Resource Scheduling and Load Balancing in Distributed Robotic Control Systems

Colin McMillen, Kristen Stubbs, Paul E. Rybski, Sascha A. Stoeter, Maria Gini*, Nikolaos Papanikolopoulos
Department of Computer Science and Engineering, University of Minnesota, U.S.A.

This paper describes the latest advances made to a software architecture designed to control multiple miniature robots. As the robots themselves have very limited computational capabilities, a distributed control system is needed to coordinate tasks among a large number of robots. Two of the major challenges facing such a system are the scheduling of access to system resources and the distribution of work across multiple workstations. This paper discusses solutions to these problems in the context of a distributed surveillance task.

Keywords: Multiple robots; Resource allocation; Load balancing; Software architecture

1. Introduction

There are many advantages in using a distributed control system to manage the operation of a group of robots. Such a system could control a wide variety of heterogeneous devices and do so over great physical distances. This system could be very modular, supporting many kinds of devices. However, designing and implementing a distributed control system such that it works effectively and efficiently across varying hardware configurations and in spite of differing computational demands is a non-trivial task.

For instance, one challenge is to ensure that various system components have access to the resources (such as communications channels, computational units, robot chassis, or sensor inputs) that they need in order to accomplish their tasks. Not all resources can accommodate many simultaneous access requests, and many resources can only handle a single request at a time. Another challenge is to determine how to spread the demand for resources over the entire system. This helps to ensure that the computational and resource load is balanced, and that no one component is taxed too heavily.

A distributed control system has been developed for managing a group of small, mobile robots which have extremely limited on-board sensing and computing capabilities. These robots, called Scouts [8], are primarily designed for military urban surveillance applications. The requirements of effective urban surveillance dictate the Scouts' primary design constraints: the robots must be small enough to hide from plain view and light enough to be thrown or fired into a building. The robots must be able to work in teams: this capability ensures that a larger physical area can be surveyed by the robots and also provides a degree of redundancy in the event that some of the robots are disabled or otherwise unable to complete their tasks. Furthermore, the robots need to operate autonomously. Due to the size and weight constraints of the Scout's design, the computational hardware presently available on a Scout is limited to two 8-bit microcontrollers. Additional hardware present on the Scout typically includes a video camera, an analog radio frequency (RF) receiver providing a low-bandwidth means of sending data to the robot, and an analog RF transmitter for broadcasting live video from the robot's camera.

The limited computational hardware present on the Scouts makes them completely reliant upon a proxy processing scheme for operation. In this scheme, the "bodies" of the robots are physically separate from the "brains" and are connected only by wireless data links. Any work-

*Corresponding author. Address: 4-192 EE/CS Building, 200 Union St. SE, Minneapolis, MN 55455, U.S.A E-mail: gini@cs.umn.edu

station within range of a particular Scout robot is capable of receiving and interpreting its transmissions. Because the control software for the Scouts runs on one or many external workstations, it has access to the resources of other participating workstations on the Internet. This means that the workstation processing the sensor input does not necessarily have to be the same as the one that is transmitting motion commands back to the Scout robots.

The communication channels that the Scouts use to send and receive information are very limited in power and available throughput. As a result, access to these channels must be explicitly scheduled so that the demand for them can be met while maintaining the integrity of the system's operation. The Scout control architecture has been developed to take these factors into account. Additionally, because the architecture can make use of multiple networked workstations, it attempts to improve the entire system's performance by balancing the computational load across these computers.

This paper provides a general overview of the Scouts' control architecture and describes the architecture's resource management and load balancing capabilities. Experimental results which illustrate the utility of these features are presented.

