SPT: 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
{vincent,bhorling,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 6) 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.

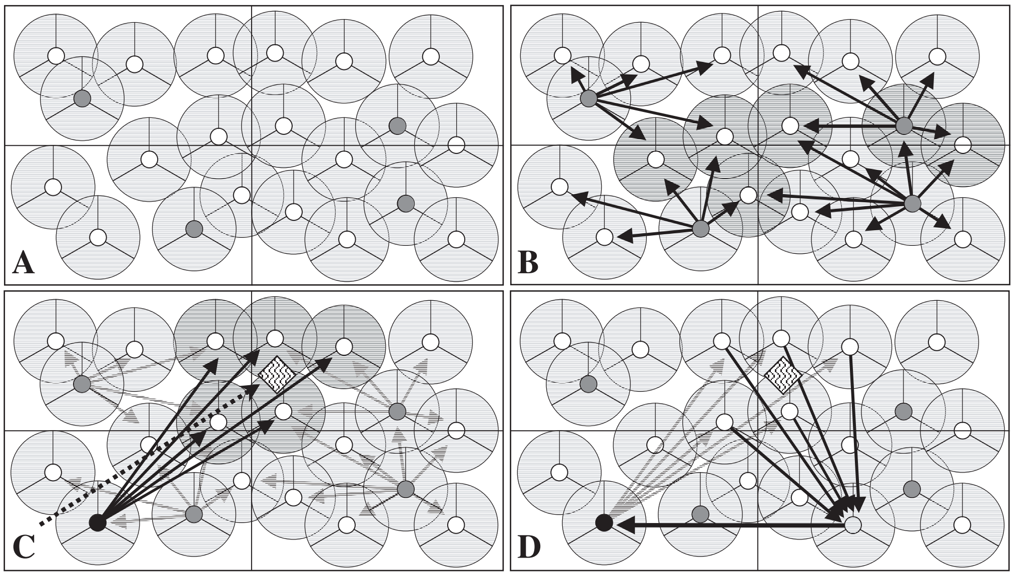


Figure 1: High-level architecture. A: sectorization of the environment, B: distribution of the scan schedule, C: negotiation over tracking measurements, and D: fusion of triangulation data.

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

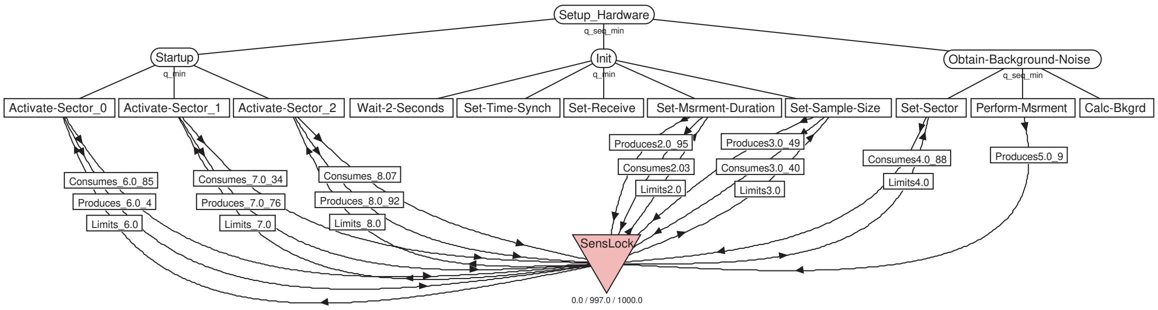


Figure 2: An abbreviated view of the sensor initialization TÆMS task structure.

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 very 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 clearly expensive and slow and not optimal for our real-time needs. We needed a new agent control architecture that could easily add new goals at any time, plan the methods required to achieve it and integrate the new methods in the current schedule. Though this is not optimal, it significantly reduces the planning and scheduling overhead. We have already accomplished this by separating the previously integrated functionalities of planning and scheduling. The planning component is in charge of selecting the set of methods required to achieve the goal, without dealing with the resources needed by the methods. The planner does have to generate a plan that is compatible with all the criteria given by the requester and by the problem solver (especially with respect to duration, cost and minimum quality). You can view this process as if the planner works only in an ideal world where it has all the resource it needs. The scheduler then takes this plan and generates a partial ordered schedule where all precedence relationships and deadlines are explicitly represented. A partial ordered schedule differs from a linear schedule by only ordering methods that have relationships between them and by not imposing any order to methods that has no relationships between them. Using this partially ordered schedule, the scheduler binds resources for all the methods and attempts to parallelize execution as much as it can.

