27

An Approach to Distributed Component-Based Software for Robotics*

A. C. Domínguez-Brito, J. Cabrera-Gámez,
J. D. Hernández-Sosa, J. Isern-González and E. Fernández-Perdomo
Instituto Universitario SIANI & the Departamento de Informática y Sistemas,
Universidad de Las Palmas de Gran Canaria
Spain

1. Introduction

Programming robotic systems is not an easy task, even developing software for simple systems may be difficult, or at least cumbersome and error prone. Those systems are usually multi-threaded and multi-process, so synchronization problems associated with processes and threads must be faced. In addition distributed systems in network environments are also very common, and coordination between processes and threads in different machines increases programming complexity, specially if network environments are not completely reliable like a wireless network. Hardware abstraction is another question to take into account, uncommon hardware for an ordinary computer user is found in robotics systems, sensors and actuators with APIs (Applications Programming Interfaces) in occasions not very stable from version to version, and many times not well supported on the most common operating systems. Besides, it is not rare that even sensors or actuators with the same functionality (i.e. range sensors, cameras, etc.) are endowed with APIs with very different semantics. Moreover, many robotic systems must operate in hard real time conditions in order to warrant system and operation integrity, so it is necessary that the software behaves obeying strictly specific response times, deadlines and high frequencies of operation. Software integration and portability are also important problems in those systems, since many times only in one of them we may find a variety of machines, operating systems, drivers and libraries which we have to cope to. Last but not least, we want robotic systems to behave "autonomous" and "intelligently", and to carry out complex tasks like mapping a building, navigating safely in a cluttered, dynamic and crowded environment or driving a car safely in a motorway, to name a few.

Despite there is no established standard methodology or solution to the situation described in the previous paragraph, in the last ten years many approaches have blossomed in the robotics community in order to tackle with it. In fact, many software engineering techniques and experiences coming from other areas of computer science are being applied to the specific area of robotic control software. A review of the state of the art of software engineering applied

*This work has been partially supported by the Research Project *TIN2008-06068* funded by the Ministerio de Ciencia y Educación, Gobierno de España, Spain, and by the Research Project *ProID20100062* funded by the Agencia Canaria de Investigación, Innovación y Sociedad de la Información (ACIISI), Gobierno de Canarias, Spain, and by the European Union's FEDER funds.

specifically to robotics can be found in [Brugali (2007)]. Many of the approaches that have come up in these last years, albeit different, are either based completely, or follow or share to a certain extend some of the fundamentals ideas of the CBSE (Component Based Software Engineering) [George T. Heineman & William T. Councill (2001)] paradigm as a principle of design for robotic software.
Some of the significant approaches freely available within the robotics community based on the CBSE paradigm are G^{en}oM/BIP [Mallet et al. (2010)][Basu et al. (2006)][Bensalem et al. (2009)], Smartsoft [Schlegel et al. (2009)], OROCOS [*The Orocos Project* (2011)], project ORCA [Brooks et al. (2005)], OpenRTM-aist [Ando et al. (2008)] and Willow Garage's ROS project [*ROS: Robot Operating System* (2011)]. All these approaches, in general incompatible among them, use many similar abstractions in order to build robotic software out of a set of software components. Each of these approaches usually solve or deal with many of the mentioned problems faced when programming robotic systems (hard real-time operation, distributed, multithread and multiprocess programming, hardware abstractions, portability, etc.), and using any of them implies to get used to its own methodology, abstractions and software tools to develop robotic software. Our group have also developed an approach to tackle with the problem of programming robotic systems. It is some years already that we have been using a CBSE C++ distributed framework designed and developed in our laboratory, termed CoolBOT [Antonio C. Domínguez-Brito et al. (2007)], which is also aimed at easing software development for robotic systems. Along several years of use acquiring experience programming mobile robotic systems, we have ended up integrating in CoolBOT some new developments in order to improve their use and operation. Those new improvements have been focused mainly in two main questions, namely: transparent distributed computation, and "deeper" interface decoupling; in next sections they will be presented more deeply. In order to do so this paper is organized as follows. In section 2 we will introduce briefly an overview about CoolBOT. Next, we will pass to focus on each one of the mentioned topics respectively, in sections 4 and 5. The last section is devoted to presenting the conclusions of this work.

2. CoolBOT. Overview

CoolBOT [Antonio C. Domínguez-Brito et al. (2007)] is a C++ component oriented programming framework aimed to robotics, developed at our laboratory some years ago [Domínguez-Brito et al. (2004)], which is normally used to develop the control software for the mobile robots we have available at our institution. It is a programming framework that follows the CBSE paradigm for software development. The key concept in the CBSE paradigm is the concept of *software component* which is a unit of integration, composition and software reuse [Brugali & Scandurra (2009)][Brugali & Shakhimardanov (2010)]. Complex systems might be composed of several ready-to-use components. Ideally, interconnecting available components out of a repository of components previously developed, we can program a complete system. Thus, it should be only necessary a graphical interface or alike, to set up a system. Hence, being CoolBOT CBSE oriented, it also makes use of this central concept to build software systems.
Fig. 1 gives a global view of a typical system developed using CoolBOT. As we can observe, there are five CoolBOT *components* and two CoolBOT *views*, forming all them four CoolBOT *integrations* involving three different machines sharing a computer network. In addition, hosted by one of the machines, there is a non CoolBOT application which uses a CoolBOT *probe* to interact with one of the components of the system. Thus, in CoolBOT we can find three types of software components: *components*, *views* and *probes*. All these three types

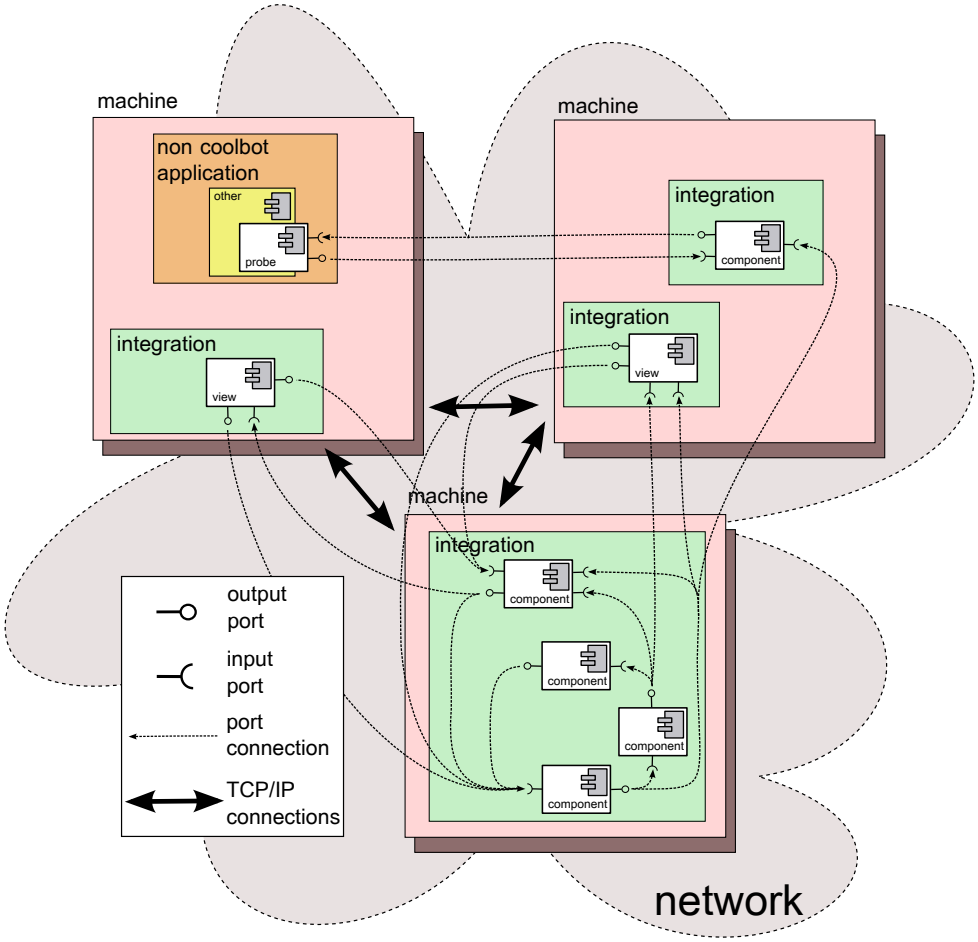

Fig. 1. Diagram of the elements and interconnections of a system designed with CoolBOT.

