Proceedings of the 2001 IEEE
International Conference on Robotics & Automation
Seoul, Korea • May 21-26, 2001

A Multi-processing Software Infrastructure for Robotic Systems

Andrew H. Jones Guilherme Nelson DeSouza Avinash C. Kak
Purdue University
{ajones, gdesouza, kak}@ecn.purdue.edu

Abstract

Robotic systems and their software design are based on the same principles found in other computer applications. However, while many computer systems to date are deeply rooted in concepts such as multi-processing, multi-threads, modularity, etc., robotic systems are still limited by mono-processing, centralized designs. This paper addresses these limitations by proposing a new philosophy for robotic system software design – software infrastructure – that allows for the design of an efficient, modular, fault-tolerant, and distributed software architecture. Two examples of this infrastructure applied to mobile robot navigation are also discussed.

1 Introduction

In the past fifteen years, specially since the works of [4, 1], researchers in robotic systems began to realize the importance of decomposing a centralized control architecture into smaller distributed units. These new distributed architectures became widely accepted [10, 5, 3], and much work has been dedicated to refine the high-level architecture. Despite some effort dedicated to message passing [11], there has been little attention to solutions addressing the practical issues in the low level infrastructure necessary for a robust, generic, and modular software architecture in the domain of robotic systems.

The resources necessary to implement this type of infrastructure have become available in computer science and have been employed in many areas [12]. While robotic control architectures are executing on the same computer systems, many of the software systems developed for robots are still designed and implemented using early monolithic computer models. Many robotic systems today use concepts such as behaviors [10, 1], subsumption [4], tiered-level [3], etc., to deal with the multi-facet requirements of the specific applications for which they were designed. However, most of these systems neglected the more profound changes that would add characteristics such as modularity, portability, encapsulation, and parallelism (distributed processing).

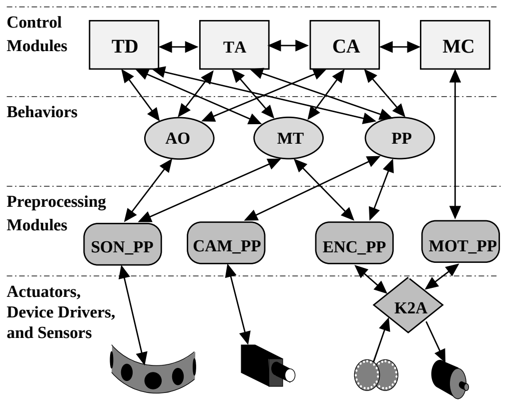

Figure 1: Behavior Based Navigation System

In this paper, we present a software infrastructure which is the foundation of some of our lab's systems, most notably in our mobile robot and visual servoing systems. This infrastructure combines concepts such as threads, processes, pipes, and software wrappers in order to provide a portable and modular environment. This allows the robot's resources (motors, cameras, sonars, range scanners, etc.) to be accessed concurrently by the different application modules operating in a distributed network of processors. Also at the end of the paper, we briefly introduce two mobile robot applications – a behavior-based navigation system (Fig. 1), and an object-oriented navigation system (Fig. 4)– that are being developed on top of this multi-processing software foundation.

2 Terminology and Background

In this section, we briefly present the definitions for some of the terms used throughout the paper. We understand that some of these terms are found in the computer science literature with a much more intricate definitions [12] than the one presented here. However, our goal in this section is to recapitulate some of these definitions for the target reader of this paper, which may not necessarily be familiar with these terms.