2. Related Research

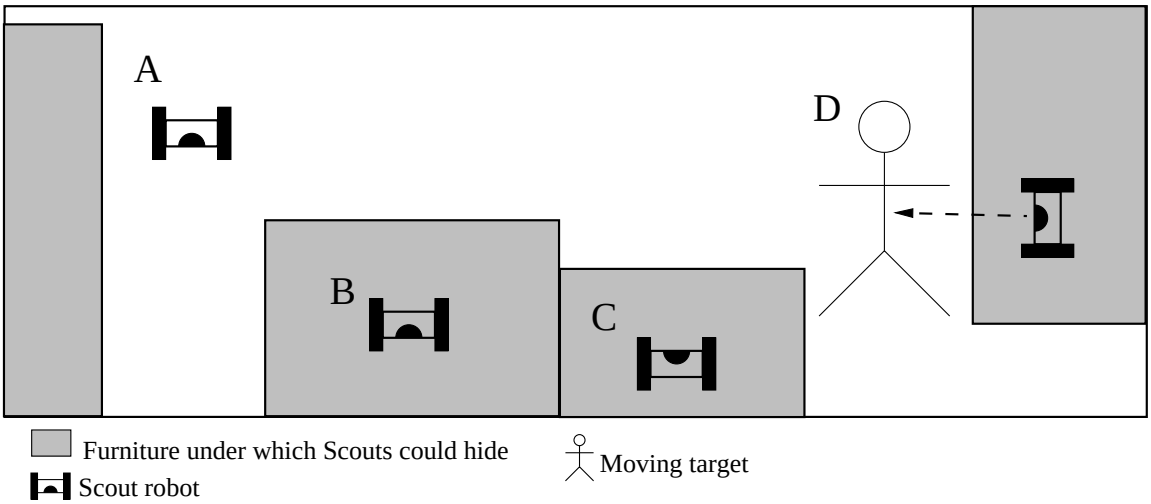
A software architecture designed for controlling groups of robots must allow for distributed operations, handle resource allocation, and support real-time operations with graceful degradation. A number of architectures have been proposed, many of them described in [5]. Our architecture has some similarities with CAMPOUT [7], a distributed hybrid-architecture based on behaviors. The major difference is that we focus on resource allocation and dynamic scheduling, while CAMPOUT is mostly designed for behavior fusion. We rely on CORBA [4] as the underlying technology for distributed processing, while in CAMPOUT each robot runs an instance of the architecture and uses sockets for communication with other robots. Our architecture also has some similarities with ALLIANCE [6], which provides distributed and fault-tolerant control for teams of homogeneous robots. The issues that our architecture addresses are more general, allowing for control of heterogeneous robot teams as well as not putting any restrictions on the methodology for the robot control (deliberative or reactive).

Resource allocation and dynamic scheduling are essential to ensure robust execution. Our work focuses on dynamic allocation of resources at execution time, as opposed to analyzing resource requests off-line, as in [1,3], and modifying the plans when requests cannot be satisfied. Our approach is specially suited to unpredictable environments, where resources have to be allocated in a dynamic way that cannot be predicted in advance. We rely on the wide body of algorithms that exists in the area of real-time scheduling [9] and load balancing [2]. Our architecture provides support for distribution of resources across robots, use of shared resources, and seamless integration of autonomous and human-supervised control [10].

3. Distributed Robotic Control

The Scout robot is a cylindrical, two-wheeled robot that is 40 mm in diameter and 110 mm in length. Designed for portability and the ability to access hard-to-reach areas, Scouts have potential applications in the fields of urban surveillance, search-and-rescue scenarios, and other situations requiring a team of small, autonomous, maneuverable robots. One of the primary missions that these robots are designed for is a distributed reconnaissance and surveillance task where larger robots called Rangers are used to transport and deploy the Scouts into areas where they can gather further information. In any particular mission, there could be up to ten Scouts assigned to a single Ranger. More detail about the Ranger/Scout hardware, the reconnaissance and surveillance missions they are designed to do, and some experimental validation of these missions is beyond the scope of this paper and instead can be found in [8].

To facilitate the operation of team-based autonomous behaviors, a software architecture has



A: This Scout has started in an open part of a room surrounded by furniture. It starts by searching for a dark place in which to hide.

B: This Scout has moved to a dark area (the shadow underneath a piece of furniture) and hidden in it.

