UNIVERSITY OF CALIFORNIA

Santa Barbara

MotionArc : Preliminary design of a system architecture for collaborative robotic missions

A Thesis submitted in partial satisfaction of the requirements

for the degree of Master of Science

in Mechanical Engineering

by

Nathan P. Owen

Committee in charge:

Dr. Francesco Bullo, chair

Dr. Bassam Bamieh

Dr. João Hespanha

June 2009

The thesis of Nathan P. Owen is approved.

João Hespanha

Bassam Bamieh

Francesco Bullo

June 2009

ABSTRACT

MotionArc : Preliminary design of a system architecture for collaborative robotic missions

by

Nathan P. Owen

This thesis presents a unified system architecture for implementing multiple distributed algorithms on a network of robotic agents for efficient completion of varied mission plans. We consider mission plans requiring simultaneous, asynchronous execution of several distibuted algorithms by variable subsets of the robotic network. The MotionArc system implements a combination of efficient task-acheiving algorithms designed for a given mission plan as *behavior sets* by providing solutions to two key architectural challenges: first, to ensure maintenance of sufficient, appropriate, and accurate situational awareness information onboard each node of the network, and secondly, to allocate an appropriate number of agents to each required behavior set such that some global cost objective is optimized. We approach the first problem by adapting the concept of a mission state estimate, and the second problem is solved using several techniques from combinatorial optimization. In practice, the MotionArc system provides a software library as a Python module for implementing these methods and a standard for porting theoretically designed algorithms into functional software components onboard simulated and real-time robotic hardware. As a primary demonstation tool, we will pesent a mission plan calling for the ex-

ecution of several algorithms for the dynamic vehicle routing problem where the global cost function is a measure of the *expected service delay* for each instance of a routing task.

Contents

Abstract **iii**

1 Introduction and Background **1**

1.1 A model for candidate missions 3

1.2 Previous Work 5

1.3 Conceptual Model 7

1.4 Preliminary System Design 12

2 Mission modeling **19**

2.1 Mission state estimation 19

2.2 Mission modeling data structure 24

2.3 Mission modeling protocols 26

2.4 User Interface 32

3 Tasks and Design of Behavior Sets **34**

3.1 Requirements and design principles 34

3.2 From algorithm to behavior set 36

4 The Behavior Allocation Problem **43**

4.1 Solving the continuous problem 43

4.2 Solving the Integer Variable Problem 45

4.3 Implementation of the allocation mechanism 46

5 MotionArc implementation and execution **55**

5.1 Implementing system components 56

5.2 The MotionArc Package 58

5.3 The MotionArc Application 59

5.4 User Interface 62

6 Mission Results and Final Considerations 64

6.1 Demonstrating example task-point selection policies 64

6.2 Preliminary results for a multi-task mission 70

6.3 Comparative analysis of task-allocation and mission modeling . 75

6.4 Concluding remarks 77

A Mission Log Results 80

List of Figures

1 System model for a basic distributed algorithm instance 9

2 Block diagram for the evolution of the Mission State Estimate . 10

3 General system model for MotionArc 11

4 Conceptual system architecture design 14

5 Conceptual model for mission modeler 24

6 Status transition model for task-point state estimates 28

7 Status transition protocol for agent state estimates 30

8 Behavior set evolution 35

9 Conceptual model for behavior manager 47

10 Convergence of the allocation algorithm for 10 and 100 agents . 50

11 Allocation of 50 agents over an extended mission 52

12 Allocation algorithm onvergence time over an extended mission 53

13 Allocation responds to addition of 10 agents 54

14 Series MotionArc application process execution 61

15 Two-agent DTRP with *mSQM* selection policy 65

16 Two-agent DTRP with *partition-TSP* selection policy 68
17 Two-agent DTRP with *G/G/m-TSP* selection policy 69
18 Agent fault detection and task-point reassignment 71
19 A hardware agent maintains a Voronoi partition 73
20 A hardware agent executes a service task 74

List of Tables

1 Task allocation for demonstration mission 72
2 Mission state estimate log for two agents completing a DTRP task using the *partition-TSP* selection policy 81
3 Mission state estimate log for two agents completing a DTRP task using the *mSQM* selection policy 83
4 Mission state estimate log for two agents completing a DTRP task using the *G/G/m-TSP* selection policy 84
5 Mission state estimate log for one read hardware agent and three simulated agents completing a mission consiting of an identification task and one class of service task, using the *GGm-TSP* and *partition-TSP* selection policies, respectively. 86

1 Introduction and Background

One of the primary research goals of the UCSB Motion Lab is to develop efficient distributed motion coordination algorithms for robotic networks. Recent efforts have developed algorithms attempting to solve a variety of geometric optimization problems for robotic networks such as senor placement and coverage, surveillance and partrol, and dynamic vehicle routing (DVR). Other research thrusts have focused on lower-level motion algorithms for obstacle avoidance, navigation, and communication connectivity maintenance. These algorithms have been analyzed extensively through theory and low-fidelity simulation and have well established time complexity and performance metrics. For the next phase of algorithm development, we would like to test these algorithms in real-time using high-fidelity simulation and demonstration hardware; ideally, we will prove the algorithms first individually on hardware and finally as part of a complex demonstration mission requiring the use of several algorithms for completion. Effectively transitioning from the process of implementing single algorithms through centralized numerical simulation to executing multiple algorithms on hardware in a complex mission plan requires a well thought out system architecture. This thesis will introduce our proposed architecture for experimental implementation of motion coordination algorithms, named MotionArc .

Current implementations of our algorithms have been designed towards ease of numerical simulation. When considering implementation on simulated or real hardware, several key adaptations must be made. Programming of algorithms to run on unique, individual hardware nodes requires software and hardware implementation of systm component methods such as message transmission and parsing, hardware acuation, and information management. Additionally, when

executing algorithms within the context of a complex robotic mission, some sort of user interface (UI) and situational awareness model must be introduced. Ideally, our system architecture will utilize a single framework for implementing multiple classes of algorithms, providing common mission management routines for actions required by all missions to effectively manage the flow of information and actuation commands amongst all components. Additionally, we will provide a library of common tools for use in algorithm development, and a standard to which developmental candidate algorithms should adhere to ensure ease of embedding within the mission management system. Ultimately, we intend on implementing algorithms to accomplish missions requiring multiple agents executing multiple behaviors; as such, an ideal architecture will also focus on optimizing combined mission performance through individual routines and actions on each agent.

This thesis will introduce the MotionArc system through a look at each of the main steps in software development. First, we will discuss a general model for the candidate demonstration missions for which we intend on deploying our algorithms. This model drives the conceptual design of our architecture - we wish that it be general enough to allow use of the architecture with a variety of classes of component algorithms; however, the sheer number of candidate problems will require us to focus primarily on a demonstration mission utilizing a few select types of algorithms. Next, we will introduce the preliminary design of the system architecture, on the big-picture mission modeling level and the level of individual components. We will then discuss the hardware and software implementation of system components, and finally introduce a demonstration mission and results.

1.1 A model for candidate missions

We consider a connected robotic network executing a particular mission plan consisting of several global "tasks" requiring action by one or more of the robotic agents in the network. Tasks can be classified as one of three types: single-robot tasks that require one and only one agent for completion such that assignment of additional agents does not improve performance; n-robot tasks that require exactly $n > 1$ agents for completion; and arbitrary-robot tasks where performance is improved by assignment of additional agents to the task. We also consider that each task comprising the mission has an associated cost function that is dependent upon the number of agents assigned to that particular task. In general, we will consider the performance cost for a particular task to be the average expected delay between instantiation of a single task-point and its completion. We can then consider a global mision cost to be some additive combination of the delay functions for each component task. Before continuing, let us provide some definitions for the key components of the mission plan:

Agent: A unique entity, usually a robotic device, that is used to complete the mission plan and is under the management of our system architecture.

Task: Some unique action or sequence of actions required to be completed by an agent or team of agents for accomplishing the mission plan.

Behavior Set: The unique algorithm, protocol, or method executed by an agent or team of agents to accomplish a specific task, also called a high-level task acheiving function.

Task-point: A particular, unique instance of a task; usually associated with some specific task parameters such as location or time of service.

Simply put, we can think of each agent or team of agents executing a particular *behavior* to accomplish some *task* on a particular *task-point*.

Our demonstrative mission is in essence a series of multiple occurances of Bertsimas and van Ryzin's dynamic traveling repariman problem (DTRP) [4]. We consider a sequence of task-points, uniformly independently and identically (iid) distributed throughout a known two-dimensional environment, each of which arrives as part of a poisson process with a known arrival rate. Each task-point requires one or more of several services to be performed by one or multiple robotic agents; the class of service and the presence of the task-point are initially unknown. As a demonstrative goal for the project, we wish that our system accomplish this mission plan while approximately minimizing the average delay between task-point instantiation and the completion of all its required services. For completion of the mission plan, we require that our system leverage distributed algorithms to accomplish two types of tasks:

- **Identify**: classify the service (or group of services) required by a task-point
- **Service**: perform the service (or group of services) required by a task-point on that task-point

A mission could easily incorporate multiple classes of service tasks; in section 3 we present several behavior sets designed to accomplish service tasks that differ by several key parameters. Additional potential service task classes could be:

- Team service, where multiple agents are required to collaborate to complete a particular task-point
- Pickup and delivery, where a task-point specifies a location to which it

must be moved

- Service of needy task-points, where the task-point requires repetitive service

Our mission model requires two primary assumptions about the nature of the abilities of individual agents. We assume that agents, at any given time, are only "outfitted" with one particular behavior set, capable of accomplishing one particular task. However, we allow agents the ability of having that behavior re-assigned at some performance cost. This allows us to reduce the problem of assigning behaviors to agents to a periodic, combinatorial optimization problem. Thus, one of the key components of our system architecture will be a method of finding the optimal initial allocation of behaviors to agents.

1.2 Previous Work

Much of the motivation for the structural design of our architecture comes from Arkin's overview of the general components of behavior-based robotic architectures in [1]. Additional motivation and general applications are drawn from the context of architectural design for situational awareness [2], and unmanned vehicle operation. Insight into hardware abstraction components of architecture design are presented by Kumar et. al. in [3]. A characterization of current task allocation methods based on problem taxonomy is given in [13]. Recently, there have been significant research efforts into combinatorial methods for task allocation such as those presented in [9]. We will show that our behavior allocation problem is related to the general integer resource allocation problem, treated with a distributed gradient descent method in [6] and with several useful heuristics in [20].

Additionally, several state-of-the-art approaches have been presented to dynamic mission planning and task allocation that form a more complete system architecture. The key differences in these approaches hinge on the characterization of mission elements or tasks and task-points, the treatment of robot controls as either adaptive behaviors unique to each task or internal methods specified within the parameters of the task model, and the presence of a central mediating agent or "auctioneer". One example, the ALLIANCE architecture [16], uses a behavioral approach to classify the ability of agents to choose tasks from those for which there is an appropriate available behavior as well as to take over or relinquish tasks to and from other agents. Each agent has a certian "motivation" or affinity for a particular task that evolves dynamically based on communication and sensor information, but assigns no distinct state to individual tasks and does not explicitly require completion of tasks by any particular agent. It also does not describe or induce an explicit matching of tasks to agents, it instead relies on the ability of agents to map behaviors to tasks, considering each available behavior to be an independent, high level task acheiving function. As another example, the Contract Net approach [10] and similar protocols rely on the use of auctions to establish an agent-task assignment. In such protocols, "winning" agents are commited to the completion of tasks, and the assignment may be reviewed dynamically based on some metric of task progress or completeness. This type of approach may typically require some store of central knowledge or a central "auctioneer" and may be susceptible to scalability issues.

Most relevent to our system goals is the concpet of task allocation through mission state estimates presented by Hedrick et. al in [14], which presents a "distributed task allocation technique based on opportunistic exchange of information". As much of our architecture will be concerned with maintenance

of situational awareness estimates amongst a network of robtic agents, the protocols for effiecent information transmission presented in [14] are quite useful. However, the mission example provided results in a rather sub-optimal, "greedy" approach to actual allocation of taskpoint instances to agents. Fortunately, the very algorithms we wish to demonstrate provide unique methods for finding a closer to optimal allocation; thus, the combination of our algorithms and the mission state estimate methods for situational awareness maintenance result in a powerful task-point allocation method.

Many of the algorithms developed in our lab are built upon the framework presented by Bullo, Martinez, and Cortes in [8], which provides an excellent model for the assumptions we will make on the nature of the distributed algorithms we wish for our architecture to manage. For the particular application to dynamic vehicle routing problems, we consider algorithms for multiple classes of demands [17], moving demands [5], and general routing [18].

1.3 Conceptual Model

To introduce the top-level conceptual model for MotionArc , we consider the concept of a *communication and control (C^2) law* for a robotic network. In general, an instance of a distributed algorithm running on a single agent of a robotic network has several properties. We consider agents with hardware abstraction, as described previously, where the onboard decision-making process is decoupled from the actual plant dynamics. As a preliminary, let $X(t)$ refer to the value of X at time step t and $X_{[k]}$ refer to the value of X held by agent k. In each iteration of its communication and control law an agent must

1. Parse and handle incoming messages from its communication neighbors

2. Parse and handle sensory data from its hardware interface

3. Generate a control signal for its hardware interface

4. Generate an outgoing message to be broadcast to its communication neighbors

The methods for mapping the incoming messages and sensory data to outgoing messages and control signals are the details of the distributed algorithm instance operating on that agent. We define those methods using a framework from sensing and information theory, as described in [15] and [8]. We define $\Omega_{[k]}(t)$, the processor state or internal state of a distributed algorithm instance (or behavior set) on agent k. We then define the signals described above as $\xi_{[k]}(t)$, the sensor readings of agent k at time t, $Y_{[k]}(t)$, the message broadcast by agent k at time t, and $u_{[k]}(t)$, the hardware control signal generated by agent k at time t. We define the message received by agent k as $\sum_{i \in C(k)} Y_{[i]}$, where $C(k)$ defines the communication network of agent k.

The mechanics of the distributed algorithm instance are carried out by two mappings. F, or `filter`, the *information filter* or *state transition function* generates a new internal or processor state based on the current state and all new information, thus building the agent's information history. A, or `control`, the *actuation and message generation function*, generates the control and communication output based on the current information state. Formally, the two mapping functions are:

$$\Omega_{[k]}(t+1) = F\left(\Omega_{[k]}(t), \sum_{i \in C(k)} Y_{[i]}(t), \xi_{[k]}(t)\right) \tag{1}$$

$$Y_{[k]}(t), u_{[k]}(t) = A\left(\Omega_{[k]}(t)\right) \tag{2}$$

The system evolution is shown as a block diagram in figure 1. In section 3 we

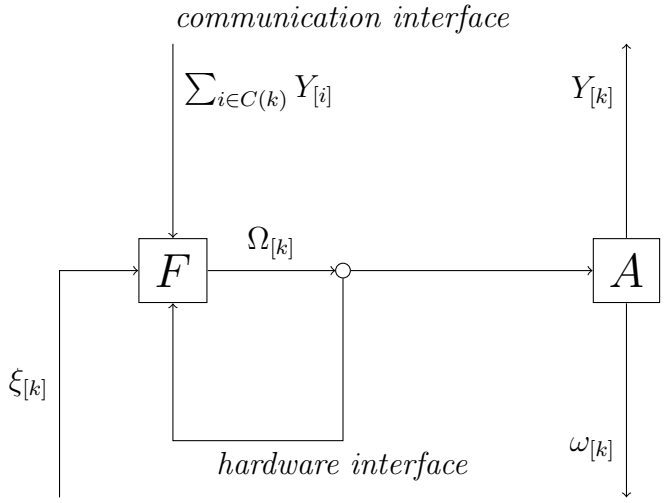


Figure 1: System model for a basic distributed algorithm instance

will present a standard for designing algorithms to fit within this framework and that of the MotionArc system, along with examples from our demonstration DVR-type mission.

To meet the system requirement of maintaining sufficient and accurate situational awareness information, we use the concept of a Mission State Estimate that evolves in a system that can be characterized as a general communication and control algorithm as described above, neglecting the hardware interface. Implementation of this mechanism on each member of a candidate robotic network, *regardless of what behavior set or distributed algorithm is active on that member*, ensures that such information is propagated throughout the network. We define $\hat{M}_{[k]}(t)$, the mission state estimate of agent k at time t, to contain an efficiently designed set of *task-point state estimates* and *agent state estimates* that contains all the information required to be shared among agents to execute any of the available behavior set algorithms. The evolution of $\hat{M}_{[k]}$ is governed

by specific filter and acutations functions $F_{[k]}$ and $A_{[k]}$, respectively:

$$\hat{M}_{[k]}(t+1) = F_{\hat{M}}\left(\hat{M}_{[k]}(t), \sum_{i \in C(k)} Y_{[i]}(t)\right) \tag{3}$$

$$Y_{[k]}(t) = A_{\hat{M}}\left(\hat{M}_{[k]}(t)\right) \tag{4}$$

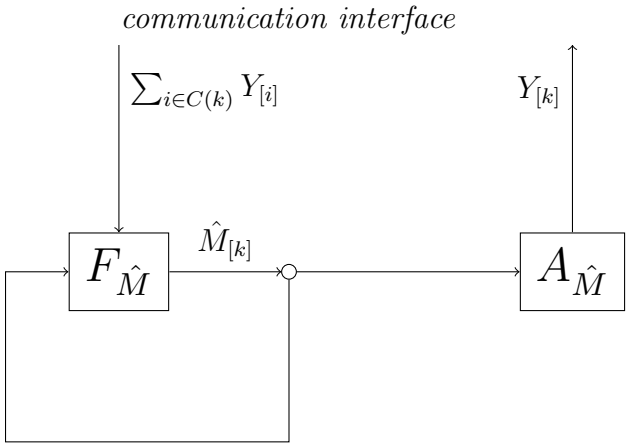


Figure 2: Block diagram for the evolution of the Mission State Estimate

The system block diagram for this mission modeling mechanism is shown in figure 2. The exact protocols that make up the filter and actuation functions for maintaining the Mission State Estimate, along with the details on the structure of the data held within the Mission State Estimate, are given in section 2.

The execution of instances of distributed algorithms as behavior sets within MotionArc is represented by a coupling of the formulation for a *particular* behavior set with the mission modeling formulation shown in figure 3. In other words, an identical mission modeling algorithm is evolving asynchronously on every member of the robotic network; for each subset of the robotic network allocated to a particular task, the respective task-acheiving algorithm or behavior set is coupled to the modelig algorithm on each agent within that subset.

