A Framework for the Development of Scalable Heterogeneous Robot Teams with Dynamically Distributed Processing

by

Adrian Martin

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

University of Toronto Institute for Aerospace Studies
University of Toronto

A Framework for the Development of Scalable Heterogeneous Robot Teams with Dynamically Distributed Processing

Adrian Martin

Doctor of Philosophy

University of Toronto Institute for Aerospace Studies
University of Toronto

2013

Abstract

As the applications of mobile robotics evolve it has become increasingly less practical for researchers to design custom hardware and control systems for each problem. This research presents a new approach to control system design that looks beyond end-of-lifecycle performance and considers control system structure, flexibility, and extensibility. Toward these ends the Control *ad libitum* philosophy is proposed, stating that to make significant progress in the real-world application of mobile robot teams the control system must be structured such that teams can be formed in real-time from diverse components. The Control *ad libitum* philosophy was applied to the design of the HAA (Host, Avatar, Agent) architecture: a modular hierarchical framework built with provably correct distributed algorithms.

A control system for exploration and mapping, search and deploy, and foraging was developed to evaluate the architecture in three sets of hardware-in-the-loop experiments. First, the basic functionality of the HAA architecture was studied, specifically the ability to: a) dynamically form the control system, b) dynamically form the robot team, c) dynamically form the processing network, and d) handle heterogeneous teams. Secondly, the real-time performance of the distributed algorithms was tested, and proved effective for the moderate sized systems tested.

Furthermore, the distributed Just-in-time Cooperative Simultaneous Localization and Mapping (JC-SLAM) algorithm demonstrated accuracy equal to or better than traditional approaches in resource starved scenarios, while reducing exploration time significantly. The JC-SLAM strategies are also suitable for integration into many existing particle filter SLAM approaches, complementing their unique optimizations. Thirdly, the control system was subjected to concurrent software and hardware failures in a series of increasingly complex experiments. Even with unrealistically high rates of failure the control system was able to successfully complete its tasks.

The HAA implementation designed following the Control *ad libitum* philosophy proved to be capable of dynamic team formation and extremely robust against both hardware and software failure; and, due to the modularity of the system there is significant potential for reuse of assets and future extensibility. One future goal is to make the source code publically available and establish a forum for the development and exchange of new agents.

Acknowledgments

I have relied on the support of many people during the last five years (and more) at the University of Toronto, and would like to extend my thanks to each of them.

To my loving girlfriend Mabel, who has supported me and been patient with me in both the good times and the bad.

To my advisor, Dr. Reza Emami, who has gone above and beyond in mentoring me during both my M.A.Sc. and Ph.D. degrees. His dedication, experience, thoroughness, and understanding have been critical in shaping this work and ensuring that it grew into everything it could be.

To the members of my examination committee, Dr. Gabriele D'Eleuterio and Dr. Christopher Damaren, who gave me the feedback and helpful criticisms that kept this research on track.

To my colleagues in the Space Mechatronics group, Robin Chhabra, Jason Kereluk, Peter Martin, and Victor Ragusila, among others, who provided a sounding board of ideas and a vent for frustrations.

And last but not least, to my family and Mabel's family, who never stopped supporting me and never stopped asking if I was done yet.

Table of Contents

List of Tables

List of Figures

List of Appendices

List of Abbreviations

AF-#	Architecture Functionality Experiment #
AP-#	Algorithm Performance Experiment #
CBBA	Consensus-Based Bundle Algorithm
DDB	Distributed Database
FIFO	First-in-first-out
HAA	Host, Avatar, Agent
HIL	Hardware-in-the-Loop
JC-SLAM	Just-in-time Cooperative Simultaneous Localization and Mapping
LIFO	Last-in-first-out
NFP	No Forward Propagation
OAC	Ordered Atomic Commit
OOO	Out-of-order
POG	Probabilistic Occupancy Grid
R-#	Robustness Experiment #
SLAM	Simultaneous Localization and Mapping
TCP/IP	Transmission Control Protocol/Internet Protocol
UFD	Unreliable Failure Detector

Chapter 1
Introduction

In response to many of the fundamental challenges to the development of mobile robot teams this research presents a new approach to control system design. Robots have been proven capable of accomplishing many useful tasks, and in many applications teams of inexpensive robots can accomplish tasks faster and more efficiently than a single more expensive robot. The advantages of robot teams are typically listed as: cost per system, robustness through redundancy, parallel processing, scalability [1,2] , and the ability to service larger areas and accomplish multiple tasks simultaneously [3]. In [4] several successful implementations are cited in areas such as search and rescue, perimeter surveillance, and mapping and exploration, and many more can be found in the literature. However, even these successes have their limitations and note many of the challenges facing the developers of robot teams. Questions of control [4], communication [5], collaboration[6], task coordination[7], heterogeneity [8], and team formation [9] have no definitive answers, and likely never will. Yet it is clear that as the applications of mobile robotics evolve it will become increasingly less practical for researchers and developers to design custom hardware for each problem. Similarly, as the applications become more complex it will be impractical to begin each control system from scratch. Despite this, for various reasons including problems of transparency, portability, and stability the near universal approach of current researchers is to build custom solutions for every team/task [10], redoing vast amounts of work and greatly hindering the growth of the field for practical real-world applications. Thus, it becomes essential to have an architecture that allows teams incorporating diverse robot hardware and facilitates extensions to the control system which require minimal or no changes to the existing implementation.

The goals of this research became to expand the conversation on control system design to include not just end-of-lifecycle performance but also control system structure, robustness, and extensibility, and to use techniques from distributed computing to develop a generic and flexible architecture for controlling robot teams. Toward these ends the Control *ad libitum* philosophy was proposed, stating that in order to make significant progress in the real-world use of mobile robot teams the control system must be structured such that teams can be formed in real-time from diverse components. This philosophy was followed in the design of the HAA (Host,

Avatar, Agent) architecture, which uses a modular hierarchical approach built on a distributed network of processors. After implementing the architecture, using provably correct distributed algorithms, a fully functional control system was developed to test its performance under a number of common mobile team tasks: exploration and mapping, search and deploy, and foraging. Since extensibility and reusability are issues close to the core of this research, one future goal is to make the source code for the architecture and control system publically available and to establish a forum for the development and exchange of new agents.

The results from extensive Hardware-in-the-Loop experimentation show the control system to perform well. The control system was capable of dynamically forming the control system based on the needs of the task, changing the set of active agents to adapt to the currently available resources. The control system was also capable of gracefully recovering from software, processor, and robot failures. The first series of experiments study the basic functionality of the HAA architecture, specifically its ability to: a) dynamically form the control system based on the task requirements, b) dynamically form the team from available robot hardware, c) dynamically form the processing network based on available processor resources, and d) handle heterogeneous teams and allocate robots between tasks based on their capabilities. The second series of experiments analyze the performance of the distributed algorithms for various system sizes, and each algorithm demonstrated highly acceptable real-time performance and no issues of scalability for the small-to-moderate sized systems tested. The Simultaneous Localization and Mapping (SLAM) problem is fundamental to the implementation of virtually any robot team, and so a distributed and scalable algorithm was developed as part of this research. The algorithm demonstrated accuracy equal to or better than traditional approaches in resource constrained scenarios, while reducing exploration time by over 17% for the tested mapping scenarios. The third and final series of experiments tested ability of the architecture to handle concurrent software and hardware failures, and all missions were able to successfully complete their tasks even with failure rates set far higher than realistic expectations, including a scenario where each software module was set to fail every 0.5-1.5 minutes.

1.1 Contributions

There are four major areas where this work has made contributions to the research community:

1. A structured, big-picture, look was taken at the design of robot teams throughout their lifecycle, leading to the Control *ad libitum* philosophy. A comprehensive set of design motivations were identified and a series of metrics arranged to evaluate aspects of control system design strategy at all stages of development.

2. The HAA architecture presents a framework that allows for the modular design of heterogeneous robot teams while inherently providing many advantages in terms of robustness, efficiency, and scalability. The two primary components of HAA are:

Scalable Hierarchical Control: A scalable control system that requires no prior knowledge of team membership in order to efficiently perform tasks. It allows the immediate integration of almost any robot into the team and allows the team to continue functioning in the event of hardware failure.

Dynamically Distributed Processing: A distributed system allows the transfer of agents in the processor network to balance the load, reduce latency between agents, or recover agents in the event of hardware failure. Such a network also enables hybrid architectures with both centralized and distributed components without sacrificing performance or robustness.

3. A fully-realized implementation of HAA was developed, including a foundation of provably correct distributed algorithms. The control system was tested in a number of common mobile robot tasks and acquitted itself well even in extreme failure scenarios.

4. Just-in-time Cooperative Simultaneous Localization and Mapping (JC-SLAM) is a real-time, distributed, scalable implementation for heterogeneous mobile robot teams. It uses an out-of-order processing strategy to efficiently make use of processing resources and in experimentation has demonstrated a 17% reduction in exploration time compared to two traditional SLAM approaches for the tested mapping scenarios.

1.2 Thesis Overview

The thesis is broken down into eight chapters. Chapter 2 reviews the topics related to multi-robot systems, including: control, collaboration, communication, resource sharing, and load balancing. Of particular interest were the underrepresented areas of dynamically formed teams and behaviour migration. Limited hardware can be utilized effectively if correctly shared among a group of robots, and distributed processing can improve the performance of a control system.

Chapter 3 studies the team lifecycle and presents the core tenets of Control *ad libitum*: Transparency, Versatility, Adaptability, Modularity, Diversity, Persistency, and Efficiency. The chapter also emphasises the importance of using the right tools to quantify aspects of a control system beyond simple end-of-lifecycle performance.

Chapter 4 introduces the HAA architecture and the control system abstraction of *hosts* (processors), *avatars* (robot hardware), and *agents* (control system modules). A distributed processing network is formed by the hosts and agent modules are allowed to transfer between hosts in order to balance load, improve communication latency, and recover from failures. A complete HAA implementation is developed using provably correct distributed algorithms.

Chapter 5 presents JC-SLAM, a distributed particle filter SLAM algorithm designed following the tenets of Control *ad libitum*. JC-SLAM adopts a strategy of out-of-order processing to allow higher rates of sensor processing in constrained systems, but performs identically to traditional ordered approaches when resources are plentiful.

Chapter 6 provides the details of the HAA implementation used in this research. Many design choices must be made to transition from the general framework provided by HAA into a fully functioning implementation. Tools were designed for logging, visualization, automated testing, and, perhaps most importantly, offline debugging by recording the inputs to each agent for later playback. Chapter 6 also introduces the experimental scenarios that were used to evaluate the effectiveness of the control system: Mapping and Exploration, Congregate, and Forage.

Chapter 7 uses a Hardware-in-the-Loop simulation to experimentally demonstrate features of the control system and evaluate performance. Features such as dynamic formation, adapting to changing resources, agent allocation, avatar allocation, and cooperation are explored. Performance was evaluated for: a) the distributed algorithms, b) JC-SLAM vs. traditional SLAM strategies, and c) failure scenarios up to and including concurrent host, avatar, and agent failure.

Chapter 8 closes the thesis by summarizing the findings and discussing several potential avenues for future research: a) an open-source agent library, b) standardization of agent interaction and recovery strategies, c) scalability of the distributed algorithms, and d) communication efficiency within the distributed database (DDB).

Chapter 2
Background

Developing and implementing a team of cooperative mobile robots is a very challenging task, yet the reward is an efficient and effective solution for many application problems. These applications span a wide range of practical, real world, scenarios, and include: working in hazardous environments, surveillance, target tracking, mine field demolition, Mars exploration, search and rescue, guarding, cleaning, and fire detection [11]. Sometimes tasks can be carried out by a single robot with powerful sensors and high processing capability, however, often these tasks can be carried out faster, more efficiently, and more robustly using a team of simpler and cheaper robots [4]. Large amounts of research has been done on a multitude of aspects of team development, but many fundamental questions are far from answered, and there are always new strategies or twists on previous techniques being studied. This chapter is concerned with reviewing the key elements which must be considered when designing a cooperative team, and identifying potential areas that could benefit from new and innovative ideas. Due to the size of the field it is impossible to mention every strategy or technique that has been developed, and so an effort is made to discuss either those that are representative of common approaches or those that present a novel and interesting take on an issue.

2.1 Multi-robot Systems

2.1.1 Teams vs. Single Robots

Some robot applications demand a robot with powerful sensors and high processing capacity, which usually corresponds to a high cost, but in many applications it is possible use a team of simpler robots to accomplish the tasks faster and more efficiently. For example, search and rescue, load pushing, surveillance, and mapping [4]. Depending on the structure of the team there may not be any inherent cost benefit, however, there are a number of other benefits that come with multi-robot systems: efficiency, cost per system, robustness through redundancy, parallel processing, and scalability[2,12]. Furthermore, larger areas can be serviced and multiple tasks can be accomplished simultaneously by spreading out the team [3], and there is a potential for self-diagnosis and self-repair of failures in robot teams [13]. One example of learning where teams have the advantage over individual robots is presented in [14]. In that research the problem

was learning to visually identify objects in the environment, and by sharing information the team was able to learn more quickly.

2.1.2 Homogenous vs. Heterogeneous Teams

When building a team of robots there are a number of possibilities regarding the homogeneity of its members. Obviously, it is possible to select a number of different types of robots to form the team, which results in a physically heterogeneous population; but even in the case of physically identical robots it is possible to introduce heterogeneity in their controllers. The concept of social entropy is introduced in [5], providing a way to quantitatively rate the diversity of a team based on factors relevant to the application.

The heterogeneous team of robots presented in [7] is an example that takes advantage of physical heterogeneity. Their team consisted of a few highly capable expensive robots equipped with powerful sensors and processors and a large number of simpler robots with weak sensors and processors. Since the simple robots were incapable of accurate localization and navigation, once the environment had been mapped the powerful robots guided the simple robots into place to form a sensor network. In this way the cost of the team was reduced by almost an order of magnitude while still providing coverage for large areas. An example of a physically homogeneous team with heterogeneous controllers is discussed in [8]. Inspired by specialization in insect colonies, as a swarm of robots learns how to perform basic tasks related to finding and collecting objects they begin to develop proficiencies in different areas and the tasks are allocated throughout the team based on fitness.

Diversity has other potential benefits in addition to allowing a balance of cost, capability, and number of robots. As in nature, heterogeneity and diversity in a population can provide much needed robustness. In experiments using robots emulating wolf-pack hunting strategies, [15] showed that heterogeneous teams composed of both peak and senescent “wolves” could outperform a team of purely peak wolves in certain scenarios. There is also the consideration of scalability and utilizing all available resources. A system that is capable of handling heterogeneity can potentially make use of whatever robots are on hand, which can reduce the cost of updating or replacing robots [16].

2.1.3 Dynamically Formed Teams

One interesting area of research that has received little study is the concept of dynamic or "pickup" teams [16]. Rather than planning the team membership and methods of interaction in advance, it is more useful to have a system that can dynamically adapt based on the available resources and the environment. At the most basic level it comes down to forming a useful and efficient team from a set of robots given no *a priori* information about their capabilities. The work in [16] outlines some reasons why this capacity is needed: 1) it is impractical for a single group to develop large teams of expensive robots simultaneously, 2) engineering coordination strategies by hand is time consuming and may not be acceptable in emergency situations, and 3) if a robot fails or is removed from the team it is necessary to replace it and a new robot of the same type may not be available. A strategy for accomplishing this goal through communication between potential team members is outlined in [16], and a treasure hunting application using two types of robots is presented. [17] also describes a need for heterogeneous "impromptu teams," and proposes an ontology-based communication protocol to allow diverse team members to communicate physical concepts.

The issue of dividing a group of robots into sub-teams is studied in [18] and [19]. In [18] sub-teams are formed and broken up depending on the effectiveness of the team formation (relative positions). It was found that dynamic formation improved performance except in cluttered environments, at which point the team spent too much time reforming and efficiency dropped. [19] starts from the idea that there are a number of tasks to perform and a number of robots in the environment, and the question becomes how to distribute the robots among the tasks. The robots are rated based on their capabilities and then the working time of different combinations of robots is estimated.

2.1.4 System Design

An unfortunate trend of single-use design is prevalent in industry and academia. More often than not a particular robot or control system is designed with a specific task in mind, and task performance becomes the only metric for success. Though many successful applications have been developed, this limited approach does not lend itself to team robustness and potential for building on top of existing control systems. The aptly-titled article *1,001 Robot Architectures for*

1,001 Robots [10] highlights this issue and asks the question "Is it really impossible to subject robot architectures and software systems to any objective performance evaluation?" The issue is also touched upon in [20], which highlights the main features of its architecture as the ability to use off-the-shelf software to develop and run standardized control system modules.

2.2 Control

There are many different strategies for controlling multi-robot systems, some cooperative, some competitive, some centralized, and some distributed. Each of these areas has been explored in depth but there is no clearly superior strategy, and in many cases hybrid architectures such as the deliberative/reactive combination developed in [3] or the partially centralized control in [21] are used. Another factor that is fundamentally tied to control and coordination is communication, which is discussed in Section 2.4.

2.2.1 Centralized vs. Decentralized Control

In its purest form centralized control means that every decision and resulting action passes through a single point, taking into account the entire state of the system. At the opposite end of the spectrum, fully decentralized control executes independently on each robot with little or no explicit communication with the other team members. Both methods have some advantages and disadvantages. Centralized control can potentially be used to find optimal solutions; however, the computational requirements increase rapidly as the number of robots in a team grows [21], and it is not robust in the event of failure of a key component. On the other hand, while a distributed system has advantages in terms of parallel computation, robustness, and fault tolerance [22], in a fully decentralized system each robot must be capable of satisfying all of its sensing and processing needs on-board, which may not be an efficient use of resources [4]. Many of these disadvantages can be mitigated by taking a balanced approach, for example the system in [23] uses a centralized task coordinator to assign tasks to a distributed team of robots, even allowing robots to join and leave the team at any time. The interesting application in [22] takes the multi-agent concepts of decentralized control and applies it to a single robot with multiple processors in order to take advantage of its modularity and robustness and reduce the complexity of the system. [24] takes the same approach of decentralized control for a single robot but also notes that such an approach has benefits in terms of simplification via modularization, user friendly design,

effective resource utilization, and future expansion of the control system. These are all very important features and are central to the foundation of this work.

2.2.2 Control Hierarchy

Many traditional control strategies use a hierarchical structure [22], where components are broken up into levels which can control or override the levels below them. This approach is common in controlling teams as well, often taking the form of supervisors which monitor the behaviour of the team and coordinates their actions. It is also possible to implement controllers without a hierarchy, where each component is considered equal. This is done in [25], by having each robot advertize their task suggestions and having other robots volunteer their services. A dynamic approach is taken in [26], and based on the observed surroundings each robot decides whether it is a leader or follower. In contrast to this distributed approach, [27] uses a hierarchical controller with a centralized path planner and local robot control, though they note that having a single point of failure is not ideal.

2.2.3 Task Coordination

The problem of task assignment and coordination has two facets. First, there is the issue of coordinating the efforts of each robot so that they work together toward a common goal. In a centralized system this can be dealt with in a straight forward manner, but the solution becomes less clear for decentralized control. Depending on the level of communication between the robots a number of routes can be taken. When communication is possible a common strategy is to hold task auctions such as in [16], [25], and [28], where tasks are put forward and each robot bids for contracts. An architecture for constructing robot teams to facilitate market-based task allocation is presented in [29]. This architecture interestingly shares some of the ideas founding the architecture in this research, including modularizing control system functions into agents for each robot and a shared database. However, it does not entirely abstract agents from the robot hardware or allow distributed processing or agent transfer. When communication is not possible it is harder to prevent overlap in tasks and other strategies must be used [7]. A number of learning algorithms have been adapted to these ends, [30,31].

The second issue is that of distributing the tasks in a way that makes efficient use of available resources and accomplishes high priority tasks in timely manner. This can be reduced to an

optimization problem, but in a system with many tasks and many robots it may not be practical to implement this in real-time [19]. Studies done in [19], [32], and [33] looked at different sub-optimal algorithms that still produce good results.

2.3 Collaboration

Collaboration in multi-robot systems can happen either competitively or cooperatively. Competition and aggression appear in many animal societies as a method for assuming roles and coordinating groups. This approach was used in [34] to successfully manage a team of robots in a transportation task where the robots would encounter bottlenecks while travelling or during pick-up and drop-off. Another competitive strategy was used in [21] to coordinate the collision free movement of a team of robots in a dynamic environment using a partially centralized sensory system. These techniques can be useful between individual robots competing for space and resources or between teams of robots performing tasks in close proximity, and do not preclude having some form of cooperation at other levels. Cooperation can occur passively, with robots sharing knowledge or recognizing when other team members are carrying out tasks and assisting them [35]. There are many examples of this in nature, and [36] demonstrates an example of the "group escape" behaviour where a team of robots rapidly elude predators with no inter-robot communication. Cooperation can also occur actively, where robots communicate their desires and make plans with other team members as in [16].

2.3.1 Knowledge Sharing

Knowledge sharing can have obvious benefits to the performance of a robot team. When carrying out tasks the decision making process is heavily affected by the knowledge of the environment, yet often a single robot's sensors provide only a highly incomplete view of its surroundings [25]. By providing robots with the ability to communicate and share their knowledge it is possible to fill in many of the gaps and thus make more informed decisions. This is particularly apparent in the collaborative map making through sensor fusion done in [25]. The "blackboard" communication technique used in many applications, [8,25], can be considered a form of knowledge sharing. Robots post information to a global blackboard which is synchronized throughout the system. [37] proposes a set of low-cost/open source middleware solutions to form a knowledge sharing/communication system via a shared database. Knowledge

sharing can also be very useful in learning tasks where learning experiences can be broadcast to other robots [5] or evolved behaviours can be transferred between robots [38]. However, these techniques rely heavily on communication and so storage and bandwidth requirements must be considered.

2.3.2 Cooperation Techniques

A number of cooperation techniques are listed in [6], including: socially acceptable decision making strategies based on social welfare functions, information based exploration using the concepts of entropy and frontiers, cooperative motion planning using a potential field, and decentralized goal planning using a shared map. A common strategy for groups of people performing complex tasks is to break the tasks down into smaller pieces and assign each job to a team member [30]. Each member may be specialized for a certain type of role or equally capable of completing any job. In this situation it becomes a question how to assign the roles and when switching roles is advantageous [30]. This type of dynamic role switching is also discussed in [35], which notes the potential risks involved if no protocol is put in place to ensure that no role is left unfulfilled. The Skills, Tactics, and Plays (STP) framework is applied to cooperation in dynamically formed heterogeneous teams in [16]. Plays consist of a set of roles, a sequence of actions for each role to perform, and methods for evaluating the applicability, completion, and selection likelihood in a given situation. The action sequences performed by each role are tightly coordinated to ensure synchronization between each team member.

Since geometric solutions to navigation are generally difficult to implement in real-time, particularly when the movement of many robots must be coordinated, a receding horizon strategy is used for coordinated navigation in [39]. At each time step every robot solves their "personal problem" based on available knowledge but only executes the first control action, and the next step the robot solves the problem again based on updated information. The information used at each step only includes the actions of the neighboring robots, thus simplifying the problem and allowing it to be solved efficiently. This issue of scope or level of awareness is an important consideration that affects the computational efficiency and effectiveness of both control and learning algorithms. The balance that must be achieved is also discussed in the target tracking application presented in [40], which considers how increasing levels of awareness impact the dimensionality of the search space for learning algorithms.

2.4 Communication

Communication, or lack of communication, within a team greatly affects the structure and capabilities of the control system. Four different levels of communication are traditionally considered: no communication, state communication, goal communication, and implicit communication [5,40]. With no communication the robots must rely on their sensors to determine the action of other team members, state communication means that robots may detect/query the current state of other members, goal communication is when robots actively broadcast information about their current plans, and implicit communication where robots communicate through the environment [40]. The results in [5] show that state and goal communication can provide significantly improved performance, up to 19%, when compared to no communication in foraging and consuming tasks. It is demonstrated in [40] that coordination can be learned using only awareness of the neighboring robots, but [40] admits that it is not always possible to achieve awareness without communication.

Unfortunately, communication is not free. The necessary hardware adds to the cost and complexity of the system [5], and as in the case of [32] even the physical size and power consumption of the components can be an issue for smaller robots. As with any hardware component there is a chance of failure, but even when the communication system is working there are factors such as range, interference, and obstacles that can degrade the signal and reduce bandwidth until eventually no communication is possible [7]. Some works such as [41] and [42] study how robots/mobile relays can be dynamically positioned to improve network quality over large areas. Robot swarms forming *ad hoc* wireless networks are studied in [43]. [44] takes the problem one step further and considers scenarios where relays have to travel back and forth to ferry information. Another concern for communication is network infrastructure. In heterogeneous teams different network formats may be used and require translators to allow robots to communicate with each other [45].

2.5 Resource Sharing and Load Balancing

In complex robot systems hardware considerations often limit the potential of the control system. Factors such as processing power, data storage, communication bandwidth, and even sensors are

generally limited due to cost, size, or power constraints [32]. Thus, it seems desirable to use these resources as efficiently as possible in order increase the effectiveness of the system.

2.5.1 Resource Sharing

Both [32] and [33] discuss the same team of robots, but the focus of each is on a slightly different aspect of resource sharing. The work uses a team of small robots for urban exploration and surveillance, but due to size limitations the robots are severely limited in terms of processing power and communication capabilities; and so scheduling access to the communication channels is critical for effective operation [32]. Each resource (communication channel, processor, robot chassis, or sensor input) has a Resources Controller and scheduling is handled through a central Resource Controller Manager (RCM). By dynamically allocating these resources at execution time it is possible to develop a flexible and robust system that degrades gracefully as resources become limited [32]. Two different scheduling algorithms are studied in [33]: the first tries to find an optimal solution which maximizes the number of resources in use at any time, while the second uses a decision tree to rank tasks based on factors such as priority and minimum run time.

The issue of sharing sensor data among a surveillance team and their controllers is considered in [46]. Due to bandwidth limitations it is impossible to transfer the bulk of data to the interested party, and so an abstraction was developed that models the sensors as a distributed database which can be queried for information. Taking the opposite approach to [32] and [33], [46] utilizes robots equipped with gigabytes of onboard storage and powerful processors. Information and processing requests are sent out over the network, processed locally by the owner of the information, and the results are sent back to the client.

2.5.2 Load Balancing

A major challenge of resource sharing is how to balance the demand for resources across the system [33]. Because of the extremely limited processors available on the robots in [33] all image processing and high level control must be done by a network of off-board workstations. Because the architecture had no method for process migration it was impossible to switch tasks between workstations once they had begun, which led to less than optimal distribution if multiple tasks were started simultaneously [33]. In a similar application, [47] demonstrates a system for distributing image processing tasks between off-board and on-board processors.

In [46] each robot is equipped with a powerful processor and only responsible for a unique set of sensor data, any task involving that data can be shipped over the network to be processed locally and then the results are transmitted back. This presents some significant advantages over a centralized processing station, since it allows processing to be distributed over a network of computers as well as greatly reducing the required communication bandwidth [46]. However, it also has certain drawbacks in some situations, for example if many controllers wish to utilize the same data simultaneously all processing must be carried out by the owner of that data and there is no way to take advantage of other processors that may be available.

2.5.3 Distributed Processing

When controlling mobile robots a number of problems must be solved in real-time, such as motion control, mapping, localization, and sensor processing [48]. Some of these tasks have high computational requirements and it may be impossible for all tasks to be handled simultaneously by a single processor. In these situations a distributed network of processors can be utilized. Such a network has many advantages: it can be shared among a group of robots, load can be balanced throughout the network, processors can be added and removed depending on the requirements of the system, the system is robust in the case of processor failure, and as long as there is a common interface the underlying architecture of each processor can be different [48]. On the other hand [48] also notes a number of new concerns that are introduced to the system. First, there is a potential for latency between control system components, and different modules can no longer be considered tightly coupled. Second, not all processors have immediate access to all information, and so communication protocols must be instituted to ensure knowledge is shared in a timely manner. And finally, some strategy must be used to integrate each of the components into a single working system. Centralized management strategies for distribution are used in [48], [32], and [33].

A slightly different approach is used in applications such as [46] and [49], where each robot has an on-board processor. As discussed previously, [46] distributes processing tasks based on the location of the required data, and ideally the demand for data is balanced throughout the system and thus the load is evenly distributed. In [49] a solution to collision free routing is presented that divides the task among the onboard processors of each robot.

One concept in distributed processing that seems to have been overlooked is the idea that a processor is a processor whether it is stationary or on-board, and can be harnessed for any computational task regardless of its location. Applications that consider sharing computational power tend to only consider stationary processors, while applications that utilize on-board processors limit tasks to those related to that robot or its sensors. There is no obvious reason why a system that merges each of these approaches could not be implemented: tasks distributed between stationary and on-board processors based on available resources and the data required for each computation task shared throughout the system. Such a system would require some framework for balancing the computational load and sharing data in a way that efficiently utilized limited communication resources.

2.5.4 Behaviour Migration

The topic of behaviour migration introduced in [38] has some very interesting potential applications to resource sharing. The focus of [38] is on autonomous and semi-autonomous robot applications with long lived and stateful tasks. These are not easily accomplished since the operating time of a typical robot is relatively short and in natural environments robots have rather high failure rates. When a robot fails or is forced to retire to replenish its resources it is desirable that its role be taken over by another robot. However, it is often the case that the robot's controller will have gathered information and perhaps learned behaviours to improve its performance and so these must be transferred to the new robot in order to maintain efficiency. This is accomplished in [38] by using Virtual Machine (VM) techniques to freeze the software, transfer it to another robot, and resume operation. Some of the limitations of this approach are outlined in [38]: VMs tend to be highly architecture dependant and may not function in a system with heterogeneous processors, transferring the entire software state might not be necessary in many cases, in teams of heterogeneous robots not every control system is suitable for execution on every robot, and finally, the time taken to transfer the VM state is dominated by the communication time even in high bandwidth systems.

This approach could have potential beyond simply moving a controller from one robot to another, for example in a distributed processing system it could be used to move processing tasks from one processor to another to help balance the load. [50] takes several steps in this direction, but makes a number of decisions that limit the application of their approach. The foundation of the

approach is to use Java VMs and thread migration libraries to cyclically transfer a Coordinating Process (CP) between robots. The CP is transferred to each robot in turn, updates its state information, issues control commands, and prepares for the next transfer. If a failure occurs during transfer or while the CP is resident on the failed robot the system is capable of restoring the CP from its latest valid state. This allows a centralized controller while eliminating the concern for a single point of failure. However, this strategy also comes with some disadvantages that limit its usefulness:

1. The choice of thread migration appears unnecessary since the implementation could be achieved more efficiently by running the thread permanently on each robot and simply passing the state data around the team.
2. Since only a single thread is considered for transfer it takes no advantage of the distributed network. Processors are limited to performing local tasks while the CP is not resident.
3. Since the CP traverses the entire team each cycle, an individual robot spends a significant percentage of the time without access to the CP and control commands.
4. The extremely large number of process transfers are dictated solely by the design of the system rather than any intelligent strategy considering the needs of the system, unnecessarily increasing the time spent on transfer and the risk of failure during transfer.
5. Despite the authors' claims that there is potential for scalability, there are core factors limiting this. Good performance was achieved with a small team on an extremely low latency network: results showed the entire transfer process took ~4.26 ms. A realistic wireless network must anticipate the latency of single messages to be of this order of magnitude or greater, leading to rapid degradation of performance. Furthermore, as the team size increases the number of transfers grows and the percentage of time that the CP spends on each robot shrinks.

2.6 Conclusions

Dynamically formed teams are an important topic that has received little attention. A technique to organize any group of robots into a functioning team could dramatically increase the team's ability to adapt to failure and make efficient use of all available resources. Two topics of research have been identified after studying the less developed areas of this field: 1) distributed processing through agent persistence and propagation, and 2) dynamically reforming semi-centralized control systems.

The concepts of agent persistence and propagation is introduced in [38] as a method to accomplish long term tasks that exceed the operating time of a robot and avoid the loss of data and learning in the event of robot failure. These concepts can be applied equally well to the transference of agents in a distributed processor network to either balance the load or reduce latency between an agent and the robot it is controlling.

Semi-centralized control offers some of the coordination of centralized control by having supervisors that organize the actions of the team. However, such a system is not robust in the event that a centralized component fails or communication channels break. By applying the strategies from dynamically formed teams, knowledge sharing, and distributed processing it may be possible to build a semi-centralized system while avoiding the pit-falls of a single point of failure.

Chapter 3
Control System Design Strategy: Control *ad libitum*

In a time when the field of mobile robotics has seen dramatic growth and technologies have reached the point where many real world applications of robot teams are being realized, perhaps now is an opportunity to step back and consider how these teams are being designed and built. It is often the case that a particular robot or control system is designed with a specific task in mind, and not much consideration is given to how that task is achieved as long as at the end of the day it works. Although this approach has yielded many successes it seems fundamentally limited in terms of its robustness and potential for far reaching application in the real world. The aptly-titled article *1,001 Robot Architectures for 1,001 Robots* [10] highlights this issue and asks the question "Is it really impossible to subject robot architectures and software systems to any objective performance evaluation?" The review of benchmarking and standardization conducted in [51] lists many initiatives, but by and large their focus appears to be on after-the-fact performance analysis rather than strategies to assist developers design better teams. Two exceptions are the Robotics Domain Task Force of the Object Management Group [52] who encourage designs using modular components, and the Joint Architecture for Unmanned systems (JAUS) which follows the five principles of vehicle platform independence, mission isolation, computer hardware independence, technology independence and operator use independence [53]. In an attempt to approach the issue of robot team design from a broader "big picture" perspective, the Control *ad libitum* philosophy is introduced in [54,55], and several tenets are proposed that can help lead to the design of more adaptable, more efficient, and more robust teams.

The original concept of Control *ad libitum* was simply a set of design ideals and goals that intuitively enhance robot teams. This chapter adds more structure to the philosophy and develops methodologies for quantifying and evaluating key aspects of a team at all three stages of the team lifecycle: Design, Development, and Performance. Considering each stage in turn, the major aspects of differentiation were distilled into seven tenets. The seven tenets are intended to be comprehensive, though not exhaustive, and encapsulate the main aspects of existing works such as [52,53] and the architectural evaluation criteria proposed in [56]. Drawing from many different fields, a set of indexes have been developed and adapted to quantify the aspects of Adaptability, Diversity, Modularity, Efficiency, and Persistency. These metrics provide a starting

point for the evaluation, and demonstrate how the right questions can be asked and answered, though of course there are many other potential criteria that may also be useful.

3.1 Team Lifecycle

Ultimately, measuring the performance of a team seems dependant on the specific implementation of hardware and software. However, it is important to take a step back and consider the entire team lifecycle: what choices were made that led to a given implementation, how easy or hard was it to get there, and how do these choices impact future applications? The robot team lifecycle typically progresses through three phases, Fig. 3.1-1; though of course it is possible at a later stage to make revisions to the design or implementation should an issue arise. The cyclical nature of the lifecycle applies to future implementations, which ideally are built on top of existing work rather than starting each new application from scratch.

In the Design phase there are two broad choices to be made: 1) the choice of Control Architecture, and 2) the choice of Control Strategy. There are various definitions of what is meant by control architecture, often changing slightly depending on the scope and the field. One prominent definition can be found in [56]: "Robotic architecture is the discipline devoted to the design of highly specific and individual robots from a collection of common software building blocks." However, a more general definition is better suited to this work, and so the simpler definition from [57] will be used: "An architecture provides a principled way of organizing a control system." This definition highlights the strategy for organizing components of the control system, which is the key differentiator between architectures. A wide variety of architectures are available for robot teams, such as centralized vs. decentralized, deliberative vs. reactive, strictly

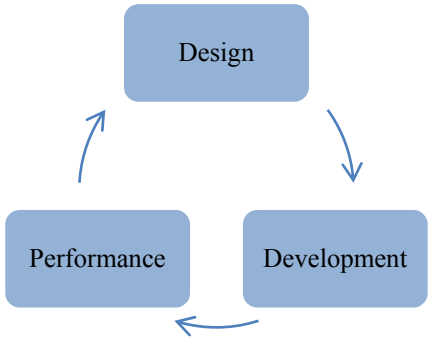


Fig. 3.1-1 Team Lifecycle

hierarchical vs. heterarchical, and of course many combinations and hybrids can be used. When a specific architecture is chosen it provides a framework to construct the control system, but it also introduces limitations on what control strategies can be employed and how they are implemented [57]. For example, a centralized architecture allows for control strategies using central planners but requires information about each agent, and thus a certain degree of communication, while a decentralized architecture can have little to no communication but must rely on control strategies that make decisions based only on local information [58]. The second choice of the Design phase is what control strategies will be used. This includes selection of control laws, algorithms, and techniques, as well as the level of information exchange in the system (communication strategies). The Control Strategy should map out all essential functions and interactions of the control system using the structured provided by the Architecture. A proper Control Strategy should resemble psudo-code for the control system, answering all questions of how the system will work at a conceptual level.

The Development phase begins the Implementation of the control system, specific to an application and installation of the team. This includes selecting the specific type and quantity of hardware for both robots and processors, and a realization of the Control Strategy via functional code/control circuits and specifying control parameters. Fig. 3.1-2 depicts the choices of Control Architecture, Control Strategy, and Implementation as a pyramid, with each level providing structure for the next step but at the same time limiting the available options for later choices.

Once the implementation is complete the Performance phase can begin, and the team can be evaluated using traditional quality of service, efficiency, and reliability metrics. These traditional metrics are the dominant form of evaluation in the field, and are the obvious way to compare or rank different robot teams. However, as was shown above, performance is only the tip of the pyramid and heavily depends on the many choices that lead to a given implementation.

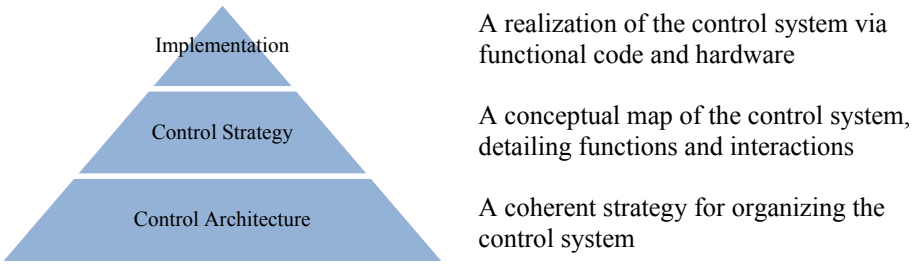


Fig. 3.1-2 Implementation Pyramid

Comparing performance can tell you which team is better in what aspects, but cannot tell you why that is the case and does little to help designers improve their decisions during or after the initial Design and Development phases.

Based on this full lifecycle approach, the breakdown in Table 3.1-1 systematically considers each phase, restating the key element of the phase and listing the fundamental aspects of differentiation and comparison in the form of questions. Each phase is considered from both a functional and a usability perspective; for example, the two sides of an architecture are the capabilities of the architecture in terms of structure and flexibility and the process of applying the architecture from the designer's standpoint. Additionally, these questions form the basis for the seven tenets of Control *ad libitum*, discussed in the following section, and each question has been identified by its corresponding tenet. These 18 questions represent a range of avenues of comparison spanning the entire team lifecycle, Fig. 3.1-3, many of which are often not considered during the design and evaluation of robot teams.

Table 3.1-1 Lifecycle Breakdown

Design Phase – Control Architecture

The central concept of a Control Architecture is the "principled way of organizing" [57]; a set of guidelines, instructions, or rules that tell the designer how elements of the control system should be structured.

1. Is the architecture easy to understand? (Transparency)
    a. Does it encourage implementations that can be properly analyzed?
    b. Does it encourage implementations that are accessible to future designers?
2. Is the architecture simple to use? (Transparency)
    a. Are the tools/guidelines straightforward?
    b. Are the tools/guidelines well defined and comprehensive?
3. Is the architecture flexible enough to be applied to a wide range of strategies/applications? (Versatility)
4. Does the architecture support and encourage modular design? (Modularity)
5. Does the architecture support and encourage robustness and failure recovery? (Persistency)

Design Phase – Control Strategy

A Control Strategy is a selection of control laws and algorithms and a strategy for their interactions.

6. Is the strategy well defined and easy to understand? (Transparency)
7. Is the strategy designed to support diverse components? (Diversity)
8. Is the strategy designed in a modular fashion? (Modularity)
9. Is the strategy designed to support adaptability and learning? (Adaptability)
10. Is the strategy designed to be reusable or retaskable? (Versatility)
11. Is the strategy designed to be robust and recover from failures? (Persistency)

Development Phase – Implementation

A realization of Control Strategy via specific code and hardware.

12. Is the implementation easy to understand? (Transparency)

13. Is the implementation reusable or retaskable? (Versatility)
14. Is the implementation modular? (Modularity)
15. Are the selected components diverse? (Diversity)
16. Was the implementation completed in a timely fashion given the allocated resources? (Efficiency)

Performance Phase

The evaluation of the real-time performance of an implementation.

17. How efficiently does the implementation perform tasks? (Efficiency)
    a. Are tasks performed in a timely manner?
    b. What is the quality of service?
    c. What is the return on investment?
18. How robust is the system? (Persistency)
    a. How susceptible is the system to disturbance?
    b. How susceptible is the system to failure?

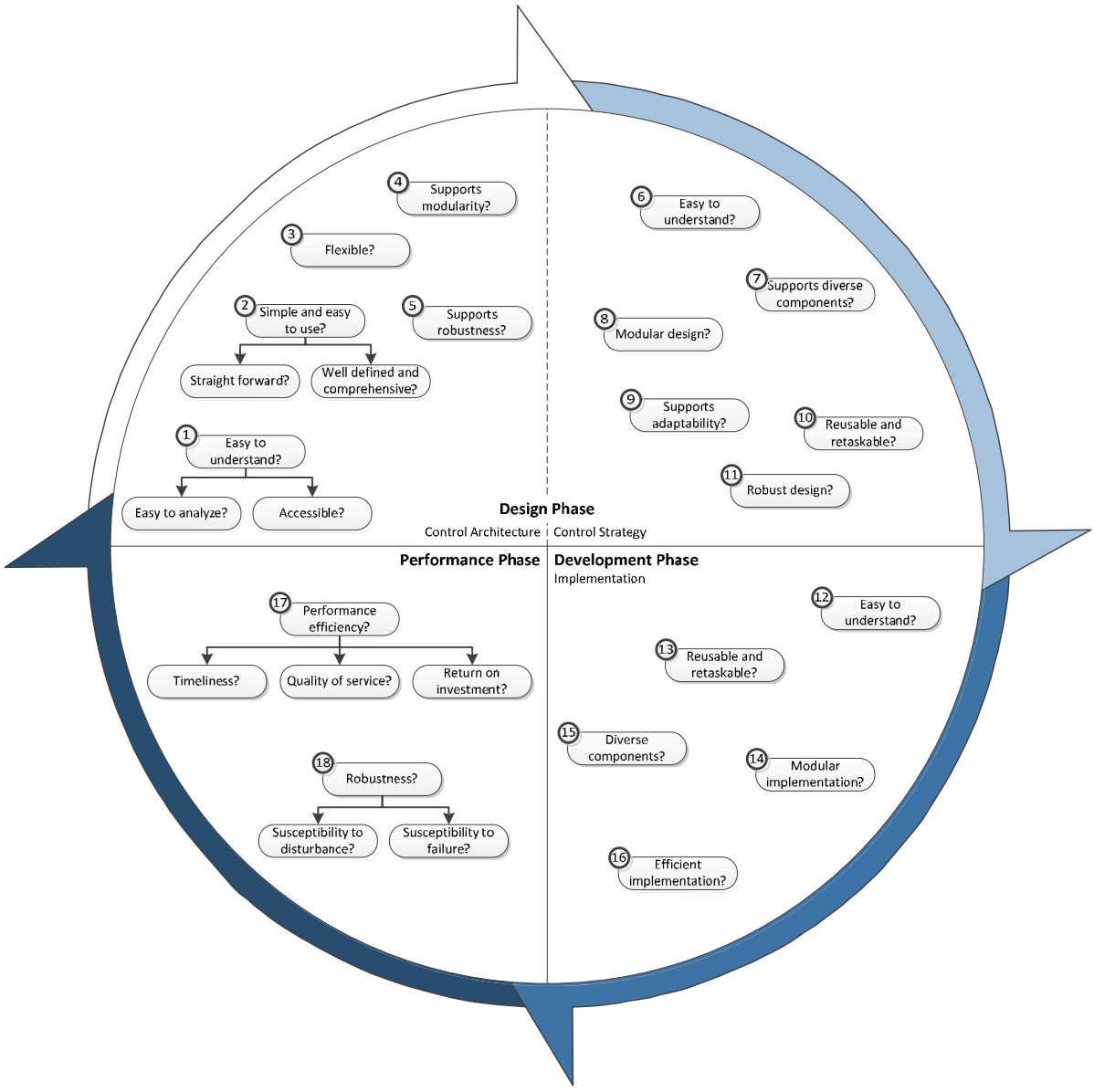


Fig. 3.1-3 Lifecycle Breakdown

3.2 Control *ad libitum*

Control *ad libitum*, literally "control at the performer's discretion," is an approach that suggests that in order to make significant progress in the real-world use of mobile robot teams the control system must be structured such that teams can be formed in real-time from diverse components. From the common modern usage, ad-lib, it can be said that wide spread application of robot teams is only practical if a control system is able to improvise using the currently available resources. In order to design such control systems these goals must be taken into consideration from the very first step, and attention must be paid to how the control system is constructed in addition to how it functions. By considering each phase of the robot team lifecycle seven tenets were identified: Transparency, Versatility, Adaptability, Modularity, Diversity, Persistency, and Efficiency. These tenets are outlined in Table 3.2-1, starting with a summary of each concept and followed by a brief discussion of how it relates to the phases of the design cycle.

Table 3.2-1 Tenets of Control *ad libitum*

*Transparency (**Design**, Development)*

- *Ease of use and simplicity of the design structure/guidelines*
- *Ease of understanding during design and development*

Most reasons why robot architectures and control systems are rarely reused stem from a lack of transparency [10]. Many designers take the stance that their time is better spent building everything from scratch than fighting with existing code that won't compile, does not have the exact features they require, or is simply hard to understand. Quite often they are right, but this problem with how architectures and control systems are designed in results-oriented environments is not irreparable. If more focus is placed on how systems are designed and there is a better understanding of how to make components reusable and retaskable, a significant amount of effort can be saved on future projects. Transparency is most important in terms of the architecture and control strategy of the Design phase, but is also important to the Development phase.

*Versatility (**Design**, Development)*

- *Flexibility of the system, ease of use for different applications*

Versatility reflects the ability of a component to be applied to many uses. Related to the Design phase, some architectures are designed for very specific uses or specialized tasks, while others attempt to be more general or open ended. Similarly, control strategies may have different requirements in terms of specific hardware or algorithms. In the Development phase, code and components of the control system can be made as flexible and independent as possible for reuse in the same system or other applications, drawing on the concept of decoupling from object oriented programming [59]. There is sometimes a tradeoff between versatility and efficiency, and in such cases there should be an understanding of what is being sacrificed on each side before a decision is made.

*Adaptability (**Design**)*

- *Ability to handle changes to the system or the environment without outside intervention*

In real-world applications of mobile robotics the available resources will change not only between different installations, but will also change during operation due to hardware failures and recoveries. In order to be practical for large-scale use the control system must be able to adapt to these changes in real-time without detriment to the

functions of the team. Such functionality can be built in at the Control Strategy stage, and can be supported by features of the architecture.

*Modularity (Design, **Development**)*

- *Structure of interactions and dependencies of components*

Modularity is a key element in the development of any large system. A truly modular system reduces the complexity of the design, reduces the effort required to develop each module, and allows modules to be reused within the control system or in future control systems. Building new functionality on top of previous control systems will be an essential part of realizing wide spread application of mobile robotics. Modularity is most easily measured at the Development phase, though if the Control Strategy is detailed enough then modularity measures can be used at that level as well.

