A comparison of two Software Architectures for General Purpose Mobile Service Robots

Mauricio MATAMOROS*, Jesus SAVAGE† and Jorge Luis ORTEGA-ARJONA‡
* BioMechanical Engineering Department, 3mE, Delft University of Technology. Mekelweg 2, 2628 CD, Delft. The Netherlands.
† Posgrado de Ingeniería. Universidad Nacional Autónoma de México. Av. Universidad 3000, CP04510, México D.F., México.
‡ Facultad de Ciencias. Universidad Nacional Autónoma de México. Av. Universidad 3000, CP04510, México D.F., México

***Abstract*—This paper exposes a set of tools which can be used to quantitatively evaluate the required effort to update the software system that operates a general purpose service mobile robot. These tools are used to compare a Blackboard-based and a Peer-to-Peer architectures in the context of mobile robotics. The analysis consider the development cost for an update, as well as the response time for both architectures. The results show that it is regularly simpler to maintain a robotics software system with Blackboard architecture than when a Peer-to-Peer architecture is used. Also results show that there is no noticeable change in the response time or *performance* of the robot when using any architecture.[1]**

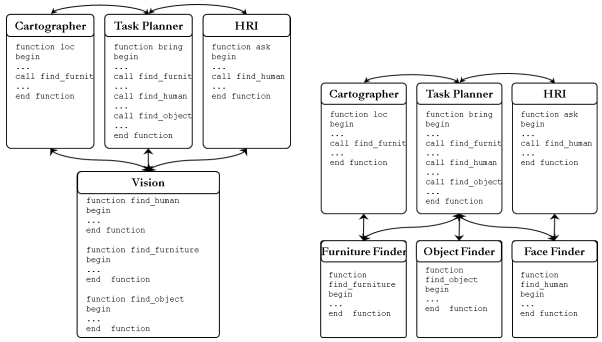
I. Introduction

A robot which interacts with humans and their environment often needs to solve several complex tasks, which requires many hours of complex software development [1], [2], [3]. During development, several algorithms and approaches have to be tested, to solve each part of the overall task the robot has to accomplish. Hence, robotics programming is commonly performed by dividing the software in several highly-specialized modules, which often require to be modified over time: software modules have to be frequently replaced, split, merged or updated. Since these modules require to communicate among themselves, any changes in a module commonly requires testing and modifying several other modules. This represents a necessary but tedious and time-consuming task.

As an example, consider a *Vision* module (Figure 1(a)), which performs all image processing for a robot's vision and has grown too big. In order to keep the maintainability of the software system, the *Vision* module is going to be divided into three specialized sub-modules (Figure 1(b)). Nevertheless, other three modules depend on the data produced by the *Vision* module: the *Task Planner*, the *Cartographer* and the *Human-Robot Interaction* (or *HRI*). Once the *Vision* module is divided, the three resulting modules need to be modified to obtain the information from all the appropriate modules with which *Vision* used to communicate.

During software development, time matters. It ts very inefficient to keep developers fixing coupling errors every time the architecture changes. Hence, applying some software architecture concepts and practices for the connection between modules, tend to reduce the update and test times of components. For example, developing loosely-coupled modules allow to exchange modules with a minimal number of modifications, or even, no modifications at all. On the other hand, the response time of any robotics software is very important. Tightly-coupled software systems tend to provide a short response time, but they also tend to be difficult to develop or maintain. Changing or updating one or several components in any tightly-coupled software system is a very painful, time-consuming task.

[1]This work was supported by DGAPA UNAM under Grant PAPIIT IG100915 and PAPIME PE102115



(a) The system before *Vision* module specialization. (b) The system after *Vision* module specialization.

Fig. 1. Specialization of the *Vision* module. Lines represent the data flow between the different modules.

In the distributed systems literature, it is generally assumed that a peer-to-peer architecture (one with direct connections between modules) has faster response than a centralized architecture [4], [5], [1], [6]. On those studies, computer power is not a constrain, and powerful tools such as redundancy are available. As result, peer-to-peer architecture is the actual choice for many robotics software development such as ROS (Robot Operating System) [1], one of the most popular robotic frameworks used nowadays. However, no quantitative comparisons on response time between peer-to-peer and centralized architectures have been performed. Measuring such a gain in response time deserves the effort of developing a tightly-coupled, complex architecture such as peer-to-peer, and the development time invested on it. The importance grows considering the constraints of embedding the software within a mobile robot.

Thus, this paper proposes two comparisons:

- The first comparison estimates the complexity of updating a peer-to-peer architecture against a Blackboard architecture, due to changes on one of its components.
- The second comparison estimates the relative response time of a peer-to-peer architecture against a centralized Blackboard architecture, both in the context of general purpose service mobile robotics.