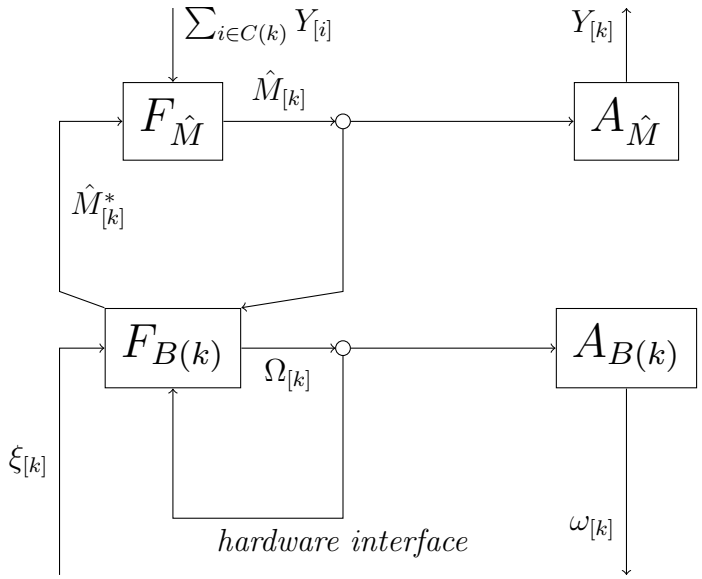


Figure 3: General system model for MotionArc

The mission modeling component takes over the function of parsing and broadcasting messages throughout the network and provides its internal mission state estimate to the active behavior set; that behavior set then parses the current mission state estimate along with its received sensor data to generate the next iteration of its own internal state and its control signal. The active behavior set also updates the mission state estimate and completes the feedback loop by returning the estimate to the modeling mechanism. Represented as a system of coupled filter and actuation functions, we have the general model for the

communication and control law for an agent k performing behavior set $B(k)$:

$$\hat{M}_{[k]}(t) = F_{\hat{M}}\left(\hat{M}^*_{[k]}(t), \sum_{i \in C(k)} Y_{[i]}(t)\right) \tag{5}$$

$$\Omega_{[k]}(t+1), \hat{M}^*_{[k]}(t) = F_{B(k)}\left(\Omega_{[k]}(t), \hat{M}_{[k]}(t), \xi_{[k]}(t)\right) \tag{6}$$

$$u_{[k]}(t) = A_{B(k)}\left(\Omega_{[k]}(t)\right) \tag{7}$$

$$Y_{[k]}(t) = A_{\hat{M}}\left(\hat{M}_{[k]}(t)\right) \tag{8}$$

1.4 Preliminary System Design

The primary conceptual model introduced above governs the interaction and execution of the mission modeling and behavior set protocols on a single agent. We now turn our attention to a preliminary system design for our architecture that takes into account the notion of task allocation across an entire robotic network. To do this, we require a dual notion of the concept of task allocation. On the one hand, we consider that there will be a subset of task-points corresponding to each unique task; within that subset each individual task-point must be assigned or selected by some agent. On the other hand, we wish that the set of behaviors required to complete the mission encompassing all task-points be distributed efficiently amongst the agents. One approach to task allocation is to break all possible tasks into unique subtasks (or task-points, in our case) and simply assign each subtask to an agent - assuming all agents are able to execute the behavior required to complete each subtask - as used in [14]. Our system instead leans closer to the ALLIANCE approach [16] where agents are assigned behaviors and the assignment of appropriate task-points within the set of agents performing the associated behavior is left as a detail of the behavioral algorithm.

However, instead of allowing agents to chose tasks based on some internal motivation function as in ALLIANCE, we will use a combinatorial approach similar to the nonlinear resource allocation problem described in [6].

We break the problem of finding an optimal allocation of agents to tasks into three component problems. First, we require that agents have at their disposal behavior sets – or collections of high-level task-acheiving algorithms – that efficiently allocate one class of task-point instances amongst the team of robotic agents assigned to a single task. Secondly, we require that there is an efficient exchange of information throughout the robotic network such that, in the infinite time horizion, each agent has sufficient knowledge of the global mission state – including information about the robotic networks, the component tasks, and individual task-point instances – to execute the currently assigned afore-mentioned behavior set. Assuming these two requirements are met (sufficient situational awareness and availablilty of efficient task-acheiving algorithm sets), the global task allocation problem is reduced to a dynamic combinatorial obimization problem of finding the assignment of agents to tasks that minimizes some global cost function.

We propose our system architecture, named MotionArc , that addresses each of these three mission planning components for a robotic network. To build the architecture, we draw on existing mechanisms from task allocation, distributed algorithms on robotic networks, and combinatorial optimization. For situational awarness and mission information exchange, we build upon the concept of Mission State Estimates and the mechanisms introduced in [14]. For efficient execution of tasks and allocation of task-point instances we will propose and algorithm model introduced in [8] and will provide specific examples given by algorithms developed in our laboratory. For the final, global combi-

natorial optimization problem, we will show that our problem is an instance of the integer capacitated resource allocation problem [6] and will explore several heuristic methods for optimization.

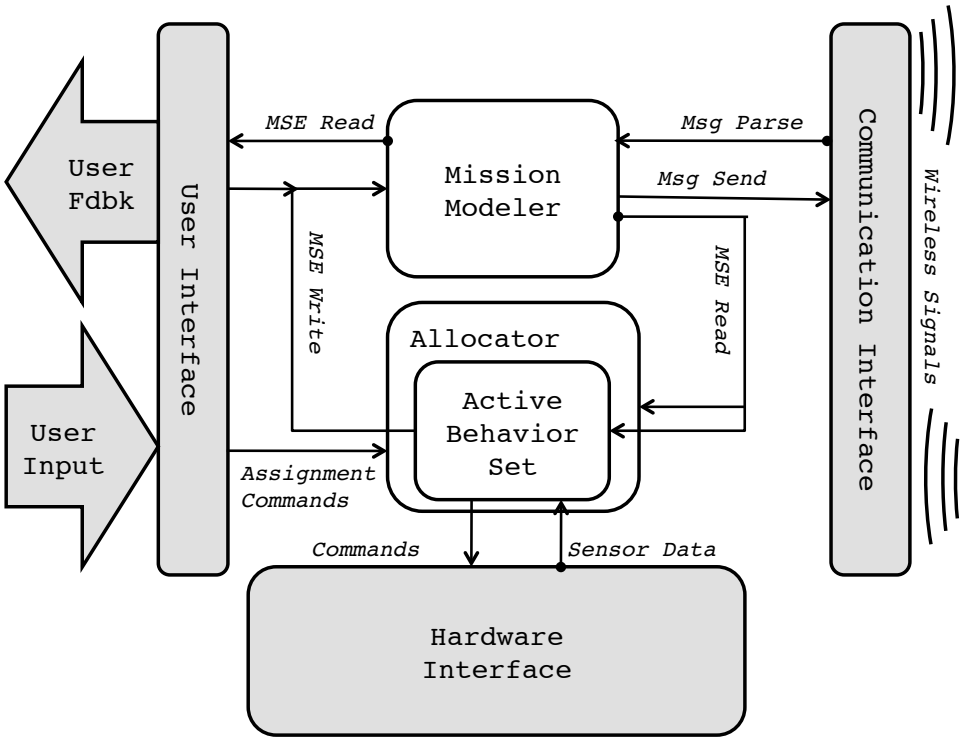


Figure 4: Conceptual system architecture design

The MotionArc system includes the three primary components described above and, onboard a robotic agent, incorporates them with fundamental components of any robotic architecture. One of the goals of our architecture is to require as little knowledge as possible of the mechanics of the system architecture from behavior set developers. For example, the architecture is designed such that behavior sets are simply provided a situational awareness model; the details of the maintenance of such a model are handled separately by the architecture and are not relevant to the design of behavior sets. With that concept in mind, we present the system components here, in order of increasing level of

abstraction from the hardware. Figure 4 gives a top level organizational flow chart showing the interaction between these componenets.

Hardware : The main hardware component of our system is the network of mobile robotic agents. Specifically, current hardware consists of three Videre Erratic mobile robot bases equiped with Hokoyo URG laser range scanners. Through hardware abstraction, knowledge of hardware details is not required of behavior sets.

Low level controls and hardware interfacing : Candidate behavioral algorithms require some level of abstraction between hardware acutation commands and their behavioral output commands. In our system, we incorporate the Player/Stage software package [12]. We use player drivers to implement low-level control algorithms such as the Vector Field Histogram Plus (VFH+) local navigation method [19], Smooth Nearness Diagram navigation [11], global path planning, and localization. The system architecture is developed with the assumption that, onboard each agent, it needs only to give access to that particular player server to component behavior sets; behavior sets must be designed to properly use that server for hardware interfacing. We will discuss hardware interfacing in more detail in the implementation section.

Communication : Many distributed algorithms require the ability to pass messages between agents; additionally, our assumption that each behavioral algorithm has sufficient situational awareness information requires the ability for such information to be shared among the agents of the network as well. Our system uses TCP/IP for communciation over a wireless network. Additionally, our communication system will supplement

mission situational awarenes with a communciation connectivity model. Again, our architecture will abstract away the details of sending messages; behavior sets need only to specify the message content and target. Communication will be discused in more detail in the implementation section.

Library of behavior sets : Each behavior set at an agent's disposal is stored in a library. As the primary task-acheving functions of the system, these behavior sets are the key components around which our architecture is designed. A behavior set should be designed to obtain sensor readings and send actuator commands using the protocols provided by the hardware interfacing component; similarly, communication requests and commands should use the protocols of the communication component. Behavior sets will also have access to the most current situational awareness model through the mission modeling and planning component. For the purposes of our architecture, we assuming behavior sets are designed following the framework laid out in [7] and section 3

Behavior set management (allocation) : As previously discussed, our system will attempt to optimize the allocation of agents to required behavior sets through combinatorial approaches. This component will carry out the details of this allocation. As combinatorial approaches will require the delay-cost functions associated with each particular behavior set, the behavior set manager will maintain estimates of these functions. Onboard each agent, these managers will handle re-allocation requests and commands and execute the details of activating and deactivating individual behavior sets.

Mission modeling and planning : The handling of situational awareness

maintenance is the key feature of the MotionArc system. Each agent must maintain some belief about the global state of the mission; to do this efficiently, we borrow from the concept of mission state estimates used in the allocation method presented by Hedrick et. al [14].

User interface and User : One of the primary goals of our system architecture is to provide a method of situational awareness feedback to the end user – both in the case where the user is an active participant in the mission ("in-the-loop") and where the user is just monitoring the situation from a more centralized position ("on-the-loop"). For our initial experiments, we will be considering the on-the-loop situation, since our goal is simply to monitor the execution of a mission plan on our robotic system. The user interface accesses the information provided by the mission modeler, and presents it in some format ideal to the particualr user.. Ideally, the system would have both a graphical user interface (GUI) and text based command-line interface (CLI). We also provide the user some control over the system level of automation.

As an example, consider a potential tour through the architectural components provided by a look into a single logic pass onboard an agent:

1. Agent receives situational awareness update information, either through communication or user interface components

2. Agent parses update and adjusts situational awareness model to reflect most current state of information

3. Currently active behavior set accesses sensor data through hardware interface, situational awareness data through mission modeler, and new mes-

sages through communication interface.

4. Behavior set makes calculations and sends actuation commands through hardware interface, message to other agents through communication interface, and updates situational awareness information through mission modeler.

5. Mission modeler parses update and generates new situational awareness model, which it sends to other agents through the communication and to the user through the user interface.

6. Behavior manager updates delay-cost function for active behavior based on situational awareness model from mission modeler.

In an actual mission implementation, these processes might occur simultaneously or in a different order.

The MotionArc system architecture effectively integrates and leverages the capabilities of each component to provide each agent with sufficient situational awareness information and efficient tools to interact with its environment to meet our specified mission goals. Additionally, the architecture provides the mission management capabilities over the entire robotic network required by our mission model. We now turn our attention to the three main system processes from a conceptual standpoint. First, we will introduce the mission modeling data framework and the processes used to guarantee availability of situational awareness information. Next, we will discuss the mechanics of the behavior set model that will put our distrubited algorithms to use on our robotic network. Finally, we will provide a descriptoin of our selected method for optimizing system performance through allocation of agents to behaviors.

2 Mission modeling

As discussed previously, the mission modeling component of the MotionArc system handles all duties related to maintaining adequeate knowledge of mission situational awareness for effective execution of behavior sets managed by the system. An estimate of the informational state of the mission is broken into information subsets – primarily regarding individual task points and agents – which are periodically shared throughout the network and updated by local behavior sets as each agent executes its own instance of a distributed algorithm. The basic framework of protocols ensuring accuracy and relevance of mission information is built upon a system of mission state estimation.

2.1 Mission state estimation

Consider the concept of 'state' for a specific mission plan. As introduced in our general mission model, a mission plan can be described by a set of tasks, each comprised of a set of task-points that represent a single instance requiring the service of a particular task. The state of a mission task-point can be thought to consit of, first, a copy of the task-point parameters provided by the mission plan and second, dynamic information about the task-point. This dynamic information could include the agent to which it is assigned, if any; whether it is currently completed, in process, or needs to be assigned; and the last time at which the information was updated. Now consider a mission involving several robotic agents with limited communication. Due to noise and uncertainty in communication, there is no global knowledge of the state of mission tasks and subtasks; rather, each agent can maintain an *estimate* of said states – a mission state estimate.

The mechanism developed in [14] describes a series of protocols by which - in the presence of communication uncertainty - an efficient allocation of task-points is achieved. An agent with a certain mission state estimate regarding the mission plan steps through this series of protocols:

1. `msg:` Defines a message sent to all agents within communication network
2. `sync:` Intelligently merges agent's own mission state estimate with those received from other agents to generate an up-to-date estimate
3. `transition:` Transitions the state estimates for each subtask through a series of transition rules provided by the mission plan. For example, a subtask with status `'assigned'` could be transitioned to status `'done'` if some qualifying criteria has been met.
4. `select:` Selects an appropriate subtask based on a cost function. Cost modifiers induce affinity for currently assigned subtask and aversion to subtask assigned to other agents.
5. `execute:` Agent generates control output based on internal state and parameters of assigned subtask.

Each of these methods are described in detail in [14]. Assuming that the message transmission protocol is suitable for integration with our communication network and that all protocols up to `select` lead to an efficient assignment, only the final two protocols are of consequence when considering the interaction of this mechanism with our library of distributed algorithms.

It is helpful to think of the mission state estimate mechanism itself as a C^2 law adhering to the standard described previously. Consider an agent k, and, at

discrete time l, let the agent have mission state estimate $\hat{M}^k(l)$ and state $X^k(l)$

$$X^k(l) := \left\{ \begin{array}{c} \texttt{AgentID} := k \\ l := \mathsf{time} \\ \texttt{dynamic\_state}(l) \\ \texttt{vehicle\_type} \\ \texttt{resources\_available}(l) \\ \hat{M}^k(l) \end{array} \right\}$$

The message sent by each agent k at time l is simply

$$Y^k_{\text{out}}(l) = \text{msg}(X^k(l)) = \begin{pmatrix} k \\ \hat{M}^k(l) \end{pmatrix}$$

Let $u^k(l)$ be the control output of agent k at time l and let $Y^k_{\text{in}}(l)$ be the collection of messages received by agent k at time l from all agents within communication range:

$$Y^k_{\text{in}}(l) = \{\Big(j, \hat{M}^j(t(j))\Big) \,|j \neq k, (k,j) \in E_{\text{cmm}}\}$$

The task allocation protocol can then be broken down into a state transition function and a control function. The state transition function updates the mission state estimate based on the current estimate and the group of received estimates:

$$\hat{M}^k(l+1) = \text{stf}(\hat{M}^k(l), Y^k_{\text{in}}(l))$$

The state transition function can be broken into components corresponding to

the `sync` protocol and the `transition` protocol:

$$\begin{aligned} \hat{M}_*^k &= sync(\hat{M}^k(l), \mathrm{Y}_{\mathrm{in}}^k(l)) \\ \hat{M}^k(l+1) &= transition(\hat{M}_*^k(l)) \end{aligned}$$

The control function involves the selection and execution of an appropriate subtask from the current state estimate:

$$u^k(l) = ctrl(\hat{M}^k(l))$$

The control function also can be broken down into components consisting of the `select` protocol and the `execute` protocol:

$$\begin{aligned} \hat{S}_*^k &= select(\hat{M}^k(l)) \\ u^k(l) &= execute(\hat{S}_*^k) \end{aligned}$$

In [14], there is an additional layer to the mission state estimate; individual task-points are grouped into corresponding classes of tasks - in other words, all "detect" task-point instances would belong to the unique task "detect". Tasks have associated state estimates as well as unique transition rules. Agents would transition from performing one task to another when specified by the appropriate rules. This is one component of the mission state estimate framework that our architecture will not incorporate; instead, agents will only consider the subset of task-points belonging to the task corresponding to their active behavior set. Thus, transitions between tasks are handled by the behavior managmenet component described in section 3. Our data model *will* include global task in-

formation in the form of a *task specification* as described below; however, this information is used for task execution and performance cost estimation only and is not used to transition between tasks and behavior sets on individual agents.

Our second deviation comes when agents select a particular task-point instance to attempt to accomplish. As formulated here, when choosing amongst task-points, agents would simply follow a greedy approach, basically choosing the task point with the lowest cost. Instead, we leave this process to be handled by the currently active behavior set.

Removing these two concepts from the mission state estimate framework, we are left with a system through which we can effectively manage mission situational awareness information; action upon this information is the subject of other components of the architecture. So far, we have only discussed mission state estimates regarding individual task-points – for the purposes of our architecture, we will extend this concept to incorporate state estimates for individual agents as well, since many of our algorithms require some knowledge about the state (particularly the location) of other agents. As such, we will refer to both task-point state estimates (TSE's) and agent state estimates (ASE's). Our specific framework, shown in figure 5, can be discussed in terms of mission modeling data strucutres and protocols. In general, the mission modeler maintains a database containing TSE's and ASE's for all known task-points and agents involved in the mission. Periodically, messages are received from other agents containing a copy of their entire mission state estimate, which is then merged with the existing estimate through the `sync` protocol. Internally, mission state estimates are run through the `transition` protocol. These two protocols ensure the mission state estimate is accurate and current. The mission modeler also may maintain a data set containing information about the mission environment.

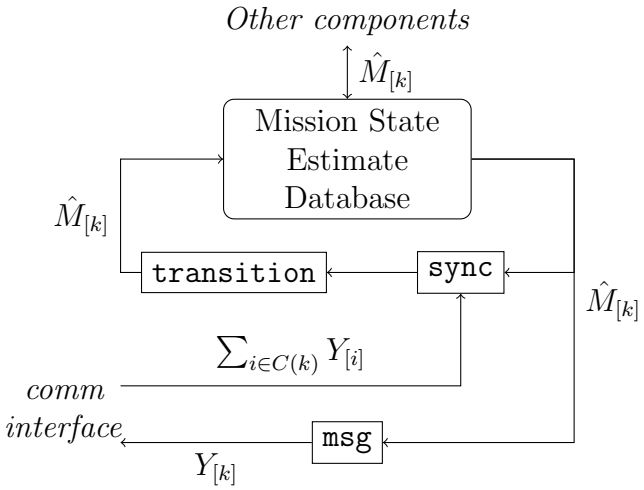


Figure 5: Conceptual model for mission modeler

These three data sets may be accessed by other components of the architecture; in particular, behavior sets will typically require access to the TSE's and ASE's representing task-points and agents involved with the task achieved by that behavior set.

