In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*
pages 1226-1231, Wailea, Hawaii, 29 October - 3 November, 2001.

Most Valuable Player: A Robot Device Server for Distributed Control

Brian P. Gerkey Richard T. Vaughan Kasper Støy Andrew Howard
Gaurav S. Sukhatme Maja J Matarić

Robotics Research Labs, University of Southern California
Los Angeles, CA 90089-0721, USA

Abstract

Successful distributed sensing and control require data to flow effectively between sensors, processors and actuators on single robots, in groups and across the Internet. We propose a mechanism for achieving this flow that we have found to be powerful and easy to use; we call it Player. Player combines an efficient message protocol with a simple device model. It is implemented as a multi-threaded TCP socket server that provides transparent network access to a collection of sensors and actuators, often comprising a robot. The socket abstraction enables platform- and language-independent control of these devices, allowing the system designer to use the best tool for the task at hand. Player is freely available from `http://robotics.usc.edu/player`.

1 Introduction

Since 1999, the robots at the University of Southern California Robotics Research Labs have had on-board TCP/IP and 802.11 wireless Ethernet as standard equipment. The same is true of many labs around the world; today the cost, availability and ease of use of the equipment has put it within reach of most professional and academic users. Communication with and between robots in the lab is now cheap and easy. Better still, it supports the standard socket interface; the system that moves sensor data between processes on the robot's on-board computer will just as easily move them to the workstation across the lab or to the web page across the Internet.

We are using this equipment to provide transparent network access to all sensing and control of our robots. This paper describes our software *Player*, a network server interface to a collection of sensors and actuators, typically constituting a robot. Player has quickly become the most-used interface to the hardware in our lab.

There are three main motivations for providing a socket-based robot server:

Distribution: A client has access to sensors and actuators anywhere on the network. Clients can connect to multiple servers; servers accept connections from multiple clients. A single program could control the behavior of several robots; several programs could control different aspects of one robot's behavior. Section 2 describes a scenario which illustrates some possibilities of remote sensing and control.

Independence: Clients can be written in any language and on any hardware platform that implements sockets; most languages support this today. The user can choose the most appropriate language and environment for the task at hand; be it C for run-time speed, Java for ease of use, MatLab for algorithm prototyping, Perl for web integration, or Tcl/Tk for GUI design.

Convenience: The server provides an abstract unified interface to the devices attached to it. Client programs (robot controllers, sensor data processors, etc.) 'subscribe' to a set of devices and specify the frequency at which data should arrive. Data from the subscribed devices come as one data packet at the requested interval. Distribution adds to this convenience, for example enabling remote display and logging of robot state and sensor data. Some of the earliest Player clients were visualization tools for debugging.

While distribution provides the primary scientific benefit by enabling an interesting and little-explored class of algorithms, the independence and convenience are of practical interest to researchers and students. Player's ease of use make it attractive even with a single client running on-board a single robot. Player also interfaces with *Stage* (Section 6) to simulate a population of devices interacting in a virtual environment.

Player is primarily a protocol, as specified in the manual [4]. Any program implementing the protocol counts as a Player. Currently there is a single implementation; our Player program written in C++ using POSIX services. It has been tested on Linux and Solaris and should compile on any POSIX-compliant system.

Player was written to support our labs' research. Some similar system is a prerequisite for any exploration of dis-

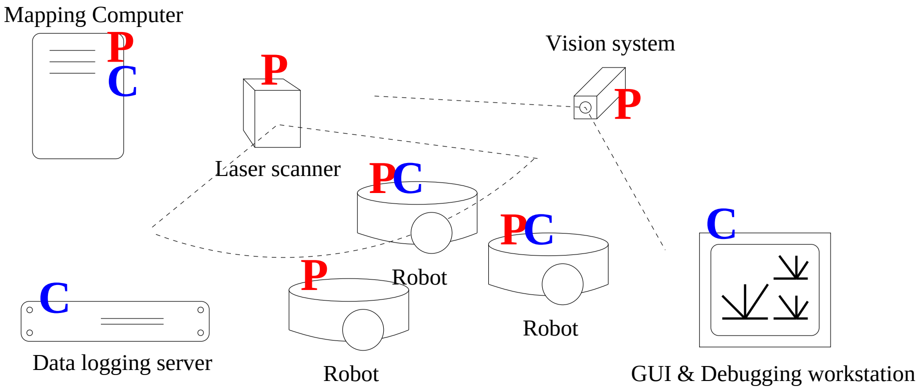


Figure 1: *Example scenario: Player servers (indicated with a 'P') distribute sensor data to clients (indicated with a 'C') across a network of wired and wireless connections.*

tributed sensing, control and coordination. The Player experiment has been so successful in-house that we believe it will be of immediate use to the robotics and sensor network communities.

