components must be notified about state changes in a particular component.
- The number and identities of dependent components is not known *a priori* or may change over time.
- Polling by dependents for new information is not feasible.
- The information publisher and its dependents should not be tightly coupled when introducing a change propagation mechanism.

2.3.3 Solution

One dedicated component takes the role of the publisher. All components dependent on changes in the *publisher* are its *subscribers*. Subscribers can be executed locally or remotely [4].

The publisher maintains a registry of currently-subscribed components. Whenever a component wants to become a subscriber, it uses the subscribe interface offered by the publisher. Analogously, it can unsubscribe [4].

Whenever the publisher changes state, it sends a notification to all its subscribers. The subscribers in turn retrieve the changed data at their discretion [4].

In general, two models of this pattern may be used: the *push* model and the *pull* model, however, many variations are possible in middle groun between these two extremes [4].

The *push model* . In the *push model* the publisher sends all changed data when it notifies the subscribers. The subscribers have no choice about if and when they want to retrieve the data [4].

This model has a very rigid dynamic behavior, but for complex data changes it can be a poor choice, specially when the publisher sends large amounts of data which may cause overheads [4].

The *pull model* . In the *pull model* the publisher only sends minimal information when sending a change notification. The subscribers are responsible for retrieving the data they need [4].

This model offers more flexibility than the *push model*, but at the expense of a large number of messages between publisher and subscriber [4].

3 The Justina's Blackboard Architecture

One of the restrictions for the general purpose service robots is that they need to be low-expense and self-contained, so all the computation must be performed within the robot itself with common technology like ordinary laptops. Thus, the software system used to operate several general purpose service robots has been designed to be modular and distributed, so it can be executed over several computers connected with each other using TCP/IP over either Fast Ethernet or Gigabit Ethernet in star topology.

For the software, a blackboard-based centralized architecture for the communication layer has been chosen, due to its maintainability, extensibility, restructuring and ease of development. However, it is important to highlight that it is not a traditional blackboard architecture as described in the literature, since its operation is not based only in the read/write of the shared variables, but also RPC (*Remote Procedure Call*) via message passing through the centralized component.

In this architecture, each component of the system is called *Module* and the central module which manages the communication between the other modules is called Blackboard, in other words, the system is made of several modules which communicates with each other via the Blackboard. The communication is achieved via *message-passing* through TCP sockets.

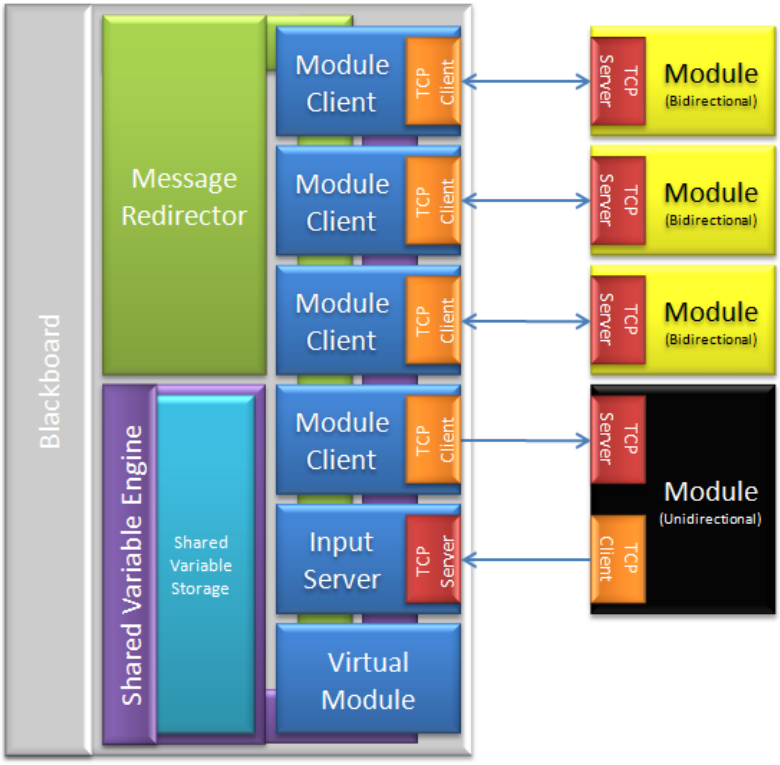

Figure 1: Block diagram of the Blackboard and connectivity with other modules

3.1 Blackboard central module architecture

In order to enhance restructuring, all modules must implement a TCP Server so they don't need to care about where the other modules are. An exception to this rule is the Blackboard, which rises a TCP Client for each of the known modules of the system. Although almost all the comunications are bidirectional, the blackboard has also a TCP Server for compatibility with unidirectional modules (see figure 1).