are software components in the whole sense, since we can compose them indistinctly and arbitrarily without changing their implementation to built up a given system. The main difference among them is that *views* and *probes* are "light-weight" software components in relation to CoolBOT *components*. *Views* are software components which implement graphical control and monitoring interfaces for CoolBOT systems, which are completely decoupled from them. On the other side, *probes* mainly allow to implement decoupled interfaces for interoperation of CoolBOT systems with non CoolBOT applications, as depicted in the figure. Both will be explained in more detail in section 5.

In CoolBOT, systems are made of CoolBOT *components* (components for short). A component is an *active entity*, in the sense that it has its own unit of execution. It also presents a clear separation between its external interface and its internals. Components intercommunicate among them only through their external interfaces which are formed by *input and output ports*. When connected, they form *port connections*, as depicted on Fig. 1. Through them, components interchange discrete units of information termed *port packets*. *Views* and *probes* have a similar

external interface of input and output ports, hence, they can also be interconnected among them and with components using port connections. The functionality of a whole system comes up from the interaction through port connections among all the components integrating the system, including views and probes.

2.1 Port connections, ports and port packets

CoolBOT components interact among them using *port connections*. A *port connection* is defined by an output port and an input port. Port connections are established dynamically in runtime, and they are unidirectional (from output to input port), and follow a *publish/subscribe* paradigm of communication [Brugali & Shakhimardanov (2010)]. In that way, we can have multiple subscribers for the same output port, as shown in Fig. 2, and multiple publishers feeding the same input port, illustrated in Fig. 3. Note that input and output ports are decoupled in the sense that component publishers do not know necessarily who is receiving what they are publishing through their output ports. The contrary is also true, component subscribers do not necessarily know who is publishing the data which are reaching them through their input ports. Data are sent through port connections in discrete units called *port packets*.

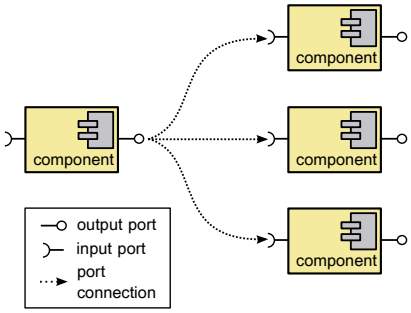

Fig. 2. Port connections: one publisher, many subscribers.

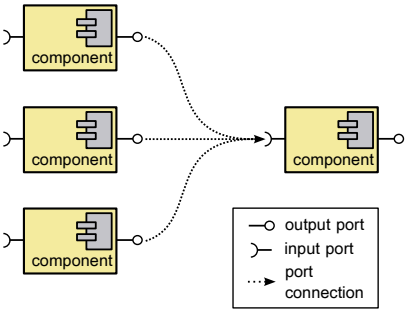

Fig. 3. Port connections: many publishers, one subscriber.

To establish a port connection the ports involved should be compatible, i.e., they must have compatible types, and should transport the same types of port packets. In particular, when defining an output or input port we have to specify three aspects for them, namely:

1. **An identifier**: This is an identifier or name for the port. It has to be unique at component (or *view* or *probe*) scope. The port identifier is what we use to refer to a specific port when establishing/de-establishing port connections.
2. **A port type**: This is the type of the port. There are several typologies available for input and output ports, and depending on how we combine them, we can establish a different model of communication for each connection. The typologies of the input and output ports involved in a connection determine the pattern and semantics of the communication through it, following the same philosophy that the communications patterns of Smartsoft [Schlegel et al. (2009)], and the interfaces for component communication available in OROCOS [*The Orocos Project* (2011)]. On Tables 1 and 2 we can see all the types of connections we have available in CoolBOT right now, we will elaborate on this later.
3. **Port packet types**: Those are the types of port packets accepted by the output or input port. Most input and output port types only accept one type of port packet through them, although we have also some of them that accept a set of different port packet types.

Bear in mind that in CoolBOT port connections are established dynamically, but the definition of each input and output port for each software component, whether a component, a view or a probe, is static. Thus, for a given component we define statically its external interface of input and output ports, each one with its identifier, port type and accepted port packet types. In opposition, port connections are established at runtime. Only a compatible pair of output and input ports can form a port connection, and we say that they are compatible when two conditions fulfill: first, they have compatible port types (the compatible combinations are shown in Tables 1 and 2); and second, the port packets the pair of port accepts also match.
As commented, Tables 1 and 2 show all the possible types and combinations of output and input ports available in CoolBOT. As we can observe we have two groups of types of port connections depending on the types of the output and input ports we connect, namely:

- ***Active Publisher/Passive Subscriber (AP/PS)* connections**. In this kind of connections we say the publisher is the *active* part of the communication through the connection, since it is the publisher (the sender) who invest more computational resources in the communication. More specifically, in these connections, there is a buffer (a cache, a memory) in the input port where incoming port packets get stored, when they get to the subscriber (the receiver) end. We say the publisher is active, because the copy of port packets in the input port buffers is made by the publisher's threads of execution. Those memories get signaled for the subscribers to access them at their own pace. Evidently, if the output port has several subscribers, the publisher has to make a copy for all of them, so the computational cost for copies increases and this cost is afforded by the publisher. Table 1 enumerates all the available types of port connections following this model of communication.

- ***Passive Publisher/Active Subscriber (PP/AS)* connections**. Those connections follow a model of communication where the subscriber plays the *active* role in the communication, in the sense that in opposition to the previous ones, the subscriber is the part of the communication which invests more computational resources. In this type of connections, we have buffers at both ends of the port connection. When the publisher sends (publishes) a port packet through the connection, it gets stores in a buffer in the output port, and the input port gets signaled, in order to notify the subscriber that there are new data on the connection. Note that the publisher does not copy the port packet on the subscriber's input port buffer. It is the subscriber the one who copies the port packet on its input port buffer when it access its input port in order to get fresh port packets, stored at the other end of the port connection. In this way, when we have several subscribers and one publisher the computational cost of copies is afforded by each one of the subscribers separately.

