Publish/Subscribe Middleware for Robotics: Requirements and State of the Art

Matteo Matteucci

January 13, 2003

Abstract

Autonomous robotic systems become more complex and based on a distributed architecture since the introduction of the multi–agent paradigm. To tackle the increased complexity of such systems, researchers have started to apply software engineering tools in designing and integrating the modules that compose a modern robotics apparatus. Heterogeneity and efficiency are only two of the key issues in choosing or developing the integration technology for multi–robot systems. In this work, we propose the publish/subscribe model as the proper communication model in this field, we describe the issues related to the use of such kind of middleware in autonomous robotics, and we provide a compared analysis of the publish/subscribe middleware currently used in robotics applications.

1 Introduction

Distributed computing is quickly moving to a new conception of computer interaction. The new scenario of ubiquitous and pervasive computing [44] is leading to a loose peer–to–peer model, in which each processing unit becomes a node of a large and complex network [59].

A similar trend characterizes also automation and industrial robotics systems. The new wave of smart–sensors and smart–actuators [2] underlines how, even in the industrial environment, the outcome in having more computational power distributed on the field is highly remarkable. In this scenario, hardwired electronic solutions and handcrafted communication protocols on slow serial lines, are leaving their place to flexible processing units networked through broadband infrastructures, and communicating via standard protocols, both at physical and at logical levels. Recently proposed communication standards over specific field–bus systems prove such a trend [54, 74].

In a similar way, autonomous robotic systems, particularly those involving multiple robots and environmental sensors, are becoming increasingly distributed and networked. This trend results in the need of carefully considering the issue of integration even in robotics, and this happens both at software level and at the hardware level. Each single device (e.g., sensor, controlled actuator, controller, CPU, DSP, etc.), that used to be a

functional part of a whole, now starts being an active and, in some way, independent member. People involved in robotics are getting conscious that developing multi–robot systems might require tools for supporting the developers in dealing with integration and communication issues [30].

Few robotic control architectures already provide a sophisticated API, but it seems that a lot of robot programmers ignore many of the features provided by such frameworks, relying instead on a more primitive "socket to the base" type of interface. Given these premises, a question become relevant: can we depict a proper framework for integration in distributed robotics systems that could help in a natural way the development of these applications? In this paper we are interested in investigating whether a publish/subscribe middleware can alleviate the unnecessary complexities of using low–level tools while still preserving system performance.

The present work has a twofold goal. First of all, we are interested in proposing the publish/subscribe model as an expressive and efficient communication substrate for distributed robotics applications. In the second place, we want to propose a set of requirements for publish/subscribe middlewares in order to be effectively applied in robotics. A qualitative analysis of inter–process communication toolkit for robotics has been done by Gowdy in [32], but this work does not consider the publish/subscribe paradigm and the requirements proposed for the toolkits differ noticeably from ours. Here, to show in what extent the publish/subscribe middlewares currently applied respect the requirements we propose, we reclassify few of the systems described by Gowdy in his report[1] and we add few others.

In Section 2 we describe a reference scenario for autonomous robotics applications, and in Section 3 we introduce the main characteristics of the data flow with respect to the robotics scenario. The main models for data exchange are reported in Section 4 and the main characteristics required for a publish/subscribe middleware to be applied in robotics are described in Section 5. In Section 6 we analyze some of the middleware actually used in this kind of applications and a brief discussion is presented in Section 7. Conclusions and future works are described in Section 8.

2 A New Reference Scenario

Since the first proposals concerning multi–robot systems were issued, topics such as work sharing, resource allocation, etc. became part of the roboticist's glossary [14]. This became even more common when multi–sensor systems appeared, and when robot swarms were proposed as a solution to complex robotic problems [8, 64].

Furthermore, many researchers are moving from the old conception of considering the sensing system of a robot as a concentration of devices, all physically located in the same place (i.e., on the robot's body) to a distributed sensing paradigm. Usually, robots are employed in an environment that, for several other reasons, is already equipped with a communication network and sensors (e.g., the cameras of a surveillance

[1] Due to the focus of this research, we will report only the systems that use the publish/subscribe model for information exchange.

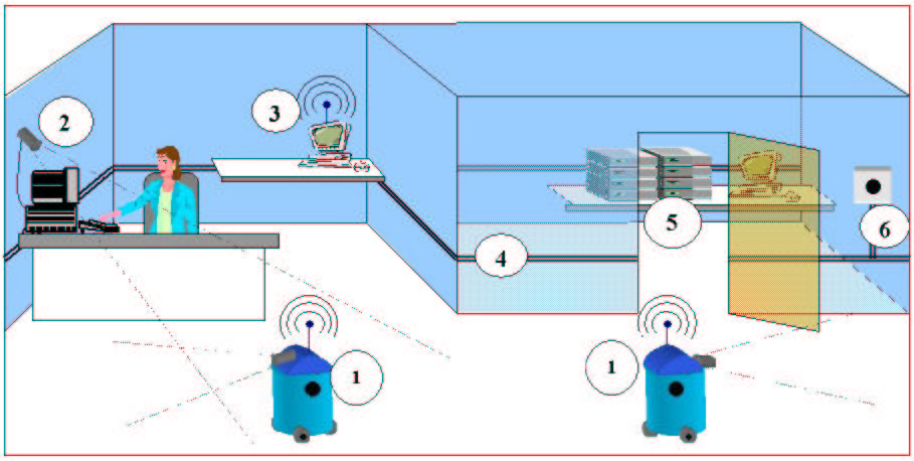

Figure 1: Possible application scenario in a department environment with two mobile robots: (1) Robots, (2) MS Windows desktop PC with surveillance webcam, (3) Linux workstation for robot supervision and gateway between ethernet and wireless network, (4) LAN, (5) Gateway to the Internet, (6) Acoustic Sensor available on the LAN

system), and they have to interact with other robots (co)operating in the same area (e.g., office, hospitals, etc.).

In Figure 1 we depict an example of the scenarios we are interested in. In such environment, two robots, provided with radio links, interact with several other computers with different operating systems connected through a LAN, and with two different "smart" sensors one acoustic and the other visual. In this scenario, it is also possible to include some "almost sensorless" robots with very small onboard computing power, remotely connected and capable of taking advantage of the whole power of the community.

While several authors [4, 43] have pointed out the analogy between robot sets and live beings (e.g., simple cells, swarms of insects, colonies), in [15] we proposed to consider each single robot as a *community of devices*, deeply connected with the other communities (i.e., other robots) and the environment in which they are embedded.

Considering robots as communities of devices that are not necessarily attached to the same physical body allows many new interesting functions to be implemented. This affects sensors (i.e., a robot can take advantage of surveillance cameras that observe the environment where it is working as a means for self–localization), but can also affect processing power, allowing processing units "outside" the robot body to co–operate, offering their services when and where required. From this assumption, it immediately follows that considering a group of electronic devices as a community requires a common communication middleware as a mean for integration. However, if we consider the way devices interact with each–other (e.g., serial/parallel connections, networking physical media, wireless and satellite

links, visual and acoustic signalling, etc.), it would be really inefficient to force all the communication flows to use the same protocol, or the same transport means. This holds true especially in the autonomous robotic field, where sensors and actuators may be interconnected in different ways.

3 Communication in Robotics

Given the scenario proposed in the previous section, it is clear how one of the key issues in distributed robotics, where several physically distributed devices (i.e., sensors, actuators, teams of mobiles robots, man–machine interfaces, etc.) interact to achieve a common task in an effective way, is data exchange. In the following we report some of the main characteristics of data flows in this scenario in order to depict the most suitable communication model for device communities.

- Computations are mostly event–driven, that is, execution is triggered by the notification of some event happening in a reactive way. For instance, a collision–avoidance plan may need to be executed as soon as a new obstacle is detected, but only if such an obstacle is detected.
- A significant portion of the event flow has a limited lifetime and the requirements in term of reliability of the communication may depend on its source or on its content. Considerable overhead can be avoided using a data transfer paradigm that exploits these facts. For instance, with many sensor readings, a limited loss of events is often not critical when event notification is an idempotent operation and when new updates simply replace old values.
- There are often multiple sources for the same type of information and similarly there can be many consumers. For instance, a robot command might be generated by an autonomous control module as well as by a tele–operation module. Similarly, a physical robot and a simulator may be both consumers of "command–data".
- The network of data producers and consumers may not be known in advance and may change dynamically as well as the kind of information requested/produced in the network. For instance, force–level measurements, normally used only by a low–level controller, may be required for a limited amount of time by a monitoring device. Moreover, a new sensor could be added at any time or could produce data only if active sensing is required.
- Most data flow may be anonymous. For instance, producers of the sensorial information can usually be unaware of who is reading them. Again, considerable overhead can be avoided using a data transfer paradigm that exploits these characteristics.
- Data exchange is often time–critical. For control purposes data must be transferred from source to destination with minimum delay or, at least, with a certain predictability (i.e., guaranteed delivery, deadline exceptions notifications, best effort, etc.)

A communication middleware for robotics applications should support in a natural way the implementation of these types of information flow. In addition, the communication system should be able to operate on readily available hardware and be portable across several architectures and operating systems.