*Diversity (Design, **Development**)*

- *Diversity of components such as hardware, software, or control parameters*
- *Support for diversity*

Diversity of components at the Development phase does not necessarily correlate to improved performance [5], though measures of diversity are still useful to provide insight between implementations. On the other hand, *supporting* diversity at the Control Strategy or Implementation level has no drawbacks and can be beneficial for two reasons. First, procuring hardware resources can be costly, particularly for large scale applications. Thus, it is desirable to be able to reuse hardware from old applications, acquire whatever hardware is currently cost effective, and in the future add or replace hardware without concern for finding exactly the same models[9]. Secondly, heterogeneity within a team can provide benefits for both team effectiveness [8] and the cost vs. capability ratio[7].

*Persistency (Design, **Performance**)*

- *Robustness, the ability to continue operation in the face of failure*
- *Failure recovery, the ability to handle failures without losing functionality or information*

Many applications of mobile robotics require operations in inhospitable environments, and complex hardware has significant failure rates even in the best circumstances. For long lived autonomous tasks it is desirable that the control system recover from such failures without loss of data, maintaining the knowledge and learning acquired up to that point [38]. To be truly robust the control system must be independent of specific hardware resources and flow between resources as their availability dictates. Such features should be built into the system at the Design phase, relying on both the Architecture and the Control Strategy. It is easiest to put these features to the test, however, in the Performance phase when the complete system implementation is available and can be evaluated under realistic failure scenarios.

*Efficiency (Design, Development, **Performance**)*

- *Cost vs. Reward, how much gain is provided for an input of resources*

The issue of efficiency has two facets. The first consideration is that the system is not designed in such a way that requires the duplication of resources or effort. For example, rather than equipping each robot with a high speed processor to extract information from images a number of processors can be shared between the whole team [33]. The second consideration is that the system uses resources appropriately when they are available in order to prevent bottlenecks. For example, [49] utilizes the processors of each robot in a distributed algorithm to compute collision free routing paths when solving the routing on a single processor would have been impossible. Both these scenarios, and almost any other question of efficiency, can be reduced to the problem of cost vs. reward. All that is required is to define what costs are being considered, e.g., time, resources, hardware investment, and what constitutes a reward, e.g., tasks complete, quality of service. Efficiency for robot teams is primarily considered in the Performance phase, though the same cost vs. reward analysis is equally valid throughout the Design and Development phases, particularly for commercial projects.

3.3 Design, Development, and Performance Indexes

Considering the seven tenets of Control *ad libitum* and the questions they were founded on, it is clear that some aspects are difficult to quantify empirically and are best suited for analytical consideration. However, there are also a large number of aspects that can be quantified, given the right tools, which can lead to better standardization and more consistent comparisons. This work develops and adapts several metrics and indexes from various fields, and shows how they can be used to help understand differences in performance and even provide useful insight before the team is assembled or the first line of code is written. The indexes are sorted according to the tenets of Control *ad libitum* and categorized by where they fall in the team lifecycle. Most indexes are considered "preference indexes" because in the majority of cases a higher value in the index is preferable, for example efficiency or robustness. It should also be noted that in the same way that performance is no longer the sole measure for comparison, improved performance is also not the only reason to prefer one choice over another. Indexes for transparency or versatility may not correlate to performance but are beneficial in other ways, such as aiding understanding of the control system or streamlining the design process. In contrast to preference indexes, it is also possible to have "distinction indexes," which simply differentiate between choices without suggesting one is better than another. For example, diversity can be beneficial in some cases and a liability in others, but quantifying diversity can still provide insight into other aspects of the team.

Five indexes related to Adaptability, Diversity, Modularity, Efficiency, and Persistency are developed in Appendix I. Some examples are provided there and the Modularity and Persistency indexes were applied to the final control system developed in Chapter 6; while Efficiency was used as the primary comparator between the different SLAM strategies tested in Chapter 7.2.

Chapter 4
Control System Architecture

4.1 The Host, Avatar, Agent Architecture

Based on the principles of Control *ad libitum*, the following HAA architecture was developed [54]. The system is separated into three components: Hosts, Avatars, and Agents. *Hosts* are the physical processors, where computations are done. Hosts can be either stationary units or mounted on avatars. *Avatars* are the physical robots, which must be the control system's eyes, ears, and hands in the world. *Agents* are the software modules of the control system that run on the hosts. This relationship is depicted in Fig. 4.1-1. The hosts form a distributed processing network and maintain a distributed database (DDB) to share data throughout the network. Software agents are spawned (instantiated) as required by the control system and communicate with each other and the avatars via the network. An agent consists of an internal state and a set of functions for manipulating that state and communicating with other agents. By transferring the state of an agent between hosts it is possible for an agent to propagate throughout the network to: a) balance processing load, task priority, and communication latency, b) conserve power on mobile hosts, c) recover in the event of hardware failure, and d) share learned behaviours with similar agents.

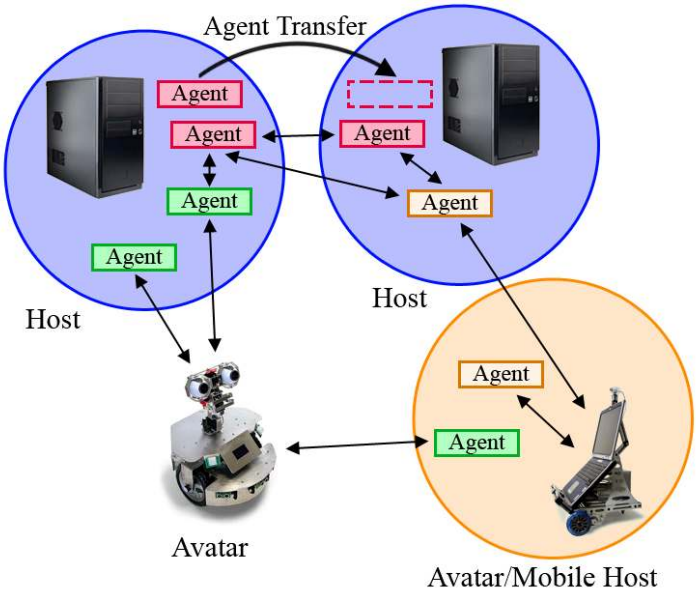


Fig. 4.1-1 HAA Architecture

Working in the HAA architecture requires three components: a) the distributed processing network of hosts, b) a DDB to share information, and c) a strategy or hierarchy for organising agent interactions. Since there are many possible implementations for each of these components, they are discussed in general terms in Section 4.2. A full implementation is then presented in Section 4.3, using provably correct distributed algorithms to build each piece of the architecture.

4.2 HAA Building Blocks

4.2.1 Distributed Processing Network

Communication is the cornerstone of any distributed processing network and dramatically affects the structure of a control system. As shown in [5] communication can improve performance, however, in some cases the necessary hardware can add significantly to the power consumption of a robot [32]. On the other hand, high speed processors required for tasks such as real-time image processing or other complex computations typically consume an order of magnitude more power than a wireless transmitter.

4.2.2 Distributed Database

The DDB is used to share various types of information throughout the network. In following the tenets of efficiency and persistency the DDB is implemented as a highly available service, improving local performance of the database and preventing data loss when failures occur. Highly available services can be implemented using various distributed algorithms, such as the gossip architecture [60] that balances the amount of communication against the required consistency of replicas [61].

4.2.3 Scalable Hierarchical Control

The distributed processing network provides an extremely flexible framework in which to build a control system. When designed in a modular fashion the control system can take advantage of the distributed resources while simultaneously implementing centralized components without sacrificing the robustness of the system. Similarly, elements of both deliberative and reactive approaches can be used together as dictated by the task requirements. One possibility is the Scalable Hierarchical Control approach presented here, which suggests an organizational hierarchy that dynamically scales depending on the current tasks and available resources. As

shown in Fig. 4.3-1, the high-level organization of the team is coordinated by a group of Executive agents, mid-level Supervisor agents perform more focused roles related to a single function or task within the team, and the lowest level of operation is carried out by Worker agents. Four different agents are outlined in Table 4.2-1 to provide examples of the different levels of interaction.

4.3 HAA Implementation

This section introduces the structure and related algorithms used to build a robust and fault tolerant distributed framework using the HAA architecture. The goal of this framework is to create a modular and dynamic system that has provably correct behaviour under common failure scenarios. To this end the basic network structure was defined and a set of distributed algorithms

Table 4.2-1 Agent Examples

1. *Mission Executive (Task Assignment Executive):* The Mission Executive decides what tasks need to be accomplished, when a task should be started, when a task is complete, and what tasks should be arranged in series or parallel to create a branching task tree. Starting from the highest nodes on the task tree, sub-tasks are added and broken down as necessary based on the available resources. Fig. 4.3-2 shows an example task tree for a mission where the team must explore an unknown environment, gather any "collectibles" that are found, and return to the staging area.
2. *Avatar Executive:* The Avatar Executive assigns avatar resources to fulfill the requests of Supervisors. Whenever possible multiple requests are fulfilled simultaneously, allowing the efficiency of the team to scale proportionally with the number of avatars. As the number of avatars grows it becomes impractical to find the optimal solution, but strategies such as that presented in [19] are able to find near-optimal solutions with significantly less work.
3. *Task Supervisor:* Each Task Supervisor is designed to handle a specific task, which may be accomplished by completing the task directly, coordinating a group of Avatar agents, or spawning and coordinating lower level Task Supervisors. For example, in Fig. 4.3-2 the Gather Supervisor spawns a Collect Supervisor for every object. The Collect Supervisor then requests avatar resources from the Avatar Executive and coordinates their actions using Form Team and Push Object agents. This level of abstraction provides significant modularity and conceptual separation between tasks, and new tasks can be programmed without affecting the current control system.
4. *Avatar Agent (Worker):* Every physical avatar has an associated agent that acts as a gateway to the control system. All off-the-shelf robots come with their own specific API and require some customization to function with a control system. Interpretation is necessary between this specific API and the generic internal interface, and the Avatar Agent consolidates these interpretations into a single module. Avatar Agents select what functionality they expose, allowing the Avatar Executive to assign them useful tasks without knowing the details of their operation.

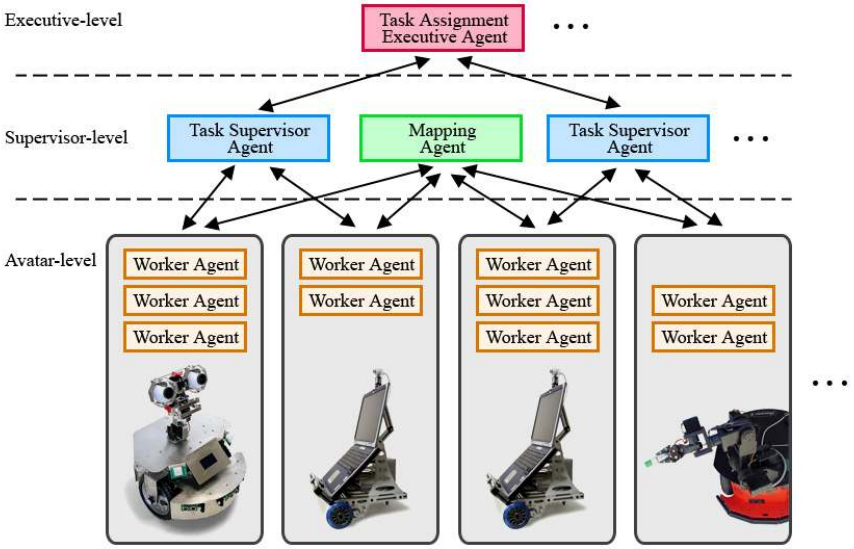


Fig. 4.3-1 Scalable Hierarchical Control

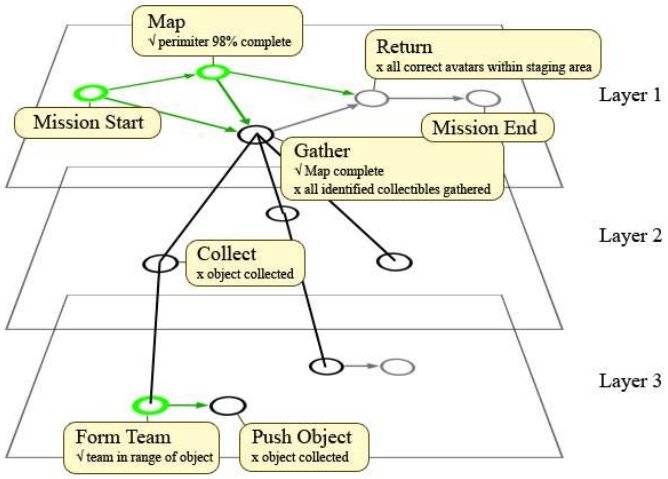


Fig. 4.3-2 Task Tree

were implemented in layers in order to provide the basic functionality of the system. These components are described below to provide an overall picture of the structure and list assumptions and failure conditions for each component, and then the following sections discuss each component and their related proofs in detail. [62] relates these details of the implementation.

4.3.1 Framework

The distributed framework consists of both the physical computer and network hardware and the software running on top. In general terms the physical network consists of processors connected by communication channels. Processors are capable of running software processes, and for this framework processes are divided into two categories: Hosts and Agents. Each processor runs one instance of the Host process, which manages all inter-process communication and the creation and allocation of the Agent processes. This framework is depicted in Fig. 4.3-3. These Hosts are responsible for all the algorithms discussed in the following sections, while the functional details of the Agent processes are unimportant so long as they support the basic agent functionality of creation, destruction, transfer, and recovery.

The Host network relies on a series of distributed algorithms to support its functions, shown in Fig. 4.3-4. These algorithms are listed in Table 4.3-1 and descriptions are provided in general terms that attempt to summarize their purpose, however these cannot be taken as a complete definition of their functionality.

Each algorithm is reliant in various ways on those before it and thus is subject to the assumptions and failure conditions of the preceding algorithms. However, they are also modular in that if another algorithm was found that implemented the same functionality the algorithm could be replaced and the system would then be subject to the assumptions and conditions of the new algorithm. Detailed psudo-code for each algorithm is provided in Appendix II, along with proofs of algorithm correctness.

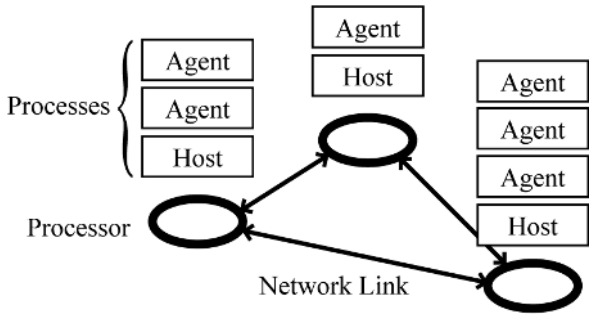


Fig. 4.3-3 HAA Network Structure

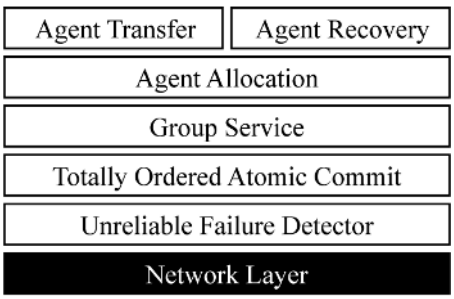


Fig. 4.3-4 HAA Algorithm Hierarchy

Table 4.3-1 Algorithm Descriptions

Algorithm	Description
Network Layer	Provides the basic network functionality of the computer hardware to the algorithms.
Unreliable Failure Detector	Monitors a connection and estimates whether that connection has failed.
Totally Ordered Atomic Commit	Delivers messages to a group while ensuring that a) if a message is delivered to one correct process the same message is delivered to all correct processes, and b) the order of delivered messages is consistent across all processes.
Group Service	Maintains a list of group members.
Agent Allocation	Determines which processes shall run on each processor based on processor usage and communication bandwidth between processes.
Agent Transfer	Facilitates the action of seamlessly halting a process on one processor and resuming it on another processor.
Agent Recovery	Facilitates the action of recovering a failed process on either the same processor or a new processor.

As a result of the current choice of algorithms the following assumptions are made about the network structure:

1. The network is fully connected. That is, every correct processor is capable of communication with every other correct processor.
2. A correct communication channel is reliable and ordered, where reliability is defined by as:

   "validity: any message in the outgoing message buffer is eventually delivered to the incoming message buffer;

   integrity: the message received is identical to one sent, and no messages are delivered twice." [61]

   The ordered condition means that any sent message must be delivered prior to the delivery of any message sent at a later time.

A correct process or channel is defined as a process or channel that does not suffered any failure. The possible classes of failure are defined in Table 4.3-2.

Each class of failure as it relates to this framework/implementation is considered as follows:

Fail-stop: Fail-stops are not considered explicitly as there is no method to directly detect a halted process, instead all fail-stops can be considered crash failures.

Crash: Crashes are allowed and the system must be able to handle up to *f* crashes before taking an incorrect step.

Omission: Omissions are reduced to the case of total channel failure, i.e. all messages after a time *t* are not delivered. The case of individual lost messages is prevented by the assumption of ordered message delivery that is enforced by the Network Layer. When a channel is suspected of failure by the Unreliable Failure Detector the process at the other end is assumed to have crashed and must be handled appropriately.

Send-omission/Receive-omission: Both types of failure are disregarded based on the assumption that the Network Layer will either correctly send/receive a message, return an error, or crash.

Arbitrary (Byzantine): Some types of arbitrary failure, such as message corruption or arbitrary stops are handled by the system, but in general this class of failure is not allowed.

Clock: No assumption of synchrony is used in any of the algorithms, but it is assumed that a processors local clock has bounded drift from real time for the interval of operation. Studies have shown that in practice the rate of drift is on the order of 10^{-6}% for over an interval of time [63].

Performance (Process/Channel): No explicit bounds are made on the interval between process steps or duration of message transmission, however the Unreliable Failure Detector will suspect any channel (and thus process) that is performing below its quality of service specifications. Thus, these classes of failure are handled as crashes.

Table 4.3-2 Classes of Failure (Adapted from [61])

Class of failure	Affects	Description
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or Channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.
Clock	Process	A process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

In summary, send-omission/receive-omission failures, general arbitrary failures, and clock failures are not allowed, while the other types of failure are all handled by the Unreliable Failure Detector as suspected crashes.

4.4 Unreliable Failure Detector

The purpose of a failure detector is to know when a process has failed or becomes unreachable. For the general case of unreliable networks the task of perfect failure detection violates the FLP impossibility [64], and so the notion of unreliable failure detection was introduced [65]. An unreliable failure detector typically provides an estimate of whether a process has failed, "suspected," but is not guaranteed to be accurate at any given time. The algorithm used in this work is heavily based on that found in [66]. This algorithm was selected because of its straight forward implementation and the ability to abstract the tuning parameters of the algorithm into real-world quality of service (QoS) metrics.

In general a failure detector monitors the connection between two processes, the observer q and the observed p. At all times q proposes whether p is trusted or suspected, but of course cannot be considered correct at any given time. Borrowing the notation of [66], the QoS metrics that determine if the proposal of q is useful are broken down into three primary metrics, and four derived metrics:

Primary:

1. Detection time (T_D): Elapsed time between p crashing and being permanently suspected.
2. Mistake recurrence time (T_{MR}): Time between two consecutive false suspicions.
3. Mistake duration (T_M): Elapsed time between making a false suspicion and correcting it.

Derived:

1. Average mistake rate (λ_M): Rate of false suspicions.
2. Query accuracy probability (P_A): Probability that q's proposal is correct at a given time.
3. Good period duration (T_G): Time between proposing trusted and proposing suspected.
4. Forward good period duration (T_{FG}): Related to T_G, the time between any given point in time and the next suspicion.

The relation between the primary metrics and the derived metrics is provided by Theorem 1 below. The notation $Pr(A)$ represents the probability of event A occurring, $E(X)$ is the expected value of random variable X, $V(X)$ is the variance of X, and $E(X^k)$ is the k^{th} moment of X.

Theorem 1 (published in [66] along with a discussion of the proofs and the selection of primary metrics). For any ergodic failure detector, the following results hold: 1) $T_G = T_{MR} - T_M$. 2) If $0 < E(T_{MR}) < \infty$ then $\lambda_M = 1/E(T_{MR})$ and $P_A = E(T_G)/E(T_{MR})$. 3) If $0 < E(T_{MR}) < \infty$ and $E(T_G) = 0$, then T_{FG} is always 0. If $0 < E(T_{MR}) < \infty$ and $E(T_G) \neq 0$, then 3a) for all $x \in [0, \infty)$, $\Pr(T_{FG} \leq x) = \int_0^x \Pr(T_G > y)\, dy / E(T_G)$, 3b) $E\left(T_{FG}^k\right) = E(T_G^{k+1})/[(k+1)E(T_G)]$. In particular, 3c) $E(T_{FG}) = \left[1 + \frac{V(T_G)}{E(T_G)^2}\right] E(T_G)/2$.

These metrics can be used to compare the performance of failure detectors, but are equally useful to specify performance bounds on a failure detector. The authors of [66] present several variations on a failure detection algorithm using bounds on T_D, T_{MR}, and T_M to determine the tuning parameters, and provide a proof that the algorithm yields the optimal query accuracy probability P_A for failure detectors with the same rate of heartbeat messages and upper bound on detection time. The basic algorithm is summarized below and then the specific implementation used for this work is explained in more detail; this implementation is heavily based on the algorithm of [66] and the text will note where augmentations have been made.

4.4.1 Failure Assumptions

No send-omission, receive-omission, arbitrary, or clock failures are allowed. Omissions with a message loss probability of p_L and message delays D in a distribution with finite $E(D)$ and $V(D)$ are allowed. Crashes/fail-stops are allowed for p but the algorithm obviously requires q to be alive.

4.4.2 Algorithm Overview

The observed process p periodically sends numbered heartbeat messages, m_i, to the observer q. Observer q knows a time, τ_i, before which it is expecting to receive m_i. If m_i arrives before τ_i then p remains trusted, if time τ_i occurs before m_i arrives then q suspects p. If m_i was delayed

and arrives before time τ_{i+1} then q is allowed to trust p again, however if m_i is delayed beyond τ_{i+1} then q must wait for m_{i+1} or a later message before it will trust p again.

Algorithm 1, found in Appendix II, implements this algorithm in a system with a) unsynchronized clocks, and message behaviour (in terms of message loss p_L and message delay D) that is b) unknown and c) changes over time.

This algorithm is based on [66]'s NFD-U algorithm, with six significant changes that allow the tuning parameters to be updated dynamically based on current estimations of network performance:

1. *INITIAL_PERIOD set by the observer:* In ll. 2-3 the initial period for the heartbeat messages is set to an initial value, selected to be smaller than the smallest η based on the possible QoS settings and worst case network performance.

2. *Including period and ${}^{p}\tau_l$ in the heartbeat message:* The current *period* and local time, ${}^{p}\tau_l$, are included in the heartbeat message to assist in the calculation of EA_{l+1}.

3. *Initialization of the FD parameters:* Because the performance of the network is not initially known the calculation of initial FD parameters must be assume a "worst case" scenario.

4. *Calculation of EA_{l+1}:* Since the clocks are unsynchronized and the period can change the calculation of EA_{l+1} must be different than in [66]. The inclusion of *period* and ${}^{p}\tau_l$ in each heartbeat message allows for two things: 1) the direct calculation of ${}^{p}\tau_{l+1}$, and 2) the estimation of the offset message delay, O_l. The delay O_l consists of the offset between the between the local clocks of p and q, which is assumed to be constant, and the actual message delay D_l. This information allows the expected time of arrival of m_i to be estimated for the local clock:

$$EA_{l+1} \approx {}^{p}\tau_l + period + \frac{1}{n}\sum_{i=1}^{n} O_i \qquad (4.4\text{-}1)$$

5. *Re-evaluation of the FD parameters:* With the understanding that $V(O) = V(D)$, since O_i is D_i plus a constant, the FD parameters η and α can be calculated using the same procedure given in [66]. This is done every time N heartbeat messages are received in order to adapt to changing network performance. In long lived applications with significant network dynamics it may be desirable to store only the M most recent offset message delays for use in the calculation of $V(O)$ and EA_{l+1} above.

6. *Introduction of the setPeriod message:* After the FD parameters have been re-evaluated p needs to be notified of the new period, which is accomplished by a simple setPeriod message. The new period is adopted by p after its next heartbeat is sent so that it does not interfere with q's expectation of the arrival of m_{l+1}.

The tuning of this algorithm is done through the selection of the QoS requirements T_D^u, T_{MR}^L, and T_M^U, and to a lesser extent to selection of constants INITIAL_PERIOD and *N*. Given the three QoS requirements the algorithm guarantees that:

$$T_D \leq T_D^u + E(D), E(T_{MR}) \geq T_{MR}^L, E(T_M) \leq T_M^U \quad (4.4\text{-}2)$$

The one caveat is that the upper bound on detection time T_D is T_D^u plus the average message delay $E(D)$, which is unknown. An estimation of message delay would require an increase in complexity of the algorithm and additional messages, due to the assumption of unsynchronized clocks. [66] argues that this does not impact the algorithm's viability, since once the upper bound on detection time is lower than the message delay $E(D)$ any failure detector begins to make too many mistakes to be useful, and so specifying an additional buffer on top of the message delay (in this case T_D^u) is effectively a requirement.

In order to understand the cost of running the algorithm the heartbeat frequency is the primary concern, i.e. the messages/minute cost of the algorithm. The QoS requirements used for this setup ($T_D^u = 10\ s,\ T_{MR}^L = 1\ week,\ T_M^U = 30\ s$) cost roughly 12 messages/minute and has an expected forward good period duration $E(T_{FG}) \geq 3.5\ days$.

4.5 Totally Ordered Atomic Commit

In Atomic Commit (AC), messages must be sent in such a way that all correct processes are guaranteed to deliver the message, and all processes must be given the option of triggering a unilateral abort even if they successfully receive the transaction message (for example if the transaction conflicts with the current state of the process/database). For this implementation the non-blocking AC algorithm of [67] has been used as a base to provide totally ordered atomic message delivery. The [67] algorithm was selected because it provides non-blocking AC for asynchronous systems using an unreliable failure detector such as that described above. Correct processes are able to agree on an outcome despite false suspicions, and when no failures occur the

algorithm performs similarly to a three phase commit protocol [67]. The AC problem is defined in [67] by the four conditions:

AC-Uniform-Agreement: If two participants decide, they decide on the same outcome.

AC-Uniform-Validity: The outcome is *commit* only if all participants have voted *yes*.

AC-Termination: Every correct participant eventually decides.

AC-Non-Triviality: If all participants vote *yes*, and no participant has ever been suspected, the outcome must be *commit*.

One important property the algorithm is missing is a guaranty of ordered delivery, which is required for some uses in this system. In order to provide this functionality an additional condition, AC-Total-Order, has been defined.

AC-Total-Order: If one correct process commits transaction *m* before committing transaction *m'* then all correct processes participating in both transactions must commit *m* before committing *m'*.

Proofs for these properties are developed in Appendix II.

4.5.1 Failure Assumptions

No send-omission, receive-omission, arbitrary, or clock failures are allowed. Omissions are not strictly allowed, but in a system with potential for message loss the channel can be made reliable by retransmitting lost or corrupted messages [67], such as is done for the TCP/IP protocol. Crashes/fail-stops are allowed for up to $f < n/2$ participants.

4.5.2 Algorithm Overview

The algorithm starts when process $p*$ proposes a transaction to a number of participants, p_1 to p_n, one of which must be $p*$. Upon receiving the proposed transaction there is a voting phase where each participant decides locally whether it is willing to commit the transaction, and then joins successive rounds of the consensus phase until a decision is reached. During the voting phase each process has the opportunity to evaluate the transaction and votes yes or no on whether to proceed with the commit. The consensus phase uses a rotating coordinator to decide on the outcome, and can only decide to commit if every participant has voted yes.

The algorithm found in [67] provided the basic structure for Algorithm 2, however significant modifications were made in order to satisfy AC-Total-Order. These modifications centre on the new $activeTransactions_p$ and $decidedTransactions_p$ lists that each process maintains, which record transactions that are known but not yet decided and transactions that are decided but not yet closed, respectively. Closing a transaction means that no more messages relevant to that transaction will arrive, and therefore the transaction can be cleared from memory. The requirement of total order is complicated by the fact that the sets of participants for two messages might differ, yet still require transaction ordering for the overlapping participants, and so decisions about one transaction may not be readily available to participants of the second transaction. To work around this issue all participants are required to propose to an insertion order prior to agreeing to commit, and the highest proposal will be accepted by all correct participants. Insertion order proposal and acceptance follows two rules: 1) when a participant first becomes aware of a transaction it must propose an order greater than that of highest open transaction (transactions that have not been closed, as defined above), and 2) a new order proposal is accepted only if it is higher than the current order of that transaction. This strategy takes advantage of the fact that each participant must provide their estimate before the pre-commit point can be reached. Thus, by ensuring that no participant agrees to an order that conflicts with their committed transactions the algorithm guarantees that AC-Total-Order is satisfied.

The algorithm is described block by block with reference to Algorithm 2.

To maintain global order each participant maintains local $activeTransactions_p$ and $decidedTransactions_p$ lists (ll. 1 and 2). As soon as a participant learns of a transaction, either from the initiator or another participant, it is assigned an order and added to $activeTransactions_p$. Once the transaction has been decided it is removed from $activeTransactions_p$ and added to $decidedTransactions_p$. The transaction remains in $decidedTransactions_p$ until all participants have acknowledged the decision, at which point the transaction can safely be closed.

To propose a transaction $p*$ calls procedure startTransaction (l. 3) and sends out a transaction message containing a globally unique transaction ID, the transaction details, and a list of participants. At this time $p*$ also proposes an order higher than any transaction $p*$ is currently aware of (l. 4).

Upon receiving the transaction message (l. 6), each participant first checks their $decidedTransactions_p$ list to ensure that this transaction is still relevant, and then calls the atomicCommit procedure. The initialization (ll. 16-32) corresponds to the voting phase, where each participant has the option to declare a unilateral abort. The participant ensures that an acceptable order is assigned (ll. 21-25), and if necessary notifies all other participants of the new order. In order to reduce the number of order changes the transaction IDs are used as a tiebreaker in the case of transactions with the same order. It is also worth noting that if there is a message priority associated with each transaction, priority can be used as the primary tiebreaker, and transaction ID as a secondary tiebreaker. If the participant votes to abort the transaction (l. 27) they notify all participants of the decision (l. 29) and proceed to Task 1, otherwise they proceed to the consensus phase by starting Task 1 and Task 2 concurrently.

Task 1 (ll. 33-42) waits until the outcome is received and rebroadcasts that decision. In the original algorithm Task 1 can then immediately act on the decision, however to ensure total order additional checks must be made (l. 37). If the decision is abort order does not matter and the participant can proceed, otherwise they must wait until the transaction has the lowest order of all active transactions (using transaction IDs to break ties) before proceeding. Once these conditions are met the participant acts on the decision and moves the transaction from $activeTransactions_p$ to $decidedTransactions_p$. A final gate must be passed before the transaction can be cleaned up and removed from memory (l. 41). This check ensures that all participants have acknowledged the decision and no further messages will be received on the subject.

Task 2 (ll. 43-71) performs successive rounds of consensus and is fundamentally the same as in the algorithm of [67], except for the inclusion of transaction order in each message. Each round of Task 2 can be considered as four steps, two that are taken by all participants (steps P1 and P2) and two that are taken only by the coordinator for that round (steps C1 and C2). Overviews of P1, C1, P2, and C2 are provided here [67]:

Step P1 (l. 47): The participant's current estimate is sent to the coordinator of the round.

Step C1 (ll. 48-56): The coordinator waits until either a) n estimates are received from the participants or b) at least $n - f$ estimates are received and the remaining participants are suspected by the failure detector. When these conditions are met the coordinator revises its estimate and

sends that as a proposal to all participants. Note that the estimate can only become *pre-commit* if at least one coordinator has received estimates for all participants (i.e. no participant voted no).

Step P2 (ll. 57-63): Each participant waits until either a) it receives a proposal from the current coordinator or b) the coordinator is suspected. If a proposal was received the participant sends an acknowledgement to the coordinator, otherwise a "negative acknowledgement" is sent to the coordinator.

Step C2 (ll. 64-71): The coordinator waits until it has received $n - f$ acks or nacks from participants. If $n - f$ acks were received then the coordinator is allowed to decide the outcome and sends the current estimate as the decision.

All participants monitor incoming messages (ll. 72-84) to a) learn of new transactions, and b) update the order of active transactions. If a participant receives a message regarding a transaction they are unaware of the transaction is added to $activeTransactions_p$ as a placeholder (l. 75). For both placeholder and active transactions the new order is compared to the current order (l. 76) and new order is accepted if it is higher (l. 77). For active transactions (l. 78) the new order is broadcast to all participants and Task 2 is reset (ll. 79-83).

The complexity of the system makes the true performance difficult to examine analytically, and so experimentation was conducted to provide some insight. These results can be found in Chapter 7.

4.6 Host Membership Service

Maintaining a list of active hosts is a requirement for several aspects of this system. This list must be known to and agreed on by every host in order to ensure consistency, and must be accurate and available even if some hosts fail. To accomplish this, a group membership service is used, where each host is a potential member of the group and can only perform their duties while they are accepted members. The basic features of a group membership service are: a) providing an interface for adding or removing members from the group, b) monitoring members for suspected failures, c) notifying members when membership changes occur, and d) performing group address expansion [61]. The implementation of such a service can vary significantly depending on the specific requirements of the application and the available tools. For this system

an algorithm leveraging OAC transactions and an UFD was developed. Using OACs ensures that consistency is maintained, which is one of the primary challenges of a membership service.

The developed algorithm has the following features:

1. A globally consistent list of group members.
2. A method to ensure new that members are consistent with the global state.
3. A single acknowledged leader at any time.
4. A method for processes to apply to the group.
5. A method for members to leave the group.
6. A method to remove suspected members from the group.

Selecting a leader is accomplished by maintaining a ranked list of members, where rank decreases with the order in which they were added to the group and the leader is the highest ranked current member. The algorithm also makes use of the notion of "core" process, in the form of an ordered list of processes with globally known addresses. The list is used during group formation and to allow potential applicants to contact the group. Global state is defined as any data that must be consistent across all members, for example this system considers the DDB to be part of the global state. This membership service algorithm is defined by the following seven conditions:

HM-Formation: A correct group will eventually be formed, where a correct group is defined as having at least one correct core member, and no incorrect group is ever formed.

HM-Termination: All messages related to an individual event (join, suspicion/remove, leave) eventually subside.

HM-Non-Triviality-Join: A correct applicant is eventually added to the group by all correct members.

HM-Non-Triviality-Leave: When a member asks to leave the group they are eventually removed by all correct members.

HM-Non-Triviality-Remove: When a member crashes they are eventually removed by all correct members.

HM-Weak-Validity: When an attempt is made to remove a set of members M, M will not be removed unless every member $\notin$ M also suspects every member $\in$ M.

HM-Agreement: Every correct member of the group commits the same series of group add or remove transactions.

Proofs of the seven conditions can be found in Appendix II.

4.6.1 Failure Assumptions

Similar to the OAC failure assumptions, only crashes/fail-stops are allowed. Specifically, up to $c_f < c/2$ core processes may fail, while any number of other processes may fail. Additionally, there is a condition that no correct process is erroneously removed from the group and no process is erroneously excluded during the formation of the group.

In the case of erroneous removal, there is a conflict in that a) if a process fails it must be removed before the group can continue to function, and b) since the failure detector is unreliable it is possible that a process will be falsely accused. In order to complete a) we must rely on the output of the UFD, however, because of b) it is possible for a process to be erroneously removed from the group. The probability of this occurrence is reduced by the fact that the algorithm requires that all members agree on a suspicion before a process can be removed. Therefore, given an appropriately tuned UFD and a group of modest size, the probability of an erroneous removal is extremely low. The probability that member *m* will be erroneously removed from a group with *n* members during a time period of duration *t*, defined as P_R, is approximately:

$$P_R \cong \Pr(T_{FG} \leq t) \cdot (1 - P_A)^{n-2} \tag{4.6-1}$$

From [68]

$$\Pr(T_{FG} \leq t) = \frac{E(\min(T_G, t))}{E(T_G)} \leq \frac{t}{E(T_G)} \tag{4.6-2}$$

Note that for $t \ll T_G$

$$\Pr(T_{FG} \leq t) \approx \frac{t}{E(T_G)} \tag{4.6-3}$$

And so

$$P_R \cong \frac{t}{E(T_G)}(1 - P_A)^{n-2} \tag{4.6-4}$$

This probability is a first order estimate, since P_R only considers a) the possibility of a single false accusation in time period *t* while disregarding the cases of multiple false accusations of the same member, and b) a single erroneous removal while disregarding the cases of multiple simultaneous false accusations.

For a group of n correct processes, the probability that are no erroneous removals, P_{NER}, can be estimated as

$$P_{NER} \cong (1 - P_R)^{n-1} \tag{4.6-5}$$

$$P_{NER} \cong 1 - (n-1)P_R, \text{for } P_R \ll 1 \tag{4.6-6}$$

Because P_R decreases as n increases, P_{NER} increases rapidly with n. Thus, the parameters of the UFD and the minimum group size must be set such that the probability of erroneous removals is acceptably low. For example, Table 4.6-1 shows P_R and P_{NER} for several group sizes with the tuning parameters used in this setup.

The case of erroneous exclusion during group formation is only relevant if the fallback formation is used, i.e., a group has not been formed by FORMATION_TIMEOUT. In this case the first c_H, $c_f < c_H \leq c$, processes use OACs to agree on the initial member list. The probability of a correct process being erroneously excluded can be estimated as

$$\begin{aligned} P_{EF} \cong \Pr(T_{FG} \leq t) \\ \cdot \big[(1-P_A)^{c_H-2} + \Pr(1 \text{ crash after } T)\,(1-P_A)^{c_H-3} \\ + \Pr(2 \text{ crashes after } T)\,(1-P_A)^{c_H-4} \\ + \ldots + \Pr\big(c_f \text{ crashes after } T\big)(1-P_A)^{c_H-1-c_f}\big] \end{aligned} \tag{4.6-7}$$

Where t is the time from FORMATION_TIMEOUT to the formation of a group, and $T = t +$ FORMATION_TIMEOUT. As with the calculation of P_R, this is only a first order estimation.

4.6.2 Algorithm Overview

The basic form of the algorithm can be understood in three parts: a) formation, b) update requests, and c) commitment of updates. When group formation is triggered (either by broadcast or some other method of loose synchronization), each core process calls the groupJoin() procedure. The first core process immediately forms a provisional group with itself as the only member, and once sufficient core processes have applied will form a correct group. In the case of the first core process failing before a group is formed, the remaining core processes will wait until

Table 4.6-1 Probability of Erroneous Removal

$(T_D^u = 10\ s, T_{MR}^L = 1\ week, T_M^U = 30\ s, t = 10\ hours)$

Group size	P_R	P_{NER}	1 - P_{NER}
2	5.95E-02	0.940473	5.95E-02
3	2.95E-06	0.999994	5.90E-06
4	1.46E-10	1	4.39E-10
5	7.27E-15	1	2.89E-14

FORMATION_TIMEOUT has elapsed and eventually form a group using OAC(*formationFallback*, ...) transactions. Update requests can contain join, leave, and remove requests. Join and leave requests are voluntary and will eventually be granted if the process is correct. A joining process must meet the conditions of being connected to every member, and becoming consistent with the global state, which is accomplished through a sponsor system. Remove requests are triggered when a leader suspects another member of failing. If all processes agree with the suspicion the remove request will eventually be granted. Updates are committed via the leader calling the updateMembership() procedure, and may require several attempts before a consensus is reached and the updates are applied. Consistency is ensured because all membership changes are done through OAC transactions, meaning that every member makes an identical series of updates. When committing an update a two phase lock is required to ensure that join and leave updates occur smoothly. Remove updates require consensus, but do not require a lock. A successful update follows these steps:

1. The leader assembles lists of members to join, leave, and remove in this update.
2. An OAC(*remove*, ...) transaction is committed, removing the agreed upon members. If there are join or leave updates as well and a lock is required, this transaction also incorporates a lock request.
3. The leader waits until the lock has been acknowledged by every member, ensuring a lock has been achieved.
4. An OAC(*membership*, ...) transaction is committed, updating the list of members based on the join and leave lists. This transaction also releases the lock.

The algorithm is discussed below according to Algorithm 3.

Each process maintains a set of variables related to the membership service (ll. 1-13). In summary, *coreProcesses* is the list of core processes, $joinList_p$ is the list of applicants who have sent p a join request, $memberList_p$ is the ordered list of current members, $removeList_p$ is the list of members p suspects, $leaveList_p$ is the list of members who have sent p a leave request. The $locked_p$ variable is used both as a flag to indicate whether the global state is locked and to store the current key to the lock. The $updatingMembers_p$ flag is used to prevent simultaneous update attempts from the same process. The $connectionsTo_p$ and $connectionsFrom_p$ lists are used prior to joining to track what connections p has established. The $sponsor_p$ and $sponsee_p$ lists are used in

the sponsor process, and the $groupCorrect_p$ flag and $coreMembers_p$ list are used in the formation process.

To leave the group a member calls the groupLeave() procedure (l. 14). This sends a leave request to each member, however, the member must wait until it has been removed from the member list (l. 16) before it stops participating in group activities to ensure a clean exit. Upon receiving a leave request (l. 18) each member adds the process to their leave list, but only the leader attempts to act on the request.

Remove requests are generating upon suspecting a process (l. 22). This action occurs on all members, though only the leader attempts to act on these requests. Conversely, when a process becomes trusted (l. 26) the remove request is rescinded; additionally, if the process is an applicant updateMembership() is called in case this changes the status of their application.

To join the group a member calls the groupJoin() procedure (l. 35). This procedure has three sections, the first sending join requests to each core process (ll. 36-39) while the other two relate group formation. The second section (ll. 41-43) is called only by the first core process, who immediately forms a provisional group of one by committing an OAC(*membership*,...) transaction. The third section (ll. 44-49) is called by the remaining core processes as a fallback if the first core process fails to form a group.

Upon receiving an apply request (l. 50) from *a*, the process *p* accepts *a* into their join list. *p* also introduces themselves to *a*, and introduces *a* to every member and other applicant. Finally, if *p* is the undisputed leader then *p* becomes a sponsor for *a* and ensures that *a* is brought up-to-date on the global state.

Upon receiving an introduction to *a* (l. 62), the process *p* performs the following actions: a) if the introduction is from *a* itself, *p* inserts *a* in $connectionsFrom_p$, b) if *p* has no connection to *a*, *p* opens *a* connection to *a*, inserts *a* in $connectionsTo_p$, and introduces themselves to *a*, and c) if *p* is an applicant themselves and has a sponsor, *p* sends a list of their current two-way connections to their sponsor.

Relating to the sponsor process, upon receiving a sponsor message from *q* (l. 71) *p* accepts *q* as their sponsor and sends them a list of their current two-way connections, and upon receiving a

connections message from *q* (l. 74) *p* calls updateMembership() in case the status of *q*'s application has changed.

Any time an event occurs that might trigger a membership change the updateMembership() procedure (l. 76) is called. The process *p* first makes sure *p* does not have an update already in progress, and that *p* is not aware of any other attempted membership updates (l. 77). *p* then decides if they are the leader or could become the leader based on their $removeList_p$ (l. 79). If so, *p* assembles a *potentialList* of applicants who met the join criteria (ll. 83-87), then from that list decides on an *acceptList* from potentials who are mutually connected (ll. 88-91). If a correct group has not yet formed, a check is done to ensure that accepting the new members would guarantee a correct group (l. 92), otherwise no members are accepted. Finally, if there are any updates to make (l. 94), *p* attempts to commit the updates:

(ll. 98-101) Determine if a lock is required, i.e. there are join or leave updates.

(l. 102) Attempt a OAC(*remove*, ...) transaction.

(ll. 103-109) a) If no lock was required, wait until OAC(*remove*, ...) is decided, then repeat updateMembership() to ensure no updates are still outstanding.

b) If a lock was required, wait until OAC(*remove*, ...) is aborted or [(*p* has either received a lock acknowledgement from all *q* or *p* suspects *q*) and all active OACs have been decided]. Here "all active OACs" is the set of OAC known about at the time OAC(*remove*, ...) is committed, and must be waited on in case one of them is a OAC(*membership*,...) transaction from the previous leader. At this point the lock has succeeded and *p* has received lock acknowledgements from all *q* or either the remove transaction was aborted or the lock has failed. In the first case, the OAC(*membership*, ...) transaction can be attempted, which upon either succeeding or failing will unlock the global state. In the second case updateMembership() will be repeated to retry the updates.

Upon receiving an OAC(*remove*, ...) transaction (l. 110), the process *p* decides whether or not to allow the transaction to proceed. This mechanism is used to prevent transactions where the members to be removed are not agreed upon. Upon committing an OAC(*remove*, ...) transaction (l. 117), *p* removes the agreed upon members, updates $removeList_p$ and $leaveList_p$, accepts responsibility as sponsor if it is the undisputed leader (ll. 121-126), and locks if necessary.

Upon committing an OAC(*membership*, ...) transaction (l. 136):

(ll. 137-139) If *p* was the leader when this transaction was sent, *p* stops sponsoring any applicants who successfully joined in this update.

(ll. 140-143) Update $joinList_p$, $memberList_p$, $removeList_p$, and $leaveList_p$.

(ll. 144-145) Release the global state lock and propose all held changes.

(ll. 146-149) Check if the new members make this a correct group.

(l. 151) Repeat updateMembership() to process any outstanding updates.

Upon aborting an OAC(*membership*, ...) transaction (l. 152), *p* first checks to ensure the key matches the current lock, then unlocks the global state and repeats updateMembership() to process any outstanding updates.

Upon receiving an OAC(*formationFallback*, ...) transaction (l. 158), *p* decides whether or not to allow the transaction to proceed. *p* will only vote yes to the transaction if it is not already part of a group and the proposed list of members exactly matches *p*'s own proposal for a group. Upon committing an OAC(*formationFallback*, ...) transaction (l. 166), *p* once again ensures that they are not already part of a group to prevent accepting multiple formation messages. If the group is accepted, $joinList_p$, $memberList_p$, $removeList_p$, and $leaveList_p$ are updated. If *p* is the undisputed leader, *p* accepts responsibility as sponsor for all applicants (ll. 172-177). The updateMembership() procedure is then called to process outstanding updates.

4.7 Agent Allocation

A strategy for agent allocation is a fundamental requirement in the HAA architecture, and shares many similarities to the process allocation problem in distributed computing. Since each agent is free to operate on any host, they can be distributed in order to optimize various attributes. The most obvious angle is load balancing to evenly distribute work across hosts, however, other factors such as communication, stability, and priority could also be taken into account. In general, computing the true optimum for process allocation is computationally complex and infeasible in real-time. Therefore many near-optimal algorithms have been developed. The allocation strategy presented here is based on the Consensus-Based Bundle Algorithm (CBBA) [69], which was originally designed as a decentralized task allocation algorithm. With the minor modifications described below, the CBBA approach can be changed from a time based allocation of tasks to a "CPU usage" based allocation of processes. Since the foundation of CBBA remains

unchanged, the desirable properties of the algorithm are maintained, specifically: distributed scalable asynchronous allocation, guaranteed convergence (given an appropriate reward function), and low message cost. Additionally, the synchronous version of CBBA guarantees at least 50% optimality, and both synchronous and asynchronous versions demonstrated roughly 90% optimality during Monte Carlo experimentation [69].