Resource - A physical device that needs to be controlled or accessed. For example: motors of a mobile robot; image grabber for cameras; etc.;

Process - An executing instance of a software function (or set of software functions) that run concurrently in the CPU. For example: a word processor and a web browser operating in different windows on the same computer where each program is a different process for the system;

Thread - A thread is frequently referred to as a *light-weight* process. In fact, a process can be made of many different threads, each one concurrently running inside the same process. Two threads inside a process can share the same variables. Spell-checkers, printing functions, and help assistants are possible examples of threads in a word processor;

Parent/Child Process - The relationship between processes and its implications to the system are not important in this paper. However, it is important to say that a parent process can create as many child processes as necessary, and those, in turn, can create their own child processes;

Server/Client - A *server* is a process or a set of processes that execute a specific task for the system. This task is only performed when a request is sent to the server by the*client* of this service. In our case, most of the servers' tasks is to provide access to a specific resource;

Software Wrapper - A set of functions or routines that give access to a server. The wrapper is the interface between processes and it is the wrapper that creates the *abstraction layer* that frees one process from knowing the details about how (or where) the other processes run;

Pipes - A *pipe* is a mechanism provided by the operating system with which processes can communicate and exchange information. In this paper, we use the term *pipe* indistinctively for intra-process, inter-process, or network (socket) communications.

Modularity/Encapsulation - A system is said to use encapsulation when all functions, routines, data structures, etc. relating to one and only one specific task of the system is enclosed in a single independent *module*. A module is a self-contained, executable program file.

3 System Infrastructure

As mentioned before, our motivation in proposing this software infrastructure is to bring some well known concepts found in distributed systems, real-time systems, etc., to the robotics domain.[1] In those systems, characteristics such as modularity, encapsulation, and portability, among others, are deemed necessary. For example, it's not uncommon for a particular task in a distributed system to migrate from one computer to another, depending on issues such as *fault-tolerance, CPU load*, etc. For this migration to happen, this same task must be well encapsulated in a single module so that it can be easily ported to another computer in the network.

In the robotics domain we found no different needs. The computer hardware for a mobile robot, for example, can be constructed using many independent CPUs performing tasks such as: a) controlling actuators; b) executing navigation software; c) supervising sensors; etc. Many of the above tasks need to communicate with one another regardless of the task allocation among the CPUs. In order to distribute the tasks, each task must run as processes, which are encapsulated in modules. Each module can be installed in its own CPU (or share CPU with other modules) completely independent of the other modules. However, since all these tasks need to cooperate in order to perform the overall system function, we need to provide a mechanism to exchange information between tasks. This mechanism, which is based on pipes and software wrappers, is the main focus of this paper. As it will be clear in the following example, this mechanism allows the design of a system with all the desired characteristics.

3.1 Example

In order to clarify these concepts, we present a simple example (Fig. 2) using our mobile robot, PETER, and its motion command service, K2A. Our mobile robot consists of a PC-based system mounted on top of a K2A Cybermotion base. The PC-based system running the Linux operating system contains two image grabbers, a network card, and a wireless interface to the internet. PETER also has an onboard controller for the active-vision stereo head, a controller for a laser range scanner, and an interface to five ultrasonic transceivers. Since the PC-based system is connected to the internet using a wireless bridge, any workstation in the network is a potential candidate to run any software task of our navigation system.[2]

To perform a simple navigation task, a basic system could be designed using the following four modules:

K2A Server: this module receives requests from various clients. These requests include *set-speed*, *turn*, *move*, *read-position*, *read-status* (e.g. battery voltage), *reset-deadman*, etc. Once a request is received, the server executes the command, monitors its completion, and returns a status along with the appropriate data regarding the current command.

Navigation Client: this module performs basic navigation decisions. It communicates with the K2A server to issue motion commands such as turn, set drive speed, etc.

[1]It is important to emphasize that despite the fact that these concepts have been widely applied in other domains, they are fairly new in the robotics domain.

[2]We usually run simulations and other matlab programs to display range maps in any of our Sun or PC workstations.

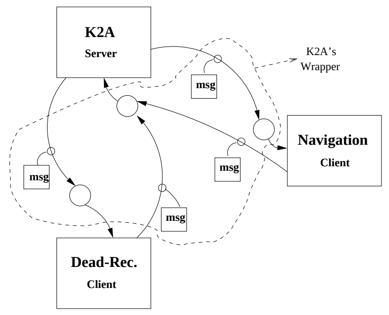

Figure 2: The clients call the K2A wrapper to exchange messages through their pipes

Dead-reckoning Client: this module reads the current position and orientation of the robot from the K2A server in order to keep track of its location within the environment.