4 Frameworks for Information Exchange

A communications framework provides a model of information exchange to the applications that communicate using it. The success of a specific framework depends on how natural and efficient is the model it provides for a given application [49]. While each framework has its own trade–off characteristics, and cannot be appropriate for all applications, several frameworks have been proposed for distributed information exchange: shared memory, request/response (a.k.a. Client/Server), and publish/subscribe. In the next paragraph we'll briefly describe these paradigms underlining their pros and cons from the point of view of a program of a robotics application.

4.1 The Shared Memory Model

This model presents the illusion that there is a shared global memory where data used by different applications is stored. Communication occurs by writing into and reading from this global memory (see Figure 2). In the shared memory model any change must be seen simultaneously and consistently by all the peers. This is a powerful and familiar model because it allows the programmers to design the applications as if it was not distributed. Algorithms and programs developed for non–distributed systems can therefore be ported to a distributed environment with a limited amount of changes.

The attractiveness of this model has generated substantial research [6, 42, 65]. The appearance of a true physical shared memory on a distributed system can only be achieved by the operating system, and several research operating systems (e.g., the V Kernel [16], Amoeba [70]) provide these facilities. A similar, but more easy to implement, model is the shared data–object model. In this model, shared data is encapsulated in objects or other structures and accessed through atomic operations on those objects. This abstraction can be provided by the operating system as in Clouds [21] and Chorus [61], by a language with a run–time system as in Linda [1], or by dedicated environments such as Distributed Blackboards [36].

Despite its ease of use, the shared memory model is not a natural paradigm for many applications, especially in the robotic field, since it is difficult to develop event–driven applications, react to data changes, and synchronize or wait for those changes using such model. The requirement that all participants maintain a consistent view of memory makes memory updates expensive and unpredictable in their delay, and such need for synchronization often implies a centralized system thus reducing the scalability of this model.

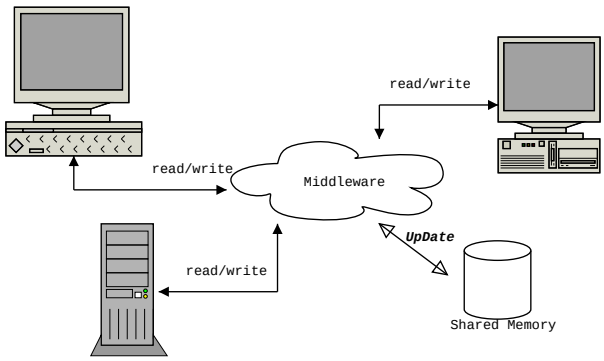

Figure 2: The Shared Memory Model

4.2 The Request/Response Model

In the request/response model, information exchange occurs by issuing requests and waiting for corresponding responses; examples of this kind of exchanges are client/server queries and remote method invocation.

The client/server model is asymmetric and intended for master/slave type queries (see Figure 3). A client (master) issues a request and waits for the response from the server. The server (slave) is normally idle waiting for requests from the clients. This model is very popular and has been used to implement file systems [69], window systems [50], and distributed databases [22]. Suites of protocols and supporting software have been developed which realize such model, notably External Data Representation (XDR) [45] and Remote Procedure Calls (RPC) [46]. However the client/server model has several drawbacks for robotics applications. The client blocks while waiting for the response and cannot issue concurrent requests or execute during the elapsing time. Two messages are required to receive a certain data (i.e., the request and the response) increasing the latency and diminishing throughput of the information flow. Finally, it is also difficult for a server to notify an event to a client. These shortcomings have been recognized and addressed in other fields by augmenting the basic client/server model with the introduction of Asynchronous RPC [3] and other mechanisms [5].

The remote methods invocation model extends object–oriented programming to a distributed environment. Objects interact by exchanging messages unaware of their physical location. In this exchange, the object–to–object relationship is symmetrical (the sender and receiver roles are specific to each invocation). To locate an object or a service, a run–time system (or the operating system) must maintain a directory of the available services, objects and/or interfaces. Several standards such as Sun's RMI [47], Object Management Group's CORBA [34], and Microsoft's DCOM [23] are in use to define standard object interfaces, data–representations, and invocation methods in order to communicate between different machines and applications from multiple vendors.

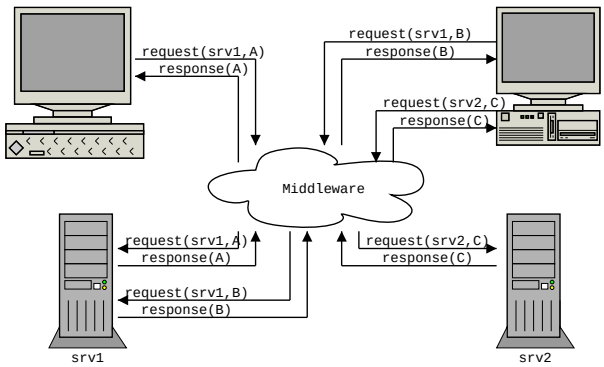

Figure 3: The Request/Respense Model

Despite its power, the request/response model realizes the concept of a synchronous exchange where one request is followed by its response and there is a strong coupling between the client and the service provider. For this reason, this model is inappropriate for applications where most of the event flow is one–way, one–to–many, computation is event–driven, and there is a dynamic structure of producers and consumers of information.

4.3 The Publish/Subscribe Model

In this model, data providers publish data and consumers subscribe to the information they require, receiving updates from the providers as soon as this information has been published. This paradigm is based on the notion of *event*. Components interested in some class of events "subscribe" expressing their interests. Components providing information "publish" it to the rest of the system as *event notification*. This model introduces a decoupling between producers and consumers through the fact that publishers and subscribers do not know each other: publish/subscribe operations, and in particular the delivery of an event notification to all the subscribers, are mediated by a component, the *event dispatcher*, whose architecture can be either centralized or distributed (see Figure 4).

This model offers significant advantages in situations where data transferred corresponds essentially to time–changing values of an otherwise continuous signal (e.g., sensed data, control signals, etc.). A single subscription replaces a continuous stream of requests and the data is transferred with minimum delay since the exchange is one way and asynchronous. The publish/subscribe model is also notification–based and this is very beneficial when a system needs to monitor a great number of perhaps infrequent events, while in a client/server model or shared memory model, the monitoring process must continuously poll for possible changes.

One–to–many communications are supported by the publish/subscribe model in a natural way, and its implementation can often take advantage of multicast and broadcast mechanisms at the network level to improve the efficiency of event notification. Publish/subscribe middlewares have

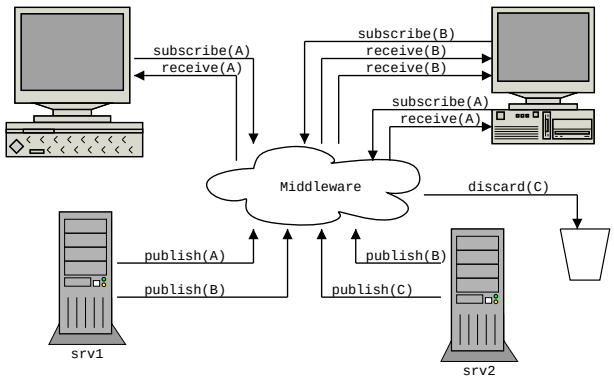

Figure 4: The Publish/Subscribe Model

recently been used for financial applications [51], command and control systems [11, 26, 75], desktop tool communications [39], and task allocation among groups of heterogeneous agents [29].

5 Robotics Middleware Characteristics

In section 2 and 3 we underlined the main characteristics of communication in robotics applications, and it is clear that, in such applications, a publish/subscribe middleware implements the most natural communication model. In robotics, the communication happens between a restricted number of devices (e.g., up to hundreds), highly heterogeneous from the hardware/software/protocol point of view, and with different quality of service or real–time requirements. Thus a publish/subscribe middleware suitable for this domain should implement inter/intra process data exchange between heterogeneous systems connected using heterogeneous networks with the following characteristics:

- *Community Support*: since the community is composed of several devices, the middleware has to support the establishment of communication in a dynamic environment with different degrees of reconfigurability and eventually supporting additional services like authentication and security.
- *Heterogeneity Support*: addressing heterogeneity in platforms, protocols, operating systems, etc.
- *Content Support*: this aspect is related to the richness of the communication support, the supported message format, and the type provided checking.
- *Subscription Support*: subscriptions are the means for filtering and routing information in an effective way, saving bandwidth and increasing efficiency. Different publish/subscribe middlewares differ in the expressiveness of the language used to subscribe to events.

- *Quality of Service and Real–Time Support*: whether and when a message is delivered might be a critical issue in many situations; the aspect of supporting quality of service or real–time delivery might be a key factor in choosing the integration technology for the specific robotic application.

5.1 Community Support

A key point for a middleware for robotics applications is the dynamics allowed in structuring the community. It might be based on static mechanisms where all the peers are known from the very beginning or provide mechanisms to dynamically change the structure of the community where nodes can come and go at any moment and the system allows them to connect and disconnect in a robust way. Different toolkits might support different levels of system reconfigurability. Some toolkits will allow components to come and go at run–time, independently of where the components are physically located, other toolkits may expect a more stable system, where components and component topology are specified in advance. The *reconfigurability* of a system can be classified as:

1. *Absent Reconfigurability*: once the network of clients and dispatchers is formed it is no longer possible to add new producers or consumers; the only dynamics in this kind of system is the one provided by the subscription mechanism that allows consumers to change their interests about the data.
2. *Client Reconfigurability*: in this kind of systems it is possible to add or remove at any moment a client from the dispatching structure while this structure is fixed at run–time.
3. *Dispatching Reconfigurability*: in this case the dispatching structure may change during execution and also dispatchers are allowed to come and go.

