TASK ALLOCATION FOR HETEROGENEOUS ROBOTS

A THESIS
SUBMITTED ON THE 28th DAY OF APRIL, 1998
TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
OF THE SCHOOL OF ENGINEERING
TULANE UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
BACHELOR OF SCIENCE
BY

BRIAN PAUL GERKEY

APPROVED:

James Jennings
CHAIR

Parviz Rastgoufard

Abstract

This paper describes the design and implementation of a multi-robot task allocation system. The system is built upon the MOVER system (see [JKW98]) and allows the user a high-level interface for posing tasks to a group of autonomous heterogeneous robots. Rather than assigning a task to an individual or group of individuals, the user simply poses the task to the system as a whole (with no knowledge of individual machines). Along with a description of the task itself, the user must supply a list of "capabilities" required for each component of the task. Given the list of capabilities, a publish/subscribe messaging system is used to discover qualified and available machines. A negotiation step ensues which results in the actual doling out of the components of the task. Tasks can range from simple one-robot jobs to multi-robot, multi-step, fully-synchronized endeavors.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

What do we want robots to do? Russell and Norvig[RN95] name the following as three of the most common robot task categories:

- Manufacturing and materials handling
- Gofer robots
- Hazardous environments

The jobs in these areas were certainly once (and in many cases still are) accomplished by humans without the aid of robots. So, for robots to make a real contribution, they must outperform[1] their human counterparts in these tasks.

[1] I take "performance" here to be a measure which weighs the productivity of the participant against the risk posed to the participant by the environment.

How can we achieve this superior performance? One obvious answer is *cooperation.* Robots who effectively cooperate with each other on tasks will assuredly be more capable than their non-cooperative brethren who must toil individually at their chores. Two striking examples of the potential for the use of cooperative robots are the pusher/steerer manipulation protocol presented in [Bro95] and the distributed map making strategy described in [JKWT97]. In both cases, the robots cooperate in order to accomplish useful tasks that would have been more difficult or even impossible with the use of only one robot. However, this cooperation is only achieved at the explicit direction of the human user. The user must decide which robots to use, ensure that they are available, and then give each robot its portion of the task. If we could delegate these decisions to the robots themselves, the robots would be considerably more useful, with a lesser need for human intervention in their activities.

1.2 The Problem

Consider the example task of moving a large[2] box through a specified path across a room; if we give this task to a robot, that robot should be able to determine whether help is required and then assemble a team of robots who will cooperate to accomplish the task. This process is by no means simple. First of all, the robot must be able to perform some kind of task analysis; this analysis will necessarily involve the decomposition of tasks into their component parts. From

[2]I mean 'large' in the way that it is used in [Bro95]: that the mass and size of the object are of roughly the same order of magnitude as those of the robot involved.

this decomposition, the robot must determine how much help, if any, he[3] needs for the task at hand[4].

In addition, he must be capable of discovering who is available to help. An important question: When is a robot *available*? How can robot A determine if robot B is available for task T? In fact, how can robot B determine if robot B is available for task T? *Availability* is clearly a function of at least the task and the state of the robot: while robot B may be not available to help on physical manipulation task M, he may indeed be able to assist with pure computation task C. However, the state of the robots is dynamic: 5 minutes from now, robot B might be available for task M. So, even if we start with a team of identical robots, they might not look so identical as their capabilities change through time. In fact, they might look like a team of heterogeneous robots; we must therefore develop a task allocation method that can deal with such a team. This system would have the added bonus of allowing for the overseeing of groups of truly heterogeneous robots. These 'robots' could range from sensor-packed mobile manipulators to simple desktop workstations; each machine has different abilities which should be fully exploited in order to meet given goals.

Once the robots' state can be clearly represented, there must be a method for communicating this information to potential team members. The metaphor used to describe this communication is inspired by object-oriented (OO) programming: each robot is modeled as a network 'object' on which various methods may be invoked. These methods, or *services*, allow a uniform interface for the remote querying of information such as current position and orientation, processor load,

[3]In this paper, "he" should probably be read as, "he, she, or it."

[4]Although originally part of this project, the problem of task decomposition and analysis has been removed as it is a field unto itself. It is assumed that the user performs this step.

and peripheral device status. To support the autonomy of the robots, these queries must be made anonymously; a user program executing on one robot should not be aware of or be affected by queries made against that robot by others. In addition, a querying robot must be able to make broadcast requests and then simply wait for one or more robots to respond. The messages will be addressed by content rather than by destination; the sending robot need not know who is receiving his message and each receiving robot will only process those inbound messages which are currently relevant. If possible, network-level features such as IP broadcasting and IP multicasting should be exploited for greater communication efficiency [Tan96].

In this paper, I present the design and implementation of an object-oriented robot control architecture which addresses all of these task automation issues.

Chapter 2

Related Work

2.1 Task Analysis

Much work has been in area of task analysis; here, we are mainly concerned with task decomposition into primitives. In [S$^+$96], Sousa describes a mobile robot control architecture based on motion primitives which allow continuous path following and path following as defined by the serial application of motion primitives.

In [MK97], Morrow and Khosla present a methodology for combining primitive robot movements into more complex robot skills. They categorize primitives based on the degrees of freedom present during the primitives' execution. The various classes of primitives can then be concisely described by a 'grammar' which Morrow and Khosla introduce. While their experiments were limited to the use of a Puma 560 robot, their taxonomy for primitive classification is rather general; in fact, one of their example primitives, ALIGN, is immediately applicable to the development of manipulation strategies for mobile robots.

Given a set of task primitives like those described in [MK97], one could apply the ideas in [MM97] to develop a procedure by which those primitives and their various compositions might be tested. In this work, Mulligan and Mackworth present a part-orienting device and a planner for that device which must vary a number of predetermined physical parameters in succession in order to achieve the assigned orientation. In order to reduce to a manageable size the number of trials that they had to run in testing their choice of parameters, Mulligan and Mackworth identified certain redundancies in the set of possible observations and derived a fractional factorial design. Their use of this specialized analysis allowed for more than an order of magnitude reduction in test runs while maintaining the requisite statistical significance of the experiments.

2.2 Information Hiding: Object-Oriented Robot Interfaces

Jeon et al. present a device-level OO interface in [J^{+}96]. Their goal is to avoid device-specific programming environments and instead provide a standard interface to a set of sensors and actuators. They implemented their system as a C++ class hierarchy with various objects representing: data processing units (DPUs), to which actual hardware devices are connected; Task Modules (TModules), which contain some number of DPUs, and a Behavior-Based Controller, which is composed of TModules. This system is not distributed; rather, it is meant to control the collection of peripheral devices available on a single mobile robot. In addition, the choice to build a behavior-based controller allowed for simple collision-avoidance/path-following exercises, but it seems that the system cannot be easily extended to deal with more complex tasks.

Another device-level OO-interface is presented in [BF96]. Based on the GISC-Kit[1] and CORBA[2], this system provides a distributed C++ interface to the control of a variety of standard devices. These devices are divided into Motion Device classes (tables, presses, mills) and Robot Device classes (multi-joint arms). One set of methods is available for Motion Devices while another set is available for Robot Devices. A Motion Controller class encapsulates the methods common to both. In true OO-style, the integration of a new device simply requires the writing of a new Driver class in which the hardware specific functions are mapped to the appropriate generic methods.

2.3 Information Sharing: Robot Communication

Explicit inter-robot communication is not necessary for all cooperative robot systems. For example, the quite effective pusher/steerer manipulation protocol in [Bro95] is accomplished with the robots' only communication being sensory feedback from the object between them. A similar system for more robots is shown in [AE97]. Also, the robots used in behavior-based approaches such as [KZ96] require no explicit communication; instead, they gather all pertinent information from their own sensors.

[1]Generic Intelligent System Communications; developed by Sandia National Laboratories as a device-independent interface to robot subsystems.

[2]Common Object Request Broker Architecture; developed by the Object Management Group (OMG) to be a vendor-neutral standard for distributed OO applications.

The overhead involved in developing a useful inter-robot communication system is definitely non-trivial. Some roboticists build their own, such as Jennings et al. in [JWE97]; some others simply assume the existence of such a system. In [C$^+$96], Cai et al. describe their *distributed autonomous robotic system* (DARS) as a collection of homogeneous robots who share some sensor information with the hope that the fusion of all the robots' sensor data will allow for optimal path planning; this system clearly requires, yet does not provide, or even describe, a communication method by which this data exchange will occur.

When a suitable communication system is available, the user is free to engage in a wider range of experiments than is otherwise possible. To effectively execute such experiments, roboticists often develop sophisticated protocols which define the various kinds of messages that can be sent and, possibly, when and to whom they can be sent. Lin and Hsu have developed two such messaging protocols. The first, called *help-based cooperation protocol* (HCP) is presented in [LjH95]. It is based on individual request/reply interactions between the robots. The use of HCP is illustrated in Lin and Hsu's Object-Sorting Task (OST): the robots perform a parallel search for the objects to be sorted; when a robot finds an object, he makes individual point-to-point help requests to other robots; using their responses, the original robot selects his partners and beckons them to his position so that the cooperative manipulation step (i.e. the *sorting*) can begin.

Lin and Hsu describe the second incarnation of their OST in [LjH96]. Instead of HCP, they use a slightly different protocol, which they term *coordination-based cooperation protocol* (CCP). Using CCP, as with HCP, the OST begins with a parallel search of the work area. However, when a robot finds an object, rather than immediately making numerous help requests, he simply broadcasts to all robots what he has discovered. So, without the use of any *a priori* knowledge about which other robots are currently searching, each individual robot has fairly complete information about who has discovered what objects and where those objects are. Given this anonymously-obtained global object information, each agent makes his own schedule for sorting all the objects in an optimal manner. A negotiation ensues and a global sorting schedule is derived.

This idea of broadcast (as opposed to point-to-point) messaging is exemplified in TIB®/Rendezvous™, a multi-platform commercial middleware[3] product created by TIBCO Software, Inc. [TIB97]. This piece of middleware is meant to facilitate the development of distributed applications (i.e. applications that run simultaneously on multiple computers) by providing messaging services that are platform- and network-independent. The TIB/Rendezvous software has two components: the TIB/Rendezvous API and the TIB/Rendezvous daemon. The API, which is available in a number of languages, including C, Java, and Perl, is the programmer's interface to the system; it provides all the functions for building, sending, receiving, and processing messages. The daemon is simply an executable program that runs on each machine involved in a distributed system and handles the dispatching of messages to the applications involved.