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 slot available for inserting new method in the current schedule. Once all the methods of the new plan can be inserted in the current schedule without breaking any deadlines or constraints, the new 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 the 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 negotiation, to compute the flexibility they have on method execution time. The execution window is maintained as new constraints arrive, like 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.

By parallelizing the execution, we can reduce the total execution time which increases the agents overall work capacity. The gain in execution time (difference between the original deadline and the end of the parallelized schedule) is also used to accommodate any resource binding problem, or more importantly, allows the scheduler to accommodate real-time changes without breaking the 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 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 of all, 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 the preconditions of a method are true before authorizing the execution. The second extension allows the agent to pause any currently executing method at anytime and to resume it later. This very powerful mechanism allows our agent to suspend working on goal if a more important goal has arrived. Later, when the important job is done, it can resuming 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. Because of these characteristics, we designed a satisficing negotiation protocol for periodic tasks to solve the problem.

In a *periodic task*, an instance of the task will be repeatedly performed along the time line (see Figure 3). For every period, the actual scheduling of the task instance can be moved around, as long as it is done once during that period. For example, once an agent commits to a tracking task, it is expected to track the target at the specified sector periodically until the target moves out of the agent's range. Whenever possible, we represent tasks as periodic, based on the repetitive nature of the underlying commitments, and the tight communication constraints of the environment.

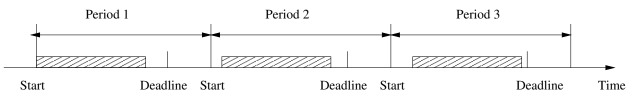


Figure 3: Periodic task example: Three periods of a task are shown, each action of the task can be shifted within a specified start time and deadline.

Every 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, based on the absolute location of the agent's sensor. For instance, the agents along the edge of an area are assigned higher importance because they are more likely to detect new targets. If the area covered by an agent can be seen by other agents, a lower importance value is assigned. For a tracking task, three factors influence the importance value: how many candidates there are for this task, how many consecutive data points have been missed for this task and the *appropriateness* of the agent to perform the task. The appropriateness of an agent to perform a task is based on the location of the sensor, the workload of the agent, the predicted future of the target, and the tracking history of the agent.

We lay out the continuous negotiation protocol for periodic tasks as follows. Upon generating a new periodic task, the manager starts a new negotiation *session* by sending a proposal to the agent, specifying the task, its periodicity, and importance value. When receiving the proposal, the agent does a "temporary" schedule of the periodic task for several periods. If this task can be placed in to the schedule without conflict, the agent will commit to the periodic task. Otherwise, it will either refuse outright or counter-propose a different period for the task. The manager will then record the commitment, consider the counterproposal, or if a refusal is received, consider other candidates. The agent who has committed to the task will attempt to schedule during each period. If it fails, the agent will compare the importance values of the conflicting tasks. The higher importance task will take precedence and be scheduled. For the task of lower importance value, a negotiation *subsession* is initiated. It will decommit this period of the whole task from the manager, which is called *temporary decommitment*. The manager may decide to *update* or *remove* the whole task based on the environment change. When a task is removed, the negotiation session is ended. We call this protocol *continuous* since the negotiation session is subdivided into subsessions and continues until the whole task is revoked.

We apply this protocol in our problem. The target 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 receiving the proposal, the sensor agent does some initial reasoning and either commits to or refuses the task. The sensor agent who has committed to the task will schedule every period of the track task. When it fails, the agent compares the importance values of the conflicting tasks. If the task of lower importance value is a track task, it will decommit this period of the whole 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. It can also predict where the target will be. When the target is moving from the current orientation of the sensor to some other orientation of the same sensor, the manager will send an update message to tell the agent about this change. When 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.