The Blackboard central module is structured in two main layers: the *Communication Layer* and the *Application Layer*. *The Communication Layer* encapsulates the communications and provides the interfaces which allow the application layer to communicate with the modules that conform the whole system. In the *Application Layer* the main tasks of the Blackboard are performed: the message redirection and the shared variable management.

As shown in figure 1, for each module in the system, the Blackboard instantiates a *Module Client* connector which allows the Blackboard to communicate with the module and monitor the state of the module. Also there is an *Input Server* for incomming data from unidirectional applications. Both the Input Server and the Module Clients belong to the communication layer of the Blackboard and have full access to the upper layer conformed by the *Message Redirector* and the *Sahred Variable Engine*.

The second layer consists of two main components: the *Message Redirector* and the *Shared Variable Engine*. The *Shared Variable Engine* manages subscriptions and provides access to the *Shared Variable Storage* where the Shared Variables lies (see figure 1). All modules have read permission on all the defined variables but the write permission can be denied for specified modules on each variable.

The *Message Redirector* is the component that allows communication between modules, redirecting each message received to its destination and generating failure responses if the destination module is not available or the timeout has been reached.

3.1.1 The Command/Response model

The communication between modules is achieved using 2 ways message-passing and represents a function execution request made by a *consumer module* to the *host module*, called **Command**, and the execution result produced by the *host module* which will be sent back to the *consumer module* and is called **Response**.

In other words, a **Command** is the execution request of the function $f_i^{m_j}$ that belongs to the module m_j made by the module m_k (denotated by $m_k \xrightarrow{call} f_i^{m_j}$). To request the execution of $f_i^{m_j}$, the module m_k only needs to send the name of the function (i.e. f_i with the execution parameters and the Blackboard will infer that the destination is m_j and will redirect the request. After that, the blackboard will wait until m_j sends the execution result of f_i which is a **Response**, and will redirect it back to m_k. If m_j gets disconnected during the execution of $f_i^{m_j}$ or the predefined maximum execution time for that function elapses, the Blackboard will automatically generate a failure response to send it back to m_k. Notice that if the Blackboard receives an unknown command or a response for an unmade function execution request, it will be discarded.

The Blackboard assumes that, no matter how many functions a module has; it can execute only one function at a time. Thus, if the module $m_k \xrightarrow{call} f^{m_j}$ the Blackboard will mark m_j as **Busy** and will no longer redirect any other command to m_j until the execution of f^{m_j} has been finished. Therefore if during the execution of f^{m_j} the request $m_i \xrightarrow{call} g^{m_j}$ is made, the Blackboard will automatically generate a failure response for m_i since m_j is not available.

However, the Blackboard supports two priority levels for command execution. A high-priority command will be always redirected to its destination module, so, if the module $m_k \xrightarrow{call} f^{m_j}$ and during the execution of f^{m_j} the request $m_i \xrightarrow{call} h^{m_j}$ is made being h^{m_j} a high-priority function, the Blackboard will redirect h to m_j even if m_j is marked as busy and will wait for the responses of both, f^{m_j} and h^{m_j}, being responsibility of m_j to handle both functions.

Also, the read and write of Shared Variables are achieved by a command/response, but the commands to read/write Shared Variables are not redirected, but managed by the Blackboard itself to execute the read/write over the involved Shared Variables.

Finally, since not all the functions may return a result, commands can be configured to not be answered (like a module shutdown). In those cases, the Blackboard will only redirect the command to the destination module but will not wait for a response nor will mark the destination module as **Busy**.

3.1.2 Conventions

Module names The module names must start with an uppercase letter and be made of uppercase letters, digits and minus, but can not end with a minus. The total length must be greater or equal than 3.

Command names The command names must start with a lowercase letter and be made of lowercase letters, digits and underscores with a total length greater or equal than 3.

Shared variable names The shared variable names are like in C. They must start with a letter or an underscore and can contain several letters, digits and underscores.

3.2 The Blackboard configuration file

The Blackboard central module is configured via an XML file called *Blackboard Configuration File* within the `<blackboard>` tag, where are defined the following elements:

- Blackboard general settings.
- Preconfigured shared variables.
- List of system modules with the commands for each module.