II. The Vision Module Example Revisited

Let us consider the *Vision* module specialization example in further detail. This module executes three functions:

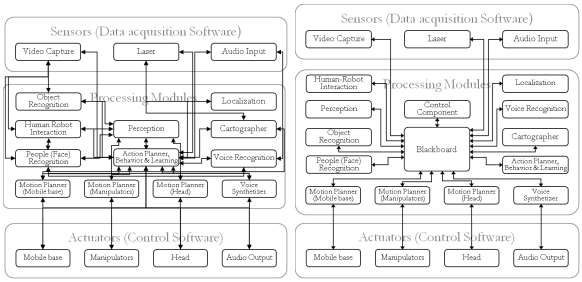
- `find_human`. A human face recognition based human detector, which is used by *Task Planner* and *HRI* modules;

978-1-4673-6991-6/15 $31.00 © 2015 IEEE
DOI 10.1109/ICARSC.2015.10

- `find_furniture`. A shape and texture based large object detector, which is used by *Cartographer* module; and
- `find_object`. A SIFT/SURF and geometry based small object detector, which is used by *Task Planner* module.

These dependencies are explicit bidirectional connections for communication between the *HRI*, the *Task Planner*, the *Cartographer* modules, and the *Vision* module (Figure 1). When the *Vision* module is divided into the *Furniture Finder*, *Object Finder*, and *Face Finder* modules, the other three dependent modules need to be updated. So, the *Cartographer* module now should connect with the *Furniture Finder* module; the *HRI* module now should connect with the *Face Finder* module; and the *Task Planner* module should connect with the *Furniture Finder*, the *Object Finder* and the *Face Finder* modules. This last update is quite difficult, since the *Task Planner* module now needs to handle three connections, and implement protection methods for three potential sources of failures (like a broken connection) instead of one.

Also, the *Task Planner* module is going to be moved to another computer to balance the CPU load and increase the response time. However, *Task Planner* is connected with other five modules. Thus, each dependence has to be updated, and so, five changes are required, which the developers have to update (Figure 1(b)).



(a) A system with a *Peer-to-peer* architecture for connections.

(b) A system with a *Blackboard* based architecture for connections.

Fig. 2. Specialization of the *Vision* module. Lines represents the data flow between modules.

Consider a similar scenario with a centralized architecture where all requests are made to a central component (*Blackboard*). Here, the *HRI*, *Task Planner*, *Cartographer* and *Vision* modules are connected to each other through the *Blackboard* module. In such scenario, the *Task Planner* doesn't need to be aware of the existence of any other module but the *Blackboard* module, and if it requires some data, the *Blackboard* module should provide it. The same applies any other module, but the *Blackboard* module, which need to know every one. So, if a module is replaced by another one, or is divided into several sub-modules, or many modules are merged into a single super-module, the *Blackboard* module is the only component to be updated.

In contrast with *peer-to-peer*, notice that in *Blackboard* the number of connections increases in one per module. There is no need of considering interdependences between modules, since all modules depend on the Blackboard module and it will provide the requested data. Updating or trying to follow the connection diagram is quite simple (Figure 2(b)).

III. Comparing Architectures for a Robotics Software System

A quick view of Figure 2(a) and Figure 2(b) shows that a Blackboard-based architecture is simpler than a Peer-to-Peer architecture, providing an intuitive notion that it is also easier to maintain. Also, it seems that the Blackboard component, centralized as it is, seems also to be a bottleneck. This may impact on the response time. Hence, the main objective of this paper is to test whether using a Blackboard-based architecture offers better extensibility, restructuring and a similar response time, compared against a Peer-to-Peer architecture when the granularity of the software system is medium or coarse. The idea is that a Blackboard-based architecture is simpler to develop than a peer-to-peer one, with no noticeable difference in the response time for general purpose service mobile robots. For this, the following definitions are used:

- Extensibility focuses on the extension of a software system with new features, as well as the replacement of components with improved versions, and removal of unwanted or unnecessary features and components [7].
- Restructuring deals with the reorganization of the components of a software system and the relationships between them; for example, when changing the placing if a component by moving it to a different subsystem [7].
- Response time is the ability of the software product to provide appropriate response, processing time, and throughput rates, when performing its function under stated conditions [8].

Extensibility and restructuring are features related to the build-time of the software system, whereas response time is a run-time feature. To evaluate extensibility and restructuring, several change scenarios are applied for both architectures. Each update task is modeled as an algorithm, in which each step is an atomic operation. By considering the number of functions and modules involved as the data length, the complexity of each algorithm is calculated and compared for both architectures. Comparison results are shown in Table I.