System Status Client: this module monitors the status of the on-board batteries in the K2A and informs the K2A by resetting the deadman timer that the main CPU is in good standing.

As the reader can infer, the above three clients can concurrently send requests to the K2A server. Those requests are sent by invoking the K2A wrapper which communicates through the input pipe of the K2A server. However, the K2A server does not know which client sent the request. Therefore, the client also provides, through the wrapper, a return address to its own reply pipe.

3.2 Wrappers

As mentioned earlier, the motivation behind the wrapper is the need for the system to be encapsulated into independent, self-contained modules. With that in mind, the wrapper behaves as an abstraction layer that interfaces with the server. In simple words, the wrapper is a set of specific functions and routines that constitutes the protocol with which the client passes information to the server. Examples of the functions in the wrapper for the K2A include ROBOT_OPEN and ROBOT_MOVE. The client uses ROBOT_OPEN to establish a connection with the server while it uses ROBOT_MOVE to specify a drive speed and a turn angle. For all functions in the wrapper, the client needs to provide a reply pipe to which the server can respond. Within the wrapper, each function constructs a message which is sent forward to the server. This message is atomic, contains a specified number of fields, and can vary in length based on the command.

Upon reception of a message from its wrapper, the server identifies the command to be executed, and extracts all other necessary information that the message may contain to carry out the request. The server then executes the request and returns the appropriate data to the client by constructing a return message which contains the requested data and the status regarding the command executed.

3.3 Pipes, Processes, and Messages

Atomic message passing is a vital mechanism for exchanging data between processes. This is especially true in our robotic system where many processes are servers while other processes are clients overseeing the use of resources or services. Pipes are handled by the operating system in an efficient manner and offer a fast mechanism of communication between independent processes that may be running on different interconnected computers. The access to this mechanism resembles the normal open, read, and write commands for files. But unlike files, pipes are internal data structures stored in local memory or distributed in the network, not stored on slow disk drives. Another advantage with pipes is that when a process is waiting for data from its input pipe, it goes into idle mode. That is, the waiting process does not consume CPU time.

As mentioned before, pipes are only part of the picture. The message itself is a very important aspect of useful interprocess communication and the message structure deserves further discussion.

The message structure is intimately related to the relationship between processes. It is not unusual for processes to become client and server to each other at the same time. In fact, the relationship between client and server is only momentary. In a large system, processes can grow so interconnected that messages may need to pass from any one process to any other process. One could choose to define a unique message structure for each process. But this choice would force each process to know the specific message structure of the processes with which it communicates. As the system escalates in the number of processes, storing and using such specific message structures becomes more and more complex. Therefore, it is desirable to clearly define a single message structure that is versatile enough to handle all possible data types and data lengths needed for each process in the system. An example of a single message structure will be presented in subsection 4.1.

Atomicity is another aspect of message passing that a flexible, distributed topology requires. An atomic message is a message that is fully contained in a single data structure and is sent in a single write operation. If these conditions were not satisfied, two or more messages could be entangled with each other in the input pipe of the receiving process. For example, process A sends a message, '123456', where each digit is written separately into the pipe of process C, violating message atomicity. Concurrently, process B sends a message, 'FOO', to process C. In this situation, process C could receive an entangled and indecipherable messages, such as '123FOO456', which is unacceptable. However, if processes A and B sent atomic messages, process C would be guaranteed to receive the messages '123456' and 'FOO'. Note that the way in which the read operation from the pipe is preformed does not violate message atomicity.

Finally, another consideration in creating a flexible topology is forcing the message to contain information about the process that originated the message. In particular, the message must contain either the reply pipe, or the *process-id* of the sending process so that the receiving process can reply appropriately.

All these considerations – use of pipes, unique message structure, and atomic message passing – form the foundation of our system infrastructure. Together they provide flexibility, efficiency, portability, modularity, and encapsulation to any robotic system.

