Broker: An Interprocess Communication Solution for Multi-Robot Systems

Matthew McNaughton
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mcnaught@cs.ualberta.ca

Sean Verret
Defence R&D Canada
DRDC Suffield
Suffield, Alberta, Canada
sean.verret@drdc-rddc.gc.ca

Andrzej Zadorozny
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, Canada
andrzej@cs.ualberta.ca

Hong Zhang
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, Canada
zhang@cs.ualberta.ca

***Abstract*— We describe in this paper a novel implementation of the interprocess communication (IPC) technology, called *Broker*, in support of the development and the operation of a complex robot system. We view each robot system as a collection of processes that need to exchange information, e.g. motion commands and sensory data, in a flexible and convenient fashion, without affecting each other's operations in case of a process's scheduled termination or unexpected failure. We argue that the IPC technology provides an ideal framework for this purpose, and we carefully make our design decisions about its implementation based on the needs of robotics applications. *Broker* is programming language, operating system, and hardware platform independent and has served us well in a RoboCup project and collective robotics experiments, in both simulation and real-world environments.**

I. Introduction

Examples abound in robotics in which the need arises for robots within a group or for the components within a single robot, e.g. a sensor module and a control program, to exchange information seamlessly, a capability that is often critical for both the development and the operation of these robot systems. A flexible and transparent communication infrastructure facilitates design modularity and enhances the fault-tolerance of the robot systems. Interestingly, the same need has long existed in networked computer systems in which processes running on a single or multiple computers must communicate to share the information required to function as an integrated system. The solution in that case is what is known as interprocess communications (IPC). In this paper, we will describe our implementation of an IPC concept, which we refer to as *Broker*, and its use in our development and deployment of a multi-robot system designed to compete in RoboCup.

Our multi-robot system (see Figure 1) has the ability to run eight physical robots, under the control of a multitude of processes. All these processes have the ability to share information with each other through *Broker*. To achieve a high level of collective intelligence and to enhance system effectiveness, it is necessary for the processes to communicate and share information with ease. These processes which may be, running on different operating systems, implemented in different programming languages, compiled on different development platforms, all require the ability to "talk" with each other. It is this requirement that presents the interesting link between a multi-agent system and IPC in a networked computer system.

Fig. 1. The robotic system in action. Initially designed to play robot soccer [15], it can monitor and control up to 8 robots.

Our team of robots have very limited sensing capability on board. Their vision is, for example, shared through an overhead camera, as is the master radio for command transmission to the robots. Each robot is equipped with a CPU and receives motion instructions over a wireless link by AI (artificial intelligence) processes running off-board. Because several people may be concurrently involved in the development of the system components, it is highly desirable for the failure of any component not to affect the rest of the system. In addition, we need a flexible and extensible data format in order to handle the constant evolution of the semantics of the communicated messages and the addition of system components and data types associated with the components. Finally, message transmissions must be of low latency, although no retransmission is attempted due to dropped packets in most cases. *Broker* has succeeded in meeting all but one of the above constraints and requirements. The *Broker* server process itself is a single point of failure. However, the implementation is very simple and stable, reducing the risk of catastrophic failure. This paper explains its design considerations and implementation, and serves as a useful reference for robot system designers faced with the same set of challenges.

The remainder of the paper is organized as follows. Section II surveys existing IPC systems designed specifically for robotic applications. Section III discusses the development

process of *Broker*. Our use of *Broker* in a multi-robot system is described in Section IV, which describes in detail the convenience provided by *Broker* for sharing information. Finally, Sections V and VI discuss the positives and negatives of *Broker*, and conclude the paper.

II. Related Work

We have developed an IPC system for robotics that is designed to facilitate seamless development and integration of a robotic system with many components. In this section we review several other IPC systems used in robotics.

The Object Oriented Toolkit for Inter-process Communication (IPT) [7] was the first IPC system specifically for robotics, followed by the Inter-Process Communications toolkit (CMU-IPC) [17]. CMU-IPC follows the message based paradigm and exchanges messages either through the IPC server or on a peer-to-peer basis. CMU-IPC can be implemented using several languages, on different machine types and operating systems [14]. We have found two flaws with CMU-IPC. First, CMU-IPC takes over the main program event loop and it cooperates poorly with other libraries such as Qt. Second, CMU-IPC does not have a simple, fixed message format.