C: This Scout has turned toward light (in this case, the open area of the room).

D: This Scout is watching for motion, alerting the user to the presence of any moving targets within its line of sight.

Figure 1. Example of the Place Sensor Net behavior.

been developed. This software architecture consists of several components. Many of these components are Resource Controllers (RCs), which control access to various system resources, such as the Scout robots, radio and video transmission frequencies, and various framegrabber cards. Some RCs, such as those that manage the radio frequencies, are *sharable*, which indicates that multiple system components may be granted concurrent access to these resources. Other RCs, such as those that manage the framegrabber cards, are *non-sharable*, which indicates that only one system component may access them at a time. Other components, called Behaviors, utilize the system RCs to perform autonomous tasks. Behaviors can be joined together in a hierarchical structure to perform arbitrarily complex tasks. To simplify the writing of Behaviors, an abstraction called the Aggregate Resource Controller (ARC) has been developed. ARCs encapsulate Behaviors' requests for simultaneous access to groups of RCs.

One of the most complex autonomous behaviors the system supports is Place Sensor Net, which is the establishment of a sensor network of Scout robots (Figure 1). This is the deployment phase of the mission, where a Ranger sends a number of Scouts into an area so they can better view the environment. When the Place Sensor Net behavior is activated, all Scouts involved first search for a dark location and servo toward it, attempting to hide. After each Scout is hidden, it turns toward the light and watches for motion. This task is relatively demanding of the software architecture, making it an ideal example of how the architecture schedules access to resources and how it shares the workload among multiple machines.

4. Resource Controller Manager

The Resource Controller Manager (RCM) is the core service responsible for scheduling access to all system resources. A behavior requests simultaneous access to a group of RCs by creating an ARC and having the ARC request a schedule from the RCM. Each ARC has several parameters, set by the behavior, which are of interest to the RCM. The most important parameter is `needed RCs`, a list of all the requested RCs. For each sharable RC in this list, the behavior must provide one or more `bandwidth` parameters, which quantify (in an RC-specific way) how the behavior intends to utilize the resource. Other ARC parameters set by the behavior include a `priority` and a `minimum runtime`. The latter parameter specifies the minimum amount of time that the behavior must control all requested RCs in order to accomplish any useful work. This parameter is necessary in our system because some components, such as the Scouts' video cameras, can take several seconds to activate, during which the behaviors can take no useful action.

Whenever the RCM computes a proposed schedule, it must query all sharable RCs that will be in use. These queries determine whether the sharable RCs can handle the proposed sets of ARCs, given the ARCs' `bandwidth` parameters. This is done because the RCM cannot be expected to know the implementation details of every sharable RC. The sharable RCs must use the `bandwidth` parameters provided to determine whether they can handle the requested load. If a sharable RC determines that it cannot handle the load, it rejects the proposed schedule and the RCM is forced to come up with a different schedule proposal. In this way, the domain-specific knowledge of a sharable RC can be used as a veto power, influencing the RCM's scheduling decisions.

A sequence diagram in the Unified Modeling Language (UML) showing the behavior of the RCM and related components during a typical schedule request is presented in Figure 2. This diagram shows a Place Sensor Net behavior creating arc0, an ARC which requests a schedule from the RCM. The RCM computes a proposed schedule and sends it off to all the sharable RCs on the system. All sharable RCs accept the schedule, so the RCM notifies all scheduled ARCs (arc0 and any others that may have already been running) that they now have access to their needed RCs. The Place Sensor Net behavior can then use these resources to complete its task. When the ARC's `minimum runtime` has expired or the RCM has determined that the ARC should be preempted, the ARC is notified that its resources are no longer available. In turn, the ARC notifies its behavior that the resources are no longer available, and the behavior stops processing.

Two different strategies for computing the schedule have been tried. In the first, an optimal schedule is produced such that the number of RCs in use at any given time is maximized. The problem with this approach is that, for large numbers of RCs, computing the optimal schedule is exponentially complex (in terms of time). The computational complexity for this approach is dominated by the calculation of all possible sets of RCs.