The real ingenuity of TIB/Rendezvous lies in the implementation of two concepts: *subject-based addressing* and *publish/subscribe messaging* (see Section4.1 for details). In the TIB/Rendezvous system, information is put on a "bus" and anyone who wants it can have it[4]. The result is a loosely-coupled distributed system in which the data producers have no knowledge of the data consumers

[3] *Middleware* is the software which separates the distributed application programmer from the details of the network (e.g. sockets, ports, etc.).

[4] I am ignoring TIB/Rendezvous's more standard features such as point-to-point messaging and message security. For a complete treatment of these topics, consult [TIB97].

and vice versa. These ideas can be easily applied to a distributed robot control architecture by, for example, realizing that devices such as infrared sensors are data producers and that applications such as path planners are data consumers. More generally, each robot could be both a producer and a consumer: he would publish current state information and also subscribe to the state information of others.

Chapter 3

The Robots

3.1 Introduction

Founded by Dr. James S. Jennings in the fall of 1995, the Tulane Mobile Robotics Laboratory[1] has quickly developed into a productive research center. Current research topics include: distributed navigation and map-making; a parallel "search and rescue" task solution; distributed manipulation of large objects; fault-tolerant programming techniques for robot teams; combining reactive and procedural programming; and distributed error detection and recovery. To support this work, we have the following hardware at our disposal:

- 2 Sun SparcStations running SunOS
- 2 PCs running Linux
- 3 RWI B14 Mobile Robots (see Section 3.2)
- Numerous empty boxes

[1]or, informally, the Cooperative Mobile Robotics Laboratory of Tulane (CaMeLoT).

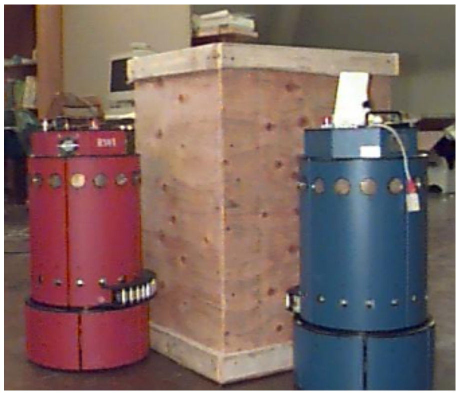


Figure 3.1: Two of our mobile robots, Ernst and Moseley, near a large box.

The desktop workstations are mainly used for code development and system maintenance (and *xkobo*). However, since they are easily incorporated into robot teams alongside their mobile counterparts (see Section 3.3), the workstations can also be used in robot experiments.

To control the robots, we use the programming system known as MOVER (see Section 3.3) which in turn interfaces to the robots through the *Tulane Robot Interface Process (trip)* (see Section 3.3.1).

3.2 Meet Ernst, Moseley, and Elvis

Ernst, Moseley, and Elvis are our three robots; they are B14 mobile robots built by Real World Interfaces, Inc. (RWI). Each robot is a cylindrical case which sits on a three-wheel base that can perform rotations in place (see Figure 3.1). Surrounding that base is a circular, rigid, six-segment skirt which offers coarse tactile sensory information by indicating which segments are depressed. More precise tactile information is provided by an array of 12 cylinderical bumpers at the front of each robot. These bumpers are used mainly for collision detection and contact mode determination and are indispensible in manipulation tasks. Above the bumpers is a ring of sixteen infrared transmitter/receiver pairs which are used for short-range distance measurements. Closer to the top of each robot is a ring of sixteen sonar transmitter/receiver pairs. We use these sonar sensors extensively for collision avoidance, target following, and map making. On top, there are four pushbuttons which allow for a user to give input the robot while away from the keyboard (arbitrary actions can be bound to the buttons). In addition, there are four LEDs through which the robot can communicate information back to the user during the execution of a program. A more flexible feedback mechanism is provided by way of a speech synthesiser and speaker for each robot.

On the inside, each robot houses a standard Intel-based PC complete with disk drives and expansion cards. They run the Linux operating system and are connected to our network via Lucent Technologies WaveLAN wireless ethernet. Normally, all robot access is performed remotely through the network, but video and keyboard ports are available so that the console may be used for maintenance or system debugging. The robots are powered by batteries which are charged through detachable tethers.

3.3 MOVER

Herein is described MOVER, the main programming interface to our robots. First, however, the reader should be familiar with *trip* and Kali-Scheme.

3.3.1 trip

The *Tulane Robot Interface Process (trip)* is a socket-based client-server architecture for providing access to robot hardware. On each robot, there is a multi-threaded server (written in C) running which listens for socket connections from clients on a certain TCP port. Once a connection is established, this server, or daemon, controls the local hardware devices in response to client requests, which are generally either: open a device (such as the wheelbase), or send a command to a device (such as "fire sonars"). Different clients may open different devices on the same robot; however no two clients may open the same device on the same robot (mutual exclusion for each device is provided by lock files). The client and server interact by passing simple ASCII messages back and forth on the socket between them. These messages adhere to a well-defined protocol, which includes framing and character stuffing.

Since the server and client are connected only by standard Internet sockets, a robot client can be implemented in any programming environment that provides socket facilities. To date, clients have been developed in Scheme (see Section 3.3.3), C, C++, Java, Tcl/Tk, Perl, Smalltalk, and Emacs.

3.3.2 Kali-Scheme

In order to facilitate complex multi-robot experiments, we make extensive use of Kali-Scheme, a dialect of the lexically-scoped LISP-like language Scheme (for a complete discussion of the system and the origin of its name, see [CJK95]). Kali-Scheme is a distributed implementation of Scheme which consists of a collection of address spaces. An address space is confined to a single node on the network, but more than one address space may exist on each node at any time. In general however, a single address space is created on each physical machine on the network. Within each address space, potentially many concurrent, lightweight, preemptible threads may execute. A Kali-Scheme program consists of a collection of such threads, each of which could reside in different address spaces. In the tradition of Scheme, the important components of this system are all implemented as ordinary Scheme data structures[2]. These components include address spaces, threads, procedures, and continuations.

Threads executing within a single address space communicate via shared memory while threads in different address spaces use an explicit message-passing system. For interaddress space communication, messages are passed using a lazy frame-by-frame copy mechanism that results in the efficient transmission of both simple and complex data structures. A significant advantage of Kali-Scheme's message-passing system is that messages can have executable content; that is, a message can contain an arbitrary procedure to be evaluated in the target address space. One consequence of this feature is that user-level thread migration is quite simple: a thread's state can be packaged into a continuation (which is modeled as a procedure) and then shipped off to a different address space where the continuation will be evaluated and the thread will pick up where it left off. In addition, the ability to send executable messages in Kali-Scheme eliminates the need for the predefined message protocols of other message-passing systems.

The programmer interfaces with Kali-Scheme's message-passing system through two primitives: `remote-run!` and `remote-apply`. The first procedure, `remote-run!` is used for asynchronous "fire-and-forget" messages; it takes as

[2]Data structures in Scheme are first-class objects and, as such, may be dynamically created, destroyed, bound to variables, passed as arguments to procedures, etc.

arguments the following: a target address space, a procedure to be evaluated, and arguments to that procedure. In the target address space, a message dispatcher returns to the sender a simple acknowldgement and then spawns a new thread in which the given procedure is applied to its arguments. Since the results of the procedure application are discarded, `remote-run!` is generally used in situations in which the evaluation of the procedure has some side-effect on the target machine. When the results of the procedure application need to be returned to the sender in a synchronous request/reply manner, the programmer uses `remote-apply`, which takes the same arguments as its asynchronous counterpart. The difference between the two is that `remote-apply` blocks while the procedure is applied in the target address space and then returns to the caller the result of that application.

3.3.3 MOVER

While there are various robot clients from which to choose, most work in the lab (and all of the work for this project) is accomplished using the Scheme client, known as MOVER (for a complete discussion of the system, see [JWE97] and [JKW98]). MOVER is implemented in Kali-Scheme (see Section 3.3.2) and operates on robot hardware through *trip* (see Section 3.3.1). In addition to providing a thorough interface to robot hardware, MOVER exposes the distributed features of Kali-Scheme in such a way that the robot programmer is given a rich set of primitives which he can use to operate on *dynamic teams*. *Dynamic teams* are explained thoroughly in [JKW98]; for this discussion, it suffices to define them as "temporary logical associations of robots" and list the related operations which the programmer can perform:

- Gather a set a robots into a *team*.

- Distribute the components of a task to team members.
- Synchronize the actions of team members (e.g. before going to step 2, wait for all to complete step 1).
- Add members to a team (and give each one a new task).
- Remove members from a team (and abandon their current tasks, if any).
- Substitute a new robot for a current team member, migrating the remainder of the current member's task to the new robot.

Thus, the programmer has the freedom to create, modify, and destroy teams of robots at will. While there are some hooks in MOVER to provide for automatic selection of team members and automatic allocation of tasks, the current implementation provides nothing more sophisticated than "select all available robots" and "allocate this task to a randomly selected team member." It is the development of a more intelligent team selection and task allocation mechanism with which this project is concerned.

Chapter 4

Murdoch

The result of this project is a body of software which is collectively known as the *Murdoch* system. The various components of Murdoch are described below. For a discussion of the user-level interface, `pose-task`, see Section 4.2.

4.1 Publish/Subscribe Messaging in Action

At the heart of Murdoch is an implementation of publish/subscribe messaging, which in turn depends on subject-based addressing.

Definition 4.1.1 Subject-based addressing *is an addressing scheme in which individual messages are addressed by content rather than destination.*

Definition 4.1.2 Publish/subscribe messaging *is a messaging paradigm that uses* subject-based addressing *to divide a network into a loosely-coupled association of anonymous data producers and consumers. A data producer simply tags a message* *with a subject and "publishes" it onto the network; any data consumers who have "subscribed" to that subject will automatically receive the message.*

The goal of a publish/subscribe messaging is to enable a loosely-coupled distributed system in which the data producers have no knowledge of the data consumers and vice versa. If designed well, a publish/subscribe system can provide a great deal of anonymity an independence for the individual machines involved in a large distributed application.