The analyzed scenarios for extensibility and restructuring are described below. For these, consider that *context* means the hardware on which the module is executed.

- *Exchange of equivalent modules*. An old module is replaced by a new one on the same context (Extensibility).
- *Change the definition of a function in a module* (Extensibility).
- *Move a function from one module to another* (Restructuring).
- *Function specialization*. A function is divided into two or more specialized functions in the same module (Restructuring).
- *Function generalization*. Two or more functions in the same module are merged into a new, more general function (Restructuring).
- *Move a module to another context* (Restructuring).
- *Module specialization*. A module is divided into two or more specialized modules on the same context (Restructuring).
- *Module generalization*. Two or more modules on the same context are merged into a new more general module (Restructuring).

For comparing response time, each local function is considered to be executed in a constant time, adding the execution time and average communication time for each remote dependency. This can be simply reduced to the execution time of the function plus the communication time. The response time analysis is carried out for both architectures, and results are shown in Table II.

IV. Analysis of Extensibility, Restructuring and Response Time

A software system used to control a mobile robot, designed for human interaction, is analyzed for Extensibility, Restructuring, and Response Time, for both peer-to-peer and Blackboard architectures. Extensibility and Restructuring impact only in the build-time of the software system, which here it is intended to be minimal. On the other hand, response time impacts only during run-time, and it is expected to be similar for both architectures, with no noticeable effects for humans.

A. A Formal Description of a Robotics Software System

For the present analysis, the following definitions are required:

A system S is $S = (M, C)$ where

- M the set of modules of S, such as $M = \{m_1, m_2, ..., m_n\} \neq \emptyset$.
- C the set of connections between modules such as $C \subseteq \{(m_i, m_j) \in M \times M \mid m_i \neq m_j\}$.

Also, a module m of the system S is defined as $m = \{f_1^m, f_2^m, ..., f_n^m\}$, where:

- $\overline{y}_i = f_i^m(\overline{x}_i)$ is the i-th function of the function set of the m module.
- $\overline{y}_i$ is the result produced after the execution of the function $f_i^m(\overline{x}_i)$
- $\overline{x}_i$ is the set of parameters required by the function f_i^m to be executed.

For convenience, the following notations provide information about the software system:

- $\{m_i\}$ is the set of modules of the system.
- $|\{m_i\}|$ is the number of modules of the system.
- $D(m_i)$ is the dependence function D, which returns the set of modules that depend on any function $f_j^{m_i}$ of the module m_i.
- $\{f_j^{m_i}\}$ is the set of functions in the module m_i.
- $|m_i| = |\{f_j^{m_i}\}|$ is the number of functions in the module m_i.
- $D(f_j^{m_i})$ is the dependence function D, which returns the set of modules that depend on the function $f_j^{m_i}$ of the module m_i.
- $m_k \rightarrow f_j^{m_i}$ implies that the module m_k calls the function $f_j^{m_i}$ of module m_i.
- $t(f_j^{m_i})$ is the amount of time required by m_i to execute its own function $f_j^{m_i}$.
- $t(c_{m_i}, c_{m_j})$ is the communication time between the modules m_i and m_j.
- $t(m_k, f_j^{m_i})$ is the amount of time required by m_k to execute the function $f_j^{m_i}$ of module m_i.

Assuming that $t(c_{m_i} \approx c_{m_j}) \cong t_k$, where t_k is a constant equivalent to the average one-way communication time between two modules directly connected, and considering that every communication between two modules is achieved by a request and a response (bidirectional communication), it is possible to model the average communication time between modules as $t_c = 2t_k \cong t(c_{m_i} + c_{m_j})$.

B. Analysis

For Extensibility the following test scenarios are considered:

- Function change: modify the definition of the function of a module.
- Module swap: exchange a module for another module in the same context, which performs equivalent functions, with different signatures or execution time.

For Restructuring the following test scenarios are considered:

- Function relocation: move a function from one module to another.
- Split function: divide a function into two or more specialized functions.
- Join functions: gather two or more functions into one generalized function.
- Module relocation: move a module from one context to another.
- Split module: divide a module in two or more specialized sub-modules using function relocation.
- Join modules: gather two or more modules into one generalized super-module using function relocation.

1) Extensibility and Restructuring Analysis for a Peer-to-Peer architecture: Algorithms 1 and 2 represent the scenarios of Function Change for Extensibility. When a function in a module is modified, its signature or execution time also changes. Thus, changes in the dependencies must be performed. Algorithm 1 is proposed to depict a change in the signature for a peer-to-peer architecture.