Computing these RC sets and sending each of them off as a query to each sharable RC for testing has a worst-case time complexity on the order of $O(2^n)$, where n is the number of RCs available to the system. This is acceptable for small sets of RCs, but it does not scale well even for moderate numbers of RC requests.

The second scheduling strategy reduces total runtime by choosing a schedule that can be computed in polynomial time. This approach considers several factors when calculating a schedule. If there is resource contention between multiple ARCs, their `priority` values are first examined, with ARCs of higher `priority` having the chance to run first. If the priorities of multiple ARCs are the same, the RCM looks at their `minimum runtime` parameters. The RCM schedules ARCs with lower `minimum runtime` values first. If the `minimum runtime`s are also identical, the RCM arbitrates the tie by giving priority to the first ARC that requested scheduling. ARCs which do not immediately get the chance to run are queued; the RCM examines this queue every

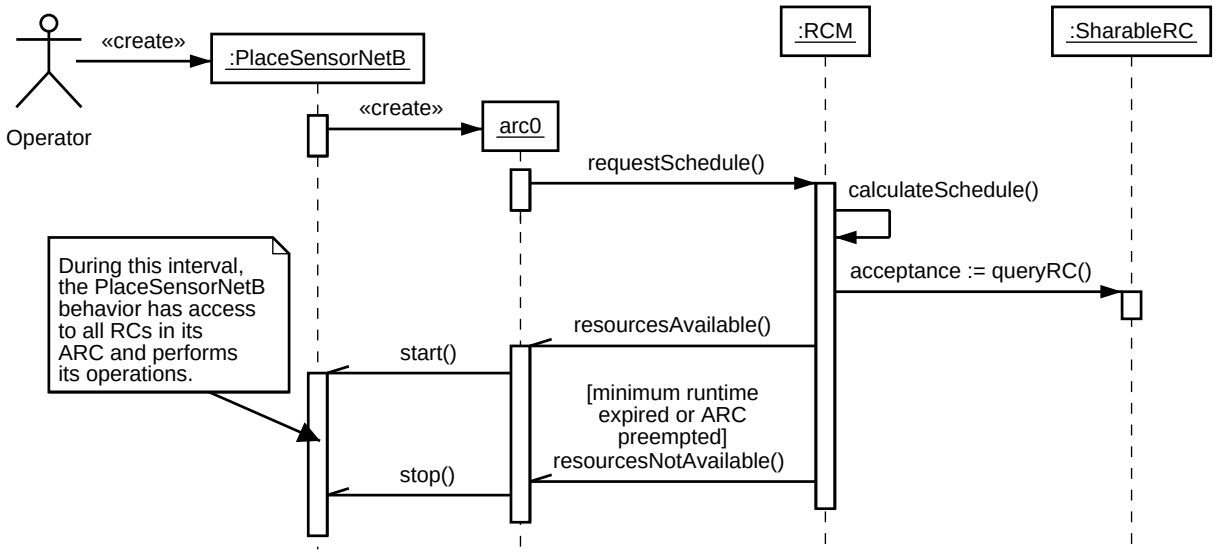


Figure 2. UML sequence diagram showing a typical RCM schedule request.

second[2] to determine if any queued ARCs can be started. If there is a queue of ARCs waiting for resources, a new ARC making a resource request must also contend with the ARCs in the queue. An additional feature of this scheduling algorithm is its ability to preempt currently-running ARCs (which may not have been running for their requested `minimum runtime`) if an ARC of higher priority requests a schedule. Any preempted ARCs that have not completed their `minimum runtime` enter the queue of waiting ARCs until their needed resources are free again. In this case, the preempted ARCs will be allowed another full slot of `minimum runtime`. The runtime complexity of this scheduling algorithm is dominated by the number of ARCs making RC requests and the number of total possible RCs each ARC can request.