Real-Time Communications (RTC) was designed as IPC middleware to provide real-time capabilities specifically for robotic applications [13]. RTC is written in C/C++ [3], is message based, and features a server for registering module names. RTC was designed for high-bandwidth, point-to-point connections over TCP and for load balancing CPU-intensive robotics tasks such as LADAR data analysis. RTC and CMU-IPC share the same marshalling API.

The following four systems are also worthy of mention. The Network Data Distribution System (NDDS) implements a publish-subscribe model to perform network operations for any number of data-sharing processes [5]. The Message Passing Interface (MPI) [4] was developed as an IPC toolkit optimized for parallel computing problems. The Adaptive Communications Environment (ACE) [10] began as a C++ platform-abstraction library. The Common Object Resource Broker Architecture (CORBA) [8] is not a toolkit but instead a specification of an architecture or infrastructure such that computer applications can cooperate together over a network [1], [9] which is inherently what IPC tries to achieve. Finally, various frameworks using their own methods of IPC have been developed like [6], [18], and [2].

All were developed to ease data sharing and to enable an efficient, robust system that was still flexible and near real-time. *Broker* was created with a focus on developer convenience, whereas CORBA was not. Building robotic systems is difficult and every piece of the system must be reliable. The more fundamental its role, the more reliable it must be. The robotic system must ease problem diagnostics and program development. With *Broker*, we have stayed away from the application control flow yielding the widest variety of OS/language choices. We do not have specific libraries for marshalling/unmarshalling data. Instead we have specified a very stable packet format. In the next section we describe *Broker*, our solution to robotic IPC.

III. IPC Design and Mechanism

Existing IPC systems are arbitrarily chosen points in the IPC design space, cementing design decisions along several independent design axes into a single implementation. The major axes are:

- Language API - locus of main program control flow, data marshalling style.
- Network connectivity graph. (e.g. hub vs. peer-to-peer)
- Message based vs. stream based.
- Data serialization format.
- Service location protocol.
- Anonymous publish-subscribe vs. peer-to-peer remote procedure call.

The ideal IPC system would be a flexible toolkit allowing the developer to choose and combine functionality at any desired point along each axis. No existing IPC system allows that but some allow a few choices. *Broker* does not reach the ideal either. It embodies a principled design made with the *convenience of developers* as a primary goal. Certain design choices exclude or enable other desirable design targets. Desirable design targets are to be small, lightweight, and convenient to integrate into a system. These qualities are enabled or made impossible by the design choices made along the major axes listed above. An IPC system is not difficult to develop. However, it is much more challenging to demonstrate its utility. The true test is this: is the system easy and convenient to use? If a developer balks at the complexity of learning and integrating an IPC system and chooses instead to implement his own, then regardless of how robust and flexible the system is, it is a failure. In the RoboCup Small-Size League, we are aware of no team that makes use of any published IPC system. This is an indictment of their obscurity and complexity. For example, the code CMU published in 2002 [16] divides responsibility among multiple processes that communicate via UDP and TCP, but each program is written to directly contact the processes they need to communicate with. Communication is accomplished by writing C structs directly into UDP packets. Other teams approach IPC in a similarly ad-hoc manner, or simply compile all functionality into a single process.

With the convenience of developers being the most important consideration, certain design choices become obvious. For example, the system should be designed, as *Broker* is, to allow any component of the system, including the central server, to abort and resume at any time without causing disruptions to the rest of the system.

A. Network Topology

A *Broker* network is centralized around a single server process called the broker, as illustrated in Figure 2. All communication is sent in the form of UDP packets which flow only between clients and the broker. For a client to join the *Broker* network, it is given the IP address and UDP listening

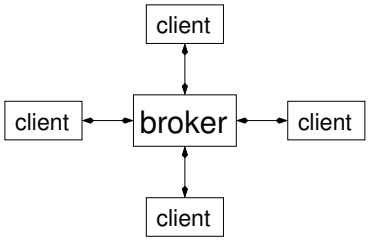

Fig. 2. The hub architecture of *Broker* in which processes (software agents) communicate via a central packet relaying mechanism.

port of the server process. Clients send request packets to the server process to subscribe to any of several defined data channels. If the client does not receive data on the channel, it periodically resends the request. This ensures that the system can regain stability if the broker server process should die and be restarted. Other valid requests are to unsubscribe from one or all channels, request system status information, or set the minimum priority required for a message to be repeated on any channel. This latter ability is used only by the system console. Clients may also publish data on a defined channel by sending packets to the server, which repeats the packets to all subscribed clients. The server does not attempt to arbitrate contention between processes that are publishing contradictory messages on the same channel.

