Using MARIE for Mobile Robot Software Development and Integration

Dominic Létourneau, Carle Côté, Clément Raïevsky, Yannick Brosseau, and François Michaud[1]

Université de Sherbrooke, Department of Electrical Engineering and Computer Engineering, Sherbrooke (Québec), CANADA
{Dominic.Letourneau, Carle.Cote, Clement.Raievsky, Yannick.Brosseau, Francois.Michaud}@USherbrooke.ca

1 Introduction

Integration is one the most fundamentals problem in designing autonomous mobile robots, especially for those that interact with people in real life settings. Such robots have to combine a multitude of capabilities such as navigation, localization and mapping, tracking and recognition, vision and audio processing, graphical or natural interaction, planning and reasoning using different abstraction levels. Rapid and specialized progress in a variety of the associated domains makes the simultaneous development of all these capabilities a very challenging task. Reimplementing them all is not recommended to make efficient progress in discovering the underlying issues with autonomous reasoning of mobile robots. Integration of available and useful software applications is a more compelling approach, allowing to build on top of validated implementations and design more sophisticated and complex systems.

Many existing programming environments, like Player [VGH03], CARMEN [MRT03], CLARity [NWB+03], OROCOS [OC03], SmartSoft [Sch03], MIRO [USEK02], ADE [AS04] and RCS [GMP+01], are all proposing different approaches for mobile robotics system development and integration. Most of them are incompatible with each other for different reasons [OC03], such as the use of specific communication protocols and/or mechanisms, different operating systems, robotics platforms, architectural concepts, programming languages, intended purpose, proprietary source codes, etc. This leads to code replication of common functionalities across different programming environments, and to specific functionalities being often restricted to one programming environment. The ability create shared software infrastructures among the robotics community, based on common requirements and objectives, is clearly an important goal to reach in order to avoid effort duplication [WMT03] and assist developers in their scientific and engineering work.

Identifying common requirements and objectives is challenging in the current context considering that the robotics field is still in an early exploration phase. A possible solution which could be valuable is to reuse existing programming environments and interconnect them through a system integration framework to benefit from their respective approaches, instead of having to choose only one of them. The main objective of system integration frameworks is to support one or many integration approaches (e.g. communication protocols, central repository, remote procedure call, dynamic and static linkage) to interconnect heterogeneous applications with their own set of concepts and requirements in a larger system.

MARIE (for Mobile and Autonomous Robotics Integration Environment) is a system integration framework oriented towards a rapid prototyping approach to development and integration of new and existing softwares for robotic systems [CLM+04] [CBL+06]. To achieve the integration challenge, MARIE proposes an extendable collection of blackbox and whitebox frameworks, as described in Section 2, allowing different development techniques to add new functionalities in the system. MARIE is designed according to three main software requirements:

1. **Reuse available solutions**. Integration of existing software components is difficult considering that they are typically developed independently, following their own set of requirements. Reusability in this context is challenging but crucial for the evolution of the field, avoiding the need for expertise in all the related areas that must be integrated.
2. **Support multiple sets of concepts and abstractions**. From high-level decision-making developers to perceptual processing and motor controllers designers, or from system analysts to testers, experts from many fields and with different objectives have to contribute concurrently on the same system. To cope with such multidisciplinary software development effort, multiple sets of concepts and abstractions need to be supported.
3. **Support a wide range of communication protocols, communication mechanisms and robotics standards**. No unified protocol or consensus has yet emerged from the robotics software community on standards to adopt. The robotics community still has to explore a great variety of ideas, application areas (each one having its own set of constraints, e.g., space, military, human-robot interaction) and to cope with continuously evolving computing technologies. Consequently, being able to interchange communication protocols mechanisms and upcoming robotics standards easily, without major code refactoring, means longer life cycle for actual and future implementations.

The following sections present how MARIE follows these software requirements to create a middleware framework, a kind of system integration framework, providing tools to create specialized middlewares for dedicated applications. MARIE's efforts have been focused on distributed robotics component-based middleware framework development, enhancing reusability of applica-

tions and providing tools and programming environments to build integrated and coherent robotics systems. Section 2 presents MARIE's software architecture in relation to the targeted software requirements. It situates the three principal frameworks used in MARIE: the Component Framework (Section 3), the Communication Abstraction Framework(Section 4) and the Configuration Framework (Section 5). Section 6 presents Spartacus as a study case of how MARIE can be used in a robotic implementation. Conclusions and future work are presented in Section 7.

2 Software Architecture

MARIE's software architecture can be explained from the following three perspectives: component mediation approach, layered architecture and communication protocol abstraction.

2.1 Component Mediation Approach

To implement distributed applications using heterogeneous softwares, MARIE adapted the Mediator Design Pattern [GHJV94] to create a Mediator Interoperability Layer (MIL), as illustrated in Figure 1. The Mediator Design Pattern primarily creates a centralized control unit (named Mediator) which interacts with each class independently, and coordinates global interactions between classes to realize the desired system. In MARIE, the MIL acts just like the Mediator of the original pattern, but is implemented as a virtual communication space where applications can interact together using a common language (similar to Internet's HTML for example). With this approach, each application can have its own communication protocols and mechanisms as long as the MIL supports it. It is a way to exploit the diversity of communication protocols and mechanisms, to benefit from their strengths and maximize their usage, and to overcome the absence of standards in robotic software systems. It also promotes loose coupling between applications by replacing a many-to-many interaction model with a one-to-many interaction model. In addition to simplifying each application communication interface, loose coupling between applications increases reusability, interoperability and extensibility by limiting their mutual dependencies and hiding their internal implementation. By using a virtual communication space approach, the MIL's design reduces the potential complexity of managing a large number of centralized classes, as observed with the original pattern. This is mainly attributed to having limited centralization of communication protocols and mechanisms, leaving most of the functionalities decentralized. Although there is no such thing as an instance of a Mediator in MARIE's implementation, mediation is possible through the Communication Abstraction Framework.