The reconfigurability of the system is clearly related to the architecture chosen for the dispatching. In the following we present some of the possible choices:

1. *Direct Connection*: No explicit event dispatcher exists. Events are dispatched by the sources to the interested parties that are directly connected via one–to–one network channels to the sources themselves. In other words, the sources act as event dispatchers of the events they want to notify.
2. *Multicast*: This is a special case of direct connection in which sources exploit multicast mechanisms to deliver the events to the destinations.
3. *Centralized*: A single event dispatcher performs the dispatching of events. Sources and recipients are connected to the dispatcher, which collects subscriptions and routes events based on them.
4. *Distributed*: A number of interconnected dispatching servers cooperate to deliver events. Each source and recipient connects to a dispatching server which collects subscriptions and, in cooperation with other dispatching servers, routes events based on subscriptions.

5. *Mixed*: The mixed approach exploits multicast channels to deliver events within a LAN and a centralized or distributed event dispatcher to forward them across different LANs. This way it is possible to take advantage of the multicasting mechanisms still overcoming their limitations in WAN communications.

In robotics applications, it is often important to know the identity of the sender of a certain message, so the integration infrastructure should give some primitive to access this kind of information. Notice how this feature can be useful in applications where trusted transactions are required (i.e., trading agents); if sender authentication is left to the application level, it becomes suddenly complicated to deploy "secure" systems. The *identification* support provided can fall in one of the following categories:

1. *Absent Identification*: it is not possible from the message to derive any information about the sender.
2. *Machine Identification*: the middleware provides some primitives to get information about the identifier of the machine hosting the process/thread that originated the message.
3. *Sender Identification*: each of the producers/consumers is identified in a unique way and the middleware provides information about the identity of the producer of some message.

5.2 Heterogeneity Support

The usual context of message oriented middleware is a wide distributed system up to the internet scale [60] exploiting homogeneous network support through the TCP/IP protocol. In robotics, information flow can follow different paths characterized by heterogeneous protocols (e.g., TCP, UDP, CANbus, Serial Link, USB, Shared Memory, I^2C, etc.), between different devices (e.g., DSP, Workstations, Microcontrollers, etc.) running different operative systems (e.g., UNIX, RTOS, Linux, MS Windows, Embedded Operative Systems, etc.). A key issue for a communication toolkit is whether or not it will run on the hardware platforms that are available for the specific application and how well it makes use of the data transport mechanisms available on that platforms.

A middleware for robotics should be able to provide a high level communication API, possibly for different programming languages, with the ability of connecting heterogenous devices independently from the connecting physical channel. Thus, heterogeneity can be supported in four complementary ways:

1. *Platform Support*: devices can use different hardware to realize their function (e.g., Motorola, x86, DSP), thus they might have different characteristics and they should be able to communicate in a transparent way using the middleware
2. *Protocol/Channel Support*: the support for a specific architecture does not imply the efficient use of that. For example, if a toolkit allows processes on the same real–time machine to communicate, will it allow them to use shared memory? Will the toolkit discriminate

between POSIX sockets and TCP/IP sockets for guaranteed communications under UNIX–like operating systems? Can the toolkit use the UDP non–guaranteed protocol when appropriate? Could the toolkit support different communication channels (e.g., USB, serial bus, etc.) using the appropriate protocols?

3. *Operating System Support*: different parts of the same distributed application can be written for different operating systems in order to gain as much as possible from the specific features of each of them.
4. *Programming Language Support*: the use of different programming languages is diffused also in robotics (e.g., LISP for planning modules, C/C++ for control systems, Java for GUIs, etc.) so the middleware should provide this kind of heterogeneity support as well.

5.3 Content Support

An important feature of publish/subscribe systems is the type of messages they support, their format, and their representation. If the granularity of messages is too small, many messages have to be generated since each of them has a poor or limited meaning. This choice might significantly complicate the programming activity, reduce the performance of the system, and make it difficult to test and monitor the system. On the other hand, an eccessive complexity of messages might generate an unjustified overhead and might not be possible on network with limited capabilities. There is no universal solution to this design problem. It is the designer's responsibility to evaluate the trade–off and select the most suitable solution based on the constraints and requirements of the problem being addressed. The middleware should support designers in this decision by providing suitable messages models and composition languages:

1. *Tuple–based*: messages are defined as a sequence of strings.
2. *Record–based*: messages are defined as sets of typed fields characterized by a name and a value. Note that within this category, different event–based infrastructures could be further classified depending on the richness of the type system they offer.
3. *Object–based*: messages have both a state and a set of methods. In this case, an event is created through the invocation of the corresponding class constructor.

A critical issue in the development of a publish/subscribe middleware is the selection of the type system to create and distribute events. There are at least two basic alternatives:

1. *Global Types*: all the components in the infrastructure see and use the same set of types, this makes it possible to perform significant consistency and compatibility checks on the information being exchanged.
2. *No Explicit Typing*: Each constituent of the infrastructure can produce events without referring to a specific type. A subscriber is supposed to know the structure of the event being received, while this structure is completely hidden to the middleware.

A global type system may result desirable, in general, but experiences of the past years have demonstrated that it is extremely difficult to define global types on a large scale system, where it is necessary to cross company boundaries and involve independent users. The issue is not merely technical, it is also related to scalability and ease of operation. This is a relevant issue on Internet scale applications since they are inherently decentralized and based on autonomous and independent operators. In this case, type compatibility cannot be enforced by an explicit, network–wide (and, thus, conceptually centralized) type system; rather, it is the result of a set of simple and voluntarily conventions[2].

5.4 Subscriptions Support

The key aspect of the pubblish/subscribe model is the capability of filtering and routing information using the subscription mechanism: events are delivered only if requested and interests can be changed dynamically. The support to subscription specification in a publish/subscribe middleware can be classified as:

1. *Content–free*: Subscription is accomplished by specifying a channel. The subscriber receives all the messages that are posted to the channel.
2. *Subject–based*: Each event is labeled by a subject. Subscriptions are specified by indicating the subject of interest. Notice that the subject–based approach is a variation of the content–free concept. We introduce this distinction because it reflects a market trend. In practice, both subjects and channels can be used to represent the "key" of the events the subscriber wants to receive. Both approaches enable the exploitation of multicast communication infrastructures and guarantee the high level of performance needed in several critical application domains (e.g., thousands events per second in stock market applications). The drawback is in the limited freedom that subscribers have in expressing the event categories they want to receive.
3. *Content–based*: Subscriptions are specified as expressions evaluated over the event contents. Within the content–based category, subscription language constructs can be further classified depending on the expressive power they provide to specify predicates:
   (a) *Disjoint elementary expressions*: it is possible to specify the value or the range of values for each event parameter[3].
   (b) *Compound expressions*: it is possible to compare different event parameters.
   (c) *Regular expressions*: the subscription request is expressed using regular expression.

[2]MIME is a typical example of such an approach. It does not define the structure of the different files being exchanged over the Internet. MIME is used just to label the documents being exchanged so that each party can access them according to agreed procedures and tools (e.g., a "text" file is what you can usually open with an editor).

[3]In some middlewares the sender id or the id of the machine originating the event can be part of the event parameters.

(d) *Event combination*: it is possible to define subscription expressions that require the combined occurrence of more than one event.

5.5 Quality of Service Support

In robotic applications, information delivery might require different quality of service. A middleware supporting real–time applications, should allow the producer/consumer to specify the duration of the information it is publishing or requesting some exception handling mechanism. Obviously, quality of service is related to the the physical channel used for the delivery or the protocol used (i.e., TCP, UDP, EIEP, RTPS, etc.):

1. *Delivery Support*: the event flows can use different channels with different performance or delivery characteristics, in such cases the middleware can give an additional support by providing different qualities of service:
   (a) *Best Effort*: the system makes an attempt to forward all the events in the fastest possible way. However, if the network becomes overrun or routes change, events can be lost, delayed, or delivered out of order.
   (b) *Guaranteed Delivery*: the middleware provides a less efficient delivery service, but guarantees to deliver all the events or notifies the application if an error happened.
   (c) *Ordered Delivery*: in this case, not only the delivery is guaranteed, but also some ordering properties like absolute time ordering, machine time ordering, causality, etc. are satisfied.
2. *Priority Support*: it might be possible to specify a priority for an event, in this case the notification of such an event is prioritized with respect to other events even if they were generated in advance.
3. *Mobility Support*: event–based infrastructures can provide mobility features allowing to move components from one side to another. In this case it should be possible to temporarily disconnect clients from the dispatching system, save their state, move them while storing the events they were subscribed to, and reconnect them.
4. *Real–Time Support*: in many applications it might be useful to give also upper/lower bounds for the delivery of the message and this might be decided at subscription time and at publish time. This is required by the fact that, in robotics applications, a communication middleware is embedded in a more complex system for task coordination and distributed control which might require such kind of service.

6 Middleware Classification

In the following we present a compared analysis of the publish/subscribe middlewares currently used in autonomous robotics. The characteristics underlined in the previous section are used as a classification grid for our analisys and the results are reported in Tables 1, 2, 3, and 4.

