A Robot Team for Exploration and Surveillance: Design and Architecture

Sascha A. Stoeter, Paul E. Rybski, Michael D. Erickson, Maria Gini,
Dean F. Hougen, Donald G. Krantz, Nikolaos Papanikolopoulos, Michael Wyman
Center for Distributed Robotics
Department of Computer Science and Engineering, University of Minnesota, U.S.A.
{stoeter, rybski, merickso, gini, hougen, npapas, mwyman}@cs.umn.edu
don.krantz@mts.com

Abstract. A novel architecture for distributed control of a team of heterogeneous mobile robots is introduced. Design as well as implementation details are presented. An example application for a surveillance task is discussed.

1 Introduction

Writing control software for mobile robots is a non-trivial task due to the needs of operating in noisy and cluttered environments. Additional complexity is introduced by inter-robot communication over a wireless network. This imposes constraints on the bandwidth, maximum distances between robots, and the need for error correction. Additional planning must be introduced to effectively control robots with heterogeneous capabilities. If a robot malfunctions, the team's performance must gracefully degrade and not fail outright. As discussed by Coste-Manière and Simmons [2], a good architecture is vital to the construction of working robotic systems.

We propose an architecture for multi-robot control that treats a distributed network of robots as a set of dynamic resources. Robot control is accomplished through a hierarchical behavior tree operating across a location-transparent wireless network. The entire system is designed to be extremely modular, allowing for rapid addition of behaviors and resources to create new missions.

There has been a decade and a half of active interest in providing robots with a behavioral or task-based decomposition of the control system, since it was introduced as an alternative to the traditional functional approach [1]. The application of behavior-based robotics to groups of robots has been explored extensively in the latter half of the 1990's (e.g., [5]) and architectures for cooperative control have recently been introduced (e.g., [6]). However, most of these systems have not tackled the problems of distributed collaborative behaviors and distribution of resources across robots.

KAMARA [4], a system developed in Karlsruhe, includes collaborative behaviors. The distributed and decentralized modes of operation are adequate for independent operations of the two manipulator arms and mobile base, but lack the coordination and communication needed for a larger group of robots. The idea of splitting behaviors into sensing, computing, and acting across multiple physical robots has been approached by treating a whole collection of robots as a single *generalized vehicle* [8]. This allows appropriate sensors, computational resources, and actuators to be chosen for each task without requiring them to all come from the same physical robot.

2 Hardware

Our heterogeneous robotic team consists of two types of robots. The first type is a larger, heavy-duty robotic platform called the "ranger." It is used to transport and deploy a number of small, mobile sensor platforms called "scouts," the second type of robot, into the environment. Together, the scouts and rangers form a hierarchical team capable of carrying out complex missions in a wide variety of environments. A team is created by pairing several scouts with one or more rangers. Because of the scout radios' range limitations, a ranger has to remain close by to act as a proxy. Proxy processing allows the scouts to engage in visual servoing – an activity they would never be able to accomplish with their own limited computational resources.

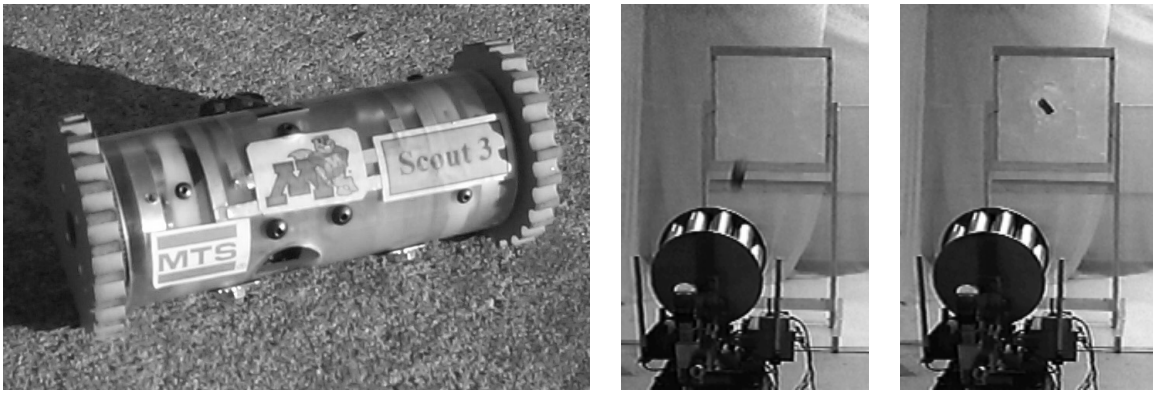
2.1 Scouts

Scouts have a cylindrical shape, 40 mm in diameter and 110 mm in length (see Figure 1). Each scout moves using a unique combination of locomotion types including rolling using the wheels mounted on both ends of its body and jumping using a spring "foot" mechanism. Rolling allows for efficient traversal of smooth surfaces, while jumping allows scouts to operate in uneven terrain and pass over obstacles.

