Distributed Sensor Network for Real Time Tracking [*]

Bryan Horling, Régis Vincent, Roger Mailler, Jiaying Shen,
Raphen Becker, Kyle Rawlins and Victor Lesser
University of Massachusetts
Dept. of Computer Sciences
Amherst, MA 01003
{bhorling,vincent,mailler,jyshen,raphen,rawlins,lesser}@cs.umass.edu

ABSTRACT

In this paper we describe our solution to a real-time distributed resource allocation application involving distributed situation assessment. The hardware configuration consists of a set of reconfigurable sensors at fixed locations, each having local processing and low-bandwidth communication capabilities with other sensor nodes. The objective is to track objects moving in the environment in real-time as best as possible, given uncertainty and constraints on sensor loads, communication, power consumption, action characteristics, and clock synchronization. Once the target is detected, the sensors must communicate and cooperate so that, within a given window of time, the data needed to triangulate the position of the target can be collected. Our solution to this problem decomposes the environment into a number of sectors, where individual sensor nodes in a sector are specialize dynamically to address different parts of the goal. We describe our solution to this problem in detail, including the high-level architecture and a number of the more interesting implementation challenges. Results and future direction are also covered.
(Video available at:
http://mas.cs.umass.edu/research/ants/ANTS.mov)

1. INTRODUCTION

Distributed vehicle monitoring as an example application of distributed situation assessment and more generally distributed resource allocation has been a problem studied extensively in the MAS community since its infancy [5][6][4]. To our knowledge, this work has been done in simulation, and not dealt with real-time issues of coordination and reconfigurable sensors, so that they are focused appropriately to track the desired object. This paper describes our work on a distributed vehicle monitoring application involving actual hardware. The hardware configuration consists of four radar sensors with associated processors (see figure 12) at fixed locations connected through a low-bandwidth radio frequency (RF) communication channel.

The goal of this application is to track one or more targets that are moving through the sensor environment. The radar sensor measurements consist of only amplitude and frequency values, so no one sensor has the ability to precisely determine the location of a target by itself. The sensors must therefore be organized and coordinated in a manner that permits their measurements to be used for triangulation. In the abstract, this situation is analogous to a distributed resource allocation problem, where the sensors represent resources which must be allocated to particular tasks at particular times, in order for the tasks to be effectively coordinated. Additional hurdles include a lack of reliable communication, the need to eventually scale to hundreds or thousands of sensor platforms, and the ability to reason within a real time, fault prone environment. In this paper, we will describe our solution to this problem.

The available sensor platforms have three scanning regions, each a 120 degree arc encircling the sensor (see figure 1A). Only one of these regions can be used to perform measurements at a time. The communication medium uses a low-speed, unreliable, radio-frequency (RF) system over eight separate channels. Messages cannot be both transmitted and received simultaneously regardless of channel assignment, and no two agents can transmit on a single channel at the same time without causing interference. The sensor platforms are capable of locally hosting one or more processes, which share a common CPU. Our solution populates each sensor platform with a single agent process, and we will use the terms sensor, agent and node interchangeably in this paper. Targets move through the environment in an arbitrary pattern as the scenario progresses. We assume that agents have basic knowledge of themselves (i.e. position, orientation, capabilities, etc.). Unless specified, all other information must be communicated by other agents over the RF medium.

This problem has several key elements that make it an interesting domain for exploration, including the need for strong coordination of activities, limited resources, a real-time environment, and varied sources of uncertainty.

[*]Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Materiel Command, USAF, under agreements number F30602-99-2-0525 and DOD DABT63-99-1-0004. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. This material is also based upon work supported by the National Science Foundation under Grant No. IIS-9812755. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory or the U.S. Government.

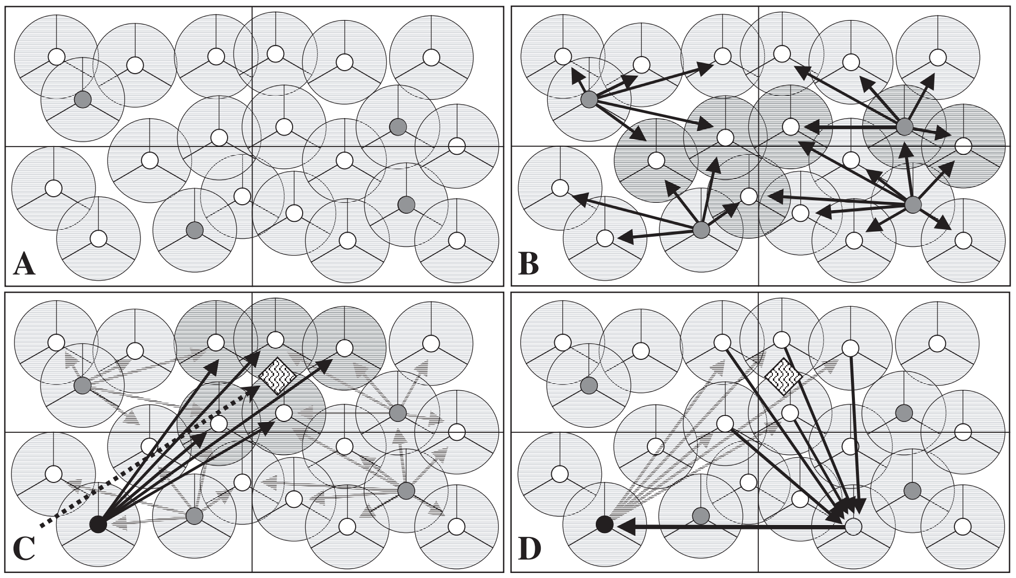


Figure 1: High-level architecture. A: sectorization of the environment, B: distribution of the scan schedule, C: negotiation over tracking measurements, and D: fusion of triangulation data.

The need to triangulate an target's position requires frequent, closely coordinated actions amongst the agents - ideally three or more sensors performing their measurements at the same time. In order to produce an accurate track, the sensors must therefore minimize the amount of time between measurements during triangulation, and maximize the number of triangulated positions. Ignoring resources, an optimal tracking solution would have all agents capable of tracking the target taking measurements at the same precise time as frequently as possible. Restrictive communication and computation, however, limits our ability to coordinate and implement such an aggressive strategy. Low communication bandwidth hinders complex coordination and negotiation, limited processor power prevents exhaustive planning and scheduling, and restricted sensor usage creates a tradeoff between discovering new targets and tracking existing ones.

Another interesting aspect of the environment is that it is real-time. A viable solution must consider issues such as the amount of time it takes to do meta-level activities, scheduling tasks with unknown or estimated execution durations and coordinating individual sensor platforms in the absence of a globally synchronizing clock.

Each of these factors contributes to a large degree of uncertainty. Noisy measurements, unreliable communications, varying hardware speeds, and sensor availability also make knowing a target's precise location and velocity very difficult. This in turn makes predicting and planning for future events more difficult, which subsequently increases usage of resources when unreliable data directs high level reasoning to incorrect conclusions and actions.

In the remainder of this paper, we will describe our solution which attempts to solve these complicated problems. We will describe how we have adapted our existing agent framework, JAF [3], by modifying our execution module to work in parallel, adding a partial order scheduler that takes advantage of parallelizable tasks and used TÆMS [1] to model meta-level tasks to handle the transition to a real-time environment. We will also discuss the agent organization, negotiation protocols, and high-level problem solving that we believe provide us with a robust, scalable and extendible solution to the problems of limited resources and uncertainty. Next, we present results from testing which has been done in both in simulation and on actual hardware. Finally, we conclude the paper by discussing future work and directions for the project.

2. HIGH-LEVEL ARCHITECTURE

As noted above, our overall objective is to track targets with the highest possible accuracy. At the same time, our solution must be scalable, robust in face of hardware failure, handle communication unreliability, and be able to conserve scarce resources, such as the battery that powers the sensor node. The high-level architecture described below attempts to address these issues.

The environment itself is divided into a series of sectors, each a non-overlapping, identically sized, rectangular portion of the available area, shown in figure 1A. The purpose of this division, as will be shown below, is to limit the interactions needed between sensors, an important element of our attempt to make the solution scalable. In this figure, sensors are represented as divided circles, where each 120 degree arc represents a direction the node can sense in. Although not represented, sensor nodes may also have heterogenous orientations and effective ranges. As agents come online, they must first determine which sectors they can affect. Because the environment itself is bounded, this can be trivially done by providing each agent the height and width of the sectors. The agents can then use this information, along with their known position and sensor radius, to determine which sectors they are capable of scanning in.

Within a given sector, agents may work concurrently on one or more of several high level goals: managing a sector, tracking a target, producing sensor data, and processing sensor data. Each sector will have a single sector manager, which serves as the locus of activity for a given sector. This manager generates and distribute plans (to the sensor data producers) needed to scan for new targets, provides directory services, and assigns target managers. Target managers are responsible for directing efforts to pinpoint and track known targets. Each known target in the environment will have a single track manager assigned to it, a role which can potentially move from one agent to another as the target moves. Agents producing sensor data perform the low level task of issuing commands to sensors and gathering the resulting data. Data processors take in sensor data and use it to generate target location and track information.

