Robust Agent Teams

by

Peter C. Barnum

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Bachelor of Science

Supervised by

Chris M. Brown and George Ferguson

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2005

Acknowledgments

I would like to thank my advisers, Chris Brown and George Ferguson, for helping guide me on my research journey. I appreciate all your questions and comments, and help with getting the most out of my research experience.

Thanks to Randal Nelson for introducing me to the world of research, and showing me how much fun it could be to work on something that hasn't been solved. To Bo Hu and Dave Costello, I had a great time working with you guys and I learned a lot from both of you. To Marty Guenther and the rest of the CS staff for making everything go so smoothly during my years here at UR.

Last but definitely not least, thanks to my family and friends for being so supportive and entertaining!

Abstract

There are certain tasks that can be completed better by a team and certain tasks that can only be completed by a team. To have efficient teamwork in a group of artificial agents, a coordination system must be used that is tolerant of mechanical errors and uncertainties in the world. In this thesis, I present two methods that each solve a different aspect of the robust teamwork problem.

The first is tolerance to failures in communication. I suggest the use of a large group of mobile software agents, with each responsible for a single goal. The agents can jump from host to host when there is a line of communication, but stay still when there is none.

The second is tolerance to uncertainty in the environment. By using a combination of Borda count voting and auctioning, the team is not easily confused by bad sensor data. A fast method of calculating the count is presented, and the two-step process is shown to be efficient and robust.

At the end, I suggest a way to combine the two, by integrating dynamic team membership with voting and auctioning. By taking advantage of parts of both techniques, maximum robustness is assured.

Table of Contents

List of Figures

1 Introduction

A question that is sometimes asked about distributed system is “why bother?” At first glance, it seems that it is simpler to just take a single supercomputer, crank out the optimal solution, and tell each unit what to do. However, in situations with poor communication, a dynamic environment, or any time that speed is required, this centralized method can lead to units spending much of their time sitting idle while the central computer calculates their next move and successfully communicates it. With a well-distributed system, units can be partially autonomous from each other, allowing for fault-tolerant activity and fast adaptation to environmental change.

In this paper, I explore two areas and techniques for distributed decision-making. The first is tolerance to communication error, which is a common problem when using wireless communication hardware. The second is tolerance to uncertainty in sensory input, which is present in some form in all kinds of real-world problem. I approach these problems separately, to try to get at the core issues with each, although in the conclusion, I propose a way to make a hybrid of the two.

1.1 General teamwork issues

When an autonomous agent decides to work on a task, it has to consider various issues and constraints in order to act efficiently. For instance, if such an agent is responsible for building a house, it may have to consider things such as whether there is enough lumber around and whether it is strong enough to lift any large rocks off the site. Alone, this is a complex and not completely solved problem, although there have been some very successful attempts [Rao and Georgeff, 1995][Gat, 1992][Mitchell, 1990][Hayes-Roth, 1995]. However, even if a single agent has the ability to solve a problem completely by itself, it may be more efficient for several to work together, which raises a host of new challenges and opportunities. Take lifting rocks off the site as an example. Which of the agents should do the lifting? Should it be the stronger one, the one that is less skilled in carpentry, or simply the one that is closest? Would it be better if they cooperated in lifting large rocks, or perhaps more importantly, how do they insure that they do not get in each other's way?

An important component of effective teamwork is the establishment of a common language and intention. Theories such as [Cohen and Levesque, 1990] [Cohen and Levesque, 1991] [Grosz and Sidner, 1990] [Kinny *et al.*, 1992] [Jennings, 1995] guarantee that a team will either remain cohesive, or collectively disband. Specifically, each agent must individually:

- Believe that a recipe (a set of actions that will complete a goal) can be accomplished

  Each agent has to understand enough about the environment, its own abilities and the abilities of others, to allow a belief in the possibility of success. Even before beginning to coordinate with others, each agent must have a separate belief. It is still possible that during the planning or execution stages, it will discover that the recipe cannot be completed.

- Decide that a recipe is better accomplished collectively

Even if it is known that a group can do something, this does not mean that it should be. Every agent involved needs to agree that the benefits of cooperation outweigh the costs in time and other resources. If they are laying a concrete foundation to a house that requires the movement of a ten-pound concrete bag, then it would be foolish to have three agents all working together on the same bag. It would be faster for a single agent to move the bag itself, rather than have two others get in the way.

- Believe that other agents have decided to work together on the recipe

  All the planning in the world does not matter if others have not agreed to help. In stigmergic systems [Grassé, 1959] [Theraulaz *et al.*, 1990], belief in cooperation is inborn in the organism, but most others need to have a mechanism to allow the intentions to be explicitly expressed or understood, even in the absence of communication [Huber and Durfee, 1995] [Wie, 2000].

- Update the beliefs of other agents if the recipe cannot be accomplished or the agent has stopped working on the recipe

  The reverse of communicating intention to act must also be taken into account. If for any reason, an agent is going to stop working, then it should update the others' beliefs to allow them to re-coordinate.

  Sometimes an agent will fail so completely that it is unable to even communicate its failure, such as if a robotic rover's battery fails, so ideally agents should be able to update their beliefs even in the absence of communication. However, for efficiency, failures should be communicated explicitly whenever practical.

- Have a recipe that will not conflict with others' recipes

  Often this means that each agent either knows what the other agents' recipes are, or else all of the recipes are designed so that they will not conflict. However, neither of these necessarily has to be true. For an example, suppose Shelly and

John are building a wall that requires wood and paint. Shelly could have a recipe for one person to go to the hardware store and get paint while the other goes to the lumberyard and gets wood. John's recipe could also have one person getting paint at the hardware store, but instead have the other get lumber by chopping down a tree and milling it themselves. Although the recipes are different, neither will conflict with the other as long as one of them gets the lumber and one gets the paint, and they both will be completed.

The main reason to have global knowledge of recipes or specifically designed recipes is to prevent low-level actions from conflicting with each other. If neither of the two is practical, then such conflicts can be approached individually. Again with the wall scenario, if Shelly wants $15 to buy wood, John wants $10 to buy paint, and they only have $20 between them, then the recipes will conflict. There is no one way to solve problems with this specific type of conflict, but if they are not addressed by the recipe designer, then time needs to be spent at runtime to find and eliminate them.

- At all times during execution, to be working on the recipe and having the belief that others are as well

  This ties back into the need to communicate the end of cooperation as well as the start. If an agent wants to either help another or else decide abandon a recipe, it should have at least an abstract understanding of what the others are doing. Certain elements of recipes could be useless or detrimental to achieving the goal if every agent is not working together.

In addition to the above criteria, if a teamwork theory is actually to be instantiated in a real team, then each agent must have additional properties in order to be robust. They must:

- Have a way to rank different options.

Unless every possible action is equally valuable, then an agent needs to be able to decide which to do. Implementations such as [Smith, 1980][Davis and Smith, 1983][Wie, 2000][Gerkey and Mataric, 2002][Dias and Stentz, 2003] all provide useful metrics for such decisions, and have been used successfully in many circumstances.

- Have an efficient method for deciding upon and distributing tasks that can operate in real-time.

  It is tempting to take enough time to make a complete decision and communicate the full intention to everyone, but this is rarely a valid option in the real world. Due to a dynamic environment, or simply because it is not a good use of time to just sit around, suboptimal decisions have to be made. Even given those constraints, the closer that the solution can approach to optimal, the better the team will perform. A successful instantiated system must be able to draw a balance between the two.

- Complete the goal, even if the environment is not as expected.

  Uncertainty is an accepted fact when operating in the real world or any other complex environment. Sometimes the best plans fail or do not perform as expected. This uncertainty due to incomplete knowledge of the environment needs to be combated by either manually making error-tolerant plans, or else endowing agents with the ability to handle probabilistic reasoning. Such reasoning has been used in many systems, such as [Wie, 1997][Wie, 2000][Mataric; *et al.*, 2003].

A teamwork system must take the above criteria into account, either explicitly through rules and procedures, or implicitly through the design of the teamwork mechanisms. In this thesis, I mainly examine uncertainty, in the context of tolerance to communication and sensory error.

1.2 Communication error

Having tolerance to communication disruption, yet also having the ability to dynamically alter recipes, is important for any successful multi-agent team. It is possible to have a very robust system by not allowing agents to communicate, but efficient coordination requires it. When groups of agents are interested in common goals, they must be able to work together, or else it may take a long time to complete their recipes. But at the same time, they should have enough autonomy from each other that they do not have to rely on communication when it cannot be guaranteed.

For tolerance to communication error, I present a general framework for the use of software agents to coordinate a wide variety of applications, from network traffic to high-level control of physical robot teams. To evaluate the theory, I tested an implementation on a rover simulation with sporadic communication in a simulated environment [Brown *et al.*, 2004], with a single rover able to complete each recipe.

1.3 Sensory error

For teamwork that is powerful enough to complete complex tasks, some robustness to communication failure needs to be traded for increased abilities. If agents cannot communicate while working on the same joint recipe, then it is difficult to coordinate interrelated behaviors. Some attempts such as [Huber and Durfee, 1995] [Balch and Arkin, 1997] [Wie, 1997] [Wie, 2000] have been successful. However, if there are a lot of possible activities for individuals in a large number of joint recipes, then such communication-free methods become decreasingly effective. (As an example, imagine fifty similar recipes, each with spots for ten agents. Differentiating between similar activities would require a tremendous amount of semantic knowledge and observation time, if it were even possible.)

For tolerance to sensory error, I present a theory and practical implementation,

named DAVS (Distributed Auctioning and Voting System.) Using a combination of voting and auctioning methods, combined with probabilistic reasoning about utility values, it allows for fast, efficient teamwork on recipes that can each require multiple agents. I demonstrate DAVS's capabilities in a simulated environment [Brown *et al.*, 2004]. The theoretical basis of DAVS is especially useful when not all agents agree on the utility of certain goals, some are untrustworthy, or some are just in error. Although error-tolerance is a common goal, little has been done specifically on making efficient implementations of autonomous agent teams that operate for the common good, even in the presence of defectors. There is a large body of theoretical work examining the same. (It is impossible to list even a selection of the work done on game theory and the like, but some useful surveys are [Axelrod, 1984][Fudenberg and Tirole, 1987][Moulin, 1995][Klemperer, 1999].)

2 Previous work

2.1 Core teamwork theories

Various researchers have theorized about how to solve coordination issues by creating unambiguous logical frameworks. Each addresses a subset that they view to be most important. Actually implementing such theories often requires extra manipulation, but having a generalized model is vital for both understanding how teams can function and how to allow them to do so. Figure 2.1 shows a comparison of the different techniques, including my proposed system, DAVS. There are dozens of teamwork implementations, but the table shows a sample that covers the major areas.

	Reasoning	Communication	Recipe Failure	Utility Ranking	Trust
Davs	Symbolic	Explicit	Renegotiate	Yes	No
Planned Team Activity	Symbolic	Explicit	Renegotiate	No	Yes
Rhino	Symbolic	Implicit	New Hypothesis	Yes	Yes
Steam	Reactive	Explicit	Joint Goals	No	Yes
Subsumption	Reactive	None	N/A	Yes	N/A

Figure 2.1: Comparisons between several different teamwork systems

2.1.1 Joint Intentions

It is difficult to say which was the first authoritative work in Computer Science on the fundamentals of teamwork, but joint intentions theory is probably the most well known and practical [Levesque *et al.*, 1990] [Cohen and Levesque, 1991]. In essence, if a group of agents have a joint intention if they all believe that they all are intending to complete a task. The central parts of the theory are:

- Persistent Goal (PG)

  If an agent has a PG, then it believes that a certain goal is uncompleted, can be completed, and that it will continue working toward the goal until either it is unachievable or has been completed. Note that it is not necessary for the agent to actually do anything to complete the goal, just that it will be done somehow or else become irrelevant.

- Weak Achievement Goal (WAG)

  A WAG is essentially a PG, except that it involves a team. If any agent with a WAG stops having the goal, then it has a new PG of informing its teammates of its termination of intention. The new PG is always added if the agent either becomes unable or simply unwilling to work on the goal, or else the goal becomes irrelevant.

- Joint Persistent Goal (JPG)

  Once again building on previous theories, a JPG is a WAG that also has every agent believing that every other agent on the team has the same WAG.

- Joint Intention (JI)

  A JI means that not only does every agent have the same JPG, but that while executing, they each believe that all others are working toward the JPG. In summary, a joint intention is where a group of agents all believe that they all are working

toward a goal, and that if the goal becomes completed or irrelevant, that they make each other aware of the change in intention.

2.1.2 Joint Responsibility

Jennings extended joint intention theory to add that as well as having the shared belief in a goal, a team of agents should also have a shared plan to accomplish it [Jennings, 1995]. This combination of intention and planning is called a joint recipe, and requires that all agents on the team have a JPG for a goal, have a common method of attaining the goal, and that they all are aware that they are all working on attaining the goal.

Establishing a joint recipe is done by a series of steps:

- An agent, dubbed the organizer, sees that there is a goal that requires a team to complete.
- The organizer contacts other agents to helps an awaits the responses
- When a response is received that an agent is willing to help, then the organizer adds its resources to the pool
- When the pool of resources becomes full enough to execute the goal, or else enough rejections have been received that the goal cannot be accomplished, then activity begins

2.1.3 SharedPlans

Created independently of Joint Intentions, SharedPlans addresses many of the same issues [Grosz and Kraus, 1993] [Grosz and Kraus, 1996] [Grosz and Kraus, 1998]. SharedPlans focuses more on the exact actions that agents take to complete a goal, rather on just their beliefs. As I have adopted in this thesis, they use “recipe” to mean

a set of steps that will complete a goal, while a “plan” includes the intention to do the steps.

They define individual and shared plans in both partial and full types. A full plan is one that a group of agents have all agreed to work on. It is not necessary for every aspect of the plan to be understood by every agent, nor even that an agent knows exactly what steps for its part in the plan. It is only necessary to have mutual belief that the plan can succeed and that unknown portions will be filled in during execution.

A partial plan is a full plan that is incomplete in either the team’s intentions or if the series of actions needed to attain the goal is unknown. Partial plans include actions that can be taken to convert it into a full one, provided proper understanding and environmental conditions. Although full plans cannot conflict with each other, this is possible for partial plans. As the team has not yet committed to a partial plan, they only have to be concerned with plans to which they have committed.