The scouts act as the mobile eyes and ears of the team. Their electronics includes transmitters/receivers, microcontrollers, magnetometers and tiltometers. In addition, they have a modular sensor payload which can carry a miniature video camera (with an optional pan-tilt unit) and video transmitter, a microphone, a vibration sensor or a gas sensor. More details on the hardware can be found in Hougen *et al.* [3].

2.2 Rangers

The rangers are based on the ATRV-Jr.™ platform from the RWI Division of IS Robotics. A single ranger can carry a payload of roughly 25 kg. With a battery life of 3 to 6 hours (depending on terrain and load), the maximum range is about 20 km. It is customized with a scout launcher, scout communication hardware and a video camera. The rangers are equipped with on-board Pentium-based computers that have framegrabber cards allowing them to capture and process images from their own cameras as well as from scout-mounted cameras.



(a) scout robot (b) (c)

Figure 1: A scout and its deployment from a ranger. In (c) it breaks through the window.

The scout launcher allows the ranger to shoot scouts through windows or over obstacles that the ranger could not itself surmount. The ranger can deploy up to ten scouts in any order from its launcher before needing to be reloaded. With control over the launch angle and propulsive force, a scout can be launched up to 30 m.

3 Design Goals of the Software Architecture

The goal of this system is to provide distributed, fault-tolerant management of all available resources to fulfill the mission objectives. At the same time, mission design time is to be minimized while resource utilization is to be maximized.

The software architecture is composed of four major subsystems. The Mission Control is the brain of the system. It consists of layers of increasingly simple behaviors that execute the desired mission. The Resource Pool provides the behaviors with priority-managed, shared access to physical hardware as well as software resources such as maps. Humans interact with the system through the User Interface. The Backbone binds the other subsystems together.

3.1 Mission Control Subsystem

The Mission Control hosts prioritized behaviors that make up a solution to the mission. A top-level behavior is recursively decomposed into simpler behaviors until elementary behaviors (e.g. Drive Forward) are reached. Behaviors can execute in parallel.

3.2 Resource Pool Subsystem

The Resource Pool maintains the interfaces to *behavior resources* which are directly requested for use by behaviors. Such behavior resources are usually specific pieces of hardware (e.g., framegrabbers), but radio frequencies or maps of the environment can also be used. This should not be confused with *system resources*, such as available CPU time or memory, to which behaviors have only implicit access. For notational simplification, the term 'resources' denotes behavior resources unless otherwise specified. The Resource Pool provides for

a. *resource subscription* to allow behaviors to request access to resources,

b. *access control* to coordinate the utilization of shared resources,

c. *preallocation* of resources to behaviors to ensure their availability when needed,

d. *constraints monitoring* to watch for resource control violations, and

e. *constraints enforcement* to take control over resources and resolve conflicts.

3.3 User Interface Subsystem

Humans access the system through the User Interface. In the off-line mission design phase, it provides tools to allow the user to construct missions. During run-time, requirements of the User Interface are highly mission-dependent. For certain scenarios, such as rescue operations, the interface must be interactive. In other scenarios, such as planetary exploration, the interface will be used more for providing occasional status since real-time human response is not possible.

3.4 Backbone Subsystem

The Backbone ties the other subsystems together. It is the responsibility of the Backbone to enable the components of the system to work seamlessly over all allocated machines. It also tries to balance system resource usage to avoid bottlenecks. The Backbone provides for

a. *component placement* to start components on a specified platform,

b. *location transparency* of components over all platforms to ease addressing,

c. *location information* of both running and available components,

d. *communication* among the subsystems,

e. *hot-plugability* to allow for run-time addition and removal of components,

f. *redundancy* for fault-tolerance,

g. *system load monitoring* to report on the workload of the system resources,

h. *component migration* from one computer to another, and

i. *load balancing* to spread the workload evenly over the available system resources.

4 Implementation

Throughout the entire distributed system, many components, i.e. resources and behaviors, are at the mission designer's disposal. Ideally, a designer is offered pre-fabricated components that can be tied together easily to form a new mission. While the designer must certainly be aware of their functional characteristics, the system should hide lower-level details such as their location unless such information is requested by the designer. After all, information about the exact location of components is not available until run-time in the general case as their availability varies.