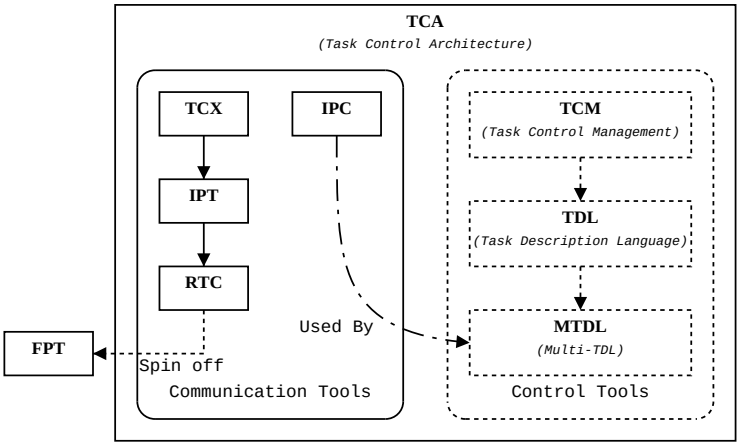

Figure 5: A schema of the Task Control Architecture and its successors

6.1 TCA and TCX

In 1988, began the work on the Task Control Architecture (TCA) [66, 67] at Carnegie Mellon University. As described in Figure 5, TCA includes capabilities for both inter–process communications and task–level control. In 1990, Chris Fedor (who helped to implement TCA) developed TCX [25], which is basically the communication infrastructure of TCA[4]. TCX was used in a number of projects, most prominently Dante [7] and the BeeSoft mobile robot software [17]. From Figure 5 it is possible to notice that TCX originated other communication tools and also a commercially available package called Fourth Planet Communication (FPT) [27] also applied in robotics research at Carnegie Mellon University [28].

Community Support A TCX system comprises one or more "modules", running processes or tasks, plus the "tcx" communications server, which coordinates module interprocess communications. The server attempts to maintain the network of communicating modules despite new modules entering and other modules leaving (including leaving due to a failure or crash). Modules started before the communications server will wait for the server before finishing initialization. New modules may enter the network and change the flow of message traffic without requiring that existing modules to be recompiled. TCX makes use of message queues to provide some mixture of both direct and indirect message based commu-

[4]The initial versions of TCX followed the client/server paradigm: every process in the system was connected to the server. When a process would like to send data to another process, it would send the data to the server, and the server would in turn send the data to the appropriate process. This method, although conceptually much easier to implement, was inefficient because the data had to make two hops, as opposed to only one hop if the data were sent directly to the receiving process. Later versions of TCX allowed for "point–to–point" connections requiring only one hop when sending data from one process to another process.

nication. When each module initializes, it establishes a communications link to the TCX server. Communication with the server is identical to communication with another module. The communication server has a message queue which is known to modules within the system and each module initializes a default message queue whose name is the same as the module name. The default message queues can be used by modules as well as the communication server for sending and receiving information. Modules are initialized with an internal message queue whose name is the same name as the module to simplify module identification as well as to provide a convenient reply path for message traffic.

Heterogeneity Support TCX is written in C and is distributed for Sun (both SunOS and Mach), SGI, Intel 386 and 486 running Mach, PMAX running Mach, NeXT machines, Motorola 68020, 68030, and 68040 processors running VxWorks. The inter–process communications uses socket–based communication with TCP/IP and supports both publish/subscribe and client/server modes of message passing. TCX also supports automatic marshalling and un–marshalling of data based on a flexible format definition language. The language supports efficient transmission of all C primitive data types, as well as structures, pointers, and fixed and variable length arrays.

Content Support Data formats provide a method for describing the structure of a user data type so that data can move between separate processes in a transparent manner. The message layer of TCX provides routines for registering data formats which are used to encode the data into a linear stream of bytes, transfer the byte stream between processes, and reassemble the data at the receiving end. TCX supports a wide class of different data formats. Primitives include data types such as integer, float, double and string as well as special data types for variable dimension arrays of floats, and for transferring parts of these arrays.

Subscription Support Patterns of message traffic and actual communication links are determined by the message queues. Message queues can be created by processes as either being local to that process or residing in the communication server process. Communication links are established by asking for a reference for a particular message queue. A TCX message queue provides both a repository for messages as well as a named mailbox for communication. Messages are sent to and read from message queues created by processes. Messages themselves can be sent to specific message queues or delivered to one or more message queues which have been defined for that message. Similarly, messages can be received from a specific message queue or set of message queues, or a message can be received with the sending processes identified.

Quality of Service When receiving messages a module needs to be able to specify the number of messages it is willing to save. For example, if a module has been ignoring incoming sensor information because it has been doing something else, when it returns to grab the next sensor message

it wishes to receive the most recent sensor message without examine each sensor message received. If a module wishes to examine only the last n sensor info messages it is able to set that. This is accomplished in TCX by specifying a limit to the number of messages a particular message queue will store. Messages whose information gets old over time should be registered to message queues with a limited capacity. Newer messages will overwrite older ones. Additional routines provide more control over what to do when a message queue is full. TCX provides limited ability to send high priority information by allowing a message to be inserted at the top of a particular message queue. In this manner the message will arrive via the same communications link or implementation as all other messages but will be processed ahead of those messages currently in the message queue. Another alternative would be to create a special message queue for high priority messages.

6.2 IPT (Inter Process Toolkit)

The Inter Process Toolkit (IPT) [31] was developed by Jay Godwy at Carnegie Mellon University for use in the Unmanned Ground Vehicle program [71]; it started from TCX and, for the most part, the functionality of TCX can be considered a subset of the functionality of IPT.

Community Support Modules use IPT to establish connections between themselves in order to send and receive messages. These connections can be considered as direct lines of communications setting up point to point links between modules without going through any center. IPT, as its precursor TCX, has the same idea of a central communications process. These central processes are not the means by which all modules communicate, but they are the means by which all modules initiate communications. Once modules are connected, the server doesnt take up any more CPU cycles. It does not die, because IPT is a dynamic system. Modules are allowed to connect and disconnect throughout the lifetime of the system, and the server needs to be around in order to make and break these connections in an orderly fashion. There is provision in IPT for multiple IPT servers, with each IPT server servicing a domain. The inter–server communications was optimized for low bandwidth links, since the design was to have an IPT server in each robotic vehicle with low–bandwidth wireless links between the vehicles. IPT provides the clients with the ability to detect when modules connect and disconnect, so there can be some measure of dynamic structure to a system, but typical use is to have a fairly stable structure for the duration of a single test.

Heterogeneity Support IPT has been ported to many non–real–time, UNIX based, operating systems, such as SGI's IRIX, various flavors of SunOs, Solaris, and Linux. In addition, there has been a relatively untested porting to some real time systems such as VxWorks and LynxOS. IPT is written in C++ and provides C cover functions. Many of the connections between IPT modules are instances of the class TCPConnection. A TCPConnection uses TCP/IP socket based communications to send

and receive messages primitives. Since, a TCPConnection is a sub–class of IPConnection, it is possible to implement shared memory connections on real–time machines which will be vastly more efficient than TCP connections by creating another sub–class of IPConnection which implements the primitives in a different way. TCPConnections and SharedMemConnections can coexist in the same process by taking advantage of some of the basic properties of C++. IPT uses UNIX sockets instead of TCP/IP sockets for communications between modules running on the same UNIX machine, and IPT reduces the amount of data allocation and copying to a minimum[5].

Content Support As in TCX, each message has an instance number and a type. The instance number can be used to keep track of the sequencing of messages, and the type contains information about how the message is formatted. Message data can be formatted to allow unpacking into C or C++ structures. Messages can be handled by user defined handling routines or by searching through a message queue. A message type consists of a several basic components. The most basic is the message type name, which is an arbitrary string which should be unique across the system. This message type name will be translated into a message type ID by the central server, and the message ID is an integral part of the message type as well. A message type can also have a message format specifier attached to it. This format specifier starts out as a string which directs how to unpack the raw message data into a C or C++ structure. The code for doing this formatting was mostly inherited from TCX

Subscription Support The first step in using message types is to register them. The registration process simply checks with the central server to get a consistent mapping between message names and message IDs. No other information is sent to the server and other message type registration, such as adding handler and destination information is done with later routines. IPT gives a mechanism whereby a module can declare that it is a publisher for a type. Subscriber modules can connect to this module and subscribe to the type. Then, when the publisher wants to update the data, it can publish it to all the modules that have registered as needing it. IPT also makes sure these publisher/subscriber relationships last when publishing and subscribing modules die.

Quality of Service IPT is designed to be fairly flexible in managing its connections. Each client provides a name for itself and an ordered list of ways in which it can be reached, or routes, and when another client requests a connection by name, the server attempts to connect the two clients together with the dynamically determined *best* connection method. For example, the server would recognize from two modules routing lists that when they are running on the same UNIX machine, they can be connected together with UNIX domain sockets. When the same two modules are running on different UNIX machines, the routing is through TCP/IP sockets. IPT provides a mechanism to alleviate the problem of backlogged

[5]IPT has been clocked at 5 times faster than TCX for large messages on a single machine.

messages by minimizing the message queuing and data processing that the consumer process would otherwise incur. IPT provides a way to declare a message type a pigeon holed message type. Declaring a message type a pigeon hole means that if a handler is called or the message queues are searched for a message of this type you are guaranteed to get the most recent message of that type from a particular connection without any additional user processing.