Algorithm 1 Function (signature) change in Peer-to-Peer: $O(n^2)$

```
for each module m_j ∈ D(f^{m_i}) do
    for each call to f^{m_i} ∈ m_j do
        Change call to f^{m_i} with a call to its equivalent g^{m_i}
    end for
end for
```

Taking $n = max(|D(f^{m_i})|, |m_j \xrightarrow{call} f^{m_i}|)$, the complexity of this algorithm is $O(n^2)$. Supposing that *(a)* the difference is the execution time, *(b)* each module has a wrapper when the remote call is performed, and *(c)* this validation occurs only once, the update algorithm can be reduced as shown in Algorithm 2. In this case, $n = |D(f^{m_i})|$, and thus, the complexity of this algorithm is $O(n)$.

Algorithm 2 Function (execution time) change in Peer-to-Peer: $O(n)$

```
for each module m_j ∈ D(f^{m_i}) do
    Update the execution time for f^{m_i} with the time required by g^{m_i}
end for
```

Algorithms 3 and 4 represent the scenarios of Module Swap for Extensibility. Here, when a module is replaced by another module with exactly the same functions (this is, the same signature and the same execution time), the case reduces, and there is properly nothing to do. However, if there are differences in the signature, changes in the dependencies have to be performed. Algorithm 3 shows the example of a module m_2 replacing a module m_1 for a peer-to-peer architecture.

Algorithm 3 Module Swap in Peer-to-Peer: $O(n^3)$

```
for each function f_i^{m_1} ∈ m_1 do
    for each module m_j ∈ D(f_i^{m_1}) do
        for each call to f_i^{m_1} ∈ m_j do
            Change call to f_i^{m_1} with a call to its equivalent f_i^{m_2}
        end for
    end for
end for
```

Again, when there are *(a)* differences in the execution time, and *(b)* assuming that this validation is made only once in each dependent module, the update algorithm is reduced as shown in Algorithm 4.

Algorithm 4 Module Swap in Peer-to-Peer: $O(n^2)$

```
for each function f_i^{m_1} ∈ m_1 do
    for each module m_j ∈ D(f_i^{m_1}) do
        Update the execution time for f_i^{m_1} with the time required by f_i^{m_2}
    end for
end for
```

Algorithm 5 represents the scenario of Function Relocation for Restructuring. Here, function $f_i^{m_1}$ is moved from module m_1 to module m_2 for a peer-to-peer architecture.

Algorithm 5 Function Relocation in Peer-to-Peer: $O(n^2)$

```
for each module m_j ∈ D(f_i^{m_1}) do
    for each call to f_i^{m_1} ∈ m_j do
        Change call to f_i^{m_1} with a call to its equivalent f_i^{m_2}
    end for
end for
```

Algorithm 6 represents a scenario of Join Functions for Restructuring. Here, functions $f_i^{m_1}(\overline{x}) = g_1^{m_1}(\overline{x_1}) \circ g_2^{m_1}(\overline{x_2}) \circ \cdots \circ g_n^{m_1}(\overline{x_n})$ in module m_1 are gathered together as function $f_i^{m_1}$. Each call to this set of functions needs to be replaced with calls to $f_i^{m_1}$. Algorithm 6 shows this change for a peer-to-peer architecture.

Algorithm 6 Function merge in Peer-to-Peer: $O(n^2)$

```
for each module m_j ∈ D(f_i^{m_1}) do
    for each call to f_i^{m_1} ∈ m_j do
        Change call to g_1^{m_1}(x_1), ..., g_n^{m_1}(x_n) with a calls to f_i^{m_1}.
    end for
end for
```

Split Functions for Restructuring in Peer-to-Peer is similar to joining. Here, function $f_i^{m_1}$ in module m_1 is divided into several

functions, in a such way that $f_i^{m_1}(\overline{x}) = g_1^{m_1}(\overline{x_1}) \circ g_2^{m_1}(\overline{x_2}) \circ \cdots \circ g_n^{m_1}(\overline{x_n})$. Each call to $f_i^{m_1}$ needs to be replaced with calls to $g_1^{m_1}(\overline{x_1}) \circ g_2^{m_1}(\overline{x_2}) \circ \cdots \circ g_n^{m_1}(\overline{x_n})$. Splitting requires the same number of operations than joining: $O(n^2)$; however, if a wrapper is used to access this function, and no further changes have to be made, then it only has to be changed once in each dependent module. Thus, the complexity becomes linear.

Algorithm 7 represent a scenario of Module Relocation for Restructuring. When a module is moved to another context, it is necessary to update each module that is dependent on a function that belongs to the module being moved with the new module location. Algorithm 7 shows the proposed modification for a peer-to-peer architecture.