The scenario starts with agents determining what sectors they can affect, and which agents are serving as the managers for those sectors. Ideally, the sector managerial duty would be delegated and discovered dynamically at runtime, but due to the lack of a broadcast capability in the RF communication medium, we statically define and disburse this information a priori. In figure 1, managers are represented with shaded inner circles. Once an agent recognizes its manager(s), it sends each a description of its capabilities. This includes such things as the position, orientation, and range of the agent's sensor. The manager then has the task of using this data to organize the scanning schedule for its sector. The goal of the scan schedule is to use the sensors available to it to perform inexpensive, fast sensor sweeps of the area, in an effort to discover new targets. The manager formulates a schedule of where and when each sensor should scan, and negotiates with each agent over their respective responsibilities in that schedule (see figure 1B). The manager does not strictly assign these tasks - the agents have autonomy to themselves decide what action gets performed when. Given that sensors can potentially scan in multiple sectors, there is also the possibility that an agent may receive multiple, potentially conflicting requests for commitments. The agent itself is responsible for detecting and resolving these conflicts. If one receives conflicting requests for commitments, it can elect to delay or decommit as needed. Shaded sensors in the previous figure show agents receiving multiple scan schedule commitments.

Once the scan is in progress, individual sensors report any positive detections to the sector manager which assigned them the originating scanning task. Internally, the sector manager maintains a list of all local track managers, and location estimates for the targets they are tracking, which it uses to determine if the sensor detected a new target, or one which is already being tracked. If the target is new, the manager selects one of the agents in its sector, using locally available expected load knowledge, to be the track manager for that target. The assigned track manager (shown in figure 1C with a blackened inner circle) is responsible for organizing the tracking of the given target. To do this, it must first discover sensors capable of detecting the target, and then negotiate with members of that group to gather the necessary data. Discovery is done using the directory service provided by the sector managers. One or more queries are made asking for sensors which can scan in the area the target is predicted to occupy. For triangulation to be possible, three or more agents must scan the target at the same time, or within a relatively small window of time (within one second or so). The track manager must therefore determine when the scans should be performed, considering such things as the desired track fidelity and time needed to perform the measurement, and negotiate with the discovered agents to disseminate this goal (see figure 1C). As with scanning, conflicts can arise between the new task and existing commitments at the sensor, which the agent must resolve locally. Importance values placed on individual commitments allow for discrimination among them.

The data gathered from individual sensors is sent to another agent (possibly the track manager itself), responsible for fusing the data and extending the computed track (see figure 1D). If enough measurements are performed, and they occur within the same window of time, and the data values returned are of high enough quality, then they are used to triangulate what the position of the target was at that time. This data point is then added to the track, which itself is distributed back to the track manager to be used as a predictive tool when determining where the target is likely to be in the future. At this point the track manager must again decide which agents are needed and where they should scan, and the sequence of activities is repeated.

More details on the exact mechanisms and technologies used in this architecture can be found in the following sections.

3. IMPLEMENTED TECHNOLOGIES

3.1 Java Agent Framework

We use the Java Agent Framework (JAF) [3] as the foundation to our implemented solution. JAF is a component-oriented framework, similar to Sun's JavaBeans technology. The JAF framework consists of a number of generic components that can be used directly or subclassed, along with a set of guidelines specifying how to implement, integrate, and use new components. Components can interact in three different ways, each having different flexibility and efficiency characteristics: direct method invocation, through event (message) passing among components, or indirectly through shared data.

JAF was designed with extensibility and reusability in mind. The use of generic components, or derived components with similar APIs, allows for a plug-and-play type architecture where the designer can select those components they need without sacrificing compatibility with the remainder of the system. The designer can therefore pick and choose from the pre-written components, derive those that aren't quite what they need, and add new components for new technologies. For example, generic components exist to provide services for such things as communication, execution and directory services. In the environment presented in this paper, special facilities are needed for communication and execution. Derived versions of these two components were written, overriding such things as how messages are sent or how certain actions are performed. The communication component was also extended to provide a reliable messaging service, using sequence numbers, acknowledgements and retransmits to cope with the unreliable RF medium. These derived components were then inserted in place of their generic counterparts within the agent. The unmodified directory service component can still make use of the communication component, and if needed, communication can also use the directory services. In all, 17 components were used in the agents described in this paper: 10 were generic, 3 were derived, and 4 were new. This translates to roughly 20,000 lines of reused, domain independent code, and 8,000 lines of domain dependent code. The specific components which were used to create the agents are: Control, Log, State, Execute, Communicate, WindowManager, Observe, Sensor, ActionMonitor, PreprocessTaemsReader, DirectoryService, ResourceModeler, PartialOrderScheduler, PeriodicTaskController, ScanScheduler, Coordinate, and AntProblemSolver.

While layers of abstraction and encapsulation certainly are not new ideas, their incorporation into this architecture

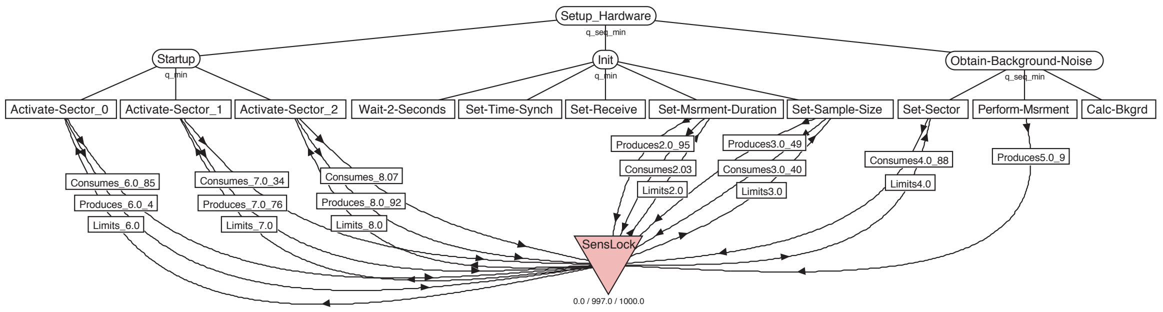


Figure 2: An abbreviated view of the sensor initialization TÆMS task structure.

is important because they both facilitate construction and motivate reusability and clean software design. A variety of components currently exist in JAF, providing services from logging and state maintenance to scheduling and problem solving.

3.2 TÆMS

TÆMS , the Task Analysis, Environmental Modeling and Simulation language, is used to quantitatively describe the alternative ways a goal can be achieved [1, 2]. A TÆMS task structure is essentially an annotated task decomposition tree. The highest level nodes in the tree, called task groups, represent goals that an agent may try to achieve. The goal of the structure shown in figure 2 is `Setup-Hardware`. Below a task group there will be a set of tasks and methods which describe how that task group may be performed, including sequencing information over subtasks, data flow relationships and mandatory versus optional tasks. Tasks represent sub-goals, which can be further decomposed in the same manner. `Setup-Hardware`, for instance, can be performed by completing `Startup`, `Init`, and `Obtain-Background-Noise`. Methods, on the other hand, are terminal, and represent the primitive actions an agent can perform. Methods are quantitatively described, in terms of their expected quality, cost and duration. `Activate-Sector_0`, then, would be described with its expected duration and quality, allowing the scheduling and planning processes to reason about the effects of selecting this method for execution. The quality accumulation functions (QAF) below a task describes how the quality of its subtasks is combined to calculate the task's quality. For example, the `q_min` QAF below `Init` specifies that the quality of `Init` will be the minimum quality of all its subtasks - so all the subtasks must be successfully performed for the `Init` task to succeed. Interactions between methods, tasks, and affected resources are also quantitatively described. The curved lines in figure 2 represent resource interactions, describing, for instance, the produces and consumes effects method `Set-Sample-Size` has on the resource `SensLock`, and how the level of `SensLock` can limit the performance of the method.

TÆMS structures are used by our agents to describe how particular goals may be achieved. Rather than hard coding, for instance, the task of initializing the sensor, we encode the various steps in a TÆMS structure similar to that shown in figure 2. This simplifies the process of evaluating the alternative pathways by allowing the designer to work at a higher level of abstraction, rather than be distracted by how it can be implemented in code. More importantly, it also provides a complete, quantitative view that can be reasoned about by planning, scheduling and execution processes. A given task structure begins its existence when it is created, read in from a library, or dynamically instantiated from a template at runtime. Planning elements are involved both in the generation of the structure, and then in the selection of the most appropriate sequence of methods from that structure which should be performed to achieve the goal. This sequence is then used by a scheduling process to determine the correct order of execution, with respect to such things as precedence constraints and resource usage. Finally, this schedule will be used by an execution process to perform the specified actions, the results of which are written back to the original task structure.