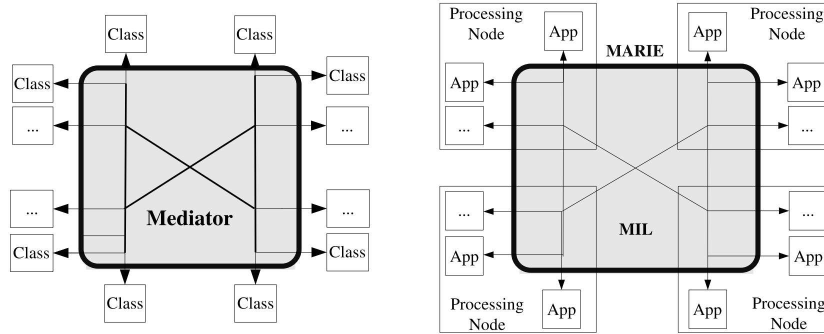

Fig. 1. Original Mediator Pattern (left) and MARIE's Distributed Mediator Adaptation (right)

2.2 Layered Architecture

Supporting multiple sets of concepts and abstractions can be achieved in different ways. MARIE does so by adopting a layered software architecture, defining different levels of abstraction into the global middleware framework. As shown in Figure 2, three abstraction layers are used to reduce the amount of knowledge, expertise and time required to use the overall system. It is up to the developer to select the most appropriate layer for adding elements to the system. At the lower level, the Core Layer consists of tools for communication, data handling, distributed computing and low-level operating system functions (e.g., memory, threads and processes, I/O control). The Component Layer specifies and implements the Component Framework, the Communication Abstraction Framework and the Configuration Framework useful to build new applications using the MIL. The Application Layer contains useful tools to build and manage integrated applications to craft robotic systems.

2.3 Communication Protocol Abstraction

Integrated applications functionalities can often be used without any concerns with the communication protocols, as they are typically designed to apply operations and algorithms on data, independently of how data are received or sent. This eases applications interoperability and reusability by avoiding fixing the communication protocol during the design phase. Ideally, the communication procol choice should be made as late as possible, depending of which applications need to be interconnected together (e.g., at the integration phase or even at runtime). Therefore, a Communication Abstraction Framework, called Port, is provided for communication protocols and applications interconnections.

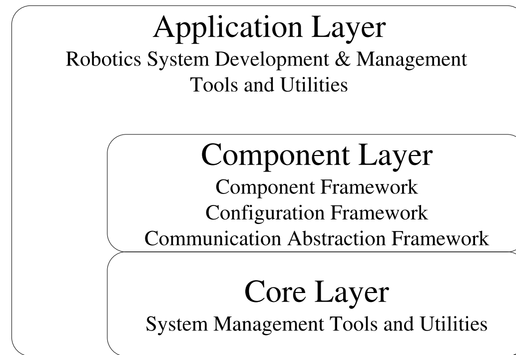

Fig. 2. MARIE's Layered Architecture

3 Component Framework

The development of robotic applications using MARIE is based on reusable software blocks, referred to as components, which implement functionalities by encapsulating existing applications, programming environments or dedicated algorithms. Components are configured and interconnected to implement the desired system, using the software applications and tools available through MARIE. Four types of components are used in the MIL:

1. **Application Adapter (AA)** : component interfacing useful applications within the MIL and to enable them to interact with each other through their standardized interface (i.e. Ports) and using MARIE's shared data types. Interconnections using Port communication abstraction are illustrated in Figure 3 with a small dot between communication links represented by arrows.
2. **Communication Adapter (CA)** : component that ensures communication between other components by adapting incompatible communication mechanisms and protocols, or by implementing traditional routing communication functions. Available Communication Adapters in MARIE are Splitters, Switches, Mailboxes and Shared Maps. A Splitter sends data from one source to multiple destinations without the sender needing to be aware of the receivers. A Switch acts like a multiplexer sending data to the selected output. A Mailbox creates a buffering interface between asynchronous components. A Shared Map is used to share data, in the key-value form, between multiple components.
3. **Application Manager (AM)** : system level component managing, on local or remote processing nodes, Application Adapters and Communication Adapters. Application and Communication Adapters initialization, configuration, start, stop, suspend and resume are handled by the Ap-

plication Manager. When starting the system, the Application Manager initializes the components following the adequate sequence.

4. **Communication Manager (CM)** : system level component dynamically managing, on local or remote processing nodes, the communications mechanisms (socket, port, shared memory, etc.).

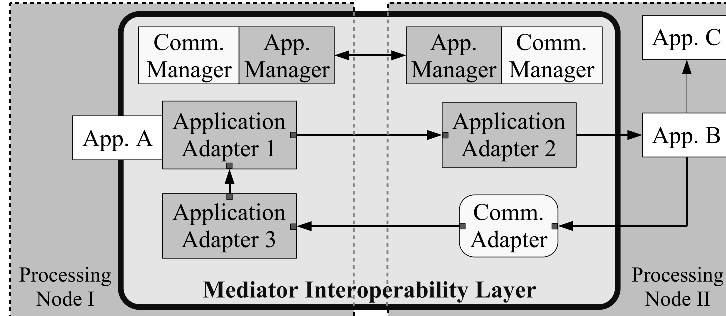

Fig. 3. Component Framework Using the Mediator Interoperability Layer

Although the use of MARIE's frameworks and software tools is highly encouraged to save time and efforts, MARIE is not limited to them. Developers can use the best solution to integrate software applications and interconnect components by having the possibility to extend or adapt existing components and available frameworks. MARIE's underlying philosophy is to complement existing applications, programming environments or software tools, and therefore it is to be used only when required and appropriate.