The worst-case time complexity of calculating the schedule is $O(mn^2)$, where m is the number of RCs in the system and n is the number of ARCs contending for access to those RCs.

[2]The time quantum of this architecture is one second. This value was chosen because behaviors and other components do not make requests faster than once a second.

Preliminary testing illustrates significant performance difference between the two RCM scheduling algorithms. The two versions of the scheduler were run in a controlled experiment in which ten RCs were available to the system. Twenty behaviors were written to simultaneously contend for these resources; each behavior requested exclusive access to up to five RCs. Over ten trials, the first version of the scheduling algorithm took a mean of 39.3 seconds to calculate a schedule, while the second version calculated a schedule in a mean time of 0.59 seconds. The second algorithm is preferred because it is clearly more responsive to access control requests.

To demonstrate the operation of this scheduling algorithm, a version of the Place Sensor Net behavior was run on an implementation of the RCM. In this scenario, there are four resources available to the system. Two of these resources correspond to Scouts, while two correspond to the framegrabber cards used to process video. (To simplify this demonstration, no sharable RCs were used.) The Place Sensor Net behavior creates a total of four ARCs to encapsulate resource requests; all of these ARCs request a schedule

Table 1
Summary of resource requests made during the Place Sensor Net scenario.

ARC	Sched. Req. Time	`Priority`	`Min. Runtime`	`Needed RCs`
arc0	$t = 0$ sec.	1	5 sec.	Scout33, FGrab1
arc1	$t = 0$ sec.	1	5 sec.	Scout35, FGrab1
arc2	$t = 1$ sec.	1	4 sec.	Scout33, FGrab2
arc3	$t = 2$ sec.	1	3 sec.	Scout33, FGrab2
arc4	$t = 11$ sec.	10	3 sec.	Scout33, Scout35, FGrab1

Table 2
Allocation of resources during the Place Sensor Net scenario.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Scout33	0	0	0	0	0	3	3	3	2	2	2	4	4	4	2	2	2	2
Scout35	-	-	-	-	-	1	1	1	1	1	-	4	4	4	-	-	-	-
FGrab1	0	0	0	0	0	1	1	1	1	1	-	4	4	4	-	-	-	-
FGrab2	-	-	-	-	-	3	3	3	2	2	2	-	-	-	2	2	2	2

within the first two seconds. After eleven seconds have elapsed, a human operator takes manual control of the system, requiring two Scouts and a framegrabber card. The human controller has priority over all autonomous behaviors, so the ARC created on the operator's behalf has a very high priority.

Table 1 summarizes the resource requests made by the various ARCs during the scenario. The ARCs are ordered by the time each requested the RCM for a schedule. Table 2 shows how the scheduler allocated the resources to this set of ARCs.

Initially, arc0 and arc1 request a schedule from the RCM. However, they contend for FGrab1, the first Framegrabber RC. Since their `priority` and `minimum runtime` parameters are identical, the RCM chooses to schedule arc0 first, because arc0 was the first to initiate a schedule request. At time $t = 1$, arc2 makes a schedule request, but is blocked because arc0 is using Scout33. At $t = 2$, arc3 makes a schedule request. This ARC has a shorter `minimum runtime` than arc2, but requires the same resources, which are still in use by arc0. Hence, it is also blocked. At $t = 5$, arc0's `minimum runtime` has expired, so it is released from the schedule, and the RCM now has to allocate the new set of free RCs among the waiting ARCs. Since arc1, arc2, and arc3 are all of the same `priority`, arc3 gets its resources allocated first, because it has the shortest `minimum runtime`. The free RCs remaining are exactly those needed by arc1, so it is also scheduled to run at this time. After three seconds have elapsed, arc3's resources are given to arc2. arc2 continues until $t = 11$, when arc4 requests control of many resources on behalf of a human operator. This ARC preempts arc2, since it has a higher `priority`. arc2 is placed back into the queue of waiting ARCs until $t = 14$, when arc4 finishes and its RCs are freed. arc2 then runs for its full four-second `minimum runtime` before it is finished. After $t = 17$, arc2's resources are freed, and the system returns to an idle state.