2.2 Mission modeling data structure

A task-point state estimate data model is described in [14] and paraphrased here. We define the *task specification* for task i as

$$T_i = \{P_i, p_i^0\}$$

where P represents the global task parameters and p^0 represents the default parameters for task-point instances of the task.

We represent the j-th task-point instance of task i by

$$S_{ij} = \{P_{ij}, R_{ij}\}$$

where P_{ij} is the set of parameters defining the task-point and R_{ij} is a set of transition rules governing state transitions on the task-point. Both task point parameters and transition rules may be general, defined for all task-points belonging to a particular task; they may also be specific to that particular task-point instance. We then introduce a task-point state estimate as

$$\hat{S_{ij}} := \{\sigma_{ij}, \alpha_{ij}, C_{ij}, \tau_{ij}, I_{ij}, P_{ij}, R_{ij}\}$$

made of of components $\sigma_{ij} \in \{\texttt{'todo'}, \texttt{'assigned'}, \texttt{'done'}\}$, the task-point status; α_{ij}, the unique identifier of agent assigned to or having completed subtask S_{ij}; C_{ij}, the reported cost for agent α_{ij} to accomplish S_{ij} if $\sigma_{ij} =$ 'assigned'; $\tau_{ij} \in \Re$, the timestamp when C_{ij} was calculated by α_{ij}; and $I_{ij} \in \Re$, the initialization time of S_{ij}.[1] For our generalized mission model, unique task-point parameters will typically consist of its pose (if it is a task on geometric service points) and its requirements for completion (such as service time). This information would be defined at task-point state estimate instantiation; typically, this would occur by an agent executing a behavior set designed to detect and identify task-points. Additionally, our task-points would include generalized parameters such as failure timeouts.

Building of this framework for TSE's, we develop a data model for estimates of agent state as well. We represent agent k as

$$A_k := \{P_k, R_k\}$$

[1]For clarity throughout this thesis, we may refer to elements of state estimate by referencing either the direct element value: $U_{i,j}$ for the status of task-point j of task i); or, by naming the value and referencing the state-estimate: $status(\hat{S}_{i,j})$

consisting again of a set of parameters and a set of transition rules. Our agent state estimate is defined as

$$\hat{A}_k := \{\sigma_k, \tau_k, I_k, Q_k, P_k, R_k\}$$

where $\sigma_k \in \{\texttt{`idle'}, \texttt{`active'}\}$ is the agent status; τ_k is the timestamp when the agent state estimate was last updated; I_k is the initialization time; and Q_k defines the agent's available resources. In the case of an agent state estimate, the parameters would typically include agent pose information, the agent's active behavior, and any information that the underlying distributed algorithms composing the agent's active behavior set would be designed to transmit to other agents within the robotic network. The transition rules should again include a fault timeout.

2.3 Mission modeling protocols

It could be argued that, in the case of task-point state estimates, state transitions need only be carried out by whatever behavior set was handling that particular task-point, and in the case of agent state estimates, state transitions need only be carried out by the particular agent which is represented by the state estimate. Information would still propogate throughout the network but the number of state transitions performed by each node would be reduced. In that case, a mission modeler such as ours needn't even be developed; such transitions could be handled by behavior sets. However, one key feature of the mission state estimate data model and transition protocols is the introduction of a degree of fault tolerance; since the mission modeler operating on any agent can execute state transitions on any component of its known mission state estimate, task-

points assigned to faulty agents can be identified and appropriately handled. For example, if agent A fails, at some point the transition rules for all task-points' state estimates indicating assignment to agent A will indicate *to any other agent's mission modeler with knowledge of the task-points* that the task-point service attempt has failed. The task-points statii will be reset and they will re-enter the pool of `'todo'` task-points while the agent status will be reset and the agent will be a candidate for re-assignment and re-activation.

We first consider the process of comparing state estimates received via the communication interface from other agents to the state estimates contained within an agent's own believed mission model. The sync protocol for task-point

---

Protocol 2.1 `sync` protocol for task-point state estimates

Consider agent A with $\hat{M}^A$, receiving message $\bar{Y}^B$ containing $\hat{M}^B$ from agent B

for all $\hat{S} \in \hat{M}^B$ such that $\hat{S} \notin \hat{M}^A$ **do**
  copy $\hat{S}$ to $\hat{M}^A$.

for all task-points S in $\hat{M}^A$ **do**
  if $I^B > I^A$ **then**
    task-point has reset, $S^A \leftarrow S^B$
  else if given σ^A and σ^B, the condition in the following table return TRUE

Task-point S		σ^A		
		'todo'	'assigned'	'done'
U^B	'todo'	$\tau^B > \tau^A*$	$\tau^B > \tau^A$	FALSE
	'assigned'	$\tau^B \geq \tau^A$	see below	FALSE
	'done'	TRUE	TRUE	$\tau^B < \tau^A*$

$*$ or IF $\tau^B = \tau^A$ AND $\alpha^B > \alpha^A$

Assigned-Assigned: IF $\{(\alpha^B = \alpha^A)$ AND $(\tau^B > \tau^A)\}$ OR
IF $\{(\alpha^B \neq \alpha^A)$ AND $(\tau^B + C^B < \tau^A + C^A)\}$ OR
IF $\{(\alpha^B > \alpha^A)$ AND $(\tau^B + C^B = \tau^A + C^A)\}$

then
  $S^A \leftarrow S^B$

□

---

state estimates, shown in protocol 2.1, compares received TSE's with existing TSE's; in general, for each task-point agent A either keeps S^A or completely

replaces S^A with S^B. Based on the statii of S^A and S^B, the algorithm executes a number of boolean tests to ensure that the most accurate, up-to-date information is propagated. The synch protocol for ASE's, shown in protocol 2.2 is

Protocol 2.2 `sync` protocol for agent state estimates

for all $\hat{A} \in \hat{M}^B$ such that $\hat{A} \notin \hat{M}^A$ **do**
  copy $\hat{A}$ to $\hat{M}^A$
for all A in $\hat{M}^A$ **do**
  if $I^B > I^A$ **then**
    agent has reset, overwrite A^A with A^B
  else if $\tau^B > \tau^A$ **then**
    overwrite A^A with A^B

□

significantly simpler; accurate and current informaiton is guaranteed by simply keeping the most recently updated estimate.

To introduce the state estimate transition protocols, we can consider each type of state estimate as a sort of finite state machine, with the finite set of state defined above by the possible statii. Each type of state estimate has a corresponding set of possible transitions, governed by the state estimate's transition rules. These rules are tested and status transitions executed through a sequence of status transition functions.

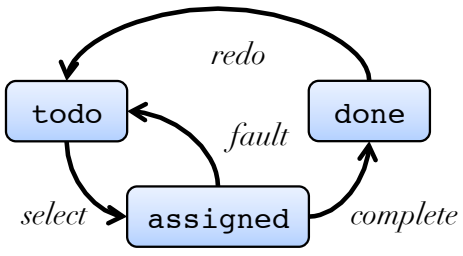


Figure 6: Status transition model for task-point state estimates

For task-point state estimates, we consider the status transitions provided

in [14] and shown in figure 6: *complete*, which takes the task-point status from `'assigned'` to `'done'`; *fault*, which takes the task-point status from `'assigned'` to `'todo'`; and *redo*, which takes the task-point status from `'done'` to `'todo'`. There is an additional status transition *choose*, taking the status from `'todo` to `'assigned'`; however, as we are leaving the details of selecting individual task-point instances to our behavior set components, this transition is not executed by the mission modeler. Transitions from `'done'` to `'assigned'` or from `'todo'` to `'done'` are not defined. The transition protocol for task-point state

Protocol 2.3 `transition` protocol for task-point state estimates

Let R_c, R_f, R_r represent the transition rules for a given task-point state estimate $\hat{S}$ for the *complete*, *fault*, and *redo* status transitions, respectively. Also let AgentID be the unique identifier of the agent executing the protocol and t represent the current system time.

for all $\hat{S}$ in mission state estimate $\hat{M}^k$ on agent k **do**
  if R_c **and** $\sigma =$ `'assigned'` **and** $\alpha =$ AgentID **then** $//complete(\hat{S})$
    $\sigma =$ `'done'`; $\alpha =$ AgentID; $\tau = t$
  else if R_f**and** $\sigma =$ `'assigned'` **then** $//fault(\hat{S})$
    $\sigma =$ `'todo'`; $\alpha =$ AgentID; $\tau = t$
  else if R_r **and**$\sigma =$ `'done'` **then** $//redo(\hat{S})$
    $\sigma =$ `'todo'`; $\alpha =$ AgentID; $\tau = t$; $I = t$

□

estimates, shown in protocol 2.3, combines the tasks of checking transition rules for each possible transition and execution of the appropriate transition into three status transition functions. For our mission plan, we consider general fault and redo transition rules based on a simple comparison of time elapsed since last update and redo and fault timeouts, provided within the task-point state estimate parameters:

$$R_f := t - \tau > \texttt{fault\_timeout}, \quad R_r := t - \tau > \texttt{redo\_timeout}$$

Transition rules for the *complete* transition are more complicated; for task-point instances of a DVR task where C is a measure of time-to-target plus service time, we could consider a transition rule

$$R_c := C < \epsilon_{complete}$$

where $\epsilon_{complete}$ is some completeness threshold specified within the task-point parameters.

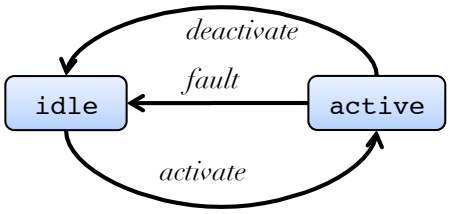


Figure 7: Status transition protocol for agent state estimates

For agent state estimates, we introduce the status transition model shown in Figure 2.3. Three status transitions are possible here; *fault* and *deactivate*, which both tranistion the status from `'active'` to `'idle'`, and *activate*, which transitions the status from `'idle'` to `'active'`. Similar to the *choose* transition for TSE's, the *activate* and *deactivate* transitions are governed by the behavior manager component; the mission modeler is only concerned with the transtion *fault* to provide fault tolerance to the mission situational awareness model. The

Protocol 2.4 Status transition protocol for agent state estimates

Let R_f be the transition rules for a given agent state estimate $\hat{\Lambda}$ for the *fault* status transition, and t be the current system time
for all $\hat{A}$ in mission state estimate $\hat{M}$ on agent k **do**
  if R_f **and** $\sigma =$ `'active'` **then** $//fault(\hat{A})$
    $\sigma =$ `'idle'`; $\tau = t$

agent state estimate transition protocol, shown in Protocol 2.4, also combines the task of checking the appropriate transition rule and executing the transition for the *fault* status transition. Similar to our transition rules for subtask state estimates for our mission model, we introduce a basic faulting transition rule:

$$R_f := t - \tau > \texttt{fault\_timeout}$$

where `fault_timeout` is defined in the agent state estimate parameters.

Behavior sets and the behavior manager must have access to several mission modeler protocols. First, the mission modeler provides a mechanism for creation of state estimates by behavior sets. Considering our specific demonstration mission plan, we see that agent state estimates initially need to be created onboard each agent, and subsequently shared (and copied) throughout the robotic network. Task-point state estimates belonging to the identification task will initially need to be created via the identification behavior set. As subsequent tasks are completed for each task-point instance, new task-point intances must be introduced for the logical "next task" required to be performed at that location. The duty of requesting the creation of a new task-point instance falls to individual behavior sets; however, our situational awareness model requires behavior sets to provide only the task required and specific task-point parameters. The mission modeler handles the details of assiging state estimates unique identifiers and building the framework and parameters required by the previously described protocols. In general, the modeler will provide a generation function to behavior sets:

$$\mathit{generate} : P, \texttt{task} \rightarrow S$$

For their internal algorithms, behavior sets must have access to read the current state estimate information for agents and task points. Thus, the mission modeler will provide read access to all task point data; additionally, it will allow other components the ability to overwrite the cost and paramater data for specific task-point state estimate instances, and the parameter data for agent state estimate instances.

2.4 User Interface

We include some background about the proposed user interface for the MotionArc architecture here, along with the mission modeling component, as the primary goal for our user interface is to provide the user with situational awareness feedback. This feedback consists, for the most part, of a representation of the beliefs held by some mission modeling component within the robotic network. In practice, this modeler component could be running on a sort of agent similar to a network packet-sniffer; the user interface function could be instantiated as the active behavior of that agent. Typically, this agent would be "virtual" in that it would not effect any type of state transition of control actuation within the robotic network; it would simply parse the current mission model into some sort of easy to understand reference frame for the user.

Our basic graphical user interface (GUI) will present some of the information contained within task-point and agent state estimates to the user. We include textual displays for details of agent and task-point state estimates. These details are intelligently sorted; for example, it might be useful for the user to see task-points sorted by status, listing all `'todo'` task points first, or it might be more helpful to sort by the agent to which each task-point is assigned, or by

type of task-point. Agent state estimates might be sorted by the behaviors the respective agents are performing or by their amount of available resources.

Since our generalized mission model handles agents and task-points that are predominantly spatially distributed, our user interface will also include a visual representation of task-point and agent poses, drawn directly from the respective state estimate parameters known by the visualizing agent. Based on task-point classes or types, additional information about the state of the task those task-points belong to could be included as well. For example, for task-points belonging to a DVR class task and mutually assigned to a single agent, task-point poses could be graphically linked based on the believed cost of each estimate. As a result, we would have a visual representation of the tour calculated by the task-acheiving agent through its assigned task-points.

The mechanics of the interactions and protocols within the user interface can be treated within the same framework as other behavior sets we intend to implement; these details will be included as an example in the following section describing that framework. We will also discuss protocols that expand the user interface from a simple feedback mechanism to one that allows user input to influence the actions of the robotic network.

3 Tasks and Design of Behavior Sets

We now turn our attention to the component for which the entire architecture is proposed: behavior sets, which are MotionArc -specific implementations of efficient task-acheiving algorithms for the tasks defined by a mission plan. Before strictly considering MotionArc behavior sets, let us briefly discuss the nature of the tasks that make up a MotionArc mission model. Independently of designing behavior sets, MotionArc end users must define the mission plan for a given robotic network mission; this plan primarily consists of describing the potential tasks that may arise throughout mission execution. In section 2 we introduced the data model for task-point instances; task definitions simply provide a template for the nature of these task-points. In practice, MotionArc will use a *task specification* to define specific transition rules and default parameters for the associated task-point instances. Recall from our description of the Mission State Estimate that those two task-point features are the only features that will vary in definition between task-points belonging to different tasks. Additionally, task specifications may describe features of the task independent of any one task point such as the envirionment in which the task-points may occur (if different from the global mission environment) and the rate at which these task-points are expected to occur. Both of these features will be used in determining the expected cost functions for a particular behavior set, a function required by the behavior allocation protocol introduced in the following section.

3.1 Requirements and design principles

In general, behavior sets need only to adhere to the model presented in section 1.3. Reiterating, behavior sets maintain an internal state and map sensor

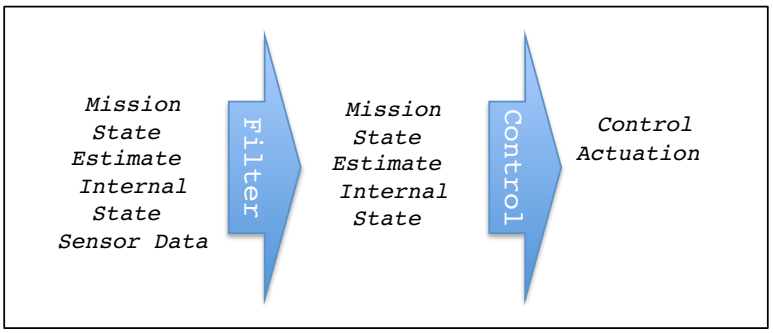


Figure 8: Behavior states contribute sensor information to the Mission State Estimate and effect control acuation on the environment

data and the current misssion state estimate to control actuation output and the next iteration of the mission state estimate. In the C^2 framework presented in [8], instances of distributed algorithms received and generated messages; in our forumulation, the Mission State Estimate is designed to maintain all information that would be passed through these messages, as MSEs are shared across the network over time.

Addressing several key design considerations will help to streamline the process of adapting algorithms as MotionArc behavior sets.

- **Sensor Data**: How will the algorithm "read" sensor data from the hardware interface provided by the Player/Stage software package, and how will it make control commands?
- **Mission State Estimates**: How will the algorithm make use of the information provided by the task-point state estimates for the associated task, and the agent state estimates for the other agents performing the task?
- **Internal State**: What is the internal state space?

The functional implementation of MotionArc , described in a subsequent section, provides the tools such that algorithm designers can address these con-

siderations. From an information perspective, the main decision to make is what algorithmic information should be maintained as internal state variables, and what should be contained within the parameters of the task-point and agent state estimates read and written by the algorithm. In general, information that passes between agents will be maintained as a part of the Mission State Estimate. For example, a behavior set designed to accomplish a DVR-type task will use the current local Mission State Estimate to determine which task-points to add to a service queue, and will execute the select task-point state estimate status transition when a task-point is selected. The behavior set would also update the estimated cost (in the DVR case, the remaining time or distance - to - complete) maintained within respective task-point state estimates. The actual ordered queue of task-point names that makes up a TSP tour, however, would be maintained as local state information.

3.2 From algorithm to behavior set

We now demonstrate the adaptation of a generic algortihm as a MotionArc *behavior set* by considering a generic algorithmic framework for the multiple vehicle, infinite capacity dynamic traveling repair-person problem (DTRP). Recall from section 1.3 the behavior set `filter` and `control` protocols F_B and A_B that result from the coupling of the behavior set with the Mission State Estimate protocols:

$$\Omega(t+1), \hat{M}^*(t) = F_B\left(\Omega(t), \hat{M}(t), \xi(t)\right) \tag{9}$$

$$u(t) = A_B\left(\Omega(t)\right) \tag{10}$$

where $\Omega(t)$ is the internal processor state, $\hat{M}(t)$ is the Mission State Estimate, and $\xi(t)$ is the sensor data. For the generic DTRP algorithm, we define the processor state as

$$\Omega := \{S_{\text{goal}}, Q, x_{\text{base}}\}$$

where S_{goal} is the task-point state estimate corresponding to the task-point instance the agent is currently working towards, Q is a (possibly empty) tour of task-point identifiers corresponding to task-points that are currently assigned to the agent, and x_{base} is the pose of a base or depot location for this agent. Assuming adequate low-level hardware control routines, the control output u is simply a desired pose toward which the robotic agent should navigate.