Figure 3 presents an example of how software applications can be integrated and interconnected in the MIL. Application A represents an integrated application directly linked with the implementation of its AA (e.g., a library or an open source application). When an application is integrated using an AA, it can use the MIL communication mechanisms to exchange data with any other components, as is the case for providing data to Application Adapter 2 and Application Adapter 3. Application B interacts with other applications in two different ways. The first one needs Application Adapter 1 to transmit data to Application Adapter 2, which convert them into a specific communication protocol not supported by the MIL to make them available to Application B. The second one is used to send back data to Application Adapter 3, using a communication protocol supported by the MIL for direct interconnection with any components. However, Application B and Application Adapter 3 do not use compatible communication mechanisms or protocols. Interfacing them requires a CA. Application Adapter 3 implements functionalities directly in the MIL by encapsulating them in a stand-alone AA (e.g., a graphical user interface implemented in the AA directly). Application C can already com-

municate with Application B, and therefore no interconnection through the MIL is required.

3.1 Main Concepts Supporting the Component Framework

The Component Framework is a whitebox framework enabling developers to extend available functionalities or create new components. Existing functionalities can be reused and extended by inheriting from framework base classes and/or overriding pre-defined hook methods. Five main concepts present in every component are provided by the Component Framework and illustrated in Figure 4 :

- **Handler** : handles the behavior of the component. Every Handler must support the init, start, stop, suspend, resume, reset and quit messages that are received through their Request Interface. Execution flow of the Handler can be customized to adapt its own implementation needs (based on iterations, messages, interrupts, states-machine, etc).
- **VisitorConfig** : holds configuration information for the component. It is extracted from the configuration data structure created by the Configurator. The configuration information will then be used by the Handler in the initialization phase of the component.
- **Director** : handles and manages execution of requests (such as init, start, stop, etc.) and forwards them to the component's Handler to execute specialized functionalities if needed.
- **Configurator** : handles configuration requests, parsing of the configuration file format, and creation of the Configuration Data Structures to be forwarded to the component's Handler.
- **Builder** : builds the component using the specialization of the different elements composing a component : the Director, the Configurator, the Handler, and their respective execution flow mechanisms.

To explain what is required to create a component, Figure 5 illustrates the Splitter Communication Adapter. The Splitter is typically used to route data from one or more source Ports to multiple destination Ports. This mechanism is handled by the Splitter Handler. The Splitter Handler uses an Iterate Execution Mechanism to monitor the Splitter Handler behavior. The Iterate Execution Mechanism refers to an internal loop that runs at a determined cycle inside the Handler. Sending and receiving data from Ports is event-based and is triggered by the main communication loop sending the data as soon as it arrives (not shown in Figure 5). As shown in Figure 5, Ports on the left (A0 to An) and Ports on the right (B0 to Bn) are grouped together to form Group A and Group B. The Splitter can support any number of ports in each group. The configuration of the Splitter supports three modes, which selects how the data flows between Group A and Group B:

1. **Mode AB**. Communicates one way from Group A to Group B.

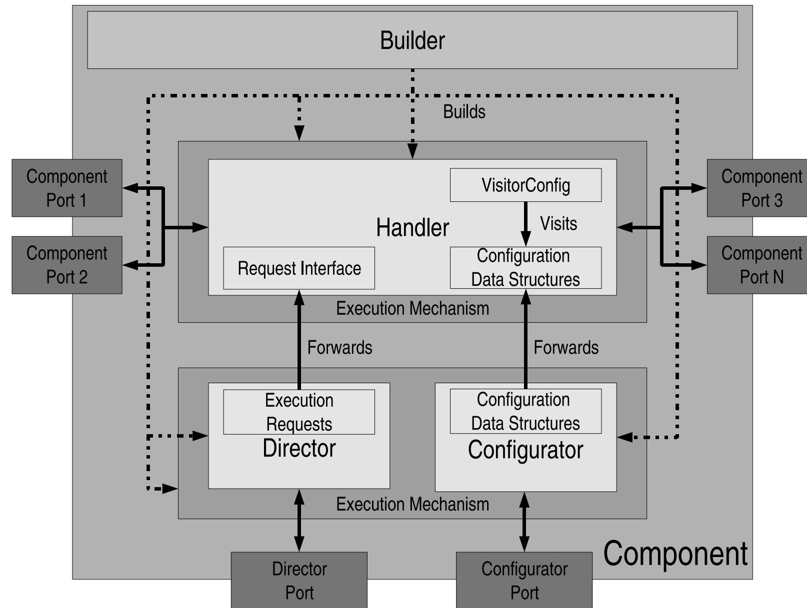

Fig. 4. MARIE's Components Composition

2. **Mode BA**. Communicates one way from Group B to Group A.
3. **Mode ABBA**. Communicates in both ways (bi-directional) between Group A and Group B.

To configure the Splitter Communication Adapter, the XML Configurator is used. It parses the XML configuration files and forwards the Configuration Data Structure to the Splitter Handler. The Splitter VisitorConfig then uses the Configuration Data Structure to get the appropriate configuration at initialization. The Default Director forwards execution requests (start, stop, etc.) to the Splitter Handler.