The schedules produced by individual TÆMS structures are the building blocks for an agent's overall schedule of execution. A valid schedule completely describing an agent's activities will allow it to correctly reason about and act upon the deadlines and constraints that it will encounter. Typically, however, schedules are only used to describe lower-level activity - in this domain, this encompasses sensor initialization, scanning and tracking activity, data fusion and the like. An important class of actions, so called meta-level activity, is missing from this list. Meta-level activities are the high-level functions which enable the lower-level activities. These include such things as scheduling, negotiation, communication, problem solving and planning. Without accounting for the time and computational resources these actions take, the schedule will be incomplete and susceptible to failure. In this study, we have begun accounting for these activities by including negotiation and coordination activities in our TÆMS task structures. From a scheduling and execution perspective, a negotiation sequence is just like any other action - it will have some expected duration and cost, a probability of success, and some level of required computational resources. By modeling negotiation sessions as a task structure, we are able to cleanly account for and schedule the time required to perform them, thus improving the accuracy of our schedules. In the future we will explore additional modeling of other meta-level activities, including planning and scheduling. We currently handle the time for these activities implicitly by adding slack time to each schedule. This is accomplished by reasoning with the maximum expected duration time for a given schedule, rather than the average time.

3.3 Real-Time Control

The nature of this project has forced us to take a close look all components that were part of our agent architecture and evaluate their capability to run in real-time. Originally our agents had just a single goal and sequential execution. If additional goals were requested, the agent had to merge the task structures together and then re-plan and reschedule all the actions. This solution was expensive and slow in dynamic environments, and thus inappropriate for our real-time needs. We needed a new agent control architecture that could easily incoporate new goals at any time, plan the methods required to achieve it and integrate the new methods in the current schedule.

While our new approach is not optimal, it does significantly reduces the planning and scheduling overhead. Prior work has addressed a similar problem by separating the previously integrated functionalities of planning and scheduling. We improve on this technique by producing partially-ordered schedules, rather than ones where methods are assigned to run at specific points in time. When a goal is received, the planning component is in charge of selecting the set of methods required to achieve the goal, without dealing with the resources needed by those methods. While the planner does have to generate a plan compatible with criteria and constraints given by the requester, it essentially works in an ideal world where all required resources are assumed to be available. The scheduler then takes this plan and generates a partial ordered schedule, where all precedence relationships and deadlines are explicitly represented. This partial ordered schedule differs from a linear one by only ordering methods that have relationships between them - no particular order is enforced between unrelated methods. The flexibility of this scheduling form is much more amenable to future goal and method integration, as well as facilitating rescheduling in the face of failure or exanticipated results. The scheduler completes by ensuring appropriate resources are available for methods in the schedule.

We use a resource modeler to keep track of the known resource uses. The partial ordered scheduler uses the resource modeler as a database to find times available for inserting new method in the current schedule. Once all methods of the new plan can be merged into the existing schedule without breaking any deadlines or constraints, the modified schedule is then published inside the agent as the new current schedule. The partial order scheduler is also responsible for propagating constraints inside the schedule, especially deadline constraints. If a goal has a deadline, this deadline is propagated to all methods involved to achieve this goal, so every method has their own execution window in correlation with the global goal deadline. This feature is used by other agent components, such as those dealing with negotiation, to compute the flexibility they have on method execution time. The execution window is maintained as new constraints arrive or are discovered, such as new goals or resource conflicts. If a constraint is broken, for instance by an event like execution taking longer than expected, the scheduler detects the constraint violation and delegates the problem to a conflict resolution module that will choose between the conflicting tasks.

Using the ordering constrainted described in the schedule, the execution component can directly determine which methods can be run concurrently. By parallelizing the execution, we reduce the total execution time, which effectively increases the agents overall work capacity. The gain in execution time, and resulting flexibility, is also used to address resource availability, as well as improving the likelihood the scheduler can accommodate real-time changes without breaking deadline constraints. The big advantage of the partial order scheduler is to be able to quickly shift methods' execution order at any point in time instead of doing costly re-planning [9]. In a real-time environment schedule adjustments are more frequent; by not imposing unnecessary ordering constraints on our agent's schedule the agent has a better chance of achieving the time, cost and quality criteria of its goal. We also attempt to reduce scheduling overhead by caching and reusing plans from similar task structures.

Flexibility in the schedule should propagate to the execution subsystem. In this agent control architecture, we augment our existing execution component by adding two new features. First, the execution module can use the partial order scheduler to get a list of all methods which can be currently executed. The partial order scheduler will check that all of a method's preconditions are true before authorizing the execution. The second extension allows the agent to pause currently executing methods and to resume work on it at a later time. This mechanism allows our agents to suspend working on a goal if a more important one arrives. Later, when the important goal is completed, it can resuming its work on the first goal. This mechanism is very similar to a UNIX kernel scheduling [8].

In the next section, we will describe how our negotiation module will assign the importance values used to resolve conflicting tasks.

3.4 Negotiation

In this environment, communication costs and time constraints make traditional complex negotiation difficult. On the other hand, some type of negotiation is still needed to effectively delegate tasks for tracking and scanning for the target. To address these requirements, we designed a satisficing negotiation protocol for periodic tasks.

In a *periodic task*, an instance of the task will be repeatedly performed over time (see Figure 3). The requesting agent will specify the task, how frequently it should be performed, and what the reference origin time is for the execution periods. The agent must then determine if it is able to repeatedly perform the task for each period past the origin time. Within each period, the actual execution of the task instance can be moved around, as long as it is done once during the specified time span. For example, if an agent commits to a tracking task, it may be expected to obtain tracking data in a particular sector within a window of one secord, occuring every five seconds relative to the epoch, until it is told to stop.

Each periodic task is assigned an *importance value*, to permit discrimination between conflicting tasks. As an example, when the sector manager creates a scan schedule, it will assign an importance value to each scan task needed by the schedule. The assigned value can depend on several factors, such as the location and expected quality of a particular agent's sensor, and the availability of other sensors in the area. For instance, agents along the border of an area are assigned higher importance because they are more likely to detect new, incoming targets. We will discuss importance values in greater detail later in this section.

Whenever possible, negotiated tasks are represented as periodic, rather than single-shot requests. Because of the repetitive nature of sensor data collection, the distributed vehicle monitoring domain is particularly amenable to this type of activity, but many other domains exhibit similar characteristics (e.g. assembly line scheduling, producer/consumer marketplaces). Tight communication constraints also drive the need for this more succinct negotiation style. Were we to start a negotiation or coordination session for each data collection action, the cost for communication would increase dramatically, potentially overloading the available

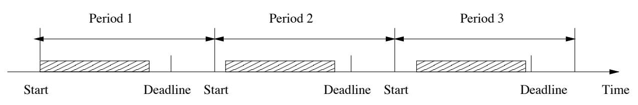


Figure 3: Periodic task example: Three periods of a task are shown, each action of the task can be shifted within a specified start time and deadline.

bandwidth.

One of the most interesting characteristics of a periodic task is that its lifetime is potentially unbounded. Regardless of our scheduling horizon, there is no way to know beforehand whether a periodic task can be successfully scheduled for every period. As a result, it is very difficult to reason about whether to commit to or refuse a proposal for a periodic task at the time of allocation, especially in a dynamic environment where new tasks can arrive at any time. A periodic task can be fit into the schedule for the first period does not mean that it can be always scheduled successfully. Similarly, a periodic task which cannot be scheduled for the next few periods might work fine for later periods. Negotiation protocols and scheduling techniques working with periodic tasks should be able to detect and handle these situations.

We lay out the continuous negotiation protocol for periodic tasks as follows. When a new periodic task is generated, the manager starts a new negotiation *session* by sending a proposal to the agent, specifying the task, its period length, and importance value. When receiving the proposal, the agent does a "temporary" scheduling of the periodic task for several periods. If this task can be placed in the schedule without conflict, the agent will commit to the periodic task. Otherwise, it will either refuse right away or counterpropose a different period for the task. The manager will then record the commitment, consider the counterproposal, or if a refusal is received, consider other alternatives. The agent who has committed to the task will then attempt to schedule for each period afterwards. If it fails to schedule some of the committed tasks for one periodic, the agent will compare the importance values of the conflicting tasks. The higher importance task will win and be scheduled. For a task of lower importance value, a negotiation *subsession* is initiated. The agent will decommit this period of the task from the manager, which is called *temporary decommitment.* If an agent has decommited from a task many times, it may realize that it is not suitable for this task any more and will *permanently decommit* from the manager. The manager may decide to *update* or *remove* the whole task based on information available. When a task is removed, the negotiation session is ended. The finite state machines for both the manager and the agent are illustrated in Figures 4 and 5. We call this protocol *continuous* since the negotiation session is subdivided into subsessions and continues until the whole task is revoked.