The developed algorithm provides allocation of agents based on four factors. First is process cost, a unitless value estimated by the average CPU usage when running on a baseline processor. The second factor is specific hardware requirements, since in some cases agents require direct access to specialized hardware not available at every host. Third is the notion of "affinity" between agents. Some agents communicate more frequently and in greater volume than others, in many cases to the same small group of agents. In these cases network load can be significantly reduced by allocating the entire group to a single host. Affinity can be estimated based on the predicted behaviour of agents, or measured in real-time during operation. The fourth factor is transfer penalty, which represents the cost associated with transferring an agent from one host to another. This cost can vary significantly depending on the amount of state data that must be transferred.

For this implementation, process costs and agent affinities are monitored in real-time, so that the system can adjust to changing performance and learn which agents communicate with each other. The allocation problem is specified as a set of agents and a set of hosts to which the agents are allocated. Each agent a_i has properties:

c_i : Estimated processing cost (normalized by the processing capacity of the host)

Q_i : Set of hardware requirements $\{q_{ia}, q_{ib}, ...\}$

A_i : Set of affinities with other agents $\{\alpha_{ij}, \alpha_{ik}, ...\}$

t_i : Transfer penalty [0,1]

Each host h has the properties:

K_h : Processor capacity

H_h : Set of available hardware $\{h_{ha}, h_{hb}, ...\}$

L_h : List of current local agents $\{a_i, a_j, ...\}$

The goal of the algorithm is to create a bundle of agents, b_h, for each host such that every agent is allocated once and only once and $\sum R(b_h)$ is maximized, where $R(b_h)$ is the reward for a bundle. Reward calculation is based on the order of agents within the bundle and the capacity of the host.

$$R(b_h) = \sum_{\forall i \epsilon b_h} \begin{cases} r(a_i), & \sum_{j=0}^{i} c_j < K_h \\ s(a_i), & \text{otherwise} \end{cases} \tag{4.7-1}$$

$$r(a_i) = (c_i + \sum_{\forall \alpha_{ij} \in A_i \text{ S.T.} a_j \in b_h} \alpha_{ij}) \cdot \begin{cases} 1, & a_i \epsilon L_h \\ (1 - t_i), & \text{otherwise} \end{cases} \tag{4.7-2}$$

$$s(a_i) = \frac{K_h - \sum_{j=0}^{i} c_j}{K_h} \tag{4.7-3}$$

In other words, reward $r(a_i)$ is given for agents that are assigned within the capacity of the host with bonuses for assigning agents with shared affinities to the same host, while agents that exceed the capacity of the host are given a penalty $s(a_i)$.

4.7.1 Failure Assumptions

No send-omission, receive-omission, arbitrary, or clock failures are allowed. Omissions are not strictly allowed, but as with the OAC algorithm, can be handled by the network layer if they occur. Crashes are handled through the group membership service, and require the algorithm to abort the current run and begin again.

4.7.2 Algorithm Overview

The allocation algorithm works in three stages: initiation, multiple asynchronous rounds of bundle building and conflict resolution, and final consensus. Initiation occurs when either an agent is added or removed or a host is added or removed, although it can also be triggered periodically in order to adjust to changing system usage. Initiation is carried out by the group leader, who defines the properties to be used during allocation, in terms of agents, costs, and affinities, and broadcasts them to begin the algorithm. Each host begins a series of asynchronous rounds where they greedily build their local bundle by bidding on agents, then processes all updates from other hosts, resolves conflicting bids, and broadcasts their latest results. These rounds continue until the group leader determines that consensus has been reached, at which point the leader broadcasts the result and the allocation is complete. Strategies for initiation and

finalization are not defined in [69], but the bundle building and conflict resolution rounds for this algorithm are fundamentally the same as CBBA. Only a minor conceptual change is required in order to apply CBBA to process allocation. Instead of avatars (agents in the terminology of [69]) creating bundles of tasks in the time domain, hosts are creating bundles of processes in the processor usage domain. This transformation is possible because in the same way tasks take a given amount of time to execute and must be completed one after the other, processes have a given processor cost and can be added up one after the other to calculate total processor usage.

The three stages are now described in more detail with reference to Algorithm 4.

Initiation occurs on the current group leader whenever there is a change in group membership or agents are added or removed (l. 17). A new session is created by incrementing the current session ID (l. 19), defining the parameters of the session (ll. 20-23), and attempting to commit an OAC(*start*, …) transaction (ll. 26-28). Session IDs are guaranteed to be unique and increasing since for a correct group every potential leader will have committed the same series of OAC(*start*, …) transactions as every other host.

Upon committing the OAC(*start*, …) transaction (l. 38), a host first verifies that the transaction is still valid (ll. 39-40), prepares the session data (ll. 41-48), and begins the first bundle building phase (l. 50). It is possible for a host to join a new session prior to committing the start transaction, since update messages (l. 59) can be received out of order. In this case the host prepares the new session and tracks all updates but does not make any bids of their own until the start transaction is committed.

In a bundle building round (l. 51) a host greedily adds to its bundle from all the currently unbundled agents until it can no longer place winning bids on any agents. The build round, r_p, replaces the bid time concept proposed in [69] to avoid issues with unsynchronized clocks. The bundle building strategy (l. 56) is not dictated by the algorithm; convergence is guaranteed so long as bid generation has the property of diminishing marginal gains [69]. In the time domain one way to satisfy this criterion is to offer time-discounted rewards, stating that the later a task is started the lower the reward. After transforming to the usage domain the analogous strategy is to have rewards that decrease as the processor usage goes up. One such strategy is presented below in Appendix II, which offers usage-discounted rewards to both individual agents and clusters of

agents with strong affinities. Using this method of bid comparison bundles are built by following the recursive strategy outlined in Appendix II. This strategy attempts to locally optimize the reward function $R(b_h)$ while ensuring that every agent has been claimed by a host. Once a bundle is built the host distributes any changes to its accepted bids (l. 57) and checks to see if consensus has been reached (l. 58); the process of checking consensus is described after the discussion of processing bid updates.

Bid updates are distributed (l. 82) through update messages. An update message contains the sender, the session ID, and the list of bids that have changed since the last update was sent. On receiving an update message (l. 59) the host insures that the update belongs to the current session. If it is from an earlier session the update is ignored (l. 60), and if it is from a later session the current session is abandoned and the host joins the new session (l. 62). Each updated bid is processed (l. 65) as follows:

(l. 66) The sender's bid is updated in the $bidTable_p$ in order to track consensus.

(ll. 67-68) If the bid is from a round greater or equal to the current round, the round is incremented.

(l. 69) The currently accepted bid is stored for later comparison.

(l. 70) Conflicts are resolved via the conflict resolution rules in Table 4.7-1 and the bid comparison operator in Appendix II. These rules are almost identical to the rules in [69], with the exception that the reset & broadcast action was changed from rebroadcasting the sender's bid to broadcasting the receiver's nil bid. This change is necessary to allow consensus checking via the $bidTable_p$, and is possible because of the condition that every host is capable of sending their updates to every other host. If this condition is not met by the network then the bid update would have to include two parts: one containing the sender's bid and the other containing the receiver's opinion.

(ll. 71-72) If the newly accepted bid has become worse than the bid that was previously accepted, the host may now have a chance of adding that agent to their bundle, and so a bundle building round is queued.

(ll. 73-76) If the session is ready, i.e., the start transaction was committed, and there are updates to distribute then a distribution is queued. Similarly, if the session is ready consensus is checked.

Table 4.7-1 Bid Conflict Resolution (Adapted from [69] with additions)

	Host q (sender) thinks z_{qj} is	**Host p (receiver) thinks z_{pj} is**	**Receiver's action* (default: leave & broadcast)**
1 2 3	q	p	if $y_{qj} > y_{pj} \rightarrow$ trim & update & broadcast if $y_{qj} = y_{pj}$ and $z_{qj} < z_{pj} \rightarrow$ trim & update & broadcast if $y_{qj} < y_{pj} \rightarrow$ update time & broadcast
4 5 6		q	if $r_{qj} > r_{pj} \rightarrow$ update & broadcast if $r_{qj} = r_{pj} \rightarrow$ leave & no broadcast if $r_{qj} < r_{pj} \rightarrow$ leave & no broadcast
7 8 9 10 11		$m \notin \{p,q\}$	if $y_{qj} > y_{pj}$ and $r_{qj} \geq r_{pj} \rightarrow$ update & broadcast if $y_{qj} < y_{pj}$ and $r_{qj} \leq r_{pj} \rightarrow$ leave & broadcast if $y_{qj} = y_{pj} \rightarrow$ leave & broadcast if $y_{qj} < y_{pj}$ and $r_{qj} > r_{pj} \rightarrow$ reset & broadcast if $y_{qj} > y_{pj}$ and $r_{qj} < r_{pj} \rightarrow$ reset & broadcast
12		none	update & broadcast
13	p	p	if $r_{qj} = r_{pj} \rightarrow$ leave & no broadcast
14		q	reset & broadcast
15		$m \notin \{p,q\}$	leave & broadcast
16		none	leave & broadcast
17 18 19	$m \notin \{p,q\}$	p	if $y_{qj} > y_{pj} \rightarrow$ trim & update & broadcast if $y_{qj} = y_{pj}$ and $z_{qj} < z_{pj} \rightarrow$ trim & update & broadcast if $y_{qj} < y_{pj} \rightarrow$ update time & broadcast
20 21		q	if $r_{qj} \geq r_{pj} \rightarrow$ update & broadcast if $r_{qj} < r_{pj} \rightarrow$ reset & broadcast
22 23 24		m	if $r_{qj} > r_{pj} \rightarrow$ update & broadcast if $r_{qj} = r_{pj} \rightarrow$ leave & no broadcast if $r_{qj} < r_{pj} \rightarrow$ leave & no broadcast
25 26 27 28 29		$n \notin \{p,q,m\}$	if $y_{qj} > y_{pj}$ and $r_{qj} \geq r_{pj} \rightarrow$ update & broadcast if $y_{qj} < y_{pj}$ and $r_{qj} \leq r_{pj} \rightarrow$ leave & broadcast if $y_{qj} = y_{pj} \rightarrow$ leave & broadcast if $y_{qj} < y_{pj}$ and $r_{qj} > r_{pj} \rightarrow$ reset & broadcast if $y_{qj} > y_{pj}$ and $r_{qj} < r_{pj} \rightarrow$ reset & broadcast
30		none	update & broadcast
31	none	p	leave & broadcast
32		q	update & broadcast
33		$m \notin \{p,q\}$	If $r_{qj} > r_{pj} \rightarrow$ update & broadcast
34		none	leave & no broadcast

*Bids are compared using the bid comparison operator. The resulting actions are handled as follows:

trim: **if** j was part of a cluster bid **then** remove j's cluster and every later bid from $bundle_p$;
else remove j and every later bid from $bundle_p$;

update: $bidTable_p[p][j] := bidTable_p[q][j]$;

update time: $bidTable_p[p][j].r = r_p$;

reset: $bidTable_p[p][j] := \{ j$, none, $\{0,-\infty\}$, 0 $\}$; /* nil bid */

broadcast: $outbox_p[j] := bidTable_p[p][j]$;

no broadcast: /* do nothing */

leave: /* do nothing */

Once all bids from every currently received update message have been processed, queued bundle builds or distributions are carried out (ll. 77-81). Waiting until all updates are processed ensures that the most current information is used when building bundles, and reduces the number of updates sent.

Consensus is checked (l. 86) by the group leader using the $bidTable_p$. The table stores all bids that each host has currently accepted, and consensus is reached once every agent is claimed and every host has accepted the same set of bids. Consensus is guaranteed from the convergence properties of the algorithm. When consensus is achieved the leader flags the session as decided (l. 88) and attempts to commit an OAC(*finish*, …) transaction (ll. 89-92) until either it is successful or the situation changes. Global agreement on the allocation result is guaranteed through the use of an OAC transaction. Upon committing the *finish* transaction (l. 93), each host accepts the allocation result and, if the ID matches the current session, flags the session as decided.

4.8 Agent Transfer

Agent transfer is considered normal behaviour even when all processes are correct, and is triggered by the Agent Allocation algorithm. In all cases, the agent a is running on one host, H_{old}, who voluntarily releases ownership and allows the agent to transfer and resume operation on a second host, H_{new}. Beyond the inevitable time delay, there should be no other impact on the operation of the transferred agent or any other agents. This specification is formally defined as:

AT-Transparency: The agent will be transferred without any change in behaviour required of any other agent.

AT-Consistency: The agent shall not lose any relevant information, either internal data or incoming messages, from the transfer.

Proofs of AT-Transparency and AT-Consistency are derived in Appendix II.

4.8.1 Failure Assumptions

This algorithm assumes that a does not crash prior to sending its state, and that H_{old} does not crash prior to releasing ownership of a. If either of these events occurs the agent has failed and a must be recovered as described under Agent Recovery. The algorithm also relies on OAC, a group membership service for the hosts, and a DDB.

4.8.2 Algorithm Overview

The transfer operation is a two step process: first, H_{old} works with a to temporarily "freeze" the agent in the DDB, then H_{new} "thaws" the agent from the DDB and allows a to resume operation.

These steps happen in order but are otherwise unrelated, and an unspecified amount of time may elapse between freezing and thawing without complication. The freezing and thawing processes are described here with reference to Algorithm 5 and Algorithm 6, respectively.

The freezing process begins when H_{old} acknowledges that it has lost the bid for *a* after a session of agent allocation, and subsequently calls freezeAgent(). H_{old} then proceeds on two avenues: a) working with *a* to freeze the agent, and b) informing other hosts that *a* is being frozen. H_{old} asks *a* to freeze (l. 2) and begins forwarding all messages addressed to *a* (a.m.a.a.) into the primary message queue (l. 4). Upon receiving the freeze request, *a* packs its state and sends it back to H_{old} before shutting down (ll. 17-20). H_{old} informs other hosts about the freeze with an OAC(*freeze*, ...) transaction (ll. 5-8), which informs them to redirect a.m.a.a. into the secondary message queue (ll. 21-24). Before finalizing the freeze, H_{old} must wait until it receives acknowledgement from each host active when the freeze transaction was committed (l. 10) so that H_{old} can be sure that it will receive no more messages addressed to *a*. At this point H_{old} can submit *a*'s state to the DDB and safely release ownership of *a* (ll. 12-16).

While *a* is frozen, hosts continue forwarding a.m.a.a. to the secondary queue until H_{new} is ready to begin the thaw process by calling thawAgent(). H_{new} then creates a shell process for *a* (l. 2), begins forwarding a.m.a.a. to a local message queue (l. 3), and claims ownership of *a* with an OAC(*claim*, ...) transaction (ll. 4-7). When a host commits the claim transaction, if *a* still belongs to H_{new} (l. 13), they begin forwarding a.m.a.a. to H_{new} (l. 15). If *a* no longer belongs to H_{new}, the claim attempt is abandoned (l. 18). Before H_{new} can complete the thaw it must wait until it receives acknowledgement from each host active when the claim transaction was committed and all active OACs are decided (l. 9) so that H_{new} can be sure no more messages will be added to the secondary queue. Then H_{new} can send a (*thaw*, ...) message to *a* and begin forwarding a.m.a.a. to *a* (ll. 10-11). When *a* receives the thaw message it unpacks the state, processes each of the primary, secondary, and local message queues in turn, and then resumes operation (ll. 19-22).

4.9 Agent Recovery

Agent recovery occurs after an agent (or host managing an agent) fails. The agent, *a*, is then assigned a host, *H*, through the agent allocation algorithm and recovered from the latest backup. This is the lossy counterpart to agent transfer, and information is lost in terms of agent state and

undelivered messages. The completeness of the backup is determined by the requirements of the agent, and can vary between containing only basic information and containing a nearly complete copy of the state. Naturally, there is a trade-off between completeness and network traffic necessary to maintain the backup, and the agent designer is responsible for ensuring that the agent recovers into a usable state. Without restricting the completeness of the backup or the state of the agent upon recovery, the process is defined by:

AR-Recovery: The agent will be recovered to the latest backup and begin receiving messages.

Proof of AR-Recovery is provided in Appendix II.

4.9.1 Failure Assumptions

This algorithm assumes *H* does not crash prior to recovering *a*, and that the agent shell does not crash before the recovery is complete. If either of these events occurs the recovery fails and new recovery run must be initiated. The algorithm also relies on OAC, a group membership service for the hosts, and a DDB.

4.9.2 Algorithm Overview

The backup procedure is straightforward. An agent calls backupAgent() as it deems necessary and sends the state backup to its local host (Algorithm 7 ll. 1-3). Upon receiving the state backup, the host submits it to the DDB (Algorithm 7 ll. 4-5). Any number of backups can occur before a recovery occurs, if one occurs at all.

Agent recovery is described with reference to Algorithm 8. The recovery process begins when *a* crashes, or if *H* was not already the owner, when *H* receives ownership of the crashed agent. *H* then attempts to recover *a* from the latest backup (l. 4), and if successful notifies other hosts that *a* has been recovered (ll. 6-9). Upon receiving the recover message for the host, *a*'s new shell recovers itself from the backup and resumes normal operation (ll. 10-12). When the hosts are notified of the recovery via the OAC(*recovered*,...) transaction they begin forwarding all messages addressed to a to *H* (ll. 13-15). Multiple hosts recovering the same agent is prevented by the Agent Allocation algorithm.

Chapter 5
Just-in-Time Cooperative Simultaneous Localization and Mapping

Although the issues of localization and mapping are not directly related to the core topic of control system design, they are very common problems in mobile robotics and had to be solved for the implementation used in the experimentation. Simultaneous Localization and Mapping (SLAM) is a popular solution to the exploration and navigation problems, but many implementations are not suitable for large-scale applications [70], relying on powerful sensors and an abundance of processing power. In addition to requiring a distributed algorithm the goals of this control system include adaptability and versatility, and so a new strategy, termed Just-in-time Cooperative SLAM (JC-SLAM), was developed. The primary feature of this strategy is the use of out-of-order processing that allows for greater flexibility in processing requirements, which typically results in a higher rate of processing for sensor readings.

The field of SLAM has become very large, and attempting to categorize all approaches is beyond the scope of this thesis. Instead, a brief overview of the basic elements of SLAM is presented to introduce the topic, and then several key issues are discussed as they relate to this implementation. An excellent and detailed review of the topic can be found in [70]. The SLAM implementation discussed here was designed to be generic in terms of sensor support, robot hardware, and processing resources. In processing-starved systems the method allows for sensor data to be selectively held and recovered later to still provide useful information to the current state, yet under ideal conditions it performs identically to traditional SLAM approaches. Furthermore, rather than competing with existing SLAM implementations, the key concepts of maintaining particle history and out-of-order processing can also be integrated into many existing particle filter SLAM algorithms to compliment their unique strategies and optimizations. The foundation and implementation of JC-SLAM are developed in this chapter, and its performance is evaluated experimentally in Chapter 7.

5.1 Background on Simultaneous Localization and Mapping

For autonomous mobile robotic applications most navigation tasks require two sets of information. First, some form of map that tells the robot where it is going, or at least where it has been, and secondly, some form of localization that tells the robot where it is relative to the map.

In unknown environments it can be difficult to obtain one set of information without the other, and so the concept of SLAM was developed. Introduced in the late 1980s, SLAM has become a widely used technique in mobile robotics, and many different approaches have been developed. The vast majority of these approaches fall under three principle paradigms: extended Kalman filter (EKF) approaches, graph-based approaches, and particle filter approaches [70]. In general, each paradigm has certain advantages and disadvantages that are consistent for their approaches. The EKF implementations have been used successfully in many applications, and it can be applied incrementally to solve the online SLAM problem. However, the computational complexity of basic EKF SLAM implementations is $O(n^2)$ where n is the number of map features [71], though many published implementations use strategies to improve real-time performance. Graph-based approaches are built from the notion that robot poses and map features are nodes that can be connected by observation constraints, and SLAM can be solved by finding the minimal energy state of the graph. Graphical implementations scale well to large problems because they are generally sparsely connected. However, although some modifications can be made to support real-time usage, many graphic-based approaches are best suited to solving offline SLAM [70], i.e., all data is gathered and then SLAM is carried out after the fact. In recent years, particle filter approaches have gained significant popularity due to their ability to model non-linear distributions [72], solve both online and offline SLAM, avoid some of the problems of data association by natively supporting multiple hypotheses for landmark/feature recognition, and can typically be implemented efficiently [70]. The major problems encountered by particle filter approaches are that i) the number of required particles increases with the dimension of the system state, and ii) particle diversity decays over time [72,73].

Beyond the basic paradigm, there are many other choices when developing a SLAM implementation, perhaps the most significant being the type of map. Typically either a grid based [74,75] or feature based [76,77] map is used, although many other options can be found in the literature. This choice is often affected by the type of sensors being used, but can also be influenced by the operating environment [72], the choice of path-planning and related algorithms, and even by the availability of communication bandwidth [78]. Depending on the complexity of the SLAM algorithm and the environment it must navigate, other tangential problems that require solutions include: uncertain data association, dynamic environments, active or passive

exploration, the presence of multiple robots, and merging maps from simultaneous exploration efforts. Various techniques for handling these issues can be found in the literature.

5.1.1 Localization with Particle Filters

Particle filters are a useful way to represent arbitrary posterior distributions, and are able to track a clearly defined history of distributions [79]. The filter consists of *n* particles, each with a state vector, $x_{0:t}^{i}$, and a weight, w_t^i, where subscript $0{:}t$ indicates the states from time 0 to time t, subscript t indicates the weight at time t, and the superscript i indicates the i^{th} particle. The combination of these weighted particles gives the posterior distribution $p(x_{0:t}|y_{0:t})$ given the history of observations $y_{0:t}$, where subscript $0{:}t$ indicates the observations from time 0 to time t. Each particle can the thought of as an estimate of the current state, and their associated weight corresponds to how believable that estimate is given the current set of observations. Typically the states at each time are considered to be a first order Markov process. That is, a state x_t depends only on the previous state x_{t-1}, and an observation taken at time τ depends only on the state x_τ [80]. The treatment of each particle filter can now be developed as follows. The filter is *initialized* to a prior distribution $p(x_0)$, where each particle is randomly sampled from the prior and assigned an initial weight $w_0^i = 1/n$. Note that the initial weights are equal since the density distribution of particles follows that of the prior. The filter is then updated using *prediction* and *correction* steps [80] to obtain posterior distributions $p(x_{0:t}|y_{0:t-1})$. The prediction step uses the transition density model, $f(x_t|x_{t-1})$, to predict the current state based on the previous state. A correction step is used to incorporate observations into the state model, and can be seen as updating the weights of each particle based on how well they explain the observed data. The final requirement for a particle filter is a technique for *resampling*, where a new sampling of particles is generated from the current distribution.

Degeneracy is a common problem in particle filters, where after a number of steps only a small set of particles have significant weights while the others have weights close to zero [81]. In fact, [82] proves that the variance of the weights is strictly increasing and so there is no way to prevent degeneracy from accumulating. The standard way to measure degeneracy is to calculate the effective number of particles n_{eff}. As stated in [81] it is impossible to calculate this quantity exactly, but an estimate n_{eff}^{*} can be found by:

$$n_{eff}^{*} = \frac{1}{\sum_i (w_t^i)^2} \tag{5.1-1}$$

From (5.1-1), $n_{eff}^{*} \leq n$ and a lower n_{eff}^{*} indicates higher degeneracy. One method of reducing degeneracy is to periodically resample the particles, in effect discarding particles with low weights and creating new particles in areas where they will be useful. Commonly a threshold value is set for the effective number of particles and resampling is done whenever n_{eff}^{*} drops below this point. There are a number of resampling algorithms available, including residual and stratified resampling, but both [81] and [83] recommend systematic resampling since it is simple to implement, and, although it is difficult to analytically evaluate its performance, it is generally comparable to the other resampling strategies.

5.1.2 Mapping with a Probabilistic Occupancy Grid

Numerous types of maps have been developed, generally either geographical, such as potential field maps [84] and occupancy grids [85], or topological, such as the feature based map in [86]. Occupancy grids were chosen for this implementation since they are able to incorporate data from many types of sensors, do not significantly limit the type of path planning algorithms that may be used, and as shown in [86] can be useful in constructing other types of maps. The map consists of a 2D grid of consistently sized cells. Each cell is assigned a value between 0 and 1 to indicate probability it is occupied by an obstacle. In this Bayesian approach values close to 1 mean that it is most likely occupied while values close to 0 mean it is most likely empty.

When the probabilistic occupancy grid (POG) is constructed in this way a new sensor reading, s, pertaining to a cell, C, can be incorporated in the map simply by applying Bayes' rule:

$$p(C|s) = \frac{p(s|C)p(C)}{p(s)} = \frac{p(s|C)p(C)}{p(C)p(s|C) + \big(1 - p(C)\big)\big(1 - p(s|C)\big)} \tag{5.1-2}$$

Where $p(C)$ is the probability currently associated with that cell, called the prior probability of C. The conditional probability $p(s|C)$ is determined by the sensor model and is a measure of the likelihood of obtaining that sensor reading assuming C is occupied. The prior probability of s, $p(s)$, acts as a normalizing constant, and can be conveniently rewritten in terms of $p(C)$ and $p(s|C)$.

5.1.3 Prediction Step

The state of each particle is advanced based on the predicted behaviour of the object during that time interval. Conveniently the tracked objects, the avatars, are willing participants in the system and are able to provide estimates of their actions, as well as a measure of the associated uncertainty. This creates a simple velocity-based model with some added noise:

$$x_t^i = x_{t-1}^i + \left(\dot{x}_{t-1}^i + N(0, \sigma^2)\right)\Delta t \tag{5.1-3}$$

where $\dot{x}_{t-1}^i$ and σ are determined when each avatar calculates its velocities and uncertainties appropriate for the physical hardware, and Δt can dynamically scale based on the magnitude of the velocity and acceleration. This model is used by the avatar agent to generate transition updates of the forms $\langle t_t, x_t \rangle$ which are appended to the particle filter history. The model assumes that the proposal distribution is equal to the transition probability, and therefore prediction steps do not impact the particle weights. If a different proposal distribution is used the weights can be updated at this point, though clearly only observations that have already been processed can be used in the proposal distribution. This weight update can happen separately from the correction step weight update, and if prediction and correction steps are taken in turn is mathematically identical to performing a single combined weight update.

5.1.4 Correction Step

Map Updates: The simplicity of the POG makes it possible to incorporate readings from a variety of sensor types. The type of sensor determines what cells are affected and how the conditional probability $p(s|C)$ is calculated. Then (5.1-2) can be used to compute the new belief of each cell. When dealing with a particle filter the sensor model is simply applied at each particle and adjusted by the particle weight. An example of a model for sonar sensors can be found in [85].

Particle Filter Updates: Almost all sensor readings yield information about the relationship between the state of an object and the map. This information can be used to create an update for the map as discussed above and can also be used to update the weights of the particle filter. The weight of each particle can be updated by the equation:

$$w_t^i = \frac{w_{t-1}^i g(y_t|x_t^i)}{\sum_j w_{t-1}^j g(y_t|x_t^j)} \tag{5.1-4}$$

which simply scales the weight of each particle based on the observation density $g(y_t|x_t^i)$, a measure of how well the particle explains the current observation, and then normalizes so that the new weights sum to 1.

5.2 Just-in-Time Cooperative SLAM

JC-SLAM was developed as a real-time, distributed, and scalable cooperative SLAM implementation that works within the HAA architecture [87]. Specifically, it uses the distributed processing network to share both computations and information between hosts and to provide useful mapping and localization data in real-time. Further, it allows historical information to be incorporated into the current state with minimal additional computation and delays processing of weight updates until they are required, which allows combining multiple observations into a single update to reduce computations. This JC-SLAM implementation is built on the probabilistic occupancy grid and particle filters. Due to its structure and the framework in which it is implemented it also provides some advantages over existing cooperative SLAM implementations. In [88] robots share sensor information but require each robot to locally repeat all the calculations as well as introducing a phase delay with respect to when the information is processed by each robot, resulting in maps that are out of sync. In [89] local maps are maintained by each robot and processing is not fully distributed, in addition to having a fully centralized feature map with which all the robots communicate. In [90] a heterogeneous system is developed, consisting of *master* robots with significant processing power and powerful sensors and *slave* robots with poor sensors, but it is not robust against failure, and uses a Kalman filter approach that is unable to incorporate observations that occurred in the past. Finally, the particle filter approach used in [79] is capable of integrating historical observations, however, their technique relies on re-simulation to reduce the effects of degeneracy. Re-simulation requires that the particle filter be reset to a point in the past and then simulated again as if all the observations were new. The authors admit this to be a very computationally intensive process, and state that future work must be done to reduce the frequency of re-simulation. The lazy belief propagation technique developed here avoids the need for re-simulation, and replaces it with a process that requires minimal computations.

The key feature of JC-SLAM is the ability to process observations out-of-order and to delay processing when resources are scarce without unnecessarily impacting the performance of other

tasks such as navigation and exploration. This feature can benefit single avatar systems and for simplicity the approach is primarily discussed in that context in the remainder of this chapter. However, the discussion is equally applicable to multi-avatar systems as shown in the implementation in Section 5.2.3. In fact, the benefits of out-of-order and delayed processing become even more important as the complexity of the system grows, since it is less likely that the processing demands will be evenly spread out over time.

In this application a particle filter is stored as a series of blocks containing a set of weights and a time-ordered sequence of states. These blocks are separated by resampling events that redistribute the particles to better reflect the sampling density. This data structure is visualized in Fig. 5.2-1, which shows two blocks separated by a resampling event at time *r*. In the figure the variables are labeled with left superscripts to indicate block, right superscript to indicate particle number, and right subscript to indicate time where applicable. When new state predictions (x^1 to x^n) arrive at time *t* they are added to the end of the newest block. When a resampling event occurs the current block is closed, but not discarded, and a new block is created, where the initial states are determined by the resampling algorithm and the initial weights are equal.

Maintaining a history of particle states in this way requires more data storage that simply tracking the current state, but by modern computing standards this additional cost is negligible. To give an example, for a particle filter with 1000 particles with state of position *x*, *y*, and rotation *r*, making one prediction a second would require roughly 12 KB/sec of storage, less 42 MB after an hour. Most computers have RAM one to two orders of magnitude larger than this. Additionally,

Block 1	*Particle 1*	*Particle 2*		*Particle n*
Weight	${}^1w^1$	${}^1w^2$		${}^1w^n$
Time	*State*			
${}^1t_0 = 0$	${}^1x_0^1$	${}^1x_0^2$		${}^1x_0^n$
${}^1t_1$	${}^1x_1^1$	${}^1x_1^2$		${}^1x_1^n$
${}^1t_r = r$	${}^1x_r^1$	${}^1x_r^2$		${}^1x_r^n$

Block 2	*Particle 1*	*Particle 2*		*Particle n*
Weight	${}^2w^1$	${}^2w^2$		${}^2w^n$
Time	*State*			
${}^2t_0 = tr$	${}^2x_0^1$	${}^2x_0^2$		${}^2x_0^n$
${}^2t_1$	${}^2x_1^1$	${}^2x_1^2$		${}^2x_1^n$
${}^1t_{new} = t$	${}^1x_{new}^1$	${}^1x_{new}^2$		${}^1x_{new}^n$

Fig. 5.2-1 Particle Filter Data Structure

old blocks can be deleted after a given amount of time; however, this means that observations occurring during that block can no longer be processed.

5.2.1 Lazy Belief Propagation

Lazy belief propagation is founded on two principles that reduce the amount of processing that is required. First, the processing of observations can be delayed but should still provide useful information to the current distribution, and second, updates to weights should be delayed until their information is required. Consider the standard particle filter that uses fixed time steps and repeats the cycle of prediction and correction updates every step. In JC-SLAM the prediction updates can have varying durations, and many predictions can occur before any correction updates are made. Further, correction updates associated with some time in the past may occur in any order. The lazy belief propagation strategy makes three claims that are sufficient to function under these conditions:

Strategy 1: Delayed calculations of weight updates

Observation densities can be accumulated between weight updates and weight calculations can be delayed until the weights are explicitly requested without impacting the performance of the particle filter.

Reasoning: Consider any two observation density updates, g_1 and g_2, that are adjacent w.r.t. observation time. Without loss of generality, assume g_1 occurs before g_2. By definition, (5.1-4) can be applied recursively as follows:

$$w_2^i = \frac{w_1^i g_1^i}{\sum_j w_1^j g_1^j} \tag{5.2-1}$$

$$w_3^i = \frac{w_2^i g_2^i}{\sum_j w_2^j g_2^j} \tag{5.2-2}$$

Substituting (5.2-1) into (5.2-2) yields:

$$w_3^i = \frac{\frac{w_1^i g_1^i}{\sum_k w_1^k g_1^k} g_2^i}{\sum_j \frac{w_1^j g_1^j}{\sum_k w_1^k g_1^k} g_2^j} = \frac{w_1^i g_1^i g_2^i}{\sum_j w_1^j g_1^j g_2^j} \tag{5.2-3}$$

(5.2-3) shows that not only is the order in which the observations are applied irrelevant, but the two observations can be combined prior to applying (5.1-4). It can be shown by induction that any number of observations can be combined and the final weights calculated in a single update. Though the final weights given this set of observations are exactly equal to the weights had the observations been processed recursively, one concern is the triggering of resampling events. Resampling events are triggered when the effective number of particles drops below a threshold, and if the weights are not calculated regularly the effective number of particles will not be known. In practice, since specification of the aforementioned threshold has considerable flexibility before it affects the particle filter performance, delaying weight updates in not an issue so long as the weights are calculated semi-regularly.

Strategy 2: Out-of-order processing

A strategy of processing observations in a last-in-first-out (LIFO) order and allowing out-of-order (OOO) processing of observations is more effective for real-time SLAM in resource constrained systems than the standard time ordered approach.

Reasoning: The justification for LIFO processing has two parts. Firstly, in systems with sufficient resources to process all observations LIFO is exactly equivalent to the standard time ordered approach. This is shown visually in Fig. 5.2-2, where a series of observations and the required processing time is plotted along the time axis. Note that the order and timing of observation processing is the same for both Standard and LIFO. Secondly, in systems with insufficient processing resources the traditional approach is to discard any readings that cannot be immediately processed [76], where as LIFO with OOO processing allows old observations, that would otherwise be discarded, to be processed and still provide useful information.

Fig. 5.2-3 shows that a) LIFO processes a similar, though not necessarily identical, pattern of near current (at the time of processing) observations, which is important in maintaining the accuracy of the particle filter; and b) LIFO processes additional OOO observations that can contribute relevant map and weight updates. In the case of a), note that the pattern of near current readings processed by Standard is dictated solely by the circumstances of timing and holds no innate advantage over a slightly different pattern such as the one resulting from LIFO. In the case of b), though the observation densities generated from an old observation are different than if the

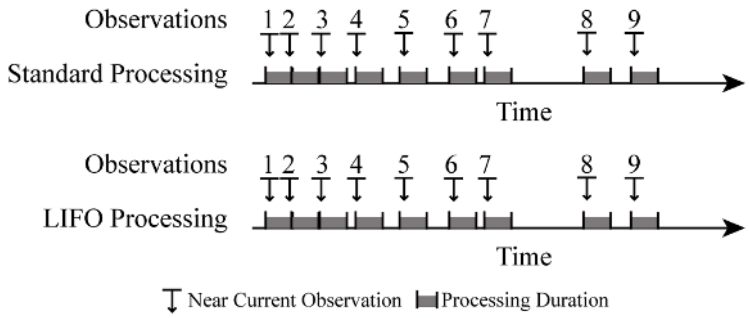


Fig. 5.2-2 Standard vs. LIFO + OOO with Sufficient Processing Resources

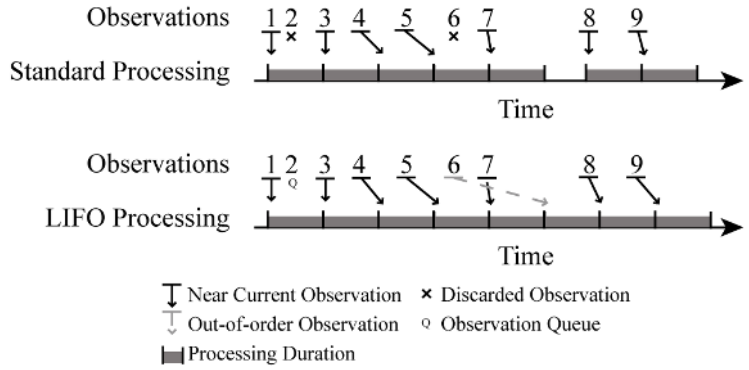


Fig. 5.2-3 Standard vs. LIFO + OOO with Insufficient Processing Resources

observation had been processed in order, they are still validly calculated based on up-to-date map information in the same way as any other observation. Since the sequence of observations is different the timing of resampling events may also change; however, since each processed observation is valid there is no reason to suspect that the performance of the particle filter will be unduly affected.

Strategy 3: Propagation of observations through resampling transforms

Observation density updates can be forward-propagated through resampling transforms via the relation:

$$ {}^{2}g_a^i = {}^{1}g_a^{p_i} \tag{5.2-4}$$

where the 1 and 2 left superscript indicates densities before and after resampling, respectively, and p_i is the parent of particle i.

Reasoning: This result relies on the fact that the weights of particles before and after resampling are independent. That is, the change in weight of a parent particle has no relation to the change in weight of its children. Once this is understood the following analogy can be constructed. Consider a scenario where particles progressed normally until time t_r when the set of parent particles is resampled into a set of child particles. At some time after t_r an observation taken at time t_o, $t_o < t_r$, i.e., taken before the resampling, is processed. Since the observation corresponds to a time in the history of the parent particles the question becomes: how is this observation related to the child particles? Now consider the corresponding scenario where no resampling has occurred, instead there is a single set particles, a number of which happen to have overlapping histories before t_r. Specifically, this is the same as if each child particle simply had its parent's history attached to the front. In this case when the observation from t_o is processed the observation density of each particle can be calculated as normal, and it is clear that all particles with the same history have equal observation densities. Moreover, this observation density is exactly equal to that of the parent particles in the first scenario. This results in the simple relation given in (5.2-4). Observation densities can still be accumulated as in Strategy 1. Replacing g_1^i and g_2^i in (5.2-3) with ${}^2g_1^i$ and ${}^2g_2^i$, respectively, and then substituting in (5.2-4) results in (5.2-5), which clearly has the same properties as (5.2-3):

$$ {}^2w_3^i = \frac{{}^2w_1^i {}_1g_1^{p_i} {}_1g_2^{p_i}}{\sum_j {}^2w_1^j {}_1g_1^{p_j} {}_1g_2^{p_j}} \tag{5.2-5} $$

Based on these three claims the lazy belief propagation strategy allows observation densities to be applied in any order and passed forward along the history of the particle filter to contribute to the estimate of the current state. Further, this technique is able to delay weight updates until they are required and to combine accumulated density updates to reduce the amount of computation. Finally, in an ideal system where every observation is processed immediately, lazy belief propagation performs identically to a standard particle filter.

The impact of lazy belief propagation on the triggering of resampling events and which particles survive each resampling is not immediately clear. Mathematically, the order in which observations are processed plays a role in when resampling occurs and which particles survive, thus changing the performance of the particle filter. However, in practice the impact seems minimal for a number of reasons. Resampling only occurs when a significant number of particles

no longer match the accumulated observations and only particles that are "bad" predictions are discarded. In order to significantly change the result of the resampling a theoretical set of observations must exist that disagree with the previously processed observations, which could potentially increase the weight of these bad particles to the point where they are no longer discarded. If such a set of observations does exist it suggests that the bad particles were incorrectly labeled in the first place, and that the previous observations were overly confident in their weight assignment (i.e., the error associated with the readings was underestimated). Furthermore, if the observation error is underestimated it can result in the same performance problems even if the observations are processed in order. By correctly estimating the error in the observations it is possible to avoid discarding particles that still have worth and prevent performance degradation from inadequate particle distribution. This justification is explored experimentally in Chapter 7, where JC-SLAM, with its aggressive out-of-order processing strategy, achieves accuracy equal to or better than algorithms that process observations strictly in order. These results have been submitted for publication in [91].

5.2.2 Example Scenario

The following simplified example is provided to illustrate how the JC-SLAM with lazy belief propagation is able to converge on the path of an avatar. The system consists of a single avatar moving backward and forward in front of a wall, periodically taking range measurements with significant uncertainty attached. Only the dimension perpendicular to the wall is considered, and it is assumed that the initial displacement is known. Fig. 5.2-4 gives a series of diagrams that depict the evolution of the particle histories and weights. The particle filter shown uses only three particles and can hardly be expected adequately track the avatar given the noise in the system, it is simply used visualize the process of delayed observation updates and propagating through resampling events.

5.2.3 Implementation of the JC-SLAM Algorithm

As described at the beginning of this section, every particle filter is stored as a time series of state predictions for each particle, grouped into blocks separated by resampling events; thus, each block corresponds to a set of predicted particle paths for a period of time. In order to implement lazy belief propagation two additional values are stored for each particle, the accumulated

observation density for that block and the observation density to be propagated to the next block. A "forward marker" is used to flag the earliest block with observation densities to be propagated forward. There are five events that must be handled to maintain and query the particle filter, these events are handled as followed with reference to Fig. 5.2-5.

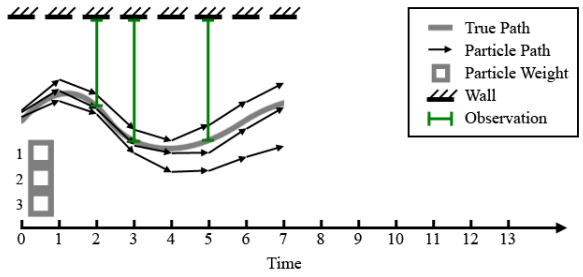
(a) Initialization: The first state predictions are sent by the Avatar Agent, particle weights are all equal, and observation densities are set to 1.

(b) Insertion of prediction updates: The new state predictions are appended to the state list and the prediction time is added to the time list. The end time of the current block is advanced. Note that unlike correction updates all prediction updates must be applied in order.

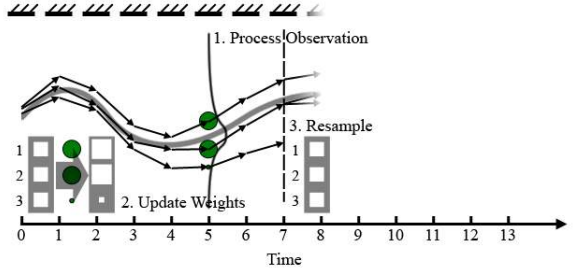
(c) Application of correction updates: The block containing the correction time is found, and then the accumulated observation density and forwarded observation density for each particle are multiplied by the new observation density. If the forward marker is greater than the current region it is set to the current region.

(d) Estimating States: The block containing the query time is found. If the forward marker is less than the query block the observation densities are propagated forward to the query block. New particle weights are calculated using the accumulated observation density and observation densities are reset to 1. The state estimate can then be calculated as a weighted sum.

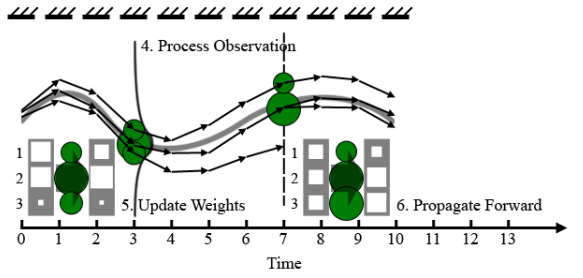
(e) Resampling: A new block is appended to the region list. A systematic resampling algorithm adapted from [81] is used to generate a new set of particle states and assign parents to the current set of particles.



(a) Starting from the initial position at time 0, the avatar begins moving backward and forward. At each time step the avatar generates prediction updates for the particles using the transition density model. Every few time steps the avatar takes observations using its range sensor, but no processing occurs at this time. Since nothing is known about the true path at this time, the weights of all particles are equal.

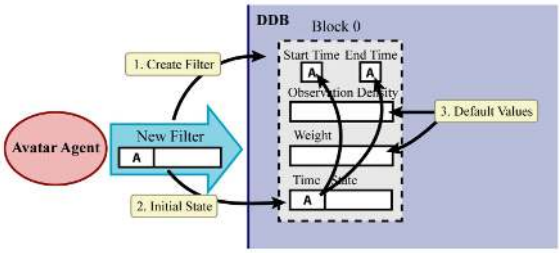


(b) At time 7 the observation from time 5 is processed (Event 1) and used to generate observation density updates for each particle, shown by the green circles. This observation gives roughly equal weights to particles 1 and 2, but particle 3 has almost no weight (Event 2). Because the number of effective particles was reduced to 2 a resampling event occurs (Event 3).

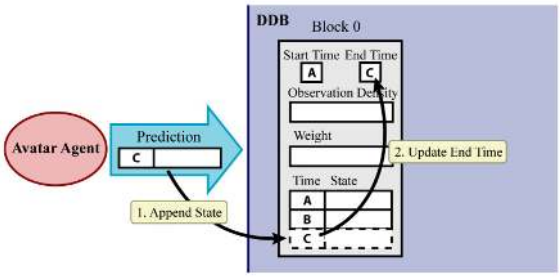


(c) The avatar continues on as before, generating prediction updates for the particles, until at time 10 the observation from time 3 is processed (Event 4). Observation density updates are generated for each parent particle (Event 5) and then propagated forward to its children (Event 6), resulting in higher weights being assigned to the children of particle 2.

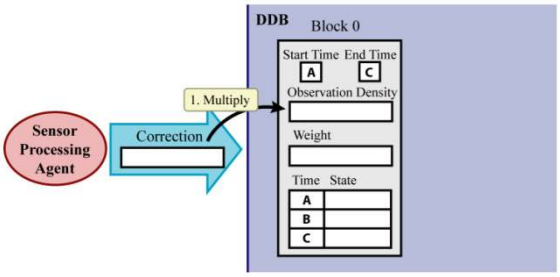
Fig. 5.2-4 JC-SLAM Example



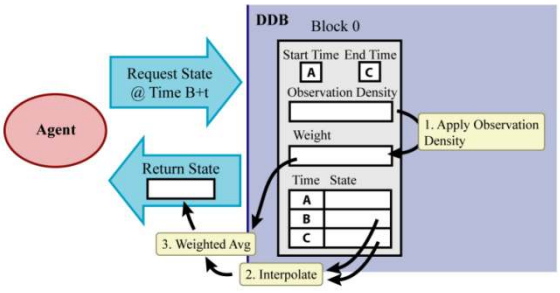
(a) Initialization



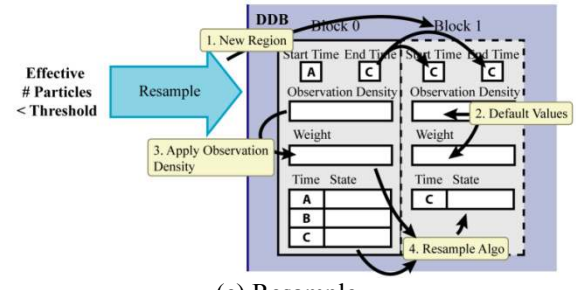
(b) Insert Prediction



(c) Apply Correction



(d) Estimate State



(e) Resample

Fig. 5.2-5 JC-SLAM Algorithm Flow

Chapter 6
Implementation

6.1 Host, Avatar, Agent Architecture

The HAA architecture provides a strategy for implementation, but many design choices are still open when developing the code platform which one needs to build the control system. The first decision is about selecting the programming language and environment, and then to choosing what building blocks are required for the architecture. In this research, C++ was selected as the programming language, due to its object-oriented structure, the strength of the existing development tools and code libraries, and its compatibility with many operating systems and processors. The object-oriented class features of C++ are particularly useful for this research since every agent shares the same basic needs for communication and agent management, and many groups of agents share additional functionality, for example Avatar agents or Sensor Processing agents.