Each of this elements are described below.

3.2.1 Blackboard general settings

The Blackboard general settings are defined inside the `<configuration>` and `</configuration>` tags. In this section, the following parameters are defined:

- The Blackboard's virtual module name.
- The Blackboard's Input Server port.
- [Optional] The maximum number of retry attempts for send operations.
- [Optional] The maximum execution time of the Blackboard.
- [Optional] The time at which the *TestTimeOut* event is fired.
- [Optional] The program startup sequence.

Blackboard's virtual module name . As it is explained in section 3.1, all the components of the architecture are modules, even the blackboard itself. Thus, within the Blackboard there is a Virtual Module which is responsible of executing all the Blackboard's tasks, and like all other modules, it has a name. The Blackboard name is defined within the tags `<name>` and `</name>`. Like

The Blackboard's Input Server port . As it is explained in section 3.1 the blackboard provides an *Input Server* for compatibility with unidirectional modules. The Input Server connection port can be defined within the tags `<port>` and `</port>`.

Send Attempts option . This setting is used to enhance robustness when the network is unstable or failing. In those cases, the Blackboard can retry to send a message several times if the first attempts fail. This setting is configured within the tags `<sendAttemps>` and `</sendAttemps>`. The default value is zero.

Auto-stop time option . This setting is used to test scenarios for the RoboCup @ Home challenges. Once the specified time elapses, the Blackboard halts itself. This setting is optional and the time is configured within the tags `<autoStopTime>` and `</autoStopTime>` in milliseconds. A value less or equal than zero will disable this feature. The default value is zero.

Test timeout option . This setting is used to test scenarios for the RoboCup @ Home challenges. Once the specified time elapses, the Blackboard fires the *Test Timeout* event. This setting is optional and the time is configured within the tags `<testTimeOut>` and `</testTimeOut>` in milliseconds. A value less or equal than zero will disable this feature. The default value is zero.

Startup sequence option . This setting enables the Blackboard to launch the listed modules during its startup. In order for the blackboard can automatically launch a module the initialization parameters of the module must be set in its definition, the module must be marked as *enabled*, the module must be on the same computer as the Blackboard (or there must be a *Blackboard satellite* running on the target machine), and the module name must appear on this list. The list is enclosed within the tags `<startupSequence>` and `</startupSequence>` and consist of several `<module></module>` tags enclosing the module names.

Finally, Code 1 shows an example of the general settings section of a Blackboard configuration file.

Code 1: An example of the general settings section of a Blackboard configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<blackboard version="1.0">

    <configuration>
        <name>BLACKBOARD</name>
        <port>2300</port>
        <sendAttemps>0</sendAttemps>
        <autoStopTime>600</autoStopTime>
        <testTimeOut>450</testTimeOut>
        <startupSequence>
            <module>ACT-PLN</module>
            <module>MVN-PLN</module>
        </startupSequence>
    </configuration>

    <sharedVariables>...</sharedVariables>

    <modules>...</modules>

</blackboard>
```

3.2.2 Shared variable configuration

The Blackboard Shared Variables section allows to define the shared variables which will be availiable immediately after the Blackboard's statup, and this is the only way to define which modules have write permission over a variable (all variables created during the execution of the Blackboard can be read and written by any module). They are defined inside the `<sharedVariables>` and `</sharedVariables>` tags. In this section, the following parameters are defined:

- [Attribute] The shared variable name.
- [Optional attribute] The shared variable type.
- [Optional attribute] The shared variable initial value.
- [Optional] The list of modules with write permission.

Shared variable names . The name of a shared variable is defined like in C programming language, i.e. that it must begin with a leter between 'a' and 'z' or 'A' and 'Z', and the underscore symbol; and be composed of alphanumeric characters and the underscore symbol. It is used to start with lowercase letters.

Shared variable types . The type of a shared variable is defined like in Java/C# programming language, i.e. that it must begin with a leter between 'a' and 'z' or 'A' and 'Z', and the underscore symbol; and be composed of alphanumeric characters and the underscore symbol. Dynamic size array types are defined by adding square braces after the type. Fixed length array types are defined by adding square braces after the type and the number of elements of the array between the braces.

There is a spetial type of variable called `var` which can store any kind of data. It is similar to `Object` in Java/C# but `var` stores data, not a reference to an object.

It is important to remark that the blackboard does not serialize nor deserialize the received data. It only check that the specified types in the read/write operation matches. This is in order to favor performance over data integrity and to allow the Blackboard to support unknown types.

Some examples of variable definition are:

- string myString

Code 2: An example of the Shared Variable Configuration section of a Blackboard configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<blackboard version="1.0">

    <configuration>...</configuration>

    <sharedVariables>
        <var name="var1" type="var"/>
        <var name="var2" type="string"
        value="string \&quot;hello\&quot;">
        <var name="var3" type="double[]">
            <writers>
                <writer>*</writer>
            </writers>
        </var>
        <var name="var4">
            <writers>
                <writer>MVN-PLN</writer>
            </writers>
        </var>
    </sharedVariables>

    <modules>...</modules>

</blackboard>
```