By making plans hierarchical, Hunsberger [Hunsberger, 1999] [Hunsberger, 2002] devised a way to make plans respond better to uncertainty and function with each other more efficiently. Hunsberger makes each part of a SharedPlan recursive, in both partial and full varieties. This allows for more complex, interweaving plans, while still staying within the general SharedPlans framework.

2.2 Teamwork implementations

It is important to have a theoretical basis for teamwork, but many aspects of abstract theories prove to be difficult or impossible to actually implement, as well as sometimes failing when real-time action is required [Pynadath and Tambe, 2002]. Some focus on creating robust plans ahead of time, while others will simply react to the environment, while others will do a hybrid of the two [Reich, 1997]. Many have an explicit mechanism for handling noise in sensor information [Mataric; *et al.*, 2003], while others simply react to the best of their ability.

2.2.1 Planned Team Activity

Kinny et. al. [Kinny *et al.*, 1992] [Sonenberg *et al.*, 1994] devised a method for planned team activity that revolves around having joint plans and intentions to do them, then decomposing the plans into elements for individual agents to solve. They composed plans in a form easily representable by a directed graph, with multiple activities that can be done in parallel. Each plan has certain synchronization points, where either the action cannot be done until all of the previous are completed, or else the subsequent action cannot be started.

That the logic automatically simplifies team plans into individual ones is both good and bad. It is helpful because each individual knows exactly what it is doing, but it makes it more difficult to re-plan in case of localized failure or for added efficiency.

2.2.2 Rhino

Instead of explicitly communicating intention, Van Wie [Wie, 1997][Wie, 2000] suggests that agents should observe each other and attempt to work on the same thing by forming hypotheses about what others are doing. He uses a Bayesian analysis that takes the visual evidence and quickly calculates what they are working on, eventually causing the entire group to converge upon a single task, without having to make a single communication.

2.2.3 Steam

Steam [Tambe, 1997a] [Tambe, 1997b] is partly modeled off of joint intentions [Levesque *et al.*, 1990] [Cohen and Levesque, 1991], SharedPlans [Grosz and Kraus, 1993] [Grosz and Kraus, 1996] [Grosz and Kraus, 1998], and Soar [Newell, 1990] [Laird *et al.*, 1987] [Lehman *et al.*, 2004]. Steam has procedures for automatically synchronizing and repairing plans made with joint intentions, as well as a redefined

communication scheme for practical application. In addition, it uses operators in the same way as Soar, with preconditions, rules for application, and a description of when to terminate.

Steam has recently been extended into Machinetta [Scerri *et al.*, 2004], which tries to increase usability by using more advanced proxy techniques to manage large groups. As well as allowing completely independent agents, it also supports adjustable autonomy for partial human interaction [Chalupsky *et al.*, 2002]. And although still using Soar-style procedures, Machinetta no longer relies on it and is now implemented in Java.

2.2.4 Subsumption

Brook's Subsumption architecture [Brooks, 1986] has proved to be one of the most effective for controlling physical robots in autonomously performing simple behaviors. Behaviors are arranged in a hierarchy, with higher levels having the ability to suppress the output of lower one. That is, if a robot is doing a task and something more important comes up, it will switch to the other behavior. All behaviors use augmented finite state machines, which allow fast calculation even on mobile rovers with little on-board computing capacity to react quickly.

In addition to the standard architecture, global values modeled on animal hormones can additionally be used to augment or suppress behaviors [Brooks, 1991]. In this way, complex behaviors can interact with a uniform hormonal model, without requiring addition design or expensive on-line analysis.

2.3 Agent architecture design

If an agent is not designed well by itself, it will not be able to work well with others. Various layered control systems have proved to be robust and modular, and

could allow for simple modifications to allow them to be effective team members as well as individuals.

2.3.1 AIS

In the Adaptive Intelligent Systems model [Hayes-Roth, 1995] [Hayes-Roth *et al.*, 1995], Hayes-Roth et. al. uses a layered architecture with message passing between layers to allow a high level of parallelism. For data sharing, all layers make exchanges through a blackboard-style procedure.

At any one time, a large number of layers can be active at one, interacting with each other and exchanging information, instead of requiring layers to shut down when not in use. This means that even if a layer is not the locus of control at a particular time, it can still motivate decision-making and behavior. Internally, each layer has behaviors that actually do action, an information base, pre-built plans for performing complex actions, and a meta-controller that activates the plan that is most pertinent.

2.3.2 Atlantis

Atlantis [Gat, 1992] takes advantage of both reactive control and high-level planning to make agents that are both robust and able to perform complex actions. It is split up into three components (arranged from simple to complex,) the controller, the sequences, and the deliberator. The controller handles low-level actions and reactive control. All instructions go through the controller, which actually performs and monitors the execution of simple actions. The sequencer is built directly above the controller and directs it to begin and end simple activities. Finally, the deliberator handles high-level planning, long-term goals, and updating the agent's overall model of the world. The deliberator's activities are the slowest and liable to task the most processing power, but by definition, do not have to be done quickly.

Splitting the system up gives all of the benefits of fast, reactive execution with the power of long term planning. Any number of different architectures could be built on this general model, with direct applicability to agent that operate in the real world.

2.3.3 BDI agents

BDI stands for Belief Desire Intention, and represents a framework for intelligent agents that is easily realizable in practical applications [Rao and Georgeff, 1995]. BDI has often been criticized for splitting up attitudes three ways, both because it is less elegant than one, and that it may not be possible to properly model everything with that particular three-way split. As stated by Rao et. al., the real benefit is when activity needs to be done in real time. Have a single, unified cognitive model tends to be far slower than splitting it up.

On the specific side, BDI uses decision trees to represent possible worlds. Each of the three attitudes has their own tree for their own worlds. Using the three types, agents attempt to find a balance between continuing in complex actions and reconsidering plans, all of which is done at a fast rate in real time.

2.3.4 Prodigy

Prodigy [Veloso *et al.*, 1995] is a planner that incorporates several learning mechanisms for creating robust plans. It splits up incomplete plans into two parts, the first being a total-order plan for accomplishing part of the goal, and the second being a partial order plan for the rest, which is built by using backward chaining.

Learning in Prodigy can by accomplished in one of five ways, EBL, static, dynamic, alpine, and analogy. Essentially, it is able to learn low-level reactive rules, high-level understanding rules, and even learn from observing itself or others. Such a wide variety of learning mechanisms increases the robustness of plans and increases the likelihood that they will be fast an efficient.

2.3.5 SOAR

It is difficult to say if Soar [Newell, 1990] [Laird *et al.*, 1987] [Lehman *et al.*, 2004] is mainly a theory or an implementation, as it is definitely some of both. In any case, it is described as an "Architecture for Human Cognition," meaning that the designer suggest that not only is it a practical method for agent design, but perhaps even the way that people think. Soar was originally designed to unify all of the various theories of cognition into a single whole, but has since evolved into an agent development platform.

Soar works by trying to solve goals and recursively subgoals. An important element of Soar is learning by chunking, which permeates all layers of the design. When a goal cannot be solved based on the information known (called an impasse,) the decision-making apparatus backs up until it is able to find the solution in the agent's knowledge base. Once found, a new production is made, linking the new knowledge directly, in a process known as "chunking."

2.3.6 Theo

The core of Theo [Mitchell, 1990] is its ability to learn complex actions and convert them into reactive rules. Whenever a Theo agent is confronted with a situation that it does not have a rule for, it performs a high level planning search to find a solution. When a solution is found, a new rule pairing the state with the action is made, in a similar way to chunking in SOAR. Over time, increasingly less processing is done as increasingly more rules are added and less new situations arise.

3 What is an agent and what does it do?

What is an agent? And what is a goal? Questions such as these are difficult to give a good answer for, as there are a variety of correct definitions. However, it is necessary to at least have consistent definitions. So to begin, I suggest that as a minimum, an agent must be able to:

- **Change its internal state due to external stimuli.** This could be as simple as automatically deciding to do an action reflexively. Even if something is following a complex program, I argue that to be defined as an agent, the environment have some effect on it.
- **Affect the world in some way.** Affecting the world might involve the agent moving its robot body or alerting a human user by putting text on a screen. Combined with the first requirement, an agent is something that interacts with the world.
- **Have at least a degree of autonomy in affecting the world.** In at least some cases, an agent has to be able to affect the world without explicit, external permission. Otherwise, it is simply a piece of tele-operated hardware.

Note that in the first condition, I use the word “decide” which seem to imply a thought process. It is beyond the scope of this paper to fully define thought, although informally, it can be thought of as an unobservable process that causes an agent to perform an

observable action. As an example, I imply that if the doctor hits your knee with a hammer, your mind's internal state changes, causing your knee to jerk.

For practical purposes, an agent is an entity that wants certain things in the world to be certain ways (in other words, it has **goals**.) For instance, an apple-picking agent could have a goal to have a bag full of apples. A goal does not necessarily give any indication about how it will be made true, while a **recipe** does. A recipe is the pairing of a goal with the steps to accomplish it. With the method in Section 5, each agent has a single recipe, which is defined by a hybrid automata (Section 5.6.) Utility calculation and execution are all encapsulated within the automata.

Although the automata were effective, I used a different technique when experimenting with tolerance to sensory error. Instead of using a recipe with actions fully encapsulated inside automata nodes, I use ordinary Java code inside a class, plus tuples for preconditions, execution constraints, and effects. Although more difficult to debug, it was easier to make complex, customizable actions.

When working with software agents, the common term for the place where the software agents reside is a **host**. Although a host could almost be considered a kind of agent, I will only use the word host to refer to the concept of a place for software agents. In addition, in this paper, a host will be inseparable from a rover.

I use agents in two different ways. In the first section on tolerance to communication error, there can be several agents in a single rover host, although only one of them is controlling the host at one time. In the second section, each agent is inseparable from a rover.

3.1 Tolerance to different types of error

The most central and important aspect of this work is how agents can be built that have the ability to work with others, yet still function well when the world is working against them. There are a variety of types of errors that crop up when working in the real

world. Some of the most common are to do with communication, imprecise sensory data, and imprecise or malfunctioning motor control. I mainly focus on the first two, as I feel that they are the most prevalent across all kinds of hardware and software systems. Exactly what constitutes them can vary. Bad sensory data could mean anything from a low resolution camera to misestimation of the time for a packet to reach its destination. Failures in communication can happen anytime a wireless link is used, causing any amount of signal noise or loss. Errors in motor control are certainly a serious issue in applications such as mobile robotics. Although I use a simulation of mobile robots to demonstrate my theories, I have tried to focus on tolerance to the most generalizable types of error, rather than just those directly related to the problem.

3.2 The principles of auctioning

Single-winner auctions are some of the fastest ways for a group of agents to make a decision. I make use of auctioning in both of my systems, although in different ways. There are a variety of systems that use auctioning to coordinate the behaviors of multiple agents, such as [Zlotkin and Rosenschein, 1996][Zlot *et al.*, 2002][Smith, 1980][Dias and Stentz, 2003][Gerkey and Mataric, 2002]. To participate in an auction, an agent calculates how much the auctioned item will benefit its utility, and sends a bid to the auctioneer. The auctioneer is not required to know why any of the bids are what they are, only that it should select the highest. In this way, a single number can communicate all of the understanding of desires and abilities.

Although the same type of single-winner auctioning is used in both Section 5 and Section 6, it is used in different ways. In Section 5, the host uses auctioning to decide which of the resident mobile agents should get to operate the rover. Each of the agents submits a bid (equal to its estimated utility) to the host, which selects the highest and gives the winning agent control of its rover. Each host can only be an auctioneer, and each agent can only be a bidder.

On the other hand, in Section 6, agents are both auctioneer and bidders at the same time. When each of the agents in a group has decided on which recipes to pursue, they send their bids to the others. Each agent receives all of the bids, and acts as an auctioneer to decide on which ones they should work.

Although seemingly different from each other, in both of the sections, auctioning is used for deciding about problems of about the same size. In the first case, each host has a small subset of all the mobile agents, and so only has to consider a portion of the possible recipes. In the second case, the voting step removes most of the possible recipes, and the agents only have to consider a few.

3.3 The simulation platform versus agent design

I use URQuake [Brown *et al.*, 2004], which is based on Id Software's Quake 2. The built in Quake monsters have been replaced by **quagents** (Quake agents), which give programmers a simple interface with which to control the simulated robots. A quagent is split into two components. One part is the **bot**, which is the physical representation of a robot inside of the URQuake simulator. The **quagent controller** is the intelligence that directs the bot by sending commands and receiving sensory input via a socket connection with the simulator. The quagent controller and URQuake are independent processes and can be run on different machines over a network connection.

Both systems use the same simulation platform (explained in more detail in Section 4.) However, the quagent controller architecture is quite different in each case. As explained earlier in this chapter, the main difference is the way the recipes are formatted. I used just the TeamQuagent API for Section 5, but later upgraded it to the AdvancedTeamQuagent, which I used for Section 6 (I explain both APIs in Section 4.)

4 Modifications to URQuake

The URQuake [Brown *et al.*, 2004] simulation platform is a useful and robust method for analyzing the effectiveness of agent design. Unfortunately, in its original form, it proved to be difficult to have several agents interact and to have items of the same type be distinguishable. I made a few small modifications that allow for increased functionality.

4.1 Creating uniqueness

In URQuake, there are large groups of objects, from items to quagents to walls and doors. Although Quake has an internal representation of each that separates it from others, a basic quagent can only see each object's item type (e.g. one health box looks the same as any other.) This representation works fine as long as quagents do not need to communicate or cooperate. However, imagine the dilemma if there were three quagents of the same, and one wanted to communicate to another. It would need some way to tell its communication protocol exactly who to talk with. To allow unambiguous message sending, I gave each quagent a unique ID number.