Several building blocks are required in order to implement HAA, but equally important are the support blocks to monitor, log, and debug the control system. In scenarios with numerous hosts and tens of agents, interactions become very complex, and proper tools are essential to develop and maintain a functioning control system. This section describes the three key blocks of the HAA implementation, AgentBase, AgentHost, and the DDB; and explains their requirements, capabilities, and, where applicable, potential alternative choices or variations. The five supporting blocks, AgentMirrior, the GUI, the Logger, AgentPlayback, and RemoteStart are then outlined to discuss their features and usefulness in developing the control system.

6.1.1 AgentBase

AgentBase is the foundation class for all agents, including AgentHost, the class that is used to maintain the host network. It provides basic functionality for all agents through tools such as timers, communication ports, and messaging infrastructure.

The most fundamental tool of AgentBase is the Universally Unique Identifier (UUID) [92], which are generated every time an agent is instantiated to ensure that there is no confusion between agents. A UUID is a 128-bit value that is algorithmically generated in a way that for a

finite set of IDs the probability of generating the same ID twice is negligible, and it can be estimated using the birthday paradox approximation from probability theory [93]:

$$p(n) \approx 1 - e^{-\frac{n^2}{2(128)}} \quad (6.1\text{-}1)$$

where *n* is the number of UUIDs in the set.

Communication in the system is done over non-blocking Transmission Control Protocol/Internet Protocol (TCP/IP) sockets that guarantee packet order and integrity, which is a requirement for the majority of the algorithms described in Chapter 4. There are two message formats implemented in AgentBase. Regular messages contain message ID, message length, message data, and optionally a forwarding address and a return address. These messages provide no guarantees on delivery in the event of agent or connection failure. Ordered Atomic messages act as a wrapper for regular messages and provide the delivery guarantees of Algorithm 2; specifically that the message will either be delivered to all targets or no targets.

A number of convenience modules are provided, including Timer, Callback, Conversation, and DataStream. Timers allow agents to set timeouts when they need to perform a task at a certain time or at regular intervals. Callbacks are a method of storing function pointers that can be passed to generic functions and called when a specific event occurs, for example after a timeout. Callbacks had to be specially designed in order to remain valid even if an agent is transferred to a different host. Conversations are a set of tools to facilitate sending a request to another agent and uniquely identifying the response so that the response can be directed to the appropriate handler. DataStream is a class that concatenates many different data types into a single block to simplify the process of sending and storing data. For example, during agent transfer an agent can pack its current state into a DataStream object, submit the block to the DDB, and then on the new host the agent can unpack the DataStream to recover its state and resume operation.

Additionally, each agent monitors their CPU usage in real-time and reports this information for use in the Agent Allocation algorithm.

6.1.2 AgentHost

The term *host* in HAA refers to the processing hardware, but of course there needs to be a corresponding software component in order to manage and operate the network. Each host runs

an instance of the AgentHost class, which provides the core of the distributed system. AgentHost has five main tasks: host group management, DDB management, agent allocation, agent management, and message routing.

The algorithms for maintaining the host group are described in Algorithm 3. These include methodologies for joining, leaving, and removing failed members, and all guarantee consistency across the members. Before a host can play a role in the control system it must successfully apply to the group and become consistent with the current global state.

Although the DDB can be considered a separate entity from the host group, it is convenient to pass all database operations though the local host, since this eliminates the need to maintain a separate group of DDB clients. This strategy of incorporating the DDB into the hosts does not dictate the specific implementation of the DDB, and the details of the implementation will be discussed in the Section 6.1.3.

Once the host group has been established, there is only one step remaining before agents can be instantiated and the control system is in operation: deciding on agent allocation. For this implementation the agent allocation algorithm developed in Algorithm 4 was used. This algorithm is run every time a new agent is requested, whenever an agent or host fails, and at regular intervals in order to balance processing load.

After deciding on the allocation, each host takes appropriate actions depending on the current state of the agents.

- If the agent is requested but not yet spawned: the host spawns an instance from the agent template and notifies the parent when it is ready.
- If the agent is already active: the previous host freezes the agent and the new host spawns an instance from the agent template, once the freeze is complete the new host gives the state data to the new instance to thaw and the agent resumes operation. The complexities of these operations are handled by Algorithm 5 and Algorithm 6.
- If the agent has failed: the new host spawns an instance from the agent template, provides the instance with the most recent backup data, and the agent resumes operation. Backup and recovery are handled by Algorithm 7 and Algorithm 8, respectively.

All inter-agent communication is routed through the hosts. This is convenient (and required) for four main reasons. First, no agent needs to be concerned about where another agent is located; the message is just sent to the local host, which has the necessary information and connections to ensure the message arrives at its destination. Secondly, and perhaps most importantly, it allows proper routing of messages to keep the agent transfer process transparent to other agents. Agents do not have to worry about the current state or closing/reconnecting a socket when another agent is transferred, they simply send messages as normal, and if the agent does not crash it will eventually be delivered. Thirdly, by routing all outside communication through the local host the number of active network connections is greatly reduced, which reduces network traffic by eliminating connection monitoring messages. Finally, it simplifies the task of monitoring inter-agent communication traffic, which is used to generate the agent affinities when generating agent allocation.

6.1.3 Distributed Database

The DDB is used to share various types of information throughout the network. In following the tenets of efficiency and persistency, the DDB is implemented as a highly available service, improving local performance of the database and preventing data loss when failures occur. Highly available services can be implemented using various distributed algorithms, and for this implementation a simple full distribution strategy was employed. The merits of full distribution versus alternative distribution strategies are discussed below, but here it was convenient to have each copy of the DDB directly managed by AgentHost. Every host maintains a copy of the DDB class, and all write operations are coordinated via OAC transactions to ensure consistency. Local agents are then guaranteed to have access to the most up-to-date information though the host, and read operations can happen independently.

The implementation also supports a “watcher” interface, where agents can register to monitor specific data objects or specific types of objects. These watchers are then notified when different events occur, including the basic events add, remove, and write, as well as a number of events specific to each object type.

Three metrics used to judge the quality of a distributed database are bandwidth, latency, and fault-tolerance. Bandwidth is a limited resource in the distributed system, and amount of data

that must be transferred increases with the amount of replication done by the database. Latency, used here to reference to the time between an agent requesting information from the database and receiving a response, is a measure of quality of service and should be kept as low as possible. Low latencies occur when the data is already locally available when the query is received. If the data must be retrieved from another host latencies can increase dramatically. Fault-tolerance is a second measure of quality of service and determines how many hosts can fail before data is lost and a client experiences a disruption of service. Specifically, an f-tolerant service can handle f host failures before a disruption occurs [61]. Maintaining low latency and high fault-tolerance has a direct cost in terms of bandwidth, and so a balance must be achieved that fits the needs of the application. The distribution strategies at opposite ends of this spectrum are *full distribution*, where each site maintains an update-to-date copy of the entire DB, and *on-demand distribution*, where updates are stored locally until specifically requested. The current implementation uses full distribution, but most applications use a strategy somewhere between the two extremes. A logical approach to selecting the amount of replication is to pick the number of failures, f, your system must be able to tolerate and use that as the lower baseline for replication.

6.1.4 Support Blocks

The five key support blocks are outlined in Table 6.1-1.

Table 6.1-1 HAA Implementation Support Blocks

Support Block	Description
Agent Mirror	The AgentMirror class simply maintains a real-time replica of the DDB, along with the notification hooks that allow other classes to track specific events. AgentMirror registers with an AgentHost, which first transmits an up-to-date copy of the DDB to AgentMirror and then proceeds to forward each new DDB update as they occur. Having a DDB replica is the basis for almost any real-time external monitoring of the system, since it tracks agent states and distribution, avatar states, mapping and localization, sensor readings, etc. Using AgentMirror, tools such as the GUI can be built.
Graphical User Interface	A Graphical User Interface (GUI) is often the most important monitoring tool. For this implementation the GUI, shown in Fig. 6.1-1, has four main functions: environment visualization, agent distribution, agent hierarchy, and DDB browser. 1. Environment visualization is the standard monitoring tool. It displays the state of the map in real-time, and can overlay information such as particle filters, avatar pose estimations, landmark estimations, avatar targets, planned paths, and sensor readings, as well as debugging information such as mission boundaries and the true obstacle positions.

	2. Agent distribution visualization (not visible in Fig. 6.1-1) is the key to monitoring performance of the distribution algorithm and the agent transfer and recovery processes. It organizes the history of each agent by host and displays changes in agent status, such as freezing, thawing, crashed, and recovering. It is the real-time version of the agent distribution graphs presented in Chapter 7, where they are explained in more detail. 3. The agent hierarchy is presented in the form of a collapsible tree with all child agents collected under their parent agent. This makes it possible to monitor how many agents are in the system and which types of agents are being requested by whom. Selecting an agent presents a summary of the agent's properties and statistics, and could potentially provide an interface to change parameters and make performance adjustments. 4. The DDB browser takes a similar form to the agent hierarchy, where DDB objects are organized in a tree based on their parentage; for example, sensor and particle filter objects are placed under their corresponding avatar object. Selecting an object provides a summary of the object data, and again could potentially present an editing interface.
Logger	Each agent has a Logger class where it records important events, statistics, and debugging messages. The Logger displays these messages in the console as they appear, but more importantly saves them to a file that can be reviewed offline. If something questionable occurs during a run, the logs are the first place to check. However, despite extensive logging it is often difficult to determine what went wrong, and even harder to determine why. For these cases more complete tracking is required, which led to the AgentPlayback system.
AgentPlayback	AgentPlayback was crucial to the debugging process, and made the challenging task of developing a complex networked system relatively tractable. During a run of the hardware the agent playback system records every external input to each agent, i.e., communication and results from external function calls. Using this information it is possible to later replay an agent, line by line of code, exactly as it occurred during the live run. Combined with the debugging software in Microsoft Visual Studio© it is possible to analyse the agent and even test small code changes without having to re-run the hardware. This strategy worked particularly well due to the modular nature of the architecture, which naturally broke the system in to manageable chunks.
Remote Start	RemoteStart was developed to facilitate large scale experimentation. External to the control system, RemoteStart is a tool that runs on every computer to allow central control over launching the host software, scheduling missions, and collecting log data after each run. It also enabled distributing code updates whenever changes were made. Though a relatively simple tool, it was essential for efficient management of a system with 10+ computers and 100s of experimental trials.

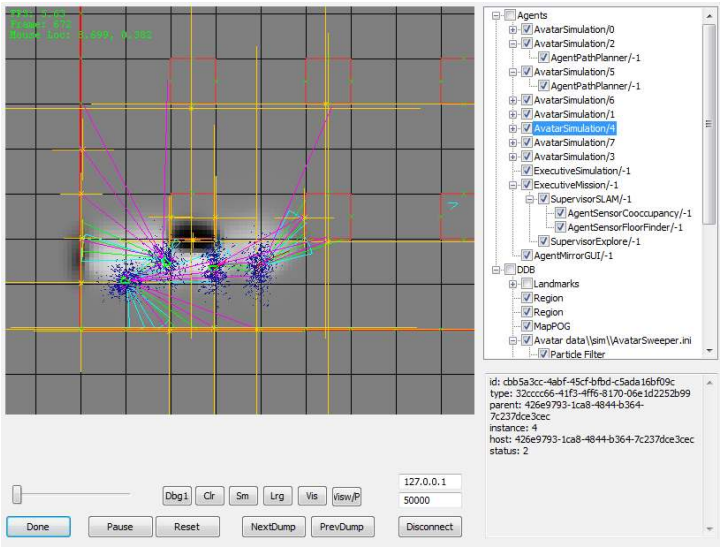


Fig. 6.1-1 Monitoring GUI. Displays maps, particle filters, avatars, and landmarks (left), as well as current agent and DDB information (right)

6.2 Experimental Scenarios and Agent Design

In order to confirm the functionality of the HAA implementation and evaluate various aspects of performance for the control system, a number of experimental scenarios and a control system capable of those tasks was required. The primary purpose of these experiments is to demonstrate and evaluate the features of HAA rather than show a capacity for any particular task, and so the chosen scenarios are those typical for mobile robotics: mapping and exploration, search and deploy, and foraging. Completing these tasks allows for significant options in control system strategy, but in following the Control *ad libitum* philosophy the guiding principles for the control system were adaptability, modularity, (support for) diversity, and persistency. To this end a set of 18 agents was designed and implemented. This section presents the details of the experimental scenarios that are used in Chapter 7, as well as the specifications for the simulated avatar hardware. An overview of the agents and a map of their interactions and dependencies is then provided, followed by an analysis of the control system structure using the relevant metrics from Chapter 3.

6.2.1 Experimental Scenarios

Three experimental scenarios where used: Mapping and Exploration, Congregate, and Forage. The scenarios, described in Table 6.2-1, are abstractions of real-world tasks that might be

completed by robot teams, and were chosen to present some variation in goals and complexity for the experiments.

The experiments took place in two arenas that are occupied by an assortment of walls and obstacles. The small arena is presented in Fig. 6.2-1, showing the locations of the five obstacles and 34 artificial landmarks along with the location of the optional congregation point used in the

Table 6.2-1 Experimental Scenarios

Scenario 1: Mapping and Exploration The most basic scenario, Mapping and Exploration requires the avatars to start in an unknown environment and explore the arena using all available sensors until the exploration threshold is reached. The threshold requires that a specified percentage of the reachable cells have high enough confidence values, specifically greater than 0.73 or less than 0.27. The initial start positions of the avatars are known, but subsequent localization occurs primarily by identifying landmarks and estimating their positions using a SLAM algorithm.
Scenario 2: Congregate The Congregate scenario begins in the same way as Mapping and Exploration and proceeds until a special "Congregation Point" landmark is found. Once the congregation point is located each avatar is assigned a position at equally spaced intervals surrounding the congregation point. The scenario ends when all avatars reach their assigned positions.
Scenario 3: Forage The Forage scenario is the most complex scenario used in these experiments. It begins in the same way as Mapping and Exploration, however a number of collectable landmarks are distributed throughout the arena. When these collectables are located an avatar must travel to the collectable, pick it up, and then deposit it in one of the specified collection regions. The scenario ends once the exploration threshold has been reached and all identified collectables have been deposited.

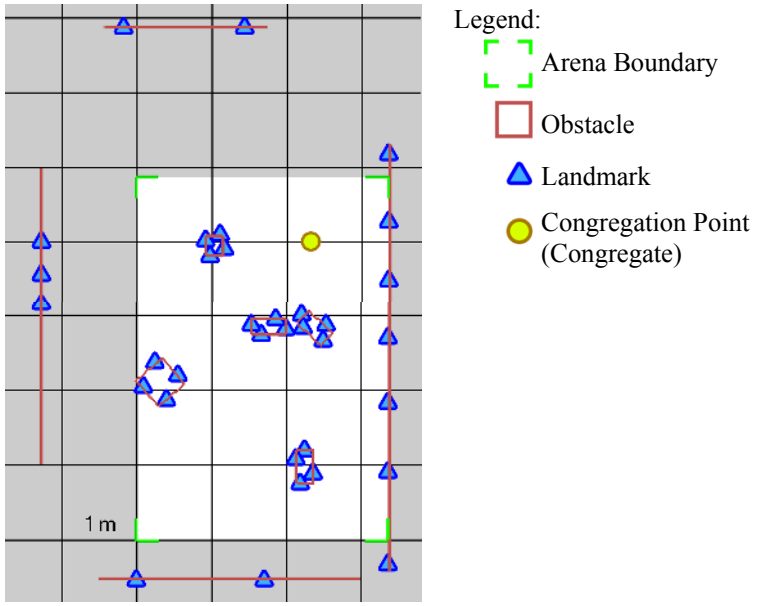


Fig. 6.2-1 Small Arena

Congregate scenario. The arena is 3.4 by 7.2 meters with an explorable area of 24.2 m^2. The large arena is shown in Fig. 6.2-2, and contains 18 obstacles and 139 landmarks. The Forage scenario has three optional collection regions and 20 collectables. The large arena is 18 by 18 meters and has an explorable area of 216 m^2.

Four different avatar types were used in these experiments. The role of each avatar type is outlined in Table 6.2-2, while Table 6.2-3 reports more detailed specifications. The specifications of the three different sensor types are provided in Table 6.2-4.

6.2.2 Agent Design

In order to control a team of avatars in completing the above scenarios a set of 18 agents was developed. To promote adaptability and modularity the required tasks and functions were broken down into components with as little interdependence as possible and virtually every component was implemented as a separate agent.

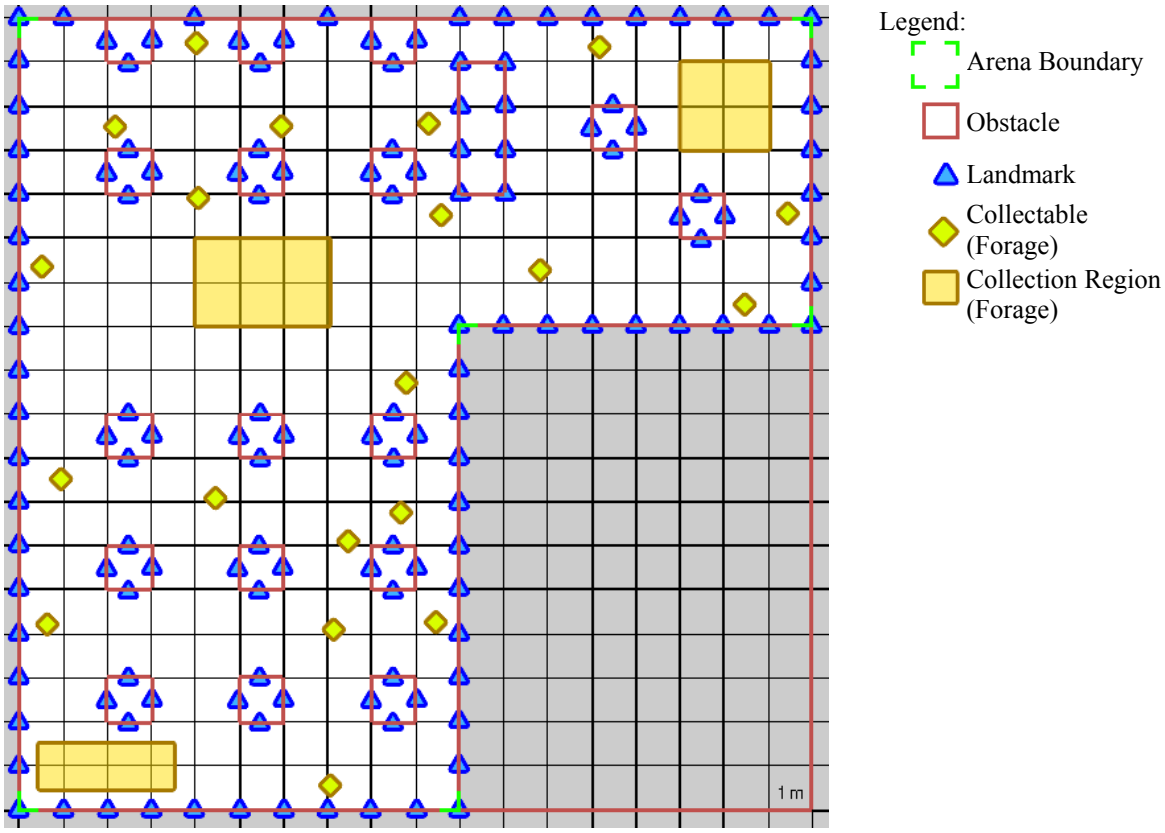


Fig. 6.2-2 Large Arena

Given the basic requirements of the control system and the three scenarios above, the functionality breakdown was as follows:

- Overall mission management
- Overall avatar resource management
- Avatar control
  - Path planning
- SLAM control
  - Sensor processing
- Exploration control
- Congregation control
- Forage control

In general each area of functionality requires little to no knowledge of the inner workings of the other components; e.g., exploration requires map data but it does not matter how it was generated,

Table 6.2-2 Avatar Roles

Seeker The Seeker avatar is small and agile but possesses limited sensing capabilities. They fill the basic role of explorers and can quickly find landmarks and collectables.
Enhanced Seeker Fills the same role as the basic Seeker but is equipped with more accurate odometry and a camera with better range and FOV. The Enhanced Seeker is used in the large arena due to the large distances travelled.
Sweeper Sweepers are slow moving but are equipped with a large sensor array that allows them to quickly generate map data for their surroundings. Their primary function is mapping and exploration, but they also have the ability to collect and deposit collectables.
Carrier The Carrier's function is to retrieve collectables. It is faster than the Sweeper but lacks the array of sensors.

Table 6.2-3 Avatar Specifications

Avatar	Speed [m/s]	Approx. Linear Drift After 10 m [m]	Approx. Angular Drift After 10 rotations [rad]	Carrying Capacity [collectables]	Sensors
Seeker	2	1.22	0.08	0	Basic Camera
Enhanced Seeker	2	0.25	0.016	0	Enhanced Camera
Sweeper	1	0.12	0.016	1	Enhanced Camera, 5 Sonar
Carrier	1.5	0.12	0.016	1	Enhanced Camera

Table 6.2-4 Sensor Specifications

Sensor	Period [s]	Field of View [rad]	Effective Range [m]	Artificial Noise* [σ]	Approx. Average Error [m]
Basic Camera	Manual	0.873	3.0	0.09	±0.14
Enhanced Camera	Manual	1.685	4.0	0.09	±0.18
Sonar	1.0	0.349	2.0	0.03	±0.03

*In most cases artificial noise is added to each sensor reading as a percentage of the reading distance following a normal distribution

and avatars require motion commands to travel from point to point but the path planning component could be interchangeable.

To meet these needs the agent classes shown in Fig. 6.2-3 were implemented. Since parent classes were used when groups of agents shared functionality, 22 classes are shown in total though ultimately only 18 distinct agent types are used. Additionally, the functionality of a number of agents were rolled into the ExecutiveMission, specifically ExecutiveAvatar and SupervisorCongregate. This was done as a simplification because these agents required minimal additional functionality, though strictly speaking for proper modularity and expandability they should be separate. An overview of each agent is provided in Appendix III, briefly describing their role and implementation, but more importantly outlining their interdependencies and recovery strategies.

Based on the agent interactions an agent dependency graph was constructed, Fig. 6.2-4. The graph allows asymmetric connections and makes a distinction between weak and strong

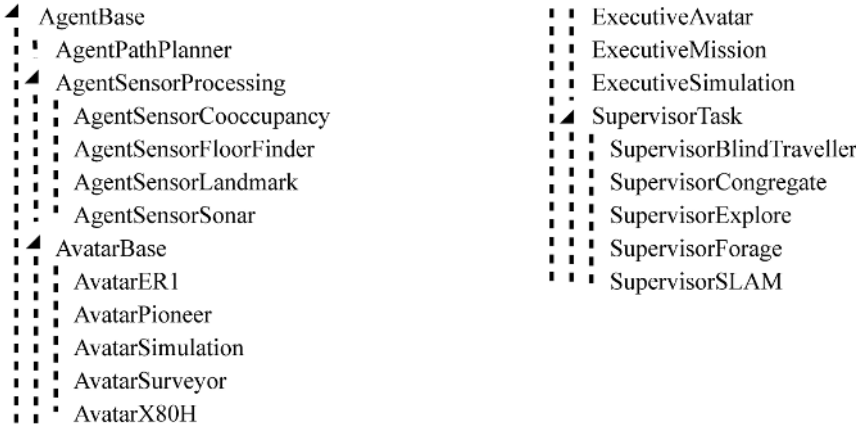


Fig. 6.2-3 Agent Class Tree

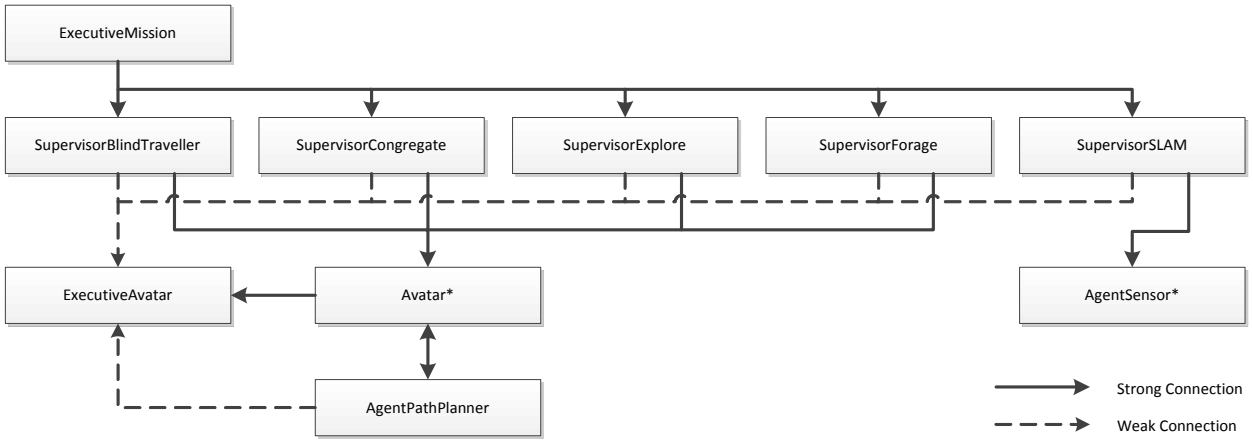


Fig. 6.2-4 Agent Dependencies

connections. A connection is considered strong if an agent requires in-depth knowledge of another agent's behaviour, particularly if they must take action during their own recovery or the recovery of the other agent. A weak connection indicates that an agent calls upon another agent through queries or task requests but otherwise has no extra knowledge of the agent.

6.2.2.1 Control System Structural Analysis

Two of the preference indexes introduced in Appendix I are used here to study the structure of the control system. First, the modularity of the agents is measured using the NZF and STD_{norm} indexes, and then a failure model of the software system is constructed to identify key points of risk.

To measure modularity the first step is to construct the DSM. Here a functionality level approach is taken, and so the elements of the DSM are the agents. The connections of the DSM are provided by the dependencies graph in Fig. 6.2-4. Strong connections are given a value of 1, and weak connections a value of 0.5. The DSM is shown in Fig. 6.2-5. With the DSM (I-6) and (I-10) can be used to calculate the NZF and STD_{norm}, respectively. An NZF of 0.17 shows that the DSM is very sparsely connected, indicating a high degree of modularity. The STD_{norm} is 0.57, which suggests the DSM has a structure between that of a bus and a chain, but leaning slightly closer to a chain.

To further study the robustness of the structure a failure model can be developed. This is done by identifying points of critical (non-recoverable) failure, and constructing a model of the dependencies that lead to those points. This method was applied to analyse the robustness of the agent structure for a hypothetical foraging scenario, with the aim of identifying key risk factors that may warrant additional consideration. In addition to the standard compliment of agents, this model assumes six Avatar* and 10 SupervisorForage agents are active. A pessimistic estimate of the agent failure rate is that for every minute of operation each agent has a 0.2% chance of failing due to an unhandled software error. This provides the foundation to build a simple failure model, though the failure model could be made more accurate if more detailed failure curves were available.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1 AgentPathPlanner						■	●	●	●	●	○						
2 AgentSensorCooccupancy																	
3 AgentSensorFloorFinder																	
4 AgentSensorLandmark																	
5 AgentSensorSonar																	
6 AvatarER1	●										●						
7 AvatarPioneer	●										●						
8 AvatarSimulation	●										●						
9 AvatarSurveyor	●										●						
10 AvatarX80H	●										●						
11 ExecutiveAvatar																	
12 ExecutiveMission													●	●	●	●	●
13 SupervisorBlindTraveller						●	●	●	●	●	○						
14 SupervisorCongregate						●	●	●	●	●	○						
15 SupervisorExplore						●	●	●	●	●	○						
16 SupervisorForage						●	●	●	●	●	○						
17 SupervisorSLAM		●	●	●	●						○						

■ Strong Connection, ○ Weak Connection

Fig. 6.2-5 Control System DSM

The complete failure model is shown in Fig. 6.2-6, while a sample of the individual agent failure continuous distribution function is shown in Fig. 6.2-7. This model identifies three paths to critical failure:

1. Any key agent suffers critical failure. ExecutiveMission, ExecutiveAvatar, SupervisorExplore, or SupervisorSLAM.
2. More than two SupervisorForage agents suffer critical failure, meaning that less than 80% of the collectables are gathered.
3. All avatars (via Avatar* or AgethPathPlanner) suffer critical failure.

It appears that the avatar failure path is low risk because of the redundancy, while the SupervisorForage failure path is quite significant even though up to two agents are allowed to fail. In order to identify problem areas and efficiently allocate development resources, a simple test can be done by suppressing the failure of various agents. Fig. 6.2-8 shows the critical failure continuous distributions for four scenarios: current behaviour, no avatar failure, no foraging failure, and no key agent failure. Suppressing avatar failure has no distinguishable impact on the curve, and so development resources are likely better spent in other areas. No foraging failure has less of an impact than suppressing key agent failure, but only requires focusing resources on a single agent. Alternatively, focusing on the key agents could yield significant gains.

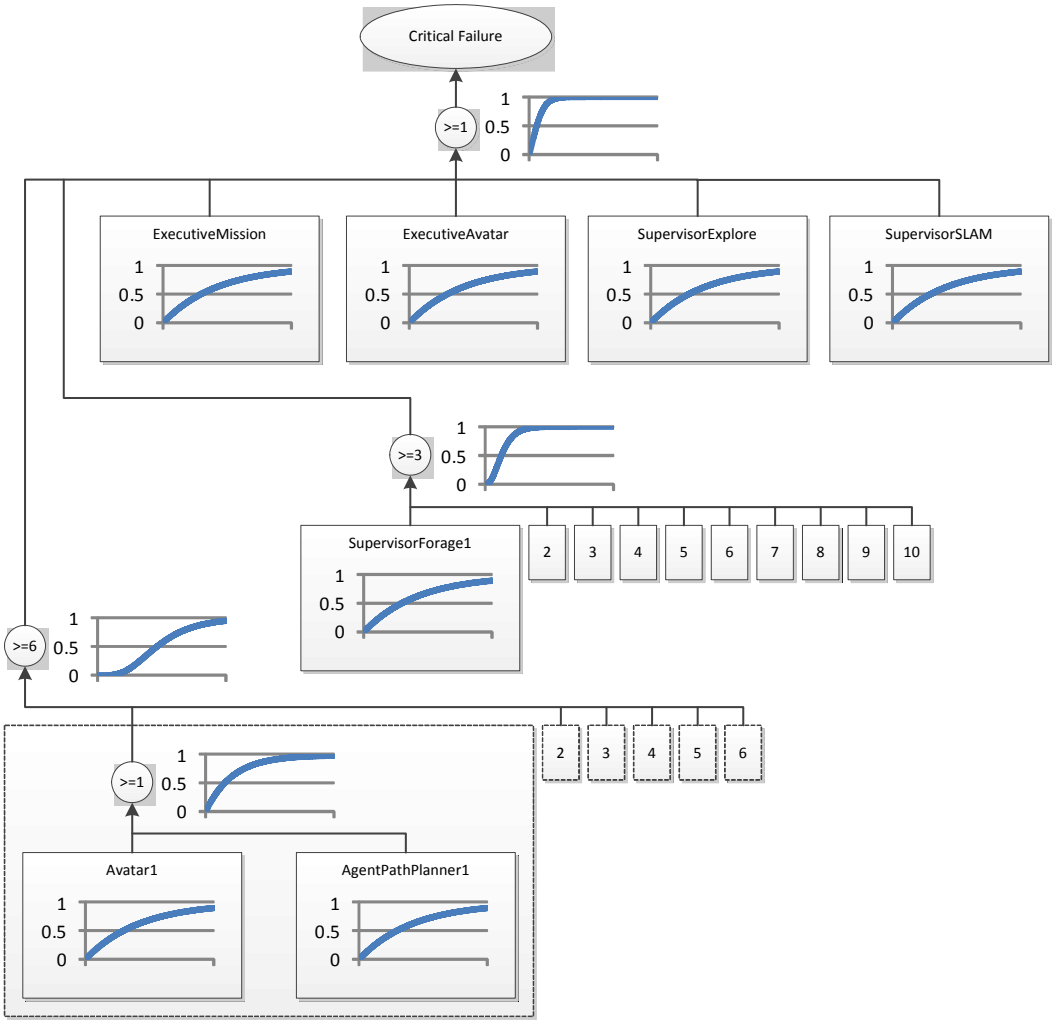


Fig. 6.2-6 Foraging Scenario Failure Model

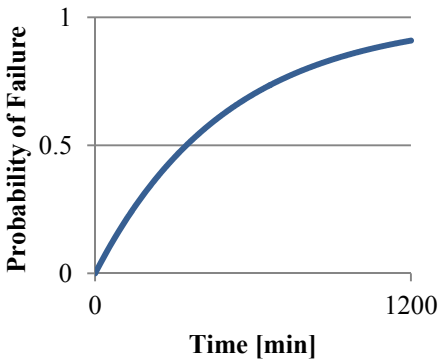


Fig. 6.2-7 Agent Failure Curve

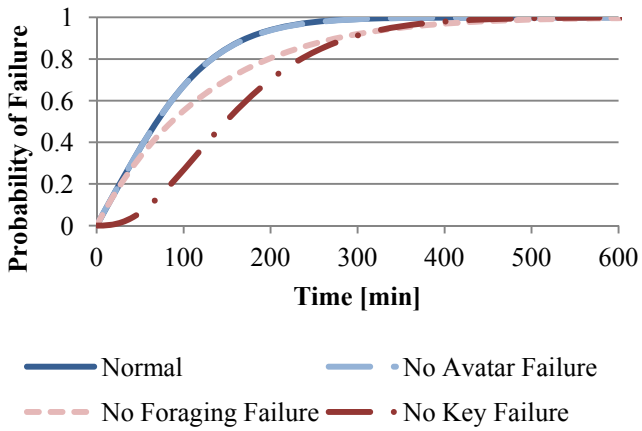


Fig. 6.2-8 Critical Failure Comparisons

Chapter 7
Hardware-in-the-Loop Experimentation and Results

Hardware-in-the-Loop (HIL) simulation is a valuable experimentation and design tool [94]. For this thesis, where the control system is the primary focus, the setup consists of the complete control system operating normally, however, instead of interfacing with a team of physical avatars the control system communicates with simulated avatars. This is accomplished by simply replacing the standard Avatar agents, who communicate with the avatar hardware, with agents that communicate with a computer simulation, while virtually every other agent remains unchanged. The simulation models avatars, sonar sensors, "visual sensors" that simulate detecting visual landmarks using a camera, and a greatly simplified collection system to approximate picking up and dropping objects. An appropriate noise signal was added to most inputs, including the avatar odometry model, to simulate the behaviour of the real hardware, the full details of which were described in Chapter 6. The HIL simulation has three major benefits for this research. The first obvious benefit over full simulation is that the real control system is used and its true behaviour can be observed. Secondly, experimentation avoids the complications of physical robot hardware, such as repositioning avatars before each run and recharging batteries, which makes running and repeating numerous experiments much more efficient. Thirdly, since the avatars are simulated the number and capabilities of the avatars are flexible, and thus the complexity of the experiments can be much higher than when working with physical hardware.

Building from the experimental scenarios described in Chapter 6, this chapter details a series of 10 experiments. The experiments are ordered by increasing complexity, and each one is designed to explore a different aspect of the control system. Each experiment was repeated 10 times, however, in most cases it is more convenient to discuss the results from a single run. And so unless otherwise noted, figures present only the results from a representative run of each experiment and the average results over multiple runs are left for the experiment summary tables. The summary tables also specifically report the representative run results in order to prove that their performance is truly representative of the average run. The remainder of this section details the experimental setup, while each subsequent section presents the results of the 10 experiments. The experiments are divided into three sections: 1) Architecture Functionality, where the core

features of the HAA architecture are demonstrated; 2) Algorithm Performance, where the distributed algorithms are tested with various system sizes and the JC-SLAM algorithm is evaluated against traditional SLAM approaches; and 3) Robustness, where robustness against both software and hardware failure is examined. A selection of these results have been submitted for publication in [95].

The experimental setup consisted of 10 computers networked using a 100 Mbps router. The computers had dual-core 2.0 GHz processors and 2 GB RAM, and were running Windows Server 2003. The average network delay under low load conditions was <1 ms. One of the computers was dedicated to running the simulation of the avatar hardware, while the others were used to supply the number of hosts dictated by each experiment. Each experiment began from a blank slate, and hosts had to start, locate other hosts to form the host group, and then launch the required agents. Once the assigned tasks were finished the hosts would gracefully shut down all agents and exit, at which point the experiment was considered complete. Because significant effort was made to ensure each agent was bug-free and hardware crashes are extremely rare, agent and host crashes had to be simulated. This was done by specifying a minimum and maximum operating time for each agent and host, and at a random time within that window the agent would become unresponsive for 30 seconds and then shut down. Though the maximum number of hosts in any experiment was 10, no experiment required more than eight hosts simultaneously, and so when additional hosts joined they were started on hosts that had already "crashed," thus ensuring that hosts always had exclusive access to the computer resources. An additional computer was used to run the GUI and monitor the experiments. During each experiment each agent logged their operations, and hosts periodically dumped data and statistics for later analysis.

7.1 Architecture Functionality

The following experiments were conducted to demonstrate the basic functionality of the HAA architecture:

1. *Experiment AF-1 – Exploration and Mapping:* To demonstrate the basic functionality of the system: the dynamic control system, dynamic team formation, and the dynamic host network.

2. *Experiment AF-2 – Congregate:* To demonstrate basic avatar allocation with multiple task supervisors in the Congregate scenario.

3. *Experiment AF-3 – Forage:* To demonstrate advanced avatar allocation and heterogeneity in the Forage scenario.

7.1.1 Experiment AF-1 – Mapping and Exploration

The purpose of Experiment AF-1 is to demonstrate the basic functionality of the control system. This primarily focuses on four features: 1) the ability to dynamically form the host group, 2) the ability to dynamically form the control system by spawning agents as they are required, 3) the ability to balance processor resources through agent distribution, and 4) the ability to manage avatar resources as they join and leave the system. This all occurs during normal operation of the control system, and agent transfers and host, avatar, or agent retirements are handled gracefully with no loss of information.

This experiment was conducted in the small arena. There were initially four hosts, every odd minute one host gracefully retires and every even minute a host joins. The mission started with three seekers, after 2 minutes one seeker retires and after 2.5 minutes two additional seekers join. The mission completion condition was that 95% of the reachable cells were explored, where "explored" is defined as a cell with a value greater than 0.73 or less than 0.27. Fig. 7.1-1 shows the mapping result, where each of the five curved lines represents the path of an avatar, while Table 7.1-1 reports the experiment summary. The average completion time for the mission was 3.8 minutes, and the mapping accuracy averaged 91% with 0.793% STD. Map coverage and map accuracy are calculated as percentages relative to an "ideal" map generated with zero localization error.

$$map\ coverage = \frac{\sum_{mission\ region}|c - 0.5|}{\sum_{mission\ region}|c_{ideal} - 0.5|} \tag{7.1-1}$$

$$map\ accuracy = 1 - \frac{\sum_{mission\ region}|c - c_{ideal}|}{\#\ of\ cells} \tag{7.1-2}$$

where c is the value of a cell, c_{ideal} is the value of the corresponding cell from the ideal map, and the sum is taken over all cells within the mission region.

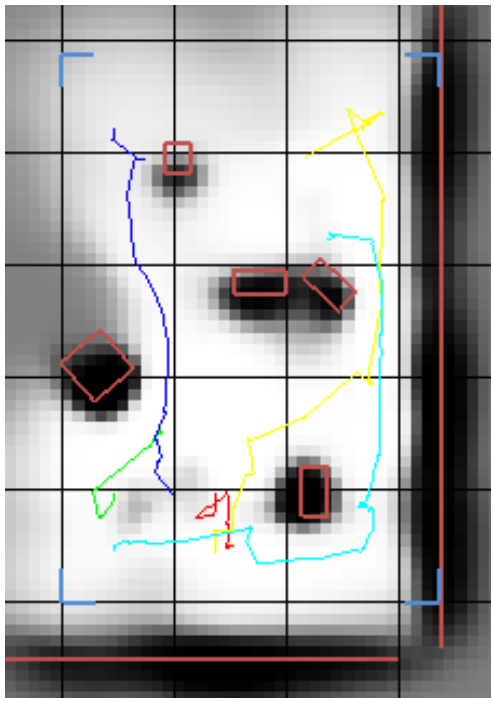


Fig. 7.1-1 Experiment AF-1 Mapping Result. True obstacle positions are indicated with thick lines, and avatar paths are shown with thin curved lines.

Table 7.1-1 Experiment AF-1 Results Summary

Title	Representative Run	Mean	RMS	STD
Mission Duration [min]	3.802	3.871	3.884	0.324816
Map Coverage [%]*	90.38	91.07	91.10	2.247400
Map Accuracy [%]*	90.64	91.19	91.19	0.793000
Localization Positional Err [m]	0.07414	0.05737	0.05947	0.015661
Localization Rotational Err [rad]	0.05221	0.04665	0.05290	0.024944
Landmark Err [m]	0.07977	0.07606	0.07646	0.007790
Landmark Covariance [m]	0.40481	0.40241	0.41953	0.118617
Average # Hosts	4.50	4.52	4.52	0.031171
Average # Agents	12.83	13.00	13.01	0.396887
Average CPU Usage [%]	29.64	28.76	28.82	1.771400
DDB Size [MB]	9.179	9.475	9.520	0.919952

* Relative to a fully explored map generated with no localization error

A visualization of the agent allocation process is presented in Fig. 7.1-2. This visualization is crucial in understanding the behind the scenes activity of the control system, since the actions of the avatars and visible mapping/exploration activities paint only a small part of the picture. In order to facilitate understanding of the agent allocation figures, the format is explained in some detail here, and Fig. 7.1-3 provides a step-by-step transcript of the events.

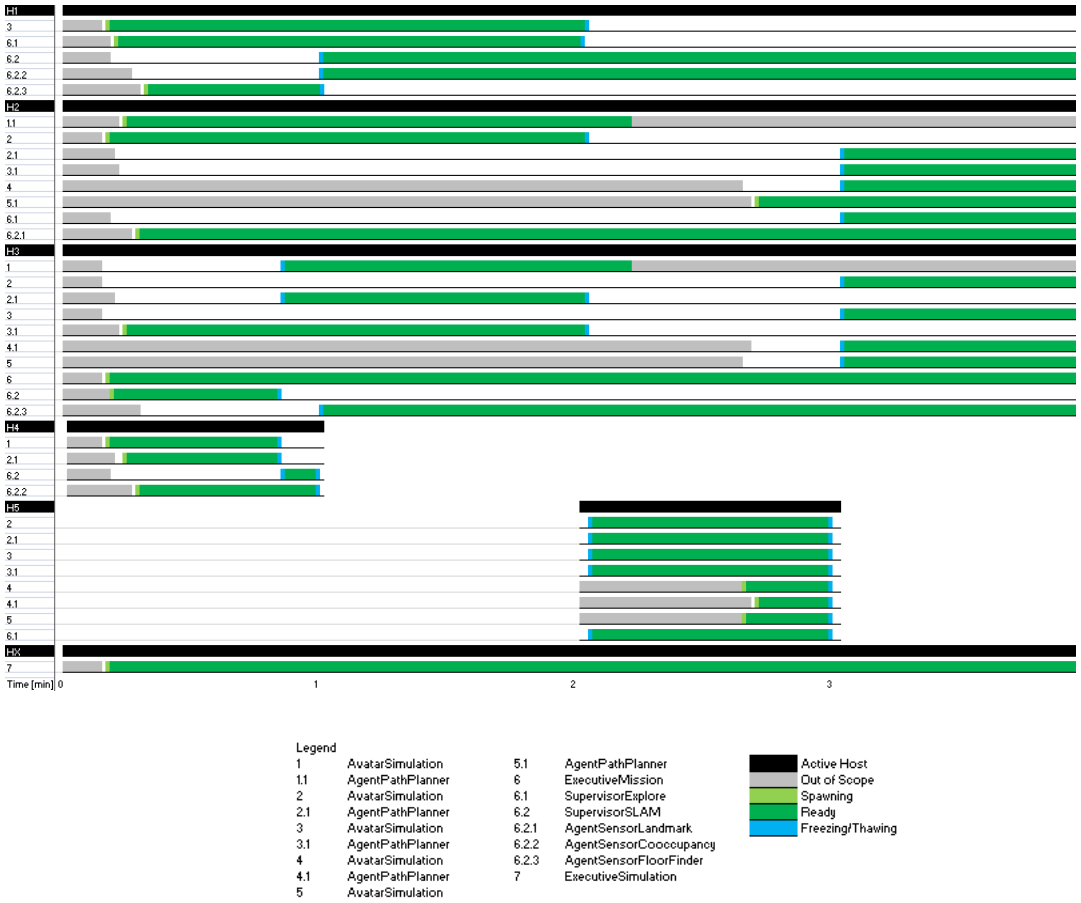


Fig. 7.1-2 Experiment AF-1 Agent Allocation

Fig. 7.1-2 is organized vertically by host and the horizontal axis represents time, starting from the moment the first host was launched and ending when the mission is complete. Many hosts are active for the entirety of the mission, however some hosts can join late and/or end early, e.g., H4 and H5 in this experiment. Each host lists every agent it has ever instantiated, and the agents are labeled based on their parentage, for example, A1.1 AgentPathPlanner is a child of A1 AvatarSimulation. Active agents are shown in green on their current host. Transition states such as spawning, freezing/thawing, or crashed, are indicated in their corresponding colour. Grey areas indicate that the agent has not been requested yet or is no longer needed, and is therefore out of scope. Following along with Fig. 7.1-3, Fig. 7.1-2 can be understood as follows:

- At 00:01 the initial host group forms.
- At 00:10 the mission starts and the first agents are requested based on the mission definition, specifically ExecutiveMission and three AvatarSimulations.