- double[] array_of_doubles
- LaserLecture[728] laser_URG_01

Shared variable initial value . Within the definition of a shared variable in the Blackboard's configuration file, the initial value of a shared variable may be defined.

Modules with write permission . When a shared variable is created in the Blackboard all registered modules have permission to read and write the shared variable; however some time is necessary that only some modules have write permission. Within the definition of a shared variable in the Blackboard's configuration file, the list of modules with write access may be defined.

The list of modules must be specified inside the Shared Variable definition scope within the `<writers>` and `</writers>` tags, entering the module name as a value of the tags `<writer>` and `</writer>`. The asterisk char may be entered to grant write access to all modules.

Finally, Code 2 shows an example of the general settings section of a Blackboard configuration file.

3.2.3 System modules

The Blackboard Modules section allows to define the list of modules comprising the system. They are defined inside the `<modules>` and `</modules>` tags, and each module is defined inside the `<module>` and `</module>` tags. In this section, the following parameters are defined:

- [Attribute] The name of the module.
- [Optional attribute] The name of the person which is responsible of the module (author).
- [Optional attribute] A flag that defines if the module is enabled or disabled (all modules are enabled by default).
- [Optional attribute] An alternative name for the module (alias).

Code 3: An example of a module configuration of a Blackboard configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<blackboard version="1.0">

    <configuration>...</configuration>

    <sharedVariables>...</sharedVariables>

    <modules>

        <module name="PRS-FND" enabled="true" author="Mauricio Matamoros">
            <ip>127.0.0.1</ip>
            <ip>192.168.1.5</ip>
            <port>2075</port>
            <aliveCheck>true</aliveCheck>
            <simulate>true</simulate>
            <commands>
                <command name="pf_find" answer="True" timeout="7500" />
                <command name="pf_remember" answer="True" timeout="25000" parameters="true"
                    />
                <command name="pf_auto" answer="True" timeout="500" priority="true" />
                <command name="pf_sleep" answer="True" timeout="500" parameters="false"
                    priority="true" />
                <command name="pf_shutdown" answer="True" timeout="500" priority="true" />
            </commands>
        </module>

    </modules>

</blackboard>
```

- The IP address of the machine running the defined module application.
- The connection port to connect to the module application.
- [Optional] The startup information to launch the module application.
- [Optional] A flag indicating if the Blackboard can monitor the status of the module application.
- [Optional] A flag indicating if the module application requires the name of the module which request a given command.
- [Optional] A flag that indicates if the Blackboard must simulate the module.
- [Optional] A list of actions the Blackboard must execute when the blackboard is started.
- [Optional] A list of actions the Blackboard must execute when the blackboard is stopped.
- [Optional] A list of actions the Blackboard must execute when the blackboard is restarted.
- [Optional] A list of actions the Blackboard must execute when the `Test Restart` event occurs.
- [Optional] A list of actions the Blackboard must execute when the `Test Timeout` event occurs.
- The list of commands executed by the module.

Code 3 shows an example of a module configuration of a Blackboard configuration file.

Module's name . The module's name is the name that the Blackboard uses to identify that module in the system. The `name` attribute of the `<module>` tag defines the name of the module.

The name of the modules must start with a uppercase letter between 'A' and 'Z', consist of uppercase letters, digits between '0' and '9' or the minus simbol, end with a uppercase letter or a digit, and its minimum length is 3 characters long. This correspond to the regular expression:

```
^[A-Z][0-9A-Z\-]+[0-9A-Z]$
```

Also, the name of the module must be unique in the whole system. If the Blackboard finds a duplicate module name in the configuration file, it will be renamed.

Module's author . The module's name is a string that contains the name of the person in charge of that module and it has no relevance, is just supported as metadata which may be helpful during development. The `author` attribute of the `<module>` tag defines the author of the module.

Module enablement . If present, the value of the `enabled` attribute of the `<module>` tag indicates enabled status of that module. By default all modules are enabled.

A disabled module will be skipped by the Blackboard as if it were not present. This setting is helpful when several versions of the same module are availiable but only one is used during test and development.

Module's alias . If present, the value of the `alias` attribute of the `<module>` tag specifies an alternative name for that module.

Some times a module is renamed but other (old) modules still uses the old name, or two modules are merged into one single module and other modules may refer to the new merged module by the old name. In those cases, an alias can be used to quickly solve the problem without modifying the code of the module dependants.

Notice that the same rules for naming modules apply, and also must be unique.

Module's IP Address . The IP address of the machine running the module application must be defined inside the `<ip>` and `</ip>` tags. Several `<ip>` tags may be defined and the Blackboard will try to connect to the module sequentially on each of these addresses until a connection be established.

Since try to establish a connection may take several seconds, the list of IP addresses must be kept as short as possible.

Module's port . The connection port on which the module application is listening for incomming connections must be defined inside the `<port>` and `</port>` tags. Although the range of available ports for TCP sockets goes from 0 to 65535, the Blackboard will reject any module working in the first 1024 ports since they are reserved for common applications.

Application startup information . The application startup information is used by the *Blackboard Launcher* 4.4 to launch the module applications during the Blackboard's bootstrap. The following startup parameters are defined inside the `<program />` tag:

- `processName` The name of the process to be executed. It is used to check if the module is running at process level and optionally kill it.
- `path` The path of the executable file which launches de module application.
- `args` The command line arguments to be passed to the executable file which launches de module application.

Monitoring flag . [Optional] During its execution the Blackboard monitors the status of the system modules. This behavior may be overridden specifying if the Blackboard can check the status of a module or not setting the value inside the `<aliveCheck>` and `</aliveCheck>` tags to `false`.

The following status are monitored:

- *Alive* The module is running and answering.
- *Busy* The module is running but it is busy and can not execute normal-priority commands.
- *Connected* The Blackboard has established a TCP connection with the module application and the socket is open.
- *Ready* The module has notified that it is ready to operate normally.

Those status and the messages used to monitor each module are described widely in Section 4.3.

Sender prefix flag Some modules requires to know which module is requesting the execution of a command. This flag forces the blackboard to pre-append the name of the module (source) which is requesting a command execution or sending a response for a command request. This feature, disabled by default, can be enabled setting the value inside the `<requirePrefix>` and `</requirePrefix>` tags to `true`.

Simulation flag . The Blackboard has the ability to simulate modules in a very simple way, acting as if the simulated module were connected and generating a success response for each command sent to that module. This feature, disabled by default, can be enabled setting the value inside the `<simulate>` and `</simulate>` tags to `true`.

Start actions . A list of actions the Blackboard must execute when the blackboard is started may be defined inside the `<onStart>` and `</onStart>` tags. These actions are explained in Section 4.4.

Stop actions . A list of actions the Blackboard must execute when the blackboard is stopped may be defined inside the `<onStop>` and `</onStop>` tags. These actions are explained in Section 4.4.

Restart actions . A list of actions the Blackboard must execute when the blackboard is restarted may be defined inside the `<onRestart>` and `</onRestart>` tags. These actions are explained in Section 4.4.