2 Scenario

To illustrate how Player can be used to support distributed sensing and control in a variety of ways, we consider the following scenario, also shown in Figure 1: An experimental team of robots patrols a building at Western University, holding a tight formation. A formation-control program runs on one of the robots. It subscribes to the sonar range sensors and wheel motors on all three robots, sending wheel speed commands to maintain a fixed range and bearing between them. On another robot, a client subscribes to the audio devices of its host and its closest teammate, processing the two audio streams to find the direction of interesting sounds.

Meanwhile an experimenter is debugging the formation controller; she examines the robots' sonar readings from a GUI client running on her workstation. A logging server keeps a record of the robots' ground-truth positions for the experimenter's future paper. The logger subscribes to the tracking device running on an overhead vision system.

Simultaneously, a colleague at Eastern University with access to a supercomputer runs an on-line mapping client that subscribes to every available ranging device at Western U., including a wall-mounted laser scanner and the three robots' sonars. The mapping application is not controlling any actuators, so the 300ms transmission delay over the Internet is not a problem. However, the map generated is available online in case a Western robot should need it.

This scenario, while complicated, is fully supported by the current Player protocol and implementation. Further, we already have examples of many of these clients.

3 Related Work

Previous work in the area of robot programming interfaces has focused primarily on providing a development environment that suits a particular control philosophy. For example, Ayllu [13], which, like Player, can be used to control the ActivMedia Pioneer robots, provides tools for creating concurrent behaviors and, further, enforces a behavior-based control structure [1]. Similarly, COLBERT/Saphira [7], which can also control the Pioneer robots (among others), is concerned mainly with the construction of fuzzily-blended behavior-based control systems [10]. While such tools are very useful, we believe that implementing them at such a low level imposes unnecessary restrictions on the programmer, who should have the choice to build any kind of control system while still enjoying device abstraction and encapsulation. Thus in Player we make a clear distinction between the *programming interface* and the *control structure*, opting for a maximally general programming interface, with the belief that users will develop their own tools for building control systems. Further, most robot interfaces confine the programmer to a single language: Ayllu uses something akin to C, Saphira uses something akin to LISP, and TeamBots [2] uses Java. In contrast, the TCP socket abstraction of Player allows for the use of virtually any programming language.

The system that is most similar to (and certainly some inspiration for) Player is the TRIP server [6]; the main difference between the two is that whereas TRIP was designed as a sophisticated server to support extremely simple clients, we strove for minimalism in our server and simplicity in our message protocol, at the possible expense of causing the client to do more work.