00:01 H1 joined
00:01 H2 joined
00:01 HX joined
00:01 H3 joined
00:01 H4 joined
00:10 Mission started
00:10 A7 (ExecutiveSimulation) requested by H1
00:10 A6 (ExecutiveMission) requested by H1
00:10 A2 (AvatarSimulation) requested by H1
00:10 A3 (AvatarSimulation) requested by H1
00:10 A1 (AvatarSimulation) requested by H1
00:11 A3 (AvatarSimulation) spawned by H1
00:12 A6 (ExecutiveMission) spawned by H3
00:12 A7 (ExecutiveSimulation) spawned by HX
00:12 A1 (AvatarSimulation) spawned by H4
00:12 A2 (AvatarSimulation) spawned by H2
00:12 A6.2 (SupervisorSLAM) requested by A6 (ExecutiveMission)
00:12 A6.1 (SupervisorExplore) requested by A6 (ExecutiveMission)
00:13 A6.2 (SupervisorSLAM) spawned by H3
00:13 A6.1 (SupervisorExplore) spawned by H1
00:14 A2.1 (AgentPathPlanner) requested by A2 (AvatarSimulation)
00:14 A1.1 (AgentPathPlanner) requested by A1 (AvatarSimulation)
00:15 A3.1 (AgentPathPlanner) requested by A3 (AvatarSimulation)
00:16 A3.1 (AgentPathPlanner) spawned by H3
00:16 A2.1 (AgentPathPlanner) spawned by H4
00:16 A1.1 (AgentPathPlanner) spawned by H2
00:18 A6.2.1 (AgentSensorLandmark) requested by A6.2 (SupervisorSLAM)
00:18 A6.2.2 (AgentSensorCooccupancy) requested by A6.2 (SupervisorSLAM)
00:19 A6.2.2 (AgentSensorCooccupancy) spawned by H4
00:19 A6.2.1 (AgentSensorLandmark) spawned by H2
00:20 A6.2.3 (AgentSensorFloorFinder) requested by A6.2 (SupervisorSLAM)
00:21 A6.2.3 (AgentSensorFloorFinder) spawned by H1
00:51 A1 (AvatarSimulation) freezing on H4
00:51 A6.2 (SupervisorSLAM) freezing on H3
00:51 A2.1 (AgentPathPlanner) freezing on H4
00:53 A1 (AvatarSimulation) thawed by H3
00:53 A6.2 (SupervisorSLAM) thawed by H4
00:53 A2.1 (AgentPathPlanner) thawed by H3
01:00 A6.2 (SupervisorSLAM) freezing on H4
01:00 A6.2.2 (AgentSensorCooccupancy) freezing on H4
01:01 H4 removed
01:01 A6.2.3 (AgentSensorFloorFinder) freezing on H1
01:02 A6.2.2 (AgentSensorCooccupancy) thawed by H1
01:02 A6.2 (SupervisorSLAM) thawed by H1
01:02 A6.2.3 (AgentSensorFloorFinder) thawed by H3
02:01 H5 joined
02:03 A6.1 (SupervisorExplore) freezing on H1
02:03 A2 (AvatarSimulation) freezing on H2
02:03 A3.1 (AgentPathPlanner) freezing on H3
02:03 A2.1 (AgentPathPlanner) freezing on H3
02:03 A3 (AvatarSimulation) freezing on H1
02:05 A3.1 (AgentPathPlanner) thawed by H5
02:05 A2 (AvatarSimulation) thawed by H5
02:05 A3 (AvatarSimulation) thawed by H5
02:05 A2.1 (AgentPathPlanner) thawed by H5
02:05 A6.1 (SupervisorExplore) thawed by H5
02:13 A1.1 (AgentPathPlanner) removed
02:13 A1 (AvatarSimulation) removed
02:40 A5 (AvatarSimulation) requested by H1
02:40 A4 (AvatarSimulation) requested by H1
02:41 A4 (AvatarSimulation) spawned by H5
02:41 A5 (AvatarSimulation) spawned by H5
02:42 A4.1 (AgentPathPlanner) requested by A4 (AvatarSimulation)
02:43 A5.1 (AgentPathPlanner) requested by A5 (AvatarSimulation)
02:44 A4.1 (AgentPathPlanner) spawned by H5
02:44 A5.1 (AgentPathPlanner) spawned by H2
03:00 A4.1 (AgentPathPlanner) freezing on H5
03:00 A4 (AvatarSimulation) freezing on H5
03:00 A2 (AvatarSimulation) freezing on H5
03:00 A6.1 (SupervisorExplore) freezing on H5
03:00 A3.1 (AgentPathPlanner) freezing on H5
03:00 A5 (AvatarSimulation) freezing on H5
03:00 A3 (AvatarSimulation) freezing on H5
03:00 A2.1 (AgentPathPlanner) freezing on H5
03:02 H5 removed
03:04 A4.1 (AgentPathPlanner) thawed by H3
03:04 A4 (AvatarSimulation) thawed by H2
03:04 A2 (AvatarSimulation) thawed by H3
03:04 A5 (AvatarSimulation) thawed by H3
03:04 A3 (AvatarSimulation) thawed by H3
03:04 A3.1 (AgentPathPlanner) thawed by H2
03:04 A2.1 (AgentPathPlanner) thawed by H2
03:04 A6.1 (SupervisorExplore) thawed by H2
03:58 Mission finished

Fig. 7.1-3 Experiment AF-1 Mission Transcript

- These agents request additional agents (e.g., SupervisorExplore, AgentPathPlanner, AgentSensorLandmark) as required and by 00:21 the control system is fully operational.
- Around 00:51 an agent allocation session occurs and A1, A2.1, and A6.2 change hosts in order to balance processor load.
- At 01:00 H4 freezes all its agents and gracefully leaves the host group.
- At 02:01 H5 joins the group and takes responsibility for several agents.
- At 02:13 one avatar leaves, and the corresponding agents, A1 and A1.1, are shut down.
- At 02:40 two additional avatars become available and H1 requests two more AvatarSimulation agents.

- Operation continues with agents occasionally being transferred to balance load, H5 leaves the group at 03:02.
- At 03:58 the mission criteria are met and the mission is complete.

This experiment demonstrates all the basic features of the control system, including adding and removing hosts, adding and removing avatars, dynamically spawning agents as required by the mission, and transferring agents between hosts. Several other points of interest are presented in the following graphs. Fig. 7.1-4 shows how the number of hosts and agents developed over time; the number of agents dips at 140 s and jumps at 170 s corresponding to the removal and addition of avatars at those times. Fig. 7.1-5 shows the development of map coverage and map accuracy. A totally blank map with no information has a nominal accuracy of 0.5 since every cell has a value of "unknown." As information is added, corresponding to increasing map coverage, the accuracy changes, in this case improving because the majority of the information added to the map is correct.

Both positional and rotational localization error are shown in Fig. 7.1-6. The values fluctuate as the avatar odometry drifts and sensor readings are processed to compensate, but remain relatively stable over time at 0.06 m and 0.04 rad, respectively. Given the relatively short duration of the mission the increase in positional error near the end of the mission might indicate that error had not settled, however, the later missions with much longer duration also demonstrate eventual stability of the localization.

Fig. 7.1-7 shows the growth of the DDB over time. The figure was split into two parts because the size of the particle filters dominates the DDB and would make the other elements impossible

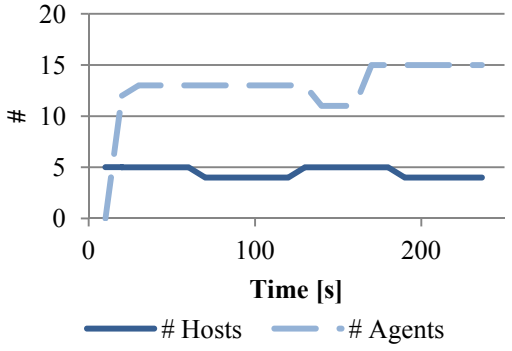


Fig. 7.1-4 Experiment AF-1 Hosts and Agents

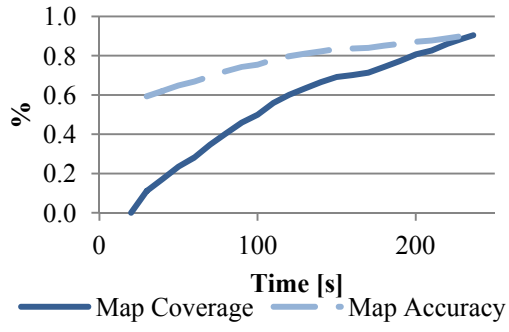


Fig. 7.1-5 Experiment AF-1 Map Coverage and Accuracy

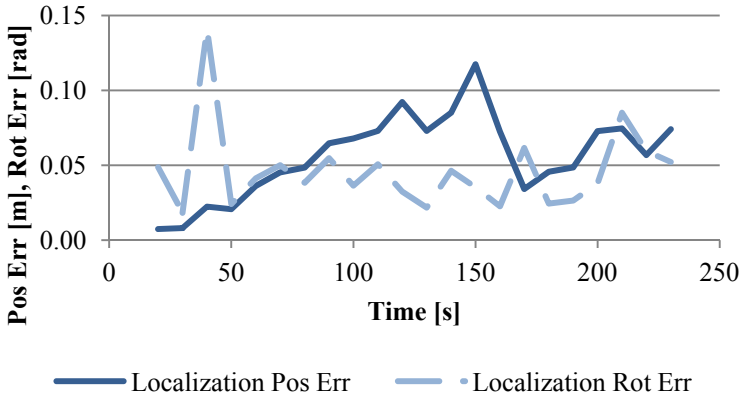
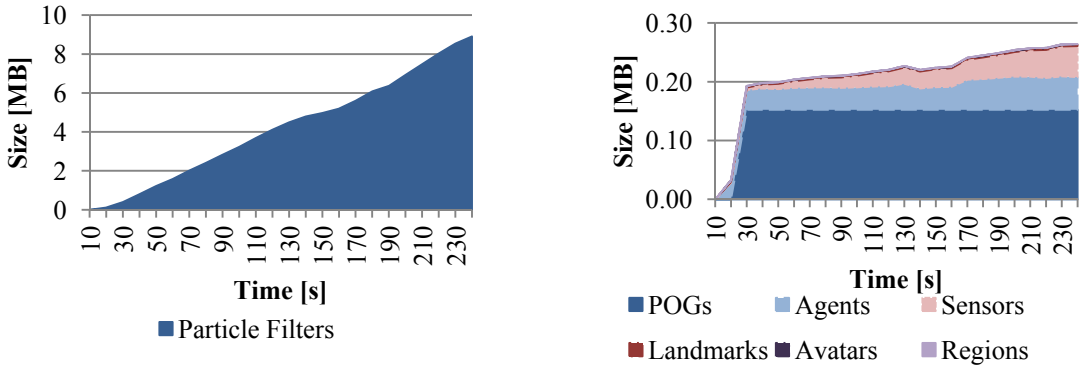


Fig. 7.1-6 Experiment AF-1 Localization Error



(a) (b)

Fig. 7.1-7 Experiment AF-1 DDB Distribution

to distinguish. The particle filter size grows linearly as more predictions are added to the DDB, and does not stabilize because the mission ends before the DDB starts discarding old predictions. The size of the POG stabilizes quickly once the first map updates are processed. The size of the agents is also relatively stable. The size of the sensor readings naturally grows as more readings are added, and the remaining items take a negligible amount of space.

A breakdown of the system wide processor usage is shown in Fig. 7.1-8. This figure simply adds the usage of all hosts and is not normalized, and so for example: a group for five hosts has a maximum usage of 5 and a usage of 1 would correspond to 20% average usage. However, the primary interest of this figure is to understand which agents require the most processing. The activities of the hosts themselves, including managing the DDB, conducting agent allocation, and forwarding messages, account for a significant 30-40% of the processing. The various sensor processing agents are also large consumers, accounting for roughly 50% of the processing at any

time. Path planners are the only other notable agent at 10%, while all the remaining agents combined typically require less than 10%. Fig. 7.1-9 demonstrates how a relatively even processor load between hosts is maintained, despite two major factors that limit the amount of balancing which can occur. Specifically, i) since the usage of each agent is set and there are a finite number of agents, achieving perfect balance is impossible, and ii) load balancing is only one of the optimization criteria of the agent allocation algorithm, since network traffic and a transfer penalty are also taken into account. Fig. 7.1-9(a) shows the usage of each host over time, while Fig. 7.1-9(b) shows the more pertinent STD in CPU usage between hosts, averaging 9% deviation.

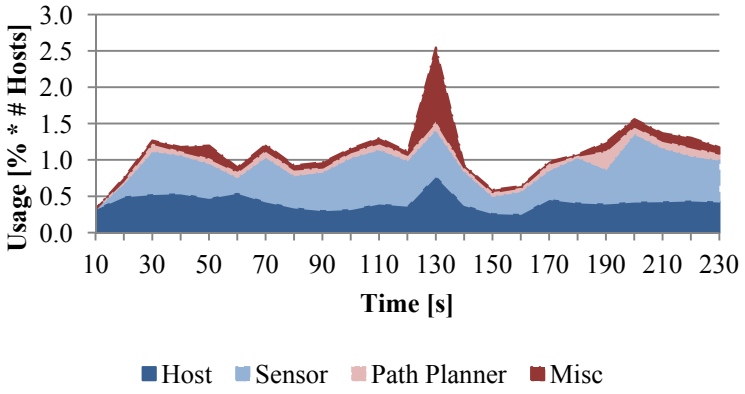
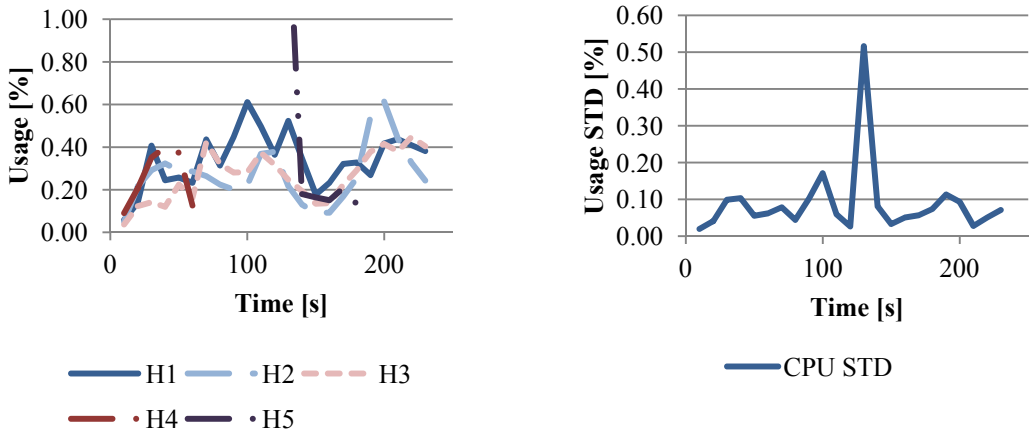


Fig. 7.1-8 Experiment AF-1 Processor Usage Breakdown



(a) Total CPU Usage by Host (b) Total CPU STD

Fig. 7.1-9 Experiment AF-1 CPU Balancing

7.1.2 Experiment AF-2 – Congregate

Experiment AF-2 is a straightforward demonstration of allocating avatar resources between multiple task supervisor agents. The experiment takes place in the small arena with three seekers and four hosts. A special landmark was placed in the arena, and when the landmark is found the avatars move to positions surrounding the landmark at equally spaced intervals. The mission is complete once all avatars reach their designated positions. Avatar allocation works on a task priority bidding system: when a supervisor requires avatar resources they place a bid on the desired avatar, and the supervisor with the highest bid gains control of the avatar. In this case SupervisorExplore by default places a low priority bid on all avatars, but once the congregation point is found SupervisorCongregate places high priority bids and gains control of each avatar (due to the simplicity of the task, the functionality of SupervisorCongregate was implemented directly within ExecutiveMission).

The results are reported in Table 7.1-2. Map coverage drops to 59% since exploration is abandoned as soon as the congregation point is located. This is readily apparent in Fig. 7.1-10, which shows the mapping result from one run as well as the final positions of the avatars around the congregation point. Map accuracy is also lower at 79%, due to the incompleteness of the map, not due to any increased inaccuracy of the portions that were generated. This is confirmed

Table 7.1-2 Experiment AF-2 Results Summary

Title	Representative Run	Mean	RMS	STD
Mission Duration [min]	2.974	3.102	3.201	0.788482
Map Coverage [%]*	57.32	58.96	59.18	4.993000
Map Accuracy [%]*	78.34	78.87	78.89	1.775000
Localization Positional Err [m]	0.02808	0.07489	0.07899	0.025108
Localization Rotational Err [rad]	0.01011	0.03719	0.04106	0.017404
Landmark Err [m]	0.04999	0.06436	0.06577	0.013556
Landmark Covariance [m]	0.50794	0.65200	0.67516	0.175332
Average # Hosts	5.00	5.00	5.00	0.000000
Average # Agents	12.32	12.41	12.41	0.355753
Average CPU Usage [%]	16.25	16.28	16.41	2.085000
DDB Size [MB]	5.973	5.453	5.532	0.928745

* Relative to a fully explored map generated with no localization error

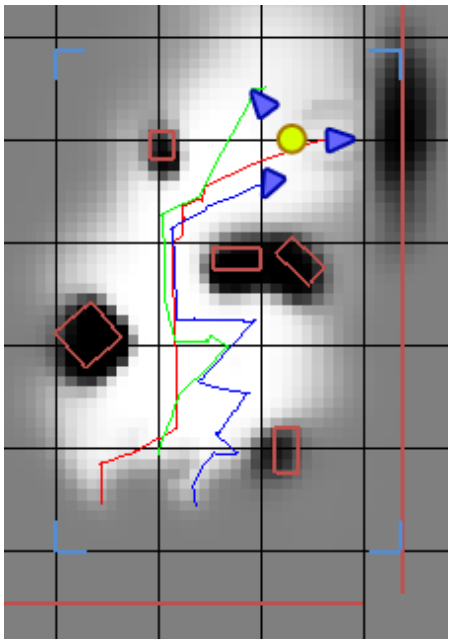


Fig. 7.1-10 Experiment AF-2 Mapping Result

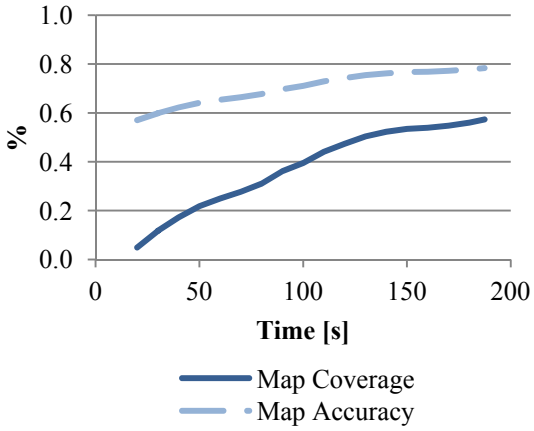


Fig. 7.1-11 Experiment AF-2 Map Coverage and Accuracy

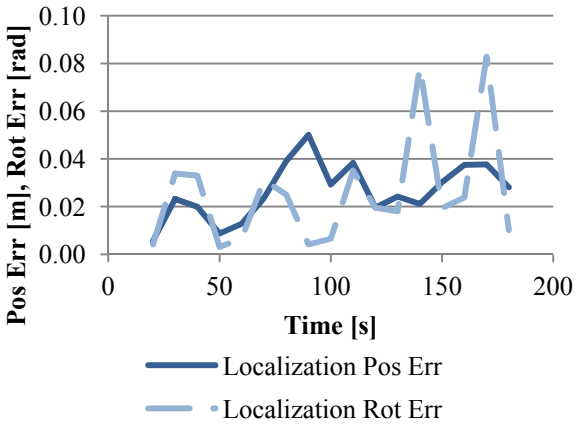


Fig. 7.1-12 Experiment AF-2 Localization Error

by noting that the localization error is still within the expected range, with averages of 0.075 m and 0.037 rad. The change over time of these values is shown in Fig. 7.1-11 and Fig. 7.1-12.

7.1.3 Experiment AF-3 – Forage

In the foraging scenario a number of collectables are placed at unknown locations in the arena, and the goal is to locate all the collectables and return them to predefined collection regions. When a collectable is located a SupervisorForage agent is spawned, who is charged with controlling an avatar to move to the collectable, pick it up, move to the nearest collection region,

and drop off the collectable. Once the collectable has been dropped off the SupervisorForage agent is finished and retires.

This scenario has dynamic integration of heterogeneous avatars and both active and passive cooperation. However, instead of just two simultaneous tasks there are many simultaneous tasks competing for avatar resources. This advanced avatar allocation uses the same bidding process, but allows SupervisorForage agents to adjust their priority based on the distance between the avatar and the collectable.

The mission takes place in the large arena with 20 collectables distributed throughout the arena and three collection regions. The host group always consists of eight hosts, and four sweeper and two carrier avatars are used. The mission completion criteria are that 95% of the reachable cells are explored, and all identified collectables have been deposited.

The mapping result from one run of the experiment is shown in Fig. 7.1-13, and the avatar paths

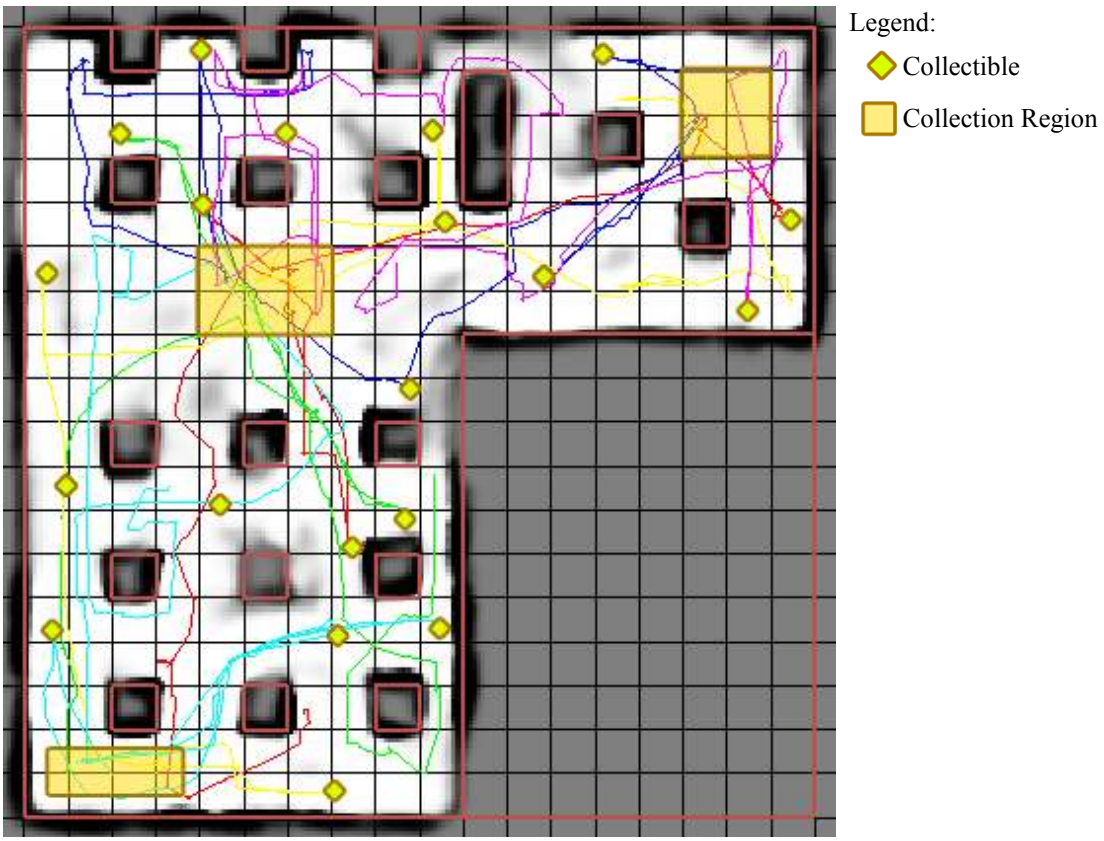


Fig. 7.1-13 Experiment AF-3 Mapping Result

exhibit much more backtracking than in the Mapping and Exploration scenario as they travel between collectables and the collection regions. The results summary is provided in Table 7.1-3. The average mission duration was 28.6 minutes, and the map accuracy of 91% and localization error of 0.105 m/0.024 rad are within expectations. In all cases every collectable was found and deposited. Fig. 7.1-14 shows the typical processor usage breakdown, which is expected since despite the large number of SupervisorForage agents they require very little processing power.

The activity in the agent allocation graph in Fig. 7.1-15 becomes interesting, particularly for the SupervisorForage agents, A7.2-A7.21. There are frequent agent spawns and semi-regular retirements, with the typical SupervisorForage taking between 3 to 10 minutes to complete its

Table 7.1-3 Experiment AF-3 Results Summary

Title	Representative Run	Mean	RMS	STD
Mission Duration [min]	28.963	28.599	28.683	2.203573
Map Coverage [%]*	98.32	97.83	97.83	0.755300
Map Accuracy [%]*	91.52	91.15	91.16	1.291200
Localization Positional Err [m]	0.08069	0.10553	0.11601	0.048174
Localization Rotational Err [rad]	0.01906	0.02403	0.02609	0.010175
Landmark Err [m]	0.09601	0.09915	0.10426	0.032232
Landmark Covariance [m]	0.05503	0.05333	0.05337	0.002213
Average # Hosts	9.00	9.00	9.00	0.000000
Average # Agents	29.95	29.48	29.49	0.822289
Average CPU Usage [%]	55.52	53.84	53.87	1.586500
DDB Size [MB]	26.122	25.384	25.420	1.353034
Cargo Collected	20.00	20.00	20.00	0.000000

* Relative to a fully explored map generated with no localization error

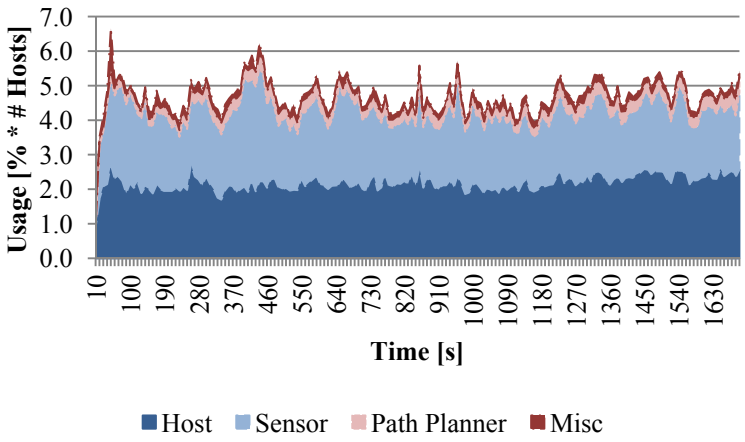


Fig. 7.1-14 Experiment AF-3 Processor Usage Breakdown

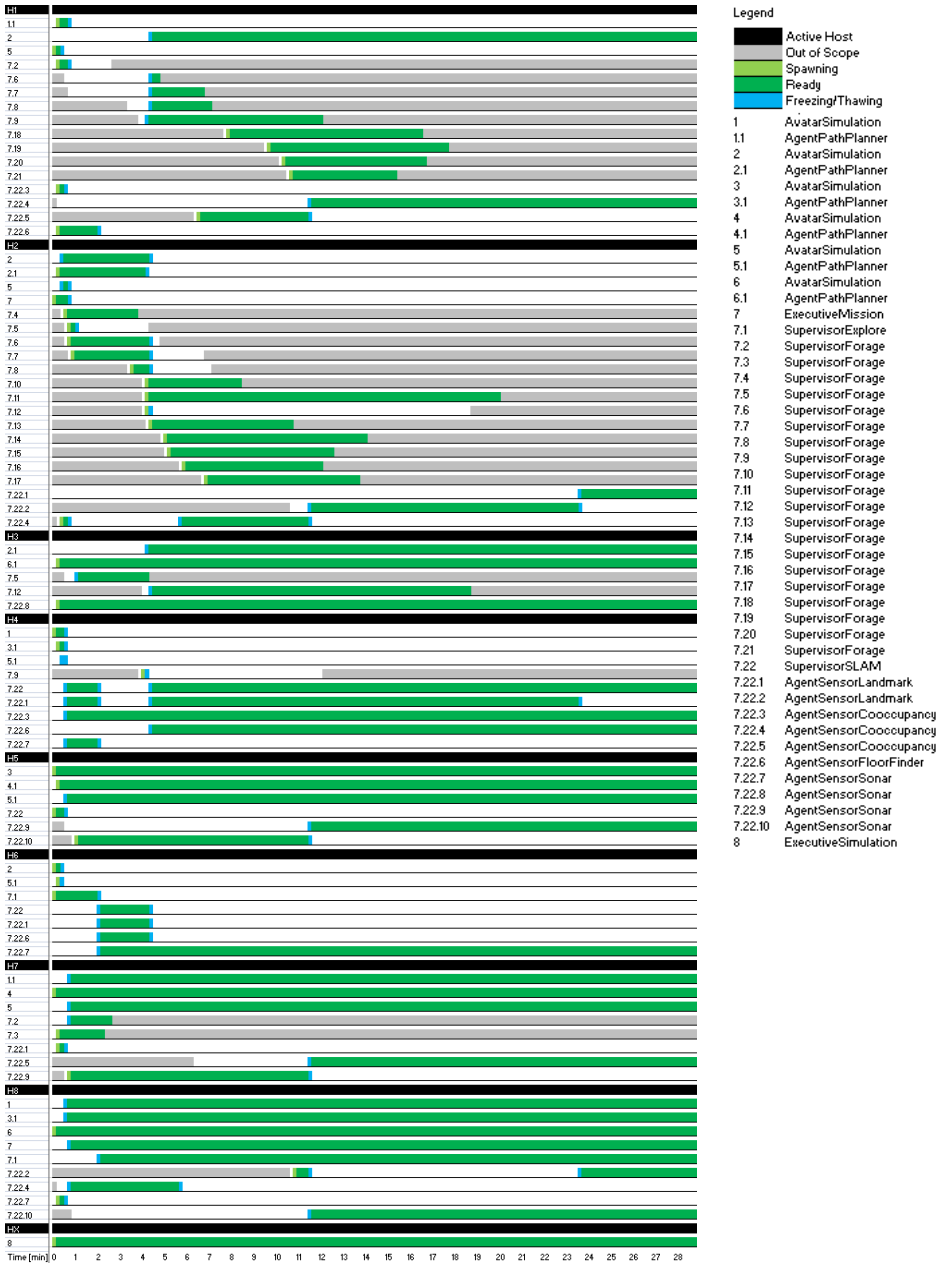


Fig. 7.1-15 Experiment AF-3 Agent Allocation

task. The variation in completion time is a result of three primary factors: i) how long it takes the Supervisor to win control of an avatar, ii) how well explored the map is between the avatar and the collectable, and iii) how much traffic there is to complicate the avatar's path, particularly around busy collection regions.

7.2 Algorithm Performance

This set of experiments evaluates the relevant metrics for the specific algorithms used for this HAA implementation:

1. *Experiment AP-1 – Ordered Atomic Commit:* To examine the performance of the algorithm with various numbers of participants.
2. *Experiment AP-2 – Host Membership:* To measure the time required for Join, Leave, and Remove actions.
3. *Experiment AP-3 – Agent Allocation:* To study the two key metrics of the algorithm: allocation time, and number of messages sent.
4. *Experiment AP-4 – Agent Transfer and Recovery:* To measure the time required for Spawn, Freeze/Transfer, and Recover actions.
5. *Experiment AP-5 – JC-SLAM:* To compare the performance of JC-SLAM versus the two traditional SLAM approaches: Delay and Discard.

7.2.1 Experiment AP-1 – Ordered Atomic Commit

The Ordered Atomic Commit algorithm is used any time the system must reach consensus and maintain consistency. Virtually all the later algorithms rely on OAC and it is used to synchronize the DDB across all hosts. The following results were derived from the algorithm being used in live conditions; specifically, the samples were gathered from all the experiments run in the other sections. Each graph shows the mean value from all the samples, and uses error bars to indicated standard deviation. Since the samples were not explicitly controlled the number of samples varies from case to case, but is at minimum 124 (for the 2-3 participant cases), and averages over 600,000. Fig. 7.2-1 plots the variation in the time taken to decide and deliver a message with different numbers of participants. The average delay appears to be stable, suggesting good scalability for the algorithm. There is typically only a small delay, < 10 ms, between deciding and delivering the message, but it does seem to increase with the number of participants, likely due to the increased number of order conflicts, indicated in Fig. 7.2-2.

The second most important metric for this algorithm is the number of messages sent in order to deliver each OAC. This counts every message sent by each participant, and Fig. 7.2-3 shows the average number of messages vs. the participant count. As expected, the increase in number of

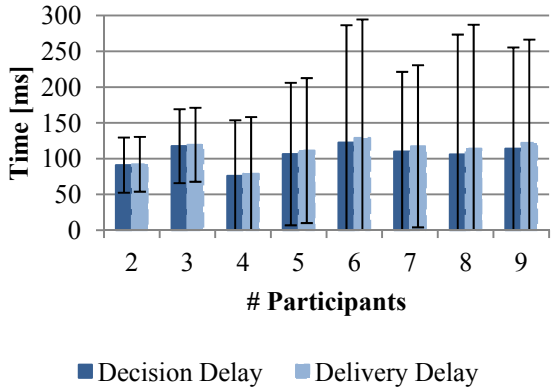


Fig. 7.2-1 Experiment AP-1 Decision and Delivery Delay vs. # Participants

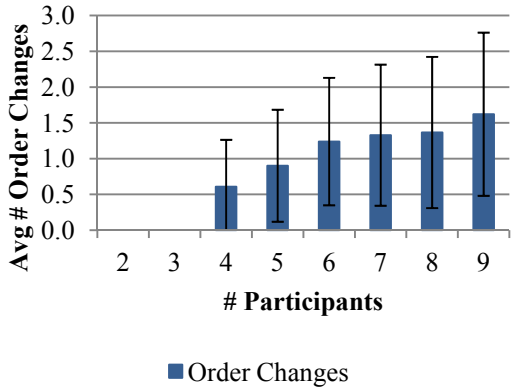


Fig. 7.2-2 Experiment AP-1 # Order Changes vs. # Participants

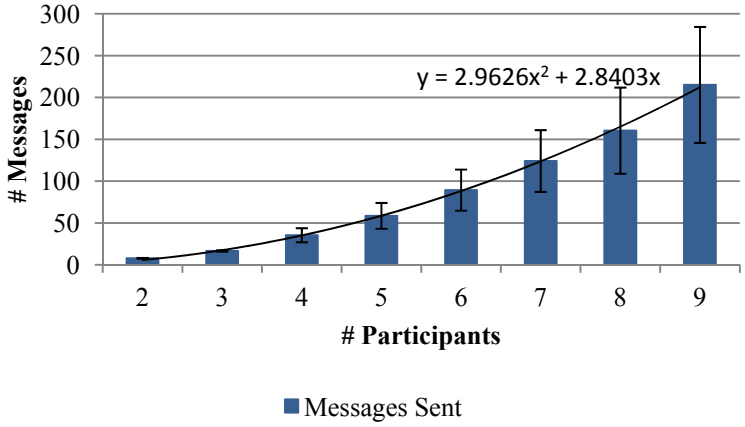


Fig. 7.2-3 Experiment AP-1 # Messages Sent vs. # Participants

messages is not linear, but fits well with a second order polynomial growth pattern. This will eventually limit the scalability, but for most applications will only present a problem once it begins impacting the delivery delay discussed above.

7.2.2 Experiment AP-2 – Host Membership

The Host Membership service is used to maintain the Host Group and has three primary functions: 1) adding hosts to the group when a join request is sent, 2) removing hosts from the group when a leave request is sent, and 3) removing hosts from the group when a host is suspected of failure. As before, the samples were gathered from all the experiments run in the other sections, however, since these events are much less common the number of samples is fewer, averaging 150 for joins, 28 for leaves, and 15 for removes. The average join, leave, and

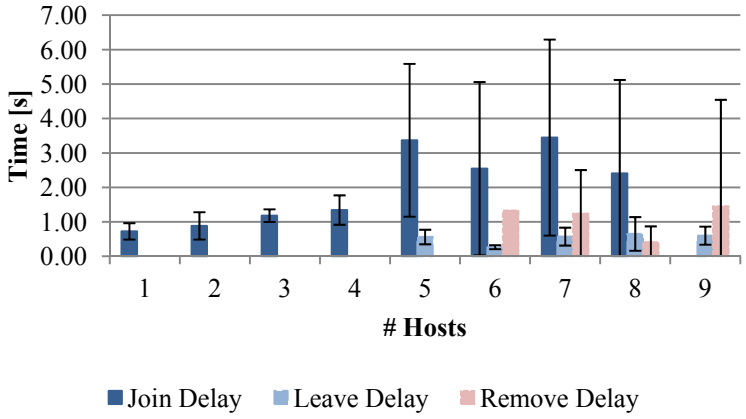


Fig. 7.2-4 Experiment AP-2 Join, Leave, and Remove Delay vs. # Hosts

remove delays are shown in Fig. 7.2-4, with the error bars used to indicate standard deviation. Join times are on the order of seconds for all cases, though they are significantly higher for group sizes more than five. Since OAC time was shown to be similar for these group sizes this notable increase must be caused by other factors, likely delays in establishing connections to each group member and synchronizing to the group state. Leaving a group is straightforward in that it only requires that you make the leave request and are removed via an updateMembership transaction, which is reflected in the consistent leave times averaging close to 0.6 seconds. Removing a failed member from a group is a much more complex process, and requires that all members suspect the failed member before a successful updateMembership transaction can occur. Despite this the average failed host is removed in less than 1.5 seconds after being suspected by the first member. Since host failures are so rare the number of samples was limited; however, 52 samples were available for the group size of nine, demonstrating good performance for what was theoretically the most complex case.

7.2.3 Experiment AP-3 – Agent Allocation

The performance of the agent allocation algorithm is primarily measured in terms of allocation time and number of messages sent. The number of hosts participating in the allocation and the number agents being allocated are two degrees of freedom in this algorithm, and the following graphs show the results for various numbers of hosts over six agent ranges. Because these samples were taken from live experiments not every case is represented, however, those that are present average at 324 samples and only three cases have fewer than 25 samples. Fig. 7.2-5

shows the average time taken to reach allocation consensus. There does not appear to be a strong correlation between time and the number of hosts, but there is a clear increase in time relative to the number of agents being allocated. As expected, the number of messages sent during allocation, Fig. 7.2-6, shows a strong correlation to both number of hosts and number of agents. Each host must share their bids with every other host, and more agents increases the likelihood of bidding conflicts, resulting in more messages and more rounds of bidding.

7.2.4 Experiment AP-4 – Agent Transfer and Recovery

The ability to spawn agents on demand, transfer agents, and recover failed agents are all key features of this architecture. Once the system is shown to function, the primary performance metric is the time cost of each action. Fig. 7.2-7 shows the average delay for a) spawning, the time a new agent is acknowledged by the system to the time it is ready begin performing its tasks, which also includes at least one session of agent allocation; b) freezing, the time between deciding to transfer the agent and the agent submitting its frozen state to the DDB; c) transferring, the time between deciding to transfer the agent and the agent resuming activities on the new host; and d) recovery, the time between detecting an agent failure and restoring the agent, this time

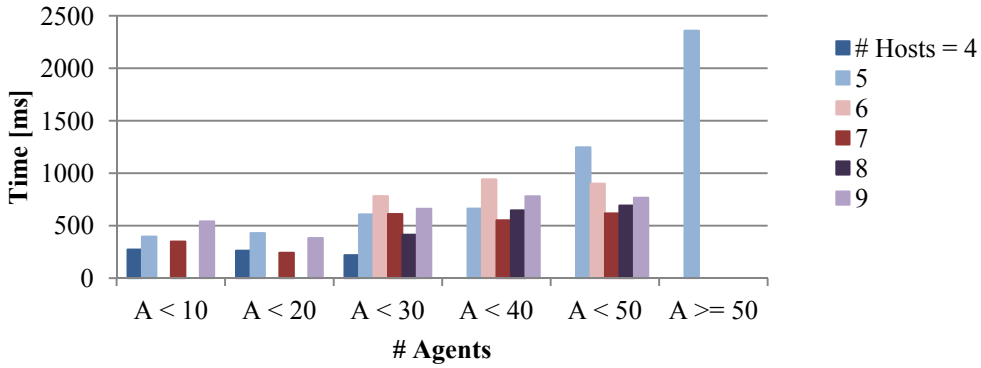


Fig. 7.2-5 Experiment AP-3 Allocation Delay vs. # Hosts vs. # Agents

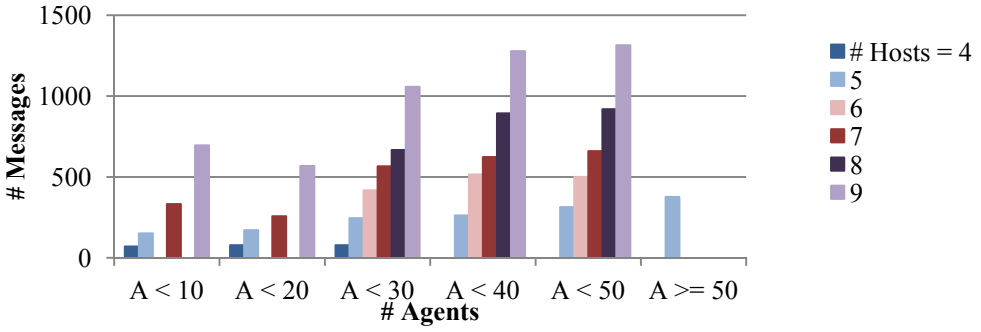


Fig. 7.2-6 Experiment AP-3 Messages Sent vs. # Hosts vs. # Agents

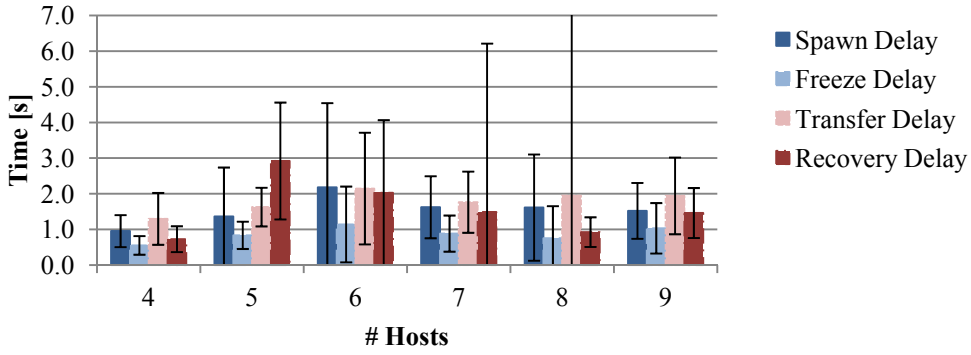


Fig. 7.2-7 Experiment AP-4 Agent Transfer and Recovery Delay vs. # Hosts

does not include the time taken to detect an agent failure since that is configurable using the UFD parameters.

The spawn delay averages between 1-2 seconds and is not tied to the number of hosts for these samples. On average agent transfer takes between 1.3-2.1 seconds, about one third to one half of which is taken freezing the agent. Given the average agent state size was 111 kB and the speed of the network, most of this time is accounted for by obtaining locks and the OAC messages required to maintain consistency and allow the transfer to be transparent to the other agents. The case of [50] where a single Java thread is transferred throughout the system has a transfer time of only 4.26 ms, which may make these transfer times seem high. However, their system does not have to consider the complexity of multiple agents and inter-agent communication, and does not appear to make efforts to robustly maintain consistency during the transfer. If the agent transfer algorithm used here only required freezing the agent state, sending the state to the next host, and unpacking the state, it would also perform on the order of milliseconds. [38] presents an application where entire sections of the control system are packed as VM states for transfer between robots. However, due to the unnecessary data stored in the complete VM state, transfers took between 20-120 seconds simply to transmit the state, in addition to ~5 seconds to save/compress and uncompress/restore the VM. On average restoring an agent takes between 0.7-2.9 seconds, and includes at least one session of agent allocation. None of the delays related to agent transfer and recovery appear to be correlated to the number of hosts in the system.

7.2.5 Experiment AP-5 – JC-SLAM

When considering the performance of a SLAM algorithm there are four primary metrics: 1) computational cost, 2) localization accuracy, 3) mapping accuracy, and 4) mapping rate. In particular, the question is how these metrics are affected by the three strategies employed by JC-SLAM: i) delayed calculation of weight updates, ii) out-of-order processing, and iii) propagation of observations through resampling transforms.

Three experiments were designed to study these questions. The first focuses on localization accuracy and compares the results from a number of different variations of the JC-SLAM strategy in three different scenarios. The second experiment simply monitors the number of observations that accumulate between each weight update to calculate the number operations saved. The final experiment takes a full mapping and exploration scenario and compares the performance of JC-SLAM against two traditional SLAM approaches.

7.2.5.1 Experiment AP-5.1 Impact of JC-SLAM Strategies

As shown in the discussion in Chapter 5, the strategies of JC-SLAM are theoretically grounded, but for practicality of implementation a number of unproven steps are made. In order to justify JC-SLAM, the strategies were verified experimentally. To accomplish this three different teams, Table 7.2-1, were run though mapping and exploration scenarios and the particle filter predictions and observations were recorded. Then 11 variations of the SLAM algorithm were used with the data to analyse the impact on accuracy, required number of particles, and rate of particle decay.

Six SLAM strategies were used under nominal and constrained processing conditions, for a total of 11 variations (the baseline strategy disregards processing conditions). The SLAM strategies are defined in Table 7.2-2.

The processing conditions were tuned for each experiment using the Discard strategy such that the nominal level was defined as processing ~90% of the readings and the constrained level

Table 7.2-1 Experiment AP-5.1 Experimental Scenarios

Experiment	Team	Explored Area [m^2]	Mission Time [mm:ss]	Uncorrected Localization Error [m]
Basic	4 Seekers	24.2	2:56	0.123
Advanced A	4 Sweepers	216	27:05	0.804
Advanced B	8 Sweepers	216	15:22	0.281

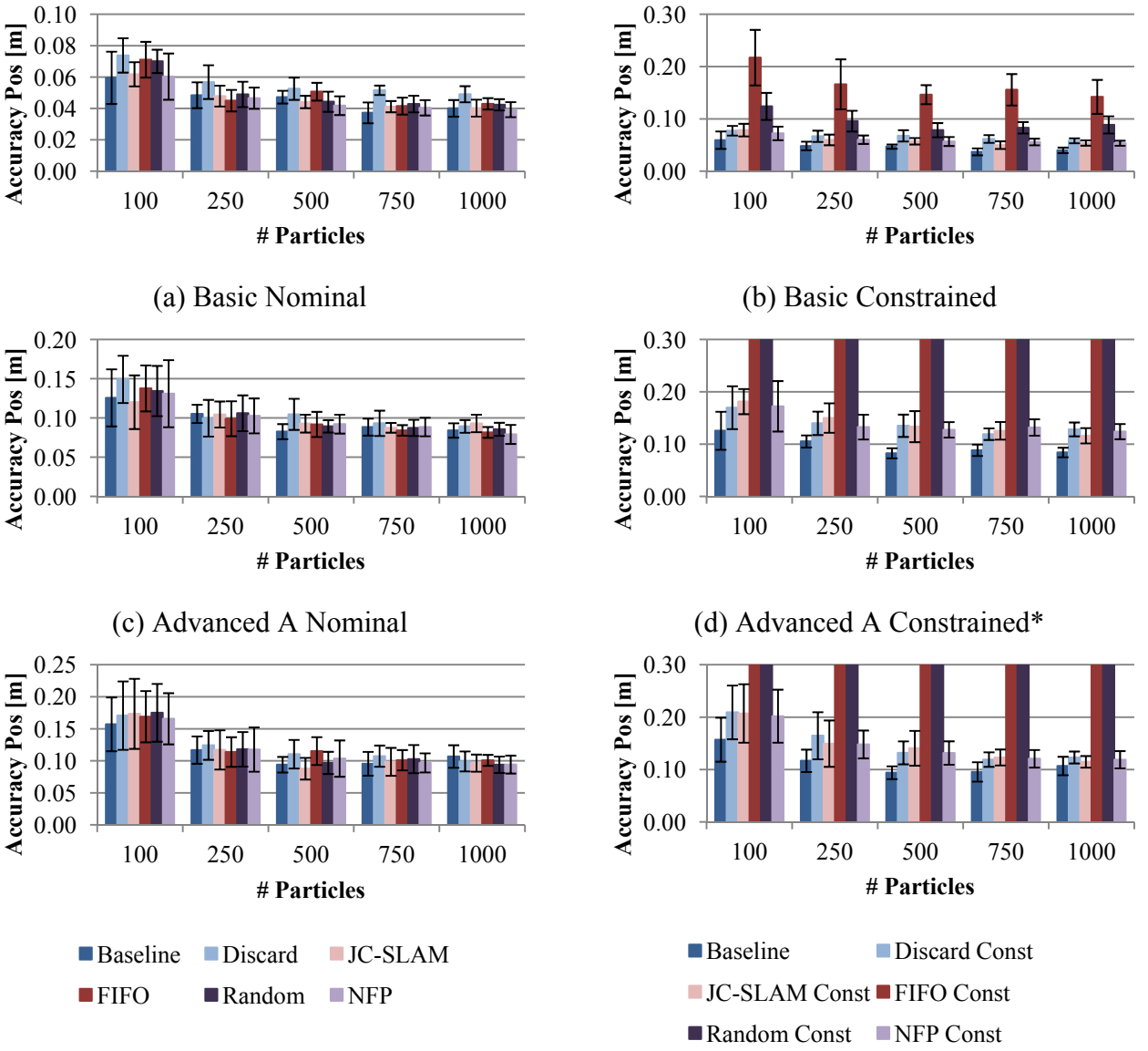
Table 7.2-2 Experiment AP-5.1 SLAM Strategies

Strategy	Description
Baseline	All readings were processed in order as soon as they became available. This is the standard SLAM approach.
Discard	Readings were processed in order as soon as they became available, however, if insufficient processing resources were available at that time the reading was discarded. This is a standard SLAM approach when resources are insufficient to process all readings.
JC-SLAM	The proposed SLAM implementation: readings were processed in LIFO (last-in-first-out) order, readings that could not be immediately processed were held until resources became available, weight updates were delayed until explicitly requested, and observation densities were forward propagated through resampling transforms.
FIFO	The same as the JC-SLAM implementation except the readings were processed in FIFO (first-in-first-out) order. In systems where all readings can be processed the end result is very similar to the normal SLAM approach, however, in constrained systems this means that processing can fall significantly behind the newest readings.
Random	The same as the JC-SLAM implementation except the readings were randomly chosen for processing from the pool of available readings. In this approach a selection of new and old readings were processed, even in constrained systems.
NFP	The same as the JC-SLAM implementation except that observation densities were not forward propagated through resampling transforms.

defined as processing ~50% of the readings. These same processing conditions were then used for the four JC-SLAM variations. Even though the set of observations and the avatar motion predictions are the same for each test there are still a significant number of random variables; and so each experiment was run 10 times and the average values are reported.