Test restart actions . A list of actions the Blackboard must execute when the `Test Restart` event occurs may be defined inside the `<onRestartTest>` and `</onRestartTest>` tags. These actions are explained in Section 4.4.

Test timeout actions . A list of actions the Blackboard must execute when the `Test Timeout` event occurs may be defined inside the `<onTestTimeOut>` and `</onTestTimeOut>` tags. These actions are explained in Section 4.4.

Registered commands list . The list of commands which can be executed by the module is defined inside the `<commands>` and `</commands>` tags. Each command is configured with several attributes inside the `<command />` tag as follows:

- `name` The name of the command.
- `answer` Specifies if the command generates a *Response*.
- `timeout` The maximum amount of time the Blackboard will wait for a *Response*. If no *Response* is received, the Blackboard will generate and send a *Failure Response*.

- `parameters` Optional. Indicates if the *Command* requires parameters to be executed. Default is `true`.
- `priority` Optional. Indicates if the *Command* has high priority and must be send even if the module is busy. Default is `false`.

Command and *Response* are explained widely in Section

3.2.4 Blackboard Configuration File example

Code 4: A Blackboard configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<blackboard version="1.0">

    <configuration>
        <name>BLACKBOARD</name>
        <port>2300</port>
        <commands />
    </configuration>

    <sharedVariables>
        <var name="hd_pos" type="double[]"/>
        <var name="hf_skeletons" type="string"/>
        <var name="hf_gesture" type="string"/>
        <var name="mp_currentRoom" type="string"/>
        <var name="mp_currentRegion" type="string"/>
        <var name="mp_currentLocation" type="string"/>
        <var name="mp_goalRegion" type="string"/>
        <var name="mp_odometryPos" type="double[]"/>
    </sharedVariables>

    <modules>

        <module name="MVN-PLN1" author="Marco Negrete" enabled="True" alias="MVN-PLN">
            <ip>127.0.0.1</ip>
            <port>2011</port>
            <aliveCheck>true</aliveCheck>
            <commands>
                <command name="mp_move" answer="True" timeout="60000" />
                <command name="mp_goto" answer="True" timeout="60000" />
                <command name="mp_getclose" answer="True" timeout="60000" />
                <command name="go_to_room" answer="True" timeout="120000" />
                <command name="go_to_region" answer="True" timeout="120000" />
                <command name="mv" answer="True" timeout="60000" />
                <command name="ic" answer="True" timeout="10000" />
                <command name="stop" answer="True" timeout="500" parameters="false" priority
                    ="True" />
            </commands>
        </module>

        <module name="TORSO">
            <ip>192.168.190.207</ip>
            <port>2040</port>
            <commands>
                <command name="trs_mv" answer="True" parameters="True" timeout="20000"
                    priority="True" />
                <command name="trs_abspos" answer="True" parameters="False" timeout="20000"
                     priority="True"  />
                <command name="trs_relpos" answer="True" parameters="True" timeout="20000"
                    priority="True"  />
            </commands>
        </module>

        <module name="SENSORS">
```

```
50              <ip>192.168.190.207</ip>
51              <port>2015</port>
52              <commands>
53                  <command name="jc_start" answer="False" timeout="300000" parameters="false"
                        />
54                  <command name="jc_stop" answer="True" timeout="1000" parameters="false" />
55
56              </commands>
57          </module>
58
59          <module name="SP-GEN">
60              <ip>127.0.0.1</ip>
61              <port>2052</port>
62              <commands>
63                  <command name="say" answer="true" timeout="20000" priority="True" />
64                  <command name="read" answer="true" timeout="120000" priority="True" />
65              </commands>
66          </module>
67
68          <module name="SP-REC">
69              <ip>192.168.1.3</ip>
70              <ip>127.0.0.1</ip>
71              <port>2022</port>
72              <commands>
73                  <command name="sprec_na" answer="False" parameters="True" priority="True" />
74                  <command name="spr_status" answer="True" timeout="2000" priority="True" />
75                  <command name="spr_grammar" answer="True" parameters="False" timeout="5000"
                        priority="True" />
76                  <command name="spr_words" answer="True" timeout="2000" priority="True" />
77              </commands>
78          </module>
79      </modules>
80
81  </blackboard>
```