Many other distributed device control and event service

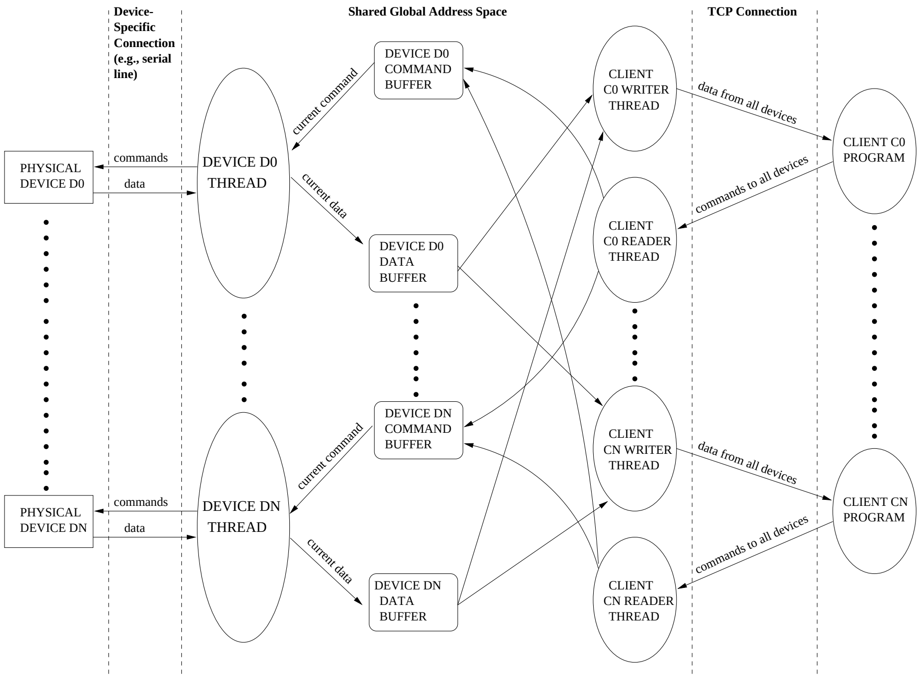


Figure 2: *Overall system architecture of Player*

systems have been developed, but we are not aware of one that provides the tradeoff between power and simplicity that makes Player well-suited to multi-agent robotic device control.

4 Architecture

Player's development was guided by our desire to concurrently support many heterogeneous devices and many heterogeneous clients. Each device operates at some inherent frequency, with wide variation among devices. For example, the popular SICK LMS 200 laser range-finder returns a full scan at approximately 5Hz, while the Sony EVID30 pan-tilt-zoom camera can give encoder feedback at almost 2500Hz. Similarly, each client operates at some inherent frequency; while a simple client written in C++ may be capable of consuming new data at 100Hz, a graphically intensive client written in Tk might operate at less than 1Hz. We want to move data and commands between clients and devices at the highest rate possible in order to fully exploit the hardware and maximize the responsiveness of the system.

4.1 Server Structure

Given the requirement to support interaction with external entities (i.e., clients and physical devices) that operate at different timescales, we designed Player in the standard model of an asynchronous threaded server. Player is implemented in C++ and makes extensive use of the POSIX-compliant `pthread` interface. A main thread listens for new client connections on a well-known TCP port, spawning threads on demand to service clients and the devices they request. The overall system structure of Player is shown in Figure 2. The center portion of the figure is Player itself; on the left are the physical devices and on the right are the clients. Each client has a TCP socket connection to Player. If the client is executing on the same host as Player, then this socket is simply a loopback connection; otherwise, there is a physical network in between the two. At the other end, Player connects to each device by whatever method is appropriate for that device (e.g., RS-232).

Within Player the threads communicate through a shared global address space. As indicated in Figure 2, each device has associated with it a command buffer and a data buffer. These buffers provide an asynchronous communication channel between the device threads and the client reader and writer threads. For example, when a client reader thread receives a new command for a device, it writes the command into the command buffer for that device. Later, when the device thread is ready for a new command, it will read the command from its command buffer and send it on to the device. Similarly, when a device thread receives new data from its device, it writes the data into its data buffer. Later, when a client writer thread is ready to send new data from that device, it reads the data from the data buffer and passes it on to its client. In this way, the client service threads are decoupled from the device service threads (and thus the clients are decoupled from the devices). Also, by the nature of threads, the devices are decoupled from each other, and the clients are decoupled from each other.

4.2 Device Model

In order to provide a uniform abstraction for a variety of devices, we chose to follow the UNIX model of treating devices as files. Thus the familiar file semantics hold for Player devices. For example, to begin receiving sensor readings, the client opens the appropriate device with `read` access; likewise, before controlling an actuator, the client must open the appropriate device with `write` access[1]. As this model has served UNIX-like operating systems well for many years, we expect that it will suffice for the devices the Player will control in the future.

Player currently supports a number of devices, including the popular Pioneer research robot and various peripherals (for a complete list, see the Player home page, listed in Section 8). We have also introduced the concept of *virtual* devices. Rather than being tied directly to a piece of hardware, a virtual device performs aggregation and processing on data gathered from one or more other sensors, and exports the result. By integrating useful algorithms directly into Player, we can easily share expertise and reuse code throughout the lab (and the world).

4.3 Client Interaction

By default, clients receive data at 10Hz[2]. Thus, every 100ms, a client can expect to receive a data packet containing the current data from all the subscribed devices. Of course, by sending all the data at once, Player might repeatedly send old data from a device that operates at less than 10Hz. We designed Player in this way for one reason: simplicity. By always transmitting the current state for all subscribed devices, regardless of the timescale of the device, we facilitate the writing of client programs. As a result, clients are able to use a simple blocking read loop to receive data from Player. If the client is multi-threaded (many are), the blocking read could be compartmentalized to a single thread, allowing the rest of the client program to proceed unhindered.