4 Infrastructure Applications

Up to this point, we have described a software infrastructure at the operating system level. In section 3, we demonstrated how one could construct a simple software architecture for a simple application of control and communication with the motion command service, K2A. But to demonstrate the true expandability and modularity of the proposed infrastructure, we will briefly discuss two system architectures being developed for a mobile robot: a Behavior Based navigation system and an Object Oriented navigation system.

4.1 Behavior-Based System

A behavior based architecture consists of many specific, task-oriented modules called behaviors that execute independently. For mobile robots, each behavior obtains input data that originates from some sensor or group of sensors and produces an output. This output combines in some fashion with outputs from other behaviors to determine vehicle motion within an environment. It is not the purpose of this paper to address behavior based navigation systems such as in [8, 10, 3, 1]. Instead, the main focus of this work is explaining the mechanisms embedded in the infrastructure and how these mechanisms allow the behaviors to communicate with each other. But, before discussing this communication, a brief description of the architecture is needed.

Architecture Overview Our architecture, Task Learning Architecture using Behaviors, TLAB (pronounced T-lab), naturally breaks into three units or levels of modules: *preprocessing modules*, *behaviors*, and *control modules* as illustrated in Fig.1. The *preprocessing modules* are strict[3] servers that encapsulate all functions and data necessary to serve a particular sensor or group of sensors. Many times these preprocessing modules are running on dedicated CPUs or microcontrollers. The *behaviors* map sensory data from appropriate preprocessing modules to a set of outputs for various control modules. As a module, each behavior is totally encapsulated and can operate at its own rate. This is important because reactive behaviors such as Avoid Object (AO) must be able to quickly affect the motion of the vehicle, while a passive behavior such as Plan Path (PP) affects the motion less frequently. The *control modules* decompose a high level task such as "navigate to room-D" into a sequence of safe, basic motion commands for the mobile robot. As Fig.1 indicates, the control modules combine the behavior outputs to carry out this task.

[3]A strict server is a process that is never a client of other servers.

Architecture Development The proposed infrastructure of messages and independent modules developed as a result of the limitations in attempting to implement TLAB under other possible infrastructures. The first possible infrastructure considered was a mono-process, mono-thread system. But given the large scale of this system and the need for each behavior to operate at a different rate, a single module that encapsulates the entire system would be very complex if not impossible. The second system design attempt was using multiple processes within a single module. In this single-module system, the Task Descriptor (TD) was the only parent process, while the rest of the system was created as a set of separate child processes. This design proved to be a poor choice for a number of reasons. First, because of operating system constraints; when one of the child processes failed (halted or exited prematurely), the entire system would fail. This alone creates an unacceptable solution for any robotic system which requires fault-tolerance. Second, due to the close relation of parent/child processes, debugging of a child process was nearly impossible. Third, child processes could not communicate with each other except through the parent process, so the parent became the bottleneck for message passing. These reasons lead us to an improved infrastructure, the one proposed, which consists of multi-processes distributed among independent modules.

Once the decision was made to create the system of behaviors by using independent modules, a message passing structure had to be clearly defined. Due to the interconnectedness of this particular system where any one module might need to communicate with any other module, all the messages within the system needed to be formulated using an identical and flexible structure. Given the above considerations, we arrived at a structure that is composed of two parts: a header and a body. See Fig.3 for a pictorial view of the message structure. The *header* is the first part of all messages while the *body* contains specific data that can vary in length. The header is based on the Unix message pro-

Header				Body
len	type	reply	command	stream of integers or characters

Figure 3: Message Structure for Behavior System

tocol [12] and contains the information such as: *len*, *type*, *reply*, and *command*. The field *len* contains the total length of the message body. The field *type* specifies both the type of data – character or integer – in the body and the message priority. The field *reply* passes the unique identification of the reply pipe. Finally, the action to be taken by the process receiving the message is provided in the *command* field. Note that no module actually fills the message. This task is performed by the wrapper of the recipient module.