4 Communication Abstraction Framework

The Communication Abstraction Framework, illustrated in Figure 6, offers an abstraction on how communications are achieved by components, using a Send & Receive Interface to send and receive data. This interface hides implementation details of communication protocols and mechanisms that execute send and receive requests. Those communication protocols and mechanisms are encapsulated in Communication Strategy (CS) classes in order to be more easily reused and extended. The Strategy Design Pattern [GHJV94] is applied to CS to define a set of interchangeable algorithms and to create a loosely coupled relation between CS's clients and implementation details of communication

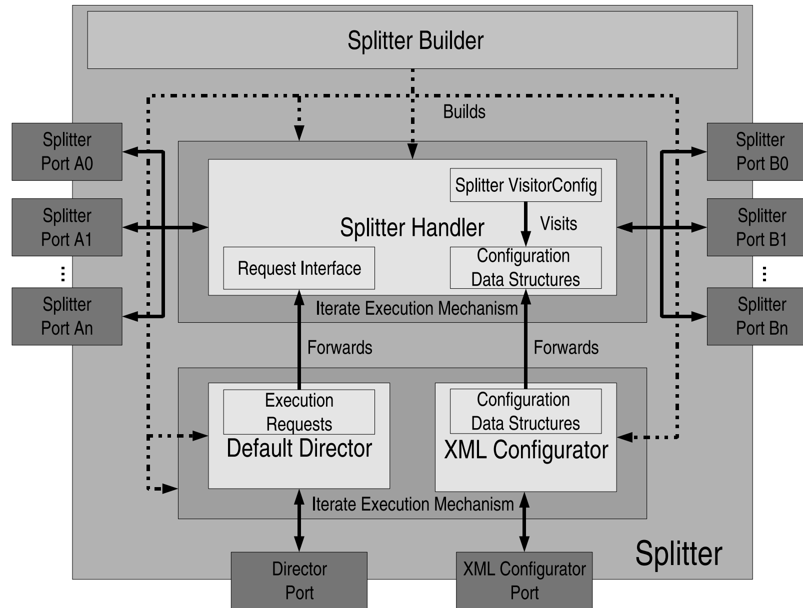

Fig. 5. MARIE's Splitter Composition

protocols and mechanisms. Currently supported Communication Strategies are :

- **SocketAcceptor**. TCP/IP socket-based server accepting one connexion on a specified port.
- **SocketConnector**. TCP/IP socket-based client connecting to the server on a specified port.
- **SharedMemAcceptor**. Memory-based server accepting one local (on the same processing node) connexion.
- **SharedMemConnector**. Memory-based client connecting to the local server.

The Send & Receive interfaces also hides data operations that needs to be applied in order to fulfill communication protocols requirements on data representation. Those operations are encapsulated in Cascading Functional Block (CFB) classes, that can be chained together in a cascaded manner to execute the appropriate sequence of operations on sent and received data. MARIE currently provides four kinds of CFBs :

- **XMLFormatter** : marshalls MARIE's objects (data structures in memory) in an homemade XML representation to be send to a byte stream CS.
- **XMLExtractor** : unmarshalls the XML data representation of MARIE's data objects from the byte stream CS to create new MARIE data objects.

- **ImageFormatter** : marshalls MARIE's Image objects in an optimized serialized data representation to be send to a byte stream CS.
- **ImageExtractor** : unmarshalls the image optimized serialized data representation from the byte stream CS to create a new MARIE Image data.

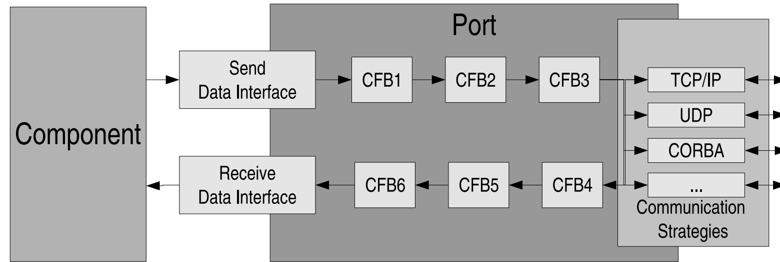

Fig. 6. Communication Abstraction Through the Port Interface

The Communication Abstraction Framework can be extended or specialized knowing that Ports are not tightly coupled to the Component Framework. This means that the Communication Abstraction Framework could be redesigned to handle specialized functionalities (error handling, signals, etc.) without having a major impact on existing implementations. However, it is is highly recommended to use the Send & Receive Interface for standardization of communications with components.

5 Configuration Framework

MARIE's Configuration Framework, shown in Figure 7, offers a generalization of configuration representation in order to use the same data structure for all the components configurations. Four types of configuration elements are available in the Configuration Framework representing the Configuration Data Structures :

1. **Configuration**. Composite configuration element that gives a name to a current configuration structure and contains other configuration elements.
2. **Type**. Composite configuration element that represents a type contained in the component's description domain. A type element can be composed of multiple configuration elements.
3. **Key-Value**. Primitive configuration element that identifies a specific property of an object. The configuration element is represented by a label (Key) and have a value (Value).
4. **Qualifier**. Composite configuration element that represents an attribute for refining configuration element semantics. It can be useful to categorize

or discriminate configuration elements from each other. A qualifier element can be applied on any other configuration element.

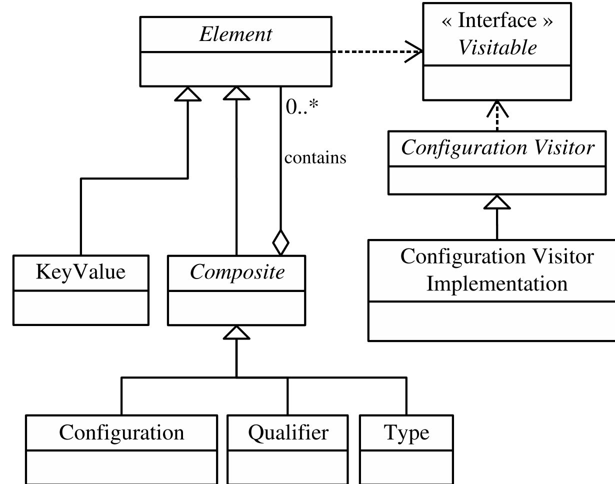

Fig. 7. Configuration Framework Architecture

To support execution of Configuration Data Structures' operations (read, write, modify, etc.), the Visitor and Composite Design Pattern [GHJV94] are used. The of the Visitor Design Pattern is to encapsulate operations to be performed on a data structure in a class, called a visitor, that can traverse the data structure. Having different kinds of configuration elements in the Configuration Framework (primitive and composite), applying the Composite Design Pattern to the Configuration Data Structures' elements permits to the visitor to treat each kind of elements as they were exactly the same, which reduces visitor's implementation complexity.