Since the system is arranged around a central server, we are able to capture logs of all communications sent by clients for later replay and analysis.

B. Message Format

The most important requirements of the data packets themselves, in order of importance are:

1) Observability/Debuggability. The packets must be easy for a human to observe and find implementation errors in transmitting agents. It must also be easy for a human to "forge" a packet of any format quickly and inject it into the system.
2) Speed. The packets must be rapidly written and read.
3) Ease of writing parsers. It must be easy to write a packet parser in any programming language. It must be easy to write a packet generator in any language.
4) Flexibility. Changes to the semantics of a packet type must not impact the parsers.

These requirements determined the packet format. The observability requirement strongly suggests a textual format. One could write tools to display and edit a binary packet format but the additional coding effort may not be worth it. It is especially difficult to write display and edit programs that operate on and diagnose incorrectly formatted packets. The third requirement also suggests a textual format, since it can be a challenge to write binary parsers. Adding new optional data fields to a simple binary dump of a C-struct necessitates updates to all parsers. The CMU-IPC format, for instance, has the same flaw as all binary formats, i.e. the parser is vulnerable to packet corruption and can bring down the whole process. This can happen if, for example, an element count field for an array is corrupted, causing the parser to access invalid memory.

The speed requirement argues that the format must be simple and contain very little excess verbiage. This rules out XML. We tried XML and found that even with a custom parser that dispensed with all of XML's complex semantics, it was noticeably slower than the textual format we settled upon. The obvious format for the fastest textual message would dispense with all field and attribute names, but this would break requirements for observability and flexibility. Both the requirements for observability and flexibility require some form of field naming.

With the above considerations in mind, we settled upon a simple textual message format, e.g.:

```
@ <vision_out> 0
ball x 5.3 y -2.5
blue id 0 x 15.2 y -98.3 xh 1.0 yh 0.0
blue id 4 x 98.4 y -87.2 xh -0.707 yh 0.707
yellow id -1 x 139.5 y 167.2
latency millis 152
```

The first line is a header, naming the channel the packet is being published to. The integer value (0 in this example) defines the message priority. The broker can be instructed to ignore all messages below a given priority. The remaining lines, not examined by the broker, contain message data. Each line consists of textual tokens. The first token on each line is a distinguished label. There may be any number of lines starting with the same token. The packet is like a struct, with each line a component structure. Multiple lines starting with the same token may be interpreted as an array. The remaining tokens on a line are name-value pairs. Each of these is a field in the line struct. Arrays are not allowed on a line, that is, a line may not contain more than one name-value pair with the same name. As with XML, the parser can parse the packet without knowing the semantics of the line-start tokens or the named fields on each line. Normally, a packet may contain any number of lines (up to 65kB, the maximum capacity of a UDP packet). The size of a packet is a semantic issue, not syntactic. In our RoboCup system, "blue" and "yellow" indicate the team affiliation of a robot, and the number of lines of each type may vary from frame to frame.

A major design option our format does not provide is arbitrary nesting of structures. The message semantics we were attempting to send initially did not require nesting. There is the question of whether we would have used nesting were it convenient to do so. Regardless, nesting should be easy to add without falling into the XML trap of excessive verbosity and redundant syntax. Simple parenthesization ought to suffice.

If a line consists of a single field, then there are two attribute names required, the first naming the line, the second naming the field. If there are many such lines, it could result in significant waste. However, there is a workaround available: simply establish a "misc" line type and aggregate all the single-attribute line types into it.

C. Marshalling API

The beauty of *Broker* is that it does not impose a strict API. The simplicity of the message packet format makes creating packets as easy as, for example, using *printf* in the C language, and renders it instantly accessible to basically all languages. Parsing is as easy as using *scanf* in C, or regular expressions. Our code base provides convenient packet parsing functions in Java and C. By contrast, other systems such as CMU-IPC require the developer to define a *C*-struct for each message type they wish to send or receive. In order to send or receive the packet, the developer must additionally provide a specially formatted string which describes the layout of the struct. CORBA provides a language (IDL) for describing message structures to be marshalled which in turn requires a special compiler to interface with the host language. Automatic data marshalling/unmarshalling APIs are ungainly. The design of choice for a lightweight IPC system is to force a manual approach and provide a library to make this convenient.