The generic DTRP algorithm can be summarized as follows: first, using some efficient method, the agent selects some number of task-points belonging to the respective task to service. If none are selected, the agent moves towards its base or depot location. Otherwise, the agent labels the first selected task-point as its goal and moves towards that task-point location. When it arrives, it performs some service at or on the task-point until the task-point is determined to be complete, at which time the agent selects the next task-point as its goal.

Recalling the Mission State Estimate data model presented in section 2, we can now define the parameters of task-points belonging to a DTRP task as the task-point pose, required service time, and minimum service distance:[2]

$$\hat{P}_{DTRP,j} := \{\hat{x}_{DTRP,j}, s_{DTRP,j}, r_{DTRP,j}\}$$

The behavior set protocols may also require global task parameters for execution,

[2]as with the other components of state estimates, we may refer to parameters of a state estimate using either the $\hat{x}_{i,j}$ notation or $pose(S_{i,j})$

such as v, the average (or constant) agent speed. Protocols 3.1 and 3.2 show

Protocol 3.1 Generic DTRP `filter` protocol

Given $\hat{M}, \Omega, \xi$ for agent k:
update the agent state estimate Λ_k in $\hat{M}$ with the pose sensed in ξ.
if S_{goal} is *Null* **then**
  $Q \Leftarrow$ `select_filter` $\left(\{i \in \{1, \ldots, |\hat{S}_{\text{DVR}}|\} \mid status(\hat{S}_{\text{DTRP},i})\}\right)$
  if Q is not empty **then**
    for $i = 1to|Q|$ **do** $//Q := q_1, q_2, \ldots$
      Execute the *select* status transition on q_i
      if $i = 1$ **then**
        $cost(q_1) \Leftarrow ||pose(q_1) - pose(\hat{A}_k)||/v + service\_time(q_1)$
      else
        $cost(q_i) \Leftarrow ||pose(q_i) - pose(q_{i-1})||/v + service\_time(q_1) + cost(q_{i-1})$
    $S_{\text{goal}} \Leftarrow q_1$ and remove q_1 from Q and re-number accordingly
else
  if $status(S_{\text{goal}}! =$ '`assigned`' **then** //If a status transition has occured
    if $|Q| > 0$ **then**
      $S_{\text{goal}} \Leftarrow q_1$ and remove q_1 from Q and re-number accordingly
    else
      $S_{\text{goal}} \Leftarrow Null$
  if $||pose(S_{\text{goal}}) - pose(\hat{A}_k)|| > service\_dist(S_{\text{goal}})$ **then** //Goal task-point not within service range
    $\delta_{\text{cost}} = ||pose(S_{\text{goal}}) - pose(\hat{A}_k)||/v - (cost(S_{\text{goal}}) - service\_time(S_{\text{goal}}))$
    for all $S \in S_{\text{goal}} \cup Q$ **do**
      $cost(S) \Leftarrow cost(S) + \delta_{\text{cost}}$
  else //Goal task-point within service range
    `service`(S_{goal})

our generic DTRP `filter` and `control` protocols and how they make use of the Mission State Estimation framework. Specific algorithms would specify a task-point selection policy `select_filter` and a service action policy `service`. For the sake of demonstration, our service action policy simply requires agents to "wait" – once they have reached goal task-point poses – for the amount of time specified by the *service_time* parameter of the associated task-point state estimate.

Protocol 3.2 Generic DTRP `control` protocol

1: **if** $S_{\text{goal}}! = Null$ **then**
2: $\quad u \Leftarrow pose(S_{\text{goal}})$
3: **else**
4: $\quad u \Leftarrow x_{\text{base}}$

It is important to note that none of the status transitions governed by the mission modeling protocol are executed during a pass through the behavior set protocols. Of particular interest is the *complete* status transition, the result of which (switching status from `'assigned'` to `'done' f`) is required for a task-point to be removed from the service queue. The actual *complete* status transition would happen during the first pass of the mission modeling execution protocol after the cost (in this case, time-to-service completion) fell below some threshold.

Several efficient task-point selection policies are given in [4] that have known performance costs for particular task situations and are shown in the following paragraphs. As will be discussed in section 4, we require that these performance costs be functions of the number of agents assigned a particular behavior, m, and the global task parameters introduced by a *task specification*. For DTRP tasks, we define these parameters as

$$P_{\text{DTRP}} := \{E, \lambda, \bar{s}, \bar{\sigma}, v\}$$

where E is the area of the mission environment, λ is the measured or predicted arrival rate of task instances, $\bar{s}$ is the expected service time for task instances, $\bar{\sigma}$ is the variance in service time, and v is the average agent speed.

The m Stochastic Queue Median (mSQM) Policy selects task-points in a manner such that each agent will serve task-points on a first-come, first-serve

(FCFS) basis within an approximately equitable partition of the mission environment. The analysis in [4] for this algorithm as well as one to follow assumes that agents serve within strictly equal partitions; in practice, we implementent the partitioning without explicitly solving for the equitable partition of the space. Instead, we consider the Voronoi partition of the space based on the agent positions at any given point in time. Agents with smaller partitions are less likely to have the required number of taskpoints within their partition and as such will tend towards moving to their "base" location. The "base" locations can be either calculated *a-priori* for m agents; or, agents can use their "base" location as the calculated centorid of their Voronoi polygon at a given point in time. This version of the policy is shown in protocol 3.3. As agents move towards their Voronoi centroids, the Voronoi partition will approximate an equitable partition of the space, a consequence illustrated in various deployment algorithms in [8]. The static, equitible partition *mSQM* policy is optimal in the light loading case, that is, when $\lambda\bar{s} \to 0$, and the expected system time in the asymptotic regime is

$$T_{\text{mSQM}} \; \frac{E[\min_{x_0} \in D^* || X - x_0 ||]}{v} + \bar{s}$$

where D^* is the set of "base" locations.

Protocol 3.3 *mSQM* task-point selection policy

1: $S^* \Leftarrow \{i \in \{1, \ldots, |\hat{S}_{\text{DVR}}|\} \mid status(\hat{S}_{\text{DTRP},i})\}$
2: sort S^* such that $\tau_i < \tau_{i+1}$ for all $S_i \in S^*$
3: calculate $V_k = V(\hat{A}_k, A^*)$, the Voronoi region associated with agent k given the set of agents A^* performing this behavior set
4: **for all** $S \in S^*$ **do**
5: **if** $pose(S) \in V_k$ **then**
6: **break**
7: $Q \Leftarrow S$

The G/G/m version of a TSP-routing policy is shown in protocol 3.4. In this

policy, task-point state estimates are grouped, as they arrive, into sets of size n; sets are served by individual agents on a FCFS basis. Assuming an optimal TSP tour for each set, an asymptotic upper bound on the system time for this policy in the high loading ($\lambda\bar{s} \to 1$) case is

$$\beta_{TSP}^2 \frac{\lambda E(m+1)}{2m^2v^2(1-\lambda\bar{s})^2} + \frac{\beta_{TSP}\lambda\sqrt{2E(m+1)(1/\lambda^2+\sigma_s^2/m^2)}}{2mv(1-\lambda\bar{s})^{3/2}} + \frac{\beta_{TSP}^2\lambda E}{mv^2(1-\lambda\bar{s})}$$

where $\beta_{TSP} = 0.72$. For stability, the policy requires

$$n > \beta_{TSP}^2 \frac{\lambda E}{m^2v^2(1-\lambda\bar{s})^2}$$

Protocol 3.4 *G/G/m-TSP* task-point selection policy

1: $S^* \Leftarrow \{i \in \{1,\ldots,|\hat{S}_{\text{DVR}}|\} \mid status(\hat{S}_{\text{DTRP},i})\}$
2: sort S^* such that $\tau_i < \tau_{i+1}$ for all $S_i \in S^*$
3: **if** $|S^*| \geq n$ **then**
4: order the first n task-point state estimates in S^* into a minimum TSP tour Q_{TSP}.
5: $Q \Leftarrow Q_{\text{TSP}}$
6: **else**
7: $Q \Leftarrow \{\}$

The Independent Partitioning - TSP policy, shown in protocol 3.5, similarily services tasks in a TSP tour of length n, this time calculating the tour through the first n tasks to arrive within an agents respective Voronoi partition. As with the *mSQM* policy, the performance results shown here apply to the case of equal partitions; in practice, we use the deployment methods described above.

Using the formulation for *behavior sets* presented in this section, algorithm designers ensure that their algorithims will fit within the MotionArc system. In section 5.3 we will introduce a configuration method for the MotionArc

Protocol 3.5 *Partition-TSP* task-point selection policy

$S^* \Leftarrow \{i \in \{1, \ldots, |\hat{S}_{\text{DVR}}|\} \mid status(\hat{S}_{\text{DTRP},i})\}$
sort S^* such that $\tau_i < \tau_{i+1}$ for all $S_i \in S^*$
calculate $V_k = V(\hat{A}_k, A^*)$, the Voronoi region associated with agent k given the set of agents A^* performing this behavior set
for all $S \in S^*$ **do**
    if $pose(S) \in V_k$ **then**
        $Q \Leftarrow Q \cup S$
        if $|Q| \geq n$ **then**
            break
if $|Q| \geq n$ **then**
    $Q \Leftarrow TSP(Q)$
else
    $Q \Leftarrow \{\}$

system in which users specify both mission information and the `filter` and `control` protocols for their designed behavior sets along with any associated performance cost functions. Once the mission plan, task specifications, and behavior sets have been designed and loaded into the MotionArc system, we are now prepared to compute the optimal allocation of agents to tasks (and the respective behavior sets) and commence mission execution.

4 The Behavior Allocation Problem

We now turn from our architectural designs to the highest level of control within the MotionArc system: the selection of appropriate behaviors by each agent within the system. We first recast our problem as a modified instance of the integer resource allocation problem:

$$\min_{x} \quad \sum_{i}^{n} f_i(x_i, \mathbf{s}_i) \tag{11}$$

$$\text{subject to:} \quad \sum_{i}^{n} x_i = N \tag{12}$$

$$x_i \in \mathbb{Z}_{\geq 1} \quad \forall i \in \{1, \ldots, n\} \tag{13}$$

We consider f_i to be a cost function for behavior i that is strictly convex and nonincreasing in x_i, the number of (integer) resources allocated to behavior i. We let $\mathbf{s}_i$ represent other variables, as functions of time, that contribute to the cost function. We wish to optimize over x – the allocation of resources to behaviors – at a given point in time such that all other input variables to the f_i's are fixed. To obtain the optimum, we use a distributed gradient descent method introduced in [20] for the continuous problem together with a heurestic presented in [6] for determining a "good" (close to optimal) integer solution.

4.1 Solving the continuous problem

Let $f_i' := \frac{\partial f_i}{\partial x_i}$. Consider the following distributed iterative algorithm:

$$x_i(t+1) = x_i(t) - W_{ii} f_i'(x_i(t), \mathbf{s}_i(t)) - \sum_{j \neq i} W_{ij} f_j'(x_j(t), \mathbf{s}_j(t)), \quad i = \{1, \ldots, n\} \tag{14}$$

for $t = 0, 1, \ldots$; or, written as a vector,

$$x(t+1) = x(t) - W\nabla_x f(x(t), s(t))$$

Xiao and Boyd [20] show that this weighted gradient descent method, for a feasibile $x(0)$ with $\mathbf{1}^T x(0) = N$, provides feasible solutions $x(t) \in \{x \mid \mathbf{1}x = N\}$ for all $t > 0$ if the weight matrix W is *doubly stochastic*:

$$\mathbf{1}^T W = W\mathbf{1} = 0$$

Note that $\mathbf{1}$ is the vector of length n containing a one at each element. Setting $W_{ii} = -\sum_{j \neq i} W_{ij}$ such that this is the case, the method becomes

$$x_i(t+1) = x_i(t) - \sum_{j \neq i} W_{ij} \left(f_j'(x_j(t), \mathbf{s}_j(t)) - f_i'(x_i(t), \mathbf{s}_i(t)) \right), \quad i \in \{1, \ldots, n\} \tag{15}$$

The change in resources for each behavior i is calculated via a weighted sum of the total cost derivatives acheived by exchanging resources with each of its neighbors. Convergence conditions on this iterative method are given in [20] assuming convex cost functions with bounded second derivatives. Several methods for computing the weight matrix are given, for our purposes, we choose a "best constant" weight selection method reproduced here. Letting the exchange weights be equal to constant α, the weight matrix becomes

$$W = -\alpha L$$

where L is the *Laplacian* matrix for a complete graph of cardinality n:

$$L_{ij} = \begin{cases} -1 & i \neq j \\ n-1 & i = j \end{cases}$$

and the maximum convergence rate for this method is obtained using optimal constant weight α^*:

$$\alpha^* = \frac{-2}{\lambda_1(L) + \lambda_{n-1}(L)}$$

where λ_1 is the largest eigenvalue and λ_{n-1} is the second-smallest eigenvalue. In our case, where the resource-exchange graph is complete, all eigenvalues $\lambda_{j \neq n}(L) = n$, therefore, our optimal constant value is $\alpha^* = -\frac{1}{n}$ and the iteration becomes

$$x_i(t+1) = x_i(t) - \sum_{j \neq i} \frac{1}{n} \left(f_j'(x_j(t), \mathbf{s}_j(t)) - f_i'(x_i(t), \mathbf{s}_i(t)) \right), \quad i \in \{1, \ldots, n\} \tag{16}$$

4.2 Solving the Integer Variable Problem

Bretthaur and Shetty [6] provide several algorithms for obtaining an optimal integer soluion to the nonlinear resource allocation problem (problem 11) given a method for computing the optimal continuous solution as provided here. One heuristic they provide for generating feasible integer solutions to problem 11 can also be used alone to provide a good-enough sub-optimal solution for our purposes, shown in protocol 4.1.

We assume that the other variables $\mathbf{s}$ determining the cost functions for each behavior are changing at a slow rate. For our demonstrative case of dynamic-vehicle-routing type tasks, this assumption is vaild in practice; the contribut-

Protocol 4.1 Heuristic for the Integer Variable Problem

Given continuous solution x^*:
for all $i \in \{j \mid x_j^* \notin \mathbb{Z}\}$ **do**
  Round down x_j^*
Let $\hat{x}_1, \ldots, \hat{x}_n$ be the resulting rounded solution
Order the resulting variables such that $\frac{\partial f_1}{\partial x_1}(\hat{x}_1, \mathbf{s}_1) < \frac{\partial f_2}{\partial x_2}(\hat{x}_2, \mathbf{s}_2) < \ldots < \frac{\partial f_n}{\partial x_n}(\hat{x}_n, \mathbf{s}_n) < 0$.
$i \leftarrow 1$
while $i \leq n$ **do**
  if $(\sum_{j \neq i} \hat{x}_j) + \hat{x}_i + 1 \leq N$ **then**
    $\hat{x}_i \leftarrow \hat{x}_i + 1$
  $i \leftarrow i + 1$

ing variables such as environment size and expected service time are observed throughout the lifetime of the mission and are typically static or nearly static. As a result, we design our system to re-calculate the optimal allocation of agents to behaviors periodically throughout the mission, but at a rate much slower than the convergence rate of the above algorithms. allow agents to re-assign themselves if necessary. We also allow for a re-calculation when one of several allocation-critical events occur:

- The number of available agents changes, i.e. when an agent is added to the system or failure is detected within one or more agents.
- The mission plan changes, i.e. new tasks are introduced requiring different behaviors and cost functions

4.3 Implementation of the allocation mechanism

Consider three behaviors (I, H, and L) making use of the selection policies similar to those disucssed in the previous section. We let m_i represent the number of agents assigned to behavior set i, and $\mathbf{s_i}$ represent the additional cost variables: λ_i, the measured task-point arrival rate; $\bar{s_i}$, the expected service

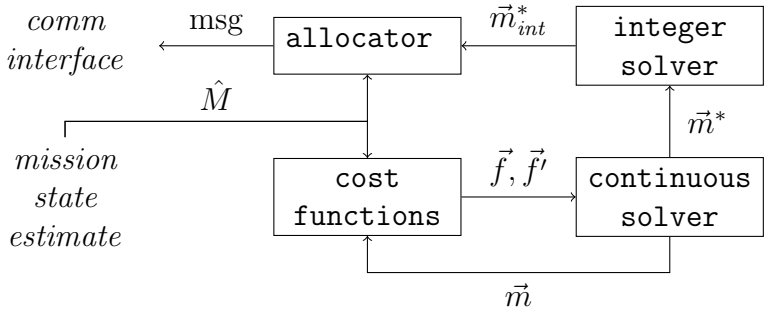


Figure 9: Conceptual model for behavior manager

time; A, the mission environment area; v, the average agent speed; σ_i, the service time variance, and ρ_i, the loading factor equal to $\lambda_i \bar{s}_i$. Behavior set I is an identification behavior, making use of the *G/G/m-TSP* task-point selection policy. Its performance cost and cost derivative assumed to approximate the proposed upper bound:

$$f_I(m_I, \mathbf{s}_I) =$$
$$\frac{\beta_{TSP}^2 \lambda_I A}{2v^2(1-\rho_I)^2}\left(\frac{m_I+1}{m_I^2} + \frac{v\sqrt{2(1-\rho_I)}}{\sqrt{A}}\frac{\sqrt{(m_I+1)(1/\lambda_I^2+\sigma_I^2/\lambda_I^2)}}{m_I} + \frac{1-\rho_I}{m_I}\right)$$
$$\frac{\partial}{\partial m} f_I(m_I, \mathbf{s}_I) =$$
$$\frac{\beta_{TSP}^2 \lambda_I A}{2v^2(1-\rho_I)^2}\left(-\frac{m_I+2}{m_I^3} - \frac{v\sqrt{2(1-\rho_I)}}{2\sqrt{A}}\left(\frac{m_I^3+3\sigma_I^2\lambda_I^2 m_I+4\sigma_I^2\lambda_I^2+2m_I^2}{\sqrt{\frac{(m_I+1)(m_I^2+\sigma_I^2\lambda_I^2)}{\lambda_I^2 m_I^2}}\lambda_I^2 m_I^4}\right) - \frac{(1-\rho_I)}{m_I^2}\right)$$

Behavior set H is a service behavior set for a task that is expected to exhibit heavy loading behavior ($\rho \to 1$), making use of the *partition-TSP* selection policy, with performance cost and cost derivative given as:

$$f_H(m_H, \mathbf{s}_H) = \beta_{TSP}^2 \frac{\lambda A}{m_H^2 v^2 (1-\rho_H)^2} \qquad \frac{\partial}{\partial m} f_H(m_H, \mathbf{s}_H) = \frac{-2\beta_{TSP}^2 \lambda A}{m_H^3 v^2 (1-\rho_H)^2}$$

Behavior set L is a service behavior set for a task that is expected to exhibit light loading behavior ($\rho \to 0$), making use of a selection policy simliar to *mSQM*, with performance cost and cost derivative given as:

$$f_L(m_L, \mathbf{s}_L) = \sqrt{\frac{A}{m_L}}\frac{1}{v} + \bar{s}_L \qquad \frac{\partial}{\partial m} f_L(m, \mathbf{s}_L) = -\frac{\sqrt{A}}{2v m_L^{3/2}}$$

To implement the gradient descent allocation method, we introduce an "idle" behavior set and cost function. The initial allocation of M agents to N behaviors is such that $m_i = 1$ for all behavior sets other than "idle", and $m_{idle} = M - N$. We give the "idle" behavior set a very high constant cost; thus, all derivatives of the "idle" behavior set are identically equal to 0 for all m_{idle}. As a result, as the algorithm evolves agents will only be removed from the "idle" behavior and assigned to others.