Apart from the computational cost of using AP/PS connections versus PP/AS connections, there is another important aspect to take into account when using any of them. PP/AS connections are *persistent* in the sense that, as explained in Table 2 the last data (port packet) sent by the publisher through the output port is stored there, in the output port buffer, so subscribers which are connected to this output port once the last port packet was sent, can access those data posteriorly. On the contrary AP/PS connections are not persistent, because port packets are stored on the subscriber end, so packets, once they have been sent, are not available for new subscribers, because port packets only gets to those subscribers who are connected to the output port right at the moment of sending them.
Other important aspect to take into account with respect to port connections is that they follow an asynchronous model of communications. Note that they are unidirectional, port packets go from the publisher's output port to the subscriber's input port, the publisher sends port

Active Publisher/Passive Subscriber (AP/PS)		
Output Port	**Input Port**	**Port Connection Type**
tick	*tick*	*tick* connections: those connections do not transport any port packet, they only communicate the occurrence of an event.
generic	*last*	*last* connections: the input port stores always the *last* port packet sent through the connection by the publisher (publishers). Only one type of port packet is accepted through the port connection.
	fifo	*fifo* connections: at the input port there is a circular fifo with a specific length. Port packets sent through the port connection by publishers, get stored there. Only one type of port packet is accepted through the port connection.
	ufifo	*unbounded fifo* connections: at the input port there is a fifo with a specific length. Port packets sent through the port connection by publishers, get stored there. When the fifo is full and port packets keep coming the fifo grows in order to store them. Only one type of port packet is accepted through the port connection.
multipacket	*multipacket*	*multipacket* connections: those connections accept a set of port packet types. There is a different buffer for each accepted port packet type, the last port packet of each type which is sent through the connection by publishers gets stored on them.
lazymultipacket		*lazymultipacket* connections: those connections accept a set of port packet types. At the input port there is a different buffer for each accepted port packet type, the last port packet of each type which is sent through the connection by publishers gets stored on them. On the output port, port packets get stored in a queue of port packets to send, they are really sent to the other end when a *flush* operation is applied by the publisher on the output port.

Table 1. Available port connections types: *active publisher/passive subscriber (AP/PS)* connections.

Passive Publisher/Active Subscriber (PP/AS)		
Output Port	**Input Port**	**Port Connection Type**
poster	*poster*	*poster* connections: there is a buffer at the output port where the last packet send by the publisher gets stored. There is another buffer at the input port which is "synchronized" with the output port buffer when the subscriber accesses its input port in order to get the last port packet sent through the connection. Therefore, the port packet gets copied to the input port only when a new port packet has been stored at the output port end. Only one type of port packet is accepted through the port connection.

Table 2. Available port connections types: *passive publisher/ active subscriber (PP/AS)* connections.

packets and keeps doing something different, the subscriber gets packets at its own pace and not necessarily at the right moment they get to its input ports.

As to port packets, when defining an output or input port, we have to specify which port packet type or types (depending on the port type being defined), the port will accept. In general, port packet types are defined by the user, as we will see in section 3, we may also use port packets types provided by CoolBOT itself (the available ones are shown in Fig. 3), port packet types previously developed, or third party port packet types.

Port packet type	Description
`PacketUChar`	Transports a C++ `unsigned char`.
`PacketInt`	Transports a C++ `int`.
`PacketLong`	Transports a C++ `long`.
`PacketDouble`	Transports a C++ `double`.
`PacketTime`	Transports a CoolBOT `Time` value (a time-stamp).
`PacketCoordinates2D`	Transports a CoolBOT `Coordinates2D` value (stores a 2D point).
`PacketFrame2D`	Transports a CoolBOT `Frame2D` value (stores a 2D frame).
`PacketCoordinates3D`	Transports a CoolBOT `Coordinates3D` value (stores a 3D point).
`PacketFrame3D`	Transports a CoolBOT `Frame3D` value (stores a 3D frame).

Table 3. Available port packet types provided by CoolBOT itself.

2.2 CoolBOT components

CoolBOT components are *active objects* [Ellis & Gibbs (1989)], as [Brugali & Shakhimardanov (2010)] states: "a component is a computation unit that encapsulates data structures and operations to manipulate them". Moreover, in CoolBOT, components can be seen as "data-flow machines", since they process data when they dispose of it at their input ports. Otherwise, they stay idle, waiting for incoming port packets. On the other side, components send processed data in form of port packets through their output ports. All in all, the model of computation of CoolBOT systems follows the *Flow Based Programming* (FBP) paradigm according to [J. Paul Morrison (2010)], so systems can be built as networks of components interconnected by means of port connections. More formally, CoolBOT components are modeled as *port automata* [Steenstrup et al. (1983)][Stewart et al. (1997)]. Fig. 4 provides a view of the structure of a component in CoolBOT. There is a clear separation between its external interface and its internal structure. Externally, a component can only communicate with other components (and views and probes) through its input and output ports. Thence, a component's external interface comprises all its input and output ports, its types, and the port packets it accepts through them. As we can see on the figure there are two special ports in any component: the *control port* and the *monitoring port*, the rest of the ports are user defined. The *control port* allows to modify component's *controllable variables*. Through the *monitoring port* components publish their *observable variables*. Both ports allows an external supervisor (i.e. another component, a view or a probe) to observe and modify the execution and configuration of a given component.

Internally a component is organized as an automaton, as illustrated in Fig. 4. All components follow the same automaton structure. The part of this automaton which is common to all components is called the *default automaton*, and comprises several states, namely: *starting*, *ready*, *suspended*, *end*, and four states for component exception handling. This structure allows for an external supervisor to control the execution of any component in a system using an standard protocol, likewise in an operating system where threads and processes transit among different states during their lifetime. To complete the automaton of a component the user defines the *user automaton* which is specific for each component and implements its functionality. This is represented in Fig. 4 with a dotted circle as the meta-state *running*. Transitions among component's automaton states are triggered by incoming port packets through any of its input ports, and also by internal events (timers, empty transition, a control variable that has been modified by an external supervisor, entering or exiting a state, etc.). The user can associate C++ callbacks to transitions, much like the *codels* for G^{en}oM [Mallet et al. (2010)] modules.

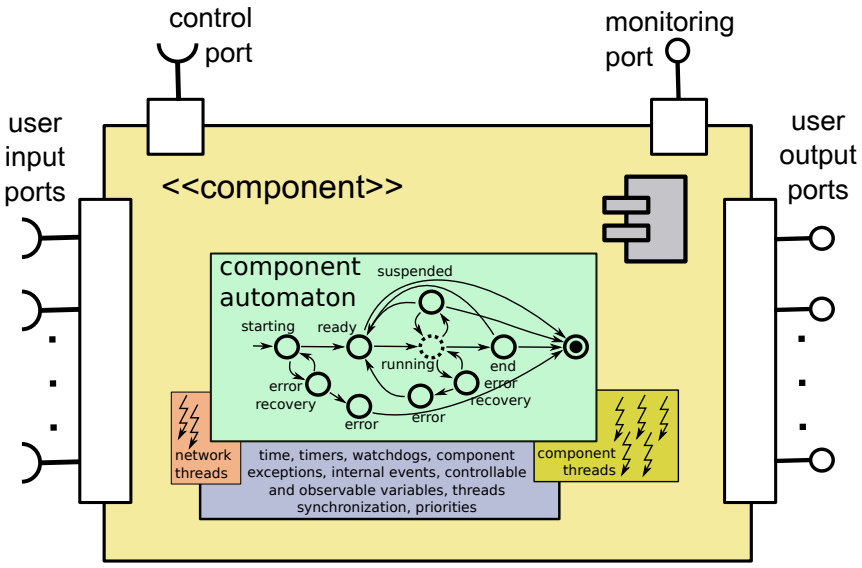

Fig. 4. CoolBOT. Component structure.