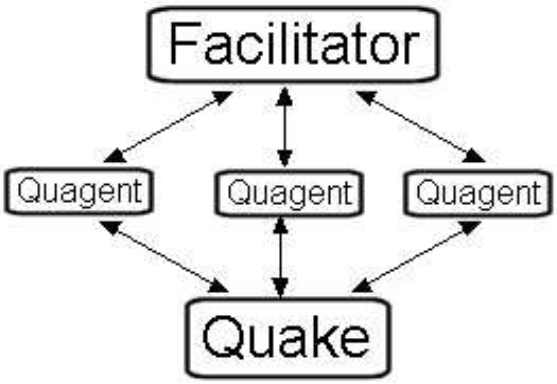

Figure 4.1: The interaction between quagents, Quake, and the facilitator

4.1.1 Unique Quagent IDs

Quagent rovers are made up of two parts. The first is the simulated physical rover in Quake, and the second is the controlling intelligence, which communicates with its rover over a socket connection, in the same manner as standard quagents. In my initial implementation, quagents can interact physically with each other, (mostly by getting in each other's way,) but had no way of coordinating their actions via communication. In keeping with the policy of not putting high-level functionality, especially that related to research, directly into the Quake engine, I created an external facilitator program through which each quagent can communicate with the others.

The facilitator has socket connections to the user-written quagent controller code (Figure 4.1.) If one quagent wants to talk with another, it sends the message through the facilitator, which then passes it on to the intended recipient. The facilitator simplifies quagent-quagent communication, as each only has to have a single, permanent socket connection, instead of needing to handle connections to every other quagent.

When a quagent looks around in the Quake simulator, it can see the ID numbers of other quagents. Using this ID, it can send messages to the quagent it sees, even though there is no direct link between Quake and the facilitator. In Figure 4.2, two bots see each other and their respective IDs. Quagent 5 wants to offer a trade to the bot it

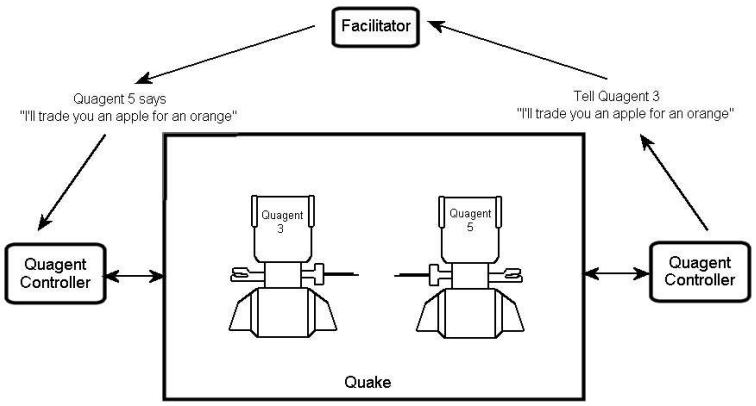

Figure 4.2: An example of interaction between two quagents that are communicating through the facilitator

sees, so it sends the message through the facilitator. Because the IDs in Quake and the Facilitator are synchronized at start-up, message passing is unambiguous, although it has a layer of indirection.

4.1.2 Unique Items

Each standard Quake item is identical to every other item in its class (e.g. every tofu is the same as every other tofu.) Sometimes, however, it is desirable to have an object with properties that are subtly distinct from others, or else are modifiable as the simulation progresses.

Modification is perhaps the most useful aspect of unique items. For example, suppose there is a need for a pile of 100 gold coins. The standard Quake would need to spawn 100 separate coin items. On the other hand, with unique items, only a single item, "pile of coins," would need to be created. The pile could be made so that as quagents take coins, the number of coins remaining diminishes. In this way, implementation is simplified and design becomes more intuitive.

Another useful property of unique items is that it is simple to create a large number of fairly similar items, without having to define a new item class for each one. Con-

tinuing with the gold coins example, it is trivial to make piles with from one to one thousand coins with a unique item, by modifying only a single number in the configuration file. By contrast, with standard items, a thousand item classes would have to be manually defined for the same functionality.

4.1.3 Trading items

Trading items requires the use of both quagent and item IDs. Expanding on Figure 4.2, two quagents use the facilitator to arrange a trade. Eventually, they mutually decide for Quagent 5 to give an apple and for Quagent 3 to give an orange in return. The actual "physical" items are stored in the Quake simulator, and the manipulation of inventories must happen there.

I modified the Quake code to add a protocol to trade items, money, and information. A trade is a transfer of any or all of the three between two quagents. Only items are handled in the actual Quake program; money and information, being new to the Quake ontology, are handled in the quagent controllers. If a trade is completed successfully, then Quake will send a message telling the quagents to transfer funds or information. It is impossible for a quagent to renege on an item trade, as it is handled directly by Quake, but money and information require good faith (or "honest" quagent controllers.) The issue of a dishonest or malfunctioning quagent is a separate problem that will not be handled here. Any number of protocols to decide on trades could be used, with no additional modification to the Quake code.

In the basic example given, the trade involves only items. Suppose now that Quagent 3 has no oranges, but instead offers to trade $50 and the location of an orange grove for the apple. Quagent 5 feels that this is fair, and they begin the trade protocol.

The modified Quake requires that bots give permission for inventory modification, to prevent stealing. So to begin, Quagent 3 must tell Quake to allow Quagent 5 to modify its inventory to give the apple. It tells Quake a message such as: "Allow Qua-

gent 5 to give me an apple. I agree to give $50 and the location of an orange grove in exchange." Quake receives this message and adds it to a list of acceptable trades. At this point, Quagent 5 can perform the actual trade. It tells Quake: "Give Quagent 3 an apple. Request $50 and the location of an orange grove in exchange." Assuming that they are in range of each other, Quake will move the apple from Quagent 5's to Quagent 3's inventory, then send a message to Quagent 3 ordering it to send the money and information. Quagent 3 will then transfer the funds and information via the facilitator, completing the trade. If for some reason, the acceptable trade and the trade attempt are not the same, then the trade will fail and nothing will be exchanged.

4.2 The TeamQuagent package

The TeamQuagent package contains low-level routines for quagent functioning, including a Java class to extend and a separate facilitator for communication between TeamQuagents (Figure 4.3.) In the picture, the relationship between different pieces of code can be seen. In the customized code section, a developer can create individualized functionality, while using the TeamQuagent package to communicate and coordinate with other Quagents and Quake. The communication and quake event modules collect events asynchronously, and high-level commands can be sent with the methods modules.

A message between quagents can contain an arbitrary collection of Java Objects, wrapped with the TeamMessage class. If a quagent wants to send a message, it makes a new TeamMessage of the appropriate type, and passes it to the communication method, which sends it on to the facilitator. As an example, a text message can use the built in extension TextTeamMessage, which allows the sending of a String Object.

To interface with Quake, TeamQuagent provides a group of methods to send commands. For instance, calling the runby(int distance) method tells Quake to make the bot run. TeamQuagent creates the appropriate string in each method, and passes that to

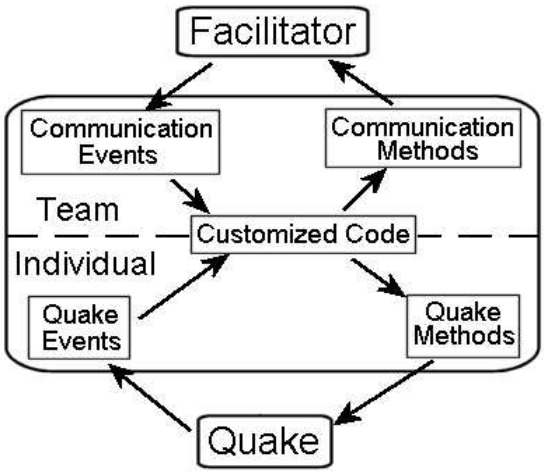

Figure 4.3: The inner working of event handling in the TeamQuagent API

Quake.

TeamQuagent has two event queues, one for Quake and the other for messages from other quagents. While running the main control loop, a TeamQuagent periodically can check the queues and handle any events or messages that came up. A Quake message is the result of a scan or trade, or an unexpected event such as running into a wall and stopping.

By using the quagent-to-Quake and quagent-to-quagent methods, a fairly sophisticated bot can be constructed without having to get bogged down in any low-level system or networking commands. Also, the multi-thread safe nature of TeamQuagent allows for robust, real-time applications to be constructed without extra checks.

4.3 The AdvancedTeamQuagent package

The AdvancedTeamQuagent package is built upon TeamQuagent and allows for the easy construction of a layered control architecture, such as those used in [Gat, 1992][Hayes-Roth, 1995][Hayes-Roth *et al.*, 1995]. Having layered control allows for the easy design of encapsulated and error tolerant agents. The bots in the URQuake

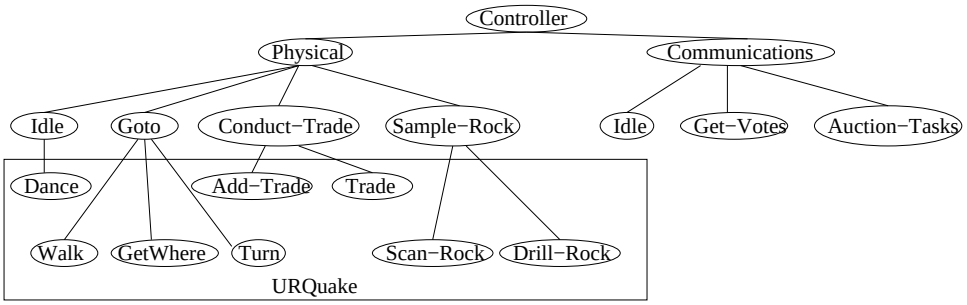

Figure 4.4: The AdvancedTeamQuagent layered architecture. Many actions are encapsulated inside of more general ones.

simulator operate in much the same manner as a layer of physical control for a rover, and so can be understood and approached in the same way. The quagent controller sends abstract commands to the game engine, such as "DO WALKBY 50." The engine keeps working on whatever the controller last told it to. In the WALKBY case, the controller sends a message, the engine handles all the details over the time of execution, and when it is finished, it notifies the controller of success or failure.

With this is mind, I used AdvancedTeamQuagent for the agents in DAVS (Figure 4.4). By using this architecture, the controller can issue abstract commands for both the URQuake bot and the communication system, and not have to worry about exactly how they will be accomplished or how long they will take.

4.4 Minor protocol changes

Apart from changes to allow uniqueness in quagents and objects, a few extra functions were added and modified:

- Added (For use with standard or team quagents)

    1. DO RUNBY - RUNBY works just like WALKBY, except that the quagent will move faster and use different animation frames.

2. ASK PING - PING acts in a similar way to RAYS, except that it sends out a dense wedge around a heading, rather than evenly spaced. In addition, unlike RAYS, PING will only respond if it hits a wall, passing through any other quagent or item.
3. DO DRILL - DRILL requires the unique item Rock. Each Rock has a certain amount of two types of minerals. DRILL will remove a unit of one of the types, and create a new mineral item in the quagent's inventory.
4. DO SETEXEMPT (0 or 1) - SETEXEMPT will make the quagent exempt from kryptonite, dying of old age, running out of energy, limited number of rays, and limited distance for RADIUS.

- Modified (Changed for use by team quagents)

1. DO SETID # - The # in SETID should be the same as the one that the quagent will be using for communication. When SETID is used, the quagent will receive modified information from RADIUS and GETINVENTORY.
2. ASK RADIUS - After DO SETID is sent, ask radius will respond with a list of objects, of type Quagent, Standard Item, and Unique Item. All have a name and position, but Quagents also have a bot type and ID, while Unique items have a list of properties.
3. DO GETINVENTORY - In addition to the standard response, after DO SETID is sent, GETINVENTORY will send a TELL with a list with the names and IDs of all the inventory objects.
4. DO ADDTRADE [with who] [Agent1 item] [Agent2 item] [Agent1 money][Agent2 money][Agent1 info][Agent2 info] - Will add an acceptable trade to the quagent's list.
5. DO GETTRADES - Returns a list of all of the pending trades that the quagent will allow.

6. DO REMOVETRADE [trade ID] - Removes a trade from the list of pending trades, so it will no longer be allowed.
7. DO TRADE [with who] [Agent1 item] [Agent2 item] [Agent1 money] [Agent2 money] [Agent1 info] [Agent2 info] - If the other quagent is in range and has agreed to the trade, then it will be completed.

5 Tolerance to communication error

Software agents are useful tools for encapsulating autonomous functionality, as discussed in [Genesereth and Ketchpel, 1994][Gray, 1995]. Their function is just as the name suggests. They are autonomous agents that are implemented in software, resident on hardware hosts. Multiple software agents can be present at once in a single host, and it is possible to have a large group of agents and a small group of hosts. If there are more agents than hosts, then it becomes necessary to decide which get the use of the hosts' resources. I suggest that for a host to decide, it should conduct an auction where each resident agent submits a bid of its estimated utility and the host selects the highest bid.

By using large groups of software agents and small groups of hosts, I have been able to design a robust and dynamic teamwork system that is tolerant to communication failure. Any implementation with multiple software agents can take advantage of the intrinsic failure tolerance such a design brings. I suggest that each software agent be built to have a single recipe to complete a single goal, and it is up to the host to decide which recipe (and therefore agent) should get to execute. With this method, the designer only has to decide which single-goal agents are needed for the system to succeed and a way to calculate their utility. Human interaction is simple too: people can interact with the system in a goal-based way, either adding or removing goals (in the form of agents,) and leaving the optimization and execution to the system.

5.1 Mobile agents

A mobile agent [Appleby and Steward, 1994] [Kotz and Gray, 1999] [Minar *et al.*, 1999] [Park, 2004] is simply a software agent that has the ability to move with its state intact to a different machine . Mobile agents are most useful when communication is either limited or uncertain. For instance, if the goal is to read and summarize news articles, it would be substantially faster for the code to move to the news server and execute there, rather then sending repeated requests over the network.

Any system where code is separated over long distances or the network is plagued with malfunctions can benefit from locally executing code. Mobile agents provide a generalized wrapper that can operate on various hardware platforms, without requiring the designer to explicitly design platform-specific code.

5.2 Auctioning within hosts and movement between hosts

I suggest the use of an auctioning system for the coordination of several agents within a single host. The host acts as the auctioneer, taking bids on the right to use its resources. The agents compute the expected utility of their recipe and report this as their bid.

For example, suppose there are two agents competing for the use of a law office's server, one wanting to email clients about new laws, and one wanting to tell everyone about a softball game. The server does not have to know anything more then the first reports a higher utility. This allows for easy addition and deletion of tasks during the life-cycle of the host, with zero modification to its programming. (This is similar to processes of claiming a higher priority in an OS.)