All of Murdoch , including its internal messaging system, is implemented in the distributed dialect of Scheme known as *Kali-Scheme* (see Section 3.3.2) and relies entirely on the Kali-Scheme primitives `remote-run!` and `remote-apply` for low-level message-passing. Due to limitations in Kali-Scheme's messaging system, Murdoch uses a kind of simulated broadcast to publish messages. Each message is actually sent individually to each machine on the network; each receiving machine checks the subject of the incoming message against its own *subscription list* (see Section 4.1.1) and decides whether or not to evaluate it[1]. This simulated broadcast method as it is used in Murdoch is certainly somewhat inefficient, but it should be noted that there exist viable commercial implementations of publish/subscribe messaging (most notably TIB/Rendezvous, as described in [TIB97]) that are quite efficient and are used in a variety of mission critical environments.

Below are a discussion of the subject namespace and descriptions of the five primitives which comprise the interface to Murdoch 's publish-subscribe messaging system: `subscribe`, `unsubscribe`, `publish`, `request`, and `request-exclusive`.

[1]All messages in the Murdoch system have executable content and so a machine accepts a message by simply evaluating its body.

4.1.1 Subject Namespace

In this system, the idea of subject-based addressing has been slightly skewed in that messages are not addressed to a single subject, but rather to an an unordered list of subjects. A potential receiver will evaluate any message addressed to a set of subjects which form a proper subset of those subjects to which he is subscribed. Each robot subscribes to a set of subjects which represent his "capabilities". For example, our mobile robot *elvis* normally subscribes to the subjects `ir`, `sonar`, `speech`, `buttons`, `lights`, and `mobile` because he is physically connected to infra-red and sonar sensors, a speech synthesiser, some buttons, some lights, and a motorized wheelbase. As another example, our desktop workstation *troy* normally subscribes to the subject `camera` because he has control of a digital camera and frame-grabber. These subscriptions can and do change over time. When the system load on a machine drops below some threshold, he will subscribe to an additional subject `idle` because he now has some unused computing power to export. When a mobile robot's battery-charging cable is disconnected, he will subscribe to the subject `untethered` because he is now free to move about the world. So, to reach all robots who have sonar sensors and are not busy, one can publish a message to the subject `(sonar idle)`; only those machines with the specified capabilities will actually evaluate the body of the message.

In addition to those subjects which represent robot capabilities (e.g. `ir`, `camera`, `idle`), each machine subscribes to two "special" subjects. One of these subjects is `all`; it is used to address messages which should be evaluated by everyone. The second "special" subject is a unique identifier that is different for each robot; this subject is used to address point-to-point messages.

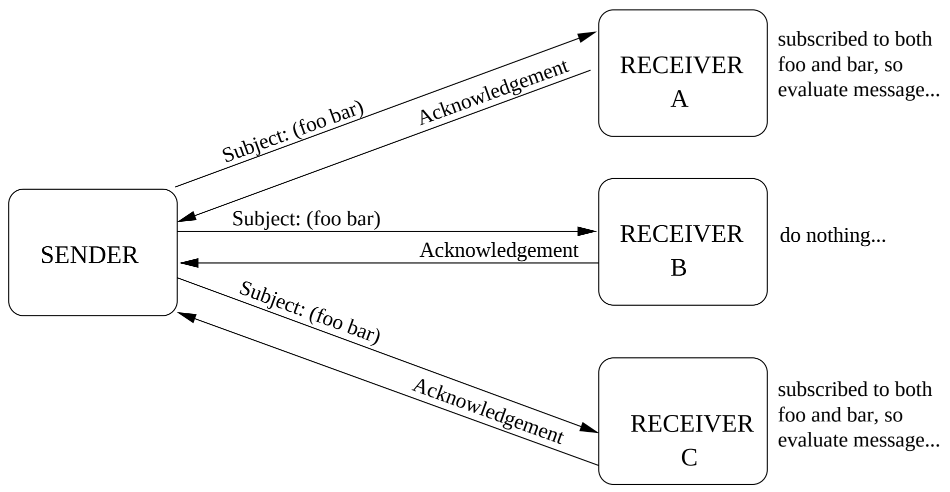


Figure 4.1: An example `publish` transaction. SENDER sends a `publish` message to the subject `(foo bar)` and a list of acknowldgements is immediately returned. Since RECEIVER B is not subscribed to `(foo bar)`, he does nothing. On the other hand, RECEIVER A and RECEIVER C are both subscribed, so they evaluate the body of the message.

4.1.2 `subscribe` & `unsubscribe`

Each robot maintains its own subscription list; a user program modifies this data structure through the use of `subscribe` and `unsubscribe`. The procedure `subscribe` takes as its sole argument a subject and, if that subject is not already present in the subscription list, adds it. If the subject is already present in the subscription list, `subscribe` does nothing. Equivalenty, `unsubscribe` takes a subject as its argument and, if the subject is present in the subscription list, removes it. If the subject is not present in the subscription list, `unsubscribe` does nothing.

4.1.3 `publish` & `request`

The `publish` procedure is used to publish asynchronous messages. Its arguments are the following: a subject list, a procedure to be evaluated, and arguments to that procedure. The immediate return value is a list containing a simple acknowldgement from each machine on the network. There is no indication of which robots, if any, actually evaluated the body of the message, and any results from those evaluations are discarded. An example `publish` transaction is shown in Figure 4.1. Primarily added for completeness and interactive convenience, `publish` is not used internally by Murdoch .

On the other hand, Murdoch internally makes extensive use of `request`, which provides a general synchronous messaging facility. `request` has the same syntax as `publish` but, instead of returning immediately, `request` waits for results from those machines, if any, which evaluated the message. The return value from `request` is a (possibly empty) list containing each result and a unique subject to be used in consequent point-to-point communication with the machine from which that result came. In this way, a programmer can use `request` to anonymously determine a list of machines who are subscribed to a specific subject (and thus have a certain set of capabilities) and then engage in private, point-to-point communication (via `publish`, `request`, or `request-exclusive`) with one of those machines. An example `request` transaction is shown in Figure 4.2.

4.1.4 `request-exclusive`

Since Murdoch exists in a multi-threaded environment, it is possible for a machine to be concurrently evaluating multiple messages, each of which might involve access to devices. When dealing with a device such as a wheelbase, which has important physical state (position and orientation) associated with it, there is a need for mutual exclusion (it would in general be bad for a wheelbase to receive interleaving commands from two different messages). For situations such as these, the third messaging primitive, `request-exclusive`, is required. `request-exclusive` has the same syntax as `request` and behaves similarly in that it waits for responses and then returns a (possibly empty) list of the responses from those machines, if any, which accepted the message. However, the responses are not the results of the evaluation the message body but rather *promises* to do that evaluation later.

When a receiving machine evaluates a `request-exclusive` message, it first unsubscribes from those subjects included in the address of the message. Thus the receiver will no longer accept messages which are addressed to any of those subjects. The receiver then constructs a procedure of one Boolean argument and returns it to the sender. This new procedure, hereafter known as a *promise*, is meant to be sent back to the receiver later for evaluation. The understanding is that the sender has exclusively reserved some of the receiver's resources for a certain job which the receiver may or may not actually need to accomplish. If the sender decides to indeed have the receiver accomplish the job, he applies the *promise* to the value TRUE, causing the receiver to:

1. evaluate the body of the original message
2. resubscribe to the previously unsubscribed subjects
3. return the results of the evaluation to the sender

On the other hand, if the sender decides not to have the receiver accomplish the job, he applies the *promise* to the value FALSE, causing the receiver to execute only the resubscription step.

In general, given a job and its resource requirements, `request-exclusive` is used in the following two-step transaction (consult the example shown in Figure 4.3(a) and Figure 4.3(b)):

1. The sender calls `request-exclusive` in order to determine who is capable. Each machine who responds has now set aside the necessary resources. The sender then examines the responses and decides which machine should actually accomplish the job.
2. The sender applies to the value FALSE the *promise* of each machine who responded but was not chosen for the job; they each "unreserve" the previously reserved resources. The sender then applies to the value TRUE the *promise* of the machine who was chosen for the job; that machine accomplishes the job and then "unreserves" the previously reserved resources.

The importance of the `request-exclusive` primitive in this transaction is that it provides enough mutual exclusion of resource use so as to allow the sender to sit back and intelligently decide who is best suited for a job. Since the necessary resources are exclusively reserved up front, the sender need not worry about another job grabbing them while he is thinking.

4.2 `pose-task`

While of general interest, the messaging primitives described in Section 4.1 are not usually called directly by user programs, but rather by the high-level procedure *pose-task*. This procedure is the standard interface to Murdoch and allows the user to pose a task in what is hopefully a natural manner. Since Murdoch is only a task allocation system and not a task analysis system, the user is expected to divide the task into its *components*. In this context, the *components* of a task are the individual executable pieces which must be concurrently evaluated on different robots. For example, [Bro95] presents a two-robot *pusher-steerer* protocol for the manipulation of a large box. This task clearly has two components: one component is the procedure to be run by the pushing robot and the other is the procedure to be run by the steering robot. After performing this task analysis step, the user is ready to supply the arguments required by pose-task: number of robots required (i.e. number of task components), list of capabilities required for each task component (see Section 4.2.1), list of metrics (see Section 4.2.2), and list of procedures (the actual executable task components). Given this information, `pose-task` enters into the negotiation protocol described in Section 4.2.3.

4.2.1 Capability Lists

For each task component in a given task, the user must specify the component's resource requirements in a *capability list*, which is used by `pose-task` to query for capable robots. A task component can have both exclusive and non-exclusive resource requirements. Thus the syntax for specifying a *capability list* allows for the user to indicate which resources are needed exclusively and which are not.

4.2.2 Metrics

When more than one robot is qualified for a task component, `pose-task` must choose one of them; so that `pose-task` can make this choice intelligently, the user is allowed to supply a *metric* for each task component. A *metric* is a procedure to be used in comparing two candidate robots. Its arguments are the two unique subjects that can be used for point-to-point communication with the candidates. The return value for a metric should be TRUE or FALSE to indicate whether or not the first candidate is "better" than the second. In this way, given the set of robots who are eligible for a task component, `pose-task` can choose the one best-suited to that component. Since `pose-task` hands the user-supplied metric a direct communication path to each candidate, the user can query them for any information needed in the decision-making process. If the user supplies no metric, `pose-task` uses a default metric which chooses randomly.