Each subsession of the negotiation can be either single-shot or multi-stage, depending on the context. For instance, having received a proposal for a periodic task from the manager, the agent can generate a counterproposal according to its schedule and send it back to the manager. The manager will then consider the counterproposal, and either accept it, reject it, or send a new counterproposal. Similarly, when an agent decides to temporarily decommit from a periodic task, the manager may consider this period of the task too important to ignore and temporarily modify the original proposal,

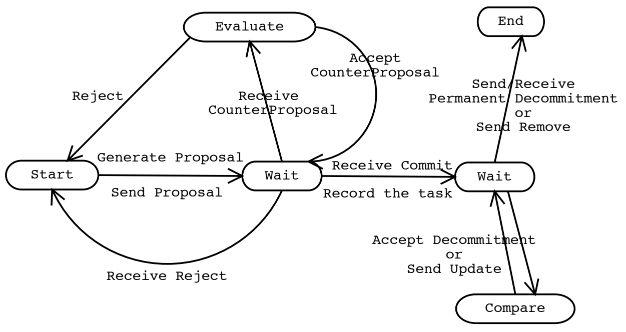


Figure 4: FSM for the manager

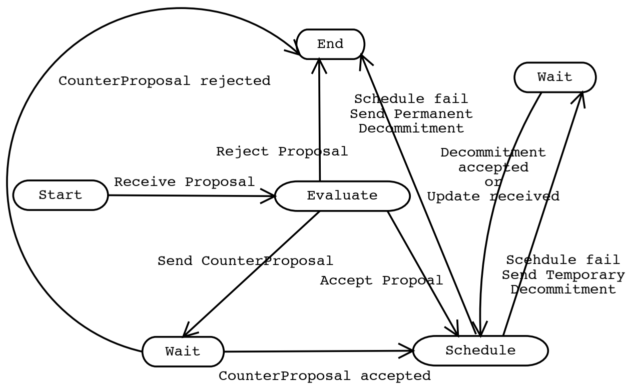


Figure 5: FSM for the agent

to convince the agent not to decommit from this period.

We apply this protocol to our problem. The track manager starts a negotiation session by sending a proposal to the sensor agent, specifying time, period, orientation and the importance value of the task. When the proposal is received, the sensor agent performs some initial reasoning and either commits to or refuses the task. A sensor agent who has committed to the task will schedule every period of the track task. When it fails to schedule a specific period, the agent compares the importance values of the conflicting tasks to determine which activity should be scheduled. It then uses a domain heuristic to determine how failure notification is handled. If the task of lower importance value is a track task, it will decommit this period of the task from the manager. If it is a scan task, the agent simply removes it from the schedule and does not report to the manager (implicit decommittal). The sensor agent sends the data back to the target manager when the measurement is performed. The manager decides whether an agent can still see the target or not based on the data it receives. By incorporating other data, the manager can also predict where the target will be. If the target is moving from the sensor's currently assigned sector to another, the manager will send an update message to notify the agent of this change. If the agent can no longer see the target, the manager will tell it to remove the committed track task, and the negotiation session will end.

As mentioned previously, we have currently only implemented the single-shot version of the protocol. We are currently working on the multi-stage version, and will compare its performance with the single-shot version in this environment. The main distinction between the two versions is negotiation over temporary decommitments. In multi-stage negotiation, one field of interest in the decommitment message is the *conflicting importance value*, which specifies the importance value of the conflicting task. When the track manager receives the temporary decommitment message, it first checks to see if there are sufficient agents satisfying their commitments. If this is true, the manager accepts the decommitment by doing nothing. If a deficiency is caused by the decomittal, it will check whether there are any other candidates for this period of the task. If there are, the manager will negotiate with the most appropriate one. If there are not, it will first recompute the importance value of the periodic task, since there might have been new changes in the environment since last time the importance value is updated. It will then compare this new value with the conflicting importance value contained in the decommitment message. If the recalculated value is larger, it will send the updated message to the agent with the new importance value, in hope that the agent will decommit from the conflicting task instead. Otherwise there is nothing the manager can do but to accept the decommitment.

The backbone of conflict resolution within this protocol is this notion of importance. The manager assignes it based on its needs, resource availability and quality. The agent then uses the value to choose from conflicting tasks. An important problem is to determine how these values are specified, and how they may change during the course of execution, to ensure that the most appropriate tasks are being performed at a given time.

The importance value of a scan task is initially assigned when the scan schedule is created, and later when the manager modifies the schedule because of changes of the environment (such as agents entering or leaving). The main consideration when assigning importance value to a scan task is the absolute location of the sensor. A new target can only enter an area from the edge, so the agents along the edge of an area are more likely to detect new targets. As a result, they are assigned higher importance. If the area covered by an agent in the proper orientation can be seen by other agents, a lower importance value is assigned, since the missing period is more likely to be covered up by other agents. Other metrics we will incorporate in the future include the historical quality of the tasks performed by an agent, and its expected workload.

For a track task, the situation is a little more complicated. There are two related but different characteristics: *appropriateness* and *importance*. Appropriateness is used to show how appropriate an agent is to perform a track task, and enable the manager to select potential agents to perform the task. A target close to a sensor will both produce better detection results, because of reduced noise, and be less likely to leave the sensor's range in a short time. Thus, the closer a sensor is to the expected location of the target, the more appropriate it is to track the target. The workload of the agent also has impact on its appropriateness. The lighter the workload is, the more appropriate the agent is. The workload is calculated based on the observed duties of the agent, including track or sector management, in addition to tracking or scanning tasks. We also take the predicted path of the target into account when deciding the appropriateness of an agent. If the target is moving toward the agent, a higher importance is assigned to it. If it is moving away from the agent, lower importance is assigned. The tracking history of the agent is considered as well. The more decommitments an agent has made, the less appropriate it will be.

On the other hand, importance indicates how important a track task is for the agents performing the task. For the track tasks, there are two points where importance value may be assigned (or reassigned) to an agent: when a new periodic track task is negotiated over, and when the manager decides to convince the agent to change its mind upon receiving a temporary decommitment from it. The latter case never appears for a scan task, as there are no temporary decommitments for scan tasks.

The manager assigns importance value to an appropriate agent based on three considerations: the number of candidates for this task, the number of consecutive data points having been missed for this task (i.e., the track history of the target), and the appropriateness of the agent to perform the task. Normally, the more candidates are available to perform a track task, the less important a task is for an individual agent. If this agent is unable to do it, it is likely that someone else would be able to, so the likelihood that a single decommitment would cause the failure of the overall objective is lessened. On the other hand, if only a minimum number of agents can perform a task, each is quite important. When a track task is initially generated and negotiated over, there is no notion of the track history for this target, so only the number of candidates is taken into account. During the negotiation over the temporary decommitments, the environment may be changed, due to decommittals from this task period. So, the importance values for individual agents should be reassigned to reflect the new number of working agents, which will affect how those commitments are respected.

When negotiating over a temporary decommitment, the track history of the target should be taken into account. If there are already several data points missing, more decommitments could cause a goal failure (e.g. loss of the target). In this situation, the manager will increase the importance values based on the number of missing consecutive data points, in an attempt to regain agents' attention.

There are still situations where reasoning process becomes complicated. Consider the situation when several agents temporarily decommit from a track task for the same period. Should the manager buffer the decommitment messages, recalculate the importance value together and make the decision? Or should it recalculate and decide individually at the time upon receiving a message? Similarly, one should be careful to avoid an "arms-race" among managers, each increasing their commitment importances in response to agent decommittals. We leave these issues as open questions.

In the future, we will extend this single shot protocol to multi-stage negotiation. Manager-to-manager negotiation will be added to increase efficiency in multi-linked scenarios. We will also design appropriate evaluation metrics to compare the protocols in environments possessing different communication characteristics.

3.5 Directory Services

The generic directory service component is capable of storing arbitrary textual data. Individual entries consist of one or more named fields, each of which will contain data. The directory itself possesses a set of one or more descriptions, which specify the type of data they are willing to accept. As a directory receives an entry to be added, it checks it against each of its descriptions, and if any match, the entry is added. Queries may be made to local or remote directories. The syntax for entry descriptions and queries is the same, consisting of a series of boolean, arithmetic or string expressions. The functionality of the directory itself is generic, and thus can serve as the supporting structure for a number of different directory paradigms, such as yellow pages, blackboards or brokers [7].

In our system, directory services are used in a yellow pages capacity, to centralize and disseminate information, thereby limiting the amount of communication needed to gather information. Individual agents post their capabilities to the sector manager's directory, which allows one to search for agents that can scan within a specified area. This sort of interaction is used to both construct the scanning schedule, and determine which agents are capable of sensing a target at a particular location. Agents also locally store descriptions of the sector managers in the environment, making it easy for them to find the managers of their own and neighboring sectors.

For example, a sector manager might have several directory entries for sensors capable of scanning in its sector. These would take the following form:

```
[E SA1 [Name->SA1] [Task->Scan] [R->20]
    [X->10] [Y->10] [O->60] [C->1]]
```

This contains such information as the sensor's name, task, radius, x and y position, orientation and communication channel. Later, when, for instance, a track manager needs to determine which nodes can scan in a given region, it might formulate the following query:

```
((((((20 + R) >= X) & ((10 - R) <= X)) &
((10 + R) >= Y)) & ((0 - R) <= Y)) & (Task == "Scan"))
```

This query matches entries who's x,y location falls within a given area, offset by the sensor's radius. In this case, it should return all sensors which are capable of scanning within the area (10,0),(20,10). If the region in question spanned multiple sectors, the track manager would assimilate the results from several queries to different sector managers.