At present, only a limited number of design goals (a–b from the Resource Pool and a–e from the Backbone) are implemented with the remaining to follow after initial proof of concept. The system relies extensively on CORBA in order to achieve a high degree of flexibility while maintaining portable code. All entities in the system are registered with a CORBA name service and can therefore be easily addressed.

4.1 The Startup Services

In order to launch a mission, a number of core system services must be in place. The first service is the Distributed Robotics Daemon, the master startup service, which runs on each computer intended to take part in a mission. When contacted by the User Interface, it launches a CORBA environment to serve the behaviors. The second service, the Component Database, keeps track of all components available to the system storing names, locations, types and multiplicity information, i.e. the number of instances of each component allowed to exist simultaneously. Component Creators, the third service, are started on all machines. On mission startup, each Component Creator is responsible for determining the available components on its host and passing that information along

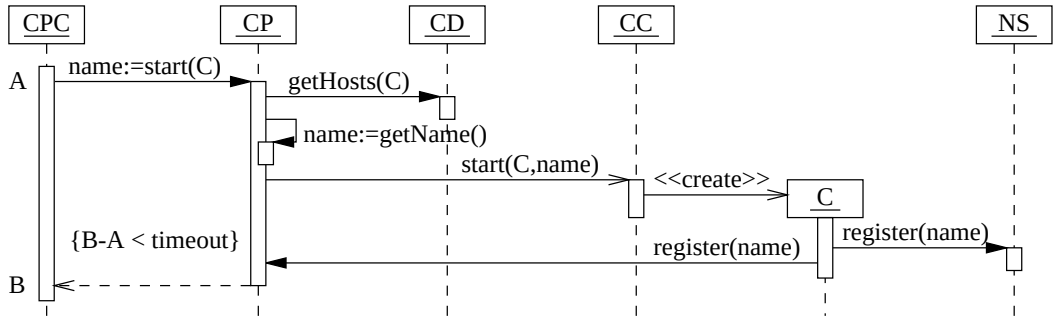


Figure 2: Component startup sequence.

to the Component Database. While the mission is running, the Component Creator serves requests for launching components and registering them with the Name Service. The final service, the Component Placer, is the global[1] service to start components and saves clients from having to know what hosts to run components on.

Figure 2 shows a typical startup sequence for a component C. When the Component Placer (CP) receives a request from a client (CPC) such as a behavior, it queries the Component Database (CD) for a host that supports execution of the component. The Component Placer then creates a unique CORBA name for the component and instructs the Component Creator (CC) on that host to start it. The component initializes and registers first with the Name Service (NS) and then with the Component Placer. The Component Placer returns either the component's CORBA name to the client or a failure condition in the case of a timeout.

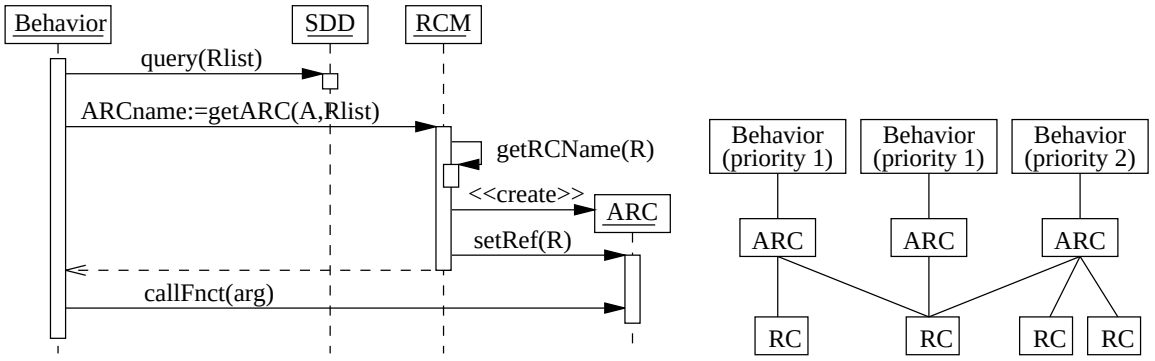
4.2 The Resource Controller Manager

The Resource Controller Manager manages components called Resource Controllers (RCs) and Aggregate Resource Controllers (ARCs). RCs provide standard CORBA interfaces to single hardware or software resources. Because multiple RCs must often be used simultaneously when controlling something complex like a robot, ARCs are used as higher-level interfaces to groups of RCs. Behaviors cannot access RCs and are given access to ARCs by the Resource Controller Manager.