As an example, given a computationally intense job, it might be best to choose the machine with the lowest system load. Figure 4.4 shows the code for `load-metric`, a simple metric which fits this purpose. A slightly more complex metric that is useful when dealing with motion-oriented tasks is `location-metric`, which chooses the candidate closest to some target position.

4.2.3 Negotiation & Allocation

Once the user has supplied the necessary information about the task at hand, `pose-task` initiates a simple negotiation protocol in order to decide which machine will play which role (for a flow chart of this process, see Figure 4.5). The first step is to iterate through each task component and query the network for capable machines. If a task component requires exclusive use of any resources, `request-exclusive` is used so that those resources are set aside now. Only those robots who are capable will respond to the query for a given task component; thus, from the responses, `pose-task` can determine a list of candidates for each component. Each of these lists is sorted by the user-supplied metric so as to rank the candidates for each job with respect to their relative suitability. At this point, `pose-task` must assign each component to a specific machine. Although the ideal situtation would be to just pick the first robot from each candidate list, the system must ensure that no single robot is assigned more than one job. In order to resolve any conflicts, the collection of sorted candidate lists is traversed in order of descending priority[2] and `pose-task` chooses from each the first candidate who has not previously been chosen. If the conflict resolution is successful, then the task can be completed and each task component has now been assigned to a specific machine. Of course, the conflict resolution could fail if not enough of the right kinds of robots volunteer their services. If this is the case, then the task cannot be completed and the user will be informed of the situation. However, before informing the user, `pose-task` needs to release any resources which were reserved for this task by the preliminary queries. This release is accomplished

[2]In the current implementation, priority is determined by the order in which the task components are given to `pose-task` (first component has highest priority and last component has lowest priority).

by simply applying to the value FALSE any *promises* that were received. Even if the task can indeed be completed, `pose-task` might still have to release some resources. Any robot who returned a *promise* for a task component but was not chosen for that component still has resources set aside for the job. Therefore, `pose-task` iterates through the task components requiring exclusive resource use and applies to the value FALSE the *promise* of each "runner-up". Now, `pose-task` actually accomplishes the task by using the MOVER construct `with-team` to form a *dynamic team* (consult [JKW98] for details) of the selected machines and give each one his own component to evaluate. The return value of `pose-task` is just the return value of the underlying `with-team` expression: a list containing the results from each robot in the team.

4.2.4 An Example: The Dual-Sonar Task

As a comprehensive example of the use of `pose-task`, I offer what has been dubbed "The Dual-Sonar Task" (see Figure 4.6 for the actual specification). This task has three components. The first job is to travel to a target location and return a sonar reading of the area. For this component of the task, the resources `sonar` and `mobile` are required, with the latter required exclusively. The metric to be used is `location-metric` (see Section 4.2.2), which will cause the robot who is physically closest to the target to be selected. The second task component is the same as the first, but with a different target location. The third component is a compute-only job: generate, sort, and return a list of 100 random numbers. The only required resource is `idle` and the metric is `load-metric` (see Section 4.2.2 and Figure 4.4).

The test environment for this task consisted of two mobile robots with sonar (*moseley* and *ernst*) and two desktop machines (*troy* and *praline*). *moseley* was placed closest to the first target and *ernst* was placed closest to the second target. By chance, *praline* had the lowest system load. The result of posing this task to the system was the following: *moseley* was sent to take a sonar reading at TARGET 1, *ernst* was sent to take a sonar reading at TARGET 2, and *praline* was given the random number component of the task. The two mobile robots did indeed move as instructed and, when everyone was finished, `pose-task` returned a list containing the two sonar readings and the sorted random number list.

The correct behavior of `pose-task` in accomplishing "The Dual-Sonar Task" demonstrates the functionality of the Murdoch system. Murdoch was also tested with various other tasks and it performed each one correctly (i.e. it allocated the task components to the best of its ability).

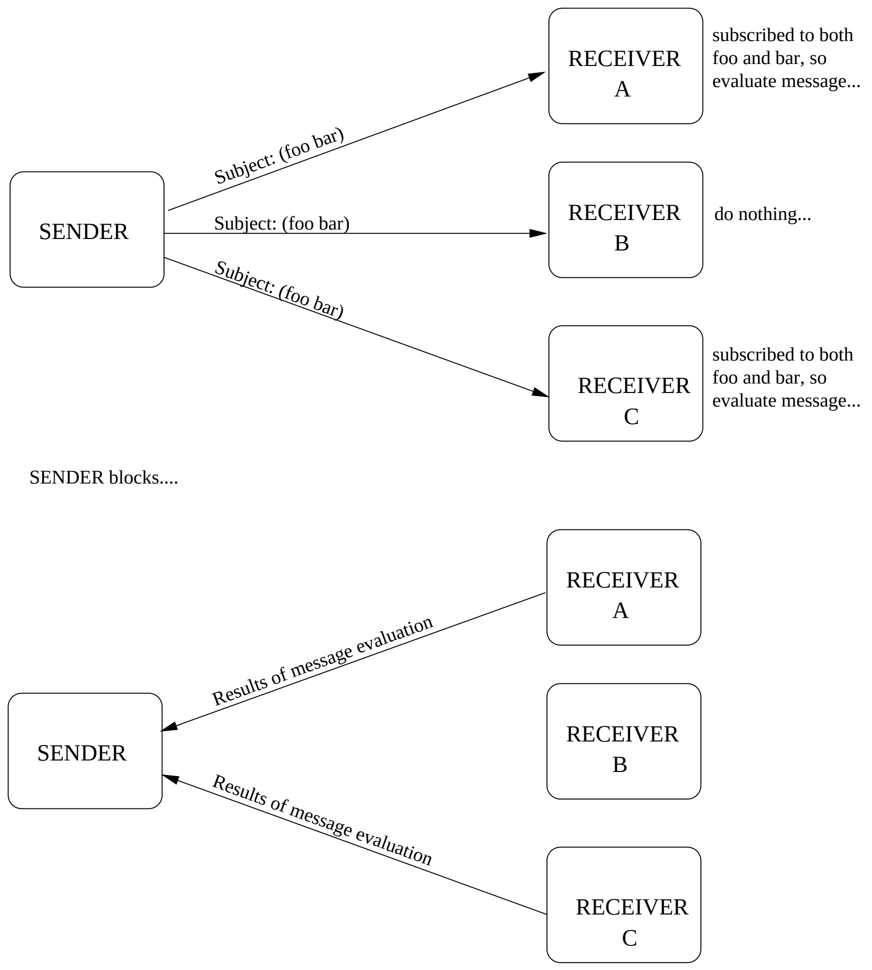
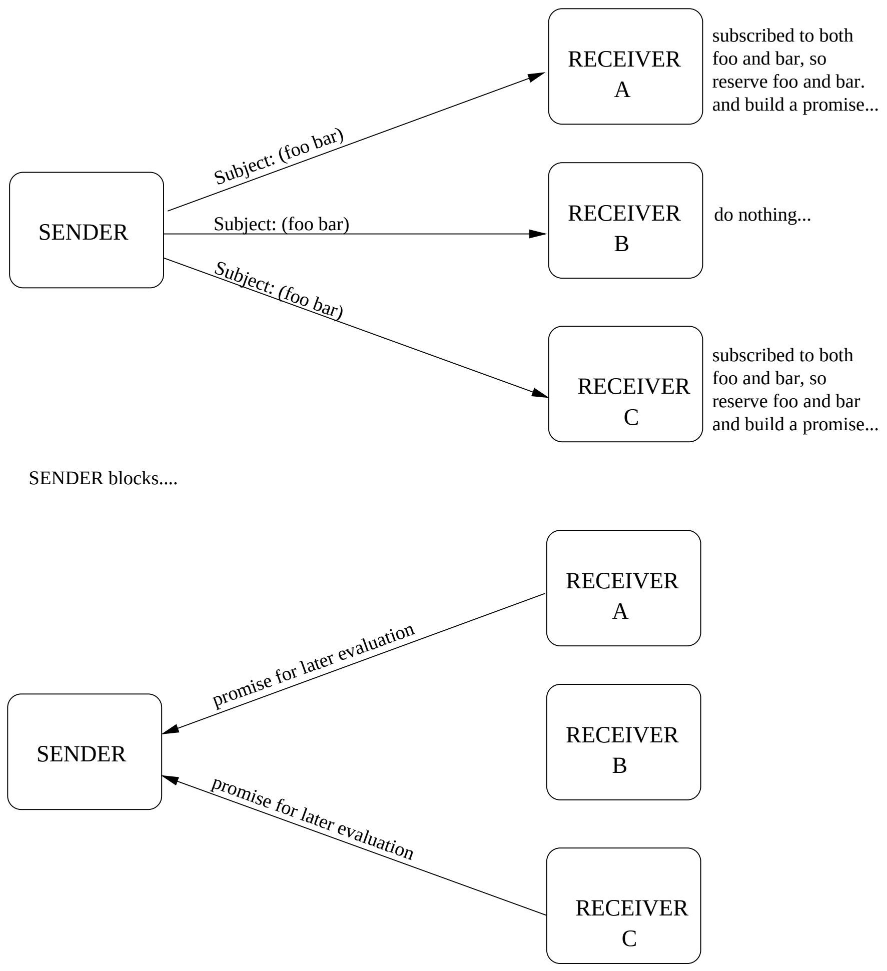
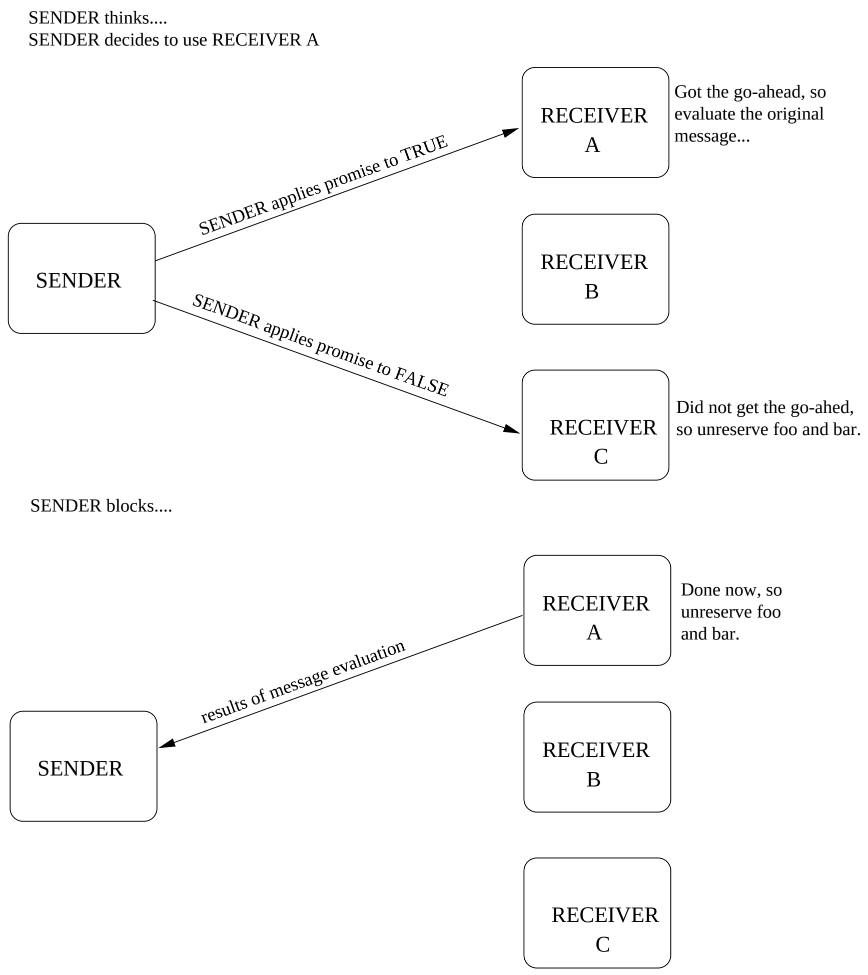