Each SLAM variation was tested using 100, 250, 500, 750, and 1000 particles for each filter. The questions of accuracy and required number of particles will be answered by looking at Fig. 7.2-8, which shows the localization accuracy for each test. The error bars are used to show the standard deviation from all trials.

The Baseline scenario shows that the SLAM algorithm is functioning as expected, demonstrating significant improvements over the uncorrected localization for all three scenarios, and showing diminishing improvement as the number of particles increases. For the nominal tests each strategy is able to process most of the readings and performs close to the Baseline. The significant standard deviation makes it difficult to make strong statements about relative performance, but from these results the JC-SLAM variations appear to have a slight advantage



*The vertical axis of these graphs has been set to highlight the differences between Baseline, Discard, JC-SLAM, and NFP. FIFO and Random are clipped because they were unable to successfully localize in these scenarios.

Fig. 7.2-8 Experiment AP-5.1 Localization Accuracy

over Discard. The constrained tests show that FIFO and Random are unable to maintain localization, since they rapidly fall behind on processing recent readings. Discard, JC-SLAM, and NFP all perform similarly and still demonstrate a marked improvement over uncorrected localization.

From these results the following statements are made:

- JC-SLAM has equivalent accuracy to Baseline when sufficient processing resources are

available.

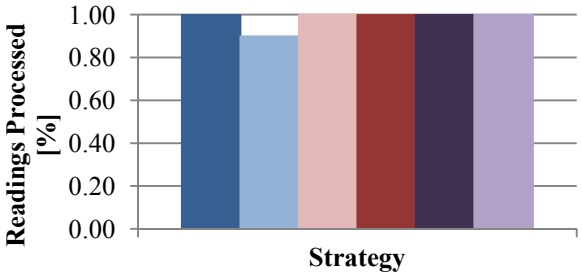
- JC-SLAM has the same or better accuracy as Discard in constrained scenarios.
- JC-SLAM follows the same improvement as Baseline or Discard as the # of particles increases.
- Forward propagating observations through resampling transforms yields no benefit for these scenarios. This suggests that, though theoretically sound, forward propagation may not be a worthwhile addition.

Taking the results of the Advanced B trials, Fig. 7.2-9 shows the number of readings processed, the processing order, and the average delay between when the observation is generated and when it is processed. By definition Discard is able to process ~90% of the readings for the nominal test and ~50% of the readings for the constrained test. In both cases the JC-SLAM variations are able to take advantage of the stored observations and process roughly 10% more of the readings during lulls in observation generation. By design Baseline, Discard, and FIFO never process any readings out-of-order (OOO). JC-SLAM and NFP use the same strategy and therefore have the same results, largely keeping up with only 2.5% of readings processed OOO during the nominal test, but processing 30% of the readings OOO for the constrained tests. From the accuracy results above it can be seen that this OOO processing has no noticeable impact on performance. This can be explained in part by observing that even though the readings are being processed OOO, the average delay between when a reading was generated and processing the reading is relatively small, 52 ms for the nominal test and 1,212 ms for the constrained test. The average delay in processing is much larger for the constrained FIFO and Random tests, 337 seconds and 240 seconds, respectively. Since these strategies fall behind as they continue to process old readings their accuracy is greatly affected. The results show that:

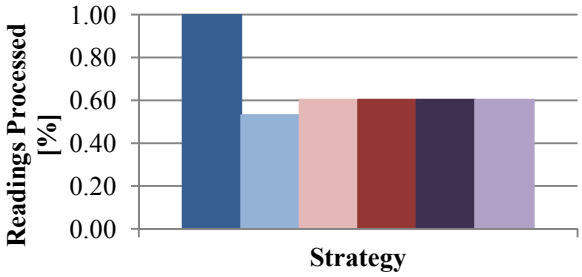
- JC-SLAM is able to take advantage of available processing resources to process additional observations, even in severely constrained scenarios.
- The OOO processing strategy does not affect accuracy so long as new readings are given priority.

To study the impact of JC-SLAM on particle diversity decay the average rate of resampling was recorded for each test. Fig. 7.2-10 plots the rate of resampling against the reading processing rate, and shows that there is a strong correlation for each scenario. With the exception of the

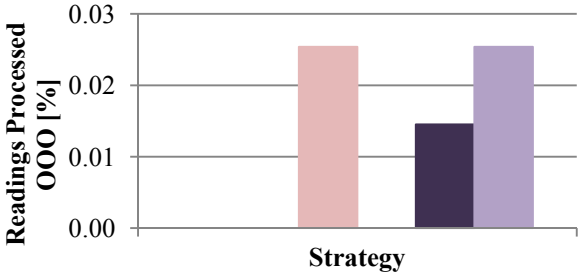
constrained FIFO and Random tests, which have poor localization and therefore reduced rates of resampling, the remaining tests are linear. This demonstrates that by itself JC-SLAM has minimal impact on particle diversity decay, and the primary factor is the reading processing rate.



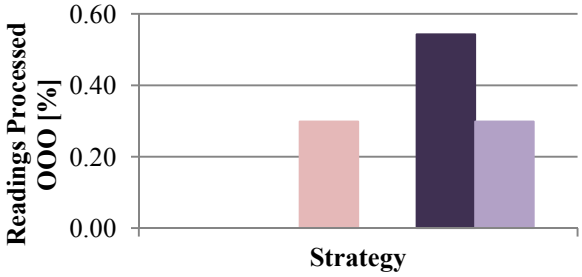
(a) % Readings Processed Nominal



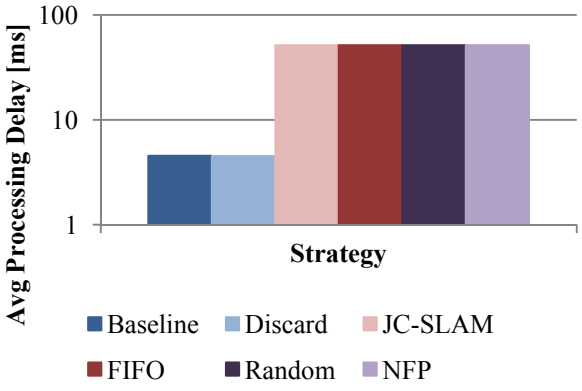
(b) % Readings Processed Constrained



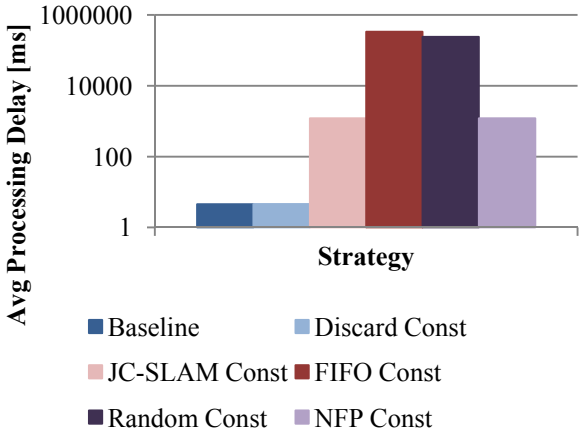
(c) % Readings Out of Order Nominal



(d) % Readings Out of Order Constrained



(e) Average Processing Delay Nominal



(f) Average Processing Delay Constrained

Fig. 7.2-9 Experiment AP-5.1 Observation Processing

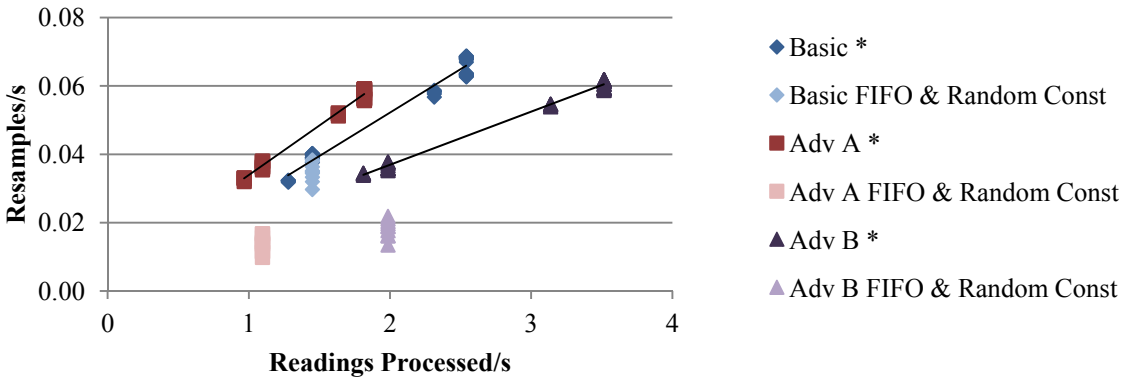


Fig. 7.2-10 Experiment AP-5.1 Reading Processing Rate vs. Resampling Rate

7.2.5.2 Experiment AP-5.2 Operations Saved from Delayed Calculation of Weight Updates

From Strategy 1, JC-SLAM accumulates observation densities and delays weight updates until particle weights are explicitly requested. As discussed in Chapter 5, this approach has only minimal impact on SLAM performance since all observations that have be processed up to that point are included when the weights are calculated, it simply changes how and when the calculations are done. Studying (5.1-4), for a particle filter with N particles a standard weight update requires N multiplications, N additions, and N divisions. It follows that M observations, when calculating the weights after each observation, require MN multiplications, MN additions, and MN divisions. If the delayed weight calculation strategy (5.2-3) is used these calculations can be reduced to MN multiplications, N additions, and N divisions. Depending on the number of observations accumulated between each weight update this could result in significant savings. To study the impact in a real scenario the average number of observations between each weight request was measured during the mapping and exploration scenarios of the previous experiments and the number of saved operations was calculated. These results are shown in Table 7.2-3.

7.2.5.3 Experiment AP-5.3 Cooperative Exploration and Mapping

This experiment studies the rate of map generation of JC-SLAM compared to the two traditional SLAM approaches, Discard and Delay, which are used when processing resources are constrained. Available processing resources are a significant concern since results must be available in real-time, and quite often the rate at which new data are collected surpasses the rate that the data can be processed. When this occurs the two basic options are either discarding the

extra readings or delaying exploration until processing is finished [76]. With JC-SLAM a third option is possible: unprocessed readings can be stored until processing resources become available and then integrated into the SLAM solution. All three approaches were implemented in the HAA control system. Each approach used the same particle filter, map, and sensor processing implementation, and the sole differences between implementations are described in Table 7.2-4.

For this experiment six Sweeper avatars were used in the large 18 x 18 m arena. To approximate changing processing resources six hosts were used at the beginning of the experiment, two hosts

Table 7.2-3 Experiment AP-5.2 Savings from Delayed Weight Updates

Experiment	# of Particles (*N*)	Average Observations between Weight Updates (*M*)	Standard Weight Update Cost (*MN* x, *MN* +, *MN* /)	JC-SLAM Weight Update Cost (*MN* x, *N* +, *N* /)	Computational Savings (% operations)
Basic	500	2.93	1465, 1465, 1465	1465, 500, 500	44%
Advanced A	500	3.97	1985, 1985, 1985	1985, 500, 500	50%
Advanced B	500	4.89	2445, 2445, 2445	2445, 500, 500	53%

Table 7.2-4 Experiment AP-5.3 SLAM Implementations

Implementation	Description
JC-SLAM	Sensor readings are added to a stack by SupervisorSLAM as they are generated. Readings are assigned to sensor processing agents one at a time in LIFO order. Once a reading is processed the processing agent is assigned the next reading. If more readings accumulate than can be readily processed by the current sensor processing agents additional agents are requested, limited by the available processing power of the hosts and up to a maximum of one sensor processing agent of each type (Cooccupancy, Sonar, Landmark, FloorFinder) per host. Avatar path planning and movement is independent of the sensor processing activities, though ultimately path planning is influenced by the generated map.
Discard	Sensor processing agents of each type are started on every host, meaning that Discard always has the maximum number of processing agents possible for the JC-SLAM implementation, and for this experiment had double the number of agents of JC-SLAM. As sensor readings are generated SupervisorSLAM assigns them to any available sensor processing agent, if no processing agents are immediately available the reading is discarded. Avatar path planning and movement is independent of the sensor processing activities, though ultimately path planning is influenced by the generated map.
Delay	Each Avatar agent is responsible for handling its own sensor readings, and requests a dedicated sensor processing agent of each type. Depending on the ratio of hosts to avatars this may result in more or less processing agents than the Discard implementation (and maximum of the JC-SLAM implementation), but in this experiment there were two times more than the JC-SLAM implementation. When sensor readings are generated the avatar halts activity until all readings are processed, at which point the avatar is free to move and generate more sensor readings.

left after 5 minutes, and two hosts joined after 10 minutes. The mission completion condition was that 95% of the reachable cells were explored, where "explored" is defined as a cell with a value greater than 0.73 or less than 0.27. Each experiment was repeated 10 times, and Table 7.2-5 reports the results from the three SLAM approaches.

Map coverage and map accuracy remained consistent across all approaches, averaging 95% and 91%, respectively. The development of map coverage and map accuracy over time for a typical trial of each approach is shown in Fig. 7.2-11 and Fig. 7.2-12, respectively; and, while the final values are similar, JC-SLAM reaches them more quickly. This results in a significant difference in mission duration, where on average JC-SLAM finishes 17% (3.8 minutes) faster than Discard and 33% (9.1 minutes) faster than Delay. The improved performance is a direct result of JC-SLAM's ability to process sensor readings at a higher rate (though note that the processing time per reading was the same for all approaches). The rates of reading generation and processing are shown in Fig. 7.2-13, and the advantage of JC-SLAM is clear. JC-SLAM is able to keep up with the high rate of reading generation, averaging 24.0 readings/s for both generation and processing. Discard has a slightly lower average generation rate of 22.8 readings/s, and only has an average processing rate of 18.4 readings/s; meaning Discard must throw away roughly four readings per second. Delay has a reduced reading generation rate because it must wait for processing to complete before generating more readings, averaging 16.7 readings/s for both generation and processing. Delaying actions while readings are being processed is also the reason why Delay has a longer mission duration.

Table 7.2-5 Experiment AP-5.3 SLAM Comparison

Title	Mean			RMS			STD		
	JC-SLAM	Discard	Delay	JC-SLAM	Discard	Delay	JC-SLAM	Discard	Delay
Mission Duration [min]	18.42	22.18	27.49	18.44	22.19	27.87	0.74	0.69	4.53
Map Coverage [%]*	0.951	0.949	0.958	0.951	0.949	0.959	0.0063	0.0095	0.0068
Map Accuracy [%]*	0.906	0.907	0.909	0.906	0.907	0.909	0.0033	0.0072	0.0075
Localization Err Pos [m]	0.086	0.097	0.107	0.087	0.101	0.114	0.0172	0.0299	0.0398

*Relative to a fully explored map generated with no localization error

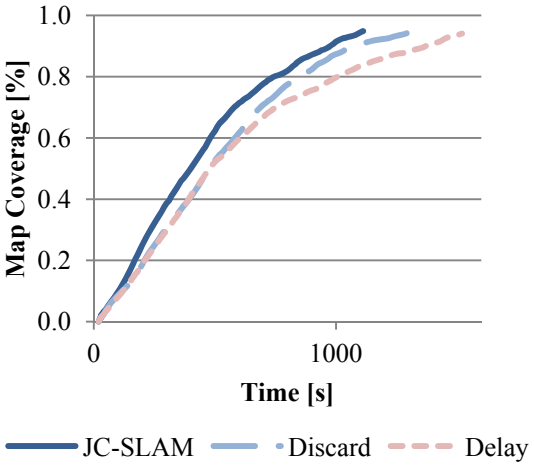


Fig. 7.2-11 Experiment AP-5.3 Map Coverage

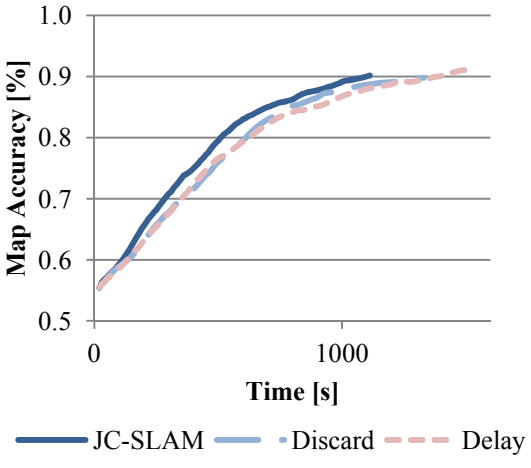
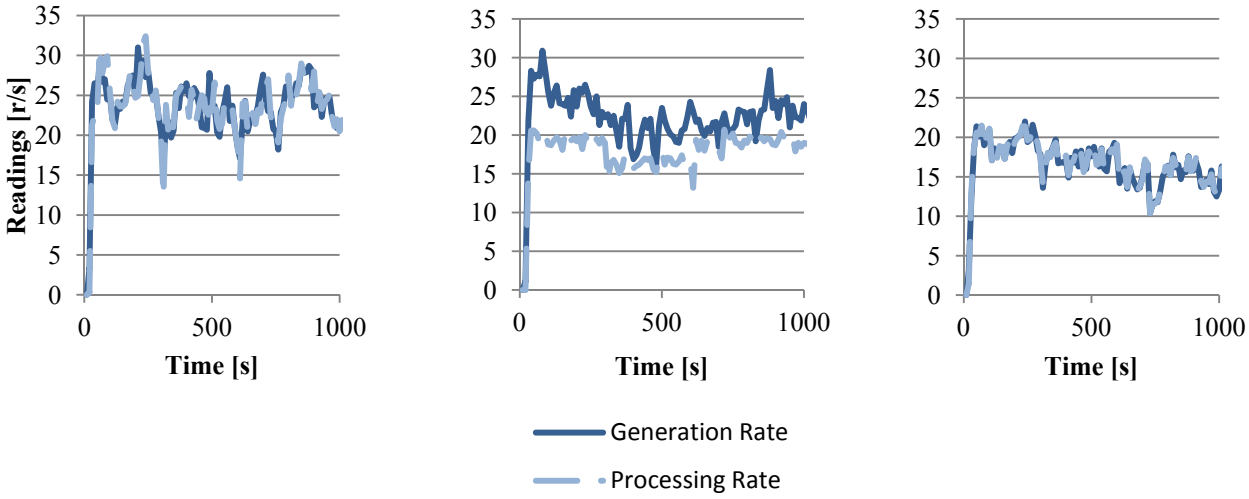


Fig. 7.2-12 Experiment AP-5.3 Map Accuracy



(a) JC-SLAM (b) Discard (c) Delay

Fig. 7.2-13 Experiment AP-5.3 Reading Generation and Processing Rates

7.3 Robustness

Two experiments were conducted to evaluate the performance of the system under various failure scenarios:

1. *Experiment R-1 – Agent Failure:* To explore the impact of increasing rates of agent failure.

2. *Experiment R-2 – General Failure:* To demonstrate the ability to handle concurrent host, agent, and avatar failures.

7.3.1 Experiment R-1 – Agent Failure

One of the most important features of the control system is the ability to detect and recover from agent failures. Experiment R-1 was designed to not only show that detection and recovery work as prescribed, but to study the impact of failure on key performance metrics. The Mapping and Exploration scenario in the small arena was chosen to reduce the performance variables and generate the fairest comparisons. Three hosts were used, along with four seeker avatars. The mission completion criterion was that 95% of reachable cells were explored. Artificial failures were introduced into the system, and were controlled by specifying a minimum and maximum operating time for each agent. The failure would then occur at a random time in that interval. Four different failure rates were tested: 1) No Failure, 2) Moderate Failure, agent life expectancy 1-10 minutes, 3) High Failure, agent life expectancy 1-3 minutes, and 4) Extreme Failure, agent life expectancy 0.5-1.5 minutes. All of the tested failure rates are much higher than any realistic scenario, yet the performance impacts were minimal. The summary of results is presented in Table 7.3-1. The primary performance metric is mission duration, and clearly differences are expected since the minimum possible impact is the delay in detecting an agent failure and recovering the agent. Two other main factors that impact mission duration are: i) agent backups are incomplete by design (a trade-off is made between completeness and network usage in creating backups),which means redoing some work or retrieving that information from the DDB or other agents, and ii) many agents have interdependencies and so one agent failing can delay multiple agents. In these experiments all cases were able to successfully complete the mission, though the moderate, high, and extreme failure rates increased the duration by 10%, 27%, and 52%, respectively. The increase to mission duration can be predicted by this first order model:

$$\Delta D = \frac{(\mathrm{MTF} + k)}{\mathrm{MTF}} \tag{7.3-1}$$

where ΔD gives the ratio of the mission duration to the mission duration with no failures, MTF is the mean-time-to-failure for the agents, and k is a constant related to the time cost of agent failure. Fig. 7.3-1 plots the experimental results alongside the first order model with k = 0.55. This suggests that, for this scenario, for every MTF that elapses the mission duration increases by

0.55 minutes. This constant will be different for each scenario and setup since it is heavily dependent on the number of agents and the complexity of their interactions; and so it is difficult to predict what the constant will be without experimental testing.

The agent allocation graphs for each failure rate, Fig. 7.3-2 to Fig. 7.3-5, show the pattern of agent failures and recoveries. The No Failure case in Fig. 7.3-2 of course shows no failures and only normal agent transfers. The Moderate Failure case in Fig. 7.3-3 shows that three agent failures occurred: two AvatarSimulation agents and one AgentPathPlanner. Despite the agents being recovered within seconds of discovering the failures, these few failures were enough to have a noticeable impact on mission duration. Fig. 7.3-4 and Fig. 7.3-5 show 17 and 46 failures for the High and Extreme Failure cases, respectively. Even with these highly unrealistic rates of

Table 7.3-1 Experiment R-1 Results Summary

Title	Mean				STD			
	No Failure	Moderate	High	Extreme	No Failure	Moderate	High	Extreme
Mission Duration [min]	2.573	2.842	3.261	3.900	0.2606	0.3476	0.4108	0.4927
Map Coverage [%]*	88.00	88.88	88.49	87.80	2.3483	1.5553	1.9932	1.9850
Map Accuracy [%]*	90.58	90.43	90.27	89.86	0.8154	0.3783	0.9082	0.9449
Localization Pos Err [m]	0.0602	0.0676	0.0774	0.1016	0.0203	0.0192	0.0261	0.0377
Localization Rot Err [rad]	0.0370	0.0576	0.0394	0.0828	0.0095	0.0257	0.0204	0.0412
Landmark Err [m]	0.0801	0.0933	0.0860	0.1136	0.0159	0.0201	0.0190	0.0483
Landmark Cov [m]	0.4445	0.4133	0.3578	0.3739	0.1318	0.1528	0.0804	0.0831
Average # Hosts	4.00	4.00	4.00	4.00	0.0000	0.0000	0.0000	0.0000
Average # Agents	14.20	14.32	15.14	16.37	0.2970	0.3454	0.5694	0.4536
Average CPU Usage	40.60	38.99	35.67	28.79	1.7048	1.9236	2.4381	2.2266
DDB Size [MB]	8.434	9.075	9.304	9.225	0.9787	1.0661	0.9106	0.8870

*Relative to a fully explored map generated with no localization error

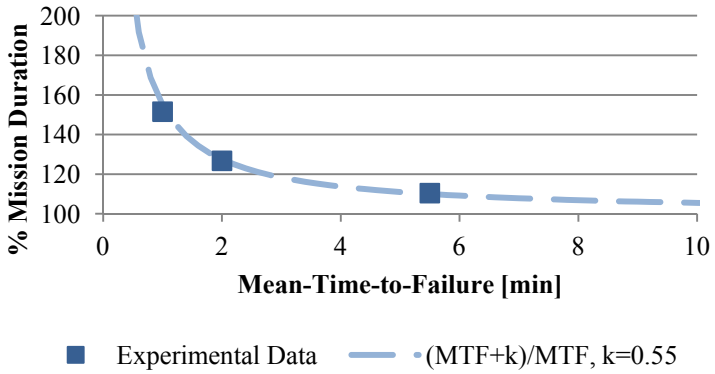


Fig. 7.3-1 Experiment R-1 Mean-Time-to-Failure vs. Mission Duration Increase

failure the control system is able to successfully recover each agent and continue operating without significant sacrifices to anything but mission duration.

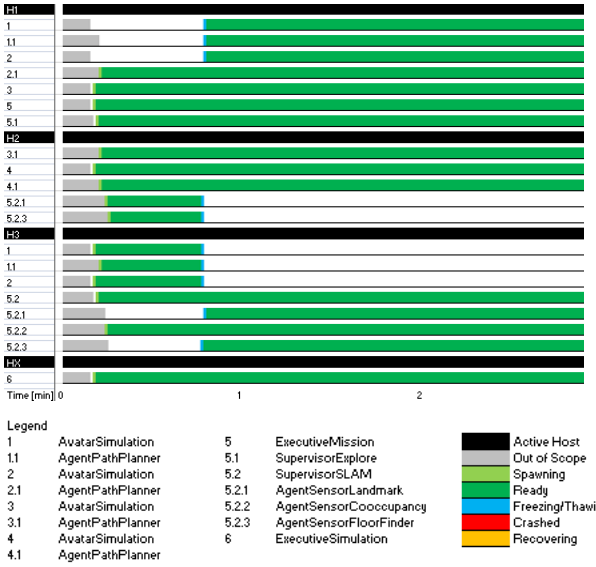


Fig. 7.3-2 Experiment R-1 Agent Allocation: No Failure

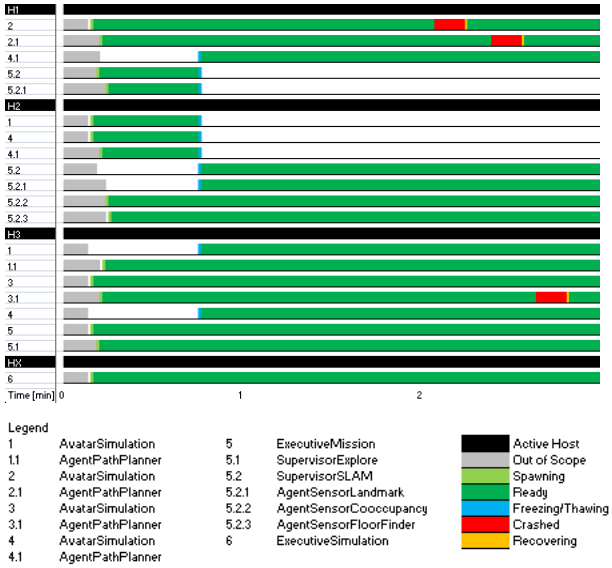


Fig. 7.3-3 Experiment R-1 Agent Allocation: Moderate Failure

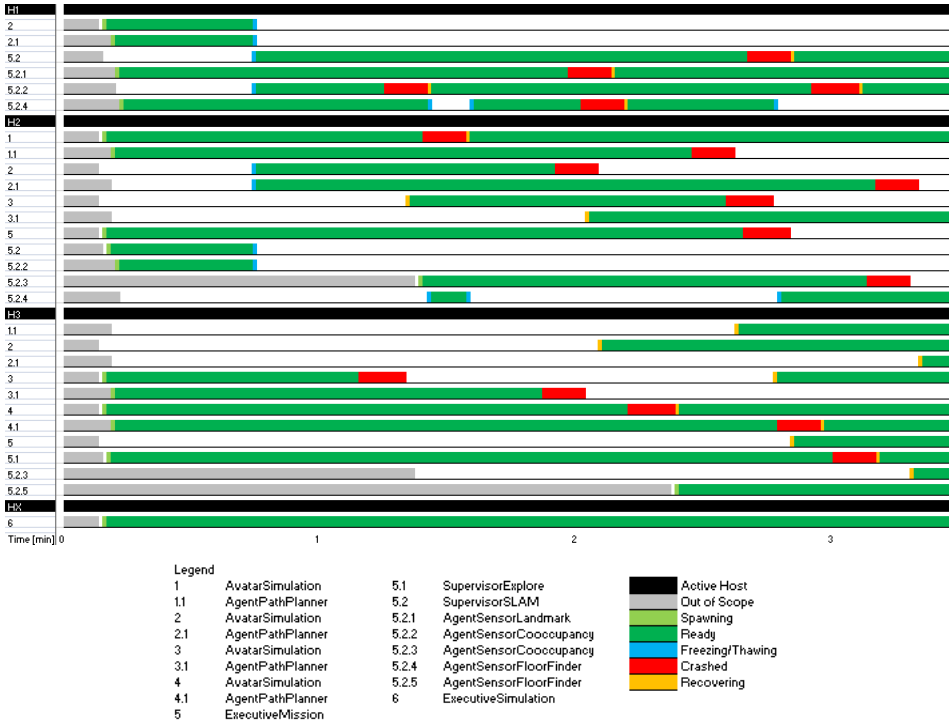


Fig. 7.3-4 Experiment R-1 Agent Allocation: High Failure

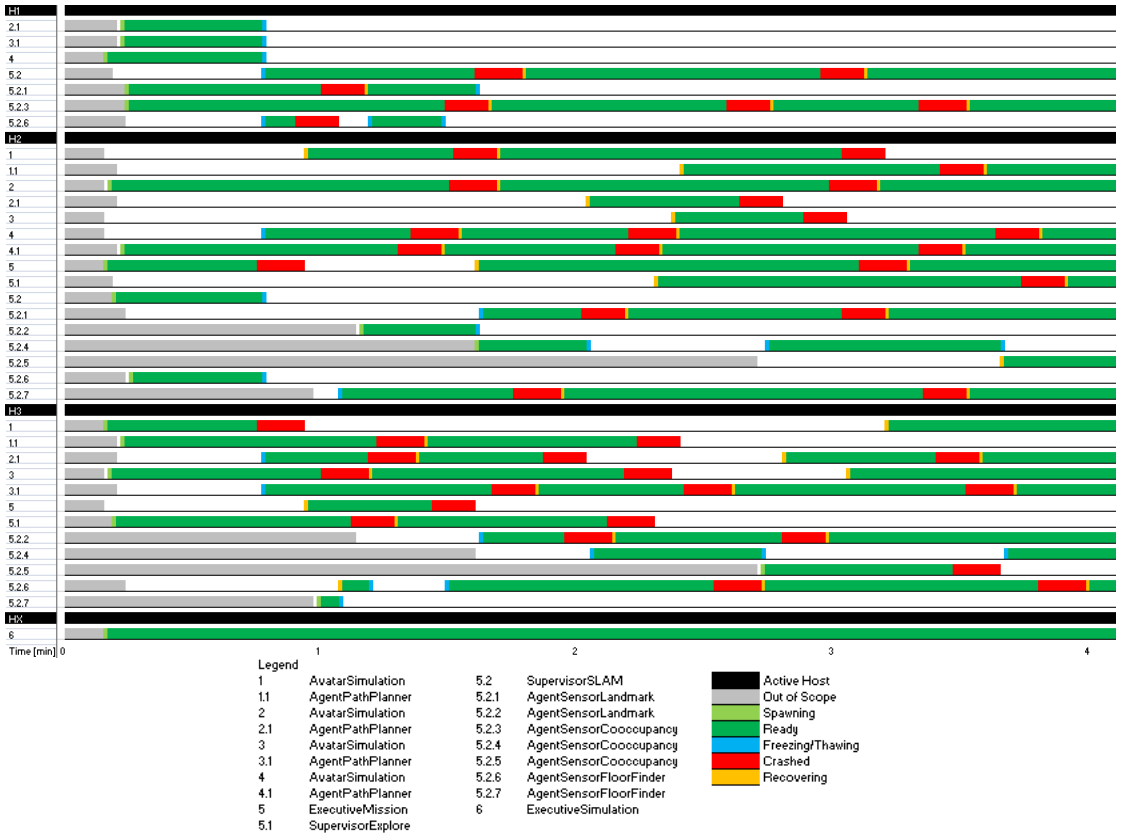
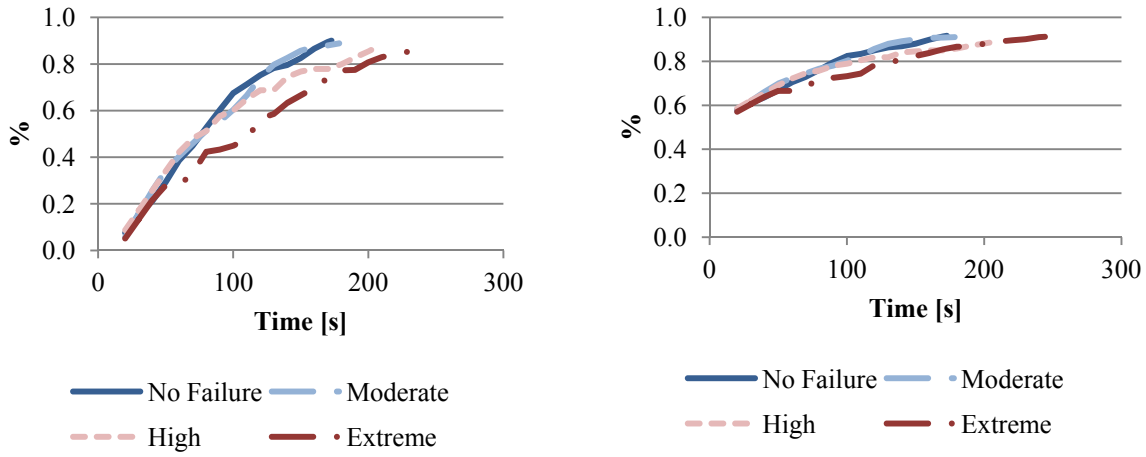


Fig. 7.3-5 Experiment R-1 Agent Allocation: Extreme Failure

Differences in final map coverage and map accuracy between cases were not significant, with means of 88% and 90%, respectively, though map accuracy appears to follow a slowly decreasing trend. The rate of growth for these values corresponded to the increased mission durations, as shown in Fig. 7.3-6. The coverage and accuracy curves also display a "roughness" that increases noticeably with higher rates of failure, explained by frequent exploration delays as agents crash and are recovered. From the comparisons of localization error in Fig. 7.3-7, error does appear to climb more rapidly with increased number of failures, but still appears to stabilize at an acceptable level and has only minimal affect on map accuracy.

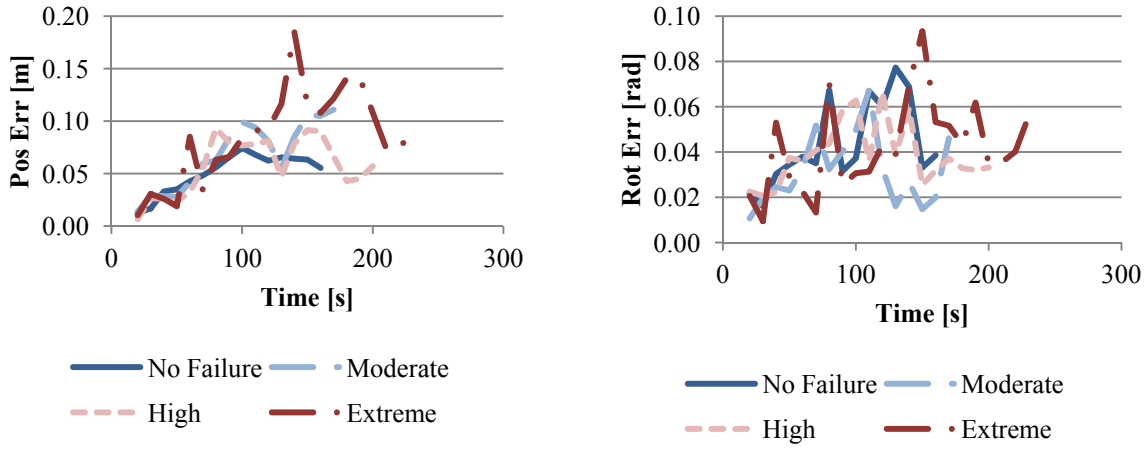
7.3.2 Experiment R-2 – General Failure

The next step after agent failure is the ability to handle general failures, which is demonstrated in this experiment. Host failures in particular present a greater challenge, since not only do they impact the host group functions, but all agents running on a failed host must be considered failed as well. When a host fails it must be removed from the host group, all outstanding atomic



(a) Map Coverage (b) Map Accuracy

Fig. 7.3-6 Experiment R-1 Map Coverage and Accuracy



(a) Positional (b) Rotational

Fig. 7.3-7 Experiment R-1 Localization Error

messages aborted, and the failed agents recovered. Host and avatar failure are also likely to be the dominant forms of failure in a well established control system, since they may result from many physical factors such as loss of communication, loss of power, or hardware damage.

Experiment R-2 takes on the challenge of demonstrating every feature of the control system in the most difficult scenario: Foraging in the large arena. This includes i) dynamic host group formation, ii) dynamic control system construction, iii) balancing processing resources through agent distribution, iv) completing multiple simultaneous tasks, v) adding hosts and avatars after the mission starts, and vi) host, avatar, and agent failure. 20 collectables were distributed within the large arena, as well as three collection regions. Eight hosts formed the initial host group and

additional hosts joined at 6 and 12 minutes. Hosts were given life expectancies and simulated pseudo-random failures. 10 avatars were used in total, two sweepers, four enhanced seekers, one of which crashed after two minutes and one who retired after five minutes, two carriers, and finally two additional carriers who joined after five minutes. Agent life expectancy was set to 1-10 minutes. The mission completion criteria were that the 95% of reachable cells had been explored and that all discovered collectables were deposited. The mission completed successfully for all runs, demonstrating that the HAA architecture makes a strong foundation for a control system, particularly in unfavourable operating conditions.

Fig. 7.3-8 shows a typical mapping result and the back and forth travel of the avatars as they collect their cargo. The results summary is presented in Table 7.3-2, reporting an average mission duration of 28.4 minutes. This is comparable to the 28.5 minute mission duration for the similar Experiment AF-3, even though fewer avatars with carrying capacity were used. Map accuracy of 91% and localization error of 0.123 m/0.025 rad are also comparable to the other experiments. The number of hosts and agents is given in Fig. 7.3-9, demonstrating the numerous host failures and the large variation of agents typical in a foraging scenario. Map coverage and accuracy, Fig. 7.3-10, both develop more linearly than normal, as expected in a foraging scenario,

Fig. 7.3-8 Experiment R-2 Mapping Result

and exhibit the roughness caused by agent failure. Fig. 7.3-11 shows that localization error is stable at normal levels, and finally, load balancing is without issue, as shown in Fig. 7.3-12. The agent allocation graph, Fig. 7.3-13, presents the difficult struggle of the scenario with four host failures and a total of 67 agents that each experienced multiple crashes, yet the control system was able to recover and continue operation.

Table 7.3-2 Experiment R-2 Results Summary

Title	Representative Run	Mean	RMS	STD
Mission Duration [min]	28.218	28.427	28.448	1.099075
Map Coverage [%]*	97.40	97.56	97.56	0.418900
Map Accuracy [%]*	91.73	91.16	91.16	1.252300
Localization Positional Err [m]	0.12305	0.12362	0.13595	0.056564
Localization Rotational Err [rad]	0.02084	0.02590	0.02822	0.011209
Landmark Err [m]	0.10291	0.11792	0.12692	0.046930
Landmark Covariance [m]	0.04004	0.03760	0.03765	0.002012
Average # Hosts	7.03	6.89	6.90	0.351378
Average # Agents	38.61	39.06	39.13	2.458903
Average CPU Usage [%]	60.13	61.71	61.89	4.787600
DDB Size [MB]	32.892	33.089	33.162	2.203648
Cargo Collected	20.00	19.90	19.90	0.300000

* Relative to a fully explored map generated with no localization error

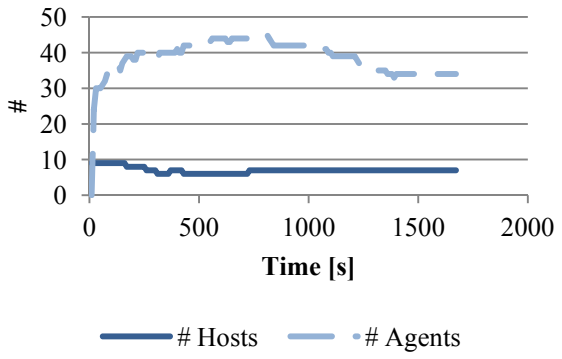


Fig. 7.3-9 Experiment R-2 Hosts and Agents

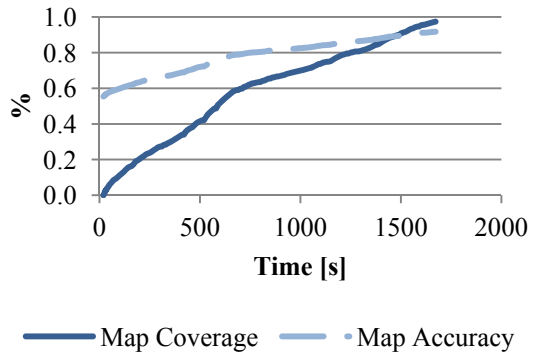


Fig. 7.3-10 Experiment R-2 Map Coverage and Accuracy

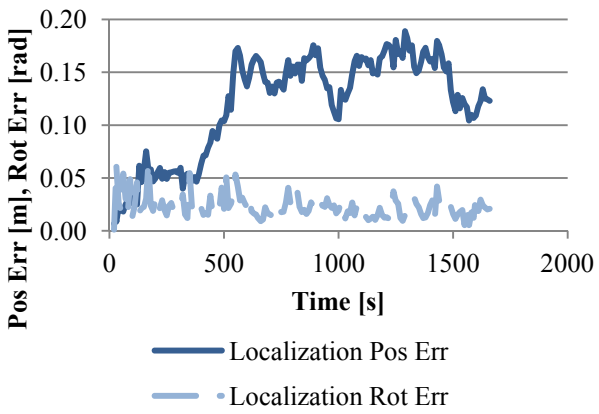


Fig. 7.3-11 Experiment R-2 Localization Error

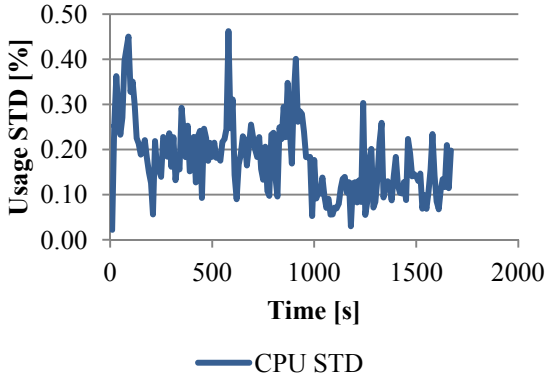


Fig. 7.3-12 Experiment R-2 CPU Balancing

Fig. 7.3-13 Experiment R-2 Agent Allocation

Chapter 8
Conclusions

Transparency and reusability are key factors in improving the process of developing control systems for real-world robot teams. A new approach is necessary in order to break the cycle of throwing away previous work and starting each team from scratch, and that requires an understanding of the how's and why's of control systems beyond simple end-of-lifecycle performance. To encourage asking the right questions and provide some guidelines the Control *ad libitum* philosophy was developed, promoting the tenets of Transparency, Versatility, Adaptability, Modularity, Diversity, Persistency, and Efficiency. The author asserts that by considering these tenets during each phase of the team lifecycle great strides can be made toward the reusability and extensibility of control systems. Following this approach the generic, dynamic, versatile, robust, and extensible HAA architecture is proposed. HAA lays a foundation on which virtually any type of control system can be built, while providing features such as a distributed processing network, a distributed database, and failure recovery. HAA also encourages modularity as a way to simplify the design process and allow for greater reusability between applications.

The viability of the HAA architecture is demonstrated with a fully developed implementation, constructed with provably correct distributed algorithms. In computer science a provably correct algorithm is one where logical proofs can be constructed for each of the specifications of the algorithm. This guarantees that within these specifications no incorrect step will ever be taken. A control system was designed to complete tasks including exploration and mapping, search and deploy, and foraging. Each of these tasks builds on those preceding it and shows the high degree of reusability of agent modules. Additionally, the JC-SLAM algorithm was developed following the Control *ad libitum* approach as a versatile and distributed solution to the mapping and localization problem.

Using HIL simulation the control system was extensively tested to demonstrate its features and performance under different conditions. Three sets of experiments were conducted. In the first series of experiments the fundamental features of the HAA architecture were studied, specifically the ability to: a) dynamically form the control system based on the task requirements, b)

dynamically form the team from available avatars, c) dynamically form the host network based on available processor resources, and d) handle heterogeneous teams and allocate avatars between tasks based on their capabilities. The second series of experiments evaluated the performance of the distributed algorithms for various system sizes, and each algorithm demonstrated highly acceptable real-time performance and no issues of scalability for the small-to-moderate sized systems tested. The SLAM problem is fundamental to the implementation of virtually any robot team, and so JC-SLAM was developed as a distributed and scalable solution. JC-SLAM demonstrated accuracy equal to or better than traditional SLAM approaches in resource constrained scenarios, and reduced exploration time by over 17% for the mapping scenarios tested. The JC-SLAM strategies are also suitable for integration into existing particle filter SLAM approaches, complementing their unique optimizations. The last series of experiments focused on robustness against concurrent agent, avatar, and host failure. Multiple scenarios were tested with artificial failure rates set far higher than realistic expectations, and in all cases every task was successfully completed, even when each agent was given a life expectancy of only 0.5-1.5 minutes.

In conclusion, the tenets of the Control *ad libitum* philosophy proved to be sound guidelines for developing a versatile and robust control architecture. Built using provably correct algorithms, the HAA implementation achieved all of its goals: dynamic formation of the initial team, run-time adaptability to changing resources in terms of host and avatars, and robustness against both hardware and software failure. And finally, due to the modularity of the system there appears to be significant potential for reuse of assets and future extensibility.

8.1 Future Work

Four potential avenues for future research have been identified: 1) making the framework available for open-source development, 2) standardizing agent interactions and recovery strategies, 3) exploring the issue of scalability, and 4) improving communication efficiency through learned database optimization.

8.1.1 Open-source Agent Library

One of the main goals of the framework is to be extensible and facilitate the reuse of control elements in order to allow future research to focus on new areas rather than retread existing work.