Each behavior is programmed with the knowledge of what ARCs it needs and what kinds of RCs each ARC controls. This allows behaviors to instantiate a particular ARC by parameterizing it on a list of RCs. To specify this information, behaviors must be aware of dependencies among pieces of hardware such as what is connected to a particular computer. Since the users can change the hardware between runs, the Static Dependency Database stores the names, locations and startup options for the RCs available in the system and provide this information to the behaviors.

When a behavior is first initialized, its first task is to access all the ARCs that it will need to run. Figure 3(a) illustrates a typical example of this. The behavior first contacts the Static Dependency Database (SDD) to determine the available hardware for the ARCs it needs. The Resource Controller Manager receives this information as part of the request from the behavior to start a particular ARC type. Before the ARC can be created, the Resource Controller Manager checks to see whether the requested RCs are already running. If they are not, the Resource Controller Manager contacts

[1]This should not be confused with undistributed. It merely states that the service exists only once in the system. "Global" does not commit to any particular implementation.



(a) Behavior requesting access to an ARC.

(b) Example of behaviors accessing resources.

Figure 3: Initialization and operation of ARCs.

the Component Placer to start them (as shown previously in Figure 2). The Resource Controller Manager then creates an instance of the ARC and initializes it with the CORBA references to all of the RCs it will use. Finally, the behavior receives the ARC's CORBA name, allowing it to access the ARC's interface methods directly.

Some RCs can be used simultaneously by multiple ARCs while others cannot. When an ARC requests a sharable RC, it passes the RC the priority of the controlling behavior and a requested time slice. As long as the sum of the time slices does not exceed a maximum allowed capacity, ARCs with the same priority can share the RC. In the example shown in Figure 3(b), each of the three behaviors are trying to share a single RC with their ARCs. The priority 1 behaviors will be able to share the RC in a round-robin fashion while the priority 2 behavior will block until the priority 1 behaviors release it.

4.3 Mission Creation and Startup

The user can start a mission once all core services are in place. This is done by starting the top-level behavior of the mission. Behaviors are organized in a hierarchical structure and so the higher-level behaviors activate lower level behaviors as needed. Mission designers need only specify a partial order plan; linearization is accomplished by the system. Complex behaviors can be built from simpler, existing behaviors. Parent behaviors request the creation of their children through the Component Placer and set their parameters after successful instantiation. Named variables are used to obtain a common interface to allow for run-time addition of new behaviors. The parent behavior then activates the children and awaits their termination. It must also react to an asynchronous shutdown message from its parent. In that case, it recursively shuts down its children.

4.4 A Walk through an Example Mission

A scenario we have demonstrated requires the robotic team to quickly explore a building and set up a surveillance sensor network. Experimental details are reported in Rybski *et al.* [7]. In this scenario, a ranger moves into the building and searches for rooms into which to deploy scouts, using navigation software to move autonomously along a corridor and to find open doors (see Stoeter *et al.* [9] for details). The ranger then launches a scout into each room. The scouts investigate the room (by transmitting

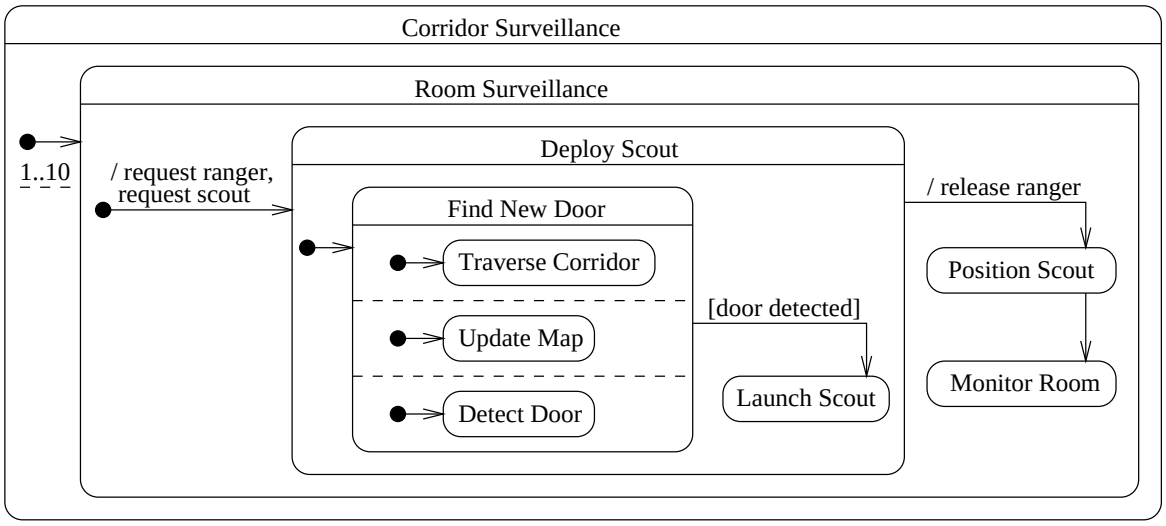


Figure 4: Example corridor surveillance mission.

images to the rangers for proxy processing), move towards dark corners in which they can hide, turn to watch their area, and wait for people to move through the environment.