RECEIVER A
subscribed to both foo and bar, so evaluate message...
Subject: (foo bar)
SENDER
Subject: (foo bar)
RECEIVER B
do nothing...
Subject: (foo bar)
RECEIVER C
subscribed to both foo and bar, so evaluate message...
SENDER blocks....
RECEIVER A
Results of message evaluation
SENDER
RECEIVER B
Results of message evaluation
RECEIVER C

Figure 4.2: An example **`request`** transaction. SENDER sends a **`request`** message to the subject **`(foo bar)`** and waits for responses. Since RECEIVER B is not subscribed to **`(foo bar)`**, he does nothing. On the other hand, RECEIVER A and RECEIVER C are both subscribed, so they evaluate the body of the message and return their results to SENDER.


RECEIVER A
subscribed to both foo and bar, so reserve foo and bar. and build a promise...
Subject: (foo bar)
SENDER
Subject: (foo bar)
RECEIVER B
do nothing...
Subject: (foo bar)
RECEIVER C
subscribed to both foo and bar, so reserve foo and bar and build a promise...
SENDER blocks....
RECEIVER A
promise for later evaluation
SENDER
RECEIVER B
promise for later evaluation
RECEIVER C

(a) An example `request-exclusive` transaction

Figure 4.3: SENDER sends a `request-exclusive` message to the subject `(foo bar)`. Since RECEIVER B is not subscribed to that subject, he does nothing. On the other hand, RECEIVER A and RECEIVER C are both subscribed, so they each reserve `foo` and `bar` and then each build a *promise* which is returned to SENDER.


SENDER thinks....
SENDER decides to use RECEIVER A
RECEIVER A
Got the go-ahead, so evaluate the original message...
SENDER applies promise to TRUE
SENDER
RECEIVER B
SENDER applies promise to FALSE
RECEIVER C
Did not get the go-ahed, so unreserve foo and bar.
SENDER blocks....
RECEIVER A
Done now, so unreserve foo and bar.
results of message evaluation
SENDER
RECEIVER B
RECEIVER C

(b) An example `request-exclusive` transaction (cont'd)

Figure 4.3: SENDER now has all the *promises* and he decides that RECEIVER A should actually do the job. Consequently, SENDER applies RECEIVER C's *promise* to the value FALSE, causing RECEIVER C to unreserve `foo` and `bar`. SENDER applies RECEIVER A's *promise* to the value TRUE, causing RECEIVER A to evaluate the body of the original message. SENDER blocks until RECEIVER A returns the results of that evaluation.

```
(define (load-metric)
  (lambda (subject1 subject2)
    (let ((load1 (cadar (request subject1 get-system-load)))
          (load2 (cadar (request subject2 get-system-load))))
      (< load1 load2))))
```

Figure 4.4: Code for `load-metric`. Given the unique subjects for two candidate robots, this simple metric queries the system load of each candidate and returns TRUE or FALSE to indicate whether or not the first candidate has the lower load.

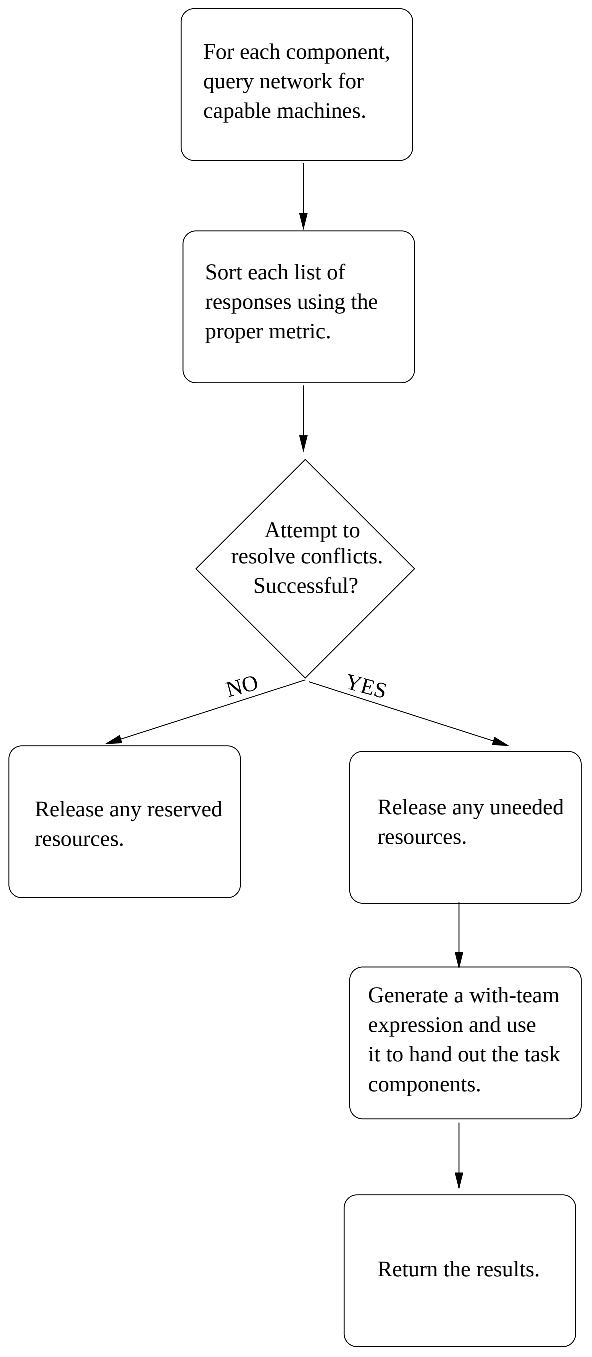

For each component, query network for capable machines.
Sort each list of responses using the proper metric.
Attempt to resolve conflicts. Successful?
NO
YES
Release any reserved resources.
Release any uneeded resources.
Generate a with-team expression and use it to hand out the task components.
Return the results.

Figure 4.5: Negotiation protocol flow chart

```
(define (dual-sonar-task)
  (let ((target1 '(300 200 0))
        (target2 '(-1000 200 90))
        (num 100))
    (list 'dual-sonar-task
          3
          (list
            (list (list 'mobile) (list 'sonar))
            (list (list 'mobile) (list 'sonar))
            (list '() (list 'idle)))
          (list
            (location-metric-maker target1)
            (location-metric-maker target2)
            load-metric)
         (list
            (sonar-task-proc-maker target1)
            (sonar-task-proc-maker target2)
            (random-number-task-proc-maker num)))))
```

Figure 4.6: Code for `dual-sonar-task`.

Chapter 5

Conclusions

Although somewhat unpolished, the system described in this paper should be considered a success. Murdoch is an intelligent task allocation system that can effectively handle groups of truly heterogeneous robots without the use of a central controller or *a priori* knowledge about the robots. This automated system offers some clear advantages over the standard manual distrution method. Perhaps most important is the anonymity that is provided for the machines involved. The user has no explicit knowledge of the existence of any of the robots and the robots themselves have no explicit knowledge of each other. Since tasks can be posed to the system by any of the robots, all the machines involved are peers. In fact, since all communication is conducted anonymously, no assumptions are made about the state of the network and so individuals can come and go as they please without deleteriously effecting the operation of the whole system. Further, rather than relying on some centralized capability repository, each robot privately keeps track of its capabilities, changing them as necessary. Thus task allocation decisions are based entirely on information that is guarenteed to be correct (or as correct as physically possible).

Murdoch is certainly a novel approach to the problem of task allocation. The underlying ideas, subject-based addressing and publish/subscribe messaging, have been borrowed directly from the industrial distributed control arena and yet seem to be applicable to the study of heterogeneous robot control. The fact that the implementation of a complete publish/subscribe messaging system required only the small amount of code that it did is a testament to the elegance of the publish/subscribe idea itself and the power and flexibility of Kali-Scheme and the MOVER system.

The result of this project is a working environment in which, from the user's point of view, a task is posed to an unknown set of hopefully qualified robots who will volunteer their available resources (e.g. sonar, wheelbase, CPU) in order to accomplish the task. The way in which that task must be phrased is of course paramount in determining the utility of the system. Experimentation in the lab with real tasks indicates that the current task framework is quite reasonable; regardless, that portion of Murdoch can be easily modified without altering the important concepts of the system. In the end, the true utility of Murdoch will be determined by future robot programmers who use and modify it.