Of course, receiving data at 10Hz may not be reasonable for all clients; for these situations, we provide a method for changing the frequency, and also for placing the server in a request/reply mode. So, if a client wants vision data at full frame rate, it can configure Player to send data at 30Hz, with the tradeoff that it will also receive (sometimes repeated) data from the other currently requested devices at the higher rate. Alternatively, if there is a low-bandwidth connection between Player and a client using laser data (which is comparatively large), that client might lower the data rate to 5Hz in order to minimize message-passing and thus conserve bandwidth, with the tradeoff that data from other requested devices will also arrive more slowly[3]. It is important to remember that even when a client receives data slowly, there is no backlog and it always receives the most current data; it has simply missed out on some intervening information. Also, these frequency changes affect the server's behavior with respect to each client individually; the client at 30Hz and the client at 5Hz can be connected simultaneously, and the server will feed each one data at its preferred rate.

Analogous to the issue of repetition of old data is the fact that there is no guarantee that a command given by a client will ever be sent to the physical device. Player does not implement any device locking, so when multiple clients are connected to a Player server, they can both write into a single device's command buffer. In general, there is no queuing of commands, and each new command will overwrite the old one; the service thread for the device will only send to the device itself whatever command it finds each time it reads its command buffer. We chose not to implement locking in order to provide maximal power and flexibility to the client programs. In our view, if multiple clients are concurrently controlling a single device, such as a robot's wheels, then those clients are probably cooperative, in which case they should implement their own arbitration mechanism at a higher level than Player. If the clients are not cooperative, then the subject of research is presumably the interaction of competitive agents, in which case device locking would be a hindrance.

5 Evaluating the implementation

To evaluate the performance of our Player implementation, we performed a series of stress tests[4]. A number of simultaneous connections were made to a Player server, and data were requested from a typical set of devices. We performed 54 experiments, testing all combinations of the following parameters:

Number of clients $\{1, 10, 50\}$
Client update frequency $\{1, 10, 50\text{Hz}\}$
Data size $\{$small (85 bytes), large (807 bytes)$\}$
Network type $\{$loopback, Ethernet, 802.11$\}$

For each of the 54 parameter combinations, we ran the system for 2 minutes and measured the end-to-end data packet latency[5] and time interval between arriving data packets for each connected client. End-to-end latency is the sum of

---

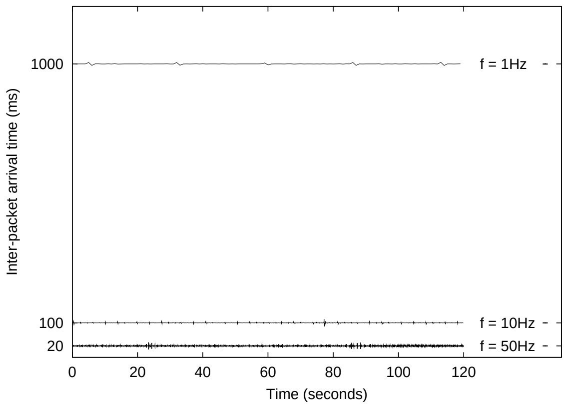
[1] Devices in Player can also be configured similarly to `ioctl()`.

[2] We chose 10Hz because most of our currently supported devices operate at or near that frequency.

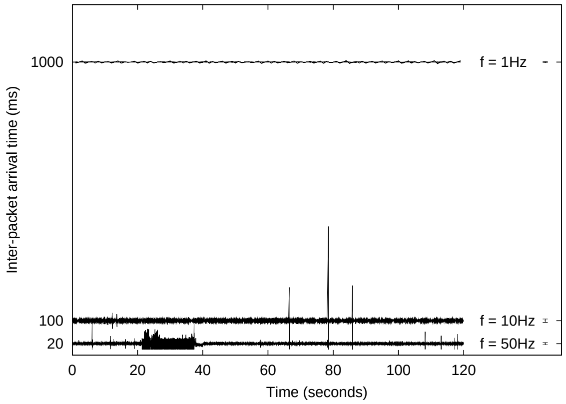
[3] If a client requires different data rates from devices, it can simply make separate connections to the server, one for each device.

[4] These tests were conducted with Player version 0.8d.