A key design principle for CoolBOT components is to take advantage of multithreaded and multicore capacities of mainstream operating systems, and the infrastructure they provide for multithreaded programming. Another key principle of design for components was to separate functional logic from thread synchronization logic. The user should be only worried about the functional logic, synchronization issues should be completely transparent to the developer. CoolBOT should be responsible for them behind the scenes. As active objects, CoolBOT components can organize its execution using multiple threads of execution as depicted on Fig. 4. Those threads are mapped on the underlying operating system (see Fig. 5). Thus, when developing a component the user assigns disjointly threads to automaton states, and to input ports and internal events provoking transitions. Those transitions, i.e. their associated callbacks, will be executed by the specific threads being assigned. The synchronization among them is guaranteed by the underlying framework infrastructure. All components are endowed at least with one thread of execution; the rest, if any, are user defined.
As depicted in Fig. 1, CoolBOT provides means for distributed computation. A given system can be mapped on a set formed by different machines sharing a computer network. Port connections among components, views and probes are transparently multiplexed using TCP/IP connections (see section 4). Furthermore, each machine can host one or several CoolBOT *integrations*. A CoolBOT *integration* is an application (a process) which integrates instances of CoolBOT components, views and probes. Integrations can be instantiated in any machine, and are user defined using a description language as we will see in next section.

3. CoolBOT development tools

CoolBOT provides several tools for helping developers. Fig. 6 shows the main ones, namely: `coolbot-ske` and `coolbot-c`. The former one, `coolbot-ske`, is used to create a directory structure for development of CoolBOT components, probes, port packets, views and integrations. It also generates CMake [Kitware, Inc. (2010)] template files for compiling them, description language template files for `coolbot-c`, and test programs for components. The

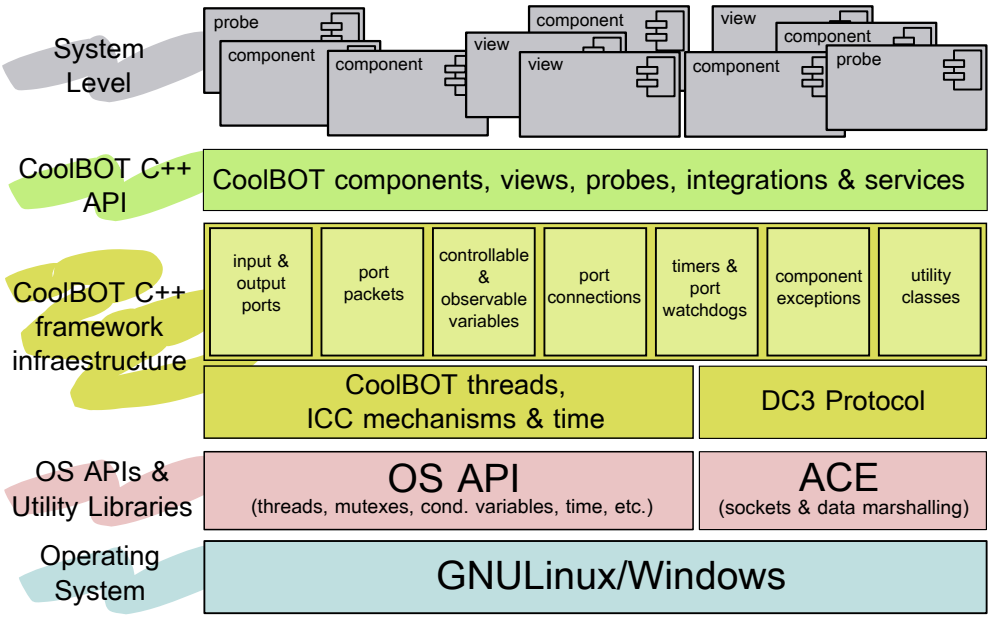

Fig. 5. Abstraction layers in CoolBOT.

latter tool, the CoolBOT compiler `coolbot-c`, generates C++ skeletons for components, port packets, views and integrations, and for each component it also generates its corresponding probe. All, the probe and the skeletons are C++ classes, and CoolBOT uses a description language as source code to generate those C++ classes. Except for probes, which are complete functional C++ classes, `coolbot-c` generates incomplete C++ classes which constitute the mentioned C++ skeletons. They are incomplete in the sense that they lack functionality, the user is responsible for completing them. Once completed, and using the CMake templates provided by `coolbot-ske`, they can be compiled. Components, probes, port packets, and views compile yielding dynamic libraries, integrations compile yielding executable programs. Moreover, the `coolbot-c` compiler preserves information when recompiling description files which have been modified, in such a way that, all C++ code introduced by the user into the skeletons is preserved.

4. Transparent distributed computation

Transparent distributed computation is the first development we have integrated on CoolBOT in order to improve its use and operation. The main idea was to make network communications as transparent as possible to developers (and components). We wanted CoolBOT to be responsible for them on behalf of components. Thus, at system level, to connect two component instances instantiated in different CoolBOT integrations should be as easy as connecting them when instantiated in the same integration. In particular, we follow three main principles related to transparent distributed computation facilities: transparent network inter component communications, network decoupling of component's functional logic, and incremental design.

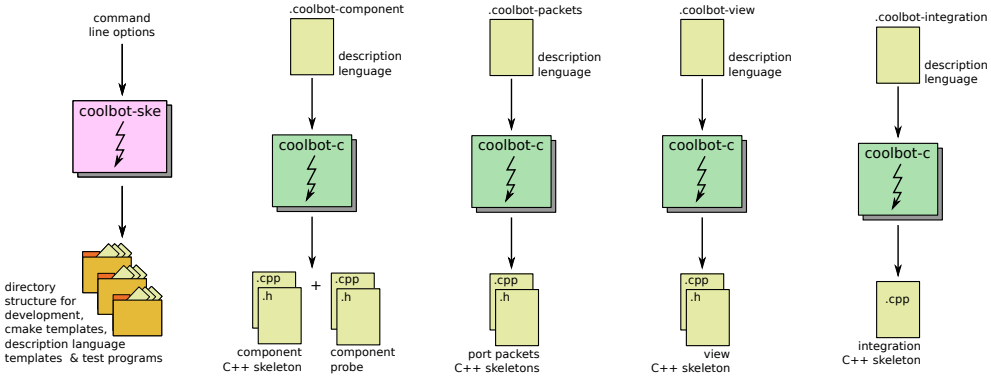

Fig. 6. CoolBOT's software development process.

4.1 Transparent network inter component communications

In order to make network communications transparent to components, we have developed a protocol termed *Distributed CoolBOT Component Communication Protocol (DC3P)* to multiplex port connections over TCP connections established among the components involved. In the current version of CoolBOT, only the TCP protocol is supported for network connections. The integration of the UDP protocol is under development, and it is expected for next CoolBOT version. DC3P has been implemented using the TCP/IP socket wrappers and the marshalling facilities provided by the ACE library [Douglas C. Schmidt (2010)], illustrated in Fig. 5. The protocol consists of the following packets:

- *Port Type Info* (request & response): For asking type information about input and output ports through network connections. This allows port connection compatibility verification when establishing a port connection through the network.
- *Connect* (request & response): For establishing a port connection over TCP/IP.
- *Disconnect* (request & response): To disconnect a previous established port connection.
- *Data Sending*: Once a port connection is established over TCP/IP, port packets are sent through it using this DC3P packet.
- *Remote Connect* (request & response): For establishing port connections between two remote component instances. Permits to connect component instances remotely.
- *Remote Disconnect* (request & response): To disconnect port connections previously established between two remote component instances.
- *Echo Request & Response*: Those DC3P packets are useful to verify that the other end in a network communication is active and responding.