Chapter 6

Future Work

There are many avenues for further work on this project. Within the messaging system itself, one useful extension would be a new primitive which would reserve resources in such a way that they are available for general use but cannot be exclusively reserved by any task. This primitive, `request-non-exclusive`, would be used by tasks which require the use of a resource for a non-mutative "reading" operation and do not mind if other tasks simultaneously use it in the same way; other tasks which need to "write" to that resource would be blocked. For completeness, there should probably also be asynchronous primitives `publish-exclusive` and `publish-non-exclusive`; however, the real applicability of these constructs remains to be seen. The subject namespace could be extended so as to offer a wider range of information to the user. One possibility is a heirarchical system in which, for example, a mobile robot whose wheelbase is now being used by a task would no longer subscribe to `mobile`, but rather to `mobile.in_use`. As demonstrated in TIB/Rendezvous ([TIB97]), a heirarchical subject namespace in conjunction with wildcard support allows the programmer a more powerful interface for developing distributed control applications.

An immediate consequence of a more informative subject namespace is the fact that a user could determine not only that a certain resource is not currently available, but why it is unavailable. Given this kind of information, when a user discovers that his task cannot be completed, he could decide to retain control of those resources which are available and wait for the others. With the addition of an optimal scheduling policy that would handle issues such as a deadlock, Murdoch would become a fully-automated task scheduling and allocation system.

From an interface standpoint, several improvements can be made. The current implementation assigns relative priorities to task components based on the order in which they are given to `pose-task`. The user should be allowed to arbitrarily specify these priorities himself; it seems possible that the metric interface for ranking robot candidates could be used to implement a flexible priority system for task components. The metric interface itself could also be greatly improved; a better version would allow the user to "compose" multiple metrics so that robots (and task components) could be compared by one metric, and then, if they are equal, compared by another (and another, and another...). However, by far the most beneficial modification that could be made to Murdoch would be to merge the entire task-posing interface with the existing team synchronization interface (see [JKW98]) to produce a single, fully-integrated system which can automate task allocation and ensure synchroniztion for arbitrarily complex, multi-step, multi-robot tasks. With this last improvement, Murdoch would become a truly useful tool for robot programmers working on other projects.

Appendix A

Source code for Murdoch

A.1 aux.scm

```
;;
;; helpers
;; (move these somewhere else...or find them already written somewhere else)
;;
(define (remove item ls)
  (filter (lambda (elt) (not (equal? elt item))) ls))


;
; useful macro for doing remote variable references
;
(define-syntax var-ref
 (syntax-rules ()
   ((var-ref uid var)
    (remote-apply (uid->aspace uid) (lambda () var)))))

;
; useful macro for doing remote variable assignments
;
(define-syntax var-set!
 (syntax-rules ()
   ((var-set! uid var value)
    (remote-apply (uid->aspace uid) (lambda () (set! var value))))))
```

```
;
; just remote-apply display somewhere else
;
(define (display-remote uid string)
  (remote-run! (uid->aspace uid)
               (lambda ()
                 (display string)
                 (newline))))

(define (robot? hostname)
  (cond
   ((equal? hostname "ernst") #t)
   ((equal? hostname "moseley") #t)
   ((equal? hostname "elvis") #t)
   (else #f)))

(define (me-robot?)
  (robot? (getenv "HOSTNAME")))

(define (set-origin! topic)
  (request topic
           (lambda ()
             (set-reckon-position! 0 0)
             (set-reckon-heading! 0))))


(define (any-duplicates? ls)
  (let loop ((main-ls ls) (copy-ls '()))
    (cond
     ((null? main-ls) #f)
     ((member (car main-ls) copy-ls) #t)
     (else (loop (cdr main-ls) (cons (car main-ls) copy-ls))))))


(define (map2 proc ls1 ls2)
  (if (null? ls1)
      '()
      (cons (proc (car ls1) (car ls2)) (map2 proc (cdr ls1) (cdr ls2)))))

(define (map3 proc ls1 ls2 ls3)
  (if (null? ls1)
      '()
      (cons (proc (car ls1) (car ls2) (car ls3))
            (map3 proc (cdr ls1) (cdr ls2) (cdr ls3)))))
```

A.2 initial.scm

```
;
; system initialization things
;


;
; a standard initialization procedure for everyone (build it into
; an image?)
;
(define (initialize)
  (map
   (lambda (uid)
     (for-each display (list "initializing: " uid))
     (remote-apply (uid->aspace uid) (make-subscription-initializer)))
   *uids*)
  (display "safe-im")
  (request '(all) safe-im)
  (display "starting load-monitor")
  (request '(all) load-monitor)
  (display "starting battery-monitor")
  (request '(mobile) battery-monitor)
  #t)

;
; set up initial subscriptions
;
(define (make-subscription-initializer)
  (lambda ()
    (let ((uid (local-aspace-uid)))
      (set! *subscriptions-lock* (make-lock))
      (set! *subscriptions* '((all ())))
      (map subscribe (list uid 'idle))
      (cond
       ((me-robot?)
        (map subscribe (list 'sonar 'speech 'lights 'bumpers 'ir 'mobile)))
       (else
        '())))))
```

A.3 loader.scm

```
(user '(open
        aspaces
        getenv
        processes
        handle
        extended-ports))
```

A.4 main.scm

```
;
;
; (gather ....)
; ,open aspaces
; ,open getenv
; ,open handle  (for ignore-errors)
; ,open processes  (for run-process)
; ,open extended-ports

;; NOTES:
; Ok, at some point, a machine's capabilities MUST be hardcoded, right?
; For example, praline can't be expected to determine at run-time if
; he is physically connected to sonar devices. So, for now, I'll write
; a cheesy safe-im which, based on hostname, set some flags in a table
; to indicate what devices are locally available.
;
; subjects
;   'all: everybody
;   '<hostname>: to address a machine directly
;   'available:
;   'tethered:
;   'idle:
;
;
; TODO:
;  (make-publisher): to do things like spawn a thread to publish sonar values
;


(map load
  '("/home/gerkey/thesis/initial.scm"
    "/home/gerkey/thesis/aux.scm"
    "/home/gerkey/thesis/safe.scm"
    "/home/gerkey/thesis/message.scm"
    "/home/gerkey/thesis/monitor.scm"
    "/home/gerkey/thesis/publish.scm"
    "/home/gerkey/thesis/goto-tanis.scm"
    "/home/gerkey/thesis/new-task.scm"
    "/home/gerkey/thesis/vector.scm"
    "/opt/geometry/common/separate-string.scm"))
```

A.5 message.scm

```
;
; useful messages
;

(define *battery-charging* 130)
(define *battery-low* 100)

(define (battery-charging? uid)
  (> (cadar (request uid safe-battery-voltage)) *battery-charging*))

(define (battery-low? uid)
  (< (cadar (request uid safe-battery-voltage)) *battery-low*))

;
; a thunk to be sent out in a request
; should return position and sonar readings.....
; Hmm...is there a way to determine whether or not the sonar units
; are currently pinging?  Ideally, we want to leave the sonar units
; how they were (possibly having to start and then stop them again
; in order to a get reading).
;
(define (context)
  (list
   (safe-reckon)
   (sonar-read)))

(define (t)
  #t)

(define ut 'untethered)
```

A.6 monitor.scm

```
;; -*- Mode: Scheme; -*-
;;
;; battery-monitor
;;
;; James S. Jennings
;; Dept. of Computer Science
;; Tulane University
;;
;;
;; modified by Brian Gerkey

(define *battery-monitor-thread* #f)

(define (battery-monitor)
  (if (thread? *battery-monitor-thread*)
      (begin
        (warn "Terminating running battery monitor thread")
        (kill-thread! *battery-monitor-thread*)))
  (set! *battery-monitor-thread*
        (spawn-on-root
         (lambda ()
           (safe-apply
            battery-monitor-procedure '()))
         'battery-monitor))
  #t)

(define battery-monitor-procedure
  (lambda ()
    (let loop ((bv (safe-battery-voltage)))
      (if (< bv *battery-charging*)
          (begin
            (subscribe 'untethered))
          (begin
            (unsubscribe 'untethered)))
      (sleep (* 60 1000))               ; check every minute or so
      (loop (safe-battery-voltage)))))

(define *load-monitor-thread* #f)

(define (load-monitor)
  (if (thread? *load-monitor-thread*)
      (begin
        (warn "Terminating running load monitor thread")
        (kill-thread! *load-monitor-thread*)))
  (set! *load-monitor-thread*
        (spawn-on-root
         (lambda ()
           (safe-apply
            load-monitor-procedure '()))
         'load-monitor))
  #t)

;
; threshold load, above which a system is no longer 'idle
; it's a string so that it can be safely sent among differently byte-orderd
; machines
;
(define *load-threshold* ".75")

(define load-monitor-procedure
  (lambda ()
    (let loop ((system-load (get-system-load)))
      (if (< (string->number system-load) (string->number *load-threshold*))
          (subscribe 'idle)
          (unsubscribe 'idle))
      (sleep (* 60 1000))               ; check every minute or so
      (loop (get-system-load)))))

;
; NOTE: returns the string representation of the load (so that I can
;       send it among the differently byte-ordered machines)
;
(define (get-system-load)
  (let* ((w-string (separate-string (run-process "/bin/sh" "-c" "w")))
         (load-string (caddr (member "average:" w-string))))
    (substring load-string 0 (- (string-length load-string) 1))))
```