Since the message varies in length, the receiving module reads from its input pipe in two operations. It first reads the header of the message to obtain the length of the body, then it can read the entire body without checking for an end-of-message marker.

Advantages of this Architecture Unfortunately, all further details about this architecture have to be omitted from this paper. However, we must list the reasons why this infrastructure offers an efficient and flexible solution for a behavior based architecture. These reasons are as follows:

Fault-tolerance - a global supervisor module can be instated to monitor all modules in the system. Because of the modularity of the design, upon the detection of a fault, the supervisor can take actions that range from re-prioritizing the scheduling of slow-responding modules, to the re-instantiation of unexpectedly terminated modules.

CPU Efficiency - due to the intrinsic characteristics of pipes, processes waiting for messages do not consume CPU time. Even for large scale systems with expanding number of processes, there is only minor impact on the CPU load.

Dissemination of Error Messages - by using a single, identical message structure, even the lowest-level modules can directly inform the highest-level modules of errors in the resources.

Elasticity - because of the encapsulation properties of this infrastructure, modules can be dynamically created and destroyed as necessary with little or no impact on other modules. For example, if a given task does not require ultrasonic transducer information, the corresponding preprocessor module can be destroyed.

4.2 Object-Oriented System

Most of the successful robot indoor navigation systems reported to date are constrained either by some kind of map of the environment [7, 2], by some artificial landmarks [6], or by the appearance of the environment where the robot is to navigate [9, 13]. In either

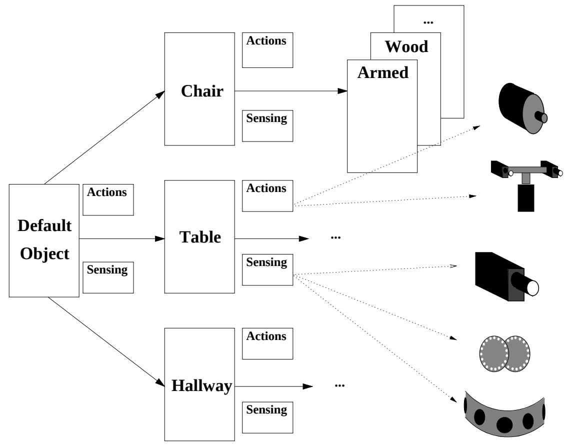

Figure 4: Object-Oriented Navigation System

case, information about the environment is stored in the robot memory[4] in a way that the system presents no generic structure that can be used in different and unexpected situations.

We believe that some of the limitations experienced with the navigation systems today derive from this lack of generic information about the environment. For that reason, we are developing a navigation system that is based on three central ideas: a) a Semantic Occupancy Map (SOM) that stores a description of each observed object; b) an object driven navigation scheme that embeds a generic representation of the environment inside directions such as *go down the **hallway** and turn at the third **door***; and c) a task subsumption architecture that allows the robot to take specific actions associated with each observed object.

In our system, the meaning of the word *object* is two-fold. An object is any element of the environment, such as: a table, a chair, a hallway, etc. But an object is also this element's representation under the Object-Oriented paradigm (OO). Objects can belong to classes of objects and inherit properties from more generic objects. For example, Fig. 4 depicts three basic objects – chair, table, and hallway – and their relationship with the generic object class, *Default Object*.

Some of the properties that an object may inherit are the class methods. Every object has a set of methods that we grouped as *Action* and *Sensing*. The Action group includes all the methods that in one way or another will cause parts of the robot to move – active vision stereo head, motors, etc. The Sensing group consists of the methods used by the Action group to monitor the execution of its actions by means of the robot sensors – odometers, image grabbers, sonar, etc. The so called *Task Subsumption Stacker* selects which objects will be active at a given time. This decision

[4]This information is either pre-loaded in the robot memory or it is acquired during a learning phase.

is based on the current configuration of the environment, which is stored in the SOM, and since all objects have access to the SOM, they can check the position and description of other objects in the environment. The objects take actions that help the robot to recognize new objects and that lead the robot to a different position toward the robot's goal.