D. Program Control Flow

A typical robotics program runs in an infinite loop that accepts input, processes it, and sends output. Libraries that perform two-way IPC, which include not only robotic IPC systems such as *Broker*, but also windowing system libraries such as *Xlib*, require the application programmer to integrate their main program event loop with the library's loop. This poses a difficulty for programs that use both an IPC system for robotics communication and another indirectly through their windowing library. These libraries often demand that the main program loop be formed around them. As a way to cooperate with the developer's other asynchronous I/O connections, they occasionally allow one to pass in a file handle and receive notification events when there is data available. However, the correct design is for the library to make its file handle available and depend upon the developer to invoke the library when data is available. The take-home design message is: *always give the developer control over the main event-handling loop*.

E. Other Considerations

There is one disadvantage to the publish-subscribe model we chose for *Broker*. The needs of a remote procedure call (RPC) appear not to be well served. However, it is easy to build an RPC framework on top of the publish-subscribe model. The developer may establish a dedicated RPC channel and clients who wish to receive RPC calls may publish their existence on the channel. Clients may then make calls by publishing their requests, or sending a request packet directly to the RPC target client.

Broker depends on fast, local connectivity between all components of the system. The server process sends a copy of each message published on a channel to each subscribed client. If all clients resided on the opposite end of a bandwidth-limited connection, this behavior might overwhelm the capacity of the link. One solution is to allow a system to contain multiple server processes, one for each high-speed domain. They would forward only one copy of each message along each low-bandwidth channel. Client implementations would not have to change and would simply connect to the server process within their local networks, ignorant of any other server processes.

Broker has good communication latency, but does not achieve the theoretical minimum. Each packet must make two hops: from the publisher to the broker and from the broker to each subscriber. On a fast local network, the performance hit for this is low. In addition, the packet body text must be generated from data in memory within the publisher, and subsequently parsed by the subscriber. Any conceivable packet format that is not a raw write of memory will have some parsing overhead. It was not our goal with *Broker* to achieve the lowest possible latencies and CPU overheads, however, we did require that it be negligibly low (e.g. transmission latencies of less than one millisecond) while possessing the ability to parse multiple megabytes of data per second.

The purpose of *Broker* was to have acceptable performance while being easy to use. Due to time constraints we were not able to reimplement our system using other IPC toolkits for the purpose of comparison.

IV. Distributed Robotic System

In the previous section we described the design parameters of IPC systems in general and the design of *Broker* in particular. In this section we describe the distributed robotic system that we have developed for conducting experiments in cooperative and adversarial collective robotics and the role *Broker* plays in it. The system has been used to compete in the RoboCup Small-Size League [15] and for experiments in collective sorting [19], [20].

Our robots are approximately 180 mm in diameter and 140 mm tall, and their work area is a flat, enclosed pen approximately 2 by 3 meters. Additional hardware includes multiple fixed cameras offering a global view of the robot work area, a radio transmitter, and several PCs. Software modules include vision analysis, wireless communications, agent processes, system monitor, miscellaneous system control inputs, and the *Broker* server process (see Figure 5).

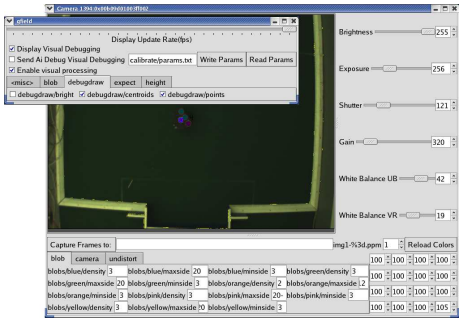

Fig. 3. Vision analysis console [12].

A. Vision processing

The vision system [12] includes two Point Grey DragonFly cameras mounted above the robot work area, providing global

vision input. Each camera provides a 640x480 colour image, via an IEEE1394 interface, directly to a PC. Each pixel is colour-classified and contiguous colour regions are identified. Robots are marked with unique identifying colour patches that also indicate their orientation. Other objects on the field are also marked with colour. Robot positions are then corrected by an internal camera distortion model and camera pose estimation algorithm provided by the Intel OpenCV library [11]. Finally, world coordinates of sensed objects in each frame of video are published along with ancillary information, such as the vision processing latency, which is used in predictive motion control.