Generally, the Configuration Framework is used as a blackbox framework by creating required visitors to fetch data configuration in Configuration Data Structures, by extending the Visitor abstract class. Other configuration manipulations, such as parsing configuration files and creating the Configuration Data Structures, are handled automatically. In the current implementation, an XML based generic parser (not shown in Figure 7) is responsible of creating the Configuration Data Structures by parsing the XML configuration file. If required, new parsers supporting other representations and languages can be added in the framework without the need to change existing visitor classes.

The following example shows a sample Splitter configuration file. All four configuration elements are present in this Splitter sample configuration file. Each XML node contains an attribute named "elem" which can be set to

four values : *conf* for Configuration element, *type* for Type element, *kv* for Key-Value element and *q* for Qualifier element. This configuration represents a Splitter with one Port in its Group A and two Ports in its group B. Group A and B are represented as qualifiers in the configuration file, providing in which group each Port must be instantiated. Data flows from Group A to Group B as the "mode" Key-Value states. Port A0 is configured to use the SocketAcceptor communication strategy listening on port 30004. Port B0 is configured to also use the SocketAcceptor communication strategy, listening on port 30000. Port B1 is configured to use the Socket Connector communication strategy, connecting to host 192.168.43.67 on port 30030.

```
<?xml version="1.0"?>
<splitter elem="conf">
   <mode elem="kv">AB</mode>
   <groupA elem="q">
      <port elem="type">
         <type elem="kv">Default</type>
         <name elem="kv">A0</name>
         <cs elem="type">
            <type elem="kv">SocketAcceptor</type>
            <portnumber elem="kv">30004</portnumber>
         </cs>
      </port>
   </groupA>
   <groupB elem="q">
      <port elem="type">
         <type elem="kv">Default</type>
         <name elem="kv">B0</name>
         <cs elem="type">
            <type elem="kv">SocketAcceptor</type>
            <portnumber elem="kv">30000</portnumber>
         </cs>
      </port>
      <port elem="type">
         <type elem="kv">Default</type>
         <name elem="kv">B1</name>
         <cs elem="type">
            <type elem="kv">SocketConnector</type>
            <portnumber elem="kv">30030</portnumber>
            <hostname>192.168.43.67</hostname>
         </cs>
      </port>
   </groupB>
</splitter>
```

6 Spartacus' Implementation Using MARIE

Spartacus [MBC+05], shown in Figure 8, is a socially interactive mobile robot designed to enter the AAAI Mobile Robot Challenge. Introduced in 1999, the AAAI Challenge consists of having a robot start at the entrance of the conference site, find the registration desk, register, perform volunteer duties (e.g., guard an area) and give a presentation [MSJ+04]. The long-term objective is to have robots participate just like humans attending the conference. We became interested by this challenge because of the need to address all design dimensions for such a robot, from the hardware level to the high-level decision-making algorithms.

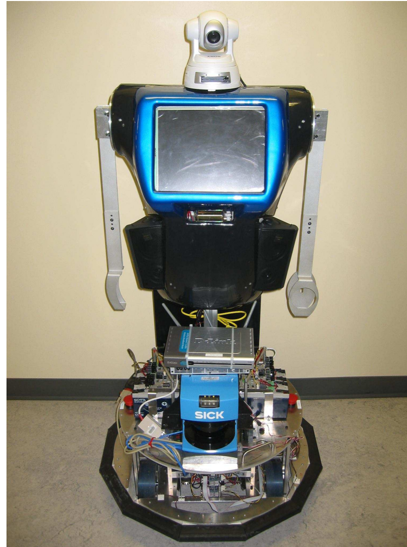

Fig. 8. Spartacus Robot

Spartacus is equipped with a SICK LMS200 laser range finder (for autonomous navigation), a Sony SNC-RZ30N 25X pan-tilt-zoom color camera, an array of eight microphones placed on the robot's body, a touchscreen and a business card dispenser. High-level processing is carried out using an embedded Mini-ITX computer (Pentium M 1.7 GHz). The Mini-ITX computer is connected to the low-level controllers through a CAN bus device, the laser range finder through a serial port, the camera through a 100Mbps Ethernet link and the audio amplifier and speakers using the audio output port. A laptop computer (Pentium M 1.6 GHz) is also installed on the platform and is equipped with a RME Hammerfal DSP Multiface sound card using eight analog inputs to simultaneously sample signals coming from the microphone array. Communication between the two on-board computers is accomplished with a 100Mbps Ethernet link. Communication with external computers can be achieved using the 802.11g wireless technology, giving the ability to easily add remote processing power or capabilities if required. All computers are running Debian GNU Linux.

Numerous algorithms are required to accomplish the Challenge, and here is what we implemented for our 2005 participation to the event:

- **Autonomous Navigation**. When placed at the entrance of the convention center, the robot autonomously find its way to the registration desk by wandering and avoiding obstacles, searching for information regarding the location of the registration desk and potentially following people moving in this direction. Once registered, the robot can use a map of

the convention center. The two navigation tools used are CARMEN and pmap. CARMEN, the Carnegie Mellon navigation toolkit [MRT03], is a software package for laser-based autonomous navigation using a map previously generated. The pmap package[1] provides a number of libraries and utilities for laser-based mapping (SLAM) in 2D environments to produce high-quality occupancy grid maps.