Algorithm 7 Function relocation in Peer-to-Peer: $O(n)$

1: **for each** call to $m_i \in D(m_1)$ **do**
2: Update the access point to module m_1.
3: **end for**

Algorithm 8 represents a scenario of Split Module for Restructuring. Based on the Function Relocation algorithm (Algorithm 5), dividing a module into several specialized modules is achieved by moving the smallest function subset in the original module to a new module. Algorithm 8 shows this modification for a peer-to-peer architecture.

Algorithm 8 Split a module into sub-modules in Peer-to-Peer: $O(|F_R| \cdot n^2)$

1: **for each** function to relocalize in $f_i^{m_1}$ **do**
2: **for each** module $m_j \in D(f_i^{m_1})$ **do**
3: **for each** call to $f_i^{m_1} \in m_j$ **do**
4: Change call to $f_i^{m_1}$ with a call to $f_i^{m_2}$
5: **end for**
6: **end for**
7: **end for**

Similar to splitting, merge several modules into a single generalized super-module is achieved by moving functions from all modules to be join into a module. It requires the same amount of operations: $O(|F_R| \cdot n^2)$.

2) Extensibility and restructuring analysis in Blackboard-based architecture: Algorithm 9 represents the scenario of Function Change for Extensibility. When a function in a module is modified, its signature or execution time may also change. However, since this information is stored in the blackboard, this is the only component which needs to be updated. Algorithm 9 shows the proposed algorithm for a change in the signature for a blackboard-based architecture.

Algorithm 9 Function change in Blackboard: $O(1)$

1: Update the signature of f^{m_i} in the blackboard and control component modules.
2: Update the execution time of f^{m_i} in the blackboard and control component modules.

If the blackboard component and its control component are separated, only four changes are required; if these components are together, only one change is required. Thus, the complexity of this algorithm is $O(1)$.

Algorithm 10 represents the scenario of Module Swap for Extensibility. When a module is replaced with another module with exactly the same functions (same signature and execution time), the case is trivial, and there is nothing to do. However, if there are differences in the signature, changes in the dependencies have to be made. Algorithm 10 shows the case in which a module m_2 replaces another module m_1.

Algorithm 10 Module swap in Blackboard: $O(n)$

1: **for each** function $f_i^{m_1} \in m_1$ **do**
2: Update signature of $f_i^{m_1}$ in the blackboard and control component modules.
3: Update the execution time of $f_i^{m_1}$ in the blackboard and control component modules.
4: **end for**

Algorithm 11 represents the scenario of Function Relocation for Restructuring. This change considers moving the function $f_i^{m_1}$ from module m_1 to module m_2. Algorithm 11 shows this for a blackboard-based architecture.

Algorithm 11 Function relocation in Blackboard: $O(1)$

1: Update ownership of $f_i^{m_1}$ from $f_i^{m_1}$ to $f_i^{m_2}$ in the blackboard and control component modules.

Algorithm 12 represents the scenario of Split Function for Restructuring. Function $f_i^{m_1}$ in module m_1 is divided into several functions, in such a way that $f_i^{m_1}(\overline{x}) = g_1^{m_1}(\overline{x_1}) \circ g_2^{m_1}(\overline{x_2}) \circ \cdots \circ g_n^{m_1}(\overline{x_n})$. Each call to $f_i^{m_1}$ needs to be replaced with calls to $g_1^{m_1}(\overline{x_1}) \circ g_2^{m_1}(\overline{x_2}) \circ \cdots \circ g_n^{m_1}(\overline{x_n})$. Algorithm 12 shows this for a blackboard-based architecture.

Algorithm 12 Function split in Blackboard: $O(n^2)$

1: **for each** module $m_j \in D(f_i^{m_1})$ **do**
2: **for each** call to $f_i^{m_1} \in m_j$ **do**
3: Change call to $f_i^{m_1}$ with calls to $g_1^{m_1}(\overline{x_1}), \ldots, g_n^{m_1}(\overline{x_n})$
4: **end for**
5: **end for**

However, if a wrapper is used to access this function, and no further changes have to be performed, it only has to modify each dependent module only once, and so, the complexity becomes linear.

Similar to splitting is joining functions. Functions $f_i^{m_1}(\overline{x}) = g_1^{m_1}(\overline{x_1}) \circ g_2^{m_1}(\overline{x_2}) \circ \cdots \circ g_n^{m_1}(\overline{x_n})$ in module m_1 are gathered in function $f_i^{m_1}$. Each call to this set of functions needs to be replaced with calls to $f_i^{m_1}$. The complexity of the algorithm is the same as joining: $O(n^2)$.

Algorithm 13 represents a scenario of Module Relocation for Restructuring. If a module is moved to another context, and since it only connects with the blackboard, then the blackboard is the only component that needs to be updated with the information about the new location of the module. Algorithm 13 shows this case for a blackboard-based architecture.