Each agent has the ability to move between hosts. Although the decision to move uses the same measure of utility as the auctions, an auction is not involved and it can travel to any host it wishes. Each will attempt to move to the host that will give it the

highest utility, using information on the structure and state of the host, as well as how busy the host is with others. In the previous example of the law office, even though the client e-mailer may get control of the first host, the softball agent can move to another, unused server to complete its goal.

5.3 Why multiple agents are useful for the control of a team

Suppose there is a large group of tasks and several robots to do them, as with a team in charge of rescuing people from a disaster area. The team would have the following goals:

- Search buildings A, B, and C
- Move rubble from Sectors 3 and 5
- When Sector 3 is clear of rubble, search it
- Transport injured people to an ambulance
- Put out a fire in Sector 1
- Direct rescue personnel to injured people
- Direct traffic away from the area

In this case, each robot would operate as an individual host. Each goal could be handled by a single agent that would automatically travel to the host that gives it the greatest ability to accomplish its goal quickly. For instance, the traffic director would want to be in a robot that is highly visible and can make signals for traffic to detour, while a rubble mover would want to go to a robot that has a shovel. If both a searching agent and an agent trying to transport injured people were vying for the same host, then

the injured people transporter would report higher utility and would be given precedence.

The process is dynamic and allows intuitive interfacing with human controllers. During the execution, if a new goal needs to be added, such as to “search building D,” then the agent (and therefore the agent’s goal) can be given to a single host, from which it will travel to its optimal location. The controller can approach the system in a goal-based way, without having to consider exactly how they will be accomplished.

The software agent based approach is automatically tolerant to communication disruption. If communication fails, then the agents simply stop hopping from one host to the next. There will be a loss in efficiency, but as long as no agent is permanently stuck on a host that is completely unable to complete its goal, then every goal will still be accomplished.

5.4 Design and simulation of a team

To test the practical application of my theory, I created a simulated world using the URQuake simulator [Brown *et al.*, 2004], with the core decision making and agent transfer components of my theory.

In the simulation, there is a large group of randomly positioned rocks that have to be drilled for samples. Each of the bots in the simulator represent a mechanized rover, and their corresponding quagent controllers work as hosts. (Note that in this context, a quagent controller is a host rather than a standard agent, however I use the term for consistency.) I tested team of one to five rovers, with a single mobile agent responsible for the drilling of each rock. Each of the hosts has a list of all of its resident agents and simulated hardware to communicate with other hosts.

5.5 Mobile agent design

In the simulation, the agents' recipes are hybrid automata that each can accomplish a single goal. Each automata directs an agent to move to a rock then drill it. The navigation portion of the automata is built with tolerance to imprecise motor control and with collision avoidance. (For more details on the design and function of the automata, see Section 5.6.) The automata are designed so that if a rock turns out to be non-drillable or unreachable, then that agent will remove itself from its host. In this way, even if an agent gets copied to multiple hosts and both try to drill the same rock, all but one will eventually admit failure, and will not be stuck trying to drill the rock forever.

The process by which an agent transfers from one host to another is designed so that even if one or more messages are lost, the agent will not be. The transfer begins with an interested agent sending a request to transfer to another host. If the host receives the message, it will add the agent to its list and confirm the transmission, so that the sending host knows to remove the agent from its list.

If there is a disruption and the sending host has sent a message that was received, yet the response is not received, then it will not remove the agent from its list, leading to two copies. I reason that it is better to have the possibility of copies, rather than risking the loss of an agent. Under ideal circumstances when communication can be assured, only one transmission is needed, as the confirmation is unnecessary.

Apart from the actual transfers between hosts, the agents also need to be able to share data, mostly related to their hosts' positions. While one agent has control of a host with several others resident, the rest are free to move around to other hosts. To ensure that they each move to the correct host, they need to be consistently updated about the hosts' locations. This is important, because in the drilling rocks scenario, the closer the host is to the rock, the quicker it can be drilled, and the higher its utility. Incorrect beliefs about rovers' states could cause an agent to make a poor choice in hopping between hosts. For example, an incorrect belief in a rover's position would

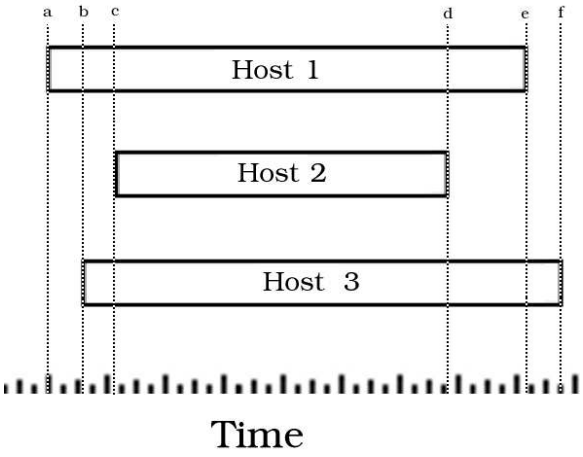

Figure 5.1: An example of estimations for the completion of recipes. Host 1 starts at time a, host 2 starts at time c, and host 3 starts at time b. When each of them starts, they give an estimate of their completion times.

require an agent to make extra hops in order to get to the correct host, or else operate sub-optimally. This would be an annoying problem, hence the frequent updates done to avoid it.

In the rock-drilling scenario, hosts are rarely idle, and so the agent currently in charge of a host has to give all others an estimate of where it will end up and how long it will take to get there. Take the example in Figure 5.1, where there are three agents, each controlling a single host. The current action that Host 1 is working on starts at “a” and ends at “e”, etc. It would not be particularly helpful to know where Host 1 was at time a, nor even where it is at the moment. Assuming that the current action is completed successfully, Host 1 will not be able to be controlled by a new agent before time e. To coordinate this, at time a, the agent controlling Host 1 will transmit its expected time of completion and state at e. During the time between a and e, interested agents can transfer themselves to Host 1, if they calculate high utility based on the estimation.

The communication protocol is similar to the contract net [Smith, 1980][Davis and Smith, 1983]. In the contract net protocol, an auctioneer first broadcasts a request to

interested agents, which then respond with their bids, and finally the auctioneer tells them which wins the bid. For n interested agents, there will be between $2n$ and $3n$ communications, and a delay while the auctioneer waits for bids to be received. On the other hand, my method requires only 2 communications per transfer, and assuming that the communications are successful, no additional waiting time. The downside is that each host has to update the others every time it completes a state change, more in the manner of a specific sharing system [Genesereth and Ketchpel, 1994]. As a result, the total amount of communications is not substantially less. However, messages are more spread out and there do not tend to be a lot of high peaks of activity, as seen in Figure 5.6. In addition, if hardware is available that can broadcast a message to all agents at a low cost, then the amount of messages will drop significantly, proportional to the number of agents. (Essentially, if an agent needs to talk with n others, it would have to send n messages in a point-to-point communication system, but only a single message with broadcast.) But even with just point to point communication, my method can utilize communication hardware efficiently.

The main difference is that once an agent is transferred to another host, the first host does not have to monitor its progress to make sure it is completed, reducing communication overhead. Also, because of the less complex decision process for task changes, where contract nets use three sets of communications, this system uses two.

One final optimization is to lower the costs of transference. The obvious way to do the transfer is to send all of the agent's code, which will not require extra messages, yet will increase the bandwidth needed. Instead, the hosts load the code for all possible agents on startup, each with a unique identifier. This way, only a single integer representing an agent's unique ID needs to be sent.

Figure 5.2: The three types of nodes used in the hybrid automata

5.6 Hybrid Automaton design

As stated before, each host has zero or more resident, single-goal agents, each vying for control of the host's resources. The host acts as an auctioneer, with all agents submitting bids of their estimated utility, from which the host picks the agent with the highest bid. Each agent is a hybrid automaton, and has the ability to either perform actions, or use the current world state to estimate its utility. The specific construction of the hybrid automata is as follows:

5.6.1 Type of nodes

There are three different types of nodes: decision nodes, action nodes, and uncertainty nodes (Figure 5.2). The current state of the agent moves along directed paths between nodes.

An action node is the part that actually uses the host's resources to progress toward the agent's goal. An action node can represent an arbitrarily long and complex sequence of actions, with the time required to do the actions and the host's estimated final state represented by a heuristic. For optimality of the overall system, the heuristic should be admissible, or else extra communication will be needed for each agent to be correctly positioned.

As an example, suppose there is an action node "go to" that directs a rover to travel

from point A to point B. A standard heuristic would be to estimate the time it would take to travel a straight-line path, while a more accurate one might have knowledge of obstacles and add time for each that has to be avoided. If there is time to calculate it, the heuristic could even call an advanced path planner to calculate the exact path and travel time, given no unexpected events such as crossing paths with another rover occur.

An action node has two possible results, either the action succeeds, or else it fails. If possible failure of an action is not handled, likely by adding a decision node on the failure path, then the agent will release control of the host and cease functioning. As a result, even if everything does not go to plan, as many goals as possible will be completed.

A decision node is a wrapper around an arbitrary piece of code that takes a set number of inputs, and chooses the next state. A decision node could be an implementation of any type of decision-making process, from a POMDP to a neural network. The only restriction on the code is that for the same world state, the same decision must be made.

Uncertainty nodes are used solely to estimate the utility of a branch of the automaton. Each uncertainty node has several branches, with a calculated probability attached to each branch. By the time the agent's state gets to the uncertainty node, one branch must be 100%, or the process will fail. (For instance, a coin flip would have a 50-50 chance before the event, but after the flip, the result must be exclusively heads or exclusively tails.) Each of these types will be explained in several examples.

5.6.2 Using nodes to make an agent

Suppose that a robot is being used to search for injured people in a building. An automaton such as in Figure 5.3 could be used. (Before any people are found, the "Can they be rescued?" will assume that they can be, unless other data is given.) As a result, the estimated utility will be (.7*50)/2 +(.3*0)/1 = 17.5. There is a 70% chance that the actions succeed, and it will take two units of time, plus a 30% chance there will be no

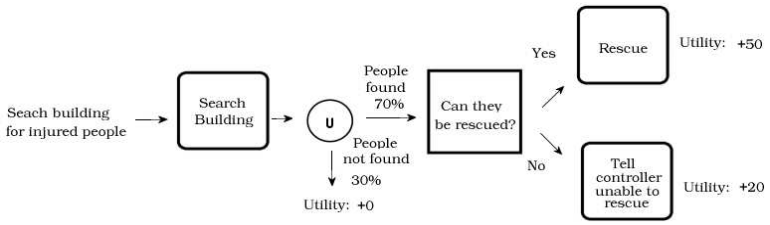

Figure 5.3: A hybrid automata that is a recipe for automated search and rescue

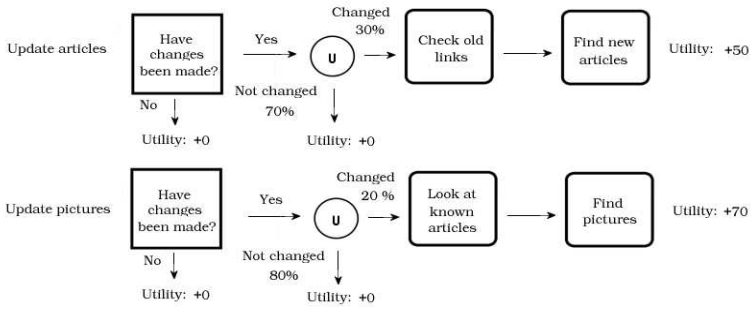

Figure 5.4: A hybrid automata that summarizes online news

people, which will take one unit of time.

The real benefit of my system is when there are multiple goals. In a web page such as FootyMania [Beri, 2004], a program searches the Internet to look for new news articles, and search those articles for pictures. It would be useful to have a process running constantly that only uses resources when there are probably updates to the news. In Figure 5.4, two agents are competing for the use of network resources. The agents assume that articles should be updated more frequently than pictures need to be. The “Have changes been made?” decision node could use any kind of process to decide if the articles have been changed, from a simple “time since last update”, to a database with statistics of the update times of other web-pages. Right after one of the agents has run, it will drop its probability that there are more changes, so it is likely that either the other agent will run, or they will both just wait.

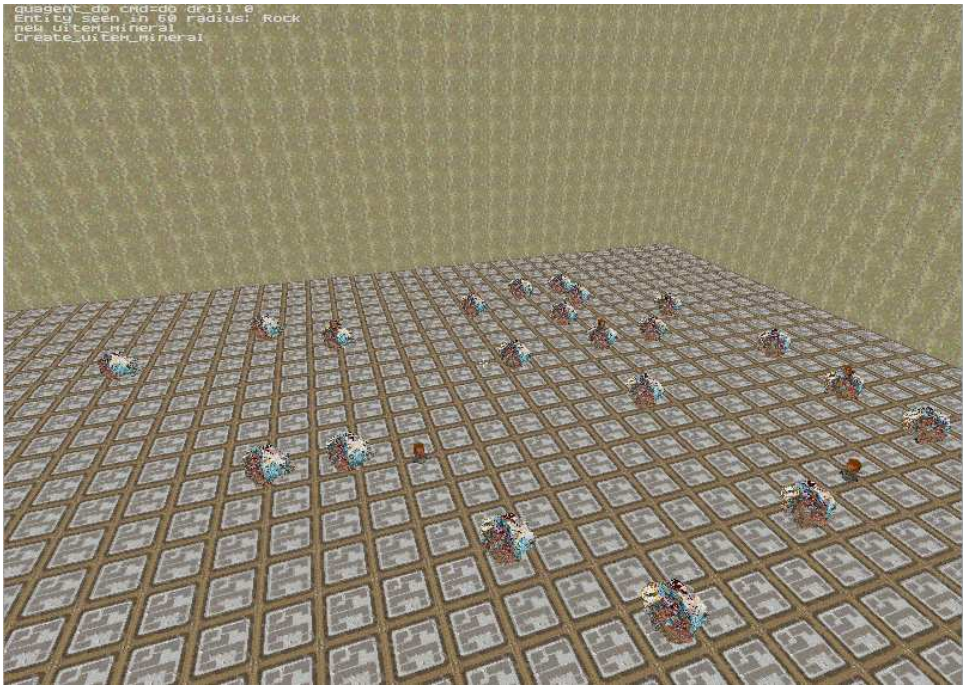

Figure 5.5: Agents in action

5.7 Experiments and Analysis

Given that the movement of the agents from one host to another is greedy, it is hard to say that performance is any degree of optimal, however, as stated by [Davis and Smith, 1983], first-choice strategies such as auctions tend to work reasonably well, even though they do not calculate the true optimal.