- **Vision Processing**. Extracting useful information in real time from images taken by the onboard camera improves interaction with people and the environment. For instance, the robot could benefit from reading various written messages in real life settings, messages that can provide localization information (e.g., room numbers, places) or identity information (e.g., reading name badges). Object recognition and tracking algorithms also makes it possible for the robot to interact with people in the environment. We use two algorithms to implement such capabilities : one that can extract symbols and text from a single color image in real world conditions [LMV04]; and another one for object recognition and tracking to identity and follow regions of interest in the image such as human faces and silhouettes.
- **Sound Processing**. Localization of sound sources provides important clues about the world. However, simply using one or two omnidirectional microphones on a robot is not enough: it proves too difficult to filter out all of the noise generated in public places. Using a microphone array is a better solution for the localization, tracking and separation of sound sources. Our approach is capable of simultaneously localizing and tracking up to four sound sources that are in motion over a 7 meters range, in the presence of noise and reverberation [VMHR1]. We also developed a method to separate in real-time the sound sources [VRM04] in order to simultaneously process vocal messages from interlocutors using software packages such as Nuance[2].
- **Touchscreen Display**. Various information can be communicated through this device, such as: receiving information from people using a menu interface; displaying graphical information such as a PowerPoint presentation or a map of the environment; and expressing emotional states using a virtual face.

MARIE's design and implementation evolved as we worked on Spartacus' implementation. MARIE's current version is in C++ (˜10 000 lines of code) and uses the ACE (Adaptive Communication Environment) library [Sch94] for the Core Layer functions (low-level operating system functions). Although ACE met Spartacus' implementation needs, MARIE does not rely on this specific library as the Core Layer and it can be replaced if required. MARIE's Application Manager is partially implemented in MARIE, meaning that AA

[1] http://robotics.usc.edu/˜ahoward/pmap
[2] http://www.nuance.com/

and CA must be initialized manually from scripting commands. Also, the CM is not yet implemented, and component configuration must be set manually.

Spartacus' implementation requires 45 components (˜50 000 lines of code) composed of 26 AA, 17 CA and two external applications (the Audio Server and NUANCE). Application Adapters are used to interface the different software applications required for decision-making by the robot. Mailboxes, Splitters, Shared Maps and Switches are used as Communication Adapters. Except for the two external applications, component interconnections are all sockets-based using Push, Pull and Events dataflow communication mechanisms [Zha03] with XML encoding for data representation; the Audio Server and Nuance use their own communication protocols.

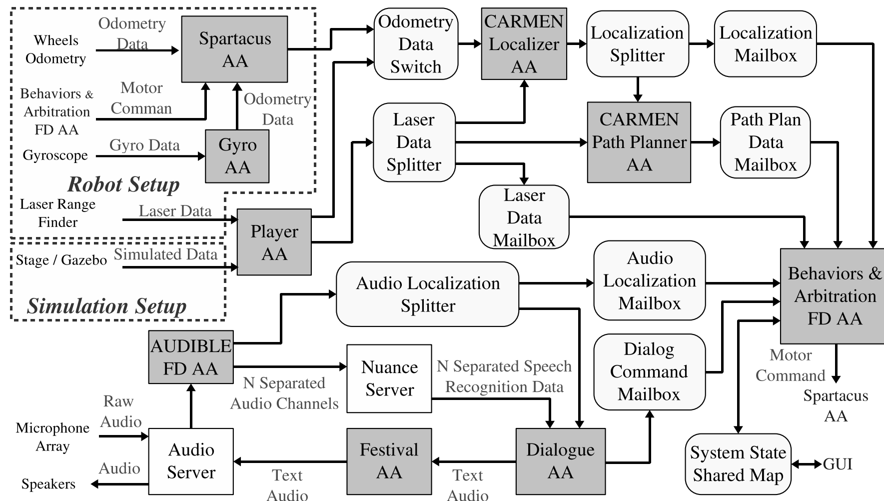

Fig. 9. Partial Representation of Spartacus' Software Architecture

Partial representation of Spartacus' software architecture is illustrated in Figure 9. It covers sensing and acting in simulation and real robot setups, localization, path planning, sound source localization, tracking and separation, speech recognition and generation, and part of the computational architecture [MBC+05] responsible for the robot's navigation, reasoning and interactions capabilities. In the real robot setup, SpartacusAA combines wheels odometry and gyroscopic (through GyroAA interfacing a gyroscope installed on Spartacus) data, and pushes the result at a fixed rate (10 Hz) to its interconnected component. Laser data is collected by PlayerAA, interfacing the Player library specialized for sensor and actuator abstraction [VGH03], supporting the SICK LMS200 laser range finder installed on Spartacus. PlayerAA pushes data at a fixed rate (10 Hz) to connected components. In the simulation setup, odometry and laser data are both collected with PlayerAA, as generated using Stage

(2D) or Gazebo (3D) simulators [VGH03]. CARMEN Localizer AA and CARMEN Path Planner AA provide path planning and localization capabilities.

RobotFlow and FlowDesigner programs [CLM+04] are used to implement Behavior & Arbitration FD AA, handling part of the computational architecture. RobotFlow (RF) [CLM+04] and FlowDesigner (FD) are two modular data-flow programming environments that facilitate visualization and understanding of the robots control loops, sensor and actuator processing. They are also appropriate for rapid prototyping since the graphical user interface enables the user to connect reusable software blocks without having to compile the program every time minor changes are made. In the Behavior & Arbitration FD AA component, RF/FD programs implement behavior-producing modules arbitrated using a priority-based approach. It uses data coming from different elements such as localization, path plan, laser, audio localization, dialog command and system states. It uses an asynchronous pull mechanism to get its data, requiring the use of Mailbox CA components, and generates motor commands at a fixed rate (5 Hz).