The scenario is depicted in Figure 4. For visual clarity, error transitions are not shown. A rounded box corresponds to a behavior, arrows denote transitions between behaviors. A transitions is triggered when its condition becomes valid. This happens when a composite behavior is entered and its default transitions (as marked with dots at their origins) are activated, when the source behavior terminates, or the transition condition (as shown next to the arrow in brackets) evaluates to true. When a transition is taken, first the source behavior is shut down, then the optional transition action (as described after a slash) is executed, and finally the destination behavior is activated.

Note that behaviors can execute in parallel. This is shown with dashed lines in the example mission. For instance, all three children of Find New Door run concurrently. Similarly, up to ten Room Surveillance behaviors could be active at the same time, as denoted by the dashed multiplicity range.

At startup, Corridor Surveillance instantiates all its children, i.e. ten Room Surveillance behaviors. Each one tries to subscribe to a ranger. If just a single ranger resource is available in the system, only one Room Surveillance behavior can proceed while the remaining nine sleep until the Resource Controller Manager is able to fulfill their request. With multiple rangers, the system would automatically grant access to more behaviors and pieces of the mission could continue in parallel. Once in control of a ranger, the behaviors use the Static Dependency Database to subscribe to a scout that is stored in that ranger's magazine.

Find New Door creates a map that it makes available to all three of its children. This is the only way for behaviors to share information, as they are unaware of each other's presence yet need to synchronize. When Detect Door terminates successfully, Find New Door shuts down the remaining two children and activation passes to Launch Scout. After the scout has been deployed, Find New Door has no need for a ranger and thus frees the resource. At this point, the Resource Controller Manager hands the resource to another instance of Room Surveillance. The scout is directed to a good location using visual servoing and subsequently begins monitoring the room for trespassers.

5 Future Work

Future work includes extensions of the system and testing. First, the remaining design goals will be implemented. The most interesting parts promise to be the component migration at run-time for load balancing and the incorporation of dynamic constraints enforcement. The latter will free mission designers and component writers from a tremendous amount of manual constraints-checking necessary at present which is both expensive and can easily be done wrong. Second, the system will be tested on a variety of different missions. We intend to develop solutions to a number of well studied problems and compare our system performance. We plan on exploring the unique opportunities offered by our robotic team.

Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency, Microsystems Technology Office (Distributed Robotics), ARPA Order No. G155, Program Code No. 8H20, Issued by DARPA/CMD under Contract #MDA972-98-C-0008.

References

[1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[2] Ève Coste-Marière and Reid Simmons. Architecture, the backbone of robotic systems. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 67–72, San Francisco, CA, April 2000.

[3] Dean F. Hougen, Saifallah Benjaafar, Jordan C. Bonney, John R. Budenske, Mark Dvorak, Maria Gini, Donald G. Krantz, Perry Y. Li, Fred Malver, Brad Nelson, Nikolaos Papanikolopoulos, Paul E. Rybski, Sascha A. Stoeter, Richard Voyles, and Kemal Berk Yesin. A miniature robotic system for reconnaissance and surveillance. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 501–507, San Francisco, CA, April 2000.

[4] Thomas Laengle and Tim C. Lueth. Decentralized control of distributed intelligent robots and subsystems. In *Artificial Intelligence in Real Time Control*, pages 281–286, Valencia, Spain, 1994.

[5] M. Matarić. Issues and approaches in the design of collective autonomous agents. *Robotics and Autonomous Systems*, 16(2-4):321–331, December 1995.

[6] Lynne E. Parker. ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, April 1998.

[7] Paul E. Rybski, Sascha A. Stoeter, Michael D. Erickson, Maria Gini, Dean F. Hougen, and Nikolaos Papanikolopoulos. A team of robotic agents for surveillance. In *Proc. of the Int'l Conf. on Autonomous Agents*, pages 9–16, Barcelona, Spain, June 2000.

[8] J. Borges Sousa and F. Lobo Pereira. A general control architecture for multiple vehicles. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 692–697, Minneapolis, MN, 1996.

[9] Sascha A. Stoeter, Frédéric Le Mauff, and Nikolaos P. Papanikolopoulos. Real-time door detection in cluttered environments. In *IEEE Int'l Symposium on Intelligent Control*, Rio, Greece, July 2000.