All DC3P packets and port packets sent through port connections are marshalled and unmarshalled in order to be correctly sent through the network. We have used the facilities ACE provides for marshalling/demarshalling based on the OMG Common Data Representation (CDR) [Object Management Group (2002b)]. In general, port packets are user defined. In order to make their marshalling/demarshalling as easy and transparent as possible for developers, the description language accepted by the `coolbot-c` compiler includes sentences for describing port packets (as we can observe in Fig. 6), much like CORBA IDL [Object Management Group (2002a)]. The compiler generates a C++ skeleton class for each port packet where the code for marshalling/demarshalling is part of the skeleton's code

generated automatically. In addition, we have endowed also CoolBOT with a rich set of C++ templates and classes to support marshalling and demarshalling of port packets (or any other arbitrary C++ class).

4.2 Network decoupling of component's functional logic

Another important aspect for network communication transparency is the decoupling of network communication logic from the functional logic of the component. Fig. 4 illustrates how this decoupling has been put into practice. Each component is endowed with a pair of network threads, and *output network thread*, and an *input network thread*, which are responsible for network communications using DC3P. CoolBOT guarantees transparently thread synchronization between them and the functional threads of the component. The network threads are mainly idle, waiting to have port packets to send through open network port connections, or to receive incoming port packets that should be redirected to the corresponding component's input ports. At instantiation time, it is possible to deactivate the network support for a component instance (and also for views and probes instances). In this manner, the component is not reachable from outside the integration where it has been instantiated, and evidently network threads and the resources they have associated are not allocated.

4.3 Incremental design

In future versions of CoolBOT, it is very possible that the set of DC3P protocol packets grow with new ones. In order to allow an easy integration of new DC3P packets in CoolBOT, we have applied the *composite* and *prototype* patterns [Gamma et al. (1995)] to their design. Those design patterns, jointly with the C++ templates and classes to support marshalling and demarshalling provide a systematic and easy manner of integrating new DC3P packets in future versions of the framework.

5. Deeper interface decoupling: Views and probes

Inspired by one of the "good practices" proposed by the authors of Carnegie Mellon's Navigation Toolkit CARMEN [Montemerlo et al. (2003)]: "one important design principle of CARMEN is the separation of robot control from graphical displays", we have introduced in CoolBOT the concept of *view* as an integrable, composite and reusable graphical interface available for CoolBOT system integrators and developers. Thus, CoolBOT *views* are graphical interfaces which, as software components, may be interconnected with any other component, view or probe in a CoolBOT system. In Fig. 7 is depicted the structure of a view in CoolBOT. As shown, CoolBOT views are also endowed with an external interface of input and output ports. Through this interface the view can communicate with other components, views and probes present in a given system. Identically to components, views are provided with the same network thread support which allows transparent and decoupled network communications through port connections. Internally, a view is a graphical interface, in fact, the current views already developed and operational which are available in CoolBOT have been implemented using the GTK graphical library [*The GTK+ Project* (2010)]. As shown in Fig. 6, C++ skeletons for views are generated using the `coolbot-c` compiler. The part which should be completed by the user is precisely the graphical implementation, which can be done using directly the GTK library or a GUI graphical programming software for designing window-based interfaces like Glade [*Glade - A User Interface Designer* (2010)].

As depicted in Fig. 7 a CoolBOT *probe* is provided with an external interface of input and output ports, and likewise component and views, as software components, this allows them to

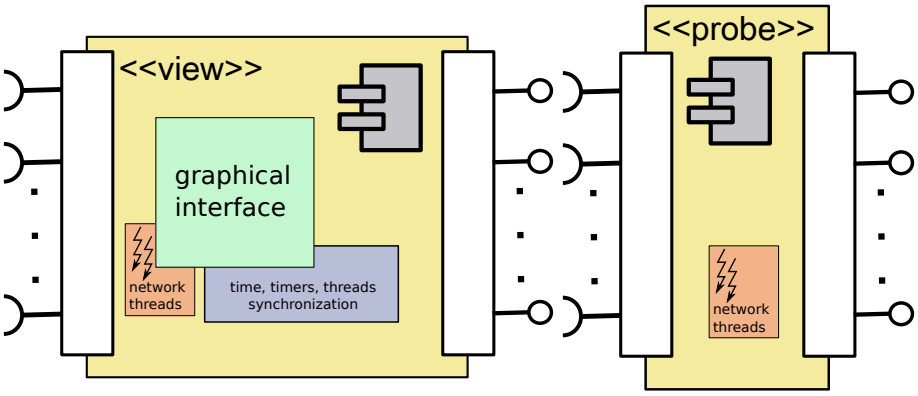

Fig. 7. CoolBOT. View and probe structures.

be interconnected with other components, views or probes. Equally they implement the same network decoupled support of threads for transparent network communications. In CoolBOT, *probes* are devised as interfaces for interoperability with non CoolBOT software, as illustrated graphically in Fig. 1. A complete functional C++ class implementing a probe is generated when a component is compiled by `coolbot-c`. The probe implements the complementary external interface of their corresponding component. Those probes generated automatically can be seen as automatic refactorings of external component interfaces in order to support interoperability of CoolBOT components with non CoolBOT software. As mentioned in [Makarenko et al. (2007)] this is an important factor in order to facilitate integration of different development robotic software approaches.

6. A real integration

In its current operating version, CoolBOT has been mainly used to control mobile robotic systems with the platforms we have available at our laboratory: Pioneer mobile robots models 3-DX, and P3-AT from Adept Mobile Robots [*Adept Mobile Robots* (2011)] (former Activ Media Robotics).

In this section we will show next a real robotic system using one of our Pioneer 3-DX mobile robots, in order to give a glimpse of a real system controlled using CoolBOT. The system is illustrated in Fig. 8. The example shows a secure navigation system for an indoor mobile robot. This is a real application we have usually in operation on the robots we have at the laboratory. The system implements a secure navigation system based on the ND+ algorithm for obstacle avoidance [Minguez et al. (2004)]. It has been implemented attending to [Montesano et al. (2006)]. In the figure, input ports, output ports, and port connections, have been reduced for the sake of clarification. Some of them represent several connections and ports in the real system.

The system is organized using two CoolBOT integrations, one only formed by CoolBOT component instances, and the other one containing CoolBOT view instances. The former one is really the integration which implements the navigation system. As we can observe, it consists of five component instances, namely: `PlayerRobot` (this is a wrapper component for hardware abstraction using the Player/Stage project framework [Vaughan et al. (2003)]), `MbICP` (this is a component which implements the MbICP scan matching algorithm based on laser range sensor data [Minguez et al. (2006)]), `GridMap` (this component maintains a grid map of the surroundings of the robot built using robot range laser scans, it also generates