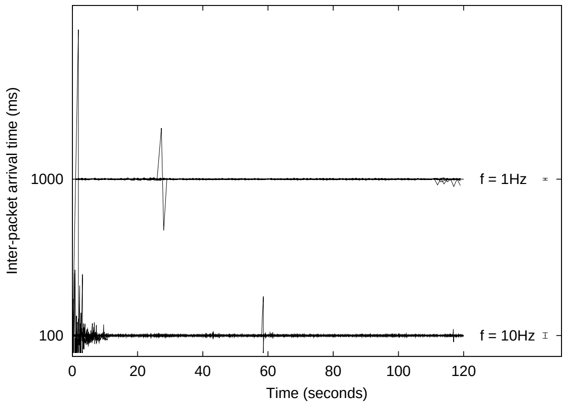
[5] The test computers' clocks were synchronized with the network time protocol (NTP) [8], which can maintain precision of less than 1ms.



(a) Population: 1
f = 1Hz : $\mu = 1000.05$, $\sigma = 1.41$
f = 10Hz : $\mu = 100.00$, $\sigma = 1.05$
f = 50Hz : $\mu = 20.00$, $\sigma = 1.12$



(b) Population: 10
f = 1Hz : $\mu = 1000.05$, $\sigma = 1.80$
f = 10Hz : $\mu = 100.00$, $\sigma = 5.73$
f = 50Hz : $\mu = 19.96$, $\sigma = 3.89$



(c) Population: 50
f = 1Hz : $\mu = 1000.02$, $\sigma = 5.70$
f = 10Hz : $\mu = 99.61$, $\sigma = 16.05$ (16 clients)
f = 50Hz : NO DATA (0 clients)

Figure 3: *Inter-packet arrival times measured while serving large amounts of sensor data across wireless Ethernet to different client populations.*

network latency and Player-induced delay due to the asynchronous client and server loops. This delay is bounded above by the lesser of the update intervals of the client and device; the expected value of the delay will be half the lesser interval. A more interesting metric is the inter-packet arrival time, as it captures Player's ability to maintain a requested data rate; if a client asks for updates at 10Hz, then, if Player is working properly, the client will receive a data packet every 100ms.

In all but the extreme case described below, Player was able to track the desired data rate very closely. The mean interval between packets was always $< 50\mu s$ from the target interval (less than our clock precision).

Player is therefore generally able to provide data in a timely manner. However, we observed that in the toughest case we examined, transferring large packets via 802.11 (of the networks tested, the wireless has the least bandwidth [at ≈1.9Mbps] and greatest overhead), Player was unable to support the larger populations of subscribers. In these cases some or all of the clients' connections were broken by Player. Due to lack of space we present here only these *worst* performing results. Figure 3 plots the measured time interval between packets received. For a population $P = 1$ client, Player is able to maintain 20, 100 & 1000ms intervals with little variance (Figure 3(a)). For $P = 10$ the 1000ms interval is maintained as before. The 100 and 20ms intervals are also maintained as means, but increased variance is observed (Figure 3(b)). For $P = 50$ the 1000ms interval is maintained, but again with relatively large variance. For the 100ms interval, Player was found to close connections to 34 of the 50 clients, leaving only 16 connected which it served at 100ms with high variance. When 50 clients connected requesting 20ms update rates, Player closed all connections within a few seconds and no data were served (Figure 3(c), no plot for 20ms interval).

These results allow us to determine rough upper bounds for Player's performance. For example, the server is unable to handle 50 clients requesting laser data at 10Hz across the wireless network, but it will work across the 10Mbps wired Ethernet. Our initial estimates of the bandwidth requirements of this service seem to show the wireless is sufficient. Our next step is to examine these failures more closely to determine if we are up against a hardware/OS limit, or whether our implementation can be improved.

6 Stage

Stage simulates a population of Player devices, allowing development and testing of clients in an environment very similar to that provided by the real hardware. Stage spawns several copies of Player, replacing the real device drivers with its simulated equivalents. The user interfaces with Player in the normal way; clients see the identical interface to real and simulated hardware.

We have found that agents developed in simulation will work with little or no modification on the real devices and vice-versa. The Stage distribution includes a variety of environments suitable for large and small-scale experiments in multi-agent sensing, communications and control.

7 Usage

Player is the default interface for the Pioneer robots at USC and has been used for many projects. Incoming graduate students write their first single-robot controllers for it, and it is used with Stage for graduate courses in robotic sensing and planning,

Some examples of our research projects using Player (and its precursor *ArenaServer*) are ant-inspired trail-following in robot teams [11], cooperative box pushing and multiple target tracking [3], reducing interference in teams by aggressive behavior [12], investigation of interaction between network and behavior designs [14], simultaneous localization and mapping [5], and online resource allocation [9].