The vision system uses a graphical monitor (Figure 3) to show the camera view, and allows an operator to optionally view superimposed debugging information to diagnose performance problems. In addition, the operator may also control any parameter of the vision system, such as camera shutter settings, colour lookup tables, or processing modes.

B. AI processes

An AI process subscribes to all available inputs from the sensors of the system. In our system, this information is limited to vision and status transmissions sent back over the radio link from the robots. The AI process then rapidly makes a decision and publishes commands for the robot or robots it is assigned to control. The wireless radio manager subscribes to the radio command channel and repeats the messages over the wireless link. AI processes can run on any networked CPU.

C. Radio transmitter

Our present radio hardware is a Linx Technologies HP-Series operating at 902-928 MHz with 8 available channels. There is one main base transmission station and one receiver module on each individual robot. The radio transmitter process forwards motion commands published by the AI processes.

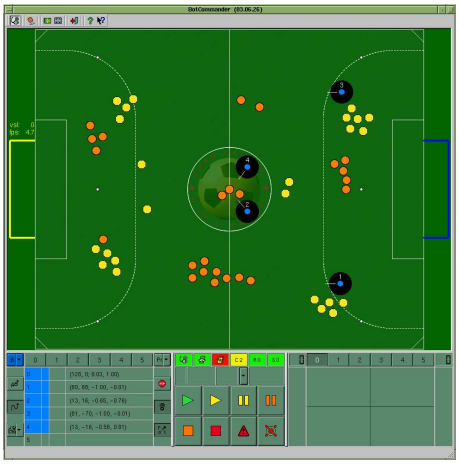

Fig. 4. The main system monitor has the ability to take control of all the robots in the system either separately or as a group. More importantly a single user has the ability to monitor the entire system from a single PC using the system monitor.

D. System monitor

The system monitor provides a graphical view of system status. It displays vision system output, i.e. robot positions in world coordinates, alongside status updates sent back by robots, and superimposed graphics sent by AI processes indicating intentions and attempted movements. The system monitor also allows the operator to override AI processes and take control of robot motion.

E. Data Flow

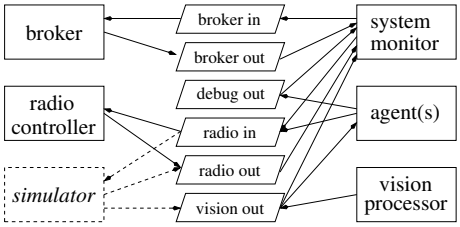

Fig. 5. Information flows throughout the system. When the simulator is active, it replaces the vision and radio modules.

Figure 5 shows the flow of data between the system components. Rectangles represent processes in the system. Rhomboids represent data channels. An arrow pointing from a process to a channel means the process publishes packets to the channel. An arrow pointing from the channel to the process means the process subscribes to the channel and receives all packets sent to it by other processes. The simulator is inactive when the vision and radio modules are active. The flexibility and robustness of *Broker* allows for the operator to kill both the radio and vision systems, and then run the simulator. It can do all of this without restarting other system components such as the agents or system monitor.

V. Discussion

In this section, we will discuss the higher-level design advantages of using *Broker*. We will also discuss its strengths and weaknesses.

A. Simultaneous Development

Our RoboCup development process is intense, with multiple developers working on the code base simultaneously. Each programmer could be aborting and restarting his processes frequently while it is actively transmitting to and receiving data from system processes run by others. Development would have come to a complete standstill if any intervention was required by other programmers to re-initialize their processes. Any process, including the broker server process, can terminate at any time without permanently disrupting other processes. Once the terminated process is replaced, communications resume smoothly and transparently.

This design requirement strongly indicates a stateless message protocol (i.e., when designing packet semantics, it must be ensured that it is possible to correctly interpret a packet without having seen any previous packet). *Broker* provides this functionality.

B. System Design

The correspondence between a physical robot and a process need not be one-to-one. Several processes may correspond to different aspects of a robot, (e.g. its radio, its camera, or its wheels). However, it is also possible that a single process may represent all the capabilities of several physical robots. The correspondence is arbitrary. Responsibilities should be partitioned among the processes so that communication resources, CPU resources, and latency requirements are all optimized.

C. Implementation Advantages

The simplicity of the *Broker* packet format and API make it trivial to extend it to any operating system (e.g. Linux, OpenBSD, Windows) or library. It can run on several OS's with UDP sockets and any language with basic string parsing functions. No special libraries are required. We have implemented *Broker* client libraries in C, C++, Java, Python, and Perl.