4 Blackboard messages

The communication between modules is achieved using 2 ways message-passing over TCP/IP and represents a function execution request made by a *consumer module* to the *host module*, called **Command**, and the execution result produced by the *host module* which will be sent back to the *consumer module* and is called **Response**. Both **Command** and **Response** are called **Messages**.

4.1 Command and Response message format

All messages are serialized as an ASCII null-terminated string with the following format:

Commands

```
SOURCE DESITNATION command_name "parameters" @id
```

Responses

```
SOURCE DESITNATION command_name "parameters" result @id
```

Under this schema, commands may be seen as functions which receives a variable number of parameters serialized as a string and passed by reference, and which returns a boolean value indicating if the execution was successfull or not. Parameters may be modified after the command execution, even if the function received no parameters it can produce new parameters (or modify the existing) as result of the execution, even if the execution was not successful.

Each component of a message is separated with the ASCII 0x20 (white space) character and briefly explained below:

- `SOURCE` The name of the module which sends the command/response. Rules for module names detailed in 3.1.2 apply.
- `DESTINATION` The name of the destination module for the command/response. Rules for module names detailed in 3.1.2 apply.
- `command_name` For commands it is the name of the command to be executed, or the name of the executed command for responses. Rules for command names detailed in 3.1.2 apply.
- `parameters` For commands it is the arguments or parameters (if any) required to execute the command. For responses it includes the command execution results. Double quotes must be escaped with the ASCII 0x5C (backslash '\') character.
- `result` In a response it indicates if the command was executed successfully or not. Possible values can be either `1` or `0`.
- `id` A two-digits numeric identifier for the command-response pair.

Detailed explanation of each element can be found in the following subsections.

4.1.1 Message source

In bidirectional connection mode (see section 3.1) the Blackboard determines the source module of a message. If the source module name specified in the message and the source module name determined by the Blackboard does not match, the message is discarded, thus, the use of the source module component should be avoided in bidirectional connection mode.

In unidirectional connection mode (see section 3.1) the Blackboard can not determine the source module of a message, and it shall be provided by the sender. Thus, the use of the source module component is mandatory in bidirectional connection mode.

If the Blackboard receives through the Input Server a message from a module to which is not connected, the message is enqueued and redirectioned normally, even when the Blackboard can not send messages to that module. Although it might seem a vulnerability this feature was planed to be exploited during test to inject both, command and responses to the Blackboard.

4.1.2 Message destination

The message destination is an optional component of a message and may be specified only if also the source of the message is specified. It indicates to the Blackboard that the message shall be delivered only to the specified module.

The Blackboard determines automatically the destination module for a message and compares it with the destination module specified in the message itself (if present). If the destination module name specified by the message and the destination module name determined by the Blackboard does not match, the message is not sent; therefore the use of the destination module component should be avoided.

4.1.3 The command name

For a command, the *command name* represents the name of the command (function) that must be executed. For a response the *command name* represents the name of the executed command (function).

As it is specified in section 3.2.3, the command names must be unique in the whole system. This to keep order in the system and because the Blackboard uses the *command name* to determine the destination module for a command and redirect to the source module the response generated for that command.

Command names are string with a length of at least 3 characters, must start with a lowercase letter and be made of lowercase letters, digits and underscores.

4.1.4 The command parameters

The *parameters* is an semi-optional component of a message that must be enclosed within the ASCII 0x22 (double quotes '"') character. If, for a given command, the parameters are set as mandatory in the Blackboard's configuration file and this component is absent or empty (two double quotes together); then the Blackboard generates a failure response and sends it to the source of the command. Further than this, the Blackboard ignores the content of the *parameters* component.

In a command, the *parameters* component contains all the arguments required to execute the function that the command represents. In a response, the *parameters* component contains the result of the execution of the function that the requested command represents. Due to the diversity of parameters which can be serialized, function overload is implicitly supported.

When serializing floating point numbers it is used to write up to four decimal characters. If greater precision is required, it is advisable to express those numbers in hexadecimal format.

In general terms, the *parameters* component of a message can not contain the ASCII 0x22 (double quotes '"') character since it is its delimiter. Also, any other ASCII character less than 0x22 or greater than 126 should be avoided. However those characters may be included if they are escaped with the ASCII 0x5C (backslash '\') character with the only exception of the ASCII NUL character which indicates the end of the string.

4.1.5 Command execution result

The *result* component of a message is used to diferentiate commands and responses. Only responses contains a *result* component which is an ASCII 0x31 (one '1') or an ASCII 0x30 (zero '0') which indicates if the requested command executed successfully or not, respectively.

4.1.6 Message ID

The *id* is an optional component of a message which aids the Blackboard to find the best response match for a command when multiple commands with the same name has been redirected. In principle, is mandatory that a response have the same *id* as the command that responds.

The *id* are always preceded by an ASCII 0x40 character (arroba '@') and is formed by two numerical digits in the range between ASCII 0x30 (zero '0') and ASCII 0x39 (nine '9')

4.2 Command/Response examples

The following are sample commands with its responses

mv The `mv "distance angle"` command orders the robot (MVN-PLN module) to turn the specified angle and then advance the specified distance. Values are in meters and radians.

```
mv "3.1415 1.0000" @0
mv "3.2000 0.9708" 1 @0
```

mp_stop The `mp_stop` command orders the robot (MVN-PLN module) to stop navigating. If the robot is not moving it has no effect.

```
mp_stop
mp_stop 1
```

spg_voice The `spg_voice` has two overloads. When no parameters are provided or the parameters are the "get" string, it queries the speech generator module (SP-GEN) for the selected voice name. For other parameters values, it request the speech generator module to select the voice specified in the parameters.

```
spg_voice
spg_voice "Susan" 1

spg_voice "0.5 alpha Q" @25
spg_voice "0.5 alpha Q" 0 @25

spg_voice "Dave"
spg_voice "Dave" 1
```

4.3 Monitoring commands

The blackboard uses a small set of spetial commands to monitor the status of the connected modules. Those commands has a double function as it is explained in section .