The Audio Server is interfacing the RME Hammerfal DSP Multiface sound card, and Nuance Server is interfacing Nuance. DialogueAA is a stand-alone AA that manages simultaneous conversations with people. This is made possible with the use of AUDIBLE FD AA, interfacing our sound source localization, tracking and separation algorithms implemented with RF/FD and using Spartacus' microphone array. It generates a number of separated audio channels that are sent to Nuance Server and Behavior & Arbitration FD AA. Recognized speech data is sent to Dialog AA, responsible of the human-robot vocal interface. Speech generated by the robot is handled by Festival [Tay99]. Dialogue AA also provides data to the Behaviors & Arbitration FD AA. The global execution of the system is asynchronous, having most of the applications and AAs pushing their results at a variable rate, based on the computation length of their algorithms when triggered by new input data. Synchronous execution is realized by having fixed rate sensors readings and actuators commands writings.

6.1 Discussion

Using MARIE with Spartacus provided interesting capabilities for software integration and team development. At the peak of Spartacus' software development process, eight software developers, including audio and image processing specialists, AI specialists, robot hardware specialists, Core Layer specialists and the integrator, were working concurrently on the system. Most of them only used Application Adapters (Component Layer) to create their components, conducting unit and blackbox testing with pre-configured system setups (Application Layer) given by the integrator. Communication protocols and operating system tools for component and application developments (CoreLayer) were added by the Core Layer specialists when required. Components were incrementally added to the system as they became available. It took around

eight days, spread over a four weeks period, to complete a fully integrated system.

Overall, nine existing specialized applications/libraries were integrated together to build the complete system: Player/Stage/Gazebo, Pmap, CARMEN, Flowdesigner/RobotFlow, AUDIBLE, Nuance, Festival, AUDIBLE, QT3 and OpenCV. Each of these applications required different integration strategies. For instance, Nuance is a proprietary application with a specific and limited interface. Integrating Nuance in an AA was challenging because its execution flow is tightly controlled by Nuance's core application, which is not accessible from the available interface. To solve this problem, we created an independent application that uses a communication protocol already supported by the MIL. CARMEN, on the other hand, is composed of small executables communicating through a central server. CARMEN's integration was realized by creating an AA that starts several of these executables depending on the required functionality and on data conversion from CARMEN's to MIL's format. Having a flexible Component Framework and Ports as the communication protocol abstraction allowed us to adapt application specificities such as external threads execution, dynamic bindings, independent protocols and timing.

Choosing XML data representation for common language communication in the MIL was based on implementation simplicity and ease of debugging. Although it was sufficient for most of the system communication needs, we clearly observed that this solution was not sufficient to support communication-intensive data like audio and vision within MARIE. To avoid using valuable time to support optimized protocols for audio and video, we decided to use FlowDesigner that already provides those protocols.

With regard to component interoperability, the ability to change between simulation and robotic setups with only few system modifications gave us the possibility to do quick simulations and integration tests. Nearly 75% of the system functionalities were validated in simulation and were also used as is in the real world setup. In both simulated and real setups, configurations of components receiving laser and odometry data are exactly the same, abstracting data sources and benefiting from components modularity and the rapid prototyping approach. Moreover, component interoperability can be extended with MARIE to do things like porting a computational architecture on robotic platforms from different manufacturers and with heterogeneous capabilities, or evaluating performances of algorithms implementing the same functionality (e.g., localization, navigation, planning) using the same platform and experimental settings.

Distributing applications across multiple processing nodes was not difficult with MARIE, having chosen network sockets as the transport mechanism. We initially used a shared memory transport mechanism to accelerate communication between components on the same computer. Changing from one transport mechanism to another was transparent using Ports and supporting shared memory interconnection in the MIL. Since no noticeable impact

was observed over the global system performances using either of them, we chose to exclusively use socket transport mechanism. It allows us to move components from one processing node to the other easily.

Meeting Spartacus' integration needs using MARIE rapid-prototyping approach highlighted three interesting consequences on robotics system development. First, it revealed the difficulty of tracking decisions made by the system simply by observing its behaviors in the environment, something that was always possible with simpler implementations. The system reached a level of complexity where we needed to develop a graphical application to follow on-line or study off-line the decisions made by the robot. This suggests that creating analysis tools and supporting them in the integration environment can play a key role in working with such a highly-integrated system. The second observation emerged from the number of components involved in the software architecture. Manually configuring and managing the system with many components executed on multiple processors, is an error-prone and tedious task. In this context, MARIE would greatly benefit from having GUI and system management tools to build, configure and manage components automatically. Third, regarding design optimization, being able to quickly interconnect components to create a complete implementation, without focusing on optimization right away, proved beneficial in identifying real optimization needs. Such an exploration strategy gave us the ability to quickly reject necessary applications, software designs or component implementations without investing too much time and effort. For Spartacus, we originally thought that tighter synchronization between components would be necessary to obtain a stable system and support real-time decision-making. For instance, having connected all of Spartacus' components together, we observed that performances were appropriate with the processing power available as long as we did not overload the computers with too many components. Noticing that, we decided to wait before investing time and energy working on component synchronization, to focus on Spartacus' integration challenges.

7 Conclusion

MARIE is a system integration framework oriented towards a rapid-prototyping approach to create robotic systems. To achieve this goal, MARIE is based on the mediation principle and uses a layered framework architecture to facilitate the creation, integration and interconnection of existing applications, programming environments or software tools available in the robotics community. Interconnections of applications are supported by a communication framework that is able to support a wide range of communication protocols, communication mechanisms, and upcoming robotics standards. To ease efforts required to integrate the work of multiple developers, MARIE also supports team development requirements with two design choices : 1) the layered architecture allows each developer to work at the appropriate level of abstraction,

related to his contribution to the system, and 2) the component architecture lets developers work independently on each component. This tends to reduce the required knowledge to contribute to the system and accelerates the overall development cycle.