6.3 RTC (Real–Time Communication)

In the late 1990s, Jorgen Pedersen developed the RTC (Real–Time Communications) package [55] for robotic projects at the National Robotics Engineering Consortium (NREC) at Carnegie Mellon University. RTC extended IPT featuring very efficient message passing for real–time applications including capabilities for shared memory communications.

Community Support RTC allows processes to form or destroy connections with each other asynchronously; each process essentially has an address at which it can be found by other processes. This address consists of an IP address and a port number. To hide low level information, like addresses and port numbers, a server is employed for creating a flexible communication system. A publisher maintains a list of all those processes which have subscribed to the publishers data. This list is initially empty. The publisher has certain data that other processes want. If these other processes need this data, the can make explicit connections to the publisher.

Heterogeneity Support RTC was originally designed for VxWorks, but should be possible to use it under many different computer platforms (e.g. x86, 68k, MIPS, SPARC,etc.) as for IPT. RTC was designed such that every process must form a TCP connection to every other process with which it wishes to communicate. This is required because, even with shared memory, low level information about other processes needs to be acquired. Since the mechanism for acquiring this data (through the TCP connection to the server) already exists, it is called upon again when making shared memory links between processes. The second reason for requiring every process to form a TCP connection is that if shared memory is not available, data which was to be sent through shared memory link can still be sent through the TCP connection. RTC was designed to run simultaneously under multiple computer architectures. Consequently, RTC supports all the different byte–ordering and alignment models used by various architectures. RTC was also designed to support different operating systems as well as the architectural differences. RTC does not perform any conversions to a standard network form. Rather RTC follows the rule that it is the responsibility of the sending process to convert the data into the correct representation for the receiving process. This methodology is possible because each process has knowledge about the architectures of the other processes.

Content Support While it uses TCX–type format strings to describe data structures for automatic marshalling and unmarshalling, it does not allow passing of data structures with pointers or variable–length arrays. Once message format strings are generated, the built–in parser of the RTC protocol can parse these strings in order to know exactly how to pack and unpack the data associated with each message. Parsing of messages only needs to be done once, during the initialization phase of a process. This speeds up transmission time because RTC looks up the pertinent information, rather than recalculating it each time a message is to be sent between processes.

Subscription Support As TCX and IPT, RTC is a queue based system. Before a process can send or receive a message, the message must first be registered and for each message type, the initial queue length allocated is five messages long. Note that efficient systems would rarely allow messages to back up in the queue to this length. But it is conceivable that a process may, sometimes, accumulate messages in their respective queues. If this happens, RTC will grow the message queues to the required length until the process is able to handle messages again.

Quality of Service As in TCX and IPT, a process can limit the length of any message queue to one message. This capability guarantees that the message in that queue is always the most recent message received. That is, with a queue length of one message, that message will continually be overwritten as new messages are read off the TCP buffer. In addition, to save memory, a process can eliminate any message queue. This is useful if the process is merely a sender of a message. Since the process only sends the message, never receiving it, there is no need for a queue of that message type.

6.4 IPC (Inter Process Communiation)

In 1994, Reid Simmons developed the IPC (Inter–Process Communications) [68] package for the NASA DS1 New Millennium mission. While, in the end, IPC was not actually used on DS1, it has since been used in numerous projects at CMU, NASA, DARPA, and elsewhere. As with TCX, IPC features efficient transmission of general C data types, anonymous publish/subscribe and client/server capabilities, and automatic marshalling and unmarshalling. IPC features both centrally routed messages and peer–to–peer communications. It has support for timers and much more flexibility for passing data messages, which trade off flexibility for efficiency.

Community Support An IPC–based system consists of a central server and any number of application–specific processes. The central server is a repository for system–wide information (such as defined message names), and routes messages and logs message traffic. IPC also supports a version of the client/server paradigm: sending a directed response to a query. Both blocking and non–blocking versions of this facility are

provided. Before starting any modules, a program named "central" must first be started. The most basic service that the central server provides is message passing. A message sent from any module connected to the server will be forwarded by the server to the module containing the handler for the message (optionally, messages my be sent directly between application modules). More than one server can run on the same machine, using separate communication ports for each server. Having multiple servers is especially useful for software development when independent developers must run their IPC servers on the same machine. There are occasions when a module needs to connect to more than one central server. For instance, if you have two robots with a relatively slow radio link connecting them, it may be desirable for reasons of bandwidth and latency to have a central server residing on each robot. However, if one robot wants to send a message to the other robot, it needs to (temporarily) access the other robot' s IPC subnetwork.

Heterogeneity Support IPC libraries exist for C, C++, and Allegro Common Lisp. IPC for Java is now available (although currently it has been tested only under Linux). IPC currently runs on the following architectures and operating systems: Sparc (running SunOS and Solaris), Intel processors (running Linux, Windows NT, Windows 98), 680xx processors (running VxWorks), SGI (running IRIX), and MacIntosh (running Mac OS). IPC is actively being supported and extended; a recent addition is a means of automatically generating format strings from XDR data structure definitions. These facility enable programs to transparently send a wide variety of data formats, including structures that include pointers (strings, variable length arrays, linked lists, etc.) to machines with possibly different byte orderings and packing schemes.

Content Support IPC can send raw byte arrays between processes, but it also provides a powerful data–marshalling facility that enables it to pass data transparently between processes, even if the hosts have different byte order or different alignment. A programmer provides such a structural specification (called a "format string") in parameters to message definition routines. Once this is done, IPC can know how to convert the data structure to a byte stream and how to reconstruct it in the receiving module. As in standard programming languages, IPC format strings are composed of primitive data type specifiers and composite specifiers that enable users to define more complex data types.

Subscription Support Modules must first connect to the IPC central server, and then they can define messages, together with a description of their data formats. Modules that want to handle messages must indicate their interest specifying the message name. Definitions and subscriptions can be done in any order, including subscribing to a message before it is defined. When messages have been thus registered, modules can publish messages.

Quality of Service IPC provides the same QoS provided by IPT, but it can also be used to invoke a user–specified function at a specific time, or with a specified frequency. These timer capabilities enable a module to perform time–critical actions, or to dispatch events at specific times. While these functions can be used for time–dependent operations, note that they are not truly interrupt driven, they will be invoked only when the module is within some IPC function that is listening for messages. If the specified time passes while the module is doing some other computation (or is swapped out), the timer function will be invoked at the next available opportunity. The IPC package supports also message logging and message data logging. The Comview tool can be used to visualize and analyze patterns of communication.

6.5 ETHNOS (Expert Tribe in a Hybrid Network Operating System)

The ETHNOS (Expert Tribe in a Hybrid Network Operating System) architecture was developed by Maurizio Piaggio at University of Genoa [58, 57, 56]. It was used in the ART RoboCup Team and service applications as well. ETHNOS provides support from two main point of views: software integration and real–time execution.

Community Support An ETHNOS application can be divided on two main entities: experts and kernels. An expert is a program that executes in a periodic fashion its functions. A kernel is a program with two main goals. It interfaces with the system scheduler to modify experts' priorities and it realizes the communication between experts. It is possible to implement a distributed application with many kernels each of them on different computers and providing services to many experts. When an expert subscribes to a message on a different machine, the kernel forwards the subscription to the other machine and provides locally the messages to the expert's local message list. There are two paradigms for kernel interaction. In the *client/server* structure for the kernels, all the message exchange passes through a single central server; if this central server crashes the system doesn't work any more, but this setting allows for executing different kernels on the same machine. Kernels can be deployed also as a *broadcast club* and they are identified with the local address IP and a port number. If a Kernel crashes in the broadcast club, the system keep working, but in this case it is not possible to run more than one kernel for each machine.

Heterogeneity Support The API has been written for the C++ language on which the entire system is based. However for communication to external devices (e.g., user consoles) a subset of the API containing the necessary communication services has also been written of Java language allowing an easy development of ETHNOS integrated Java applets. Currently, ETHNOS has been developed in compliance to Posix RT specifications as an add–on to the Linux operating systems. Experts are linux kernel threads, scheduled with the Posix round–robin policy. In the

client/server Kernel architecture the protocol used for message passing is TCP/IP while in the broadcast club ETHNOS uses EEUDP (Ethnos Exstended UDP) an extension of UDP.

Content Support An ETHNOS message is composed by an header and a data field (i.e., a block of byte); in the header are reported the dimension of the data block and the type of the message. Messages are created by providing a pointer to the memory area to transfer and data are moved without any marshalling from one system to the other. In some cases it is possible to share a memory area in order to speed up the data flow and this is supported as an extra feature that does not use the publish/subscribe model (i.e., modules can pass pointers to shared memory areas).

Subscription Support Experts use types to subscribe for different messages; types are integer numbers up to 100. Since a message is simply a sequence of bytes, the application has to know its internal format, and no typing is used.

Quality of Service In ETHNOS, network communication is often wireless (i.e. radio link, wavelan, etc.). Because of interferences or because the robot may have moved to a blind zone, transmission packets are more frequently lost. ETHNOS implements a protocol for this type of applications, called EEUDP (Ethnos Extended UDP). It is based on the UDP and it extends it with the necessary properties. The EEUDP allows the transmission of messages with different priorities. The minimum priority corresponds to the basic UDP (there is no guarantee on the message arrival) and should be used for data of little importance or data that is frequently updated (e.g., the robot position in the environment that is periodically published). The maximum property is similar to TCP because the message is sent until its reception is acknowledged. However, it differs because it does not guarantee that the order of arrival of the different messages is identical to the order in which they have been sent. Different priorities allow the re–transmission of a message until its reception is acknowledged for different time periods (e.g., 5 ms, 10 ms, etc.).