To test the implementation, I ran five trials each with one to five quagents drilling fifty rocks, with 0%, 50%, and 100% communication failure (for a total of 75 runs.) I varied the number of agents in the team, tested different levels of communication uncertainty, and used several randomly generated groups of rocks, with results in Figure 5.7. In general, the more rovers that are added, the faster the rocks are drilled, which is as expected. Also, as more uninterrupted communication is allowed, the solution becomes somewhat faster, but even if there is partial or total communication failure, the task is still completed at reduced efficiency.

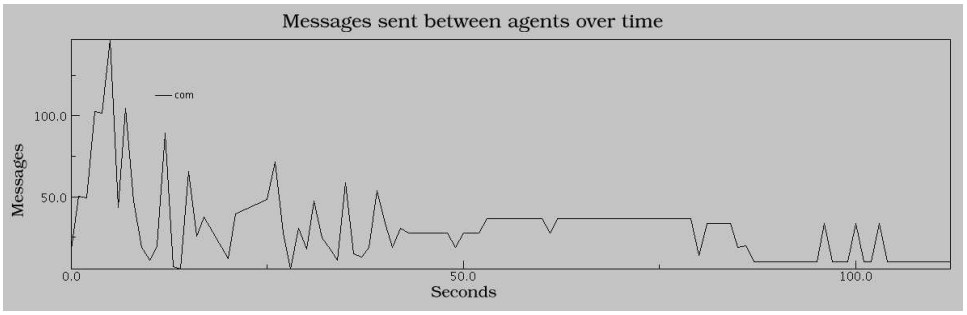

Figure 5.6: A typical graph of the number of messages sent over time

Although it is not evident from the graph, on one of the configurations of rocks, the agents who had no inter-host communication actually performed the fastest. This demonstrates that although the greedy agent transfers tend to work efficiently, they can potentially be less efficient in certain cases. But even given the losses in efficiency, the biggest advantage of low communication costs remains. Even as the number of agents increases, the number of communications varies mostly with the number of tasks to be completed. Even if the worst case where there are far more rovers than tasks, communication will not increase. Each agent will be required to spend more time calculating which host to travel to, but will still only have to make a single transmission.

It may seem odd that even though the drilling is completed faster as more agents are added, the gains seem to decrease exponentially. This is largely due to the formulation of the scenario. Figure 5.8 shows the main cause. It takes almost no time for a quagent to drill a rock, and so when there is a large group of densely clustered rocks, the total time for one agent to travel to them all is not necessarily significantly slower than having a group of agents, with one visiting each rock. If the time needed to drill a rock were increased, then the benefits of having more agents would increase.

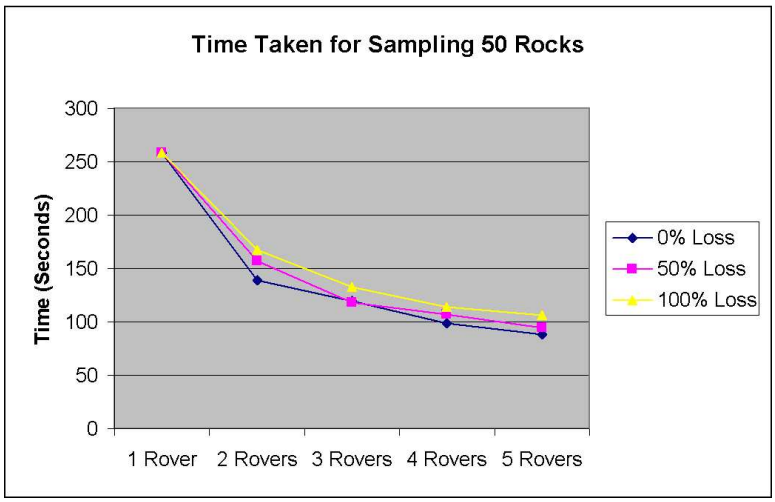

Figure 5.7: Graph showing the time taken to drill all of the rocks

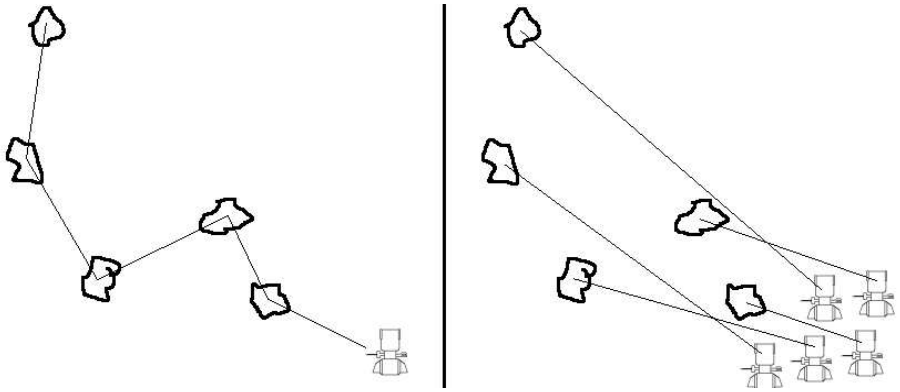

Figure 5.8: An explanation of the exponentially decreasing speedups from adding extra agents

5.8 Summary and future work on communication-error tolerant coordination

For communication-error tolerant coordination, I suggested an approach to coordinate multiple software agents toward the completion of high-level tasks in both software and hardware environments. By using intra-host auctioning accompanied by inter-host movement, a robust and dynamic system arose. Using software agents allows the decision making to be automatically distributed to a great degree, and eliminates the need to rely on any specific controller.

Some papers, such as [Huber and Durfee, 1995][Balch and Arkin, 1997] suggest that performance can be guaranteed for robots by having them observe their environment and see what their teammates are doing instead of explicitly communicating. More complex behaviors in communication-poor environments, or even actual agent transfer, could take place by having robots give either explicit physical cues, or else creating agents that can infer other's though processes from observation.

To further reduce the need to rely on communication, elements from swarm robotics and stigmergic theory could be added [Reynolds, 1987][Steels, 1990][Brooks, 1989]. Even though the agents themselves move from host to host, it may be possible to make the interactions simpler, so as to allow for greater functionality, even in uncertain environments. Another similar way would be to start from the ground up, with a simple insect-like robot, and then to build functionality on top of it to create a complex organism/agent. This way, rather then trying to squeeze standard software agents into a stigmergic framework, the system could be make with high-level communication built upon emergent behavior, which could theoretically make for a very robust system.

Any of these methods could increase performance and usability. There are as many ways to improve the design as there are to design it in the first place, but most likely those that warrant the most study are with emergent behavior. It would allow for agents that can perform in uncertain and hostile situations without severe penalties.

6 Tolerance to sensory error

Complex, coordinated behavior under uncertain conditions is a challenging task. When there are several agents all trying to estimate the true state of the world, it is important that one with a particularly bad estimate does not mess the others up too. (This is a similar issue as is addressed by robust statistics [Huber, 1981] [Nasraoui, 2005].) An obvious technique is to just average all of the estimations, but averaging tends to miss a lot of detail and agents' true desires, even if it is not skewed by relatively extreme values. Instead, I use a combination of voting and auctioning to coordinate the agents, integrated into DAVS.

6.1 Error tolerant coordination

In DAVS, when an agent decides to work together with others, there are two coordination steps that must be completed. First, the agents have to agree on which goals are the most valuable and worthy of completion. Second, they have to figure out which agent is best capable of working on each task.

At first glance, it may seem odd that this is done in two steps; why not just have a combinatorial auction and be done with it? There are fast ways of calculating winners in a combinatorial auction [Sandholm, 2002] [Fujishima *et al.*, 1999], and they can be

effective even given a large group of agents and goals. However, there is not a good way to decide on a correct allocation when agents make incorrect bids. Even if the bidding is inaccurate, the voting step combines the opinions of all agents and gives an error-tolerant result, yet preserves agents' main desires.

6.2 The voting step

Voting methods allow for fast, robust coordination among multiple agents. One method of voting is the Borda Count [de Borda, 1781], in which each voter ranks all alternatives against the others. For example, for four options (A,B,C,D), a voter could give the ballot (3,2,1,2), meaning that alternative C is the best, followed by B and D, and last, alternative A. The Borda Count is calculated by figuring out how a voter values an alternative compared to the others, and then summing the preferences of all voters.

The initial step, which is computed independently for each voter, counts how many times each alternative is ranked higher or equal to the others. A win receives two points, a tie receives one point, and a loss no points. The count for the previous ballot would be (0,3,6,3). For each voter, the counts are summed and the winners are determined. For instance, if a second voter's votes were (1,2,3,4) and count was (6,4,2,0), then the total for both would be (6,7,8,3), and the winners would be $O3 < O2 < O1 < O4$, with $a < b$ meaning a defeats b.

I prefer the Borda Count method of voting, as it tends to select candidates that are if not perfect, then at least tolerable to most of the voters. Also, as explained by Saari, it has few of the paradoxes of other systems [Saari, 1994]. One of the largest criticisms of Borda is that it is possible to change the winner simply by running additional candidates. Even though it is not strictly fair, I feel it is still reasonable, especially given that no election system can be completely fair [Nanson, 1882]. Preferred candidates are still ranked higher, and artificial agents would not have to be made to be worried about fairness of the system.

```
calculateBorda(int votes[][], int numAgents, int numAlternatives)
  int count[numAgents][numAlternatives] := 0
  int total[numAlternatives] := 0

for A in numAgents
  for B in numAlternatives
    for C in numAlternatives
      if B = C
        continue;
      if votes[A][B] < votes[A][C]
        count[A][B] += 2
      else if votes[A][B] = votes[A][C]
        count[A][B] += 1

for A in numAgents
  for B in numAlternatives
    total[B] += count[A][B]
```

Figure 6.1: Pseudo-code for the trivial method of calculating the Borda count

6.2.1 The trivial implementation

For n agents and m alternatives, Borda Count lends itself to a straightforward implementation, in $O(m^2 * n)$ time for m alternative and n agents. If done in parallel, then it is reduced to $O(m^2 + m * n)$. Pseudo-code in show in Figure 6.1.

6.2.2 Two ways of distributing the work

I worked out two different ways to distribute the work. Section 6.2.3 describes a method in which each agent computes part of its solution before sending it to the others, which then verify that the counts given are possible. Section 6.2.4 is simply a fast compilation. It is more intuitive and does not require honesty, but it will not

catch any transmission errors. I have completely solved and proved the first method, but although the second appears valid, it still remains to be fully explored.

6.2.3 Verification

If communication is accurate and all agents trust each other to be honest and correctly compute their counts, then the computation can be done as shown in Section 6.2.1. However, suppose that there are communication errors or not all agents are trustworthy. A dishonest agent could fix the election by sending an impossible count, such as (500,0,0,0), in order to cause alternative 1 to win. To combat this, every agent should have a way to verify that the calculations were done correctly, without having to completely redo all of the work. It turns out that this verification can be done in linear time by exploiting the fact that only a small number of different counts are possible, and they all follow a common pattern.

Each agent is responsible for determining the count for its own votes, then sorting the results. The sorted results are sent to all other agents in the group, where every individual agent is responsible for verifying and summing all of the counts that it receives. Although a single agent could determine the winners, it would take no less time and require an extra communication, as well as being less tolerant to agent failure. Instead, each agent completes its own count, sorts it, and sends it off, then verifies and tallies the counts it receives from the others.

This verification is possible, as there are only a small number of possible counts, and they follow a simple pattern. The following theorems and corollaries show exactly how this is done.

Theorem 1 *The sum of the count for m alternatives will be* $n(n-1)$.

Proof: *There is a relationship between every pair of voters (Figure 6.2.) Either one dominates the other, or else they tie. If one dominates, then it receives two points while the other receives zero, and if they tie, then both receive one point. In either*

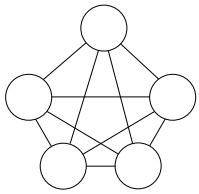

Figure 6.2: A connected graph representing voters

case, there are two points total for each relationship. Since a fully connected graph has $(n(n-1))/2$ *edges, and each edge is worth* 2*, then the sum of all relationships is* $n(n-1)$.

The theorem above is helpful because if the sum of a count is not equal to $n(n-1)$, then it is incorrect. Unfortunately, this says nothing about correctness for counts with that sum. The possible correct counts for one through six alternatives are show in Figure 6.3, ignoring permutations. The remainder of this section assumes that the counts are sorted from highest to lowest.

Corollary 1 *The* m *highest ranking alternatives are each equal to* $2n-m-1$*, where each alternative in the* m *dominates all others not in* m*.*

Proof: *As stated by Theorem 1, the sum of all alternatives is* $n(n-1)$*. By the same measure, the sum of the* $n-m$ *smallest alternatives is* $(n-m)(n-m-1)$*. Therefore, the sum of the* m *largest alternatives must be* $n(n-1)-(n-m)(n-m-1) = m(2n-m-1)$*. For m alternatives, this means that each should be equal to* $2n-m-1$.

Theorem 2 *If the* m *highest ranking alternatives, where each alternative in the m dominates all others not in the* m*, are correct as of Corollary 1, then the remaining* $n-m$ *alternatives can be verified recursively by the same method.*

Proof:

1. *Start with the set* m *of alternatives that are higher than all others. These can be verified by Corollary 1. They collectively win against all the others.*

2. *The remaining $n - m$ alternatives can only lost against the greatest m. Since losses give no points, all of their points must be from those they win and tie against. This is equivalent to having another count of size $n - m$.*

With the help of Theorem 2, an algorithm can be written to verify a count.

Algorithm:

1. If there is only one element, it must be zero, as it does not tie or win against any other alternatives.

2. There are m alternatives that are greater than all others, with $0 < m \leq n$. Using Corollary 1, they can be verified.

3. Recursively check the remaining, smaller $n-m$ numbers. If all recursive subsets are verified, then the count is valid.

6.2.4 Calculating Borda efficiently for each agent

It appears that there is a way to calculate the count in only $O(mlogm)$ per agent, which in the end, takes about the same amount of time as the more convoluted way in the previous section. Although it looks promising, I have not proved its correctness.

Essentially, each agent will sort its alternatives first in $O(mlogm)$, then transmit. I think that with a sorted set of alternatives, each count can be calculated in $O(m)$, for each of the n agents. Both ways should take about the same amount of time, although this way is simpler, but has no built in tolerance to communication errors.