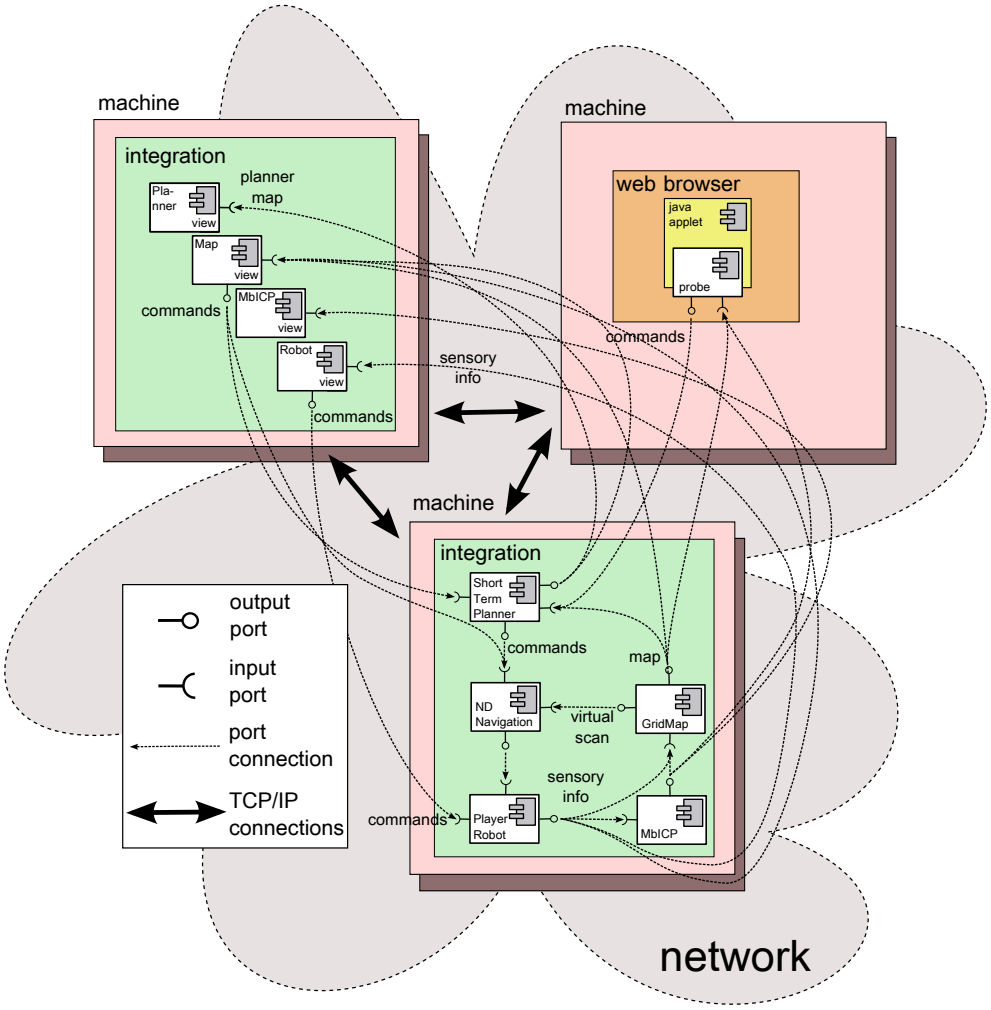

Fig. 8. CoolBOT. Secure Navigation System.

periodically a 360^o virtual scan for the ND+ algorithm), `NDNavigation` (implements the ND+ algorithm) and `ShortTermPlanner` (a planner which uses the grid map for planning paths in the robot surroundings using a modification of the numerical navigation function NF2 found in [Jean-Claude Latombe (1991)]). On other machine another integration is shown hosting four view instances through which we can control and monitor the system remotely. In addition, in another machine, there is a web browser hosting a Java applet using a CoolBOT *probe* to connect to some of the components of the system.

In order to clarify how the integration of Fig. 8 has been built, and also to clarify the process of development of each of its components, in next paragraphs we will have a look at the description files used to generate some of them, including the whole integration shown in the figure. Thus, in Fig. 9 we can see the description file accepted by `coolbot-c` of one of the

```
/*
 * File: player-robot.coolbot
 * Description: description file for PlayerRobot component
 * Date: 02 June 2010
 * Generated by coolbot-ske
 */

component PlayerRobot
{
  header
  {
    author "Antonio Carlos Domínguez Brito <adominguez@iusiani.ulpgc.es>";
    description "PlayerRobot component";
    institution "IUSIANI - Universidad de Las Palmas de Gran Canaria";
    version "0.1"
  };

  constants
  {
    LASER_MAX_RANGE="LaserPacket::LASER_MAX_RANGE";
    SONAR_MAX_RANGE=5000; // millimeters

    private FIFO_LENGTH=5;
    private ROBOT_DATA_INCOMING_FREQUENCY= 10; // Hz
    private LASER_MIN_ANGLE= -90; // degrees

    ...
  };

  // input ports
  input port Commands type fifo port packet CommandPacket length FIFO_LENGTH;
  input port NavigationCommands type fifo port packet NDNavigation::CommandPacket length FIFO_LENGTH;

  //output ports
  output port RobotConfig type poster port packet ConfigPacket;
  output port Odometry type generic port packet OdometryPacket network buffer FIFO_LENGTH;
  output port OdometryReset type generic port packet PacketTime;
  output port BumpersGeometry type poster port packet "BumperGeometryPacket";
  output port Bumpers type generic port packet BumperPacket;
  output port SonarGeometry type poster port packet "SonarGeometryPacket";
  output port SonarScan type generic port packet "SonarPacket";
  output port LaserGeometry type poster port packet PacketFrame3D;
  output port LaserScan type generic port packet LaserPacket;
  output port Power type generic port packet PacketDouble;
  output port CameraImage type poster port packet CameraImagePacket;
  output port PTZJoints type generic port packet "PacketPTZJoints";

  exception RobotConnection
  {
    description "Robot connection failed.";
  };

  exception NoPositionProxy
  {
    description "Position proxy not available in this robot.";
  };

  exception InternalPlayerException
  {
    description "A Player library exception has been thrown.";
  };

  entry state Main
  {
    transition on Commands,NavigationCommands,Timer;
  };
};
```

Fig. 9. `player-robot.coolbot-component`: `PlayerRobot`'s description file.

components of Fig. 8, file `player-robot.coolbot-component`, corresponding to component `PlayerRobot`. As to views, in Fig. 10, we can see the description file for one of the view instances of Fig. 8, concretely for the `Map` view in the figure, which is an instance of view `GridGtk`. As we can observe, the description file specifies mainly the view's external interface formed by input and output ports.

```
/*
 * File: grid-gtk.coolbot-view
 * Description: description file for GridGtk view
 * Date: 29 April 2011
 * Generated by coolbot-ske
 */

view GridGtk
{
  header
  {
    author "Antonio Carlos Domínguez-Brito <adominguez@iusiani.ulpgc.es>";
    description "GridGtk View";
    institution "IUSIANI - ULPGC (Spain)";
    version "0.1"
  };

  constants
  {
    private DEFAULT_REFRESHING_PERIOD=500; // milliseconds
    ...
  };

  // input ports
  input port ROBOT_CONFIG type poster port packet PlayerRobot::ConfigPacket;
  input port GRID_MAP type poster port packet GridMap::GridMapPacket;
  input port PLANNER_PATH type last port packet ShortTermPlanner::PlannerPathPacket;

  // output ports
  output port PLANNER_COMMANDS type generic port packet ShortTermPlanner::CommandPacket;
  output port ND_COMMANDS type generic port packet NDNavigation::CommandPacket;
};
```

Fig. 10. `grid-gtk.coolbot-view`: `GridGtk`'s description file.

In Fig. 11 it is shown a snapshot of the view in runtime. Once developed CoolBOT views are graphical components we can integrate in a window-based application like the one shown in the figure. In particular, in the application we can see, views are plugged in as "pages" of the GTK widget `notepad` (a container of "pages" with "tabs") we can observe in the figure. In fact, the GUI application shown integrates the four view instances of Fig. 8, whose C++ skeleton has been generated using also `coolbot-c` from a description file (the `.coolbot-integration` file in Fig. 6). In Fig. 12 we can see part of this file. Notice that `coolbot-c` generates C++ skeletons for integrations where the static instantiation and interconnection among components and views are generated automatically. If we want to build a dynamic integration, in terms of dynamic instantiation of components and views, and also in terms of dynamic interconnections among them establishing port connections, we must complete the dynamic part of the skeleton generated by `coolbot-c` using the C++ runtime services provided by the framework (Fig. 5).
With respect to CoolBOT probes, as of now we have used them to interoperate with Java applets inserted in a web browser, as shown in Fig. 8. More specifically we have used SWIG [*SWIG* (2011)] in order to have access to the probe C++ classes in Java with the aim of implementing several Java GUI interfaces in Java equivalent to some of the CoolBOT views we have already developed. In Fig. 13 we can see a snapshot of the Java equivalent of a view to represent the range sensor information of a mobile robot.