6.6 DCDT (Device Communities Development Toolkit)

DCDT (Device Community Development Toolkit) is a publish/subscribe middleware recently developed by Paolo Meriggi at University of Brescia and Alessandro Mazzini at Politecnico di Milano [15] and it is a project largely inspired from ETHNOS, sharing similar structure and software interface. The main differences regards the aim in using different physical communication means in a transparent way, the scalability (i.e., it is possible to have multiple instance of the main structure on the same machine) the portability (achieved by using standard libraries), and the

"openness" of the software (i.e., it is distributed according to the LGPL licence).

Community Support DCDT is a multithreaded architecture and consists in a main active object, called Agora, hosting and managing various software modules, called Members. Members are basically concurrent programs executed periodically or on the meeting of some requested conditions. It is possible to realize distributed applications running different Agora on different devices/machines, each of them hosting many Members. It is possible to have on the same machine more than one Agora (hosting its own Members), in order to achieve a little scalability or simply to emulate the presence of different robots without the need of actually having them connected and running. Agoras on different machines are able to communicate with each other trhough a particular member, called Finder, which is responsible for finding them dinamically with short messages via multicast.

Heterogeneity Support Similarly to ETHNOS, the API developed is written for the C++ language, which is the language used for the realization of all the systems the software has been actually used for. There is no Java support yet, even though it is one of the programmed extensions like the porting under Microsoft Windows Operating Systems. DCDT is Posix–compliant, and, for stability reasons, the threads lay in user space rather in the kernel one as in the ETHNOS case. The peculiar attitude of DCDT toward different physical communication channels (e.g., RS–232 serial connections, USB, Ethernet or IEEE 802.11b , etc.) is one of the main characteristics of this publish/subscribe middleware.

Content Support DCDT messages are very similar to the ETHNOS ones, with a header and a payload fields. In the header there are contained the unique identification of the message type, the size of the data contained in the payload and some information regarding the producer (e.g., producer id, time of construction, etc.). As in ETHNOS, data are moved from one machine to another without any marshalling. No typing is used and each Member has to know the internal structure of each incoming message.

Subscription Support Members use unique identification numbers to subscribe and unsubscribe messages available throughout the community. The maximum number of different messages types at the present moment is fixed to 128, but it can be extended.

Quality of Service Communications in DCDT are carried out in several ways (RS–232 serial connection, Ethernet, IEEE 802.11b, etc.) and the messages can be shared basically according three modalities: without any guaranty (e.g. UDP), with some retransmissions (e.g. UDP retransmitted, similar to EEUDP in ETHNOS) or with absolute receipt guaranty (e.g. TCP). Although at the present time DCDT only achieves soft–realitime performances, it is planned to integrate it with RTAI [9],

a real–time library which could help scheduling threads in a quasi–hard–realtime fashion, in order to cover even those application which require higher and more strict time constraints.

6.7 NDDS (Network Data Delivery System)

The Network Data Delivery System (NDDS) is a publish/subscribe middleware commercially distributed by Real–Time Innovation (RTI) [37, 38]. It was initially developed by Gerardo Pardo–Castellote in his PhD thesis [52] at Standford University and it has been used in many robotics application [12, 19]. Related to NDDS there is an interesting work done by Pardo–Castellote to standardize the Real–Time Publish/Subscribe (RTPS) protocol for IP–based real–time communication systems [53].

Community Support NDDS's implementation is totally symmetric and quasi–stateless, absence of central servers or privileged nodes. All communicating peers are identical and use data–aging to decay any cached state. All information regarding subscriptions and productions is refreshed periodically. NDDS assumes that its various modules are connected by a network, and all communications is done via UDP. Each processor in an NDDS system runs an NDDS agent which acts as a broker for information types. The NDDS agents are told what other machines are in their *peer group*, and information subscriptions and publishings are transparently transported across the processors in the NDDS peer group. Since subscribed messages are anonymous, i.e., the publisher is unknown, the user will have to encode the message source in the message in order to be able to reply to the appropriate module if necessary. Consumers are notification based. They subscribe to a set of instances (identified by their NDDSname) by providing call–back functions for each instance to which they subscribe. When a data update arrives, the call–back function of every consumer is called with the data–item as a parameter.

Heterogeneity Support NDDS is a commercial product from RTI, source code is not available and there is a monetary cost for non–academic usage. Since it is commercial, it has been ported to a wide variety of operating systems (i.e., VxWorks and popular desktop platforms such as Windows, Solaris, Linux, and HP–UX) and is well supported and integrated with other useful products from the same company. NDDS uses UDP/IP as means of communication. To allow communications between computers with different data formats the External Data Representation (XDR) is used.

Content Support NDDS requires all data instances to be of a known type. NDDS has some built in types (such as strings and arrays) but most data flow consists of user–defined types. Creating a new NDDS type involves binding a new type–name with the functions that will allow NDDS to manipulate instances of that type. NDDS provides a tool that can be used to automatically generate code for user–defined types from the type–specification given in the XDR language.

Subscription Support NDDS identifies data instances by name. The scope of this name extends to all tasks sharing data through NDDS. Two instances with the same NDDS name are viewed by NDDS as different updates of the same data instance and are otherwise indistinguishable to the client.

Quality of Service NDDS allows explicit specification of custom update rates, deadlines and the actions to take if a deadline is missed. Within NDDS all data are time–tagged and decisions are made based on the time when the data was generated, sent and/or received. A producer is characterized by three parameters: production rate, strength and persistence. The strength and persistence parameters are used to resolve multiple–producer conflicts. Producers' data are used while it is the strongest source that has not exceeded its persistence. Typically, a producer that will generate data updates every period T will set its persistence to some value T_p where $T_p > T$. Thus, while that producer is functional, it will take precedence over any producers of less strength. Consumers are characterized by two parameters, the minimum separation and the deadline. These parameters are used to regulate consumer update rates. Consumers are guaranteed updates no sooner than the minimum separation time and no later than the deadline. Typically the minimum separation protects the consumer against producers that are too fast whereas the deadline provides a guaranteed call–back time which can be used to take appropriate action. Although NDDS' basic transport mechanism, UDP, does not guarantee message arrival, NDDS does implement a protocol on top of UDP to ensure delivery of messages when the user desires that feature. By default NDDS provides unreliable, unacknowledged data updates from producers to consumers. A producer may specify any one of its productions to be delivered reliably. Reliable updates are grouped together in special packets that are individually acknowledged. The producer is notified if the update is not acknowledged by at least one consumer after a specified deadline and/or if another reliable production is attempted before the previous one was acknowledged.

6.8 TelRIP (TeleRobotics Interconnection Protocol)

The TeleRobotics Interconnection Protocol is an architecture developed at Rice University by Lawrence Ciscon in 1992 [18] to communicate and share data in distributed telerobotic applications in the University Space Automation and Robotics Consortium.

Community Support In the TelRIP systems, processes exchange data through separate processes called *routers*; each user process has a communication channel to a router. From a user's point of view, processes access the router via a uniform, processor independent, interface. Each processor typically contains a single router that handles all communication between its processes and other processor's routers. Routers also handle adding new and removing old processes as well as various maintenance

tasks. Routers are connected with a fully connected architecture in order to speed up the transferring process along the shortest path. Several properties are attached to a data object these include also the source address, a unique identifier of the module in which it was created.

Heterogeneity Support The TelRIP environment is composed of a library written in C and the byte ordering of the external data representation is the Internet byte order. The implementation was used on different flavors of UNIX machines (Suns, Lynx Unix, Silicon Graphics) and on MS–DOS machines as well. Translation between the internal and the external byte ordering in handled in a transparent way by the environment, allowing user programs written in different languages or running on different processor architectures to use the same data object. Protocols for information exchange uses network sockets, inter–process connections (i.e., pipes) and share memory. However, each library only uses one type of connection at a time; the router has the responsibility for managing a mixture of different kinds of connections simultaneously.

Content Support Information is exchanged as data object with an associated Base Data Type (BDT) that define the content of it. The layout of the data within a data object is described by a C–like language and each data object contains an identifier to specify its BDT. In adition to the BDT identifier and the data itself, a data object, contains additional information about the creation, disposition, and nature of the data. This information may include the source of the data, the time it was created, an intended destination, and a list of other properties that help differentiate a particular data object from others of the same basic type. Each of these properties consists of a property–name/property–value pair.

Subscription Support Processes indicate the type and properties of data they are interested in sending and receiving. Processes merely make the data available, with the data communication environment fulfilling interested recipient processes' requests. It is possible to specify subscription regarding the BDT of the data object, the properties, and the sender.

Quality of Service Each network node contains a dedicated daemon process to serve as a gateway. These daemons are connected to each other using TCP/IP and route all the messages between applications. Message delivery is guaranteed but no priorities or real–time support are given.

6.9 SPLICE (Subscription Paradigm for Logical Interconnection of Concurrent Engines)

The Subscription Paradigm for Logical Interconnection of Concurrent Engines (SPLICE) [10] was developed by Maarten Boasson in 1993 to provide the communications backbone for command and control applications.