In the future we can see the role of directory services being expanded. Directories may automatically check outdated entries, or send updated information to agents that have made prior queries. Directory services are also used locally at each agent in the system, to serve as a local cache of remote query responses.

3.6 Problem Solver

The problem solver is one of the few components in the agent that is strictly domain dependent. Its principle purpose is to coordinate the activities of the other system components, while reasoning about the environment at hand. For this particular problem, the problem solver we constructed is required to take on one or more roles, in the environment, including sector manager, track manager, and sensor.

To accomplish each of these roles, the problem solver uses a set of modified Finite State Machines (FSMs) that known as pulse actions (PA). A PA is defined as $PA = (A, C, \delta, a_0, F)$ where A is a set of actions, C is the set of input conditions, $\delta : A \times C \rightarrow A$ is the transition function, $a_0 \in A$ is an initial action, and $F \subseteq A$ is the set of final actions. For this problem, we restricted $C = \mathbf{N} \cup (M \times S)$ where $\mathbf{N}$ is the set of all nonnegative integers representing discrete time (bounded by MAX LONG), M is the set of all message types allowed in our communication protocol and S is a subset of the agents the multi-agent system. Actions are by no means implied to be simple states, but are in fact executable code meant to perform some discrete task or set of tasks. By using this model, we are able to break the execution of complex activities into smaller pieces, to specify a set of conditions which must be met before execution of an action continues, and to execute multiple actions in parallel. In the remainder of this section, we will employ two notational conveniences when talking about PAs. First, all PAs have an initiating condition, typically a message, which spawns a new instance of the PA, and will be mentioned in the description of the PA. This indicates that the initial action is always the action which follows the initial condition being met. Second, we reduce the size of the PAs by removing actions that have no executable code and combining conditions which lead to transitions from one state to the next. In this way, we specify a set of messages that must be received in order to make a transition from one action to the next.

3.6.1 *Initialization*

Before an agent can take on a role in the organization, it is important to have it initialize itself. The process of initialization includes preparing the communication channels, warming up the sensors, calculating the background noise of the area, and registering itself with its nearby sector manager(s). All of these tasks are handled by the TÆMS ask structure seen in figure 2. As you can see, we split up the major steps of the process into tasks, while individual steps are represented as methods.

The most notable task (and one that is not shown in the condensed version of the initialization task structure) is that of registering the sensor with the sector manager. This will be discussed in the next section.

3.6.2 *Sector Manager*

The sector manager plays a pivotal role in the organization of the agents in the system. It is responsible for managing all proximal tasks which happen within its area of effect. Because of this, the sector manager acts as a directory service, scan manager, and target resolver. Directory services is discussed in detail in section 3.5.

The scanning phase is used to discover new targets within the environment. By taking agent locations into account and using fast and cheap sensor mechanisms, high probability of target detection can be achieved while using relatively little power. The "scan schedule" consists of a periodic sequence of scanning actions organizing this effort. Scan scheduling is therfore done by evaluating where an agent lies within the environment and the positional relationship of the agent to other agents around it. The scan schedule is then set up by evaluating the trade-off between discovering a new target in a timely manner, power consumption, and potential conflicts with existing tasks. Currently, the scan scheduler employs a simple heuristic in making its schedule. It has each of the agents scan in a circular pattern, activating each sensor every 3.3 seconds. In practice this works quite well, although work is being done to make this process more efficient.

The next major role of the sector manager is that of target resolver. When a sensor agent detects a target, it reports a target detection message to the sector manager, with the observed amplitude and frequency measurements, as well as the sector and time the detection occurred. The sector manager checks two caches to see if the target is already known. The first cache keeps a list of targets that have not yet been

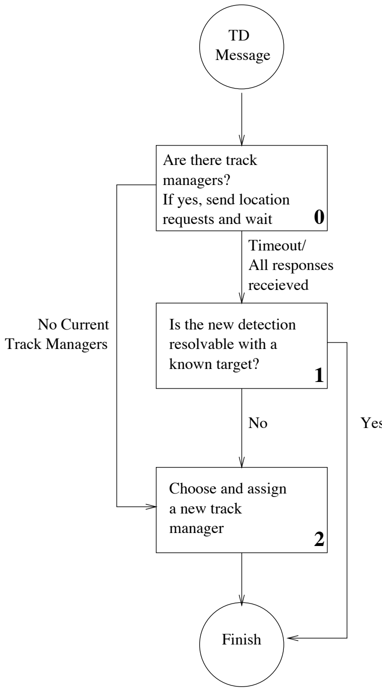

TD Message
Are there track managers? If yes, send location requests and wait 0
Timeout/ All responses receieved
No Current Track Managers
Is the new detection resolvable with a known target? 1
No
Yes
Choose and assign a new track manager 2
Finish

Figure 6: The target detection pulse action is used to determine if a new track manager should be assigned and to whom the new role belongs.

resolved. This permits mutiple concurrent detections to be resolved. The second cache is a time critical list of the known location of targets being tracked in the sector. This cache prevents two detections of the same target when they are temporally spaced far enough apart not to be resolved by the first cache. If the target is not resolved by these caches, then a Target Detect (TD) PA is activated.

Before we discuss the TD pulse action, the target resolution procedure should be covered. When a target is first detected, we only have a vague notion of where the target actually is, because it takes at least three coordinated measurements from three separate scanners to triangulate the position of a target. Lacking this data upon initial detection, we can only approximate an estimated region and bounding circle. This approximation is later refined as better measurements are made, although due to background and measurement noise a bounding circle is still needed to describe where the target is likely to be. The radius of the circle is used as a measure of uncertainty in our understanding of where the target is actually located. Each time multiple targets are resolved in the system, a check is made to determine if their bounding circles intersect. When we want to determine which agents are capable of sensing the target, the directory service is queried for sensors whose observable area intersects the bounding circle. So, when the target is first located, we assign the bounding circle that has a center and radius which provides maximal coverage of the sector it was discovered in, while minimizing overlap into adjacent sectors or outside the scanning range of the agent. It then becomes a trade-off though between falsely resolving two targets to be the same and not resolving to targets that should be the same.

The TD PA has three states and is responsible for determining through further scrutinization whether or not a new detection is actually a new target, and if it is, to assign a new

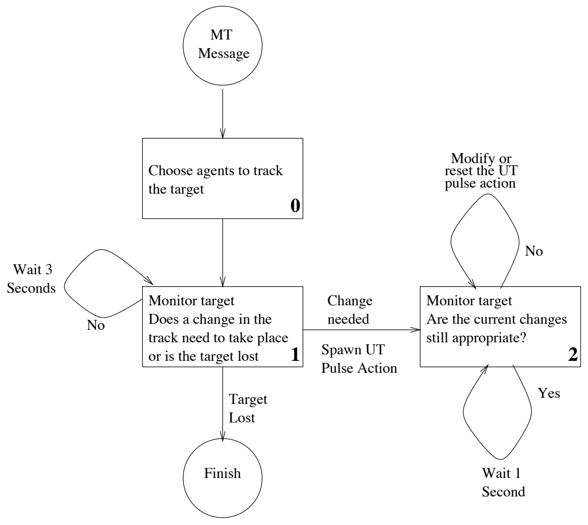

MT Message
Choose agents to track the target 0
Wait 3 Seconds
No
Monitor target Does a change in the track need to take place or is the target lost 1
Change needed
Spawn UT Pulse Action
Modify or reset the UT pulse action
No
Monitor target Are the current changes still appropriate? 2
Yes
Wait 1 Second
Target Lost
Finish

Figure 7: The track pulse action starts and monitors the track until the target is determined to be lost. It spawns update track pulse actions to make changes to the sensors employed for tracking.

track manager to monitor the target (see figure 6). Action 0 is responsible for determining if the target can be resolved by track managers. This is done by asking the track mangers that are currently tracking targets in the sector where each of their targets are currently located. Because the responses to the messages are not instantaneous, the PA is sent into a waiting state until responses are received from all of the track managers or a timeout has occured. When all of the messages come back, the machine goes into action 1. If there are no track managers then the machine moves into action 2. State 1 checks all of the replies from the track managers to see if the target is already known. Again, the new target is given a bounding circle that is compared to the bounding circle returned by the track manager. Track managers assign bounding circle based on the relative speed of the target, the direction of the target, and the last time a measurement of the target was successfully taken. If the target is resolved, the PA is terminated. If not, it immediately moves to action 2. The final action, action 2, actually chooses a track manager for the new target and sends out the notification message. Choosing a track manager is done by finding the agent within the sector which has the least number of task currently assigned to it. So, if an agent is the sector manager and is tracking a target, they get a score of 2. If the agent is just scanning, they get a score of 0. The agent with the lowest score (and therefore load) gets the job. Ties are broken randomly.