Since most of these cost equations have $1/m_i$ terms, the second-derivatives are not bounded as $m \to 0$ for any behavior set. We alleviate this by requiring that all non-"idle" behaviors be assigned at least 1 agent at all times; in practice, so long as there was any cost associated with a task, we would want at least one agent performing that task, even if the optimal integer allocation would have assigned 0 agents. Our final, modified allocation algorithm is shown as protocol 4.2. We allow the algorithm for the continuous solution to run until the marginal cost improvement of subsequent iterations is less than 0.0001, and then apply protocol 4.1 to get an approximate integer solution.

We now present the assignment results from implementing the algorithm for our proposed mission plan consisting of an identify task and two service tasks corresponding to the behaviors presented above. We assume that the distribution of service tasks is such that $\lambda_H = 0.8\lambda_I$ and $\lambda_L = 0.2\lambda_I$. The

Protocol 4.2 An iteration of the allocation mechanism with $\mathbf{s}_i$ constant for all behaviors i

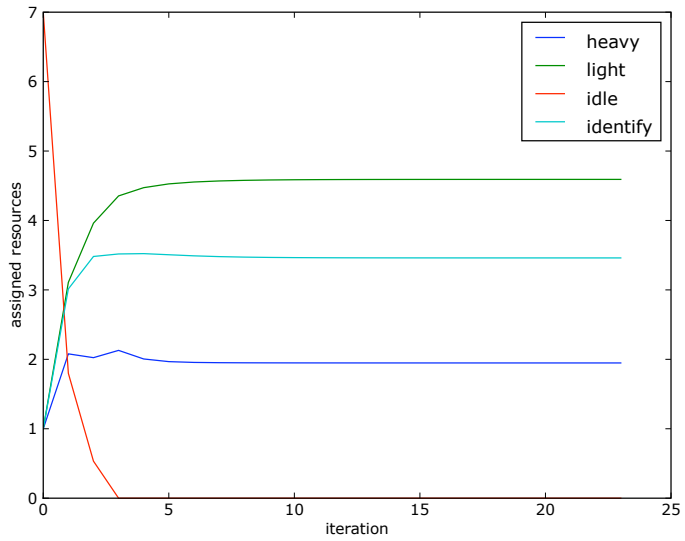
Let $m_i^* = 0$ for $i = idle$ and $m_i^* = 1$ otherwise
for all behavior nodes i **do**
  for all behavior nodes $j \ = i$ **do**
    $\delta \Leftarrow \frac{1}{N}(f_j'(m_j, \mathbf{s_j}) - f_i'(m_j, \mathbf{s_j}))$
    if $\delta >= 0$ **then** //sending resources from j to i
      $\delta \Leftarrow \min\left\{m_j - m_j^*, \delta\right\}$
    else //sending resources from i to j
      $\delta \Leftarrow \min\left\{m_i - m_i^*, \delta\right\}$
    $m_i \Leftarrow m_i + \delta$
    $m_j \Leftarrow m_j - \delta$

evolution of the algorithm is shown in figure 10a for an initial allocation of 10 agents to the three behaviors, and in figure 10b for an initial allocation of 100 agents to the three behaviors. For this initial allocation, we used constant parameters $\bar{s}_I = \bar{s}_H = \bar{s}_L = 1$ and $\lambda_I = 0.99$ such that behavior H is in the heavy-loading situation and behavior L is in the light-loading situation. We assume $A = 1$ and $v = 0.3$ for all behaviors. We also assume that $\sigma = 0$ for all behaviors, which significantly simplifies the cost function associated with the $G/G/m - TSP$ task-point selection policy.

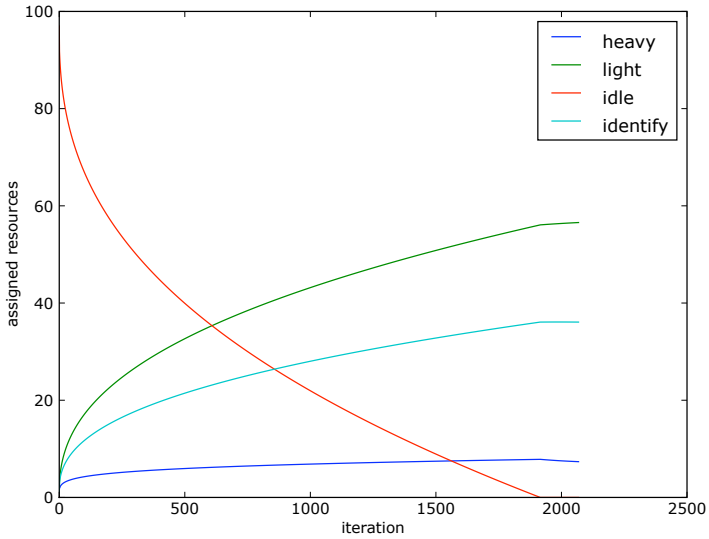
In MotionArc , once the allocation algorithm is used to obtain an initial assignment, the optimal assignment is periodically re-calculated to account for changes in mission parameters $\mathbf{s}$ as well as any faulted or additional agents. To study the response of the system to a slowly varying general allocation parameter, we varied the expected service time for each task type according to

$$\bar{s} = 1 + 0.5\sin(2\pi t/T)$$

where T is the total number of observations and the assigment was re-calculated at each $t = 1, 2, 3, \ldots, T$. We also allowed the estimated arrival rate for each



(a) 10 agents

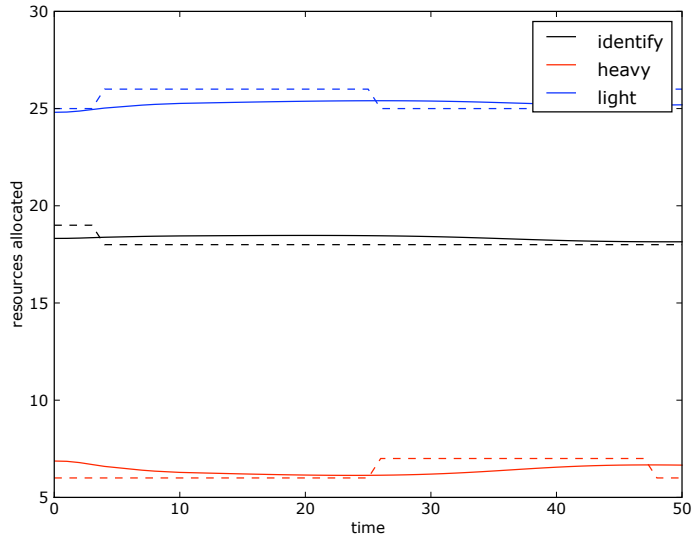


(b) 100 agents

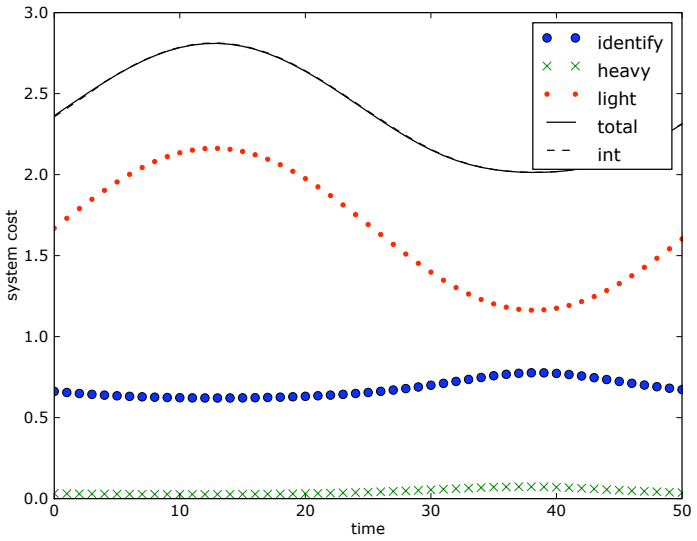
Figure 10: The allocation algorithm converges quite quickly for 10 agents and much slower for 100 agents; in both cases, introducing the "idle" task cost function and limiting the minimum number of assigned agents to be one for any non-"idle" task allows for a solution to be reached despite unbounded second derivatives

task type to vary such that the overall loading factors remained constant and equal to the values given above. In figure 11a, we show the continuous and integer optimal assignments of 50 agents over a range of 50 observations; the associated individual behavior costs along with the total cost are shown in figure 11b. We see that the overall assigment responds well to such a slowly-varying system parameters. Figure 11b validates the use of a "good enough" heuristic for finding the integer solution, as the total sytem cost using integer assignment is quite close to the optimal system cost obtained from the continuous solution. The number of iterations required for convergence of the allocation algorithm at each observation point are shown in figure 12. After the initial effort to find an optimal assigment from a significantly sub-optimal starting point, we see that the number of iterations required at each subsequent time step is quite small. This makes sense, considering that at each subsequent time step, the algorithm is starting with a very-near-optimal initial assignment. Finally, in figure 13, we present the results when 10 agents are added mid-mission. We see that the algorithm efficiently distributes these injected resources amongst the three behavior sets. In this example, we used a varying parameter $\bar{s} = 1 + 0.5\sin(\pi t/T)$.

These results indicate that such a combinatorial method can be used quite effectively within a system such as MotionArc to manage the assignment of agents to mission tasks. In our preliminary experimental implementations, we have only used this method in a centralized manner, where a control agent periodically monitors the assignment state and issues assignment commands to the network of agents as need be. In the future, however, it is easy to imagine that such a method could easily be adapted into a distributed allocation mechanism. Each agent could maintain an estimate of the assignment state, monitor the mission environment, calculate the appropriate assignment, and



(a) Continuous (solid line) and integer (dashed line) allocation



(b) Costs for each behvaior and combined system cost

Figure 11: A relatively optimal allocation is acheived for a system of 50 agents and 3 tasks in the presence of a varying system parameter

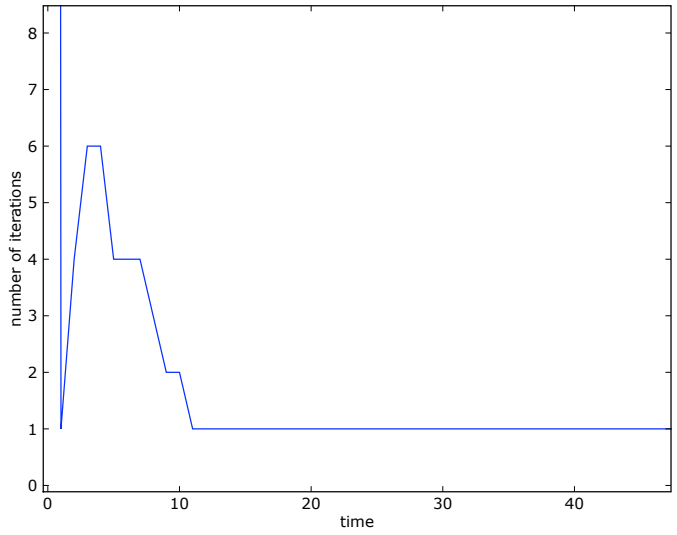
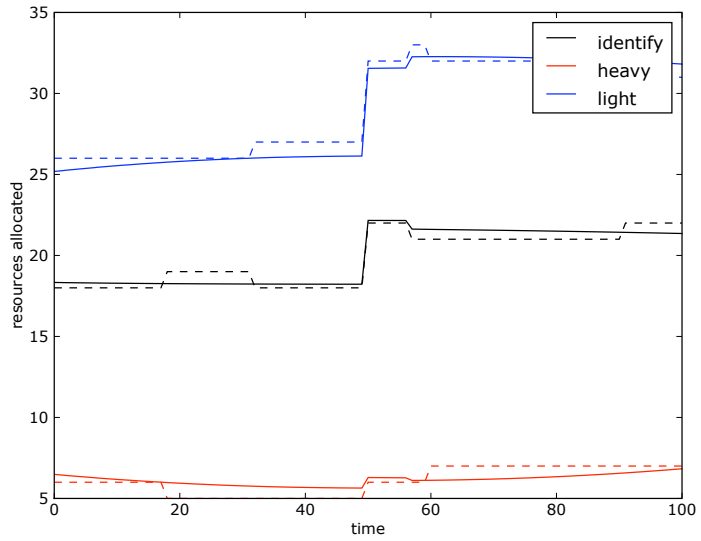
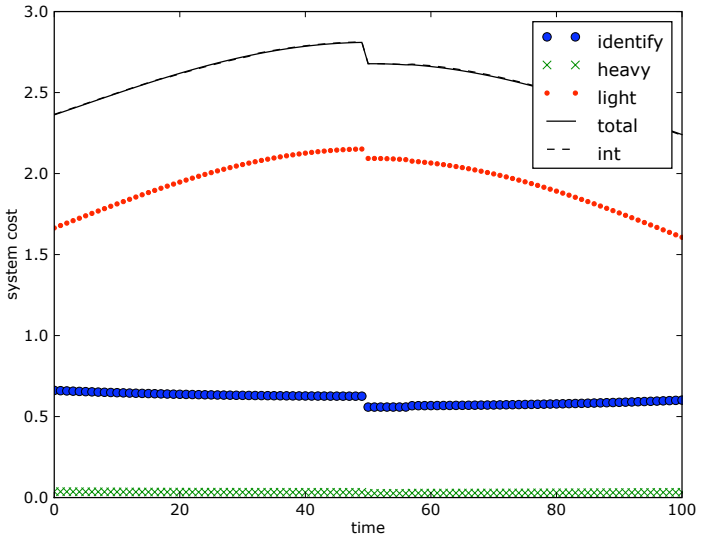


Figure 12: Iterations required for convergence of the allocation algorithm are significantly reduced once the initial optimal allocation is calculated

switch behaviors if it is determined that the active behaivor is not the most efficient. Certain safety protocols would have to be established to preven the entire network of agents from oscillating between two tasks in situations where there are many near-optimal assignments, but once this issue is addressed, such a mechanism could be quite powerful.



(a) Continuous (solid line) and integer (dashed line) allocation solution



(b) Costs for each behavior and combined system cost

Figure 13: The algorithm handles the injection of 10 additional agents as mission resources.

5 MotionArc implementation and execution

To this point, we have introduced the theoretical and conceptual background on which the MotionArc system is built. The remainder of this thesis will focus on the practical implementation of the MotionArc system with the specific target of managing missions involving the DVR-type tasks we would like to demonstrate.

To write and distribute the MotionArc software, we use the Python scripting language. The selection of Python was based on several key features, not the least of which was the desire to explore the use of Python to write our stand-alone mobile robotic algorithms. Some of the influential factors were:

Scripting: Python's nature as a scripting language lends it to use for developing algorithms. In fact, the language and grammer used in writing Python scripts is quite similar to the generally-accepted psuedocode used academically to describe algorithms.

Resources: There are a number of packages available for Python that are useful for robotic algorithm development including an advanced, MATLAB-like numerics package; several computational geometrey packages; plotting and graphics packages; and most importantly, a set of bindings for the Player/Stage client interface that allows us to use Python to develop algorithms that efficiently make use of the Player/Stage hardware abstraction.

Tools: Efficient tools within the Python standard library that make develpment of MotionArc components straightforward.

The end user of the MotionArc system will have access to two forms of the software. The first is a Python package against which end users can design

behavior sets and mission plans. The second is an executable script which, when properly configured, will load user designed behavior sets and execute user defined mission plans within the MotionArc framework on each agent within a robotic network.

5.1 Implementing system components

In section 1.4 we introduced the core components of the MotionArc system. The hardware and hardware interface components are implemented separately from the MotionArc system; the Player/Stage abstraction software provides the interface to the remaining components. Inter-agent communication is handled within MotionArc using implementations of the Python *socket* and *SocketServer* packages. These packages open up a TCP communication channel between agents using a specified communication address.

Other MotionArc components send messages by calling the message send method of the message sender. This submits each message to a message queue, from which messages are removed and sent over a TCP socket. The message transmitter maintains a list of all communication addresses within the network and after each successful (or failed) transmission, maintains a status record for that address which, in effect, builds an estimate of the local communication graph. The messages themselves consist of binary strings with a unique message identifier used for message confirmation. Python provides the pickle data-persistance package which converts any Python object into a binary string represenation that can be unpacked by another script running the appropriate library. For example, an entire Mission State Estimate, implemented as a Python object, can be packaged for transmission using this method.

Incoming messages are handled by the MotionArc *message server*. This server continuously listens on a TCP socket for any incomming signals, and invokes a message handling routine based on the data contained within the message. Our server assigns incoming messages to either a mission modeling message queue (for state estimate information messages) or an application message queue (for commands, such as assignment requests or user input). The message modeler and the application each have message parsing routines that take messages from the respective queues and unpack and handle the data appropriately.

To maintain the local mission information, we design a custom MotionArc *mission state estimate* that maintains the agent and task-point state estimates as a database. Other components may access or modify the mission state data through several data management routines. Use of these routines, as opposed to accessing the data directly, ensures integrity of the data. For example, for the task-point state estimate `merge` protocol to ensure that the most up-to-date information is propagated throughout the network TSE timestamp attributes must be updated whenever the data is updated. The MotionArc mission state estimate parameter and cost settor routines take care of the details of updating this timestamp whenever a user desires to update a parameter or cost, respectively. This data class also implements individual state estimate status transitions and merges the current data with a provided database of update data.

The mission state estimate `filter` and `control` protocols are implemented by the MotionArc *mission modeler*, which also handles incoming message data appropriately flagged by the *message server*, and periodically packages and submits state estimate update messages to the *message sender*. The `filter` protocol is broken into component processes for parsing and merging mission state

estimate update data and executing mission state estimate status transitions. The `control` protocol consists simply of the broadcasting of mission state estimate update messges to other agents on the communciation network.