Community Support SPLICE applications are independent, autonomous processes totally isolated from each other that interact with the rest of the system through so–called agents. An agent embodies both storage capacity and process facilities for handling all communication needs. The agents in their global communication rely on broadcast, which must be supported by the underlying communication network.

Heterogeneity Support The SPLICE system is written in C and was used on Transputers running Helios and in Sun machines runing SunOS. It uses network sockets over TCP/IP.

Content Support Data is typed and is declared using SPLICE's own Pascal–like language, but the actual data instances (identifiers) produced and consumed by each SPLICE application must be known and specified at compile time using SPLICE's language. An application may specify fields in the data structure as key field in order to improve the efficiency of the filtering process and indexing mechanism.

Subscription Support In SPLICE applications refer to data using unique identifiers. SPLICE provides a mechanism that allows specific fields within a data–type to be designated as keys; consumers can choose to treat updates that differ in their keys as completely different items stored and retrieved independently or just different updates of the same item (i.e., only one the latest copy is stored).

Quality of Service The agent of each the producer guarantees that data are sent to all established consumer agents. SPLICE also contains a mechanism for arbitrating among different producers of the same data.

7 Discussion

The present work is meant to present the publish/subscribe model in a robotics context and for this reason the analysis done in Section 6 is restricted to middleware examples applied in this research field. In software engineering, the publish/subscribe model has been deeply investigated expecially for wide–area event notification and content–based networking [60] [13]. The research in this field has leaded to several publish/subscribe middleware for software integration that could be used in robotics as well if they fulfil application requirements.

The Java Messaging Service (JMS) [48] is an API developed by Sun Microsystems. It aims at representing the standard, common interface for Java messaging products. Sun does not provide any implementation of this interface, and assumes that other tool vendors will adopt it. Several JMS–compliant publish/subscribe systems implement this API adopting a centralized event dispatcher and reliable channels between the dispatcher and its clients.

OMG has defined a standard for the implementation of an event service on top of the CORBA object request broker [33]. In particular, the

standard defines the IDL interfaces for three types of components that are involved in an event–based interaction. These are the *event supplier*, the *event consumer*, and the *event channels*. The CORBA–compliant event channels that are currently available on the market mostly present a centralized architecture. Event channels can be pipelined. This constitutes a sort of distributed event dispatching architecture. However, such distribution is not transparent to application developers since they have to explicitly manage it. In order to enhance the capabilities of the event service, OMG is currently working on specifying the interfaces of a notification service [35].

SmartSockets [20] is a commercial event–based infrastructure developed by Talarian. It provides a rich environment for the development of event–based applications and supports monitoring of the events exchanged among the components of an application. It provides also APIs for synchronous communication and supports fault tolerant connections. Smartsockets supports a record–based event model and predefines a set of commonly used event types. Developers can either create events as instances of these types or define their own application–specific types. The subscription approach adopted by Smartsockets is subject–based and subjects can be organized in hierarchies. The internal structure of the Smartsockets event dispatcher is distributed. Each dispatching server is aware of all the subscriptions that have been issued in some point on the system and is able to dynamically route events based on the cost of network connections and on their load.

These examples (i.e., Java Message System, CORBA Notification Service, and SmartSockets) can be considered as well established general purpose alternatives to the application–specific middlewares presented in Section 6. A possible drawback of these systems is the wide–area target they where designed for and their intrisic trade off between efficiency and generality. For JSM the need for a virtual machine running the server and the use of Java as a uniforming language could affect the efficiency of the system. Available implementations of JMS and CORBA event services use a centralized dispatcher and also this might be seen as a drawback since it presents single failure point. SmartSocket is a commercial product and thus it is not an open standard and would impose to all the module developers the purchase of this software[6].

8 Conclusion and Future Work

In this work we have proposed the publish/subscribe model as an integration middleware in robotics application. While in software engineering the publish/subscribe middlewares have been deeply investigated, in robotics there are only few systems that implement this communication model. We have presented a requirement grid to be used in choosing a publish/subscribe middleware for robotics and we have classified the available tools according to this grid. In Tables 1, 3, 2, and 4 we summarize this classification work.

[6]The following observation could be done also for the NDDS middleware presented in Section 6.7, but in that case the use is free for non–commercial use.

	Recofigurability	Community Architecture	Indentification
TCX	Client	Direct Centralized	Sender
IPT	Client	Direct	Sender
RTC	Client	Direct	Sender
IPC	Client	Direct Centralized Mixed	None
NDDS	Client	Distributed	None
ETHNOS	Dispatch	Distributed	Machine
DCDT	Dispatch	Distributed	Machine
TelRIP	Client	Distributed	Sender
SPLICE	Dispatch	Distributed	Sender

Table 1: Community Support for Classified Publish/Subscribe Middlewares

Many other researchers have recognized the need for a network communications layer allowing both data–transfer and event–signalling, but many of these approaches have been tailored to specific applications or embedded within an architectural model (e.g., MICA [41], APHRODITE [72], MIRO [24]). Recent requests for proposals from the European Robotics Research Network (Euron) to build a standard set of tools for robotics proove that a definitive choice has not been taken yet [62]. This set of tools (OROCOS – Open Robot Control System) is still lacking an inter–component communication middleware and only recently the CORBA–based middleware Smartsoft [63] has been proposed for this purpose.

As a final note, it can be noticed how the new robotics scenario described in Section 2 resembles the appliance integration one [40] or the ubiquitous computing framework [73]. Requirements in these research fields are, in fact, quite similar to tose described in Section 5. Our intuition is that middleware developed for robotics applications could be easily applied in those fields and vice–versa.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[2] H.A. Akeel and S.W.Holland. Product and technology trends for industrial robots. In *Proceedings of ICRA '00. IEEE International Conference on Robotics and Automation*, pages 696 – 700, 2000.

[3] A.L. Ananda, B.H. Tay, and E.K. Koh. A survey of asynchronous remote procedure calls. *ACM SIGOPS Operating Systems Review*, 26(2):92–109, 1992.

[4] R. Arkin. *Behavior–based Robotics*. MIT Press, 1998.

	Heterogeneity			
	Platform	Protocol	Operating System	Language
TCX	Sparc Intel x86 Motorola 68K SGI	TCP	VxWorks Mach	C
IPT	Sparc Intel x86 SGI	TCP Sockets	Solaris Linux IRIX	C C++
RTC	VME Cage	TCP Shared Memory	VxWorks	C C++
IPC	Sparc Motorola 68K Intel x86 SGI	TCP UDP	SunOS Solaris Linux VxWorks Win98/NT IRIX	C C++ LISP Java
NDDS	Sparc Motorola 68K Intel x86	UDP + RTPS	HP–UX Solaris Linux VxWorks Win98/NT	C++ Java
ETHNOS	Intel x86	Sockets UDP + EEUDP Shared Memory	Linux	C++
DCDT	Intel x86	Sockets UDP/TCP Shared Memory Serial Bus	Linux	C++
TelRIP	Sparc SGI Intel x86	Sockets TCP Shared Memory	SunOS Lynx Unix MS–DOS IRIX	C
SPLICE	Transputer Sparc	TCP/UDP	Helios SunOS	C

Table 2: Heterogeneity Support for Classified Publish/Subscribe Middlewares

	Content		Subscription
	Data	Type	
TCX	Record–based	Global	Content–free
IPT	Record–based	Global	Content–free
RTC	Record–based	Global	Content–free Subject–based
IPC	Record–based	Global	Subject–based
NDDS	Record–based	Global	Subject–based (hierarchical)
ETHNOS	Record–based	None	Subject–based
DCDT	Record–based	None	Subject–based
TelRIP	Record–based	Global	Content–based
SPLICE	Record–based	Global	Content–based

Table 3: Content and Subscription Supports for Classified Publish/Subscribe Middlewares

	Quality of Service			
	Delivery	Priority	Mobility	Real–Time
TCX	Guaranteed (ordered)	Yes	No	No
IPT	Guaranteed (ordered)	Yes	No	No
RTC	Guaranteed (ordered)	Yes	No	No
IPC	Guaranteed (ordered)	Yes	No	Timers
NDDS	Guaranteed Best–Effort	?	No	Deadlines Update Rates
ETHNOS	Guaranteed (not ordered)	Yes	No	Scheduling
DCDT	Best Effort Guaranteed (ordered)	Yes	No	Scheduling
TelRIP	Guaranteed (ordered)	No	No	No
SPLICE	Guaranteed (ordered)	No	No	Delivery Update

Table 4: Quality of Service for Classified Publish/Subscribe Middlewares

[5] A.V. Bakre and B.R. Badrinath. Reworking the rpc paradigm for mobile clients. *Mobile Networks and Applications*, 1(4):371–385, 1996.

[6] H.E. Bal and A.S. Tannenbaum. Distributed programming with shared data. In *Proceedings of ICCL – International Conference on Computer Languages*, pages 82–91, Miami, Florida, 1988.

[7] J. Bares and D. Wettergreen. Dante II: Technical description, results and lessons learned. *International Journal of Robotics Research*, 18(7):621–649, July 1999.

[8] B.Dalton and K.Taylor. Distributed robotics over the internet. *IEEE Robotics and Automation Magazine*, pages 22–27, June 2000.