Algorithm 13 Function relocation in Blackboard: $O(1)$

1: Update the access point to module m_1 in blackboard and control component modules.

Algorithm 14 represents the scenario of Split Module for Restructuring. Based on the Function Relocation algorithm (Algorithm 11), dividing a module into several specialized modules is achieved by moving the smallest function subset in the original module to a new module as follows. Algorithm 14 shows this for a blackboard-based architecture.

Similar to splitting, merge several modules into a single generalized super-module is achieved by moving functions from all modules to be join into a module. It requires the same amount of operations: $O(|F_R|)$.

Algorithm 14 Split a module into sub-modules in Blackboard: $O(|F_R|)$

1: **for each** function to relocalize in $f_i^{m_1}$ **do**
2: Update in blackboard the reference $f_i^{m_1}$ with $f_i^{m_2}$
3: Update in the control component the reference $f_i^{m_1}$ with $f_i^{m_2}$
4: **end for**

C. Comparison results

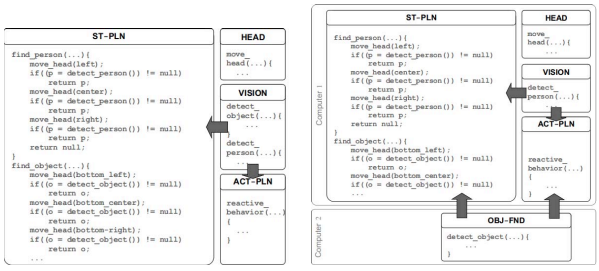
Table I summarizes the complexity of each one of the modifications, expressed as algorithms, for both peer-to-peer architecture and blackboard-based architecture.

Scenario	Peer-to-Peer	Blackboard
Exchange of equivalent modules	$O(n^3)$	$O(n)$
Function definition change	$O(n^2)$	$O(1)$
Function relocation	$O(n^2)$	$O(1)$
Function specialization	$O(n^2)$	$O(n^2)$
Function generalization	$O(n^2)$	$O(n^2)$
Module relocation	$O(n)$	$O(1)$
Module specialization	$O(\|F_R\| \cdot n^2)$	$O(\|F_R\|)$
Module generalization	$O(\|F_R\| \cdot n^2)$	$O(\|F_R\|)$

TABLE I. COMPARISON OF THE EXTENSIBILITY AND RESTRUCTURING FEATURE ANALYSIS OF PEER-TO-PEER AND BLACKBOARD ARCHITECTURES.

D. An Example: Specialization of the Vision module

Continuing with the example of the *Vision* module which is going to be divided into a person detector module (with the same name: *Vision*) and an object detector module (*OBJ-FND*). The vision module and its dependencies *ACT-PLN*, *ST-PLN*, and *HEAD*) will be kept in the same computer, while the (*OBJ-FND*) module will be executed on a second faster computer (see Figure 3).



(a) Initial scenario prior to the specialization of the *VISION* module. The *ACT-PLN* and *SP-PLN* modules depends on *VISION*'s functions.

(b) Final scenario (after the specialization of the *VISION* module). The *ACT-PLN* and *SP-PLN* modules now depends on *VISION* and *OBJ-FND* modules.

Fig. 3. Specialization of the *Vision* module. Lines represents data dependency.

In order to achieve this, the *Vision* module goes through a modular specialization, followed by a module relocation. Also, the execution time of the function changes, so it is required an update of the function definition.

Specializing a module may be seen as creating a new empty module and move the smallest function subset into it. For this example, the smallest function subset consists only in the `detect_person` function which is moved to the empty *OBJ-FND* module. After this, settings and references must be updated:

1) Update the system of the `detect_person` to the *OBJ-FND* module.
2) Update the system location of the *OBJ-FND* module.
3) Update the system execution time of the `detect_person` function.

1) Specialization of the Vision module under a Peer-to-Peer architecture: Using algorithm 8 to divide a module for a peer-to-peer architecture, `detect_object` is the only function to relocalize from *Vision* to *OBJ-FND*. Algorithm 15 shows the sequence of steps for a peer-to-peer architecture.

Algorithm 15 Vision specialization (Peer-to-peer)

1: **for each** module $m_j \in D(detect\_object^{VISION})$ **do**
2: **for each** call to $detect\_object^{VISION} \in m_j$ **do**
3: Change call to $detect\_person^{VISION}$ with a call to $detect\_person^{OBJ-FND}$
4: **end for**
5: **end for**

The dependencies of the function $detect\_object^{VISION}$ are given by the function $D(detect\_object^{VISION}) = \{ACT - PLN, ST - PLN\}$. Analyzing the pseudo code shown in Figure 3, the module *ACT-PLN* performs one call to `detect_object`, while the module *ST-PLN* calls it three times, so the number of changes by now is four.