MARIE was experimented during Spartacus' software architecture development, which is the first software architecture implementation using MARIE. From this experience, we have observed that an integrated programming environment such as MARIE helps us focus on the decision-making issues and the high-level capabilities development rather than on low-level software programming considerations and integration issues. MARIE's integration framework was flexible enough to support the integration and interconnection of all the existing and new applications required for Spartacus' software architecture. Using a rapid-prototyping approach is well suited to rapidly identify critical development sections from less-critical ones, just by being able to work with the complete system at the very beginning of the development cycle.

More testing will be performed on Spartacus to validate MARIE's architectural design and implementation. We are currently working on identifying and implementing tools to measure system real-time performances and on software metrics to quantify MARIE's computational overhead. Additional work is also planned on the Application Layer, in which we hope to develop further useful applications and automated tools to manage the overall system and the underlying components.

8 Acknowledgements

F. Michaud holds the Canada Research Chair (CRC) in Mobile Robotics and Autonomous Intelligent Systems. Support for this work is provided by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chair program. MARIE is an open-source project available at http://marie.sourceforge.net.

References

[AS04] V. Andronache and M. Scheutz, *Ade - a tool for the development of distributed architectures for virtual and robotic agents*, IEEE Transactions on Systems, Man and Cybernetics, Part B, 2004, pp. 2377– 2395.

[BMA97] D. Brugali, G. Menga, and A. Aarsten, *The framework life span*, Communication of the ACM **40** (1997), no. 10, 65–68.

[CBL+06] C. Cote, Y. Brosseau, D. Letourneau, C¿ Raievsky, and F. Michaud, *Using marie in software development and integration for autonomous mobile robotics*, International Journal of Advanced Robotic Systems, Special Issue on Software Development and Integration in Robotics (2006).

[CLM+04] C. Cote, D. Letourneau, F. Michaud, J.M. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran, *Code reusability tools for programming mobile robots*, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.

[CS95] J.O. Coplien and D.C. Schmidt, *Pattern languages of program design*, ch. Frameworks and Components, pp. 1–5, Addison-Wesley, 1995.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns : Elements of reusable object-oriented software*, Addison-Wesley, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Villisides, *Design patterns: Elements of reusable object oriented software*, Addison-Wesley, 1995.

[GMP+01] V. Gazi, M.L. Moore, K.M. Passino, W.P. Shackleford, F.M. Proctor, and J. Albus, *The rcs handbook: Tools for real-time control systems software development*, ch. 1, p. 1, Wiley, 2001.

[JF88] R.E. Johnson and B. Foote, *Designing reusable classes*, Journal of Object-Oriented Programming (1988).

[LMV04] D. Letourneau, F. Michaud, and J.-M. Valin, *Autonomous robot that can read*, EURASIP Journal on Applied Signal Processing, Special Issue on Advances in Intelligent Vision Systems: Methods and Applications **17** (2004), 1–14.

[MBC+05] F. Michaud, Y. Brosseau, C. Côté, D. Létourneau, P. Moisan, A. Ponchon, C. Raievsky, J.-M. Valin, E. Beaudry, and F. Kabanza, *Modularity and integration in the design of a socially interactive robot*, Proceedings IEEE International Workshop on Robot and Human Interactive Communication, 2005, pp. 172–177.

[MFB+02] Hafedh Mili, Mohamed Fayad, Davide Brugali, David Hamu, and Dov Dori, *Enterprise frameworks: issues and research directions*, Software Practice and Experience **32** (2002), 801–831.

[MRT03] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun, *Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit*, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (Las Vegas, NV), vol. 3, October 2003, pp. 2436–2441.

[MSJ+04] B. Maxwell, W. Smart, A. Jacoff, J. Casper, B. Weiss, J. Scholtz, H. Yanco, M. Micire, A. Stroupe, D. Stormont, and T. Lauwers, *2003 aaai robot competition and exhibition*, AI Magazine **25** (2004), no. 2, 68–80.

[NWB+03] I. A. D. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, *Claraty and challenges of developing interoperable robotic software*, Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003, pp. 2428–2435.

[OC03] Anders Orebäck and Henrik I. Christensen, *Evaluation of architectures for mobile robotics.*, Autonomous Robots **14** (2003), no. 1, 33–49.

[Sch94] D.C. Schmidt, *Ace: an object-oriented framework for developing distributed applications*, Proceedings of the 6th USENIX C++ Technical

[Sch95] H.A. Schmid, *Creating the architecture of a manufacturing framework by design patterns.*, Proceedings of OOPSLA'95 (1995).

[Sch03] C. Schlegel, *A component approach for robotics software: Communication patterns in the orocos context*, 18 Fachtagung Autonome Mobile Systeme (AMS), 2003, pp. 253–263.

[Tay99] P. Taylor, *The festival speech architecture*, URL: http://www.cstr.ed.ac.uk/projects/festival/, 1999.

[USEK02] Hans Utz, Stefan Sablatnöog, Stefan Enderle, and Gerhard K. Kraetzschmar, *Miro – middleware for mobile robot applications*, IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures **18** (2002), no. 4, 493–497.

[VGH03] R. T Vaughan, B. P. Gerkey, and A. Howard, *On device abstractions for portable, reusable robot code*, Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003, pp. 2421–2427.

[VMHR1] J.-M. Valin, F. Michaud, B. Hadjou, and J. Rouat, *Localization of simultaneous moving sound sources for mobile robot using a frequency-domaine steered beamformer approach*, Proceedings IEEE International Conference on Robotics and Automation, 1, pp. 1033–1038.

[VRM04] J.-M. Valin, J. Rouat, and F. Michaud, *Enhanced robot audition based on microphone array source separation with post-filter*, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.

[WMT03] E. Woo, B. A. MacDonald, and F. Trépanier, *Distributed mobile robot application infrastructure*, Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003, pp. 1475–1480.

[Zha03] Y. Zhao, *A model of computation with push and pull processing*, Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, December 2003.