[9] E. Bianchi, L. Dozio, G.L. Ghiringhelli, and P. Mantegazza. Complex control systems, applications of diapm-rtai at diapm. In *Proceedings of Realtime Linux Workshop*, Vienna, 1999.

[10] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.

[11] M.M. Bonsangue, J.N. Kok, M.Boasson, and E.D. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 159–168. ACM Press, 1998.

[12] J.L. Bresina, M.Bualat, M. Fair, R. Washingtony, and A. Wright. The K9 on-board rover architecture. Autonomy and Robotics Area, NASA Ames Research Center, 2000.

[13] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide–area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, December 1998.

[14] R. Cassinis. An application of automatic resource sharing to robot programming. In *Proceedings of III International Symposium of Robotics Research*, 1986.

[15] R. Cassinis, P. Meriggi, A. Bonarini, and M. Matteucci. Device Communities Development Toolkit: An Introduction. In *Proceedings of EUROBOT 01*, Lund, Sweden, September 2001.

[16] D.R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[17] W. Christiansen, J. Kurien, G. More, T. Sawyer, B. Smart, and S. Thrun. Beesoft user's guide and software reference. Real World Interface, Inc., 1997.

[18] L.A. Ciscon, J.D. Wise, and D.H. Johnson. A distributecl data sharing environment for cooperative intelligent robots. In *Proceedings of Fourth Annual Conference on Intelligent Robotic Systems for Space Exploration*, pages 95–108, 1992.

[19] C. Clark and S. Rock. Randomized motion planning for groups of nonholonomic robots. In *Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, 2001.

[20] Talarian Corporation. Mission critical interprocess communications: An introduction to SmartSockets. *White Paper*, 1997.

[21] P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, U. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wilkenloh. The design and implementation of the clouds distributed operating system. *Computing Systems*, 3(1):11–46, 1990.

[22] C. Dye. Oracle distributed systems, 1999.

[23] G. Eddon and H. Eddon. *Inside distributed COM*. Microsoft Press, 1998.

[24] S. Enderle, H. Utz, S. Sablatnog, S. Simon, G.K. Kraetzschmar, and G. Palm. Miró: Middleware for autonomous mobile robots. In *IFAC Conference on Telematics Applications in Automation and Robotics*, 2001.

[25] C. Fedor. TCX: an interprocess communication system for building robotic architectures. Technical report, Carnegie Mellon University, Robotics Institute, 1994.

[26] S. Fleury, M. Herrb, and R. Chatila. Genom: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots & Systems (IROS)*, pages 842–848, 1997.

[27] T.W. Fong. Fourth planet communicator. Fourth Planet Inc., 1998.

[28] T.W. Fong, C. Thorpe, and C. Baur. A safeguarded teleoperation controller. In *IEEE International Conference on Advanced Robotics 2001*, Budapest, Hungary, August 2001.

[29] B.P. Gerkey and M.J. Matarič. Murdoch: publish/subscribe task allocation for heterogeneous agents. In *Proceedings of the fourth international conference on Autonomous Agents*, pages 203–204. ACM Press, 2000.

[30] C.D. Gill and W.D. Smart. Middleware for robots? In *Proceedings of the AAAI Spring Symposium on Intelligent Distributed and Embedded Systems*, 2002.

[31] J. Gowdy. IPT: An object oriented toolkit for interprocess communication. Technical Report CMU-RI-TR-96-07, Carnegie Mellon University, Robotics Institute, March 1996.

[32] J. Gowdy. A qualitative comparison of interprocess communications toolkits for robotics. Technical Report CMU-RI-TR-00-16, Carnegie Mellon University, Robotics Institute, June 2000.

[33] Object Management Group. *CORBAservices: Common Object Services Specification*, July 1997.

[34] Object Management Group. *CORBA Services book*, 1998.

[35] Object Management Group. *Notification Service*, August 1999.

[36] B. Hayes-Roth. An architecture for adaptive intelligent systems. *Artificial Intelligence*, 72(1–2):329–365, 1995.

[37] Real Time Innovations Inc. Network data delivery service: The real-time connectivity solution 1.0 ed. Sunnyvale, California, June 1996.

[38] Real Time Innovations Inc. NDDS: Real-time networking made simple, 2000.

[39] SunSoft Inc. *Introduction to the ToolTalk Service*, September 1991.

[40] H. Ishikawa and T. Nakajima. A case study of implementing home appliance middleware on linux and java. In *Proceedings of 2002 Symposium on Applications and the Internet (SAINT)*, pages 31–34, Narar City, Nara, Japan, 2002. IEEE Computer Society Press.

[41] P. Judson, P. Lones, and L. Butler. A modular control architecture for real–time synchronous and asynchronous systems. In *Proceedings of the SPIE - Applications of Artificial Intelligence: Machine Vision and Robotics*, volume 1964, pages 287–298, 1993.

[42] M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software – Practice and Experience*, 22(11):985–1010, 1992.

[43] M.J. Matarič. Interaction and intelligent behavior. Technical Report AITR-1495, 1994.

[44] F. Mattern and M. Naghshineh, editors. *Pervasive Computing – First International Conference*, Zurich, Switzerland,, August 2002. Springer.

[45] Sun Microsystems. *XDR: Standard*, June 1987.

[46] Sun Microsystems. RPC: Remote procedure call protocol specification version 2. Technical report, Internet Network Working Group Requets for Comments RFC 1057, June 1988.

[47] Sun Microsystems. *Java remote method invocation specification*, 1997.

[48] Sun Microsystems. *Java Message Service Specification Version 1.1*, April 2002.

[49] E. Di Nitto and D.S. Rosenblum. On the role of style in selecting middleware and underwear. In *Proceedings of ICSE '99 Workshop on Engineering Distributed Objects*, May 1999.

[50] A. Nye and T. O'Reilly. *X Toolkit intrinsics programming manual: second edition for X11, release 4*. O'Reilly & Associates, Inc., 1990.

[51] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68. ACM Press, 1993.

[52] G. Pardo-Castellote, S. Schneider, and M. Hamilton. NDDS: The real-time publish-subscribe middleware. In *in Proceedings of the IEEE Real-Time Systems Symposium*, 1999.

[53] G. Pardo-Castellote, S.S. Thiebaud, M. Hamilton, and H. Choi. Real-time publish-subscribe protocol for IP–based Real–Time communications. Real Time Innovations Inc., September 2001.

[54] R. Patzke. Fieldbus basics. *Computer Standards & Interfaces*, 19(5-6):275–293, 1998.

[55] J.D. Pedersen. Robust communications for high bandwidth real-time systems. Technical Report CMU-RI-TR-98-13, Carnegie Mellon University, Robotics Institute, May 1998.

[56] M. Piaggio, A. Sgorbissa, and R. Zaccaria. ETHNOS–II a programming environment for distributed multiple robotic systems. In *Proceedings of the 32th IEEE Hawaii International Conference on System Sciences*, 1999.

[57] M. Piaggio, A. Sgorbissa, and R. Zaccaria. ETHNOS: a light architectue for real–time mobile robotics. In *Proceedings of the 1999 IEEE/RSJ International Conference on Intelligent Robots and System*, 1999.

[58] M. Piaggio and R. Zaccaria. An information exchange protocol in a multi–layer distributed architecture. In *Proceedings of the 30th IEEE Hawaii International Conference on System Sciences*, 1996.

[59] G.C. Roman, G.P. Picco, and A.L. Murphy. Software engineering for mobility: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 241–258. ACM Press, 2000.

[60] D.S. Rosenblum and A.L. Wolf. A design framework for internet–scale event observation and notification. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 344–360. Springer–Verlag, 1997.

[61] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.

[62] C. Schlegel. Communication patterns for OROCOS hints, remarks, specification, February 2002.

[63] C. Schlegel and R. Worz. The software framework SmartSoft for implementing sensorimotor systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS '99*, pages 1610–1616, Kyongju, Korea, October 1999.

[64] D. Schulz, W. Burgard, D. Fox, S. Thrun, and A.B. Creemers. Web interfaces for mobile robots in public places. *IEEE Robotics & Automation Magazine*, 7(1):48–56, March 2000.

[65] J. Silcock. Distributed shared memory: A survey. Technical Report TR C95/22, Deakin University, Geelong, Victoria, Australia, 1995.

[66] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1), February 1994.

[67] R. Simmons, R. Goodwin, C. Fedor, and J. Basista. *Task Control Archiecture: Programmer's Guide to Version 8.0*, May 1997.

[68] R. Simmons and D. James. *Inter Process Communication: A Reference Manual*, February 2001.

[69] H. Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[70] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

[71] C. Thorpe, O. Amidi, J. Gowdy, M. Hebert, and D. Pomerleau. Integrating position measurement and image understanding for autonomous vehicle navigation. In *Proceedings of 2nd International Workshop on High Precision Navigation*, November 1991.

[72] Aleksandar Timcenko, Steven Abrams, and Peter K. Allen. Aphrodite: Intelligent planning, control and sensing in a distributed robotic system.

[73] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[74] A. Willig. Analysis and tuning of the profibus token passing protocol for use over error–prone links. Technical Report TK-99-001, Technical University Berlin, March 1999.

[75] J.D. Wise and L. Ciscon. TelRIP distributed applications environment operating manual. Technical Report 9103, Rice University, Houston Texas, 1992.