8 Conclusion and future work

In this paper we have attempted to make explicit some opportunities presented by ubiquitous network communications for robotics. We have identified a niche for a novel piece of software that provides language and platform independent network access and an abstract interface to collections of sensors and actuators: a *Robot device server*. Player is our candidate design for such a server. We have described it here and made the source and documentation available for evaluation by the community.

Our Player implementation is deliberately simple and is based on the well-understood multi-threaded blocking server design. A modular device driver interface makes adding additional devices straightforward.

We are working on an implementation for the QNX real-time operating system for use with the USC AVATAR robot helicopter. We also aim to create a lightweight, low memory, low thread-count version for handhelds and small embedded devices. The Player protocol and implementations will evolve as we push them with more complex clients, additional I/O devices and smaller server platforms.

Player and Stage distributions, including source code, example client programs and documentation are freely available under the GNU General Public License at the Player home page: `http://robotics.usc.edu/player`.

The acid test of Player will be its uptake. We hope that Player will develop over the next few years into a well-used tool. It will not suit every application, but it has proved useful in a variety of roles in our labs. Our work would be less fun without it.

Acknowledgments

This work is supported by DARPA contract DAAE07-98-C-L028, DARPA grant DABT63-99-1-0015, JPL contract No. 1216961, NSF grants ANI-9979457 and ANI-0082498 and ONR grants N00014-00-1-0140, N0014-99-1-0162 and N00014-00-1-0638.

Many thanks to Boyoon Jung, Jakob Fredslund and Esben Østergård for their testing, fixes and ideas.

References

[1] Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.

[2] Tucker Balch. *Behavioral Diversity in Learning Robot Teams*. PhD thesis, College of Computing, Georgia Institute of Technology, 1998.

[3] Brian P. Gerkey and Maja J Matarić. Principled communication for dynamic multi-robot task allocation. In D. Rus and S. Singh, editors, *Experimental Robotics VII, LNCIS 271*, pages 353–362. Springer-Verlag Berlin Heidelberg, 2001. (ISER 2000).

[4] Brian P. Gerkey, Kasper Støy, and Richard T. Vaughan. Player robot server. Technical Report IRIS-00-392, Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, November 2000.

[5] Andrew Howard, Maja J Matarić, and Gaurav S. Sukhatme. Relaxation on a mesh: a formalism for generalized localization. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1055–1060, Wailea, Hawaii, October 2001.

[6] James S. Jennings. Threaded servers enable thin robot clients. Technical report, Computer Science Dept., Tulane University, 1998.

[7] Kurt Konolige. COLBERT: A language for reactive control in saphira. In *Proceedings of the German Conf. on Artificial Intellgence*, pages 31–52, Freiburg, Germany, 1997.

[8] David L. Mills. RFC 1305: Network time protocol (version 3) specification, implementation and analysis, March 1992.

[9] Esben H. Østergård, Maja J Matarić, and Gaurav S. Sukhatme. Distributed multi-robot task allocation for emergency handling. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 821–826, Wailea, Hawaii, October 2001.

[10] A. Saffiotti, E. H. Ruspini, and K. Konolige. Blending reactivity and goal-directedness in a fuzzy controller. In *Proceedings of the IEEE Intl. Conf on Fuzzy Systems*, pages 134–139, San Francisco, CA, 1993.

[11] Richard T. Vaughan, Kasper Støy, Gaurav S. Sukhatme, and Maja J Matarić. Blazing a trail: insect-inspired resource transportation by a robot team. In *Proc. of the Intl. Symp. on Distributed Autonomous Robotic Systems (DARS)*, pages 111–120, Knoxville, TN, 2000.

[12] Richard T. Vaughan, Kasper Støy, Gaurav S. Sukhatme, and Maja J Matarić. Go ahead, make my day: Robot conflict resolution by aggressive competition. In *Proc. of the Intl. Conf. on Simulation of Adaptive Behavior (SAB)*, pages 491–500, Paris, France, 2000.

[13] Barry Brian Werger. Ayllu: Distributed port-arbitrated behavior-based control. In Lynne E. Parker, George Bekey, and Jacob Barhen, editors, *Distributed Autonomous Robotic Systems 4*, pages 25–34. Springer-Verlag, Knoxville, Tennessee, October 2000.

[14] Wei Ye, Richard T. Vaughan, Gaurav S. Sukhatme, John Heidemann, et al. Evaluating control strategies for wireless-networked robots using an integrated robot and network simulation. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 2941–2947, Seoul, South Korea, 2001.