3.6.3 Track Manager

The track manager is responsible for only one thing, managing the agents that are needed to track a given target. Because of this, the track manager utilizes information from the sector manager (acting as directory services) and the estimated target location to reason about which sensors to employ in the track. The track manager is composed of two PAs: track and update track (figures 7 and 8). The track PA is initiated by a manage target message from the sector manager. This message includes information about the current estimated position of the target, the time that the target was at that location, and a list of agents that can track the target and their estimated loads. The track PA has three actions. The first, action 0, is responsible for determining which agents actually get assigned to track the targets as well as what sensors they should use to do it, and how often it should be done. The first question is answer by figuring out how important an agent might be to tracking the target. We do this with the following formula.

$$I = \alpha(\frac{R - D_i}{R}) + (1-\alpha)(\frac{\max_{a \in Agents} L_a - L_i}{\max_{a \in Agents} L_a})$$

Here, Agents is the set of agents in the system, L_i is the load of the agent i, R is the scan radius of a sensor node, D_i) is the distance of the target to agent i. In the future, we hope to employ velocity in the calculation so that if a target is moving toward a sensor, then that sensor has a higher importance (because it can track longer). Currently we have $\alpha = 0.75$. Once we have determined the importance of the agents, we choose the top four agents and start a negotiation session (see section 3.4). Periods are assigned to the track commitment based on the estimated duration of a track measurement with some slack time added to it. The period slack time is chosen to allow flexiblity. This prevents overloading of the agent and subsequent decommitments on the tracking task when the estimated measurement time is incorrect.

Starting the track is only a small portion of the work that needs to be done. The remaining two actions of the track PA monitor the status of the track. Details such as the target's location, direction, and speed, as well as the agents that have decommited (explicitly or implicitly) from their periodic tracking commitment are monitored closely to prevent the track manager from losing the target. Action 1 monitors the target under normal conditions. Contained within it, is the reasoning needed to determine when one or more of the agents involved in tracking the target needs to be changed or when the track should be terminated all together. This action is run every three seconds as long as a change to the tracking task is not being made. To determine if a target is lost, the results from the agents tracking are monitored. The criteria currently used is to check whether or not all of the agents that have reported results in the three second window have amplitude measurements below the background threshold. We additionally require that at least half of the agents must report this condition. This means with four agents tracking, if three reported results and none of the results exceed the background noise then the track is dropped and the target is considered lost.

Determining how and when to make changes to the agents involved in the track is the second part of this action. This is done by estimating the target's location in the future, and then determining if the agents that are currently tracking should continue or need to track using a different sensor. Optimally, it would be much easier to make changes based on the current state, but because it takes time to enact changes, the projected location must be used. Projections are generated based on the current location and velocity of the target. Using this, we can derive the new location at some time in the future. More succinctly, if we have X_t, Y_t, Vx_t, Vy_t we can calculate the future position of the target at time p as $X_p = Vx_t * (p-t) + X_t, Y_p = Vy_t * (p-t)+Y_t$. In the case where the target does not change velocity or where $(p-t) \approx 0$, we can get a very accurate position estimate. To help deal with larger values of $p-t$ or changes in velocity we create a bounding circle around the target and calculate its radius to be $r = \beta(p-t)(\sqrt(Vx_t^2 + Vy_t^2))$.

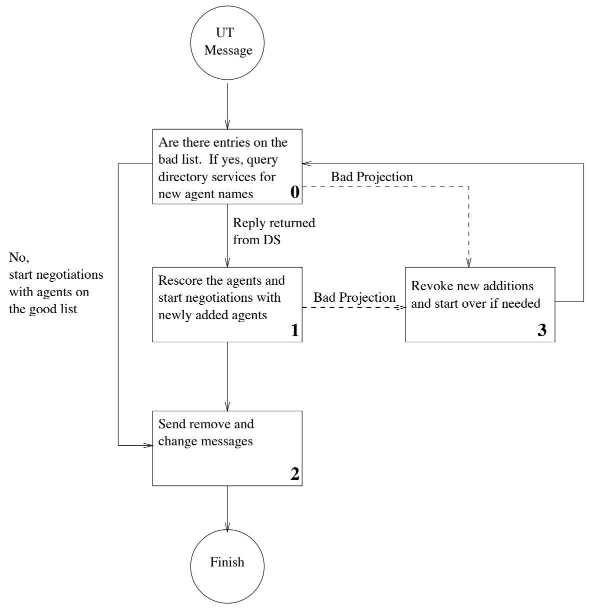

UT Message
Are there entries on the bad list. If yes, query directory services for new agent names
0
Bad Projection
Reply returned from DS
No, start negotiations with agents on the good list
Rescore the agents and start negotiations with newly added agents
1
Bad Projection
Revoke new additions and start over if needed
3
Send remove and change messages
2
Finish

Figure 8: The update track pulse action modifies the agents used to track a target. Note the dashed lines indicate the pulse action being externally modified by the track pulse action (see text).

The constant β is a tunable parameter that allows us to set the amount of uncertainty in the predictive capability of velocity on the target location. Higher values indicate less belief in the predictive capability of current velocity, lower values indicate more belief in the predictive capability of current velocity. In the future we may employ methods of changing β based on environmental conditions. Using this estimate, our knowledge of locations of the sensor that are being employed to track, and the location of sensors we have previously used (not currently tracking), we create three lists; the good list, bad list, and ugly list.

The good list contains the names of agents that are locally known that will be able to see the target in the future and are not currently involved in tracking the target. Agents are only added to this list if the number of agents currently tracking is less than the desired number. For example, if we want four agents to track and only have three tracking then we would add one new agent to the list. The bad list contains the names of all of the agents that will no longer be able to see the target if the projected position is correct. The ugly list includes the names of all agents who must change scanning direction in order to maintain the track if the projected position is correct. If after analyzing this information, the good, bad, or ugly list contains the name of an agent then the track PA goes to action 1 and spawns an update track PA. If the lists don't contain a name, then the track is presumed to be operating correctly. The track PA waits for another 3 seconds and checks the state of the track.

Track PA action 2 is a monitoring state that is used during the updating of a track. Because of this, it never spawns an update track PA. The purpose then of action 2 is to monitor the target and ensure that the changes being applied by the update track PA are in fact valid. If they are not, the track PA causes the update to cease and reverses any changes that have already taken place. The mechanisms for how it monitors the changes are simple, on each check, it generates the good, bad, and ugly lists and checks for changes between the new list and the one being used for the update. Depending on which action is being executed or waiting to be executed by the update track PA and the differences in the lists, several things can occur. For example, if an agent that was listed on the good list is no longer on the good list, then the update track action (if it hasn't started negotiation) is removed and the track PA transitions to action 1. Like action 1, action 2 transitions to itself on a fixed interval until the update is concluded or a change is made to the update that returns it to action 1.

The last of the PAs in the track manager is the update track PA. As the name implies, this PA is instantiated whenever the track PA has determined that a change needs to be made to the current tracking plan. The update track PA consists of four actions. The first action, action 0, checks to see if the bad list has entries. If it does, then the directory service is queried for the agents that will be able to see the target at the future location and the PA goes to action 1. If there are no entries on the bad list then it spawns negotiations for all agents on the good list and transitions to state 2 at the time that the changes should be enacted (the original projection time). So, if we were at time 2000 and projected three seconds into the future (3000), then the actual enact time would be at time 5000. The update track PA waits until the projection time to enact changes so that they can be stopped if the projection is incorrect and to allow for time to finish negotiation. Action 1 of this PA collects the results from the directory services query and rescores the importance levels of all agents involved in the track. This could be considered a more global reorganization of the tracking assets and is used to prevent multiple small changes in the future. After the agents are rescored then negotiation is started for all new agents being added and the PA transitions at the enact time to action 2. Action 2 is responsible for finishing all of the changes that need to be made to the track plan. It sends out remove messages to agents that should no longer track the target and change messages to agents that need to alter the sensor they are using for tracking. Change and remove messages are considered nearly instantaneous actions are therefore not started when negotiation for an add is done. This scheme is used to prevent the communication expense of revoking the action if the predicted target location is significantly wrong. The last action of this PA is the contingency action. This action can only be reached when the update track is waiting to transition to action 2 (enact the changes) and the track PA finds an error in the changes that are being made. This action is an emergency revocation of any adds that have started and a prevention of change and remove messages being sent out when they are no longer needed. Currently, all actions that have begun are stopped by a transition to action 3 and the update track PA is completely reset to action 1. In the future, a more managed contingency plan will be developed that will correct the execution of the update track PA and allow it to finish completion without restarting.

3.6.4 Sensor Agent

As a sensor, the agent is only responsible for performing scan and track tasks that have been committed to during a negotiation with the sector and track managers. The agent is responsible for returning results to the track manager, for a track task, or determining if a given scan is actually a positive detection. These tasks are all accomplished through the use of the real time control module discribed in section 3.3.