A.7 new-task.scm

```
;
; task framework stuff
;


;
; a simple metric-maker.  it returns a procedure which will
; compare two uids and pick the one who is closest to the given
; target.  intended for use with sort-response.
;
; target should look like what reckon returns: '(x y theta 0)
;
(define (location-metric-maker target)
  (lambda (uid-info1 uid-info2)
    (let* ((uid1 (car uid-info1))
           (uid2 (car uid-info2))
           (reckon1 (cadar (request uid1 safe-reckon)))
           (reckon2 (cadar (request uid2 safe-reckon)))
           (d1 (reckon-distance-between target reckon1))
           (d2 (reckon-distance-between target reckon2)))
      (if (< d1 d2)
          #t
          #f))))

(define (load-metric-maker)
  (lambda (uid-info1 uid-info2)
    (let* ((uid1 (car uid-info1))
           (uid2 (car uid-info2))
           (load1 (cadar (request uid1 get-system-load)))
           (load2 (cadar (request uid2 get-system-load))))
      (if (< (string->number load1) (string->number load2))
          #t
          #f))))
;
; sort the responses according to metric
;
(define (sort-response response metric)
  (sort-list
   response
   metric))


;
; MODS:
; - need to move resubscribtion to at end (postamble) of request-exclusive
; - 2-step transaction
;      - return (#t uid (proc #t or #f)
;      - return (#f uid)
;
; pub/sub useful for lots of things
; awkward when good coord needed.
; demo some awkard examples
; propose extensions to pub/sub (exclusive/ 2-step)

; might want one vanilla god knows what will happen request and one
; bullet-proof all-exclusive request

; talk about 'non-exclusive locking' (like for sonar, etc)

; in vanilla, you could try it. if it fails, the just loop with
; other candidates.
(define (pose-task task-name num-robots capability-list metrics procs)
  (let*
      ; type-list indicates for each task component whether it
      ; needs exclusive resources or not
      ((type-list
        (map
         (lambda (topiclist)
           (cond
            ((and (not (null? topiclist))
                  (list? topiclist)
                  (list? (car topiclist)))
             'exclusive)
            (else
             'non-exclusive)))
         capability-list))
       ; filled-capability-list is the original capability-list but possibly
       ; with null lists added in order to make it exactly conform to
       ; the ((exclusive resources) (non-exclusive resources)) layout
       (filled-capability-list
        (let ((foo
               (map
                (lambda (topic-list)
                  (cond
                   ((not (list? topic-list)) (list '() (list topic-list)))
                   ((not (list? (car topic-list))) (list '() topic-list))
                   (else topic-list)))
                capability-list)))
          (display-remote 0 foo)
          (display-remote 0 type-list)
          foo))
       ; responses is what we got back from the initial request, which
```

```
   ; was either:
   ;   - an exclusive request to reserve resources, in which
   ;     case, we get back an encapped proc
   ;   - a non-exclusive request for the simple thunk t
   (responses
    (map3
     (lambda (topic-list type proc)
       (if (eq? type 'exclusive)
           (request-exclusive topic-list task-name proc)
           (request (cadr topic-list) t)))
     filled-capability-list type-list procs)))
(cond
 ((member #t (map null? responses))
  ; right...we didn't get everybody. so, we'd better go through
  ; and evaulate the procedures of anyone who was grabbed
  ; exclusively
  (display-remote 0 responses)
  (map2
   (lambda (response-list type)
     (if (eq? type 'exclusive)
         (map
          (lambda (response)
            ((cadr response) #f))
          response-list)))
   responses type-list)
  'not-enough-takers)
 (else
  (let* ((sorted-lists (pose-task-helper responses metrics))
         (final-task-uids (resolve-conflicts sorted-lists)))
    ;
    ; OK, we've sorted the resonses.  Now we actually need to get things
    ; done.  First, run through each of the backups in final-uids:
    ; if the task is exclusive then evaluate the encapped proc
    ; (third elt in response) with #f to effect an ABORT.
    ; Next, actually do the with-team with the winners.
    ;
    (map2
     (lambda (sorted-task-uids type)
       (if (eq? type 'exclusive)
           (map
            (lambda (runner-up)
              (if (not (member #t
                                (map (lambda (winner)
                                       (eq? (car runner-up) (car winner)))
                                     final-task-uids)))
                  ((cadr runner-up) #f)))
            sorted-task-uids)))
         sorted-lists type-list)

        (if (= num-robots 1)
            (request (caar final-task-uids) (car procs))
            (request (caar final-task-uids)
                     (with-team-proc-maker
                      final-task-uids type-list procs)))))))

(define (pose-task-helper responses metrics)
  (cond
   ((null? responses) '())
   (else
    (cons
     (if (not (null? (car metrics)))
         (sort-response (car responses) (car metrics))
         (car responses))
     (pose-task-helper (cdr responses) (cdr metrics))))))

;
; this proc takes a list of lists.  each list contains the (sorted) uids
; available for a single task. assuming the first task is the most
; important, resolve-conflicts picks exactly who should do what
; and returns a list of the uids in that order (or #f if it couldn't be done)
;
; for the future:
;   - would be nice to offer the user some general method for specifying
;     the relative priority of a task.
;
(define (resolve-conflicts sorted-lists)
  (let loop1 ((task-uid-lists sorted-lists) (final-uids '()))
    (cond
     ((null? task-uid-lists) (reverse final-uids))
     (else
      (let ((this-task-uid
             (let loop2 ((this-task-uid-list (car task-uid-lists)))
               (cond
                ((null? this-task-uid-list) #f)
                (else
                 (cond
                  ((member (caar this-task-uid-list) (map car final-uids))
                   (loop2 (cdr this-task-uid-list)))
                  (else (car this-task-uid-list))))))))
        (if (not this-task-uid)
            #f
            (loop1 (cdr task-uid-lists) (cons this-task-uid final-uids))))))))
```

```
(define (simple-sonar-task) (list 1
                                 (list
                                  (list
                                   (list 'mobile)
                                   (list 'sonar)))
                                 (list
                                  (location-metric-maker '(100 100 0)))
                                 (list
                                  (simple-sonar-task-thunk '(100 100 0)))))


(define (dual-sonar-task)
  (let ((target1 '(300 200 0))
        (target2 '(-1000 200 90))
        (num2 100))
    (list 'dual-sonar-task 3
          (list
           (list (list 'mobile 'sonar) '())
           (list (list 'mobile 'sonar) '())
           'idle)
          (list
           (location-metric-maker target1)
           (location-metric-maker target2)
           (load-metric-maker))
          (list
           (dual-sonar-task-proc-maker-1 target1)
           (dual-sonar-task-proc-maker-1 target2)
           (dual-sonar-task-proc-maker-2 num2)))))

(define (dual-sonar-task-proc-maker-1 coords) ;coords is '(x y theta)
  (lambda ()
    (display-remote 0 *subscriptions*)
    (sonar-start)
    (goto-xy (list (car coords) (cadr coords)))
    (rotate 'by (angle-to-heading (caddr coords)))
    (sonar-stop)
    (sonar-read)))

(define (dual-sonar-task-proc-maker-2 num) ;num is length of list to sort
  (lambda ()
    (sort-list
     (let loop ((num-left num))
       (if (zero? num-left)
           '()
           (cons ((make-maxrandom 100)) (loop (- num-left 1)))))
     <)))
```

```
;
; a proc to build a proc which will run with-team
;
;
(define (with-team-proc-maker task-uids type-list procs)
  (let ((proclist
         (map3
          (lambda (task-uid proc type)
            (if (eq? type 'exclusive)
                (lambda ()
                  ((cadr task-uid) #t))
                proc))
          task-uids procs type-list))
        (uids (map car task-uids)))
    (lambda ()
      (with-team uids
        (on-list ((fixed uids)) proclist)))))

;
; NOTES:
;
; Well, except for the macro I need to pass a list of procs as individual
; args to on, the with-team stuff looks pretty good.  Must make sure
; the same robot doesn't get picked more than once.  How to decide?
; It would be nice to determine some 'best fit' over the whole set of
; tasks.  Would be really nice for user to give some idea of weight to
; each task (i.e. relative to the other tasks, how important is it that this
; task get the best qualified machine?).  For now, I'll probably use the
; order in which the tasks are given as an implicitly declared (decreasing)
; priority list.
;

(define (simple-sonar-task-thunk coords)  ;coords is '(x y theta)
  (lambda ()
    (sonar-start)
    (goto-xy (list (car coords) (cadr coords)))
    (rotate 'by (angle-to-heading (caddr coords)))
    (sonar-stop)
    (sonar-read)))
;
; new version of 'on'.  it takes as the last arg a list of procs
;
(define-syntax on-list
  (letrec ((find (lambda (key default stuff)
                  (cond ((null? stuff) (list key default))
                    ((and (list? (car stuff))
                          (eq? key (caar stuff)))
                     (car stuff))
                    (else (find key default (cdr stuff))))))
(lambda (exp rename compare)
  (let* ((options (cadr exp))
         (fixed (cadr (find 'fixed ''() options)))
         (time (cadr (find 'timeout 0 options)))
         (collect (cadr (find 'collect 'and-collector options)))
         (unlikely (rename 'unlikely))
         (on-entire-team?
          (not (equal? (list unlikely) (cdr (find 'all unlikely options)))))
         (arb-clause (find 'arbitrary 0 options))
         (arb-n (if on-entire-team?
                   -1              ; means "select the entire team"
                   (cadr arb-clause)))
         (arb-from (if (or (not (= 4 (length arb-clause)))
                          (not (eq? 'from (caddr arb-clause))))
                      '#f
                      (list-ref arb-clause 3)))
         (debug (cadr (find 'debug #f options)))
         (error-handler (cadr (find 'on-error '%%team-error options)))
         (procs (caddr exp))                ; at least one proc!
         (local-var (rename 'fixed-robot-list)))
    `(,(rename 'let) ((,local-var ,fixed))
                    (apply %%exec-on ,local-var
                           (%%select-arbitrary ,arb-n ,arb-from ,local-var)
                           ,collect
                           ,time
                           ,debug
                           ,error-handler
                           ,procs)))))
```