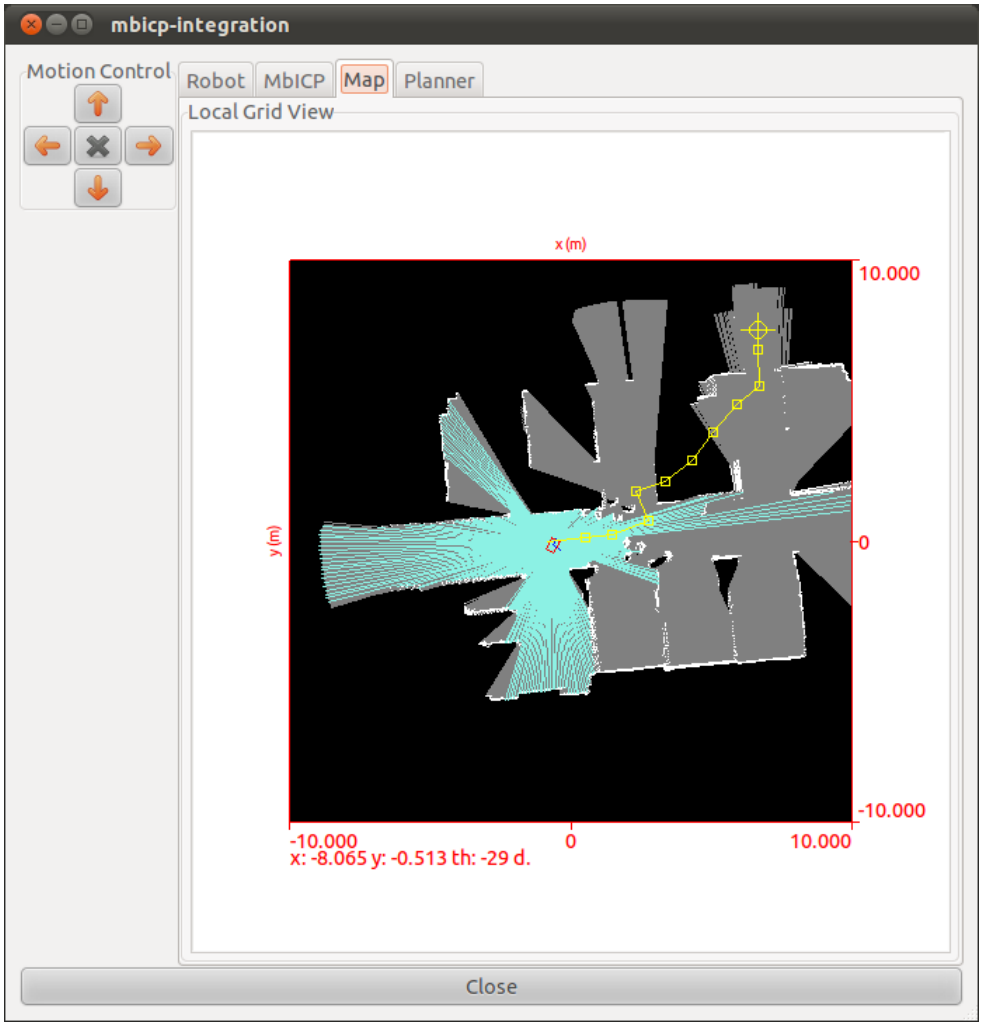

Fig. 11. View `GridGtk`'s snapshot.

```
/*
 * File: mbicp-integration.coolbot-integration
 * Description: description file for mbicp-integration integration.
 * Date: 29 April 2011
 * Generated by coolbot-ske
 */

integration mbicp_integration
{
  header
  {
    author "Antonio Carlos Domínguez-Brito <adominguez@iusiani.ulpgc.es>";
    description "MbICP's views integration";
    institution "IUSIANI - ULPGC (Spain)";
    version "0.1"
  };

  machine addresses
  {
    local my_machine: "127.0.0.1";
    the_other_machine: "...";
  };

  listening ports // TCP/IP ports
  {
    ROBOT_PORT: 1950;
    MBICP_PORT: 1965;
    NAVIGATION_MAP_PORT: 1970;
    ND_PORT: 1980;
    NAVIGATION_PLANNER_PORT: 1990;

    ROBOT_VIEW_PORT: 1955;
    MBICP_VIEW_PORT: 1985;
    NAVIGATION_MAP_VIEW_PORT: 1975;
    NAVIGATION_PLANNER_VIEW_PORT: 1995;
  };

  local instances
  {
    view robotView:PlayerRobotGtk listening on ROBOT_VIEW_PORT with description "Robot";
    view mbicpView:MbICPGtk listening on MBICP_VIEW_PORT with description "MbICP";
    view mapView:GridGtk listening on NAVIGATION_MAP_VIEW_PORT with description "Map";
    view navigationPlannerView:PlannerGtk listening on NAVIGATION_PLANNER_VIEW_PORT with description "Planner";
  };

  remote instances on the_other_machine;
  {
    component robot:PlayerRobot listening on ROBOT_VIEW_PORT;
    component mbicpInstance:MbICPCorrector listening on MBICP_PORT;
    component navigationMap:GridMap listening on NAVIGATION_MAP_PORT;
    component nd:NDNavigation listening on ND_PORT;
    component navigationPlanner:ShortTermPlanner listening on NAVIGATION_PLANNER_VIEW_PORT;
  };

  port connections // static connections
  {
    connect robot:ODOMETRY to robotView:ODOMETRY;
    connect robot:LASERGEOMETRY to robotView:LASER_GEOMETRY;
    connect robot:LASERSCAN to robotView:LASER_SCAN;
    connect robot:POWER to robotView:POWER;
    connect robot:SONARGEOMETRY to robotView:SONAR_GEOMETRY;
    connect robot:SONARSCAN to robotView:SONAR_SCAN;

    connect robot:ODOMETRY to mbicpInstance:ODOMETRY;
    connect robot:LASERGEOMETRY to mbicpInstance:LASER_GEOMETRY;
    connect robot:LASERSCAN to mbicpInstance:LASER_SCAN;

    connect robotView:COMMANDS to robot:COMMANDS;

    ...
  };
};
```

Fig. 12. The integration containing `Robot`. `MbICP`, `Map` and `Planner` views of Fig. 8