By making the source code available to everyone and providing a forum to exchange ideas and agent modules, the HAA implementation could become an ever growing repository of control systems for robotic teams. Researchers and developers would be able to take existing agents to quickly form a functioning control system, and then spend their efforts on creating the new agents required for their specific tasks/research. The first steps toward such a library would be a) cleaning up the code base to make it more transparent and streamline the process of running missions, b) providing documentation for the code, the functionality of each agent, and the process for creating new agents, c) replacing the custom simulation interface with one or more industry standard simulations, e.g., Microsoft® Robotics Developer Studio[96] or Player/Stage [97], and d) providing a web interface to facilitate the development and exchange of new agents. Standardizing agent interactions and recovery strategies, discussed below, would also be of great benefit to this initiative.

8.1.2 Standardizing Agent Interactions and Recovery Strategies

The HAA architecture does not specify any standard for the internal workings of agents or for their interactions. In Chapter 6 it can be seen that considerable effort must be devoted to designing recovery strategies for each agent, a process that is sometimes complicated by their interactions with other agents. However, during the design of the 18 agents developed for this research it became apparent that many agents have similar components and require similar recovery strategies. A method of standardizing agent interactions could be very useful in both designing the agent network and developing recovery strategies of individual agents.

8.1.3 Scalability

The performance of distributed algorithms almost always degrades with the size of the network. This was not apparent in the experiments in Chapter 7, where the performance of the atomic messaging, host membership service, and agent allocation algorithms scaled well to the system sizes tested, but ultimately there will be a point where scalability becomes a concern. Fortunately, with few exceptions, such as the failure detector, the system is designed without timeouts and therefore has significant flexibility it terms of latency. Thus, an interesting area of research would be to explore what those limits are, identify the primary bottlenecks, and develop strategies to work around these issues.

8.1.4 Communication Efficiency of the DDB

Related to the issue of scalability, there is a concern for communication efficiency, both in terms of latency and required bandwidth. In order to provide robustness some redundancy in the database is required, which has an unavoidable bandwidth cost. However, one of the major costs of executing queries is the data transfer cost between sites in the network [98], and so allocating the redundancy effectively can improve performance without increasing bandwidth costs. This shares many similarities to the data allocation problem that has been well treated for standard DDBs, which typically focus on reducing storage/transfer costs and minimizing response time [99]. The exact solution to this problem is NP-complete and therefore most of the work in this area is devoted to finding efficient near-optimal solutions [100,101]. Few papers in the field of robot teams mention the issue of data allocation at all, and those that do discuss only strategies where the data needed at each site is already known or can be computed based on knowledge of the system [37,102]. To the author's knowledge there is no research that attempts to study the dynamic data allocation problem under the unique demands of a distributed robot team. The determining factor that differentiates data allocation for a robot team from a standard distributed database is that data rapidly loses relevance as it ages. This leads to a scenario where it is critical that the initial allocation as data is created is optimal, and periodically redistributing data at a later time is unlikely to provide significant gains.

Bibliography

[1] L. Geunho and N. Young Chong, "Decentralized Formation Control for a Team of Anonymous Mobile Robots," in *6th Asian Control Conference 2006*, 2006, pp. 990-995.

[2] T. Yasuda, K. Ohkura, and K. Ueda, "A Homogeneous Mobile Robot Team That is Fault-tolerant," *Advanced Engineering Informatics*, vol. 20, no. 3, pp. 301-311, 2006.

[3] M.K. Hajjawi and A. Shirkhodaie, "Cooperative Visual Team Working and Target Tracking of Mobile Robots," in *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, 2002, pp. 376-380.

[4] C.C. Gava, R.F. Vassallo, F. Roberti, R. Carelli, and T.F. Bastos-Filho, "Nonlinear Control Techniques and Omnidirectional Visoin for Team Formation on Cooperative Robotics," in *2007 IEEE International Conference on Robotics and Automation*, 2007, p. 6.

[5] T. Balch, "Communication, Diveristy, and Learning: Cornerstones of Swarm Behavior," in *Swarm Robotics. SAB 2004 International Workshop*, 2005, pp. 21-30.

[6] L. Giannetti and P. Valigi, "Collaboration among Members of a Team: a Heuristic Strategy for Multi-Robot Exploration," in *Proceedings of the 14th Mediterranean Conference on Control and Automation*, 2006, p. 6.

[7] A. Howard, L.E. Parker, and G.S. Sukhatme, "Experiments with a Large Heterogeneous Mobile Robot Team: Exploration, Mapping, Deployment and Detection," *International Journal of Robotics Research*, vol. 25, no. 5-6, pp. 431-447, 2006.

[8] L. Wang, D. Zhang, G. Xie, and J. Yu, "Adaptive Task Assignment for Multiple Mobile Robots via Swarm Intelligence Approach," *Robotics and Autonomous Systems*, vol. 55, no. 7, pp. 572-588, July 2007.

[9] E. Jones et al., "Dynamically Formed Heterogenous Robot Teams Performing Tightly Coordinated Tasks," in *2006 Conference on International Robotics and Automation*, 2006, pp. 570-575.

[10] E. Prassler and K. Nilson, "1,001 Robot Architectures for 1,001 Robots," *IEEE Robotics & Automation Magazine*, vol. 16, no. 1, p. 113, March 2009.

[11] A. Tsalatsanis, A. Yalcin, and K.P. Valavanis, "Automata-based Supervisory Controller for a Mobile Robot Team," in *2006 IEEE 3rd Latin American Robotics Symposium*, 2006, p. 7.

[12] G. Lee and N. Y. Chong, "Decentralized Formation Control for a Team of Anonymous Mobile Robots," in *6th Asian Control Conference 2006*, 2006, pp. 990-995.

[13] M. Kutzer, M. Armand, D. Scheid, E. Lin, and G. Chirikjian, "Toward Cooperative Team-Diagnosis in Multi-Robot Systems," *International Journal of Robotics Research*, vol. 27, no. 9, pp. 1069-1090, September 2008.

[14] A. Palacios-Garcia, A. Munoz-Melendez, and E. Morales, "Collective Learning of Concepts using a Robot Team," in *Proceedings of the 7th International Conference on Informatics in Control, Automation and Robotics*, 2010, pp. 79-88.

[15] J. Madden and R. Arkin, "Modeling the Effects of Mass and Age Variation in Wolves to Explore the Effects of Heterogeneity in Robot Team Composition," in *2011 IEEE International Conference on Robotics and Biomimetics*, 2011, pp. 663-670.

[16] E. Gil Jones et al., "Dynamically Formed Heterogenous Robot Teams Performing Tightly Coordinated Tasks," in *Proceedings. 2006 International Conference on Robotics and Automation*, 2006, pp. 570-575.

[17] R. Mead and J. Weinberg, "Impromptu Teams of Heterogeneous Mobile Robots," in *Proceedings of the National Conference on Artificial Intelligence*, 2007, pp. 1890-1891.

[18] P. Dasgupta, K. Cheng, and L. Fan, "Flocking-Based Distributed Terrain Coverage with Dynamically-Formed Teams of Mobile Mini-Robots," in *Proceedings of the 2009 IEEE Swarm Intelligence Symposium*, 2009, pp. 96-103.

[19] W. Iida and K. Ohnishi, "Mobile Robot Teamwork for Cooperated Task," in *Proceedings of the Industrial Electronics Conference (IECON)*, 2002, pp. 1561-1566.

[20] E. Cervera, J. Sales, L. Nomdedeu, R. Marin, and V. Gazi, "Agents at play: Off-the-Shelf Software for Practical Multi-Robot Applications," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008, pp. 2719-2720.

[21] K. Skrzypczyk, "Control of a Team of Mobile Robots Based on Non-cooperative Equilibria with Partial Coordination," *International Journal of Applied Mathematics and Computer Science*, vol. 15, no. 1, pp. 89-97, 2005.

[22] J. Wen, H. Xing, X. Luo, and J. Yan, "Multi-agent Based Disributed Control System for an Intelligent Robot," in *Proceedings. 2004 IEEE International Conference on Services Computing*, 2004, pp. 633-637.

[23] S. Hasgul, I. Saricicek, M. Ozkan, and O. Parlaktuna, "Project-Oriented task Scheduling for Mobile Robot Team," *Journal of Intelligent Manufacturing*, vol. 20, no. 2, pp. 151-158, April 2009.

[24] A. Vlasov and A. Yudin, "Distributed Control System in Mobile Robot Applicaiton: General Approach, Realization and Usage," in *International Conference on Research and Education in Robotics*, 2011, pp. 180-192.

[25] A. Kasinski and P. Skrzypczynski, "Perception Network for the Team of Indoor Mobile Robots: Concept, Architecture, Implementation," *Engineering Applications of Artificial Intelligence*, vol. 14, no. 2, pp. 125-127, April 2001.

[26] K. Xu and P. Song, "A Coordination Framework for Weakly Centralized Mobile Robot Teams," in *2010 IEEE International conference on Information and Automation*, 2010, pp. 77-82.

[27] H. Mehrjerdi, M. Saad, and J. Ghommam, "Hierarchical Fuzzy Cooperative Control and Path Following for a Team of Mobile Robots," *IEEE/ASME Transactions on Mechatronics*, vol. 16, no. 5, pp. 907-917, October 2011.

[28] J. Nicolas and Y. Zhi, "Improved Trade-Based Multi-Robot Coordination," in *Proceedings of the 2011 6th IEEE Joint International Information Technology and Artificial Intelligence Conference*, 2011, pp. 500-503.

[29] U. Gurel and O. Parlaktuna, "A New Architecture for Multi-Robot Teams in Market-Based Applications," in *Proceedings of the 7th International Conference on Electrical and Electronics Engineering*, 2011, pp. 11430-11433.

[30] E. Martinson and R.C. Arkin, "Learning to Role-Switch in Multi-Robot Systems," in *2003 IEEE International Conference on Robotics and Automation*, 2003, pp. 2727-2734.

[31] Y. Wang and C. De Silva, "Multi-robot Box-pushing: Single-Agent Q-Learning vs. Team Q-Learning," in *IEEE International Conference on Intelligent Robots and Systems*, 2006, pp. 3694-3699.

[32] P.E. Rybski, S.A. Stoeter, M. Gini, D.F. Hougen, and N.P. Papanikolopoulos, "Performance of a Distributed Robotic System Using Shared Communications Channels," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 5, pp. 713-727, October 2002.

[33] C. McMillen et al., "Resource Scheduling and Load Balancing in Distributed Robotic Control Systems," *Robotics and Autonomous Systems*, vol. 44, no. 3-4, pp. 251-259, September 2003.

[34] S. Brown, M. Zuluaga, Y. Zhang, and R. Vaughan, "Rational Aggressive Behaviour Reduces Interference in a Mobile Robot Team," in *2005 12th International Conference on Advanced Robotics*, 2005, pp. 741-748.

[35] R. Emery, K. Sikorski, and T. Balch, "Protocols for Collaboration, Coordination and Dynamic Role Assignment in a Robot Team," in *Proceedings 2002 IEEE International Conference on Robotics and Automation*, 2002, pp. 3008-3015.

[36] H. Min and Z. Wang, "Group Escape Behavior of Multiple Mobile Robot System by Mimicking Fish Schools," in *2010 IEEE International Conference on Robotics and Biomimetics*, 2010, pp. 320-326.

[37] F. Santos, L. Almeida, P. Pedreiras, and L. Lopes, "A real-time distributed software infrastructure for cooperating mobile autonomous robots," in *International Conference on Advanced Robotics*, 2009, p. 6.

[38] H. Raj et al., "Spirits: Using Virtualization and Pervasiveness to Manage Mobile Robot Software Systems," in *Proceedings Second IEEE International Workshop Self-Managed Network Systems and Services*, 2006, pp. 116-129.

[39] M. Aicardi, "Coordination and Control of a Team of Moblie Robots," *WSEAS Transactions on Systems*, vol. 6, no. 6, pp. 1116-1123, June 2007.

[40] C.F. Touzet, "Robot Awareness in Cooperative Mobile Robot Learning," *Autonomous Robots*, vol. 8, no. 1, pp. 87-97, January 2000.

[41] E. Stump, A. Jadbabaie, and V. Mumar, "Connectivity Management in Mobile Robot Teams," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2008, pp. 1525-1530.

[42] J. Fink, A. Ribeiro, and V. Kumar, "Robust Control for Mobility and Wireless Communication in Cyber-Physical Systems with Application to Robot Teams," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 164-178, 2012.

[43] M. Li et al., "Architecture and Protocol Design for a Pervasive Robot Swarm Communication Networks," *Wireless Communications and Mobile Computing*, vol. 11, no. 8, pp. 1092-1106, August 2011.

[44] J. De Hoog, S. Cameron, A. Jimenez-Gonzalez, J. De-Dios, and A. Ollero, "Using Mobile Relays in Multi-Robot Exploration," in *Proceedings of the 2011 Australasian Conference on Robotics and Automation*, 2011, p. 8.

[45] G. Stambolov, T. Sperauskas, and R. Simutis, "Holonic Communication Structure in the Team of Cooperative Autonomous Mobile Robots," in *2008 Conference on Human System Interaction*, 2008, pp. 458-463.

[46] A. Cowley, H.-C. Hsu, and C.J. Taylor, "Distributed Sensor Databases for Multi-Robot Teams," in *2004 IEEE International Conference on Robotics and Automation*, 2004, pp. 691-696.

[47] H. Bistry and J. Zhang, "A Cloud Computing Approach to Complex Robot Vision Tasks using Smart Camera

Systems," in *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems*, 2010, pp. 3195-3200.

[48] A.A. Loukianov, H. Kimura, and M. Sugisaka, "Implementing Distributed Control System for Intelligent Mobile Robot," *Artificial Life and Robotics*, vol. 8, no. 2, pp. 159-162, 2004.

[49] T. Nishi, Y. Mori, M. Konishi, and J. Imai, "An Asynchronous Distributed Routing System for Multi-robot Cooperative Transportation," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 1730-1735.

[50] R. Quitadamo et al., "The PIM: An Innovative Robot Coordination Model Based on Java Thread Migration," in *Proceedings of the 6th International Conference on Principles and Practice of Programming in Java*, 2008, pp. 43-51.

[51] R. Madhavan, R. Lakaemper, and T. Kaimar-Nagy, "Benchmarking and Standardization of Intelligent Robotic Systems ," in *Proceedings of the International Conference on Advanced Robotics*, 2009, p. 7.

[52] OMG®. (2012, August) Robotics Domain Task Force. [Online]. http://robotics.omg.org/

[53] SAE International. (2012, August) SAE Standards. [Online]. http://standards.sae.org

[54] A. Martin and M. R. Emami, "Control ad libitum: an approach to real-time construction of control systems for unstructured robotic teams," in *The 14th IASTED International Conference on Robotics and Applications*, 2009.

[55] A. Martin and M. R. Emami, "Exploration and Mapping for Unstructured Robot Teams," in *8th IEEE International Symposium on Computational Intelligents in Robotics and Automation*, Korea, 2009, p. 6.

[56] R. Arkin, *Behavior-Based Robotics*. United States of America: The MIT Press, 1998.

[57] M. Mataric, "Behavior-Based Systems: Main Properties and Implications," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 1992, pp. 46-54.

[58] S. Waslander, G. Inalhan, and C. Tomlin, "Decentralized Optimization via Nash Bargaining," in *Theory and Algorithms for Cooperative Systems*, D. Grundel, R. Murphey, and P. Pardalos, Eds. Singapore: World Scientific Publishing, 2004, ch. 25, pp. 565-585.

[59] M. Weisfeld, *Object Oriented Thought Process*, 3rd ed. United States of America: Pearson Education, 2008.

[60] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability using Lazy Replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 360-391, November 1992.

[61] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 4th ed. Harlow, England: Pearson Education Limited, 2005.

[62] A. Martin and M. R. Emami, "A Dynamically Distributed Control Framework for Robot Teams," *Autonomous Agents and Multi-Agent Systems*, 2012, Submitted.

[63] F. Cristian, "Probabilistic Clock Synchronization," *Distributed Computing*, vol. 3, no. 3, pp. 146-158, 1988.

[64] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no. 2, pp. 374-382, April 1985.

[65] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225-267, March 1996.

[66] W. Chen, S. Toueg, and M. Aguilera, "On the Quality of Service of Failure Detectors," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 561-580, May 2002.

[67] R. Guerrauoi, M. Lerrea, and A. Schiper, "Non Blocking Atomic Commitment with an Unreliable Failure Detector," in *Proceedings of the 14th Symposium on Reliable Distributed Systems*, 1995, pp. 41-50.

[68] W. Chen, "On the Quality of Service of Failure Detectors," Cornell University, United States of America, PhD Thesis 2000.

[69] L. Johnson, S. Ponda, H.-L. Choi, and J. How, "Improving the Efficiency of a Decentralized Tasking Algorithm for UAV Teams with Asynchronous Communications," in *AIAA Guidance, Navigation, and Control Conference*, 2010, p. 22.

[70] S. Thrun, "Simultaneous Localization and Mapping," in *Robotics and Cognitive Approaches to Spatial Mapping*, M. E. Jefferies and W-K. Yeap, Eds. Berlin: Springer-Verlag, 2008, pp. 13-41.

[71] D. Rodriguez-Losada, F. Matia, A. Jimenez, and R. Galan, "Local Map Fusion for Real-time Indoor Simultaneous Localization and Mapping," *Journal of Field Robotics*, vol. 23, no. 5, pp. 291-309, April 2006.

[72] D. Rodriguez-Losada, P. San Segundo, F. Matia, and L. Pedraza, "Dual FastSLAM: Dual Factorization of the Particle Filter Based Solution of the Simultaneous Localization and Mapping Problem," *Journal of Intelligent and Robotic Systems*, vol. 55, no. 2-3, pp. 109-134, July 2009.

[73] T. Bailey, J. Nieto, and E. Nebot, "Consistency of the FastSLAM Algorithm," in *Robotics and Automation 2006.*

ICRA 2006, Orlando, 2006, pp. 424-429.

[74] H. Liu, L. Gao, Y. Gai, and S. Fu, "Simultaneous Localization and Mapping for Mobile Robots Using Sonar Range Finder and Monocular Vision," in *IEEE International Conference on Automation and Logistics*, Jinan, 2007, pp. 1602-1607.

[75] H.J. Chang, C.S.G. Lee, Y.H. Lu, and Y.C. Hu, "P-SLAM: Simultaneous Localization and Mapping With Environmental-Structure Prediction," *IEEE Transactions on Robotics*, vol. 23, no. 2, pp. 281-293, April 2007.

[76] A. Gil, O. Reinoso, M. Ballesta, and M. Julia, "Multi-robot visual SLAM using a Rao-Blackwellized particle filter," *Robotics and Autonomous Systems*, vol. 58, no. 1, pp. 68-80, January 2010.

[77] M. Wu, F. Huang, L. Wang, and J. Sun, "Cooperative Multi-Robot Monocular-SLAM using Salient Landmarks," in *International Asia Conference on Informatics in Control, Automation and Robotics*, 2009, pp. 151-155.

[78] M. Pfingsthorn and A. Birk, "Efficiently Communicating Map Updates with the Pose Graph," in *IEEE/RSJ International Conference on Intelligent Robotics and Systems (IROS)*, Nice, 2008, pp. 2519-2524.

[79] M. Rosencrantz, G. Gordon, and S. Thrun, "Decentralized sensor fusion with distributed particle filters," in *Uncertainty in Artificial Intelligence*, 2003.

[80] O. Cappe, S. Godsill, and E. Moulines, "An overview of existing methods and recent advances in sequential Monte Carlo ," *Proceedings of the IEEE*, vol. 95, no. 5, pp. 899-924, 2007.

[81] M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174-188, 2002.

[82] A. Doucet, S. Godsill, and C. Andrieu, "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering ," *Statistics and Computing*, vol. 10, pp. 197-208, 2000.

[83] R. Douc, O. Cappe, and E. Moulines, "Comparison of resampling schemes for particle filtering," in *4th International Symposium on Image and Signal Processing and Analysis*, 2005, pp. 64-69.

[84] J. Liu, J.-B. Wu, and D. Maluf, "Evolutionary Self-organization of an Artificial Potential Field Map with a Group of Autonomous Robots," in *1999 Congress on Evolutionary Computation*, 1999.

[85] A. Howard and L. Kitchen, "Generating sonar maps in highly specular environments ," in *Fourth International Conference on Control Automation Robotics and Vision*, 1996, pp. 1870-1874.

[86] S.-J. Lee et al., "Evaluation of Features through Grid Association for Building a Sonar Map," in *2006 IEEE International Conference on Robotics and Automation*, 2006, pp. 2615-2620.

[87] A. Martin and M. R. Emami, "Just-in-time Cooperative Simultaneous Localization and Mapping," in *International conference on Control, Automation, Robotics and Vision*, 2010, pp. 1-6.

[88] A. Howard, "Multi-robot Simultaneous Localization and Mapping using Particle Filters," *The International Journal of Robotics Research*, vol. 25, no. 12, pp. 1243-1256, 2006.

[89] M. Bryson and S. Sukkarieh, "Co-operative Localisation and Mapping for Multiple UAVs in Unknown Environments," in *2007 IEEE Aerospace Conference*, 2007, pp. 1-12.

[90] M. Walter and J. Leonard, "An Experimental Investigation of Cooperative SLAM," in *Fifth IFAC Symposium on Intelligent Autonomous Vehicles*, 2004.

[91] A. Martin and M. R. Emami, "Just-in-time Cooperative Simultaneous Localization and Mapping: A Robust and Efficient Particle Filter Approach," *International Journal of Robotics and Automation*, 2012, Submitted.

[92] Microsoft Corp. (2012, March) UUID structure (Windows). [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/aa379358(v=vs.85).aspx

[93] Wolfram Research Inc. (2012, August) Birthday Problem. [Online]. http://mathworld.wolfram.com/BirthdayProblem.html

[94] D. Maclay, "Simualiton gets into the Loop," *IEE Review*, vol. 43, no. 3, pp. 109-112, 1997.

[95] A. Martin and M. R. Emami, "A Fault-tolerant Approach to Robot Teams," *Robotics and Autonomous Systems*, 2012, Submitted.

[96] Microsoft®. (2012, October) Microsoft Robotics Developer Studio. [Online]. http://www.microsoft.com/robotics/

[97] Player Project. (2012, October) Player Project. [Online]. http://playerstage.sourceforge.net/

[98] A. S. Mamaghani, M. Mahi, M. R. Meybodi, and M. H. Moghaddam, "A Novel Evolutionary Algorithm for Solving Static Data Allocation Problem in Distributed Database Systems," in *2010 Second international Conference on Network Applications, Protocols and Services*, 2010, pp. 14-19.

[99] S. Rahmani, V. Torkzaban, and A. Haghighat, "A New Method of genetic Algorithm for Data Allocation in Distribed Database Systems," in *2009 First International Workshop on Education Technology and Computer Science*, 2009,

pp. 1037-1041.

[100] I. Hababeh, "Improving network systems performance by clustering distributed database sites," *Journal of Supercomputing*, p. 19, 2010, Article In Press.

[101] R. Mahmoudie and S. Parsa, "Data Allocation in Distributed Data Base in the Network Using Modularization Algorithm," in *12th International Conference on Computer Modelling and Simulation*, 2010, pp. 503-508.

[102] E. Nerurkar, S. Roumeliotis, and A. Martinelli, "Distributed Maximum A Posteriori Estimation for Multi-robot Cooperative Localization," in *IEEE International Conference on Robotics and Automation*, Kobe, 2009, pp. 1402-1409.

[103] T. Balch, "Hierarchic Social Entropy: An Information Theoretic Measure of Robot Group Diversity," *Autonomous Robots*, vol. 8, no. 3, pp. 209-238, June 2000.

[104] C.E. Shannon, "The Mathematical Theory of Communication," *Bell System Technical Journal* , vol. 27, no. 3, pp. 379-423, 1949.

[105] M. Perez, "An overview of Behavioural-based Robotics with simulated implementations on an Underwater Vehicle," Institut d'Informàtica i Aplicacions Universitat de Girona, Girona, PhD Thesis 2000.

[106] T. Browning, "Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions," *IEEE Transactions on Engineering Management*, vol. 48, no. 3, pp. 292-306, August 2001.

[107] K. Holtta-Otto and O. Weck, "Degree of Modularity in Engineering Systems and Products with Technical and Business Constraints," *Concurrent Engineering: Research and Applications*, vol. 15, no. 2, pp. 113-126, June 2007.

[108] J. Pandremenos, C. Chatzikomis, and G. Chryssolouris, "On the Quantification of Interface Design Architectures," *Asian International Journal of Science and Technology in Production and Manufacturing Engineering (AIJSTPME)*, vol. 2, no. 3, pp. 41-48, 2009.

[109] J. Bondy and U. Murty, *Graph Theory*, S. Axler and K. Ribet, Eds. New York, USA: Springer, 2008.

[110] M. Lazzaroni, L. Cristaldi, L. Peretto, P. Rinaldi, and M. Catelani, *Reliability Engineering: Basic Concepts and Applictions in ICT*. Berlin: Springer-Verlag, 2011.

[111] J. Nachlas, *Reliability Engineering: Probabilistic Models and Maintenance Methods*. United States of America: Taylor & Francis, 2005.

[112] J. Hoffmann, M. Jungel, and M. Lotzsch, "A Vision Based System for Goal-Directed Obstacle Avoidance," in *RoboCup 2004: Robot Soccer World Cup VIII*, D. Nardi et al., Eds.: Springer, 2004, pp. 418-425.

[113] D. Scharstein and A. Briggs, "Real-time Recognition of Self-Similar Landmarks," *Image and Vision Computing*, vol. 19, pp. 763-772, 2000.

Appendix I
Design, Development, and Performance Indexes

I.1 Adaptability

Adaptability is used here to represent the ability of the system to respond to changes in resource availability in real-time while continuing to achieve its goals. There are two major factors contributing to resource changes. The first is failure, hardware or software, planned or unplanned, which can occur at a disconcerting frequency in complex systems operating in real-world environments. The study in [38] reported a mean time between hardware failures in a robot team to be as low as 8 hours. The second factor is differences between installations. For large scale applications it may not be possible or practical to have the exact same setup across all installations.

An adaptability scorecard has been developed as a tool to score different system options or identify weaknesses in a system. The strategy is to first construct a checklist that is appropriate to the system and tasks it will be performing, then rate the adaptability in response to each item. The final score can be used to compare system options and low scores for individual items could indicate weaknesses in the system that could be remedied.

A list of common points of evaluation is provided here:

- Single/Few hardware failure/change/unavailability
  - Sensor, Communication, Processor, Mobility, Task specific hardware
- Multiple/Many hardware failure/change/unavailability
  - Sensor, Communication, Processor, Mobility, Task specific hardware
- Software failure
  - Non-critical component, Multiple non-critical components, Critical component, Multiple critical components
- Information failure
  - Data disagreement, Data corruption, Data uncertainty

I.2 Diversity

The concept of diversity can be applied to a number of areas of a robot team, including behavioural differences between team members[5]. As previously mentioned, these metrics are not meant to be exhaustive but merely to provide useful and interesting tools of evaluation, and so only five facets of diversity relating to the control system and robot hardware are considered here. A combined diversity index is developed below that considers robot hardware, population size, team structure, hardware flexibility, and population flexibility; it is also shown how the index can be customized to focus on any subset of the above facets. This distinction index can be applied to either fully realized team after the Implementation phase or to speculative teams after the Control Strategy phase.

This team diversity index is defined using the concept of Hierarchic Social Entropy (HSE), and follows the techniques developed in [103]. The HSE index builds on the concept of "simple" social entropy from Shannon's work in information theory [104], and maintains its six key properties, being: continuous, monotonic, recursive, lower bounded, globally maximal when diversity is evenly distributed, and lacking local maxima. The HSE index overcomes the inability of simple social entropy to account for the degree of difference between elements of a society, and provides a continuous and uniform measure of diversity. HSE can also distinguish differences in diversity between two systems regardless of scale [103].

We first define:

$R \equiv$ a set of N elements; $r_1, r_2, \ldots, r_N$

$C \equiv$ a set of M possibly overlapping subsets of the elements of R; $c_1, c_2, \ldots, c_M$

$p_i \equiv$ the proportion of elements in the ith subset

$|c_i| \equiv number\ of\ elements\ in\ set\ c_i$

$$p_i = \frac{|c_i|}{\sum_{j=1}^{M} |c_j|} \qquad \text{(I-1)}$$

$$\sum p_i = 1 \qquad \text{(I-2)}$$

The simple social entropy of system X can then be calculated using Shannon's formula [104]:

$$H(X) = -K\sum_{i=1}^{M} p_i \log_2(\mathrm{p_i}) \qquad \text{(I-3)}$$

where K is a positive constant that defines the scale of the measure, typically 1. In order for HSE to reflect the degree of diversity between elements, each element is represented as a point in A-dimensional space, where axes are defined to represent each of A relevant potential differences. Although it is not discussed in [103], it may be necessary to normalize and/or weight each axis in order to balance the diversity represented by that attribute. This is accomplished using normalizing and weight factors, η_a and w_a respectively, and applying them at each axis.

$$\text{For each axis } a\text{:}\quad r_i^*|_a = \frac{w_a}{\eta_a} r_i|_a \qquad \text{(I-4)}$$

There is no fixed procedure for determining the correct normalization and weight factors since it depends on the axes and how important their influence on diversity is considered. An intuitive starting point is to normalize each axis so that the distance between the minimum and maximum values of the expected range is approximately 1, and then weight the axis according to the priorities of the application. [103] then uses the clustering algorithm C_u, with a variable cluster level h to group the elements into subsets, c_i. The cluster level h is increased from 0 to infinity, and the simple social entropy calculated for the corresponding system, $H(R,h)$. The HSE is then defined in [103] as:

$$S(R) = \int_0^\infty H(R,h)\,dh \qquad \text{(I-5)}$$

For this application each robot is considered as a separate element/point, thus considering population size of different robot types, and axes typically represent a feature of the robot hardware, such as mass, acceleration, or number of sonar sensors.

I.3 Modularity

Modularity is generally considered as a desirable trait for any complex system. It can reduce the complexity of the design, reduce the effort required to develop each module, and allow modules to be reused within the system or in future systems. Proponents of behavioural control also put forward robustness and incremental design as other benefits of modularity [105]. Beyond hardware modularity, which can improve robustness and simplify repairs, developers of robot teams are more concerned about modularity of the software of the control system. In particular, this means identifying the fundamental modules of a control system and their connections to other modules, noting that the connections are not necessarily bidirectional. Once the modules and their connections are known it is convenient to form them into a Design Structure Matrix (DSM), which is popular for its use in a number of design and analysis methodologies [106]. Of interest here is using the DSM to quantify modularity, which can be done through several methods including, [107] and [108], among others. Each method has certain advantages and disadvantages, many of which are summarized in [108], and a combination of [107] and a modified version of [108] will be used here. The reasoning for this choice will be discussed below when the methodologies are explained.

Breaking a control system into its fundamental modules is not an obvious task, though two basic strategies stand out:

1. Code Level (analogous to an Architecture DSM [106]) – Treat each function in the code as a module and any function that calls this module is connected to it. Very basic functions might be ignored, and tightly knit groups of small functions might be combined into a single module.
2. Functionality Level (analogous to an Organization DSM [106]) – Instead of looking at functions in the code consider modules as areas of *functionality*. For example a basic mapping control system might have the modules mapping, exploring, sensor processing, path planning, and navigating.

Whatever module breakdown strategy is chosen, one should apply it consistently in order to allow comparisons between multiple control systems. In general, multiple instances of the same module should be considered as one, even if their connections appear different. Any change to that module could potentially impact all of the instances, and therefore the modules should be considered as a single entry with the combined connections of all instances. This may lead to seemingly counterintuitive results, for example it is possible to implement a centralized control algorithm in a more modular fashion than a distributed one.

Once the DSM has been constructed, modularity indexes can be calculated. [108] propose a two step process, first calculating a "Modularity Performance" (MP) index based on the number of connections in the DSM, and then calculating a "Modularity Type" standard deviation (STD) index that indicates whether the structure is closer to a bus-modular or chain-modular configuration. However, the MP index they describe is only suitable for symmetric binary DSMs and ultimately not significantly different than the non-zero fraction (NZF) index proposed in [107]. Both indexes measure the sparsity of the DSM, which is tied to modularity but does not paint the whole picture. The singular value modularity index (SMI) proposed in [107] claims to measure the degree of modularity and claims the following desirable properties:

1. SMI is theoretically bounded between 0 and 1.
2. SMI is independent of subjective module boundaries and ordering of rows and columns in the DSM.
3. SMI is largely scale-free, it can be calculated for systems of different sizes with the same architecture and will return nearly the same value. (In a series of simple experiments the authors observed that this is only true for moderate to large systems, $N > 15$, and appears to converge as N approaches infinity)

However, [108] points out a basic inconsistency where the SMI for an idealized bus-modular system is lower than that of a fully connected "integral" system. There are also inconsistencies when non-binary connections are used, for example a chain modular system with mixed weak and strong connections has a lower SMI than a chain modular system with either all strong or all weak connections. Despite these inconsistencies, [107] present a significant number of test cases that seem to validate SMI for moderate sized, real-world systems. The STD index developed by [108] also has all three properties listed above and yields the expected results in the case of idealized systems. Because of this the STD index will be used as the primary differentiator between systems with similar NZF, and SMI will be used to corroborate the results.

The following paragraphs explain how NZF, SMI and STD are calculated, using the formulation found in [107] and [108], respectively.

The NZF index is simply a measure of the sparsity of the DSM, with a value of 1 indicating all modules are connected to all other modules and a value of 0.29 being the minimum value for a binary system that has a connected graph, meaning a path can be found between any two modules [109]. Values lower than 0.29 are possible if non-binary connections or unidirectional connections are used. NZF can be calculated as

$$\mathrm{NZF} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}\mathrm{DSM}_{ij}}{N(N-1)} \qquad \text{(I-6)}$$

where DSM_{ij} represents the i,j element of the DSM and N is the number of modules.

The SMI approximates the decay rate of the singular values of the DSM. Taking the set of sorted singular values σ_1 to σ_N, SMI is calculated by finding the α factor that minimizes the difference between the singular values and the exponential decay function $e^{-(i-1)/\alpha}$.

$$\mathrm{SMI} = \frac{1}{N}\arg\min_{\alpha}\sum_{i=1}^{N}\left|\frac{\sigma_i}{\sigma_1} - e^{-(i-1)/\alpha}\right| \qquad \text{(I-7)}$$

The STD index measures the standard deviation of the number of connections of each module, then normalizes it based on the maximum and minimum standard deviation achievable for the total number of connections in the system, $\mathrm{STD}_{\max}$ and $\mathrm{STD}_{\min}$, respectively. The base STD is calculated by

$$\mathrm{STD} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(p_i - \bar{p})^2} \qquad \text{(I-8)}$$

where p_i is the weighted number of connections of module i and $\bar{p}$ is the average weighted number of connections in the system. [108] consider only bidirectional connections, and so in the case of DSMs with unidirectional connections the DSM should be made symmetric prior to calculating STD using

$$\mathrm{DSM}_{\mathrm{sym}} = \frac{(\mathrm{DSM} + \mathrm{DSM}^{\mathrm{T}})}{2} \qquad \text{(I-9)}$$

$\mathrm{STD}_{\max}$ and $\mathrm{STD}_{\min}$ can be found by arranging the connections in the symmetric DSM into configurations $\mathrm{DSM}_{\max}$ and $\mathrm{DSM}_{\min}$, where they are most condensed and most evenly distributed, respectively. [108] describes a general procedure that works well for binary DSMs, however in the non-binary case a more

complicated algorithm is required. Since the DSM is forced to be symmetric consider only the lower triangle of the matrix. Let A be an array of all elements in the triangle ordered highest to lowest. DSM_{max} can be constructed by filling each element column by column from the entries in A. The construction of DSM_{min} is much more complex due to the restriction of symmetry. A brute-force solution is computationally intractable for all but the smallest systems and no provable algorithmic solution presents itself. However, a strategy of taking alternating high-low-high-low elements of A and filling DSM_{min} starting with the innermost diagonal and moving outward has achieved the smallest STD in all tested cases. As long as STD_{min} is close to the true minimum the impact on the results is insignificant. Once STD_{max} and STD_{min} are known the normalized STD value can be calculated:

$$\mathrm{STD}_{\mathrm{norm}} = 1 - \frac{\mathrm{STD} - \mathrm{STD}_{\mathrm{min}}}{\mathrm{STD}_{\mathrm{max}} - \mathrm{STD}_{\mathrm{min}}} \qquad \text{(I-10)}$$

As a control set a number of test cases are presented in Table I-1, where STD demonstrates the expected results in all cases and the inconsistencies of SMI discussed above are apparent. A selection of DSMs from the control set is visualized in Fig. I-1.

I.4 Efficiency

Cost-return analysis and Return on Investment (ROI) are commonly used in business and financial analysis to judge the quality of investments or compare different project proposals. The same techniques can be applied to quantify the efficiency of a robot team. Depending on the nature of the application it may be possible to apply the techniques directly in terms of dollar value cost of the purchasing and operating the robot hardware and the monetary return of tasks they perform, or abstractions can be made to measure other factors. For example, if energy efficiency is the primary concern the cost can be measured in Joules and the return is the number and quality of tasks performed. A more efficient team is one that completes more tasks with the same or lower energy expenditure, and this is reflected in a higher ROI.

ROI is a ratio measure, calculated as

$$\mathrm{ROI} = \frac{\mathrm{return}}{\mathrm{cost}} \qquad \text{(I-11)}$$

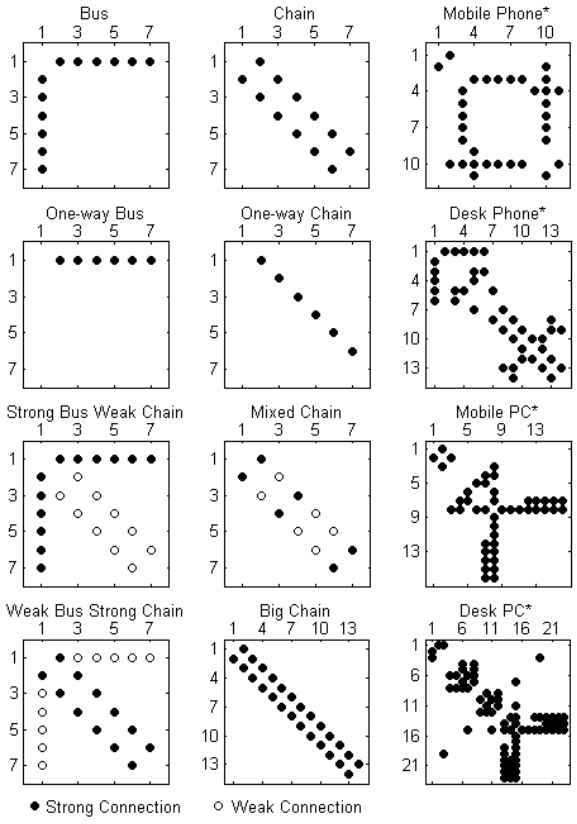
Whereas Gross Profit (GP) measures the absolute size of the return

$$\mathrm{GP} = \mathrm{return} - \mathrm{cost} \qquad \text{(I-12)}$$

Table I-1
Control Set Modularity Indexes

Case	N	SMI	NZF	MP	STD_{norm}
Integral	7	0.16	1	0	0
Bus	7	0.12	0.29	1	0
Chain	7	0.81	0.29	1	1
One-way Bus	7	0.11	0.14	1.2	0
One-way Chain	7	0.81	0.14	1.2	1
Weak Bus	7	0.11	0.14	1.2	0
Weak Chain	7	0.81	0.14	1.2	1
Mixed Bus	7	0.11	0.21	1.1	0
Mixed Chain	7	0.53	0.21	1.1	1
Big Bus	14	0.06	0.14	1	0
Big Chain	14	0.96	0.14	1	1
Strong Chain Weak Bus	7	0.5	0.4	0.83	0.81
Weak Chain Strong Bus	7	0.3	0.4	0.83	0.11
Desk Phone*	14	0.45	0.22	0.91	0.91
Mobile Phone*	11	0.23	0.29	0.87	0.33
Desk PC*	23	0.24	0.15	0.93	0.63
Mobile PC*	16	0.2	0.18	0.93	0.26

* Original DSM breakdown from **[107]**



* Original DSM breakdown from [107]

Fig. I-1 Control Set DSMs

Both values can be useful in different situations, and are easily calculated once the definitions for cost and return are in place. How cost and return are defined depends on the priorities of the application and can vary greatly in complexity. Costs typically fall into four categories:

1. *Initial Costs:* hardware, infrastructure, and setup.
2. *Operating Costs:* energy/fuel consumption and operator salaries.
3. *Repair Costs:* replacing and repairing hardware between missions.
4. *Opportunity Costs:* penalties for missing critical tasks, or downtime for a facility while the team working.

Returns are simply the value of each task performed, either monetary or a value abstraction. However, these values need not be constant and can be modified by various factors, such as: quality of service, time taken to complete the task, or when the task was completed if a time window was specified.

I.5 Persistency

Persistency is not about avoiding or eliminating failures, because for any complex system operating in real environments failures are inevitable. A truly robust control system should be able to recover from failures with minimal loss of data, maintaining the knowledge and learning acquired up to that point [38]. The question becomes what types of failures can occur, how are they handled, and when do they become critical failures where the system loses its ability to function. Mean time between critical failures (MTBCF) is a useful tool to evaluate persistency. As the name implies, MTBCF is simply the average uptime of the system before it suffers a critical failure. If the system is repaired and continues operation the downtime during repairs is not counted towards the next sample, only uptime is considered. It is also possible, indeed desirable, that the system does not encounter a critical failure during a mission, in which case the uptime continues accumulating over subsequent missions until a critical failure does occur.

As a comparator MTBCF is easy to apply: first define what constitutes a critical failure and then simply record the MTBCF over the lifetime of the system. MTBCF can then be compared between multiple systems and used as an indicator of robustness and persistency.

When used as a predictor the goal is different. Rather than attempting to predict what the MTBCF will be, the goal is to identify the most probable causes for critical failure so that they can be planned for and handled more robustly. Detailed methodologies for constructing reliability models are beyond the scope of this thesis but many techniques can be found in reliability handbooks and textbooks [110,111]. The general premise is to break the system into components and construct a network of their relationships. Then the failure probabilities of each component can be used to generate a failure probability for the entire system. This process can be used at the system implementation level to identify weaknesses and areas for improvement, or at the system architecture or strategies levels to compare different approaches and identify fundamental limitations. An example of persistency analysis can be found in Section 6.2.2.1.

Appendix II
HAA Algorithms

Algorithm 1 Unreliable Failure Detector (HAA-UFD)

Process *p* (Observed):

1 Initialization:
2 *period* := INITIAL_PERIOD;
3 *nextPeriod* := INITIAL_PERIOD;
4 ${}^{p}\tau_0$:= the local time;
5 l := 0;

6 **upon** ${}^{p}\tau_l$ = the local time:
7 *period* := *nextPeriod*;
8 send heartbeat m_l := { l, *period*, ${}^{p}\tau_l$ };
9 ${}^{p}\tau_{l+1}$:= ${}^{p}\tau_l$ + *period*;
10 l := l + 1;

11 **upon** receive message setPeriod = { *newPeriod* }
12 *nextPeriod* := *newPeriod*;

Process *q* (Observer):

13 Initialization:
14 initialize PF parameters { η, α };
15 ${}^{q}\tau_0$:= the local time;
16 l := -1;

17 **upon** ${}^{q}\tau_{l+1}$ = the local time:
18 *proposal* := SUSPECTED;

19 **upon** receive message m_j = { j, *period*, ${}^{p}\tau_j$ } at time t:
20 If $j > l$ then
21 l := j;
22 ${}^{q}\tau_{l+1} := EA_{l+1} + \alpha$;
23 If t < ${}^{q}\tau_{l+1}$ then
24 *proposal* := TRUSTED;

25 **after** every *N* heartbeat messages received:
26 re-evaluate PF parameters { η, α };
27 send setPeriod message := { η };

Algorithm 2 Totally Ordered Atomic Commit (HAA-OAC)

Note: the notation $\Diamond S_p$ indicates the set of processes suspected by *p's* failure detector.

1 $activeTransactions_p$: list; /* List of transactions that the local process is participating in that are undecided */
2 $decidedTransactions_p$: list; /* List of transactions that have been decided */
3 **procedure** startTransaction(*id*, *transaction*, *participants*):
4 *order* := highest order of ($activeTransactions_p$ U $decidedTransactions_p$) + 1; /* move to back of queue */
5 send(*id, transaction*, *participants*, *order*) to all participants;

6 **upon** receive (*id, transaction*, *participants*, *order*):
7 **if** ($decidedTransactions_p$[*id*] = NULL) **then**
8 atomicCommit(*id, transaction*, *participants*, *order*);

9 **procedure** atomicCommit(*id, transaction*, *participants, order*):
10 *outcome* : { *commit*, *abort* };
11 $state_p$: { *decided*, *undecided* };
12 est_p : { *pre-abort*, *pre-commit* }; /* Estimate */
13 r_p : integer; /* Round */
14 ts_p : integer; /* Timestamp */
15 $order_p$: integer; /* Commit order */

16 Initialization:
17 $state_p$:= *undecided*;
18 est_p := *pre-abort*;
19 r_p := -1;
20 ts_p := 0;
21 **if** ($activeTransactions_p$[*id*] = NULL) **then** *order* := $activeTransactions_p$[*id*].*order*; /* Have updated order */
22 **if** [(*order*, *id*) < highest in ($activeTransactions_p$ U $decidedTransactions_p$)] **then** /* Confirm order */
23 $order_p$:= highest order of ($activeTransactions_p$ U $decidedTransactions_p$) + 1; /* back of queue */
24 send(*id, p,* $order_p$) to all participants;
25 **else** $order_p$:= *order*;
26 $activeTransactions_p$[*id*] := { *id, transaction*, *participants*, $order_p$ }; /* Add to active transactions */
27 *vote* := evaluateTransaction(*transaction*);
28 **if** *vote* = *no* **then** /* Unilateral abort */
29 send(*id*, *p*, -1, r_p, *abort*, *decide*) to all participants;
30 **begin** Task 1;
31 **else**
32 **cobegin** Task 1|| Task 2 **coend**; /* Concurrent */

33 Task 1:
34 **wait until** receive (*id*, -, *order*, -, *outcome*, *decide*);
35 $state_p$:= *decided*;
36 send(*id, p*, *order*, -, *outcome*, *decide*) to all participants; /* Notify all of decision */
37 **wait until** [(*order*, *id)* is lowest in $activeTransactions_p$ **or** *outcome* = *abort*];
38 decide(*transaction*, *outcome*);
39 $activeTransactions_p$[*id*] := NULL;
40 $decidedTransactions_p$[*id*] := { id, order };
41 **wait until** [for *n* participants *q*: received (*id*, *q*, -, -, *outcome*, *decide*) from *q* or *q* is permanently ∈ $\Diamond S_p$];
42 $decidedTransactions_p$[*id*] := NULL;

```
43    Task 2:
44      while state_p = undecided
45        r_p := r_p + 1;
46        coord := p_(rp mod n) + 1;
47        send( id, p, order_p, r_p, est_p, ts_p) to coord; /* P1 */
48        if ( p = coord ) then/* Step C1 */
49          wait until [(for n –f participants q: received (id, q,
    order_p, r_p, est_q, ts_q) from q) and (for n participants q:
    received (id, q, order_p, r_p, est_q, ts_q) from q or q ∉ ◊S_p)];
50            msgs_p[r_p] = {(id, q, order_p, r_p, est_q, ts_q) such that
    p received (id, q, order_p, r_p, est_q, ts_q) from q};
51            if |msgs_p[r_p]| = n then
52              est_p := pre-commit;
53            else
54              t := largest ts_q such that (id, q, order_p, r_p, est_q,
    ts_q) ∈ msgs_p[r_p];
55              est_p := select one est_q such that (id, q, order_p,
    r_p, est_q, t) ∈ msgs_p[r_p];
56            send (id, p, order_p, r_p, est_p) to all participants;
57        wait until [received (id, coord, order_p, r_p, est_coord)
    from coord or coord ∉ ◊S_p]; /* Step P2 */
58          if received (id, coord, order_p, r_p, est_coord) then
59            est_p := est_coord;
60            ts_p := r_p;
61            send (id, p, order_p, r_p, ack) to coord;
62          else
63            send (id, p, order_p, r_p, nack) to coord;
64        if p = coord then   /* Step C2 */
65          wait until [for n - f participants q: received (id, q,
    order_p, r_p, ack) or (q, order_p, r_p, nack)];
66            if [for n – f participants q: received (id, q,
    order_p, r_p, ack)] then
67              state_p := decided;
68              if outcome = pre-commit then
69                send (id, p, order_p, r_p, commit, decide) to
    all participants;
70              else
71                send (id, p, -1, r_p, abort, decide) to all
    participants;

72    upon receive( id, q, order_q, …); /* Check the order */
73      if ( activeTransactions_p[id] = NULL and
    decidedTransactions_p[id] = NULL ) then
74        order_p := order_q;
75        activeTransactions_p[id] := { id, order_p}; /* add */
76      if order_q > order_p then
77        order_p := order_q; /* accept new order */
78        if Task 2 is active then
79          send( id, p, order_p ) to all participants;
80          est_p := pre-abort;
81          r_p := -1;
82          ts_p := 0;
83          abort Task 2; /* Reset Task 2 */
84          begin Task 2;
```

Proofs for HAA-OAC

Proof that, for $f < n/2$, the original algorithm satisfies the conditions AC-Uniform-Agreement, AC-Uniform-Validity, AC-Termination, and AC-Non-Triviality is provided in [67], along with the note that if $f \geq n/2$ the algorithm fails to terminate but still prevents participants from reaching different decisions. It can readily be observed that none of the modifications affect AC-Uniform-Agreement, AC-Uniform-Validity, or AC-Non-Triviality. However, a short proof is provided to show that AC-Termination still holds and then a proof for AC-Total-Order is derived.