It is not the scope of this paper to drudge through the details of each of the ideas above. Instead, we wish only to emphasize the importance of the proposed infrastructure for this navigation scheme. This importance derives from the fact that each object instance, by nature of OO, is encapsulated in a module. The intrinsic property of objects as being modules lends to the ability of dynamically instantiating objects (create/destroy) based on the observed environment. Finally, since a method is defined within the scope of each object and the servers can be implemented as objects, the wrappers of a service becomes the method itself. Therefore, all the above discussions on distribution, modularity, multi-processing, and encapsulation present in our proposed infrastructure also offers an efficient and flexible solution for the object-oriented architecture.

5 Conclusions

We have presented a new philosophy for robotic system software design – software infrastructure – that allows the development of efficient, distributed, fault-tolerant control architectures. This infrastructure makes use of widely accepted concepts such as multi-processing, software wrappers, pipes, message passing, etc. The application of these concepts leads to the design of systems that are intrinsically characterized by encapsulation, modularity, and fault-tolerance.

Two examples, in which we applied this infrastructure, were also introduced. These examples constitute two completely different paradigms of robotic control, but the proposed infrastructure proved to be indispensable in both cases.

The same philosophy presented here has also been applied and tested by integrating a previously developed system [7] on top of this new infrastructure. Other systems developed in the lab, such as visual servoing for tracking moving objects, are also benefiting from this infrastructure.

Acknowledgement

The authors would like to thank Ford Motor Company for supporting this work.

References

[1] R. C. Arkin, "Motor schema-based mobile robot navigation," International Journal of Robotics Research Vol. 8, No. 4, pp. 92-112, 1989.

[2] S. Atiya and G. D. Hager, "Real-time vision-based robot localization," IEEE Trans. on R&A, Vol. 9, No. 6, pp. 785-800, Dec. 1993.

[3] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. Miller, M. Slack, "Experiences with an Architecture for Intelligent, Reactive Agents", JETAI, Vol. 9, pp. 237-256, 1997.

[4] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," IEEE Journal of R&A, Vol. RA-2, No. 1, pp. 14-23, 1986.

[5] A. A. D. de Medeiros, R. Chatila, and S. Fleury, "Specification and Validation of a Control Architecture for Autonomous Mobile Robots", in Proc. IEEE IROS, pp. 162-169, 1996.

[6] M. R. Kabuka and A. E. Arenas, "Position verification of a Mobile Robot Using Standard Pattern," IEEE Journal of R&A, Vol. RA-3, No. 6, pp. 505-516, Dec. 1987.

[7] A. Kosaka and A. C. Kak, "Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties," Computer Vision, Graphics, and Image Processing – Image Understanding, Vol. 56, No. 3, pp. 271-329, 1992.

[8] J. Kosecka, H. Christensen, and R. Bajcsy, "Experiments in behavior composition", Robotics and Autonomous Systems, Vol. 19; pp. 287-298; 1997.

[9] T. Ohno, A. Ohya and S. Yuta, "Autonomous navigation for mobile robots referring pre-recorded image sequence," in Proc. of 1996 IEEE IROS, Vol. 2, pp. 672-679, Nov. 1996.

[10] J. Rosenblatt and D. Payton, "A Fine-Grained Alternative to the Subsumption Architecture for Mobile Robot Control", in Proc. of IEEE/INNS International Joint Conference on Neural Networks, Vol. 2, pp. 317-323, 1989.

[11] R. Simmons, "Structured Control for Autonomous Robots", IEEE Trans. on R&A, Vol. 10, No. 1, pp. 34-43, 1994.

[12] W. R. Stevens, UNIX Network Programming: Interprocess Communications, Vol. 1, Ed. 2, Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[13] J. Weng, and S. Chen, "Vision-guided navigation using SHOSLIF," Neural Networks, Vol. 11, No. 7-8, pp. 1511-1529, Oct-Nov. 1998.