6.3 The auctioning step

Once the voting step is completed, the team has a small set of recipes on which they implicitly agree on which to work. They then use an auction to decide the exact

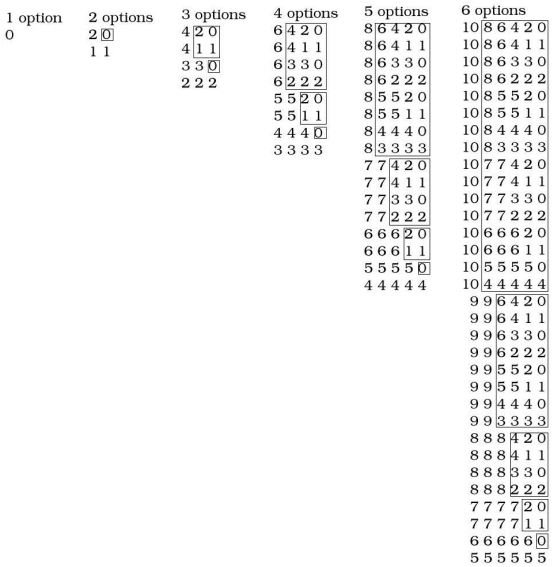

Figure 6.3: The possible counts for between one and six alternatives (ignoring permutations)

recipes and tasks they will work on within the recipes. There is still a possibility that this decision process will yield poor results due to agents' errors in their beliefs and therefore their bidding, but this is unavoidable and should be minimal in most cases. In DAVS, auctioning is a simple process where each agent wants a single task in a recipe. For each task that needs to be done, it just find the highest bidder that is not otherwise occupied. This is the way that many instantiated auctioning systems work, although it is clearly not always optimal, it is fast.

6.4 The exact algorithm

The recipes in DAVS are pretty vanilla. They have a set of tuples each for preconditions, constraints, and effects. Preconditions include things such as if a rock has already been drilled, or if there are enough unoccupied agents of a certain type. The actual actions performed in the recipes are written inside a class, in the implementation's native language (Java). Once one or more agents have agreed to work on a recipe, Java code

and the AdvancedTeamQuagent package are used to control each of them. This lets the system work at a high level on voting and auctioning and ignore most of the exact workings of a recipe, yet allows for actions with an arbitrary level of complexity.

In the coordination process, there is some repetition in calculation, which is done to minimize the amount of communication needed. Apart from any other associated costs, latency in communication can cause delay, and it can actually be quicker for each agent to work out some things on its own. Each agent computes the winning recipes when voting and the winning bids when auctioning. In this way, two less rounds of communications are required than if a single agent or group was used instead. (Specifically, there are three rounds versus five rounds, as once an agent has figured out which team it is on, it verifies intention with its teammates, which insures robustness to agent failure.)

The steps a single agent goes through, from the beginning of the coordination process to the execution of recipes are as follows:

1. Calculate utilities and votes
2. Compute its count and send the result to the others
3. When all counts have been received, verify them and compute the winning recipes
4. Bid on recipes, using the previously calculated utilities as bids
5. When all bids have been received, compute which recipe and task on which it should work
6. Verify intention to work on the recipe with its team

To begin, each agent calculates the expected utility of each recipe. Any number of methods could be used at this stage, but for simplicity, agents have either an exact preference for each tuple (such as true or false), or else monotonically prefer increasingly large (or small) amounts. Once the preferences have been decided upon, it is simple to

use them to rank all alternatives. Once the agent has decided on its votes, it partially computes the solution, as discussed in Section 6.2.3, then sends it to the others.

After it has received the counts from everyone, it verifies that they are correct, compiles them, and decides on the winners based on preferences and the satisfaction of preconditions. It then calculates which it will bid on, and sends out its bids. When it has received all of the others' bids, it calculates which job it will work on and the team with which it will work. Finally, it sends a message to all others on its team to verify that they are working together. When it has received similar messages from all its teammates, it begins execution of the recipe.

6.5 Simulation

To test the teamwork theory, I created a scenario similar to that in 5.7 with a team of autonomous rovers on Mars, which are responsible for sampling a group of rocks for evidence of life. In addition to the previous scenario of testing tolerance to communication failure, this one also requires several heterogeneous agents to work together to drill rocks. There are four types of rovers, Surveyors, Drillers, Trucks, and Landers, as seen in Figure 6.4.

1. **Surveyor** A surveyor is a quick rover capable of doing a fast surface analysis of a rock (such as with a spectrograph) to determine if it may be of interest.

2. **Driller** A driller takes a core sample of a rock. Drillers can interface with Trucks and other Drillers for extra functionality.

3. **Truck** Carries core samples (like a dump truck).

4. **Lander** Has a full range of equipment for testing a core sample, but is stationary.

The scenario is a basically a variant of the Vehicle Routing Problem (VRP), with additional constraints about when a path can be taken and requiring some places to be

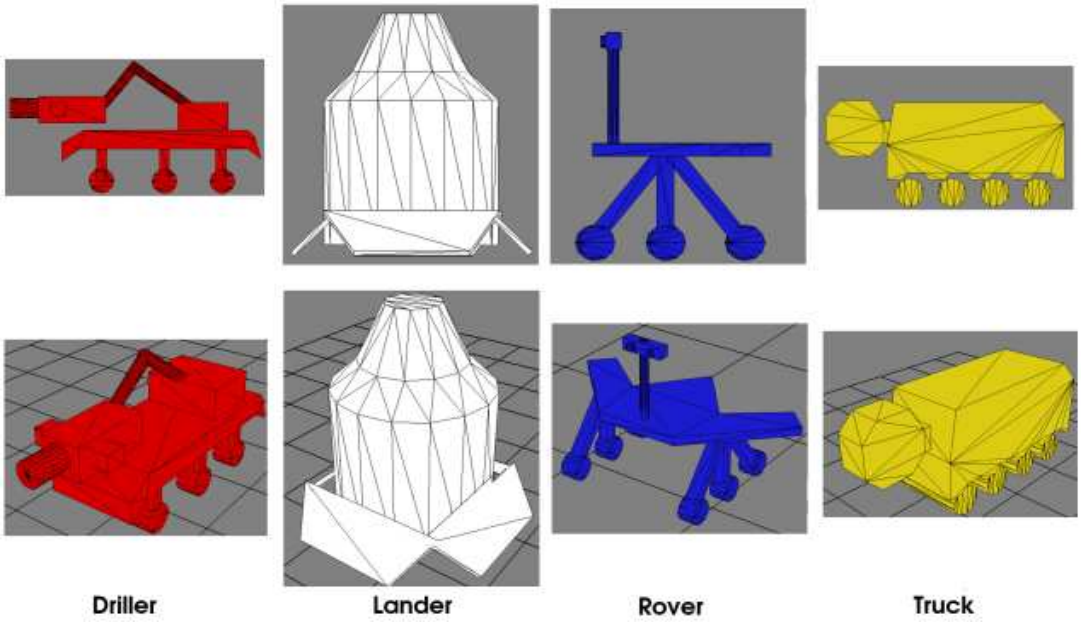

Figure 6.4: The four types of rovers used in the simulation

visited more than once. The rovers are responsible for taking a core sample of any rock that may have evidence of life. In brief, the surveyor will look at a rock to see if it is interesting at all. If there is some chance that the rock is worth something, a driller will quickly examine it to figure out which part is the best to drill. When the quick sampling has been completed, a small team goes to the rock and takes a large sample, which the truck then transports back to the lander. In detail, the types of recipes are:

1. **Survey** A single surveyor can do a spectrograph scan of a rock, to determine its possible worth. This recipe is not completely necessary, but it makes the work of the others more efficient by eliminating rocks that have a small probability of being interesting. *Requires:* (1)Surveyor.

2. **Sample Rock** A single Driller can take a few quick samples of a rock. This gives a more accurate picture of the possible composition of the rock, and will allow for a more detailed sampling. *Requires:* (1)Driller.

3. **Drill Rock** Once the rock has been sampled, a pair of drillers can take a larger

sample and put it on a dump truck. *Requires:* (2)Drillers, (1)Truck.

4. **Analyze Sample** If a dump truck has a sample in it from a Drill Rock recipe, it can bring it to the Lander, which will conduct a detailed analysis. *Requires:* (1)Lander, (1)Truck.

6.6 Analysis

The main group of experiments test how well the system performs under inaccurate positioning of individual rovers. Finding the absolute position of a physically instantiated robots is a daunting task, and sometimes even impossible. A multi-agent coordination technique must be tolerant to such errors. To test this tolerance, three solutions are presented and compared:

1. **The Optimal** The optimal solution is the shortest paths between rocks that have minerals. Since actual agents cannot immediately tell if a rock may be interesting, it is not likely that any real solution will be optimal, although it is a useful metric for measuring different systems.

2. **Averaging** Averaging is presented as the trivial method of coordinating a group. When sensor data is uncertain, rather than voting and auctioning, a single leader agent compiles and averages all of the guesses and assigns each other agent to a task. Although clearly not optimal or tolerant to communication failures, this averaging is a real solution, as it does not assume omniscience, and so is more easily compared to others.

3. **Voting and Auctioning** Voting and auctioning is as presented in DAVS, with a voting stage for general decisions and an auctioning stage for specific ones.

To understand the problem with uncertain sensory data, look at the left diagram in Figure 6.5. All of the agents have to be together to sample each rock. The fastest path

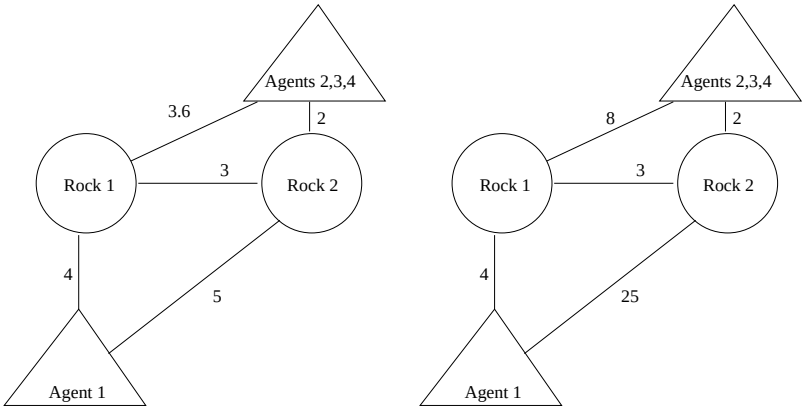

Figure 6.5: An example of correct versus incorrect distance measurements

is for all to go to Rock 1 first, then all go to Rock 2. If all the agents are completely accurate, then any one of them can be asked to calculate the optimal path. However, if the agents are inaccurate and when asked, square the distance to the further of the two options, as in the right diagram in Figure 6.5, then the answer changes, and apparently the fastest path is the other way! Unlike simple calculation, voting will correct for this. Even if the agents' error increases exponentially, then voting will often still give the correct answer.

6.6.1 The GUI

In order to be able to better show the teamwork process in action, I created an additional GUI interface that displays both global and agent-specific information 6.6. The specific numbers refer to:

1. When the rovers have completed all tasks, clicking the done button will display statistics on the run and show all the message traffic.

2. The Messages per Second window shows how many messages were sent over the last minute. When the done button is clicked, it will display all messages from time zero.

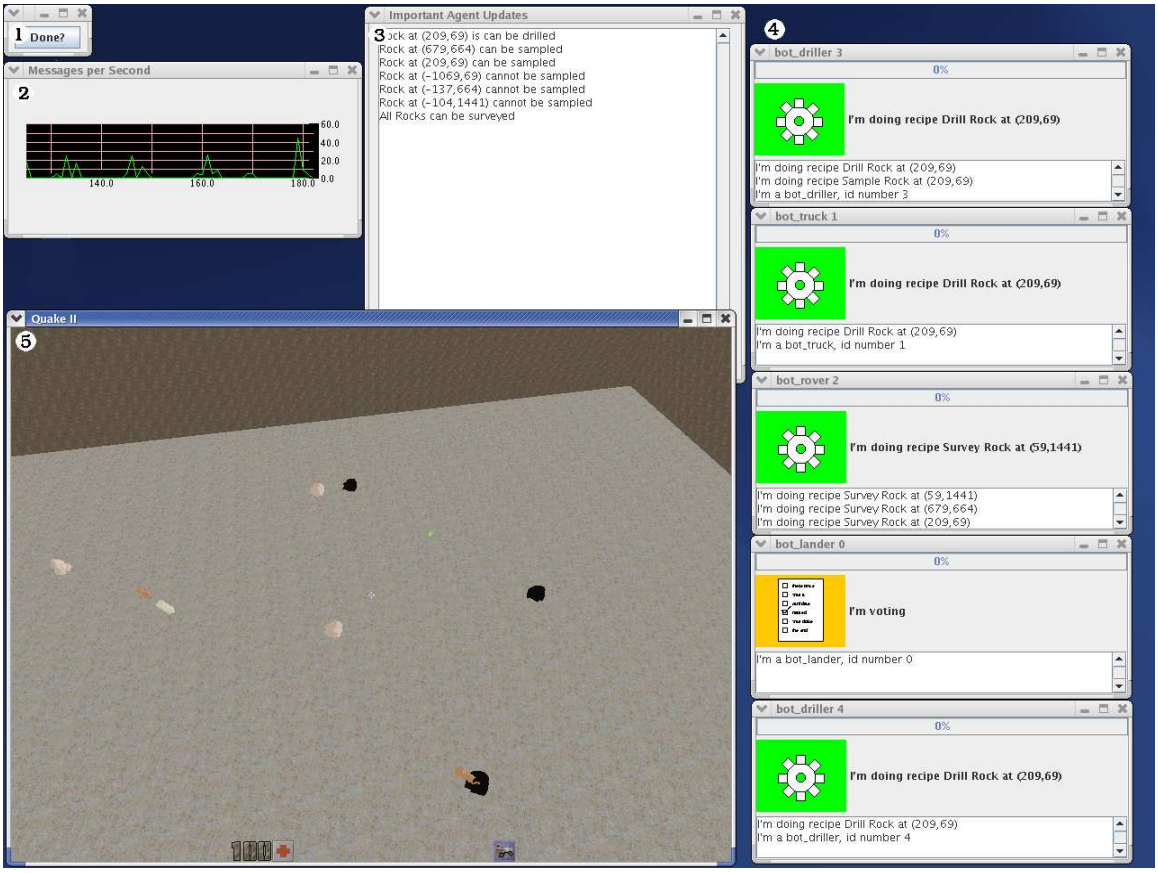

Figure 6.6: The quagent GUI

3. As the rovers discover things about the rocks, they will update each other's world state. The Important Agent Updates window is a summary of all the essential communications for all agents.