The MotionArc *behavior manager* component maintains the behavior set execution protocols, behavior set internal states, sensor access methods, and control output methods for each available behavior set. It also handles assignment commands received from other agents and securely activates and deactivates behavior sets. User specified or standardized cost and cost derivative functions are maintained for each available behavior set as well. If the MotionArc instance is labeled as `CONTROL` (the central command center and user interface), these functions are used to monitor the current assignment state, calculate the optimal assignment, and send assignment commands as necessary.

5.2 The MotionArc Package

For end users to implement distributed algorithms as behavior sets within the context of the MotionArc framework, we provide the MotionArc Python package. The primary components are template classes from which MotionArc implementations of algorithms can be derived. Designers will implement their algorithm as a derived form of the *behavior set* MotionArc class, in which they provide the behavior set `filter` and `control` algorithms. The template class provides references to the methods of MotionArc components required for successful interoperation between the algorithm and the MotionArc architecture. Specifically, users have access to the local *mission state estimate* management methods (data accessors, settors, and transition functions), the *message sender* message queue, and a Player client for the local agent. To access and control the

hardware through the Player server, users would then instantiate device proxies through the provided Player client.

The MotionArc package also provides templates for developing cost functions and task-specifications required for the mission plan.

The MotionArc package will also provide several useful tools for algorithm development. Currently, the package includes several basic elements for demonstration purposes, including

- A *TSP Tour Queue* for building and managing TSP tours through specified task-point instances
- A greedy method for calculating a TSP tour
- A routine for determining local and global Voronoi partitions and Delaunay triangulations

5.3 The MotionArc Application

Once algorithms have been adapted as MotionArc behavior sets, the focus shifts to implementing these behavior sets within the context of a mission, using the MotionArc system. For this purpose, we provide the MotionArc application script. This script is intended to run on each agent within the network and, based on a specified mission plan, implement the required behavior sets. Mission execution using the MotionArc application can be summarized in two categories

1. Mission plan specification, lodaing of behavior sets, and application configureation
2. Operation of components and mission completion

The MotionArc application script essentially manages the inter-operation of all the previously described MotionArc components, along with the user-specified behavior sets. The task specification and behavior set information, along with general mission parameters, are specified by the user in the mission-specific Python configuration script. As the final command of the configuration script, the MotionArc application is started. Upon loading the configuration information, the application starts each component process and serves as a local process manager.

Two options are provided for MotionArc component exectution, which amount to either a mostly-series or mostly-parallel implementation of the mi-sison modeling and behavior execution protocols. The communication compo-nents the *message sender* and *message server* are always run as independent, threaded processes. In the mostly-series implementation, the application will execute each of the processes comprising the mission modeling and behavior set components once, in order, before repeating the cycle:

1. Parse application messages (commands)
2. `Parse` mission modeling messages
3. Mission state estimate `merge` protocol
4. Behavior set `filter` protocol
5. Behavior set `control` protocol
6. Mission state estimate `transition` protocol
7. Mission state estimate `broadcast` protocol

Figure 14: Series MotionArc application process execution

Processes 2,3, and 6 make up the mission modeling `filter` protocol. As each subsequent process does not execute until the prior has completed, there is no need to worry about the integrity of the mission state estimate data throughout the process.

The mostly-parallel implementation starts the component modeling and behavior processes as separate, asynchronous threads:

1. Application message parsing thread
2. Modeling message parsing and state estimate merging thread
3. Behavior set `filter` thread
4. Behavior set `control` thread
5. Modeling `transition` thread
6. Modeling `broadcast` thread

To ensure integrity of the mission state estimate data, it is protected with a Python *thread lock* element, which allows only one thread to access the data at any given time. The date is not locked for the entire execution time of the thread process; rather, it is locked each time a thread calls one of the mission state estimate accessor or settor methods. For example, while the behavior set `filter` thread is executing a *select* transition on one task-point state estimate, the mission state estimate data is locked. If, prior to completion of the status transtion, the modeing broadcast thread attempts to read the mission state

estimate data to prepare a message, it will not acquire access to the data until the prior process has completed.

One advantage to the threaded implementation is that it allows each process to operate at a separate rate. Within the mission configuration, process rates (such as the rate of mission state estimate data broadcast) may be specified, and, as an example, it might be advantageous for the broadcast rate to be significantly slower than the message parsing rate.

5.4 User Interface

One advantage to using the Mission State Estimate method is illustrated by how the GUI is able to represent mission situational awareness. For DVR-type behaviors, our GUI is able to visually plot the projected linear path of an agent through its TSP-ordered service task-points. Instead of requiring each agent to either maintain an ordered list of task-points and task-point poses in its own agent state estimate or through explicit communication with the command and control agent, this information may be *inferred* through the database of task-point state estimates at any point within the communication network. Since any DVR task-point that is within the calculated TSP tour of a single agent will have status 'assigned' and and agentID identical to the ID of that agent, for each agent executing a DVR behavior, the GUI can simply plot a line through all task-points with status 'assigned' and the appropriate agentID, in order of increasing estimated cost. The order of the task-points in this plot will be identical to the TSP tour held in the respective agents internal behavior set state at the time those task-point state estimate costs were last updated by that agent.

The `CONTROL` agent provides a graphical, map-based display of situational awareness information. As we will use GUI screenshots to illustrate results of our demonstration missions in the following section, we will briefly describe our method of graphically representing information. Each individual agent and task-point are plotted at the position contained within the `CONTROL` agent's mission state estimate. Based on the agent or task-point's status, the state estimate is represented in different ways. TSE's with status `'todo'` are represented as hollow circles, colored to indicate to which task they belong. Assigned TSE's are filled, and completed TSE's fade into the background until reset, if at all. While assigned, the current cost is displayed. ASE's are illustrated by black discs with an edge color indicating the currently assigned behavior. Additional situational awareness information, such as the TSP tour describe above, may be implemented through the use of a user-defined "behavior monitor function" designed our GUI framework. The MotionArc application provides several basic such functions to plot TSP tours and Voronoi partitions.

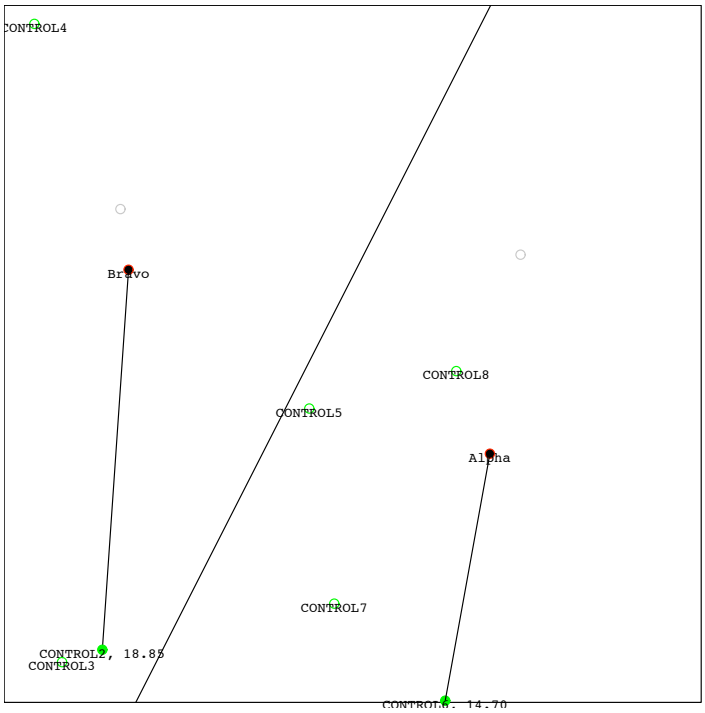
6 Mission Results and Final Considerations

To illustrate the effectiveness of the MotionArc system and verify that the Mission State Estimate protocols achieve a sufficient level of situational awareness, we now present several demonstrations MotionArc implementations for the dynamic traveling repairperson problem (DTRP) introduced earlier. This will also present examples of the level of awareness afforded by our chosen graphical user interface component in regards to agent and task-point state estimates as well as the internal behavior set states that may be *inferred* from mission state estimates. To create the *task specifications* associated with these missions, we use the standard format of dynamic vehicle routing tasks introduced in section 2. Task-point cost estimates are calculated as the estimate time-to-arrival at the task-point location (using the difference between poses of task-point and agent state estimates and the current measured agent speed) plus the required service time for that task-point and all prior task-points in a tour. We use the standard MSE task-point transition rules, with parameters given by

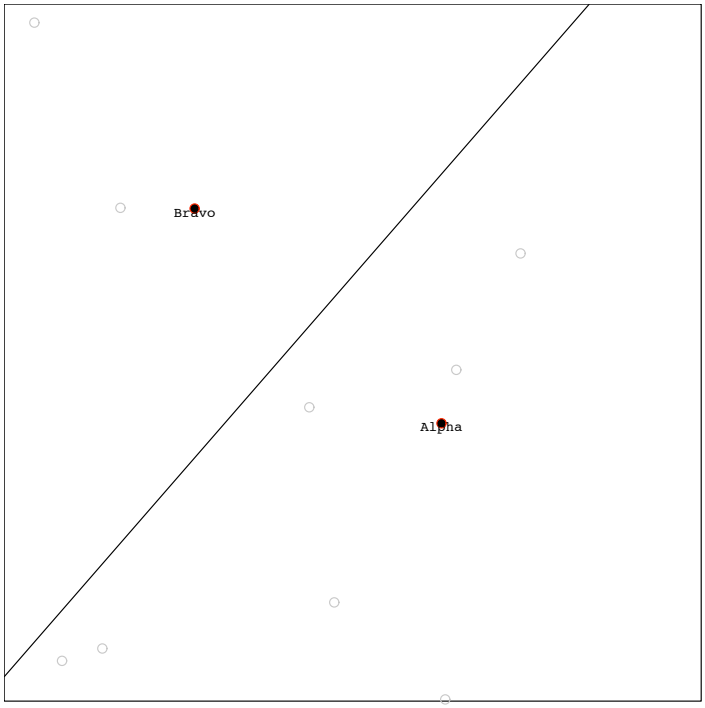
$$\texttt{fault\_timeout} = 15\text{s}, \quad \texttt{redo\_timeout} = \infty, \quad \epsilon_{complete} = 0.1$$

6.1 Demonstrating example task-point selection policies

We'll now discuss each of the three task-point selection policies introduced in section 3.2. Recall from our introduction of the MotionArc graphical user interface that `'todo'` task-points are represented as open circles and `'assigned'` task-points are represented as filled circles. Figure 15 shows two points in the execution of a DTRP task by two agents each using the *mSQM* task-point selection policy. In figure 15a, we see that each agent is assigned and moving towards



(a) Servicing demands



(b) Maintaining Voronoi partition

Figure 15: Two agents execute a DTRP using the *mSQM* selection policy.

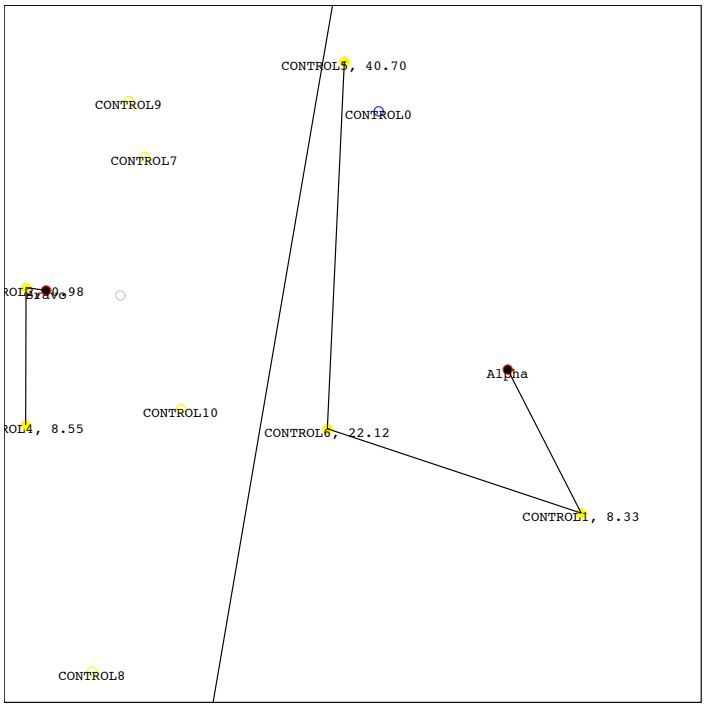
a single task-point in its respective Voronoi partition. Task-points with lower identification numbers represent task-points introduced to the system earlier; according to the *mSQM* policy, agents would be working towards the task-point within their region with the lowest identification number. This is the case for agent `Bravo` but not so for agent `Alpha`; that agent is assigned to task-point `CONTROL6` while there exists a task-point `CONTROL5` within its Voronoi region. However, this is most likely due to the proximity of `CONTROL5` to the edge of the Voronoi region; as Voronoi regions are calculated using the current agent positions at the last moment in time when the agent was not assigned, it is likely that `CONTROL5` was not within the Voronoi region at the point of the selection query. In part (b) of the figure, we see the steady-state locations of the agent when no task-points are assigned. As predicted when introducing the policies, The agents have taken up positions that generate a near-equitable Voronoi partition due to their desire to move towards the centroid of their own partitions.

Table 3 in the appendix presents a log file illustrating the evolution of the task-point state estimates for this mission over time. Confirming our observation from above, we see that agent `Alpha` executed the *select* transition on task-point `CONTROL6` at 13.79 seconds mission time, `CONTROL7` at 42.69 seconds mision time, and `CONTROL5` at 61.14 seconds mission time, indicating that in the moment just prior to each of the first two selections, `CONTROL5` was not within the Voronoi region for agent `Alpha`. This is a noticable drawback of our implementation of the *mSQM* policy; in rare cases when taskpoints are situated in-between agents, the Voronoi regions could waver back and forth and a task-point could remain un-selected for an extended period of time. In this case, the task-point in question goes 52.26 mission seconds from creation until selection, while the other two task-points selected in that time period waited a total of 3.02 and

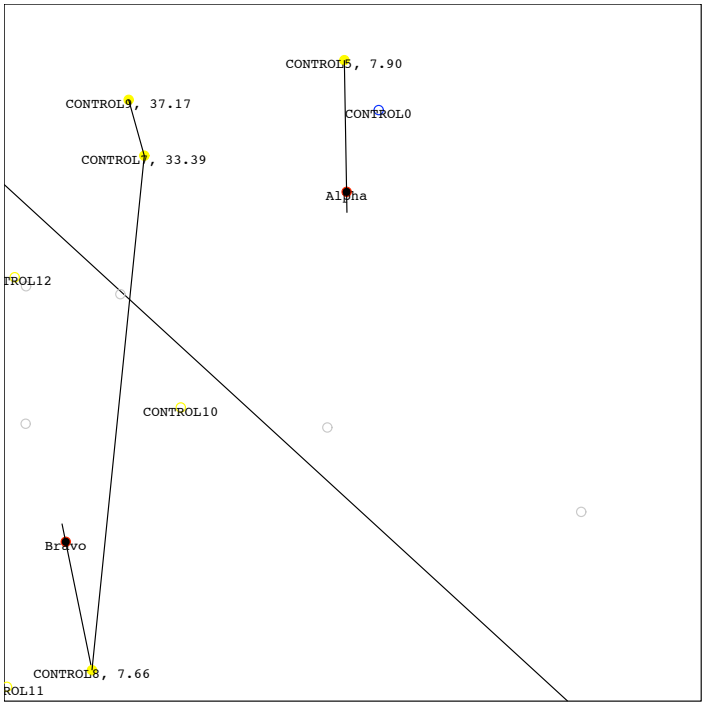
30.91 seconds, respectively. Future iterations of this policy sould address this drawback, perhaps by using a more rigid method of determining each agent's region of influence.

In figure 16a, we show a snapshot of a two-agent DTRP mission using the *partition-TSP* selection policy. In this case, tours are calculated through sets of 3 task-points. On the left hand side, we see that the agent will soon complete the tour (numbers labeled next to task-point names represent the estimated remaining cost until task-point completion) and should calculate a TSP tour through the next three task-points in its region. This is confirmed in figure 16b, a snapshot from a later moment in the mission, showing the next tour selected by the left hand agent. Note that, as the tour was calculated at a previous time when the Voronoi partition looked more like that of figure 16a, the tour now is not contained within the Voronoi polygon for the agent. This is expected, and when the current tour is completed, the next tour will be selected from task-points strictly within the region calculated at that point. Note that task-point `CONTROL10` is not selected – while in the region and seemingly close to the tour – because the tour is only calculated through the three next-earliest created task-points. Future implementations might modify any of the TSP policys to include quick "side trips" to any un-selected task-point within an arbitrary distance from the tour path, a method that is discussed throughout vehicle routing literature. The information inferred from this visualization is confirmed in Table 2 in the appendix.

Figure 17 shows tours executed by two agents using the *G/G/m-TSP* selection method. In this instance, our agents are actually running an identification task at each task-point where the task-point is determined to belong to one of two classes. Later, we will use additional agents running additional policys to



(a) Executing tour



(b) Tour crosses partition

Figure 16: Two agents execute a DTRP using the *partition-TSP* selection policy.