5. Load Balancing

Load balancing is another important part of the architecture. Without load balancing, the placement of components across computers will likely be done in a suboptimal way. If many components are started on the same workstation, or if the selected workstation has limited computational power, the overall performance of the system will decrease.

Since components can be started at any time

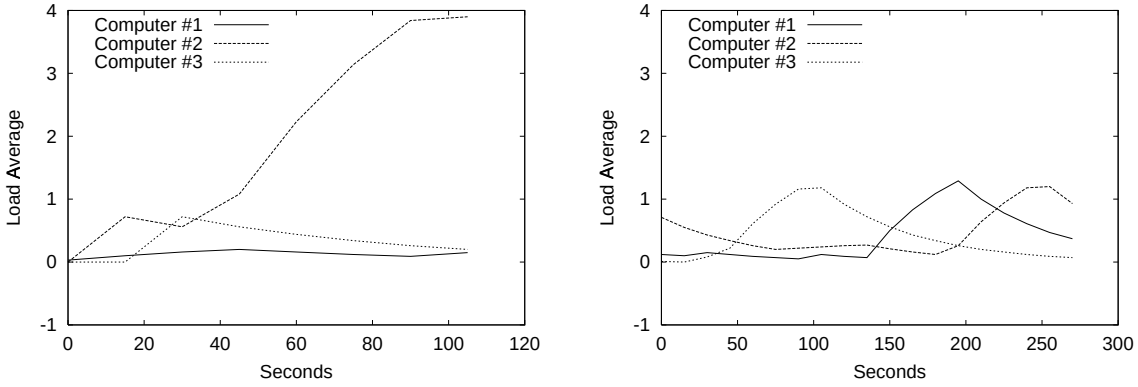


Figure 3. Load average on three computers with (a) simultaneous behavior start times and (b) staggered behavior start times.

during the running of the architecture, and since the architecture currently does not support process migration (the movement of a running process from one workstation to another while maintaining its internal state), there is no other way to guarantee that all computational resources will be used equally.

Load balancing is done by the Load Balancer and the Load Reporter, two core services dedicated specifically to this task. When a new component (such as a behavior, an ARC, or an RC) is to be started, several conditions are analyzed to determine the workstation with the least amount of load. The component is then started on that machine. Each machine capable of starting software components hosts a Load Reporter, which periodically sends relevant load information to the central Load Balancer service. When the Component Placer needs to start a new component, it queries the Load Balancer for the best host on which to start the component, given the list of possible hosts. The Load Balancer returns the name of the best host based on a set of specified conditions, such as current load average, processor speed, and available memory. The Component Placer may choose which metric it wishes the Load Balancer to use based on runtime conditions.

The current load balancing system has both advantages and disadvantages; these can best be seen by examining two simulated Place Sensor Net scenarios (Figure 3), which illustrate how the Load Balancer operates in different situations. In the first situation (Figure 3(a)), one Place Sensor Net is started using all of the available Scouts simultaneously. Since none of the computers in the network have a very high load, all of the components are started on one machine. Once the behavior begins to run, it is obvious that system resources are not being used effectively. Without process migration, components are confined to the machine on which they were started, regardless of how its load changes. The current system does not compensate well for this, but it does perform load balancing well for components which are started at staggered intervals. The second situation, shown by Figure 3(b), illustrates the starting of three different Place Sensor Net behaviors. These sensor nets are placed in three different areas which require varying amounts of time for the three groups of Scouts to reach, so the Place Sensor Net behaviors are not started at the same time. As the graphs indicate, the Load Balancer starts up components on each of the available machines because it takes into account

which machines have heavier loads each time the Component Placer needs to start a new component. The new Load Reporter and Load Balancer core services help to ensure that new components are distributed appropriately and efficiently over the set of available hosts.