4. Each of the windows here is a glimpse into the mind of a single agent. It displays the agent's name and id number, its progress (on voting and auctioning tasks,) its current task, and a history of previous recipes that is has worked on.

5. This is the main 3D rendered window, which shows the rovers moving around in real time.

6.6.2 Evaluation

The measure used is the total distance traveled by all agents. Results are shown in Figures 6.8 and 6.7.

Figure 6.7 shows the results from DAVS and from the simple averaging scheme. The upper-left diagram shows how DAVS performs under different degrees of error and number of rocks. (The error numbers refer to how far off an estimate can be, for instance .5 means it can be from 50% to 150% of the correct value, randomly calculated each time.) At first glance, the algorithm appears to be remarkably efficient. However, I discovered that due to the formulation of the problem, the distance traveled by the truck is almost constant and also dominates the result, so the unprocessed results are deceiving. The upper right shows the distance traveled by all agents, except for the truck. Most of the variance occurs from changing the number of rocks, although as error increases, efficiency goes down slightly. The lower-left and right show the results from the averaging technique.

Figure 6.8 shows results, averaged for all numbers of rocks. DAVS tends to be slightly more efficient than simple averaging, although not close to optimal. The main reason that the results are far from optimal in the data is that the agents use a simple, greedy search to decide on which rocks they should work. They could use any number of techniques to get greater efficiency. Even given the current results, they show that the voting and auctioning system is fast and reasonably effective, even under uncertainty.

Typical message traffic if shown in Figures 6.9, 6.10, 6.11, and 6.12. Figure 6.9 shows messages over a thousand second run. The large spikes are auctions and verifications of intentions, where the agents are sending messages all at the same time. The small spikes are votes, which are less in number and more spaced out than the auctions. When all possible recipes have been completed, the agents search for tasks, rapidly vote and abandon voting, resulting in the traffic at the end. Figure 6.10 shows a minute during execution. In most cases, the initial, small spike is voting, the largest, middle

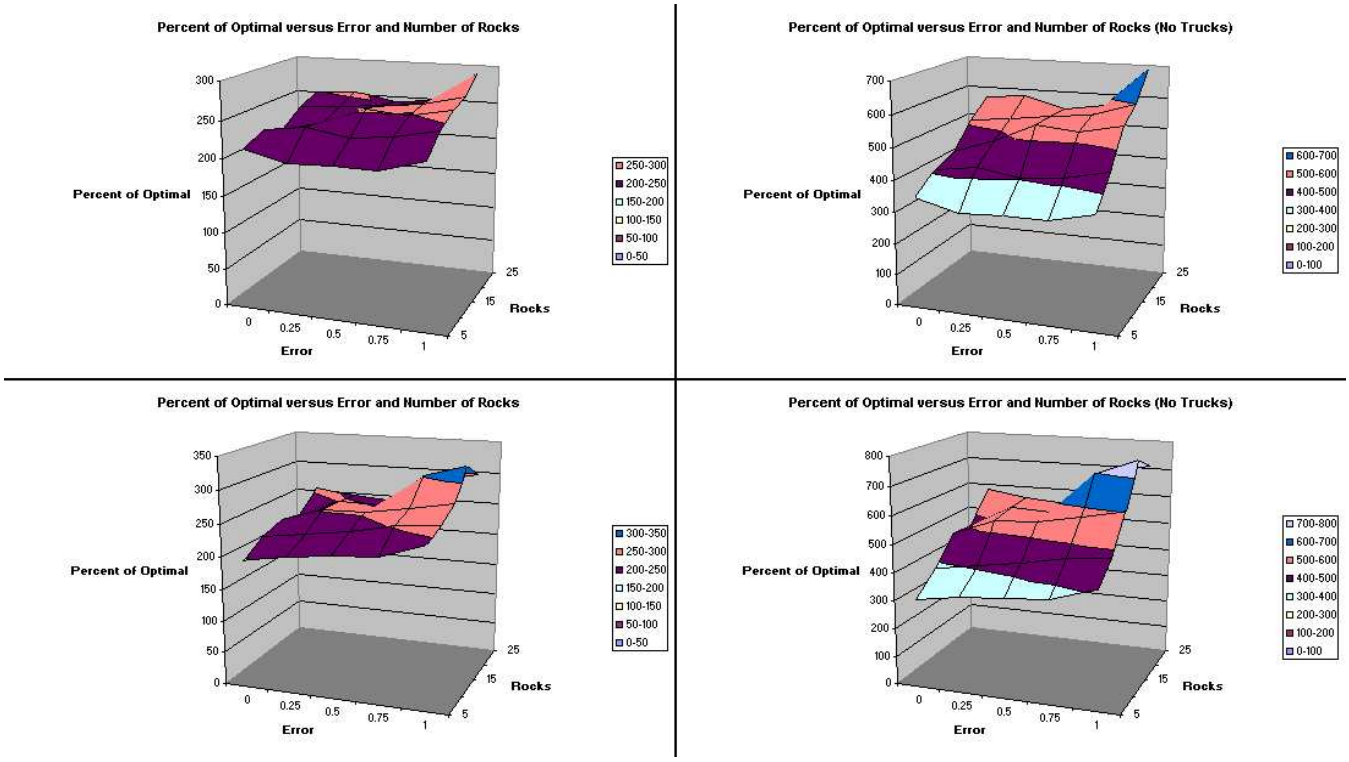

Figure 6.7: Total distance traveled with different coordination techniques. Clockwise from upper-left: DAVS including trucks, DAVS without trucks, averaging without trucks, and averaging with trucks

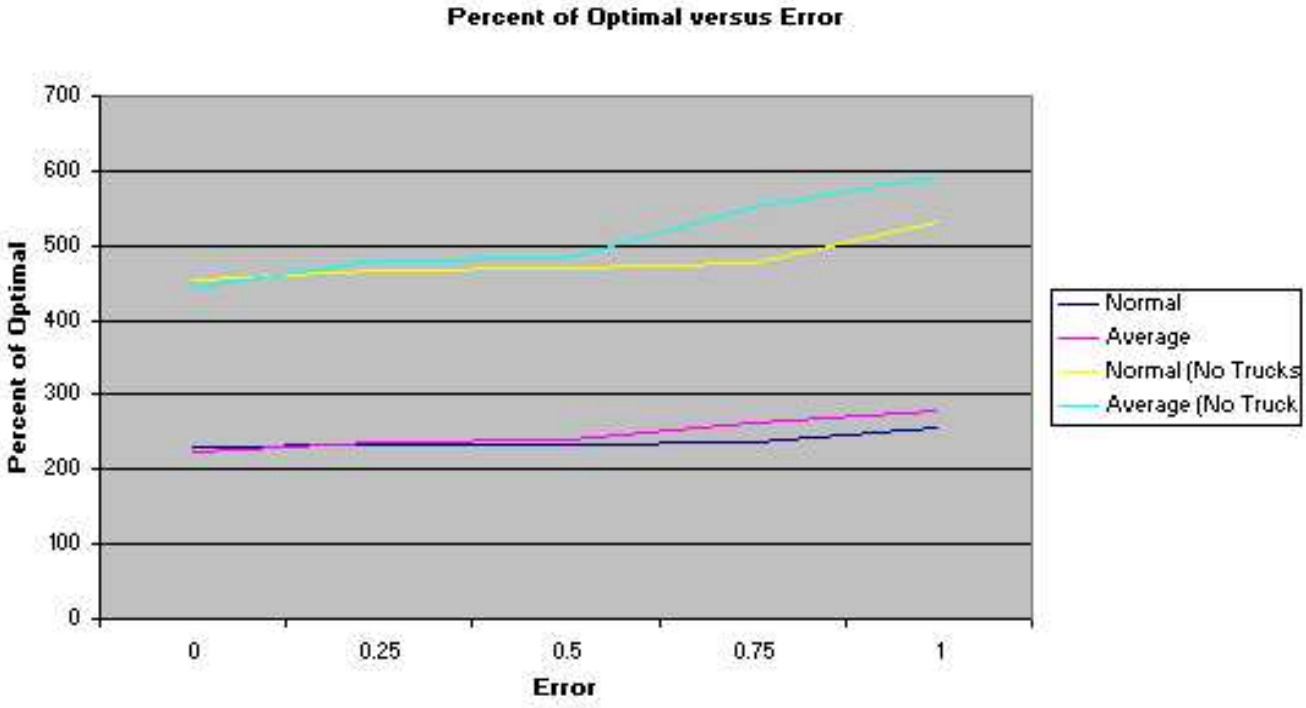

Figure 6.8: Total distance traveled with different coordination techniques (averaged over all numbers of rocks)

spike is auctioning, and the final, medium spike is verification of intentions. Sometimes some agents will finish early and vote, so the same pattern does not always occur. Figure 6.11 is overall messages for the averaging technique. There are fewer messages than in the first two, as it is really just a single agent telling the other what to do. When there are no tasks left, it sends no messages. The final figure, Figure 6.12 is a minute of messages for averaging. Each spike is when the leader agent tells the others what to work on. The more rovers that are assigned a task, the higher the spike. Figure 6.12 is a minute slice for averaging.

6.7 Summary and future work on sensory-error tolerant coordination

For sensory-error tolerant coordination, I suggested a way to coordinate a group of agents with a voting step and an auctioning step. The system automatically weighs different agents beliefs and desires to find a good compromise that tended to be realistic and lead to effective work.

The main improvement that could be done is to improve agents' estimation of utility. POMDPs look promising, as they have the ability to handle uncertainty and can be used to look several steps into the future. For complex environments, using a full planner such as [Blum and Langford, 1999] [Majercik and Littman, 1998] could work well. Truly, any utility-calculation technique could be used, and certain techniques could work better for certain problems. Still, a general, robust, probabilistic utility-calculation method would be a good step forward.

Another improvement to the implementation would be to make agents be able to form teams asynchronously. Currently, they all have to wait for all the others to finish their current tasks before completing voting. Ideally, if there is a recipe that could be completed with a subset of the agents and they are free, then there is not a good reason

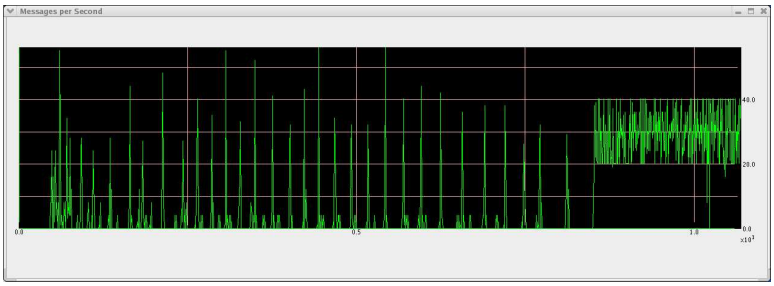

Figure 6.9: Overall Message Traffic for DAVS

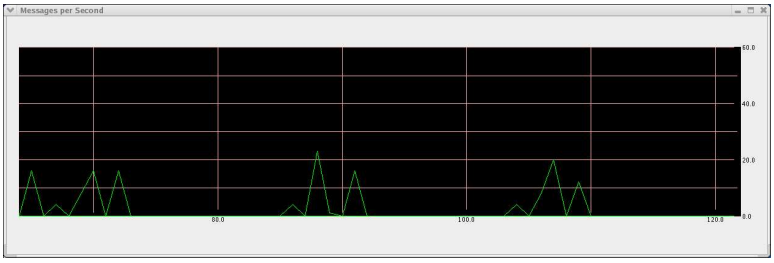

Figure 6.10: A Minute of Message Traffic for DAVS

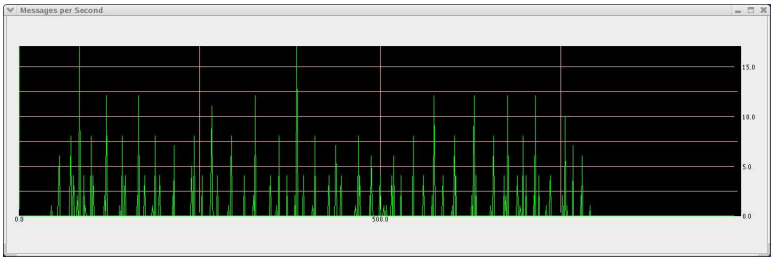

Figure 6.11: Overall Message Traffic for averaging

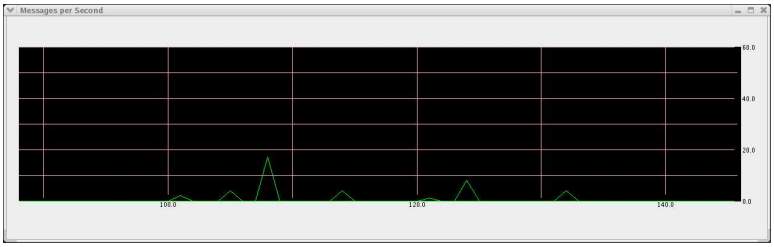

Figure 6.12: A Minute of Message Traffic for averaging

to wait for the rest to become free. This leads to a host of new issues, such as perhaps the recipe can be done quicker with more agents or a different subset, and whether it is worth waiting for others to finish. Deciding exactly how and when to form teams is an interesting problem in itself, and solving it could lead to increased efficiency and robustness.

7 Conclusion

As discussed in the previous chapters, for efficient teamwork, agents must be created that take into account uncertainty in the world. Two separate systems were analyzed for tolerance to communication failure and tolerance to uncertainty in the environment. They were shown to be effective in simulations on the surface of Mars, but they have additional, generalizable characteristics that can make them suitable for a host of different problems.

7.1 Other Applications

- **Remote home-appliance control** There has been talk for years about making a household full of electronic gadgets that can all work together. Using teams of software agents as presented here could make human interface easy. A user could input a goal on any device, which would then be able to move to where it could be accomplished most easily, then execute. All code except for the URQuake simulator is written in Java, which demonstrates how easily portable it is.