A.8 publish.scm

```
;
; publish/subscribe things
;



;
; Hmm...there must be a global var somewhere which
; holds all the aspaces (or uids), but for now, we'll fake
; it.
;
(define *uids* (list 0 1 2 3))

;
; global var to hold topics we're currently listening to
;
(define *subscriptions* '())

;
; must lock the subscription list so that things like the battery-monitor
; thread may safely modify it
;
(define *subscriptions-lock* '())



;
; publish a message (asynchronous)
;
(define (publish topic proc . args)
  (map
   (lambda (uid)
     (remote-run! (uid->aspace uid)
                  receive-publication topic (aspace-uid (local-aspace))
                  proc args))
   *uids*))

;
; publish and wait for replies (synchronous)
;
(define (request topic proc . args)
  (map (lambda (elt)
         (list (cadr elt) (caddr elt)))
       (filter (lambda (response) (car response))
               (map
               (lambda (uid)
                 (remote-apply (uid->aspace uid)
                               receive-request topic
                               (aspace-uid (local-aspace)) proc args))
               *uids*))))

;
; publish and wait for replies (synchronous)
; NOTE: this one is meant for getting exclusive hold on a resource,
;       such as 'mobile.
;
; topic can be a list of atoms, in which case those atoms are
; interpreted as capabilities which are not needed exclusively.
;
; topic can also be a list of two lists, in which case:
;      first sublist: those topics needed exclusively
;      second sublist: those topics not needed exclusively
;
(define (request-exclusive topic task-name proc . args)
  (let ((topiclist
         (cond
          ((not (list? topic)) (list '() (list topic)))
          ((not (list? (car topic))) (list '() topic))
          (else topic))))
    (map (lambda (elt)
           (list (cadr elt) (caddr elt)))
         (filter (lambda (response) (car response))
                 (map
                  (lambda (uid)
                    (remote-apply (uid->aspace uid)
                                  receive-request-exclusive topiclist task-name
                                  (aspace-uid (local-aspace)) proc args))
                  *uids*)))))
;
; start listening to a topic
;
(define (subscribe topic . already-have-lock)
  (dynamic-wind
   (lambda ()
     (if (not already-have-lock)
         (obtain-lock *subscriptions-lock*)))
   (lambda ()
     (if (not (member topic (if (null? *subscriptions*)
                                *subscriptions*
                                (map car *subscriptions*))))
         (set! *subscriptions*
               (cons (list topic '()) *subscriptions*))))
   (lambda ()
     (if (not already-have-lock)
         (release-lock *subscriptions-lock*)))))

;
; subscribe somebody else to a topic
;
(define (subscribe-remote uid topic . once-or-always)
  (let ((topiclist
         (if (list? topic)
             topic
             (list topic))))
    (map
     (lambda (topic)
       (remote-apply (uid->aspace uid) subscribe topic once-or-always))
     topiclist)))

;
; stop listening to a topic
;
(define (unsubscribe topic . already-have-lock)
  (dynamic-wind
   (lambda ()
     (if (not already-have-lock)
         (obtain-lock *subscriptions-lock*)))
   (lambda ()
     (let ((new-subscription-list
            (filter
             (lambda (topic-record)
               (not (eq? topic (car topic-record))))
             *subscriptions*)))
       (set! *subscriptions* new-subscription-list)))
   (lambda ()
     (if (not already-have-lock)
         (release-lock *subscriptions-lock*)))))

(define (topic-status topic)
  (let ((result
         (filter (lambda (elt) elt)
                 (map
                  (lambda (topic-record)
                    (if (eq? topic (car topic-record))
                        (cadr topic-record)
                        #f))
                  *subscriptions*))))
    (if (null? result)
        #f
        (car result))))

(define (set-topic-status! topic status)
  (set! *subscriptions*
        (map
         (lambda (topic-record)
           (if (eq? topic (car topic-record))
               (begin
                 (list topic status))
               topic-record))
         *subscriptions*)))


;
; stop somebody else from listening to a topic
;
(define (unsubscribe-remote uid topic)
  (remote-apply (uid->aspace uid) unsubscribe topic))



;
; these will be remote-run'd by message senders
;


;
; this one will handle requests (i.e. synchronous and needing a reply)
;
(define (receive-request topic uid proc args)
  (if (not (member #f
                   (map
                    (lambda (elt)
                      (member elt
                              (if (null? *subscriptions*)
                                  *subscriptions*
                                  (map car *subscriptions*))))
                    (if (not (list? topic))
                        (list topic)
                        topic))))
      (list #t (aspace-uid (local-aspace))
            (ignore-errors (lambda ()
                             (apply proc args))))
      (list #f (aspace-uid (local-aspace)))))

;
; this one will handle exclusive requests
; (i.e. synchronous, needing a reply, and needing to grab a resource)
;
```

```
; topiclist is a list of two lists.
;      first sublist: those topics needed exclusively
;      second sublist: those topics not needed exclusively
;
;
(define (receive-request-exclusive topiclist task-name uid proc args)
  (let ((continue? #f)
        (unsubscribed-topics '()))
    (dynamic-wind
     (lambda () ;preamble; get the lock
       (obtain-lock *subscriptions-lock*))
     (lambda () ;body: edit *subscriptions*
       (let ((status-list
              (map
               (lambda (elt)
                 (if (or (eq? task-name (topic-status elt))
                         (null? (topic-status elt)))
                     #t
                     #f))
               (append (car topiclist) (cadr topiclist)))))
         (if (not (member #f status-list))
             (begin
               (map
                (lambda (elt)
                  (set-topic-status! elt task-name))
                (car topiclist))
               (set! unsubscribed-topics (car topiclist))
               (set! continue? #t)))))
     (lambda () ;postamble: release the lock
       (release-lock *subscriptions-lock*)))
    (if continue?
        (list #t (aspace-uid (local-aspace))
              (encap
               (lambda (commit?)
                 (let ((result
                        (if commit?
                            (ignore-errors
                             (lambda ()
                               (apply proc args))))))
                   ;resubscribe to subjects
                   (map
                    (lambda (elt)
                      (set-topic-status! elt '()))
                    unsubscribed-topics)
                   result))))
        (list #f (aspace-uid (local-aspace))))))
```

```
;
; this one will handle publications (i.e. asynchronous and needing no reply)
;
(define (receive-publication topic uid proc args)
  (spawn (lambda ()
           (if (not (member #f
                            (map
                             (lambda (elt)
                               (member elt
                                       (if (null? *subscriptions*)
                                           *subscriptions*
                                           (map car *subscriptions*))))
                             (if (not (list? topic))
                                 (list topic)
                                 topic))))
               (list #t (aspace-uid (local-aspace))
                     (ignore-errors (lambda ()
                                      (apply proc args))))
               (list #f (aspace-uid (local-aspace)))))))
```

A.9 safe.scm

```
;
; safe versions of robot-related commands
;
; I want to be able to safely do things like evaluate (battery-voltage)
; on any machine.  So, I'll use these intermediary procedures
; which do simple checks against hostname to determine what should
; actually be done.
;


(define *tile-size* (* 12 25.4))

;
; these coordinates will be reported by workstations when (safe-reckon)
; is invoked  (what are elements of reckon supposed to be?)
;
(define *plantain-position* (list (* -5.5 *tile-size*) (* 10 *tile-size*)
                                  -90 0))
(define *praline-position* (list (* -8 *tile-size*) (* 10 *tile-size*) -90 0))
(define *choctaw-position* (list (* -10 *tile-size*) (* -9 *tile-size*) 90 0))
(define *troy-position* (list (* -5 *tile-size*) (* -9 *tile-size*) 90 0))

;
; a new version of im
;
(define (safe-im)
  (if (robot? (getenv "HOSTNAME"))
      (im)
      #t))

;
; if we're not a robot, the report a value high enough to indicate
; that charging? is true
;
(define (safe-battery-voltage)
  (if (robot? (getenv "HOSTNAME"))
      (battery-voltage)
      140))

;
; ok...it seems like the workstations should be able to respond
; to reckon; they should simply report a static position
;
```

```
(define (safe-reckon)
  (let ((hostname (getenv "HOSTNAME")))
    (if (robot? hostname)
        (reckon)
        (cond
         ((equal? hostname "plantain") *plantain-position*)
         ((equal? hostname "praline") *praline-position*)
         ((equal? hostname "troy") *troy-position*)
         ((equal? hostname "choctaw") *choctaw-position*)))))
```

Bibliography

[AE97] Majid Nili Ahmadabadi and Nakano Eiji. Constrain and Move: A new concept to develop distributed transferring protocols. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2318–2325, 1997. Albuquerque, New Mexico.

[BF96] Ross L. Burchard and John T. Feddema. Generic robotic and motion control API based on GISC-Kit technology and CORBA communications. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 712–717, 1996. Minneapolis, Minnesota.

[Bro95] Russell Gregory Brown. *Localization, Mapmaking, and Distributed Manipulation with Flexible, Robust Mobile Robots.* Ph.D. Dissertation, Cornell University, May 1995.

[C^+96] Anhui Cai et al. Cooperative path planning and navigation based on distributed sensing. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2079–2084, 1996. Minneapolis, Minnesota.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. In *ACM Transactions on Programming Languages and Systems*, September 1995.

[J^+96] Yunho Jeon et al. An object-oriented implementation of behavior-based control architecture. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 706–711, 1996. Minneapolis, Minnesota.

[JKW98] James S. Jennings and Chris Kirkwood-Watts. Distributed mobile robotics by the method of dynamic teams. In 4^{th} *Intl. Symp. on Distributed Autonomous Robotic Systems*, 1998. Karlsruhe, Germany.

[JKWT97] James S. Jennings, Chris Kirkwood-Watts, and Craig Tanis. Distributed map-making using online generalized voronoi graph. Currently in review, 1997.

[JWE97] James S. Jennings, Greg Whelan, and William F. Evans. Cooperative search and rescue with a team of mobile robots. In *Proceedings of the International Conference on Advanced Robotics*, pages 193–200, 1997. Monterey, California.

[KZ96] C. Ronald Kube and Hong Zhang. The use of perceptual cues in multi-robot box-pushing. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2085–2090, 1996. Minneapolis, Minnesota.

[LjH95] Fang-Chang Lin and Jane Yung jen Hsu. Cooperation and deadlock-handling for an object-sorting task in a multi-agent robotic system. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2580–2585, 1995. Nagoya, Japan.

[LjH96] Fang-Chang Lin and Jane Yung jen Hsu. Coordination-based Cooperation Protocol in multi-agent robotic systems. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1632–1637, 1996. Minneapolis, Minnesota.

[MK97] J. Daniel Morrow and Pradeep K. Khosla. Manipulation task primitives for composing robot skills. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3354–3359, 1997. Albuquerque, New Mexico.

[MM97] Jane Mulligan and Alan K. Mackworth. Experimental task analysis. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3348–3353, 1997. Albuquerque, New Mexico.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Series in Artificial Intelligence. Prentice Hall, 1995.

[S$^+$96] J. Borges Sousa et al. On the design and implementation of a control architecture for a mobile robotic system. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2822–2827, 1996. Minneapolis, Minnesota.

[Tan96] Andrew S. Tannenbaum. *Computer Networks.* Prentice Hall PTR, third edition, 1996.

[TIB97] TIBCO Software, Inc. *TIB®/Rendezvous™ Concepts*, August 1997.

Biography

The author, Brian Gerkey, never has enough time to sleep.