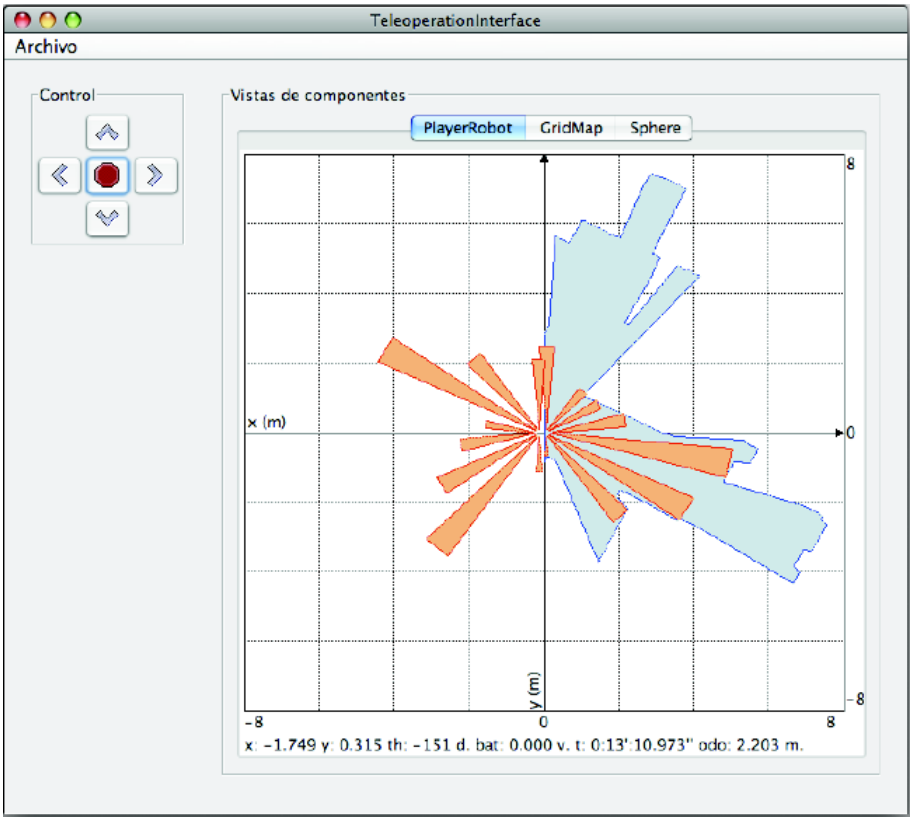

Fig. 13. Java view implemented using a probe to access range sensor information for a mobile robot. A snapshot.

7. Conclusions

In this document we have presented the last developments which have been integrated in the last operating version of CoolBOT. The developments have been aimed mainly to two questions: transparent distributed computation, and "deeper" interface decoupling. It is our opinion that the use and operation of CoolBOT has improved considerably. CoolBOT is an open source initiative supported by our laboratory which is freely available via **www.coolbotproject.org**, including the secure navigation system depicted in Fig. 8.

8. References

Adept Mobile Robots (2011). http://www.mobilerobots.com/.

Ando, N., Suehiro, T. & Kotoku, T. (2008). A Software Platform for Component Based RT-System Development: OpenRTM-Aist, *in* S. Carpin, I. Noda, E. Pagello, M. Reggiani & O. von Stryk (eds), *Simulation, Modeling, and Programming for Autonomous Robots*, Vol. 5325 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 87–98.

Antonio C. Domínguez-Brito, Daniel Hernández-Sosa, José Isern-González & Jorge Cabrera-Gámez (2007). *Software Engineering for Experimental Robotics*, Vol. 30 of *Springer Tracts in Advanced Robotics Series*, Springer, chapter CoolBOT: a Component Model and Software Infrastructure for Robotics, pp. 143–168.

Basu, A., Bozga, M. & Sifakis, J. (2006). Modeling heterogeneous real-time components in BIP, In Fourth IEEE International Conference on Software Engineering and Formal Methods, pages 3-12, Pune (India).

Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I. & Thanh-Hung, N. (2009). Designing autonomous robots, *IEEE Robotics and Automation Magazine* 16(1): 67–77.

Brooks, A., Kaupp, T., Makarenko, A., S.Williams & Oreback, A. (2005). Towards component-based robotics, *In IEEE International Conference on Intelligent Robots and Systems*, Tsukuba, Japan, pp. 163–168.

Brugali, D. (ed.) (2007). *Software Engineering for Experimental Robotics*, Springer Tracts in Advanced Robotics, Springer.

Brugali, D. & Scandurra, P. (2009). Component-based robotic engineering (part i) [tutorial], *Robotics Automation Magazine, IEEE* 16(4): 84 –96.

Brugali, D. & Shakhimardanov, A. (2010). Component-based robotic engineering (part ii), *Robotics Automation Magazine, IEEE* 17(1): 100 –112.

Domínguez-Brito, A. C., Hernández-Sosa, D., Isern-González, J. & Cabrera-Gámez, J. (2004). Integrating Robotics Software, IEEE International Conference on Robotics and Automation, New Orleans, USA.

Douglas C. Schmidt (2010). The Adaptive Communication Environment (ACE), www.cs.wustl.edu/∼schmidt/ACE.html.

Ellis, C. & Gibbs, S. (1989). *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley, chapter Active Objects: Realities and Possibilities.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley.

George T. Heineman & William T. Councill (2001). *Component-Based Software Engineering*, Addison-Wesley.

Glade - A User Interface Designer (2010). glade.gnome.org.

J. Paul Morrison (2010). *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*, CreateSpace.

Jean-Claude Latombe (1991). *Robot-motion planning*, The Kluwer International Series in Engineering and Computer Science, Kluwer Academic.

Kitware, Inc. (2010). The CMake Open Source Build System, www.cmake.org.

Makarenko, A., Brooks, A. & Kaupp, T. (2007). On the benefits of making robotic software frameworks thin, IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07), San Diego CA, USA.

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S. & Ingrand, F. (2010). GenoM3: Building middleware-independent robotic components, IEEE International Conference on Robotics and Automation.

Minguez, J., Montesano, L. & Lamiraux, F. (2006). Metric-based iterative closest point scan matching for sensor displacement estimation, *Robotics, IEEE Transactions on* 22(5): 1047 –1054.

Minguez, J., Osuna, J. & Montano, L. (2004). A "Divide and Conquer" Strategy based on Situations to achieve Reactive Collision Avoidance in Troublesome Scenarios, IEEE International Conference on Robotics and Automation, New Orleans, USA.

Montemerlo, M., Roy, N. & Thrun, S. (2003). Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit, Vol. 3, pp. 2436 – 2441 vol.3.

Montesano, L., Minguez, J. & Montano, L. (2006). Lessons Learned in Integration for Sensor-Based Robot Navigation Systems, *International Journal of Advanced Robotic Systems* 3(1): 85–91.

Object Management Group (2002a). OMG IDL: Details, (http://www.omg.org/gettingstarted/omg_ idl.htm).

Object Management Group (2002b). The Common Object Request Broker: Architecture and Specification, Ch. 15, Sec. 1-3. (http://www.omg.org/cgi-bin/doc?formal/02-06-01).

ROS: Robot Operating System (2011). http://www.ros.org.

Schlegel, C., Haßler, T., Lotz, A. & Steck, A. (2009). Robotic Software Systems: From Code-Driven to Model-Driven Designs, In Proc. 14th Int. Conf. on Advanced Robotics (ICAR), Munich.

Steenstrup, M., Arbib, M. A. & Manes, E. G. (1983). Port automata and the algebra of concurrent processes, *Journal of Computer and System Sciences* 27: 29–50.

Stewart, D. B., Volpe, R. A. & Khosla, P. (1997). Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, *IEEE Transactions on Software Engineering* 23(12): 759–776.

SWIG (2011). http://www.swig.org/.

The GTK+ Project (2010). www.gtk.org.

The Orocos Project (2011). http://www.orocos.org.

Vaughan, R. T., Gerkey, B. & Howard, A. (2003). On Device Abstractions For Portable, Reusable Robot Code, *IEEE/RSJ International Conference on Intelligent Robot Systems (IROS 2003), Las Vegas, USA, October 2003*, pp. 2121–2427.

View publication stats