Lemma OAC-1: Every correct participant eventually reaches l. 37 of Task 1.

Since the behaviour of Tasks 1 and 2 is the same as [67] until l. 37 of Task 1, the previous proof of AC-Termination guarantees that for transaction m every correct participant will eventually reach l. 37 of Task 1.

Proof of AC-Termination:

There always exists a transaction m_{low} that is the lowest order of $activeTransactions_p$ and by Lemma OAC-1 every correct participant of m_{low} will eventually reach l. 37 of Task 1. Since m_{low} has the lowest order of $activeTransactions_p$ it is free to proceed immediately no matter what the outcome, at which point m_{low} is removed from $activeTransactions_p$ and the next transaction in line becomes m_{low}. Since no participant will add a new transaction with lower order than any of its current transactions, for correct participant p with an active transaction m waiting at l. 37 of Task 1 there are a finite number of current transactions with lower order and so m will eventually become m_{low} and terminate.

Lemma OAC-2: Once a transaction m has reached the pre-commit stage its order o will never change.

Proof: Each participant proposes an order when and only when they first learn of the transaction (ll. 23-25 or 76-79), and there exists a participant p^+ who proposes an order $o^+ \geq$ all proposed orders. From (l. 78), p^+ will never change its proposal since all other proposals must be $\leq o^+$. To reach the pre-commit stage (l. 52) every participant must agree on an order o, and so it follows that if the pre-commit stage is reached o must equal o^+, the highest proposed order.

Proof of AC-Total-Order:

Proof by contradiction. Assume correct process p commits transaction m and then commits transaction m', while correct process q commits transaction m' and then commits transaction m. Without loss of generality, assume that p commits m (Event 1) before q commits m' (Event 2), and that q committing m (Event 3) can occur before or after p commits m'. For m to reach the pre-commit stage (l. 52) there must exist an order o that both p and q agreed to, and by Lemma OAC-2 o will never change after reaching this point. Similarly, for m' to reach the pre-commit stage there must exist an order o' than both p and q agreed to that will never change. There are four possible conflict cases:

1. p and q believe m and m' conflict.

Contradiction: Since p believes m and m' conflict it follows that $o < o'$. Since q also agreed to o and o' it cannot commit m' before committing m.

2. p believes that m and m' conflict, but q does not.

Contradiction: For Event 1 to occur q must have agreed to commit m and thus be aware of m prior to Event 1. Since Event 2 occurs after Event 1, at the time of Event 2 it is impossible for q to believe m and m' do not conflict.

3. q believes that m and m' conflict, but p does not.

Contradiction: In order to commit m' q must have received agreement from p. For p to believe that m and m' do not conflict p must have committed m (Event 1) and discarded all knowledge of m (l. 42) before learning about m' (which must occur before Event 2). In order to reach l. 42 p must have received a message from q stating that q has committed m (l. 41). Since Event 3 happens after Event 2, at the time of Event 2 it is impossible for p to think m and m' do not conflict.

4. p and q believe m and m' do not conflict.

Contradiction: For p to believe that m and m' do not conflict p must have committed m before learning about m'. Conversely, for q to believe that m and m' do not conflict q must have committed m' before learning about m. This is impossible, since Event 1 occurs before Event 2, q must know about m before committing m'.

Algorithm 3 Host Membership Service (HAA-HM)

Note: the notation $\Diamond S_p$ indicates the set of processes suspected by p's failure detector.

```
1  coreProcesses : list /* list of "core" processes */
2  joinList_p : list /* list of applicants waiting to join */
3  memberList_p : ordered list /* list of members, ordered by
   insertion order */
4  removeList_p : list /* list of members who are suspected */
5  leaveList_p : list /* list of members who wish to leave */
6  locked_p : universally unique id /* flags whether the lock is
   set and stores the current key */
7  updatingMembers_p : boolean (FALSE) /* flags whether p is
   currently trying to update membership */
8  connectionsTo_p : list /* processes p has connections to */
9  connectionsFrom_p : list /* processes p has connections
   from */
10 sponsor_p : id /* id of p's sponsor */
11 sponsee_p : list /* list of applicants p is sponsoring */
12 groupCorrect_p : boolean (FALSE) /* group formed */
13 coreMembers_p : list /* list of all core members that have
   been part of the group */

14 procedure groupLeave(): /* Leave request */
15   send (leave, p) to memberList_p;
16   wait until p is removed from memberList_p;
17     stop participating in all group activities;

18 upon receiving (leave, q):
19   if (q ∈ memberList_p ∪ joinList_p) then
20     insert q into leaveList_p;
21     updateMembership();

22 upon suspecting process q: /* Remove request */
23   if (q ∈ memberList_p) then
24     insert q into removeList_p;
25     updateMembership();

26 upon trusting process q:
27   if (q ∈ joinList_p) then
28     updateMembership();
29   else if (q ∈ removeList_p) then
30     remove q from removeList_p;
31     updateMembership();

32 upon permanently suspecting process q:
33   if (q ∈ joinList_p) then
34     remove q from joinList_p;

35 procedure groupJoin(): /* Join request */
36   for each q in coreProcesses
37     open connection to q;
38     insert q into connectionsTo_p;
39     send (apply, p) to q;
40   coreHead := first c_H elements of coreProcesses;
41   if (p = front(coreProcesses)) then
42     locked_p := key := unique id;
43     OAC (membership, key, p, p) to p; /* group of one */
44   else if (p ∈ coreHead) then
45     wait until FORMATION_TIMEOUT has elapsed;
46       while (p ∉ memberList_p)
47         newMemberList := coreHead - (coreHead ∩ ◊S_p);
```

```
48          OAC(formationFallback, p, newMemberList) to
    newMemberList;
49          wait until OAC(formationFallback, p,
    newMemberList) is decided;

50  upon receiving (apply, a):
51    insert a into joinList_p;
52    insert a into connectionsFrom_p;
53    open connection to a;
54    insert a into connectionsTo_p;
55    send (introduce, p, p) to a;
56    send (introduce, a, p) to memberList_p ∪ joinList_p;
57    if ( p = front(memberList_p) ) then /* we are the
    undisputed leader */
58      insert a into sponsee_p;
59      send (sponsor, p) to a;
60      send current global state to a;
61      begin forwarding global state changes to a;

62  upon receiving (introduce, a, q):
63    if ( a = q ) then
64      insert q into connectionsFrom_p;
65    if (a ∉ connectionsTo_p) then
66      open connection to a;
67      insert a into connectionsTo_p;
68      send (introduce, p, p) to a;
69    if (p ∉ memberList_p and sponsor_p != NULL) then
70      send (connections, p, connectionsTo_p ∩
    connectionsFrom_p) to sponsor_p;

71  upon receiving (sponsor, q):
72    sponsor_p := q;
73    send (connections, p, connectionsTo_p ∩
    connectionsFrom_p) to sponsor_p;

74  upon receiving (connections, q, connectionList ):
75    updateMembership();

76  procedure updateMembership():
77    if (updatingMembers_p = TRUE or OAC(remove, …) in
    progress or OAC(membership, …) in progress) then
78      return;
79    if (p = front(memberList_p - removeList_p)) then /* we
    think we could be the leader */
80      acceptList := EMPTY; /* applicants to accept */
81      potentialList := EMPTY; /* applicants who are ready */
82      curMemberList := memberList_p - (removeList_p ∪
    leaveList_p);
83      for each applicant a in joinList_p:
84        if [a ∈ ◊S_p or a ∉ sponsee_p or (groupCorrect_p =
    FALSE and a ∉ coreProcesses)] then /* disqualified */
85          continue;
86        if (connections of a ∩ curMemberList =
    curMemberList) then
87          insert a into potentialList;
88      for each applicant a in potentialList:
89        if (connections of a ∩ acceptList = acceptList) then
    /* a is connected to everyone already accepted */
90          potentialList := potentialList ∩ (a ∪ connections
    of a); /* remove anyone a is not connected to */
91          insert a into acceptList;
92      if (groupCorrect_p = FALSE and count(acceptList) < c_f)
    then /* make sure we can form a correct group */
93        acceptList := EMPTY;
94      if (acceptList ∪ removeList_p ∪ leaveList_p != EMPTY)
    then
95        activeList := memberList_p - removeList_p; /* members
    we expect a response from */
96        removalList := removeList_p;
97        updatingMembers_p := TRUE;
98        if (acceptList ∪ leaveList_p = EMPTY) then /* only
    removal, don't need to lock */
99          key := NULL;
100       else
101         key := unique id;
102       OAC (remove, key, p, removalList) to activeList;
103       wait until (OAC(remove, NULL, p, removalList) is
    decided or OAC(remove, key, p, removalList) is aborted
    or [(received (locked, key, q) or q □ ◊S_p) for all q □
    activeList and all active OACs are decided]);
104         if (key != NULL and received (locked, key, q) for
    all q □ activeList) then
105           newMemberList := (memberList_p ∪ acceptList)
    - leaveList_p;
106           OAC (membership, key, p, newMemberList) to
    activeList ∪ acceptList;
107         else
108           updatingMembers_p := FALSE;
109           updateMembership();

110 upon receiving OAC(remove, key, q, removalList):
111   if (q != front(memberList_p - removeList_p)) then
112     vote no; /* unilateral abort: not accepted leader */
113   else if (removalList != removeList_p ∩ removalList) then
114     vote no; /* unilateral abort: don't agree with list */
115   else
116     vote yes;  /* can proceed */

117 upon committing OAC (remove, key, q, removalList):
118   memberList_p := memberList_p - removalList;
119   removeList_p := memberList_p ∩ ◊S_p; /* reset */
120   leaveList_p := memberList_p ∩ leaveList_p; /* reset */
121   if (p = front(memberList_p)) then /* leader */
122     for each applicant a in joinList_p - (joinList_p ∩
    sponsee_p):
123       insert a into sponsee_p;
124       send (sponsor, p) to a;
125       send current global state to a;
126       begin forwarding global state changes to a;
127   locked_p := key;
128   if (key != NULL) then /* we need to lock */
129     hold all future global state changes;
130     send (locked, key, p) to q;
131   else
132     propose held global state changes;
133   updateMembership();
```

```
134 upon aborting OAC (remove, key, q, removalList):
135   updateMembership();

136 upon committing OAC(membership, key, q,
    newMemberList):
137   if ( q = p ) then
138     stop forwarding global state changes to all a ∈
    (newMemberList ∩ sponsee_p);
139     sponsee_p:= sponsee_p - (newMemberList ∩ sponsee_p);
140   joinList_p := joinList_p - (newMemberList ∩ joinList_p);
141   memberList_p := newMemberList;
142   removeList_p := memberList_p ∩ ◊S_p; /* reset */
143   leaveList_p := memberList_p ∩ leaveList_p; /* reset */
144   locked_p := NULL;
145   propose held global state changes;
146   if (groupCorrect_p = FALSE) then
147     coreMembers_p:= coreMembers_p ∪ memberList_p;
148     if (count(coreMembers_p) > c_f ) then
149       groupCorrect_p:= TRUE;
150   updatingMembers_p := FALSE;
151   updateMembership();

152 upon aborting OAC(membership, key, q,
    newMemberList):
153   if (locked_p = key ) then
154     locked_p := NULL;
155     propose held global state changes;
156     updatingMembers_p := FALSE;
157     updateMembership();

158 upon receiving OAC(formationFallback, q,
    newMemberList):
159   coreHead := first c_H elements of coreProcesses;
160   if (memberList_p != EMPTY) then
161     vote no; /* unilateral abort: group already exists */
162   else if (newMemberList != coreHead - (coreHead ∩
    ◊S_p)) then
163     vote no; /* unilateral abort: don't agree with list */
164   else
165     vote yes;  /* can proceed */

166 upon committing OAC(formationFallback, q,
    newMemberList):
167   if (memberList_p = EMPTY) then /* only accept one
    formation message */
168     joinList_p := joinList_p - (newMemberList ∩ joinList_p);
169     memberList_p := newMemberList;
170     removeList_p := memberList_p ∩ ◊S_p; /* reset */
171     leaveList_p := memberList_p ∩ leaveList_p; /* reset */
172     if (p = front(memberList_p)) then /* we are the
    undisputed leader */
173       for each applicant a in joinList_p - (joinList_p ∩
    sponsee_p):
174         insert a into sponsee_p;
175         send (sponsor, p) to a;
176         send current global state to a;
177         begin forwarding global state changes to a;
178     groupCorrect_p:= TRUE;
179     updateMembership();
```

Proofs for HAA-HM

First a proof of HM-Formation is provided. This guarantees that a correct group is eventually formed, which is necessary for many of the remaining proofs.

Lemma HM-1: When a member calls updateMembership() it will be called iteratively until the member is satisfied that all updates are complete or the member crashes.

Proof: A call to updateMembership() has four return paths:

a) In progress abort (l. 78). This means that another call to updateMembership() is in progress along return path d, or that the member is currently participating in either a *remove* or *membership* transaction. For the second case, all possible outcomes of *remove* and *membership* transactions include calls to updateMembership().

b) Not leader abort (l. 79). The member does not consider itself to be the leader and is not responsible for updates.

c) No updates abort (l. 94). The member does not have any updates to make. It is possible that a join request may be temporarily denied due to suspicions (l. 84) or incomplete connections (l.86), and in either case updateMembership() is called when those conditions change, l. 26 and l. 74, respectively.

d) Attempt updates (ll. 95-109). The member attempts to make the updates and eventually calls updateMembership() (l. 109), or obtains a lock with *key** and sends an OAC(*membership*, *key**, ...) message. To obtain a lock the member must be the acknowledged leader, and therefore the member will never accept a lock from another member. Thus, upon deciding OAC(*membership*, *key**, ...) the member will call updateMembership() (l. 151 or l. 157).

Lemma HM-2: The probability that update attempts from a member calling updateMembership() will fail indefinitely approaches 0 with increasing time, and therefore if the member does not crash every individual update from that member will either eventually be committed or rescinded.

Proof: Update attempts can fail for three reasons:

a) A process crashes, thus preventing OAC(*remove*, ...) or OAC(*membership*, ...) from being committed, or preventing a (*locked*, ...) message from being delivered.

In this case the process will eventually be suspected and be added to the remove update or removed from the membership update.

b) A member disagrees with the proposed *removalList* (l. 113). This may happen if the leader has falsely suspected a member, or the dissenting member has not yet suspected a crashed process. Both situations are temporary and will eventually be corrected by their respective members.

c) A member does not accept the proposer as leader (l. 111). This may happen if the proposer has incorrectly assumed leadership due to false suspicions, or the dissenting member has not yet suspected crashed processes that would lead to the promotion of the proposer. Both situations are temporary and will eventually be corrected by their respective members.

Since all three situations are resolved with time, no individual cause for failure can prevent updates indefinitely. Thus, for updates to be prevented indefinitely there must be an unending chain of overlapping (or closely packed) causes. Since the number of current group members is finite, the number of potential crashed processes is also finite, and every crashed process will eventually be suspected by every correct process. Therefore the unending chain of causes must be built with false suspicions of a leader. Considering each instant when the leader attempts an update, the probability that it will have no false suspicions is $P_A{}^{n-1}$. The probability that there is a false suspicion at each time the leader attempts an update rapidly approaches 0 with time.

Lemma HM-3: No incorrect group is ever formed.

Proof: Groups can be formed in two ways:

1. Formed by the first core process via an OAC(*membership*, ...) transaction. This first membership transaction must include at least c_f applicants (l. 92), and so including the first core process there are at least $c_f + 1$ members. Therefore the group cannot be incorrect.

2. Formed via an OAC(*formation*, ...) transaction. All formation transactions propose a list of unsuspected processes out of the first c_H core processes, the "core head," which must be agreed on by every process in the list in order to be committed. For an incorrect group to be formed at least $c_H - c_f$ correct processes must be erroneously excluded from a successful formation transaction. As discussed above, the probability of an erroneous exclusion, P_{EF}, is insignificant and is one of the failure conditions for the algorithm. Therefore no incorrect group will be formed during a valid run.

Lemma HM-4: If the first core process, c^*, is correct then the first core process eventually forms a correct group.

Proof: Since c^* is correct it will eventually form two-way connections with every other correct core process, of which there are at least c_f. From Lemma HM-1, c^* will continue calling updateMembership() and eventually form an *acceptList* with at least c_f applicants (l. 92). From Lemma HM-2, the join update will eventually be committed. Thus, a group with more than c_f members will be formed, which is by definition a correct group.

Lemma HM-5: If the first core process is incorrect and fails in forming a correct group then the first $c_f + 1$ processes eventually form a correct group.

Proof: If the first core process has failed to form a correct group by FORMATION_TIMEOUT, the first c_H processes begin sending OAC(*formation*,...) transactions. Each transaction proposes a complete list of unsuspected processes in the core head. Eventually all crashed processes are suspected by every correct process, and since formation transactions are repeated until one is committed, false suspicions cannot prevent formation indefinitely. Therefore a group is eventually formed, and from Lemma HM-3, this group must be correct.

Proof of HM-Formation: From Lemmas HM-3, HM-4, and HM-5.

The remaining proofs are provided under the assumption that a correct group has been formed.

Lemma HM-6: If a leader crashes another member will assume leadership.

Proof: When the leader crashes it is eventually suspected by all correct members. Upon suspecting a process a member will attempt to assume leadership if every member of higher rank in the group is suspected. If those suspicions are correct the new leader will eventually succeed in removing those members by

Lemma HM-2 and be acknowledged as the leader. Since we know there is at least one correct core member, there will always be a leader.

Lemma HM-7: If a correct process becomes the leader it will remain the leader.

Proof: Since no erroneous removals occur, a leader can only be removed if it crashes. Therefore once a correct process becomes the leader it will not be removed.

Lemma HM-8: If every alive member up to and including the highest ranked correct core member is aware of an update that update will eventually be committed.

Proof: The set of potential leaders is the highest ranked correct core member, *m**, and every higher ranked member, since there is always a leader (Lemma HM-6) and if every higher ranked member crashes *m** eventually becomes the leader and remains the leader (Lemma HM-7). Every potential leader is aware of the update and that leader will either eventually commit the update or crash (Lemma HM-2). If every higher ranked member crashes, *m** will eventually commit the update.

Lemma HM-9: When a leader suspects a member, if the leader does not crash, either that member will eventually be removed or the member will become trusted again.

Proof: Upon suspecting a member the leader calls updateMembership(). From Lemma HM-1 updateMembership() iterates until all updates are complete. From Lemma HM-2, individual updates are eventually committed during a call to updateMembership(), therefore either the suspect is removed and the update is completed or the suspect becomes trusted and the update is rescinded.

Proof of HM-Non-Triviality-Remove: When a member crashes it is eventually permanently suspected by every alive member, notably including the highest ranked correct core member and every member of higher rank. Suspecting a member is equivalent to being aware of a remove update, and therefore by Lemma HM-9 the remove update will eventually be committed.

Proof of HM-Non-Triviality-Leave: When a correct member asks to leave the group they notify all current members, notably including the highest ranked correct core member and every member of higher rank. By Lemma HM-8 the leave update will eventually be committed. When an incorrect member asks to leave they will either be removed by the same path as a correct member, or crash and be removed by HM-Non-Triviality-Remove.

Lemma HM-10: A leader will eventually be satisfied that each correct applicant is ready to join, or crash.

Proof: In order to join an applicant *a* must meet three conditions (l. 84):

1. *a* must have received all pre-lock data.

To ensure this the leader takes responsibility for sending pre-lock data to all applicants. Upon accepting leadership the leader becomes a sponsor to all current applicants and sends them the pre-lock data (ll. 122-126, 173-177). Similarly, when a new applicant applies the leader becomes their sponsor and sends them the prelock data (ll. 58-61).

2. *a* must have two-way connections to all other members and applicants that will be added at the time of their acceptance.

To ensure this every member who receives an application introduces the new applicant to every member and every current applicant. Every process that receives an introduction will connect to the applicant or crash, in which case they will be removed from the group/applicant pool. Similarly, every later applicant will be introduced to *a* and *a* will connect to each of these processes. In this way *a* establishes connections to every member and every other applicant, and notifies the leader of their connection status (l. 74).

3. *a* must not be suspected.

Since *a* is correct it will eventually be trusted.

If the leader does not crash they will eventually be convinced the applicant is ready to join.

Proof of HM-Non-Triviality-Join: When a correct applicant asks to join the group they notify all core members, including the highest ranked correct core member, *m**. Since *m** forwards the application to every higher ranked non-core member, every member of higher rank than *m** is aware of the join request. Being aware of the join request is not sufficient to be considered for a join update, since there are three other join conditions that must be met. Lemma HM-10

guarantees that each potential leader will either eventually be satisfied these join conditions are met or crash, and therefore by Lemma HM-8 the join update will eventually be committed.

Proof of HM-Weak-Validity: To remove a member the leader must successfully commit an OAC(*remove*, ..., *removalList*) transaction, where *removalList* is the set of members being removed. One of the conditions of committing this transaction is that each participating member also suspects the members being removed (l. 113). This result is a key component in ensuring that the probability of violating the NER failure condition is insignificant.

Proof of HM-Termination: There are three types of events that trigger messages:

1. *Join requests:* A join request has three phases. First in the application phase, application messages are sent to all core members, and each of these core members may forward the application to a subset of non-core members. Next in the preparation phase, each member receiving an application message broadcasts introduction messages to each member and current applicants, who in turn introduce themselves to the new applicant. The leader and applicant then exchange messages relating to the join conditions until the applicant joins the group. The number of application and introduction messages is finite and determined by the number of members and other applicants at the time of application, while the join condition messages continue until either the applicant joins the group or crashes. Finally, in the join phase, messages relating to the commitment of the join update are sent until either the applicant joins the group or crashes and is suspected by the leader. Via HM-Non-Triviality a correct applicant will eventually join the group, and an incorrect process will eventually crash. In either case the messages related to this join event subside.

2. *Leave requests:* A leave request begins with sending leave messages to all current members. Once these messages arrive, messages relating to the commitment of the leave update are sent until the member is removed. From HM-Non-Triviality-Leave the member is eventually removed from the group.

3. *Suspicions:* A suspicion is equivalent to a remove update, and initiates messages relating to the commitment of the remove update. If the suspicion is correct then from HM-Non-Triviality-Remove the member will eventually be removed, if the suspicion is false then eventually the suspicion will be corrected and the remove update rescinded. In either case the messages related to this suspicion event subside.

Proof of HM-Agreement: Group add and remove transactions occur only through OAC(*remove*, ...) and OAC(*membership*, ...) transactions. Since all members of the group, excluding those that are removed by the remove transactions, are participants in each transaction, if OAC(*remove*, ...) or OAC(*membership*, ...) is committed by any member it must be committed by all members.

Algorithm 4 Agent Allocation (HAA-AA)

1 *hostGroup* : list /* List of host group member, maintained by the membership service */
2 *agentList* : list /* List of agents, maintained by hosts */

3 $sesId_p$: integer /* Id of the current session */
4 $sesGroup_p$: list /* List of hosts involved in the session */
5 $sesAgents_p$: list /* List of agents, a_i, in the session */
6 $sesCosts_p$: list /* List of agent processing costs, c_i */
7 $sesAffinity_p$: list /* List of agent affinities, A_i */

8 $bundle_p$: list /* List of agents in host *p*'s bundle */
9 $bidTable_p$: table /* Table of accepted bids indexed by host and agent */
10 $outbox_p$: list /* List of bid updates to distribute */
11 r_p : integer /* Current round number */
12 $lastBuildRound_p$: integer /* Round number at the time of last build */
13 $buildQueued_p$: boolean /* Flags whether a build is required once the current updates are processed */
14 $distributeQueued_p$: boolean /* Flags whether a distribute is required once the updates are processed */
15 $decided_p$: boolean /* Flags whether the session has finished */
16 $sesReady_p$: boolean /* Flags whether bundle building can start for the session */

17 **upon** [*hostGroup* membership change **or** *agentList* change]:
18 **if** (*p* = front(*hostGroup*)) **then** /* undisputed leader */
19 $sesId_p$:= $sesId_p$ + 1; /* id of the current session */
20 $sesGroup_p$:= *hostGroup*; /* current host group */
21 $sesAgents_p$:= *agentList*; /* current agents */
22 $sesCosts_p$:= agent costs; /* agent processing costs */
23 $sesAffinity_p$:= agent affinities; /* agent affinities */
24 newSession(); /* prepare new session */
25 **do**
26 OAC(*start*, *p*, $sesId_p$, $sesGroup_p$, $sesAgents_p$, $sesCosts_p$, $sesAffinity_p$) to $sesGroup_p$;

```
27          wait until OAC(start, p, sesId_p, sesGroup_p,
    sesAgents_p, sesCosts_p, sesAffinity_p) is decided;
28        while [OAC(start, p, sesId_p, sesGroup_p, sesAgents_p,
    sesCosts_p, sesAffinity_p) is not committed and sesGroup_p =
    hostGroup and sesAgents_p = agentList] /* give up if the
    situation changes */

29  procedure newSession():
30    bidTable_p := EMPTY;
31    outbox_p := EMPTY;
32    buildQueued_p := FALSE;
33    distributeQueued_p := FALSE;
34    decided_p := FALSE;
35    sesReady_p := FALSE;
36    r_p := 0;
37    lastBuildRound_p := -1;

38  upon committing OAC(start, q, sesId_q, sesGroup_q,
    sesAgents_q, sesCosts_q, sesAffinity_q):
39    if (q != front(hostGroup) or sesGroup_q != hostGroup or
    sesAgents_q!= agentList )
40        return; /* expired session */
41    if (sesId_q >= sesId_p ) then
42        if (sesId_q > sesId_p ) then
43            sesId_p := sesId_q;
44            newSession(); /* prepare new session */
45        sesGroup_p := hostGroup;
46        sesAgents_p := agentList;
47        sesCosts_p := sesCosts_q;
48        sesAffinity_p := sesAffinity_q;
49        sesReady_p:= TRUE;
50        buildBundle();

51  procedure buildBundle():
52    buildQueued_p:= FALSE;
53    if (r_p = lastBuildRound_p ) then
54        r_p := r_p + 1; /* increment round */
55    lastBuildRound_p := r_p;
56    build bundle_p according to chosen strategy;
57    distribute();
58    checkConsensus();

59  upon receiving (update, q, sesId_q, for each updated bid j:
    {j, z_qj, y_qj, r_qj}):
60    if (sesId_q < sesId_p) then /* old message, ignore */
61        return;
62    else if (sesId_q > sesId_p) then /* higher session id */
63        sesId_p := sesId_q; /* join session */
64        newSession(); /* prepare new session */
65    for each updated bid j:
66        bidTable_p [q][j] = {j, z_qj, y_qj, r_qj}
67        if (r_qj ≥ r_p) then
68            r_p := r_qj + 1; /* keep round up to date */
69        oldBid := bidTable_p [p][j];
70        resolve conflicts via Table 4.7-1 (bidTable_p [q][j] vs.
    bidTable_p [p][j] );
71        if ( oldBid.y > bidTable_p [p][j].y ) then /* got worse */
72            buildQueued_p := TRUE;
73        if (sesReady_p = TRUE and buildQueued_p ≠ FALSE and
    outbox_p ≠ EMPTY ) then
74            distributeQueued_p := TRUE;
75        if (sesReady_p = TRUE ) then
76            checkConsensus();
77    wait until [all messages currently in the inbox are
    processed or sesGroup_p ≠ hostGroup or sesAgents_p ≠
    agentList];
78    if (sesReady_p = TRUE and buildQueued_p = TRUE ) then
79        buildBundle();
80    if (distributeQueued_p = TRUE ) then
81        distribute();

82  procedure distribute():
83    distributeQueued_p := FALSE;
84    send (update, p, sesId_p, outbox_p) to hostGroup;
85    outbox_p := EMPTY;

86  procedure checkConsensus():
87    if ( p = front(sesGroup_p) and decided_p ≠ TRUE and (for
    all j ∈ agentList: bidTable_p [p][j].z ≠ none) and (for all h ∈
    sesGroup_p: bidTable_p [h] = bidTable_p [p])) then
88        decided_p := TRUE;
89        do
90            OAC(finish, p, sesId_p, bidTable_p [p]) to sesGroup_p;
91            wait until OAC(finish, p, sesId_p, bidTable_p [p]) is
    decided;
92        while [OAC(finish, p, sesId_p, bidTable_p [p]) is not
    committed and sesGroup_p = hostGroup and sesAgents_p =
    agentList] /* give up if the situation changes */

93  upon committing OAC(finish, q, sesId_q, agentAllocation):
94    accept agentAllocation;
95    if (sesId_p = sesId_q) then
96        decided_p := TRUE;
```

HAA-AA Bid Comparison Operator

Bids are composed of three parts: *agents*, *reward*, and *support*. *Agents* is simply a list of the agents involved in the bid, while *reward* and *support* are used when comparing two bids. Reward indicates how desirable the agent or cluster is, and is based on the cost of the agent plus bonuses from other agents in the bundle who share affinity. Cost is used as the base of the reward to encourage that high cost agents are bundled first, leaving lower cost agents to be distributed after. Support indicates how well the host can meet the costs of the agent/cluster, and is calculated as:

$$support = \frac{K_h - \sum_{i \in bundle_p} c_i - \sum_{j \in y.agents} c_j}{K_h} \qquad \text{(II-1)}$$

In this way, support starts at 1 and decreases as the cost of the bundle grows; support < 0 means that the host is over capacity.

Two competing bids, *A* and *B*, are then ranked according to the criteria depicted in Fig. II-1. If necessary, ties can be broken by taking the bidder with the lower ID.

HAA-AA Bundle Building Strategy

Bundles are greedily built using the following strategy:

1. Calculate rewards for all unbundled agents:
   $reward[i] := (c_i + \sum_{\forall \alpha_{ij} \in A_i \text{ S.T.} a_j \in bundle_p} \alpha_{ij})$;
   if $i \notin L_h$ **then** $reward[i] := reward[i] * (1\text{-}t_i)$;
2. Identify clusters of unbundled agents that have mutual affinity bonuses using a recursive algorithm.
3. Iteratively add to the bundle by identifying the winning bid with the highest return. A winning bid is defined as a bid that is strictly greater than the currently accepted bid ($bidTable_p\ [p][j]$) according to the bid comparison operator described above.
   a. Check all permutations of clusters to find the bid with highest return. The reward and support for a cluster bid are calculated by summing the individual rewards for each agent in the cluster plus all affinity bonuses from the cluster and summing the individual costs, respectively. For a successful cluster bid, the entire cluster must fit within the capacity of the host and the bid must beat the current bids for every agent in the cluster.
   b. Check all unbundled agents individually to find the bid with highest return.
   c. Select the highest bid, y_{high}. If no winning bid was found then the bundle building process is complete, otherwise:
      i. Add all agents associated with the bid to the bundle:
      for each *i* in $y_{high}.agents$: append *i* to $bundle_p$;
      ii. Update the accepted bids:
      for each *i* in $y_{high}.agents$: $bidTable_p\ [p][i] := \{i, p, \{y_{high}.reward, y_{high}.support\}, r_p\}$;
      iii. Prepare to distribute the updates:
      for each *i* in $y_{high}.agents$: $outbox_p\ [i] := \{i, p, \{y_{high}.reward, y_{high}.support\}, r_p\}$;
      iv. Repeat Step 3.

Algorithm 5 Agent Freeze (HAA-ATF)

```
1  procedure freezeAgent( a ): /* current host */
2    send (freeze, p) to a;
3    DDB.agents[a].queue1 := DDB.agents[a].queue2 :=
   EMPTY; /* clear message queues */
4    begin forwarding all messages addressed to a to
   DDB.agents[a].queue1;
5    do
6      OAC (freeze, a, p) to hostGroup;
7      wait until OAC (freeze, a, p) is decided;
8    while [OAC (freeze, a, p) has not been committed];
9    commitGroup := hostGroup; /* hostGroup at time of
   commit */
10   wait until [received (state) from a and for each h ∈
   commitGroup: received (freezeAck, a) from h or h is
   removed from hostGroup]
11     begin forwarding all messages addressed to a to
   DDB.agents[a].queue2;
12     DDB.agents[a].state := state; /* submit state */
13     do
14       OAC (release, a) to hostGroup; /* release agent */
15       wait until OAC (release, a) is decided;
16     while [OAC (release, a) has not been committed];

17 upon receiving (freeze, q): /*agent */
18   state := writeState(); /* pack state */
19   send (state) to q;
20   shutdown;

21 upon committing OAC (freeze, a, q): /* other hosts */
22   if ( p ≠ q ) then
23     begin forwarding all messages addressed to a to
   DDB.agents[a].queue2;
24   send (freezeAck, a) to q;
```

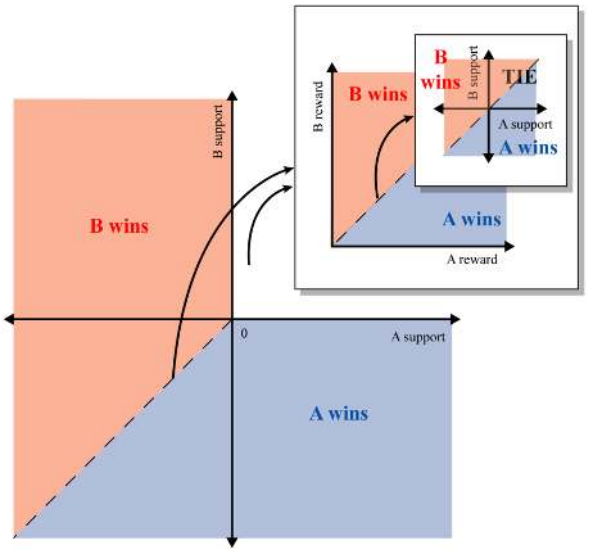


Fig. II-1 Bid Comparison Operator

Algorithm 6 Agent Thaw (HAA-ATT)

```
1  procedure thawAgent( a ): /* new host */
2    spawn shell for a; /* new agent thread */
3    begin forwarding all messages addressed to a to
     localQueue;
4    do
5      OAC (claim, a, p) to hostGroup; /* claim ownership */
6      wait until OAC (claim, a, p) is decided;
7    while [OAC (claim, a, p) has not been committed];
8    commitGroup := hostGroup; /* hostGroup at time of
     commit */
9    wait until [for each h ∈ commitGroup: received
     (claimAck, a) from h or h is removed from hostGroup) and
     all active OACs are decided]
10     send (thaw, DDB.agents[a].state,
       DDB.agents[a].queue1, DDB.agents[a].queue2,
       localQueue) to a;
11     begin forwarding all messages addressed to a to a;

12 upon committing OAC(claim, a, q): /* all hosts */
13   if ( a ∈ q's agent allocation and a is frozen ) then
14     if ( p ≠ q ) then
15       begin forwarding all messages addressed to a to q;
16     send (claimAck, a) to q;
17   else if ( p = q ) then
18     abandon claim attempt;

19 upon receiving (thaw, state, queue1, queue2, queue3 ): /*
   new agent */
20   readState( state ); /* unpack state */
21   process messages from queue1, queue2, and queue3 in
     that order;
22   resume normal agent behaviour;
```

Proofs for HAA-ATF and HAA-ATT

Proof of AT-Transparency is trivial from the fact that nothing in the algorithm impacts the behaviour of other agents. AT-Consistency can be proven as follows.

Lemma AT-1: No message addressed to *a* is lost.

Proof: All messages addressed to *a* (m.a.a.) are forwarded through the host network. Consider the freezing and thawing process in three stages: freezing, frozen, and thawing. Prior to the freezing stage all hosts forward m.a.a through H_{old}. During the freezing stage each host behaves as follows, with reference to Algorithm 5:

- H_{old} forwards m.a.a. to *a* until reaching l. 2, when it sends the freeze message to *a*. Messages up to this point are received and proceed by *a*, and thus integrated into a's state prior to receiving the freeze message. After l. 4, H_{old} forwards m.a.a. to the primary queue in the DDB until H_{old} reaches l. 11. At this point H_{old} is ensured that it will receive no more m.a.a. from other hosts, because all other hosts are now forwarding m.a.a. to the secondary queue in the DDB. H_{old} can then forward m.a.a. to the secondary queue in the DDB and end the freezing stage.
- All other hosts forward m.a.a. to H_{old} until they commit the freeze transaction (l. 21), at which point they forward m.a.a. to the secondary queue in the DDB. Note that a new host joining after the freeze transaction will already be aware that *a* is freezing/frozen and so will automatically forward m.a.a. to the secondary queue.

Upon reaching and throughout the frozen stage, all hosts forward m.a.a. to the secondary queue in the DDB. During the thawing stage each host behaves as follows, with reference to Algorithm 6:

- H_{new} begins the thawing stage by calling thawAgent(), at which point it begins forwarding m.a.a. to a local message queue (l.3). When H_{new} reaches l. 10, it is ensured that no more m.a.a. will be added to the secondary queue in the DDB and H_{new} can complete the thawing process. H_{new} forwards the primary, secondary, and local queues to *a*, and forwards any new m.a.a. to *a*.
- All other hosts continue forwarding m.a.a. to the secondary queue in the DDB until they commit the claim transaction and reach l. 15. At this point the hosts forward m.a.a. to H_{new}, who handles the messages appropriately. Note that a new host joining after the claim transaction was committed will automatically know to forward m.a.a. to H_{new}.

In this way, every m.a.a. reaches *a* prior to freezing, or is directed to the primary, secondary, or local queue and later processed by *a* upon thawing.

Lemma AT-2: Sender order is preserved for all messages sent to *a*.

Proof: Since all agents' messages are directed through their local host, it is sufficient to show that order is preserved for all messages sent by an individual host. Consider the behaviour of each host:

- All hosts $\notin \{H_{old}, H_{new}\}$ prior to committing the freeze transaction forward m.a.a. to H_{old}, after which they forward m.a.a. to the secondary queue in the DDB until committing the thaw transaction, at which point they forward m.a.a. to H_{new}. All m.a.a. received by H_{old} are processed in order and either forwarded to *a* if they arrives before the freeze message is sent to *a* or placed

in the primary queue. All m.a.a. received by H_{new} are processed in order and either forwarded to the local queue if they arrive before the thaw message is sent to *a* or are forwarded to *a*.

- All m.a.a. originating from H_{old} are forwarded to *a* until the freeze message is sent to *a*, forwarded to the primary queue until *a* is ready for release (Algorithm 5 l. 11), after which H_{old} behaves the same way as any other host.
- H_{new} behaves the same was as any other host until it initiates the thaw process. At this point all m.a.a. originating from H_{new} are forwarded to the local queue until the thaw message is sent to *a*, after which m.a.a. are forwarded to *a*.

Thus any m.a.a. has a possible destination of: *a*, DDB primary queue, DDB secondary queue, H_{new} local queue, or *a*, in that sequence, and so order is maintained.

Proof of AT-Consistency: From Lemmas AT-1 and AT-2, upon thawing *a* receives all messages sent to it, in correct sender order. Internal data is preserved when the agent freezes and thaws its state.

Algorithm 7 Agent Backup (HAA-AB)

```
1  procedure backupAgent():  /* agent */
2     b := backupState(); /* backup critical portion of state */
3     send (backup, a, b) to host; /* submit to local host */

4  upon receiving (backup, a, b): /* current host */
5     DDB.agents[a].backup := b;
```

Algorithm 8 Agent Recovery (HAA-AR)

```
1  procedure recoverAgent( a ): /* new host */
2     spawn shell for a; /* new agent thread */
3     wait until [all active OACs are decided]
4        send (recover, DDB.agents[a].backup) to a;
5        begin forwarding all messages addressed to a to a;
6        do
7           OAC (recovered, a, p) to hostGroup; /* claim */
8           wait until OAC (recovered, a, p) is decided;
9        while [OAC (recovered, a, p) has not been
   committed];

10 upon receiving (recover, b): /* new agent */
11    recoverState( b ); /* recover from backup */
12    resume normal agent behaviour;

13 upon committing OAC (recovered, a, q): /* other hosts */
14    if ( p ≠ q and a ∈ q's agent allocation ) then
15       begin forwarding all messages addressed to a to q;
```

Proof for HAA-AB and HAA-AR

Proof of AR-Recovery: The use of the latest backup is ensured by waiting for all OACs to be committed before using the backup (Algorithm 8 l. 3). *H* begins forwarding m.a.a to *a* as soon as the recover message is sent (Algorithm 8 l. 4), and if the recovery is valid all other hosts will begin forwarding m.a.a. to *H* (Algorithm 8 l. 15).

Appendix III
Agent Descriptions

AgentPathPlanner

Accepts a target position and orientation and attempts to plan a path through clear spaces using the available map data. The path uses a weighted distance algorithm that penalizes routes that are too close to obstacles.

Interactions:

Avatar* → Configure and assign targets
Avatar* ← Send action commands
Avatar*→ Send action updates
ExecutiveAvatar ← Request list of avatars

Recovery Strategy:

Restore configuration and wait for instructions.

AgentSensorCooccupancy

Uses avatar position information to generate map updates, following the logic that if the area is occupied by the avatar it is not occupied by an obstacle.

Interactions:

SupervisorSLAM → Assign processing requests
SupervisorSLAM ← Report success or failure of processing

Recovery Strategy:

Restore configuration and wait for instructions.

AgentSensorFloorFinder

Extracts the floor from an image by colour matching with an area sampled directly in front of the avatar, following the strategy in [112]. The floor region is then transformed from image space to map space using a planar homography transformation and used to generate a belief template for JC-SLAM.

Interactions:

SupervisorSLAM → Assign processing requests
SupervisorSLAM ← Report success or failure of processing

Recovery Strategy:

Restore configuration and wait for instructions.

AgentSensorLandmark

Scans images to locate landmarks building on the algorithm described in [113]. Landmarks are then used to generate localization updates.

Interactions:

SupervisorSLAM → Assign processing requests
SupervisorSLAM ← Report success or failure of processing

Recovery Strategy:

Restore configuration and wait for instructions.

AgentSensorSonar

Generates belief templates from sonar readings. The belief templates are used create map and observation density updates for JC-SLAM.

Interactions:

SupervisorSLAM → Assign processing requests
SupervisorSLAM ← Report success or failure of processing

Recovery Strategy:

Restore configuration and wait for instructions.

Avatar* (ER1, Pioneer, Simulation, SRV-1, X80H)

Builds on AvatarBase, which provides the basic functionality for all avatar agents. In particular AvatarBase provides the basic *action* interface exposed by all avatar agents. This interface allows other agents to control all avatars through commands such as wait, move, rotate, capture image, etc. without knowledge of an avatar specific API.

Interactions:

AgentPathPlanner ← Configure and assign targets
AgentPathPlanner → Send action commands
AgentPathPlanner ← Send action updates
Supervisor* → Assign targets
Supervisor* ← Report target status
ExecutiveAvatar ← Register avatar

Recovery Strategy:

Restore configuration and retrieve AgentPathPlanner status. Reassign target for AgentPathPlanner if appropriate.

Agent Status Monitoring:

AgentPathPlanner ← If failure occurs reassign target
ExecutiveAvatar ← If failure occurs re-register

ExecutiveAvatar

Controls the distribution of avatar resources using a bidding algorithm. Task supervisors place bids for resources and the ExecutiveAvatar assigns resources to the highest bidders. Resource assignments are stored in the DDB so that they are independent of agent failures.

Interactions:
Supervisor* → Place bids on avatar resources
Recovery Strategy:
Restore configuration.

ExecutiveMission

Defines mission parameters and controls when tasks are started. When a new task begins the ExecutiveMission spawns the appropriate task supervisor and supporting agents.
Interactions:
Supervisor* ← Spawn and configure agents
Recovery Strategy:
Restore configuration, confirm that appropriate Supervisor agents are active.

ExecutiveSimulation

Handles the simulation of avatars, sensors, and environment during HILS experiments.
Interactions:
AvatarSimulation → Send action commands
AvatarSimulation ← Send action updates

SupervisorCongregate

When the congregation point is identified SupervisorCongregate requests control of all avatars and assigns them targets at equally spaced intervals around the congregation point.
Interactions:
ExecutiveMission → Configure
ExecutiveAvatar ← Request control of avatars
Avatar* ← Assign target
Avatar* → Send target updates
Recovery Strategy:
Restore configuration and retrieve Avatar* statuses, reassign targets if appropriate.
Agent Status Monitoring:
Avatar* ← If failure occurs reassign target

SupervisorExplore

Carries out the exploration task by directing avatars to unexplored regions. Unexplored cells are divided between avatars using an Expectation-Maximization (EM) algorithm. Once the cells are divided each avatar is assign a target within their area.
Interactions:
ExecutiveMission → Configure
ExecutiveAvatar ← Request avatar resources
Avatar* ← Assign target
Avatar* → Send target updates
Recovery Strategy:
Restore configuration and retrieve Avatar* statuses, reassign targets if appropriate.
Agent Status Monitoring:
Avatar* ← If failure occurs reassign target

SupervisorForage

Each SupervisorForage agent is assigned a single collectible to pick up and deposit in a collection region. The SupervisorForage bids on suitable avatar resources until it gains control of an avatar.
Interactions:
ExecutiveMission → Configure
ExecutiveAvatar ← Request avatar resources
Avatar* ← Assign target and collection actions
Avatar* → Send action updates
Recovery Strategy:
Restore configuration and retrieve Avatar* status, reassign target if appropriate.
Agent Status Monitoring:
Avatar* ← If failure occurs reassign target

SupervisorSLAM

Monitors the DDB for new sensor readings and assigns them to sensor processing agents. The SLAM supervisor also maintains the active set of sensor processing agents and spawns new agents as required.
Interactions:
ExecutiveMission → Configure
AgentSensor* ← Spawn and configure agents, assign readings
AgentSensor* → Report processing success or failure
Recovery Strategy:
Restore configuration and retrieve AgentSensor* statuses, reassign readings if appropriate. Confirm that reading list is up to date.
Agent Status Monitoring:
AgentSensor* ← If failure occurs reassign reading