6. Summary and Future Work

Details of the scheduling and load-balancing capabilities of a distributed robotic control architecture have been presented. The scheduling algorithm used by the Resource Controller Manager does not necessarily guarantee the generation of an optimal schedule of RCs, but it is capable of generating a reasonable schedule within an acceptable amount of time. The Load Balancer service tries to make sure that the computational load is spread equally among all workstations in the system.

The schedule generated by the new implementation of the Resource Controller Manager is not necessarily optimal. It is possible to devise degenerate cases in which the Resource Controller Manager would generate schedules which are substantially inferior to the optimal schedule. It is not clear at this time whether these cases ever occur in practice. An extensive empirical analysis of the performance of the Resource Controller Manager on a variety of situations could be performed. This empirical analysis could help to determine where the heuristic scheduling algorithm is detrimental to the overall performance of the system. This data could then be used at runtime to detect degenerate cases, switching over to the optimal algorithm when necessary.

One area of future research is process migration. The current load balancing system decides only where components should be started, but it cannot move them from one machine to another. The next step would be to allow components to be moved from one computer to another while minimizing the interruption of their operation and preserving their internal state. Because this is a distributed system running over connections that may have high latency, there is currently no way to guarantee real-time performance of robotic behaviors. All behaviors are written with this in mind and must explicitly adapt if the rate of sensor data reception diminishes. More advanced load balancing capabilities would help to decrease this potential performance bottleneck.

One shortcoming of the current implementation of the Load Balancer is that it performs poorly when many behaviors are started simultaneously. A simple solution to this problem might be to add a controlled amount of randomness to the load reported by each machine. If there were several suitable machines with approximately equal load, this strategy would distribute new components across the set of machines with relatively low load values.

Another area for future work involves determining which data is most useful to provide to the Load Balancer so that it can make better decisions about the best host on which to start a component. This may also involve allowing the core services to detect at runtime which pieces of information should be used to make this decision.

Further research is also needed to help the Load Balancer make its decision of the best host for a new component based on specific properties of that component. This would spread out the load even further, allowing computationally intensive components to be started on different workstations.

Acknowledgements

This material is based upon work supported in part by Microsoft Corporation, the National Science Foundation through grant #EIA-0224363, and 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. E. M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):57–78, Mar. 2001.

2. G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel Distributed Computing*, 7(2):279–301, 1989.
3. E. H. Durfee. Distributed continual planning for unmanned ground vehicle teams. *AI Magazine*, 20(4):55–61, 1999.
4. O. M. Group. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1998.
5. D. Kortenkamp, R. P. Bonasso, and R. Murphy. *Artificial Intelligence and Mobile Robots*. AAAI Press/MIT Press, 1998.
6. L. E. Parker. ALLIANCE: An architecture for fault tolerant multi-robot cooperation. *IEEE Trans. on Robotics and Automation*, 14(2):220–240, 1998.
7. P. Pirjanian, T. Huntsberger, A. Trebi-Ollennu, H. Aghazarian, H. Das, S. Joshi, and P. Schenker. CAMPOUT: a control architecture for multirobot planetary outposts. In *Proc. SPIE Conf. Sensor Fusion and Decentralized Control in Robotic Systems III*, pages 221–230, Nov. 2000.
8. P. E. Rybski, N. Papanikolopoulos, S. A. Stoeter, D. G. Krantz, K. B. Yesin, M. Gini, R. Voyles, D. F. Hougen, B. Nelson, and M. D. Erickson. Enlisting rangers and scouts for reconnaissance and surveillance. *IEEE Robotics and Automation Magazine*, 7(4):14–24, Dec. 2000.
9. J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, Boston, 1998.
10. S. A. Stoeter, P. E. Rybski, M. D. Erickson, M. Gini, D. F. Hougen, D. G. Krantz, N. Papanikolopoulos, and M. Wyman. A robot team for exploration and surveillance: Design and architecture. In *Proc. of the Int'l Conf. on Intelligent Autonomous Systems*, pages 767–774, Venice, Italy, July 2000.