Next step consists of adding or updating the dependencies of `detect_object` (*ACT-PLN* and *ST-PLN*). The location of *OBJ-FND* is moved from computer 1 to computer 2 (change of context). This is a single change on each module, two more changes for a total of 6.

Finally, since the execution time of `detect_object` has changed, the dependencies need to be updated, adding 2 more changes for a total of 8. Supposing that each changes takes 5 minutes to be performed, updating the whole system requires about 40 minutes.

2) Specialization of the Vision module under a Blackboard architecture: Using Algorithm 14 to divide a module of a Blackboard-based architecture, the only function to relocalize is `detect_object` from *Vision* to the *OBJ-FND*. Algorithm 16 shows the sequence of steps to achieve this for a Blackboard-based architecture.

Algorithm 16 Vision specialization (Blackboard)

1: Update in blackboard the reference $detect\_person^{VISION}$ with $detect\_person^{OBJ-FND}$
2: Update in the control component the reference $detect\_person^{VISION}$ with $detect\_person^{OBJ-FND}$

Hence, only two changes are required for update the system. Next step consist on adding or updating the *Blackboard* with the new location of *OBJ-FND*, which is moved from computer 1 to computer 2 (change of context). The total of changes by now is 3.

Finally, since the execution time of the `detect_object` changes, the new value has to be updated in the *Blackboard* module, adding one more change to the number of changes, for a total of 4. Supposing that each changes takes about 5 minutes, updating the system requires about 20 minutes.

E. Response Time Analysis

Here, an analysis of the response time of both architectures is carried out, estimating the amount of time required to execute a function present in each architecture. Since the robot interacts with humans and operates in a human-friendly environment, the response time of the system is not critical, it has to be close to the human response time.

1) Performance Analysis for a Peer-to-peer Architecture: In a Peer-to-Peer architecture, if a module m_1 requires a function $f_i^{m_2}$ of another module m_2, it is assumed that there is a direct connection between those two modules. In other words, there is a $f_i^{m_1}(\overline{x})$ such as $f_i^{m_1}(\overline{x}) = g(\overline{x_1}) \circ f_j^{m_2}(\overline{x_2}) \Rightarrow \exists c_{m_1,m_2}$, where $\overline{x_1} \cup \overline{x_2} \subseteq \overline{x}$.

Based on Section IV-A, the execution time of a module m_1 requesting an execution of the function $f_i^{m_2}$ in a module m_2 is $t(m_1, f_i^{m_2}) = t(f_i^{m_2}) + t_c$. However, when the granularity of the system is coarse, this reduces to $t_c \ll t(f_i^{m_2})$ so $t(m_1, f_i^{m_2}) \cong t(f_i^{m_2})$. Notice that t_c is the average bidirectional communication time between two modules. Regardless the size of the data, and without considering network delays, broken packages or connections, message loss, etc.

2) Response Time Analysis for a Blackboard-based Architecture: In a blackboard-based architecture, all operations are carried out by reading and writing shared variables, stored within the blackboard component. Let $V?\{f_1^{m_1}, \ldots, f_p^{m_1}, \ldots, f_1^{m_n}, \ldots \ f_r^{m_n}\} \cup \{x_1, \ldots, x_m\} \cup \{y_1, \ldots, y_k\}$ be the set of all the shared variables and $\overline{v} = \{v_i, v_2, \ldots, v_n\} \in V$ be a vector of shared variables stored in the blackboard. Then, let be:

1) $read^{m_i}(\overline{v_j})$ a read of all the shared variables in $\overline{v_j}$ performed by m_i.
2) $write^{m_i}(\overline{v_j})$ a write of all the shared variables in $\overline{v_j}$ performed by m_i.
3) $t_r = t(read^{m_i}(\overline{v_j}))$ the amount of time for read the shared variables in $\overline{v_j}$.
4) $t_w = t(write^{m_i}(\overline{v_j}))$ the amount of time for write the shared variables in $\overline{v_j}$.

Since in a blackboard-based architecture all modules are connected through the blackboard, if a module m_1 requires a function $f_i^{m_2}$ from m_2, then it writes to the blackboard the execution request in a single write operation. Then, the blackboard sends all required data to m_2 (a read operation of the data is performed). Once the execution of $f_i^{m_2}$ finishes, m_2 writes back the results in the blackboard with a single write operation. The blackboard report these results back to m_1. With this schema, there are two write operations, two read operations and four communications between modules, so $t(m_1, f_i^{m_2}) = t(f_i^{m_2}) + 2t_r + 2t_w + 4t_c$. However, since the read and write operations are performed in local memory, which is by far much faster than message-passing communications, then $t_r \ll t_c$ and $t_w \ll t_c$. Since in comparison they are neglectable, it is valid that $t(m_1, f_i^{m_2}) \cong t(f_i^{m_2}) + 4t_c$.