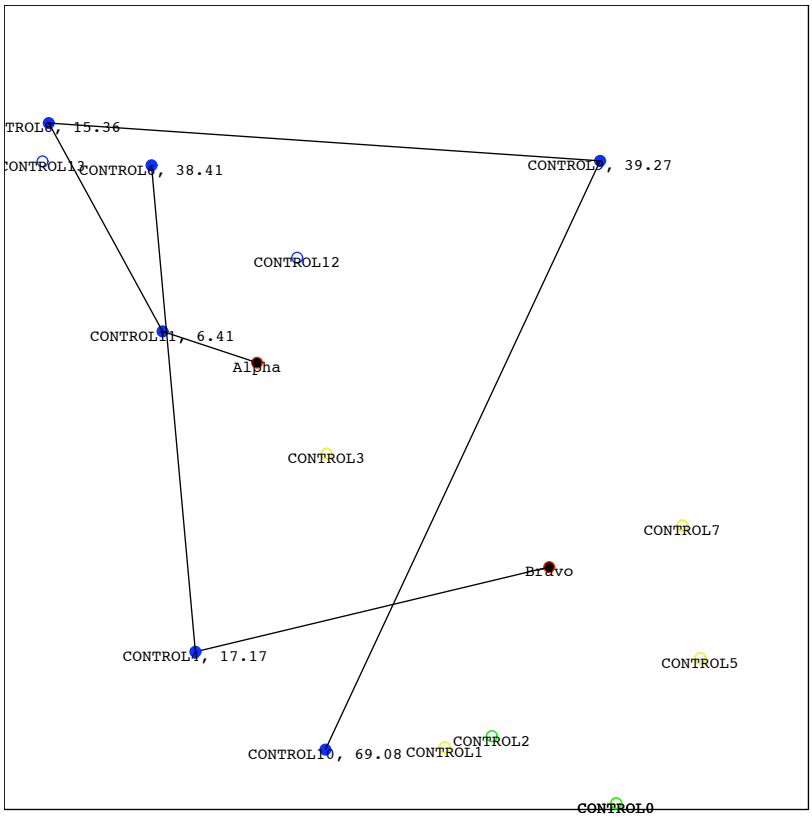
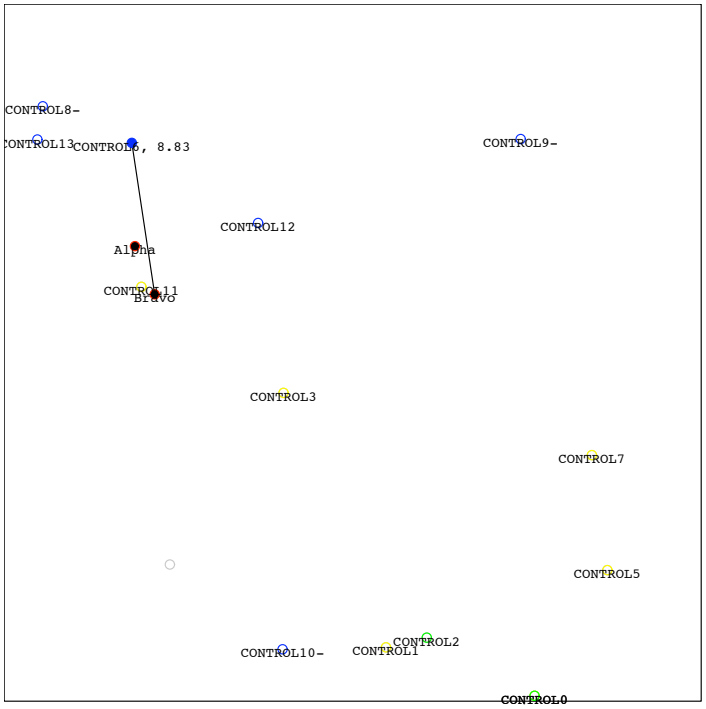


Figure 17: Two agents execute a DTRP using the *G/G/m-TSP* selection policy.

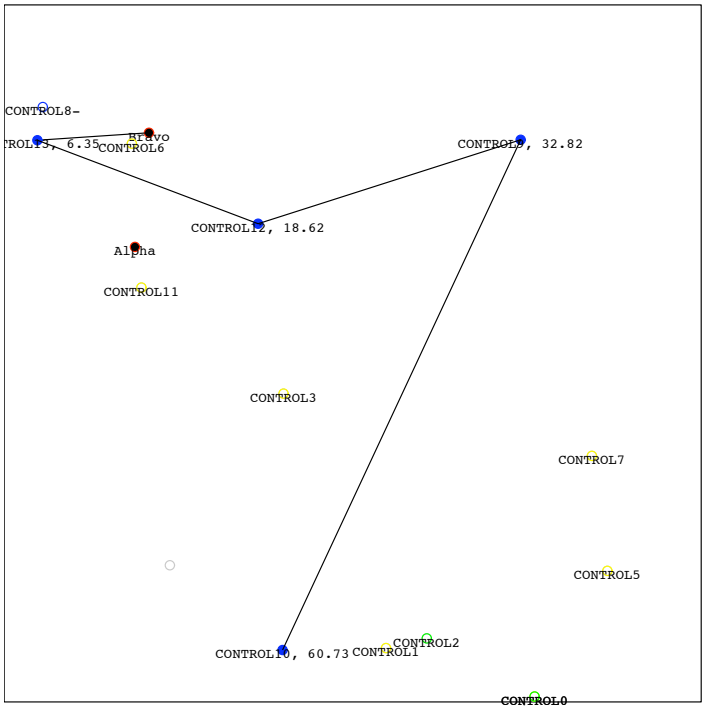
service these second-generation task-points. In the figure, we see two agents in the middle of executing a TSP tour route through a sequence of task-points. The lack of partitioning is shown clearly in this behavior, as tours cross multiple times. From table 4 and the image, we see that agent `Alpha` has just completed its first tour, this time through 4 task-points, and has just selected its next tour. Agent `Bravo` has two task-points remaining in its tour. Figure 18 illustrates the fault-tolerance capabilities arising from the mission state estimate protocols. After completing task-point `CONTROL11`, we caused agent `Alpha` to fail at a mission time of approximately 189 seconds. According to the task-point transition rules, which in this case specify a task-point state estimate fault time-out of 15 seconds, the remaining task-points in the tour are recognized as having failed by agent `Bravo`. Table 4 reflects this as it indicates taskpoints `CONTROL8`, `CONTROL9`, and `CONTROL10` all faulted at 204 mission seconds. Figure 18 shows those task-points as having returned to status `'todo'` and no tour is plotted through them. Figure 18b shows that, after completing its current tour, agent `Bravo` incorporates the three failed task-points in its next calculated tour.

6.2 Preliminary results for a multi-task mission

To demonstrate the multi-task capabilities of the MotionArc system, we executed a mission plan consisting of both identification and a single class of service task with four agents. Agent `unit` is a real hardware robot functioning within the MotionArc framework. The robotic agent makes use of a low-level navigation Player/Stage driver based on the Smooth-Nearness-Diagram algorithm introduced in [11]. A summary of the task allocation for this mission is shown in table 1, and the task-point state estimate log is shown in the appendix in



(a) Fault detected in agent *Alpha*



(b) Agent *Bravo* selects a tour including the faulted tasks

Figure 18: An illustration of the detection of faulted task-points and the re-assignment of those task-points to another agent using the mission state estimate protocols during execution of a DTRP using the *G/G/m-TSP* selection policy.

AgentID	Type	Task	Selection Policy
`unit`	hardware	service-B	*partition-TSP*
`Alpha`	simulation	identify	*G/G/m-TSP*
`Bravo`	simulation	identify	*G/G/m-TSP*
`Charlie`	simulation	service-B	*partition-TSP*

Table 1: Task allocation for demonstration mission

table 5.

The primary goals of this demonstration mission were twofold: first, we desired to illustrate that real hardware agents fit within the MotionArc framework as easily as simulated agents, and second, that multi-task missions exhibit the same successful level of situational awareness as our demonstration missions involving a single task. From the log file, it is evident that this mission made strides towards acheiving both goals. During the early stages of this mission, the simulated robots were all located within a very small region of space; as a consequence of our chosen navigation method, the agents did not leave this area for a significant period of time. During that time, agents did select tasks apropriately, but the fact that there was little to no movement towards goal completion resulted in several fault transitions for many task-point instances. This behavior is exactly what we desire from the Mission State Estimate protocols; were there additional agents assigned to the same tasks but located in less obstructed areas, those agents would calculate better cost estimates and thus the most efficient allocation of task-points to agents would result. As another alternative, were we using our combinatorial task allocation method, as the observed rate of task completion decreased due to the faulted tasks, other agents might be re-allocated to alleviate this cost increase. The point of this demonstration was simply to show that the *fault*, *select*, and *complete* transitions functioned properly through a multi-task mission.

The behavior of the hardware unit can be best shown through several situational awareness visualization snapshots. In the early stages of the mission, prior to any `service-B` class task-points being detected, the agent waits for task-points, the result of its behavior being a near-equitible Voronoi partition between itself and agent *Charlie*, shown in figure 19. Once three `service-B` class

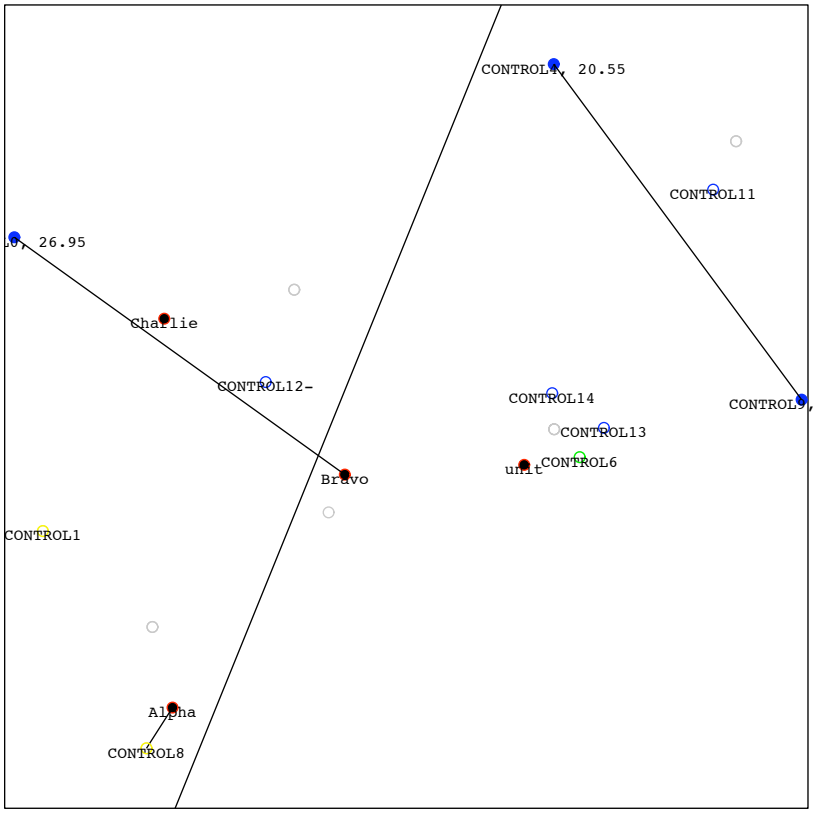


Figure 19: Hardware agent maintaining Voronoi partition with no task-points during multi-task mission

task-points have been detected within its Voronoi region, agent `unit` executes a TSP tour through those task-points, as perscribed by the DTRP behavior set utilising the *partition-TSP* selection policy. Agent `unit` can be seen moving toward the final task-point in its first tour in figure 20.

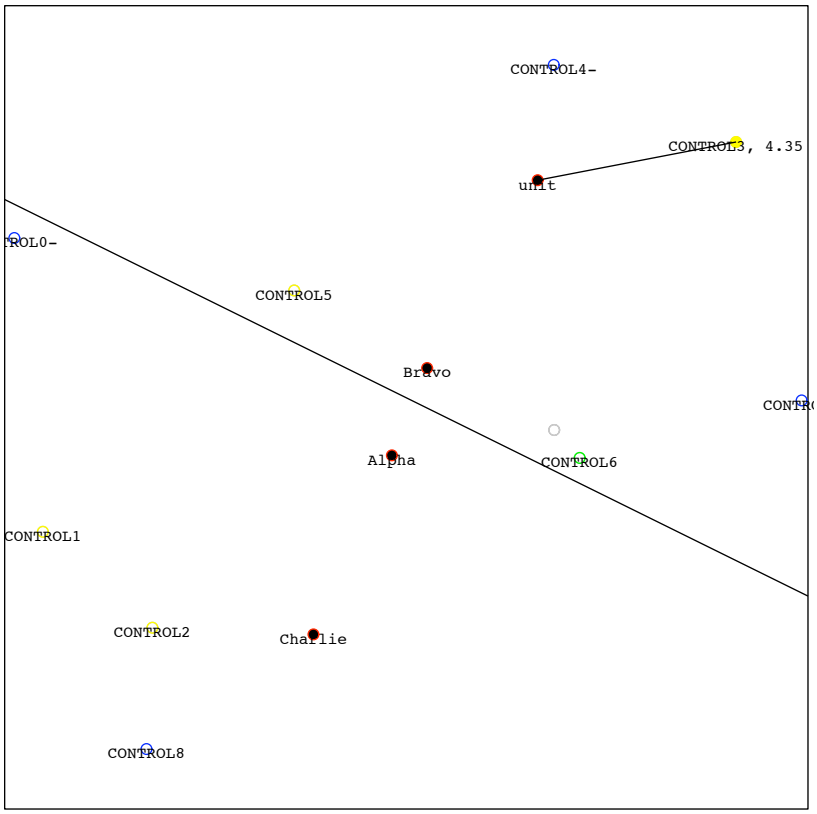


Figure 20: Hardware agent executing service task as part of a mutli-task mission.

6.3 Comparative analysis of task-allocation and mission modeling

It is clear – even from these simple demonstration missions – that the total-system-time behavior of missions executed on real and simulated hardware through the MotionArc system does not necessarily agree with the performace cost functions respective to each behavior set that were introduced as motivation for our combinatorial approach to task and behavior set allocation. These cost functions, developed in [4] and derivative works, are highly theoretical in nature, and are proven only to hold in the asymptotic regime with regards to the number of agents, number of task-points in a tour, and system loading factor. These considerations, along with other assumptions, indicate that these cost functions are not perfect indicators of the expected performance of a distributed, real-time hardware system such as ours.

From our mission logs, we see that when working with real or simulated hardware agents exhibiting highly complex dynamics and inter-agent comunication over a wireless network, there can be a significant slow-down in system performance. However, we feel that our results from section 4 show that using a combinatorial approach with these cost functions can result in a *reasonably good* initial estimate of the best assignment of agents to tasks. To acheive more accurate combinatorial results for a complex, real-hardware system, two approaches might be used. First, new cost functions could be developed analytically that take into account the complexities of real mobile robot dynamics and message transmission over a network. Second, and perhaps the more practical option, approximately-fit cost functions could be developed through extensive system experimentation and simulation. In either case, real-time system implementa-

tions will typically use a relatively small number of agents (as opposed to the high numbers required to validate the asymptotic nature of the cost functions describe above). With smaller, more manageable system sizes, it is less important to get the most optimal allocation and of greater consequence to quickly and cheaply find a *good enough* initial approximation.

After fully illustrating the preliminary design and conceptual background of the MotionArc system, let us draw some conclusions regarding the comparision of our system with two task-allocation architectures briefly introduced in our introduction. We feel that the MotionArc system attempts to draw from several systems that each excel in addressing a particular area of our experimental needs and incoorporates them in a highly effective package; in other words, that "the whole is greater than the sum of the parts." Recall the ALLIANCE approach of [16], a well-accepted method of performing inter-agent task allocation using the notion of a motivational behavior. The ALLIANCE method effectively encourages agents to switch from task to task in a fault-tolerant manner to ensure mission completeness. ALLIANCE does not specify how task-point instances are selected by an agent performing a behavior, so it is reasonable to assume that our behavior sets could be implemented as ALLIANCE behavior sets. However, as presented, ALLIANCE allows for only one agent executing each behavior; the simple fact that an agent is performing a behavior completely reduces all other agents' motivation to switch to that behavior. As one of our primary goals is to introduce tasks that directly benefit from having more than one agent assigned to them, the motivational behaviors of ALLIANCE would require adjustment to be an effective task-allocation method for MotionArc . Appropriately adapted, though, ALLIANCE motivational behaviors could provide an interesting alternative to the MotionArc combinatorial approach of

toward behavior allocation. Future efforts could attempt to compare results obtained using each method. The ALLIANCE approach is strictly limited to managing the assignemnt of tasks to behaviors and does not discuss maintenance of situational awareness information.

As much of MotionArc is modeled after the Mission State Estimate task-allocation architecture in [14], it is only natural to provide a similar comparision for this system. As mentioned previously, the strict Mission State Estimate protocols result in a fairly greedy approch to the allocation of task-point instances to agents performing each task; the ability to introduce behaviors that provide a better method of task-point instance allocation is an advantage of the MotionArc system. As with ALLIANCE, it is not clear whether the combinational approach of MotionArc is more or less accurate of efficient than the Mission State Estimate method of using tranition rules to switch between tasks; with further experimentation and simulation, a better comparision could be drawn. So far as mission situational awarenes information maintenance is concerned, the deviations from the Mission State Estimate protocols fit the specific needs of the types of DVR tasks we wish to implement. Mission State Estimates were similarly designed for a specific mission implementation; in general, both systems provide a good model by which to develop future information maintenance architectures.

6.4 Concluding remarks

This thesis has shown how the MotionArc system, theoretically, and the MotionArc application, in practice, are able to combine several useful approaches to task allocation and mission management on robotic network to develop an ex-

perimentaion and demonstration framework. Recall the primary motivation for developing the MotionArc system was the desire to implement several well-established algorithms in the context of a multi-task mission on real hardware. This thesis has demonstrated the successful implementation of several fundamental algorithms in exactly such a context. However, a robust implementation of significantly more complex tasks will most likely require a highly evolved and validated version of this preliminary system architecture.

In developing the preliminary architecture, many components were observed to lend themselves to further developement, several of which have already been indicated. In addition to continuing to adapt the architecture to handle more complex tasks, future efforts should include attempts to

- Use the communication components to establish a network connectivity model, and incorporate algorithms for network connectivity maintenance into the system architecture.

- Develop more accurate methods of establishing performance cost for a real-time implementation of theoretical DVR algorithms, either analytically or through expermientation

- Allow for messaging directly by behavior sets. It may not always be possible to reduce all information required by a behavior set into the mission state estimate, and it certainly may not be efficient to do so. Even with these simple tasks, the amount of information passed around the network caused significant slow-down at times. This could be alleviated by writing the communication components using a lower level language, such ats C++; however, it would be quite simple to allow behavior sets to access the message sending and queuing functions as well.

- Implement a distributed version of the allocation method, where individual agents are able to decide whether to switch to another behavior.

This preliminary architecture, by itself, is not a "start and forget" application; it is highly experimental and developmental in nature and its use for implementation of algorithms will require significant knowledge of its conceptual and practical details. Perhaps, then, the most significant contribution of the MotionArc system is not a specific program or set of algorithms, but rather a set of tools, mind-set, and way of thinking that will be beneficial in further attempts to implement more complex algorithms within the context of full scale multi-task mission execution. While methods and architectures such as ALLIANCE and the original Mission State Estimate approach may perform admirably and near optimally with regards to their specific goals, the tools and protocols introduced in MotionArc are definitely a step in the right direction towards establishing a global framework for mobile robotic mission execution.

A Mission Log Results

In this appendix we include log tables indicating the evolution of mission state estimates throughout several robotic missions. Some of these missions are associated with the images included in section 6. These log tables are presented from the point of view of a `CONTROL` agent; namely, the instance of the MotionArc control application running the mission. The log table indicates – for each named task-point instance – each task-point event, the associated timestamp, the agent executing the event, and the time at which the `CONTROL` agent was made aware of the event throught the mission state estimate broadcast and merging protocol.