Broker does not have a complex implementation so there is no need to reimplement code when porting to a new language. Scripting languages typically allow the developer to link in "native" compiled machine-code but this causes build system and installation headaches. Whenever possible, it is better to avoid using libraries that require this style of scripting language integration.

VI. Conclusion and Future Work

In this paper, we have demonstrated the utility of an IPC concept for handling communication and sharing information and resources within a complex robot system which may have a number of software and hardware modules under the control of separate processes. Upon reviewing the IPC literature and the design constraints for a typical robot system, we have implemented our own IPC system, *Broker*, which has successfully served as the communication infrastructure that was required for the RoboCup project. The system can work equally well in both a simulation and real-world environment.

Broker has several key characteristics that prove to be extremely valuable. First, it provides a communication channel among any pair of processes within the system with a simple API and yet, at the same time, prevents the processes from affecting each other in case of a process failure. Secondly, *Broker* uses a text-based message format which makes it easy to develop parsers and which can be easily modified and extended to handle changing communication needs. Finally, since *Broker* is based on a standard network protocol (UDP in our case), it is capable of linking processes written in any language and running under any OS that support this standard protocol.

One deficiency of our current implementation of *Broker* is its scalability with respect to the number of processes that run in the system. We believe that this deficiency can be addressed with simple technical measures using, for example, multicast to send packets rather than multiple unicasts so that the processes or agents can communicate directly. *Broker*'s role will change to that of a directory to set up the initial connections. Another potential problem is the system's dependence on *Broker*, which introduces a single point of failure. This can be mitigated by the similar measure, i.e., by allowing processes to talk to each other directly upon detecting *Broker* failure. Our current implementation does not include these measures as it would considerably complicate the implementation. Our future work includes analyzing how these can be implemented without compromising the strengths of *Broker*.

Acknowledgment

The authors would like to thank the Natural Sciences and Engineering Research Council of Canada, the Department of Computing Science, the Faculties of Science, and Engineering at the University of Alberta for their support of this project and the members who contributed to the RoboCup project.

References

[1] F. Bolton. *Pure CORBA: A code intensive premium reference*. SAMS, 2002.

[2] Luiz Chaimowicz, Anthony Cowley, Vito Sabella, and Camillo J. Taylor. Roci: A distributed framework for multi-robot perception and control. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1:266–271, 2003.

[3] Robotics Engineering Excellence. Robotics engineering excellence - software products, 2004. http://www.resquared.com/RTC.html.

[4] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, 1995.

[5] S. Schneider G. Pardo-Castellote and M. Hamilton. Ndds: The real-time publish-subscribe middleware. *Real-Time Innovations, Inc.*, 1999.

[6] Brian Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

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

[8] Object Management Group. Introduction to omg specifications, 2004. http://www.omg.org/gettingstarted/specintro.htm.

[9] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[10] S. Huston, J. Johnson, and U. Syyid. *The ACE Programmer's Guide*. Addison-Wesley, 2004.

[11] Intel. Intel research - microprocessor research - media, 2004. http://www.intel.com/research/mrl/research/opencv/.

[12] Matthew McNaughton and Hong Zhang. Color vision for robocup with fast lookup tables. In *IEEE International Conference on Robotics, Intelligent Systems and Signal Processing*, October 8-13 2003.

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

[14] Dale James Reid Simmons. *IPC - A Reference Manual*. Carnegie Mellon University - School of Computer Science / Robotics Institute, February 2001. IPC Version 3.4.

[15] RoboCup. Robocup official site, 2004. http://www.robocup.org.

[16] Carnegie Mellon University. Carnegie mellon robot soccer, 2002. http://www.cs.cmu.edu/ robosoccer/small/.

[17] R. Simmons Carnegie Mellon University. The interprocess communications (ipc) system, 2004. http://www-2.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html.

[18] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, June 2002.

[19] Sean Verret, Hong Zhang, and Max Q.-H. Meng. Collective sorting with local communication. In *Proc. IROS'04, IEEE/RSJ International Conference on Intelligent Robots and Systems (in press)*, pages 2687–2692, Japan, September 2004.

[20] Sean R. Verret. Perception and communication – their relationship in collective sorting. Master's thesis, University of Alberta, Edmonton, Alberta, September 2004.