Each monitoring command is briefly explained below:

- `alive` Used to check if the module is responding.
- `bin` Reserved (unimplemented).
- `busy` Queries the module if it is still busy.
- `ready` Queries the module if it is ready for normal operation.

The status of the modules is polled in a strict order as follows

1. `busy`
2. `ready`
3. `alive`

Detailed explanation of each monitoring command can be found in the following subsections.

4.3.1 alive

Every 10 seconds the Blackboard sends an `alive` command to each idle module to check that the module is receiving commands and responding normally. Every time the blackboard receives data from a module (even if the received data is discarded), the idle time counter is reseted and 10 seconds must pass before an alive command is sent.

It does not matter what the module answer to an `alive` command since that indicates that the module is able to send data to the blackboard. However an `alive 1` response is expected.

A module can send an `alive 1` response at any time to indicate that it is still alive and operational.

4.3.2 busy

If a module is executing a normal-priority command, the Blackboard will send a `busy` command insted of an `alive` or `ready` command in order to check if the module is still busy. It is expected that the module answer with a `busy 1` response to indicate that the module is still busy, however if a `busy 0` response is received, the Blackboard will set the module as *not busy* and will allow it to receive another normal-priority commands.

A module can send an `busy 1` response at any time to indicate that cannot receive normal-priority commands.

4.3.3 ready

When the Blackboard connects to a module it sends a `ready` command to query the module if it is ready for normal operation. After that, the Blackboard will keep sending `ready` commands insted of `alive` commands until the module becomes ready for normal operation.

A module can send an `ready 1` or a `ready 0` response at any time to indicate its *ready* status. If a module closes the connection with the Blackboard its *ready* flag is reseted.

4.4 Blackboard builtin commands

As it is explained in section 3.1, all the components of the architecture are modules, even the Blackboard itself. The Blackboard has a *Virtual Module* which supports several built in commands known as *Blackboard Commands* which are briefly explained below:

- `alive` Queries the Blackboard for a list of *alive* modules.
- `bbstatus` Queries the Blackboard for any of the Blackboard environment variables.
- `bin` Queries the Blackboard for the list of modules which support binary message format.
- `busy` Queries the Blackboard for a list of *busy* modules.
- `connected` Queries the Blackboard for a list of *connected* modules.
- `idletime` Queries the Blackboard for the amount of time (in seconds) a module has been idle.
- `modules` Queries the Blackboard for the list of all modules in the system.
- `mstatus` Queries the Blackboard for the status of a connected machine (battery charge, cpu usage, memory usage, disk usage, etc).
- `mystat` Notifies Blackboard the status of a connected machine (battery charge, cpu usage, memory usage, disk usage, etc).
- `querymodule` Queries the Blackboard for full information about a module
- `ready` Queries the Blackboard for *ready* modules.
- `reset` Reinicia el elemento indicado en los parámetros (Blackboard, test, tiempo de ejecución, módulo especificado).
- `restart_test` Dispara el evento `Restart-Test` en el módulo especificado.
- `setupmodule` Changes the connection parameters for an existing module

Detailed explanation of each *Blackboard Command* can be found in the following subsections.

References

[1] Mary Shaw and David Garlan. An introduction to software architecture. 1994.

[2] Mary Shaw, David Garlan, Robert Allen, Dan Klein, John Ockerbloom, Curtis Scott, and Marco Schumacher. Candidate model problems in software architecture. 1995.

[3] Dean Milojicic, Vana Kalogeraki, Rajan Luko, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rillins, and Zhichen Xu. *Peer-to-Peer Compuing*. HP Laboratories Palo Alto, 2003.

[4] Frank Buschmann, Regine Meunier, Hans Ronhert, Peter Sommerland, and Michael Stal. *Pattern-Oriented Software Architecture. A system of Patterns*. John Wiley & Sons, Ltd., 1996.

[5] Ian. Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, 2007.

[6] D. Rudenko and A. Borisov. An overview of blackboard architecture application for real tasks. 2007.