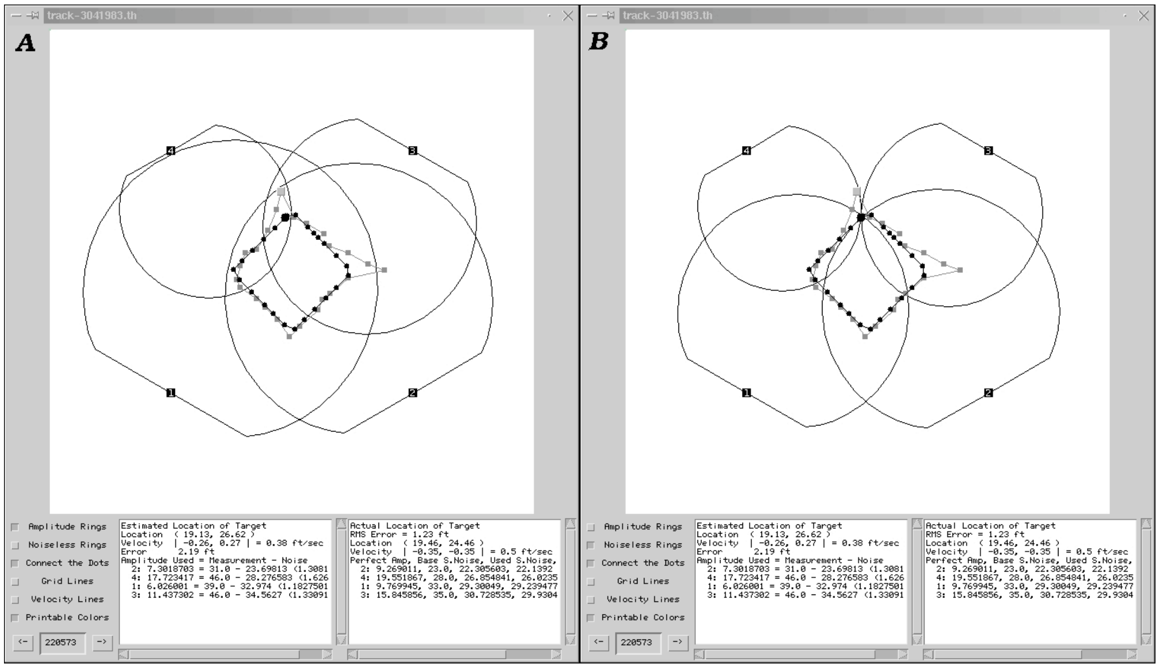
4. RESULTS

Because the hardware version of the environment is not directly available to us, we used a simulated environment, called RADSIM, to develop and test our implementation. The simulator was designed to closely emulate the actual hardware environment and provides a common software interface for our agents. We can configure RADSIM to scale to a large number of nodes (currently we have tested up to 32) and and arbitrary number of targets (we have tested up to 2). The target's path can be programmed in order to test specific aspects of our agents reasoning.

To aid in the evaluate of our results, we developed the visualization tools seen in figures 9 and 10. The views in figure 9 allow us to compare our measured tracks against the actual track taken by the target. The data fusion process interprets the amplitude values returned by the sensors as the ring shapes seen there, and triangulation is done by finding the strongest point of intersection among several of those rings. A substantial portion of the uncertainty in our solution is derived from the unavoidable noise values that affect amplitude measurements. As can be seen by comparing A and B in figure 9, the noise can dramatically shift the estimated location of the target, which will in turn affect both the generated track and future sensor allocations. Figure 10 shows the target location error over time, and how closely synchronized the agents measurements are in relation to one another. Each "dot" in the lower half of the timeline represents a measurement which was performed. An idealized run would have each column of dots be perfectly aligned, which would represent measurements that were completely synchronized. As you can see, our measurements are relatively aligned - each contained within a roughly 500ms window, which is sufficiently synchronized for our purposes. These tools allow us to easily and quickly evaluate changes that made to the underlying architecture.

To test how well our system performs, we ran 600 test runs of 8 minutes each while varying the reliability of communications. We chose to vary reliability of communication as a way of seeing how our system adapted to changing timing conditions and increased uncertainty in the actual time of a task will take to be completed, due to the potential need to retransmit information. We used four agents running on Pentium III PCs of varying speeds. The simulator ran on a separate machine to prevent an agent process from slowing the simulator down. The target's specified track can best be described as a diamond in the center of the environment (see figure 9A). The sensors were positioned in the environment such that they had overlapping coverage in the center of the environment.

A summary of the results of our testing is shown in figures 11A, 11B, and 11C. As expected, as communication loss increases, we observe an increased RMS error of the actual versus estimated track location, a measure of how different the two tracks are. As communication loss increases, the number of measurements successfully sent from the agents to the track manger decreases. We see the exponential increase in error because the track manager fails to achieve the three synchronized measurements needed to triangulate often enough. In fact, if you look at figure 11B, you can see that the average time between updates of the track position increases at a functionally equivalent rate. One positive aspect of the tests can be seen in figure 11C. This figure shows the average duration between the earliest and lat-



A: [✓ Amplitude Rings] Noiseless Rings [✓ Connect the Dots] Grid Lines Velocity Lines [✓ Printable Colors]

```
Estimated Location of Target
Location  ( 19.13, 26.62 )
Velocity  | -0.26, 0.27 | = 0.38 ft/sec
Error      2.19 ft
Amplitude Used = Measurement - Noise
  2: 7.3018703 = 31.0 - 23.69813 (1.3081
  4: 17.723417 = 46.0 - 28.276583 (1.626
  1: 6.026001 = 39.0 - 32.974 (1.1827501
  3: 11.437302 = 46.0 - 34.5627 (1.33091
```

```
Actual Location of Target
RMS Error = 1.23 ft
Location  ( 19.46, 24.46 )
Velocity  | -0.35, -0.35 | = 0.5 ft/sec
Perfect Amp, Base S.Noise, Used S.Noise,
  2: 9.269011, 23.0, 22.305603, 22.1392
  4: 19.551867, 28.0, 26.854841, 26.0235
  1: 9.769945, 33.0, 29.30049, 29.239477
  3: 15.845856, 35.0, 30.728535, 29.9304
```

B: Amplitude Rings [✓ Noiseless Rings] [✓ Connect the Dots] Grid Lines Velocity Lines [✓ Printable Colors]

```
Estimated Location of Target
Location  ( 19.13, 26.62 )
Velocity  | -0.26, 0.27 | = 0.38 ft/sec
Error      2.19 ft
Amplitude Used = Measurement - Noise
  2: 7.3018703 = 31.0 - 23.69813 (1.3081
  4: 17.723417 = 46.0 - 28.276583 (1.626
  1: 6.026001 = 39.0 - 32.974 (1.1827501
  3: 11.437302 = 46.0 - 34.5627 (1.33091
```

```
Actual Location of Target
RMS Error = 1.23 ft
Location  ( 19.46, 24.46 )
Velocity  | -0.35, -0.35 | = 0.5 ft/sec
Perfect Amp, Base S.Noise, Used S.Noise,
  2: 9.269011, 23.0, 22.305603, 22.1392
  4: 19.551867, 28.0, 26.854841, 26.0235
  1: 9.769945, 33.0, 29.30049, 29.239477
  3: 15.845856, 35.0, 30.728535, 29.9304
```

Figure 9: The track visualization tool gives information about estimated target location versus the actual target location. A shows the actual amplitude rings returned by the sensors, B shows the same rings without the effects of noise.

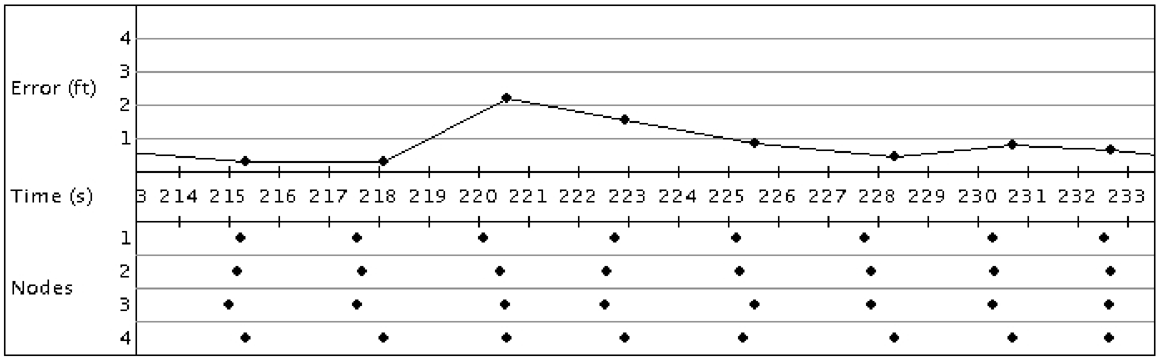


Figure 10: Target location error (top) and agent synchronization (bottom) over time.

est measurement used to do a triangulation which we call the update window. Note that as communication becomes more restrictive, the update window does not significantly increase. This is testament to the ability of periodic commitments to operate in a communication degrades environment.