- **Remote surveillance** Suppose that a large group of cameras are tasked with watching an area for suspicious people. If computing resources are limited, then the system would have to prioritize what it is looking for, from simple motion

tracking to full face and vehicle recognition. By viewing the cameras as a team of agents, they can negotiate and be automatically tasked with the most important goals. It would be easy for even an untrained human operator to interact with the system and shift goal priorities to maintain optimal coverage.

- **General computer vision** Voting has been used extensively in computer vision, but usually only the plurality method. The fast Borda count as presented here could be useful in any task that needs to compile a large amount of uncertain data quickly. As an example, Borda count voting could prove effective in a large object recognition database that is searchable by attributes.

7.2 Future Work

Even after all the design and experimentation in this paper, the biggest question still remains. How can a system be created that combines tolerance to both communication error and environmental uncertainty at the same time? The answer lies in the dynamic forming of teams. Just as the software agents could remain in their current host when communication failed (Section 5.8,) agents can choose to form teams when it appears it will be beneficial (Section 6.7) and when enough messages get through that they can verify a joint intention. Combined with a small amount of activity recognition (i.e. noticing that others are starting to move,) such coordination becomes simple. Take the following example, based on the drill rocks on Mars scenario:

Rover A looks around and sees three idle agents, Rovers B, C, and D. Rover A decides to try to work with them to drill a nearby rock, so it communicates a desire to form a team. All three respond, and they conduct a vote and auction. The auction is completed, and Rover A believes that they are ready to begin working, so it starts moving to the rock. Halfway, it notices that only Rover B is moving. The other two are still sitting there, so it attempts to verify the recipe with them again. Rover C had missed a communication, but catches this one, so it begins moving as well. Rover D

does not respond, and Rover A concludes that D has failed. However, the recipe only needs three agents, so A, B, and C continue and complete it.

Such a scenario demonstrates that the success of the ultimate goal of an autonomous team that can handle any circumstance can only be possible by a merging of techniques. Using several different methods in parallel and creating hybrid methods can create agents stronger than the sum of their pieces. By combining tolerance to disruption with tolerance to uncertainty, truly robust teamwork can be attained.

Bibliography

[Appleby and Steward, 1994] Steve Appleby and S. Steward, "Mobile software agents for control in telecommunications networks," *BT Technology Journal*, 12(2):104–113, April 1994.

[Axelrod, 1984] Robert Axelrod, *The Evolution of Cooperation*, Basic Books, 1984.

[Balch and Arkin, 1997] Tucker Balch and Ronald Arkin, "Behavior-based Formation Control for Multi-robot Teams," 1997.

[Beri, 2004] Umang Beri, "FootyMania.com," July 2004.

[Blum and Langford, 1999] Avrim Blum and John Langford, "Probabilistic Planning in the Graphplan Framework," In *5th European Conference on Planning*, 1999.

[Brooks, 1986] Rodney Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, RA-2, No1:14–23, 1986.

[Brooks, 1989] Rodney A. Brooks, "A Robot that Walks; Emergent Behavior from a Carefully Evolved Network," *Neural Computation*, 1(2):253–262, 1989.

[Brooks, 1991] Rodney A. Brooks, "Integrated systems based on behaviors," *SIGART Bulletin*, 2(4):46–50, 1991.

[Brown *et al.*, 2004] Chris Brown, Peter Barnum, David Costello, George Ferguson, Bo Hu, and Michael Van Wie, "Quake II as a Robotic and Multi-Agent Platform," Technical Report 853, University of Rochester, October 2004.

[Chalupsky *et al.*, 2002] Hans Chalupsky, Yolanda Gil, Craig Knoblock, Kristina Lerman, Jean Oh, David Pynadath, Thomas Russ, and Milind Tambe, "Electric Elves: Agent Technology for Supporting Human Organizations," *Artificial Intelligence*, 23:11–24, 2002.

[Cohen and Levesque, 1990] Phil Cohen and Hector Levesque, "Intention is Choice with Commitment," *Artificial Intelligence*, 42(2-3):213–361, 1990.

[Cohen and Levesque, 1991] Phil Cohen and Hector Levesque, "Teamwork," *Nous*, 25(4):487–512, 1991.

[Davis and Smith, 1983] Randall Davis and Reid Smith, "Negotiation as a Metaphor for Distributed Problem Solving," *Artificial Intelligence*, 20(1):63–109, JAN 1983.

[de Borda, 1781] Jean-Charles de Borda, *Memoire sur les Elections au Scrutin*, 1781.

[Dias and Stentz, 2003] M. Bernardine Dias and Anthony Stentz, "TraderBots: A market-based approach for resource, role, and task allocation in multirobot coordination," August 2003.

[Fudenberg and Tirole, 1987] Drew Fudenberg and Jean Tirole, "Noncooperative Game Theory for Industrial Organization: An Introduction and Overview," Working papers 445, Massachusetts Institute of Technology (MIT), Department of Economics, April 1987.

[Fujishima *et al.*, 1999] Yuzo Fujishima, Kevin Leyton-Brown, and Yoav Shoham, "Taming the Computational Complexity of Combinatorial Auctions: Optimal and Approximate Approaches," In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 548–553. Morgan Kaufmann Publishers Inc., 1999.

[Gat, 1992] Erann Gat, "Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-world Mobile Agents," *Artificial Intelligence*, 3(4):251–288, 1992.

[Genesereth and Ketchpel, 1994] Michael R. Genesereth and Steven P. Ketchpel, "Software agents," *Commun. ACM*, 37(7):48–ff., 1994.

[Gerkey and Mataric, 2002] Brian P. Gerkey and Maja J. Mataric, "Sold!: Auction methods for multi-robot coordination," *IEEE Transactions on Robotics and Automation Special issue on multi-robot systems*, 18(5):758–768, October 2002.

[Grassé, 1959] Pierre-Paul Grassé, "La Reconstruction du nid et les Coordinations Inter-Individuelles chez Bellicositermes Natalensis et Cubitermes sp. La théorie de la Stigmergie: Essai d'interpretation du Comportement des Termites Constructeurs," *Insectes Sociaux*, 6:41–81, 1959.

[Gray, 1995] Robert S. Gray, "Transportable Agents," 1995.

[Grosz and Kraus, 1993] Barbara Grosz and Sarit Kraus, "Collaborative Plans for Group Activities," In *Proceedings of the 1993 International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 367–373, 1993.

[Grosz and Kraus, 1996] Barbara Grosz and Sarit Kraus, "Collaborative Plans for Complex Group Action," *Artificial Intelligence*, 86(2):269–357, 1996.

[Grosz and Kraus, 1998] Barbara Grosz and Sarit Kraus, "The evolution of SharedPlans," 1998.

[Grosz and Sidner, 1990] Barbara Grosz and Candace Sidner, "A Reply to Hobbs," In Phil Cohen, Jerry Morgan, and Martha Pollack, editors, *Intentions in Communication*, pages 461–462. MIT Press, Cambridge, MA, 1990.

[Hayes-Roth, 1995] Barbara Hayes-Roth, "An Architecture for Adaptive Intelligent Systems," *Artificial Intelligence*, 72:329–365, 1995.

[Hayes-Roth *et al.*, 1995] Barbara Hayes-Roth, Karl Pfleger, Philippe Lalanda, Philippe Morignot, and Marko Balabanovic, "A Domain-Specific Software Architecture for Adaptive Intelligent Systems," *IEEE Transactions on Software Engineering*, 21(4):288–301, 1995.

[Huber and Durfee, 1995] Marcus J. Huber and Edmund H. Durfee, "On Acting Together: Without Communication," In *Spring Symposium Working Notes on Representing Mental States and Mechanisms*, pages 60–71. American Association for Artificial Intelligence, Stanford, California, 1995.

[Huber, 1981] Peter J. Huber, *Robust Statistics*, Wiley-Interscience, 1981.

[Hunsberger, 1999] Luke Hunsberger, "Making Shared Plans More Concise and Easier to Reason About," In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 81–98. Springer-Verlag: Heidelberg, Germany, 1999.

[Hunsberger, 2002] Luke Hunsberger, "Group Decision Making and Temporal Reasoning," 2002.

[Jennings, 1995] Nicholas R. Jennings, "Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions," *Artificial Intelligence*, 75(2):195–240, 1995.

[Kinny *et al.*, 1992] David Kinny, Magnus Ljungberg, Anand Rao, Elizabeth Sonenberg, Gil Tidhar, and Eric Werner, "Planned Team Activity," In C. Castelfranchi and E. Werner, editors, *Artificial Social Systems — Selected Papers from the Fourth European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW-92 (LNAI Volume 830)*, pages 226–256. Springer-Verlag: Heidelberg, Germany, 1992.

[Klemperer, 1999] Paul Klemperer, "Auction Theory: A Guide to the Literature," Microeconomics 9903002, Economics Working Paper Archive at WUSTL, March 1999.

[Kotz and Gray, 1999] David Kotz and Robert S. Gray, "Mobile Agents and the Future of the Internet," *ACM Operating Systems Review*, 33(3):7–13, August 1999.

[Laird *et al.*, 1987] J. E. Laird, A. Newell, and P. S. Rosenbloom, "SOAR: an architecture for general intelligence," *Artificial Intelligence*, 33(1):1–64, September 1987.

[Lehman *et al.*, 2004] Jill Lehman, John Laird, and Paul Rosenbloom, "A Gentle Introduction to Soar, an Architecture for Human Cognition," 2004.

[Levesque *et al.*, 1990] Hector Levesque, Phil Cohen, and José Nunes, "On Acting Together," In *In Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI-90*, pages 94–99, 1990.

[Majercik and Littman, 1998] Stephen M. Majercik and Michael L. Littman, "MAXPLAN: A new approach to probabilistic planning," In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 86–93, 1998.

[Mataric; *et al.*, 2003] Maja J. Mataric;, Gaurav S. Sukhatme, and Esben H. Astergaard, "Multi-Robot Task Allocation in Uncertain Environments," *Autonomous Robots*, 14(2-3):255–263, 2003.

[Minar *et al.*, 1999] Nelson Minar, Kwindla Hultman Kramer, and Pattie Maes, *Cooperating Mobile Agents for Dynamic Network Routing*, chapter 12, Springer-Verlag, 1999, ISBN: 3-540-65578-6.

[Mitchell, 1990] Tom Mitchell, "Becoming Increasigly Reactive," In *Proceedings of AIII-90, Cambridge, MA*. MIT Press, 1990.

[Moulin, 1995] Herve Moulin, *Cooperative Microeconomics*, Princeton University Press, 1995.

[Nanson, 1882] E.J. Nanson, "Method of election," In *Transactions and Proceeding of the Royal Society of Victoria*, volume 18, pages 197–240, 1882.

[Nasraoui, 2005] Olfa Nasraoui, "A Brief Overview of Robust Statistics," 2005.

[Newell, 1990] Allen Newell, *Unified theories of cognition*, Harvard University Press, 1990.

[Park, 2004] Anthony Sang-Bum Park, "A service-based agent system supporting mobile computing," 2004, Doctoral thesis.

[Pynadath and Tambe, 2002] David Pynadath and Milind Tambe, "Team coordination among distributed agents: Analyzing key teamwork theories and models," 2002.

[Rao and Georgeff, 1995] Anand Rao and Michael Georgeff, "BDI-agents: from theory to practice," In *Proceedings of the First Intl. Conference on Multiagent Systems*, 1995.

[Reich, 1997] Barry Reich, "An architecture for behavioral locomotion," 1997.

[Reynolds, 1987] Craig W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics*, 21(4):25–34, 1987.

[Saari, 1994] Donald G. Saari, *Geometry of Voting*, Springer-Verlag, 1994.

[Sandholm, 2002] Tuomas Sandholm, "Algorithm for optimal winner determination in combinatorial auctions," *Artificial Intelligence*, 135(1-2):1–54, 2002.

[Scerri *et al.*, 2004] Paul Scerri, David Pynadath, Nathan Schurr, Alessandro Farinelli, Sudeep Gandhe, and Milind Tambe, "Team Oriented Programming and Proxy Agents: The Next Generation, Springer, LNAI 3067," In *In Proceedings of 1st International Workshop on Programming Multiagent Systems*, 2004.

[Smith, 1980] Reid G. Smith, "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *IEEE Transactions on Computers*, C-29(12):1104–1113, DEC 1980.

[Sonenberg *et al.*, 1994] Elizabeth Sonenberg, Gil Tidhar, Eric Werner, David Kinny, Magnus Ljungberg, and Anand Rao, "Planned Team Activity," 1994.

[Steels, 1990] Luc Steels, "Cooperation Between Distributed Agents Through Self-organization," *Decentralized A.I.*, pages 175–196, 1990.

[Tambe, 1997a] Milind Tambe, "Agent architectures for flexible, practical teamwork," In *National Conference on Artificial Intelligence (AAAI)*, 1997.

[Tambe, 1997b] Milind Tambe, "Towards Flexible Teamwork," *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[Theraulaz *et al.*, 1990] Guy Theraulaz, Simon Goss, Jacques Gervet, and Jean-Louis Deneubourg, "Task differentiation in Polistes wasp colonies: a model for self-organizing groups of robots," In *Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, pages 346–355. MIT Press, 1990.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe, "Integrating planning and learning," *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), 1995.

[Wie, 1997] Michael Van Wie, "Establishing Joint Goals through Observation," In Carnegie Mellon University David P. Casasent, editor, *Intelligent Robots and Computer Vision XVI:Algorithms, Techniques, Active Vision, and Materials Handling*, Pittsburgh, PA, USA, October 1997. The International Society for Optical Engineering.

[Wie, 2000] Michael Van Wie, *Role Selection in Teams of Non-communicating Agents*, PhD thesis, U. of Rochester Dept. of Computer Science, 2000, In DSpace: http://hdl.handle.net/1802/803.

[Zlot *et al.*, 2002] Robert Michael Zlot, Anthony Stentz, M. Bernardine Dias, and Scott Thayer, "Multi-Robot Exploration Controlled By A Market Economy," In *IEEE International Conference on Robotics and Automation*, May 2002.

[Zlotkin and Rosenschein, 1996] Gilad Zlotkin and Jeffrey S. Rosenschein, "Mechanisms for Automated Negotiation in State Oriented Domains," *Journal of Artificial Intelligence Research*, 5:163–238, 1996.