Again, when the granularity is coarse, then $t_c \ll t(f_i^{m_2})$, and so $t(m_1, f_i^{m_2}) \cong t(f_i^{m_2})$, which is the same than in Peer-to-Peer.

3) Response Time Comparison: Here, the response time required for both architectures while executing the function $f_i^{m_2}$ of module m_2, when requested by module m_1 (denoted as $t(m_1, f_i^{m_2})$) is shown in Table II for different levels of granularity.

Granularity	Peer-to-Peer	Blackboard
Medium	$t(f_i^{m_2}) + t_c$	$t(f_i^{m_2}) + 4t_c$
Coarse	$t(f_i^{m_2})$	$t(f_i^{m_2})$

TABLE II. COMPARISON OF THE RESPONSE TIME OF PEER-TO-PEER AND BLACKBOARD ARCHITECTURES FOR EACH TYPE OF GRANULARITY.

V. CONCLUSIONS

Results for Extensibility and Restructuring in Table I, shows that the time required to update the robot's software often reduces when using a Blackboard architecture, or remains equal. Therefore, there is an important saving during the software build-time. Notice that in the example of Section IV-D, about 20 minutes when updating the robotics software system were saved.

For the analysis performed in Section IV, access to the source code is required. In practice, this assumption may not always true, making necessary to use wrappers or intermediary components, fact that is more critical for peer-to-Peer architectures than for blackboard-based architecture. Using middlewares or frameworks like CARMEN[9], MIRO[10], MOOS[11], OpenRDK[12], Orca[13], PLayer/Stage[14], and ROS [1] may aid to solve these problems at the cost of sharpening the learning-curve, or reduce the performance of the whole system.

In the summary of the response time analysis, shown in Table II, note that there is no noticeable difference between the response time of both architectures, and when the granularity of the system is medium, the difference tends to reduce as the granularity goes from coarse to medium.

Further research in this area includes the comparison of robustness, fault tolerance, maintainability, and adaptability among others. Also, hybrid architectures, may be analyzed.

REFERENCES

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[2] D. Calisi, A. Censi, L. Iocchi, and D. Nardi, "Openrdk: a modular framework for robotic software development," in *Proc. of Int. Conf. on Intelligent Robots and Systems (IROS)*, Sep. 2008, pp. 1872–1877.

[3] S. Enderle, H. Utz, S. Sablatnög, S. Simon, G. Kraetzschmar, and G. Palm, "Miro: Middleware for autonomous mobile robots," in *In Telematics Applications in Automation and Robotics*, 2001.

[4] I. Sommerville, *Software Engineering*, ser. International Computer Science Series. Addison-Wesley, 2007. [Online]. Available: http://books.google.com.mx/books?id=B7idKfL0H64C

[5] D. Milojicic, V. Kalogeraki, R. Luko, K. Nagaraja, J. Pruyne, B. Richard, S. Rillins, and Z. Xu, *Peer-to-Peer Compuing*. HP Laboratories Palo Alto, 2003.

[6] M. Matamoros, "Análisis de extensibilidad, reestructuración y desempeño de software para robots móviles," Master's thesis, Universidad Nacional Autónoma de México, 2013.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Chichester, UK: Wiley, 1996.

[8] F. Losavio and C. Isys, "Towards a standard eai quality terminology," in *Proceedings of the 23rd International Conference of the Chilean Computer Science Society (SCCC'03*. Society Press, 2003, pp. 119–129.

[9] D. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit," in *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, vol. 3, 2003, pp. 2436–2441 vol.3.

[10] S. Enderle, H. Utz, S. Sablatnög, S. Simon, G. Kraetzschmar, and G. Palm, "Miro: Middleware for autonomous mobile robots," *Telematics Applications in Automation and Robotics*, 2001.

[11] O. M. R. Group. (2008) The moos homepage. [Online]. Available: http://www.robots.ox.ac.uk/ mobile/MOOS/wiki/pmwiki.php

[12] D. Calisi, A. Censi, L. Iocchi, and D. Nardi, "Openrdk: a modular framework for robotic software development," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*. IEEE, 2008, pp. 1872–1877.

[13] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *International Conference on Intelligent Robots and Systems (IROS)*, 2006, pp. 163–168.

[14] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, vol. 1, 2003, pp. 317–323.