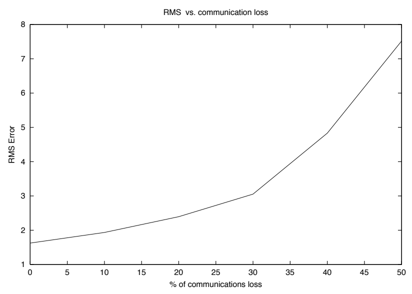
There are a number of interesting metrics that we have not included at this time. For example, we would like to make a comparison of the sensor utilization versus the number of the targets within the environment. In addition, we would like to explore how communications is utilized as the number of agents increases. Both of these metrics are central to the goal of the project and will be available when we scale up the system.

We also conducted testing of our system on hardware. For the hardware test, we used a configuration with four sensors (figure 12A) on the corners of a 10 by 12 foot rectangular area. Each of the platform had one of its sensors pointed directly toward the center of the area. For a target, a model railroad train with a copper radar reflector (used to increase the signal to noise ratio) was employed. The train was placed on an oval 9 by 6 foot track and operated at a speed of about one foot per second (See figure 12B). The sensors were connected to Pentium computers operating at various speed from 333MHz to 450MHz. The agents were started and after a one minute calibration time, the target was put in motion. We ran the target for approximately two minutes before concluding the test.

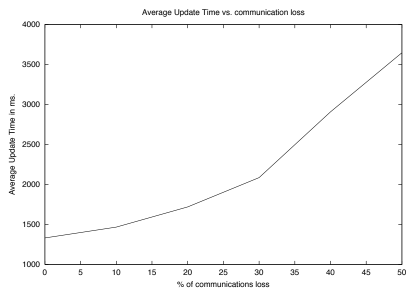
The results for these tests were mixed. We found that after some initial calibration of the expected run times for methods on the hardware, we were able to synchronize the agents almost as precisely as on the simulator. This showed to us that our architecture was capable of operating in a real time environment as the simulator predicted. Unfortunately, we were not able to track the target as accurately as we had on the simulator. We are currently investigating the reason for this but believe it may be simply due to a incorrect sensor calibration.

4.1 Visualization Tool

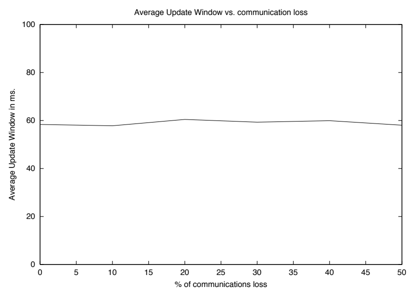
The visualization tool graphically shows how the agents



A) Average RMS Error for actual versus estimated target location.



B) Average time between subsequent triangulations.



C) Average synchronization error.

Figure 11: Performance results from 600 eight minute trials with varied communication reliability.

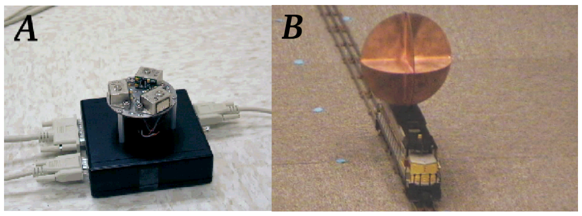


Figure 12: A: The hardware sensor platform. B: The mobile target used in hardware tests.

are performing and the results of their coordinated actions. Two different log files are kept while the system is running. One file records the true location of the target when it was detected by a sensor. The other file records the details about each measurement taken, and how it was used to calculate the location of the target.

When the tool is run, two graphs are displayed on the screen (Figure 9 & 10). The first display shows the track of the target as estimated by our agents and the actual track of the target if that information is available. One can step through each measured target location to get information like the X and Y coordinates, velocity, measurement values returned by the sensors, and its distance from the actual location of the target. There are also many overlays available that present additional information:

1. Amplitude rings - The amplitude value returned by the sensors represents an arc along which the target resides. This overlay shows what the sensor measured minus the background noise. See Figure 9A.
2. Perfect amplitude rings - These amplitude rings are what the sensors should have reported had there been no noise in their measurement. See Figure 9B.
3. Velocity - This overlay adds the velocity vector to each of the data points, both real and measured. It allows one to see the direction and speed of the target at each point.
4. Connect-the-dots - This draws a line between each point and the following point to help illustrate the path of the target.

Figure 10 shows error over time and synchronization of the agents. Both graphs share the X-axis, which is time in seconds. The upper graph plots the points where enough data had been collected to estimate the location of the target. Its Y-axis is the error in feet from the actual location of the target. The lower graph plots the measurement times for each agent. That allows one to quickly examine the synchronization achieved between agents. Using the two graphs together, one can start to correlate agent synchronization with error in target location. This graph is also very useful in lower-level debugging showing where problems occur, which can then be examined in the more detailed log files for more information.

5. FUTURE WORK

There are an abundance of areas for future exploration in this domain. We are currently exploring the effects induced by the scale of the system, first addressing scenarios with six and later 32 sensor nodes. In these scenarios we must evaluate both how the system scales in general, and how it handles the nuances that come with these larger areas (i.e. boundary cases for object detection, the potential for communication degradation over distance). We also will explore new, more sophisticated negotiation strategies, and investigate how the details of our organizational design should be implemented when dealing with larger roaming areas. Multiple targets will be added to the environment, increasing the probability of conflicting, high-importance commitments. In other scenarios the sensor platforms themselves will become mobile, sensors may fail or be added during runtime and the sensors may be jammed by adversaries in the environment.

6. CONCLUSIONS

In this paper we have described our solution to a real-time distributed tracking problem. The environment is first partitioned, reducing the level of potential interaction between agents. Within each sector, agents dynamically specialize to address scanning, tracking, or other goals. The agents must reason within a resource and communication-constrained environment, handling uncertainty in measurements, timing and coordination. We have successfully demonstrated our approach in both simulation and actual hardware.

A number of interesting technologies used by our agents or implemented for this problem were described. The JAF agent framework was used to implement the agents, allowing the reuse of a large code base, in addition to facilitating the construction itself. TÆMS was used to provide agents with domain problem solving knowledge, and to model the costs of meta-level activities. Our control architecture, including the DTC [10]planning component, partial-order scheduler, and resource modeler, enabled the agents to function effectively in a real-time environment. Negotiation, using periodic tasks capable of temporary decommital and updates, was used to disburse tasks to agents and synchronize their activity without significant use of bandwidth. Finally, the directory service component provided a simple way of keeping information up to date and getting that information to the agents that needed it.

Our results indicate that our architecture is robust enough to operate in both simulated and real world environments. They also indicate that using a continuous negotiation protocol for periodic commitments may in fact provide the necessary framework for handle the difficulties associated with a real-time distributed resource allocation problem in a communications degraded environment.

Acknowledgments

We would like to acknowledge the following organizations for their important contributions to this research. DARPA designed the problem space for the environment described in this paper, called the EW Challenge Problem, as part of the Autonomous Negotiating Teams (ANTs) program. Researchers at Rome Labs implemented the RADSIM simulator used to test our solution. Sanders, a Lockheed Martin company, designed and constructed the hardware sensor platforms, and implemented libraries needed to access that hardware and perform tracking and sensor fusion operations.

7. REFERENCES

[1] K. S. Decker and V. R. Lesser. Quantitative modeling of complex environments. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 2(4):215–234, Dec. 1993. Special issue on "Mathematical and Computational Models of Organizations: Models and Characteristics of Agent Behavior".

[2] B. Horling et al. The tæms white paper, 1999. http://mas.cs.umass.edu/res-earch/taems/white/.

[3] B. Horling and V. Lesser. A reusable component architecture for agent construction. Master's thesis, Department of Computer Science, University of Massachusetts, Amherst, 1998. Available as UMASS CS TR-98-30.

[4] V. Lesser and D. Corkill. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, 1983.

[5] V. R. Lesser and L. D. Erman. Distributed interpretation: A model and an experiment. *IEEE Transactions on Computers*, C-29(12):1144–1163, Dec. 1980.

[6] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transctions on Computers*, 29(12):1104–1113, 1980.

[7] K. Sycara, K. Decker, and M. Williamson. Middle-agents for the internet. In *Proceedings of IJCAI-97*, January 1997.

[8] A. S. Tannenbaum. *Distributed Operating Systems.* Prentice Hall, Saddle River, 1995.

[9] R. Vincent, B. Horling, V. Lesser, and T. Wagner. Implementing soft real-time agent control. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 355–362, 2001.

[10] T. Wagner, A. Garvey, and V. Lesser. Criteria-Directed Heuristic Task Scheduling. *International Journal of Approximate Reasoning, Special Issue on Scheduling*, 19(1-2):91–118, 1998. A version also available as UMASS CS TR-97-59.