Table 2: Mission state estimate log for two agents completing a DTRP task using the *partition-TSP* selection policy

taskpoint	task	event	agent	actual	local
CONTROL0	identify	created	CONTROL	29.82	29.82
CONTROL1	service-B	created	CONTROL	33.54	33.54
CONTROL1	service-B	selected	Alpha	64.48	63.83
CONTROL1	service-B	completed	Alpha	80.08	80.38
CONTROL10	service-B	created	CONTROL	64.69	64.69
CONTROL10	service-B	selected	Bravo	169.31	168.21
CONTROL10	service-B	completed	Bravo	198.81	199.20
CONTROL11	service-B	created	CONTROL	108.74	108.74
CONTROL11	service-B	selected	Bravo	167.15	166.42
CONTROL11	service-B	completed	Bravo	221.66	222.66
CONTROL12	service-B	created	CONTROL	112.57	112.57
CONTROL12	service-B	selected	Bravo	169.31	168.14
CONTROL12	service-B	completed	Bravo	185.07	184.77
CONTROL13	service-B	created	CONTROL	125.65	125.65
CONTROL13	service-B	selected	Alpha	136.53	135.91
CONTROL13	service-B	completed	Alpha	190.88	190.79
CONTROL14	service-B	created	CONTROL	125.71	125.71
CONTROL14	service-B	selected	Alpha	136.53	135.93
CONTROL14	service-B	completed	Alpha	151.99	151.96
CONTROL15	service-B	created	CONTROL	127.16	127.16
CONTROL15	service-B	selected	Alpha	136.53	135.95
CONTROL15	service-B	completed	Alpha	167.64	168.92
CONTROL16	service-B	created	CONTROL	157.73	157.74
CONTROL16	service-B	selected	Alpha	196.57	196.84
CONTROL17	service-B	created	CONTROL	159.25	159.25
CONTROL17	service-B	selected	Alpha	196.52	196.99
CONTROL17	service-B	completed	Alpha	214.52	215.59
CONTROL18	service-B	created	CONTROL	162.01	162.01

Table 2 (continued): Mission state estimate log for two agents completing a DTRP task using the *partition-Tsp* selection policy

CONTROL19	service-B	created	CONTROL	192.74	192.74
CONTROL19	service-B	selected	Alpha	196.53	196.14
CONTROL2	service-B	created	CONTROL	34.84	34.84
CONTROL2	service-B	selected	Bravo	46.26	45.38
CONTROL2	service-B	completed	Bravo	71.84	72.19
CONTROL20	service-B	created	CONTROL	194.43	194.43
CONTROL3	service-B	created	CONTROL	36.10	36.10
CONTROL3	service-B	selected	Bravo	46.26	45.87
CONTROL3	service-B	completed	Bravo	56.49	55.66
CONTROL4	service-B	created	CONTROL	42.37	42.37
CONTROL4	service-B	selected	Bravo	46.30	45.93
CONTROL4	service-B	completed	Bravo	108.52	109.27
CONTROL5	service-B	created	CONTROL	53.00	53.00
CONTROL5	service-B	selected	Alpha	64.64	63.82
CONTROL5	service-B	completed	Alpha	126.65	127.40
CONTROL6	service-B	created	CONTROL	54.68	54.68
CONTROL6	service-B	selected	Alpha	64.59	63.84
CONTROL6	service-B	completed	Alpha	101.86	102.95
CONTROL7	service-B	created	CONTROL	59.61	59.61
CONTROL7	service-B	selected	Bravo	109.69	109.16
CONTROL7	service-B	completed	Bravo	157.88	158.42
CONTROL8	service-B	created	CONTROL	61.00	61.00
CONTROL8	service-B	selected	Bravo	109.69	109.06
CONTROL8	service-B	completed	Bravo	126.43	127.31
CONTROL9	service-B	created	CONTROL	62.78	62.78
CONTROL9	service-B	selected	Bravo	109.69	109.23
CONTROL9	service-B	completed	Bravo	166.63	166.41

Table 3: Mission state estimate log for two agents completing a DTRP task using the *mSQM* selection policy

taskpoint	task	event	agent	actual	local
CONTROL0	service-A	created	CONTROL	0.00	0.00
CONTROL0	service-A	selected	Bravo	3.74	2.78
CONTROL0	service-A	completed	Bravo	19.72	21.49
CONTROL1	service-A	created	CONTROL	1.12	1.12
CONTROL1	service-A	selected	Alpha	3.54	2.91
CONTROL1	service-A	completed	Alpha	12.39	12.86
CONTROL2	service-A	created	CONTROL	3.87	3.87
CONTROL2	service-A	selected	Bravo	20.35	21.43
CONTROL2	service-A	completed	Bravo	49.14	50.39
CONTROL3	service-A	created	CONTROL	4.80	4.80
CONTROL3	service-A	selected	Bravo	50.84	50.45
CONTROL3	service-A	completed	Bravo	59.29	60.66
CONTROL4	service-A	created	CONTROL	6.48	6.48
CONTROL4	service-A	selected	Bravo	61.04	60.80
CONTROL4	service-A	completed	Bravo	120.51	121.43
CONTROL5	service-A	created	CONTROL	8.78	8.78
CONTROL5	service-A	selected	Alpha	61.14	60.66
CONTROL5	service-A	completed	Alpha	78.57	78.59
CONTROL6	service-A	created	CONTROL	9.77	9.77
CONTROL6	service-A	selected	Alpha	13.79	12.91
CONTROL6	service-A	completed	Alpha	42.56	41.91
CONTROL7	service-A	created	CONTROL	11.78	11.78
CONTROL7	service-A	selected	Alpha	42.69	41.96
CONTROL7	service-A	completed	Alpha	59.53	60.41
CONTROL8	service-A	created	CONTROL	13.70	13.70
CONTROL8	service-A	selected	Alpha	79.54	78.70
CONTROL8	service-A	completed	Alpha	93.45	93.25

Table 4: Mission state estimate log for two agents completing a DTRP task using the *G/G/m-TSP* selection policy

taskpoint	task	event	agent	actual	local
CONTROL0	identify	created	CONTROL	22.84	22.84
CONTROL0	identify	selected	Alpha	29.92	28.93
CONTROL0	service-A	created	Alpha	102.56	103.05
CONTROL0	service-B	created	Alpha	103.92	102.93
CONTROL0	identify	completed	Bravo	116.69	117.37
CONTROL1	identify	created	CONTROL	24.11	24.11
CONTROL1	identify	selected	Alpha	29.86	28.79
CONTROL1	service-B	created	Bravo	56.14	55.69
CONTROL1	identify	completed	Bravo	56.23	55.52
CONTROL10	identify	created	CONTROL	143.29	143.29
CONTROL10	identify	selected	Alpha	154.63	153.80
CONTROL10	identify	faulted	Bravo	204.90	205.23
CONTROL10	identify	selected	Bravo	239.35	238.39
CONTROL11	identify	created	CONTROL	147.09	147.09
CONTROL11	identify	selected	Alpha	154.63	153.97
CONTROL11	service-B	created	Alpha	177.89	178.79
CONTROL11	identify	completed	Alpha	177.94	178.48
CONTROL12	identify	created	CONTROL	167.96	167.97
CONTROL12	identify	selected	Bravo	239.35	238.58
CONTROL13	identify	created	CONTROL	169.79	169.79
CONTROL13	identify	selected	Bravo	239.35	238.76
CONTROL13	service-B	created	Bravo	292.07	292.75
CONTROL13	identify	completed	Bravo	292.25	293.39
CONTROL2	identify	created	CONTROL	26.16	26.16
CONTROL2	identify	selected	Alpha	29.86	28.68
CONTROL2	service-A	created	Bravo	62.39	61.78
CONTROL2	identify	completed	Bravo	62.69	61.85

Table 4 (continued): Mission state estimate log for two agents completing a DTRP task using the *G/G/m-TSP* selection policy

```
CONTROL3   identify   created    CONTROL     28.07  28.07
CONTROL3   identify   selected   Alpha       29.79  28.79
CONTROL3   service-B  created    Bravo       35.86  34.98
CONTROL3   identify   completed  Bravo       35.98  35.31
-------------------------
CONTROL4   identify   created    CONTROL     36.78  36.78
CONTROL4   identify   selected   Bravo      117.86 116.96
CONTROL4   identify   completed  Bravo      189.73 189.03
-------------------------
CONTROL5   identify   created    CONTROL     38.33  38.33
CONTROL5   identify   selected   Bravo      117.86 117.98
CONTROL5   service-B  created    Bravo      143.06 143.41
CONTROL5   identify   completed  Bravo      143.24 146.51
-------------------------
CONTROL6   identify   created    CONTROL     39.57  39.57
CONTROL6   identify   selected   Bravo      117.86 118.20
CONTROL6   service-B  created    Bravo      237.72 238.86
CONTROL6   identify   completed  Bravo      237.82 238.63
------------
CONTROL7   identify   created    CONTROL     41.72  41.72
CONTROL7   identify   selected   Bravo      117.86 117.90
CONTROL7   service-B  created    Bravo      157.41 157.87
CONTROL7   identify   completed  Bravo      157.52 158.01
-------------------------
CONTROL8   identify   created    CONTROL    140.67 140.67
CONTROL8   identify   selected   Alpha      154.63 154.18
CONTROL8   identify   faulted    Bravo      204.91 205.48
-------------------------
CONTROL9   identify   created    CONTROL    142.36 142.36
CONTROL9   identify   selected   Alpha      154.63 153.74
CONTROL9   identify   faulted    Bravo      204.90 205.34
CONTROL9   identify   selected   Bravo      239.35 238.41
```

Table 5: Mission state estimate log for one read hardware agent and three simulated agents completing a mission consiting of an identification task and one class of service task, using the *GGm-TSP* and *partition-TSP* selection policies, respectively.

taskpoint	task	event	agent	actual	local
CONTROL0	identify	created	CONTROL	37.50	37.50
CONTROL0	identify	selected	Alpha	47.17	46.22
CONTROL0	identify	faulted	Charlie	134.64	133.89
CONTROL0	identify	selected	Bravo	249.33	249.99
CONTROL0	identify	faulted	unit	303.37	303.58
CONTROL0	identify	selected	Bravo	308.84	307.83
CONTROL0	identify	faulted	CONTROL	345.28	345.28
CONTROL0	identify	selected	Bravo	356.67	358.66
CONTROL0	identify	completed	Bravo	376.98	378.99
CONTROL1	identify	created	CONTROL	39.86	39.86
CONTROL1	identify	selected	Alpha	47.17	46.15
CONTROL1	service-B	created	Alpha	95.12	96.10
CONTROL1	identify	completed	Alpha	95.28	95.89
CONTROL1	service-B	selected	Charlie	224.67	224.05
CONTROL1	service-B	faulted	CONTROL	256.28	256.28
CONTROL10	identify	created	CONTROL	230.00	230.00
CONTROL10	identify	selected	Alpha	235.78	235.30
CONTROL10	identify	completed	Alpha	323.64	324.12
CONTROL11	identify	created	CONTROL	231.76	231.76
CONTROL12	identify	created	CONTROL	232.38	232.38
CONTROL12	identify	selected	Bravo	247.27	246.71
CONTROL12	identify	faulted	CONTROL	295.21	295.21
CONTROL12	identify	selected	Bravo	302.74	302.60
CONTROL12	identify	faulted	CONTROL	330.45	330.45
CONTROL13	identify	created	CONTROL	268.68	268.68
CONTROL13	identify	selected	Alpha	380.38	381.94
CONTROL13	identify	faulted	CONTROL	395.58	395.58
CONTROL13	identify	selected	Alpha	399.86	399.86
CONTROL14	identify	created	CONTROL	269.00	269.00
CONTROL14	identify	selected	Alpha	388.61	389.36

Table 5 (continued): Mission state estimate log for one read hardware agent and three simulated agents completing a mission consiting of an identification task and one class of service tasks, using the *GGm-TSP* and *partition-TSP* selection policies, respectively.

```
-------------------------
CONTROL15  identify   created    CONTROL    352.13 352.13
CONTROL15  identify   selected   Bravo      377.22 378.58
CONTROL15  identify   completed  Bravo      390.52 394.95
CONTROL15  service-A  created    Alpha      394.14 398.48
-------------------------
CONTROL16  identify   created    CONTROL    352.53 352.53
-------------------------
CONTROL17  identify   created    CONTROL    405.03 405.03
-------------------------
CONTROL2   identify   created    CONTROL     41.17  41.17
CONTROL2   identify   selected   Alpha       47.17  46.15
CONTROL2   service-B  created    Alpha       70.41  70.80
CONTROL2   identify   completed  Alpha       70.52  70.81
CONTROL2   service-B  selected   Charlie    226.76 226.09
CONTROL2   service-B  completed  Charlie    235.10 236.60
-------------------------
CONTROL3   identify   created    CONTROL     43.62  43.62
CONTROL3   identify   selected   Alpha       47.17  46.34
CONTROL3   identify   faulted    unit        64.33  64.11
CONTROL3   identify   selected   Alpha       65.97  65.31
CONTROL3   identify   faulted    unit       105.83 105.44
CONTROL3   identify   selected   Alpha      107.26 106.23
CONTROL3   service-B  created    Alpha      148.88 149.59
CONTROL3   identify   completed  Alpha      148.97 149.45
CONTROL3   service-B  selected   unit       155.85 156.01
CONTROL3   service-B  completed  unit       223.98 222.53
-------------------------
CONTROL4   identify   created    CONTROL     65.97  65.97
CONTROL4   identify   selected   Bravo       90.48  89.08
CONTROL4   identify   faulted    unit       134.17 133.55
CONTROL4   identify   selected   Alpha      235.83 235.31
CONTROL4   identify   faulted    unit       269.54 269.72
CONTROL4   identify   selected   Alpha      270.90 271.57
CONTROL4   identify   faulted    CONTROL    349.94 349.94
CONTROL4   identify   selected   Alpha      351.39 355.00
CONTROL4   identify   faulted    unit       383.00 384.12
```

Table 5 (continued): Mission state estimate log for one read hardware agent and three simulated agents completing a mission consiting of an identification task and one class of service task, using the *GGm-TSP* and *partition-TSP* selection policies, respectively.

```
-------------------------
CONTROL5   identify   created    CONTROL     68.08  68.08
CONTROL5   identify   selected   Bravo       90.48  89.12
CONTROL5   identify   faulted    Charlie    105.50 104.44
CONTROL5   identify   selected   Bravo      106.78 105.58
CONTROL5   service-B  created    Bravo      109.83 109.72
CONTROL5   identify   completed  Bravo      109.93 109.68
CONTROL5   service-B  selected   unit       157.97 156.50
CONTROL5   service-B  faulted    Bravo      193.79 192.79
CONTROL5   service-B  selected   Charlie    226.76 226.35
CONTROL5   service-B  completed  Charlie    260.76 263.13
-------------------------
CONTROL6   identify   created    CONTROL     70.02  70.02
CONTROL6   identify   selected   Bravo       90.48  89.11
CONTROL6   service-B  created    Bravo       98.22  97.36
CONTROL6   identify   completed  Bravo       98.33  97.35
-------------------------
CONTROL7   identify   created    CONTROL     71.13  71.13
CONTROL7   identify   selected   Bravo       90.48  89.02
CONTROL7   service-B  created    Bravo       94.42  93.21
CONTROL7   identify   completed  Bravo       94.49  93.26
CONTROL7   service-B  selected   unit       155.85 156.05
CONTROL7   service-B  completed  unit       161.42 160.95
-------------------------
CONTROL8   identify   created    CONTROL    146.50 146.50
CONTROL8   identify   selected   Alpha      237.89 240.53
CONTROL8   service-B  created    Alpha      334.38 333.67
CONTROL8   identify   completed  Alpha      334.45 333.67
-------------------------
CONTROL9   identify   created    CONTROL    147.75 147.75
CONTROL9   identify   selected   Alpha      235.83 235.09
CONTROL9   identify   faulted    Bravo      292.10 291.82
CONTROL9   identify   selected   Alpha      293.56 293.08
CONTROL9   identify   faulted    CONTROL    370.59 370.59
```

References

[1] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[2] M. Ashdown and M. L. Cummings. Asymmetric synchronous collaboration within distributed teams. *Engineering Psychology and Cognitive Ergonomics*, 4562:245–255, 2007.

[3] G. Baliga, S. Graham, and P. R. Kumar. Middleware and abstractions in the convergence of control with communication and computation. In *IEEE Conf. on Decision and Control*, 2005.

[4] D. J. Bertsimas and G. J. van Ryzin. Stochastic and dynamic vehicle routing in the Euclidean plane with multiple capacitated vehicles. *Operations Research*, 41(1):60–76, 1993.

[5] S. D. Bopardikar, S. L. Smith, F. Bullo, and J. P. Hespanha. Dynamic vehicle routing for translating demands: Stability analysis and receding-horizon policies. *IEEE Transactions on Automatic Control*, March 2009. Submitted.

[6] K. M. Bretthauer and B. Shetty. The nonlinear resource allocation problem. *Operations Research*, 43(4):670–683, 1995.

[7] F. Bullo, J. Cortés, and S. Martínez. Distributed algorithms for robotic networks. In R. A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*. Springer, 2009. To appear.

[8] F. Bullo, J. Cortés, and S. Martínez. *Distributed Control of Robotic Networks*. Applied Mathematics Series. Princeton University Press, 2009. Available at http://www.coordinationbook.info.

[9] D. A. Castañón and C. Wu. Distributed algorithms for dynamic reassignment. In *IEEE Conf. on Decision and Control*, pages 13–18, Maui, HI, December 2003.

[10] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63 – 109, 1983.

[11] J. W. Durham and F. Bullo. Smooth nearness-diagram navigation. In *IEEE/RSJ Int. Conf. on Intelligent Robots & Systems*, pages 690–695, Nice, France, September 2008.

[12] B. Gerkey and contributors. The Player Project. `http://playerstage.sourceforge.net/`, September 2007. version 2.03.

[13] B. P. Gerkey and M. J. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, 23(9):939–954, 2004.

[14] M. F. Godwin, S. Spry, and J. K. Hedrick. Distributed collaboration with limited communication using mission state estimates. In *American Control Conference*, pages 2040–2046, Minneapolis, MN, June 2006.

[15] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006. Available at `http://planning.cs.uiuc.edu`.

[16] L. E. Parker. ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–40, 1998.

[17] M. Pavone, S. L. Smith, F. Bullo, and E. Frazzoli. Dynamic multi-vehicle routing with multiple classes of demands. In *American Control Conference*, St. Louis, MO, June 2009. To appear.

[18] S. L. Smith and F. Bullo. Monotonic target assignment for robotic networks. *IEEE Transactions on Automatic Control*, 54(10), 2009. (Submitted June 2007) to appear.

[19] I. Ulrich and J. Borenstein. VFH*: Local obstacle avoidance with look-ahead verification. In *IEEE Int. Conf. on Robotics and Automation*, pages 2505–2511, San Francisco, CA, April 2000.

[20] L. Xiao and S. Boyd. Optimal scaling of a gradient methods for distributed resource allocation. *Journal of Optimization Theory & Applications*, 129(3):469–488, 2006.