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].

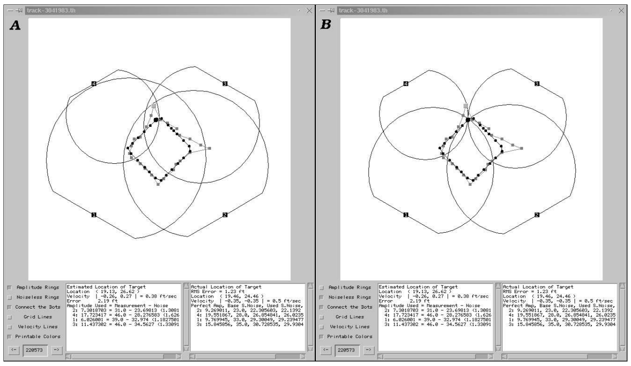


Figure 4: The track visualization tool gives information about estimated target location versus the actual target location. A shows the actual amplitude rings returned by the sensors, B shows the same rings without the effects of noise.

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.

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.

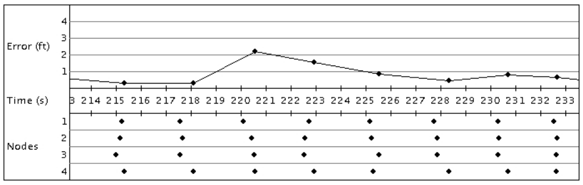


Figure 5: Target location error (top) and agent synchronization (bottom) over time.

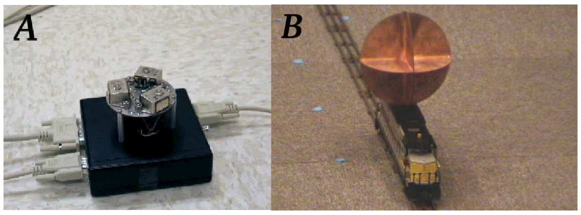


Figure 6: A: The hardware sensor platform. B: The mobile target used in hardware tests.

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 4 and 5. The views in figure 4 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 4, 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 5 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 4A). 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 7A, 7B, and 7C. 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 7B, 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 7C. This figure shows the average duration between the earliest and latest 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.

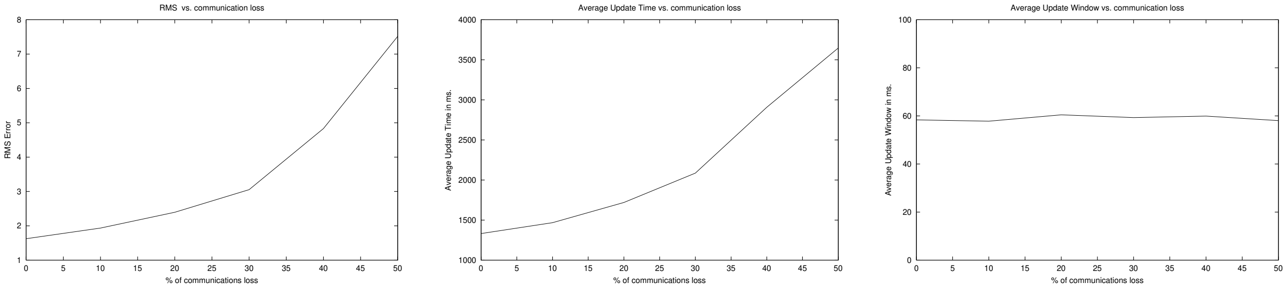
There are a number of interesting metrics that we have not included at this time. For example, we would like to make a comparison of the sensor utilization versus the number of the targets within the environment. In addition, we would like to explore how communications is utilized as the number of agents 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 6A) 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 6B). 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.

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



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

B) Average time between subsequent triangulations.

C) Average synchronization error.

Figure 7: Performance results from 600 eight minute trials with varied communication reliability.

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.

[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. Submitted to Autonomous Agents 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.

APPENDIX

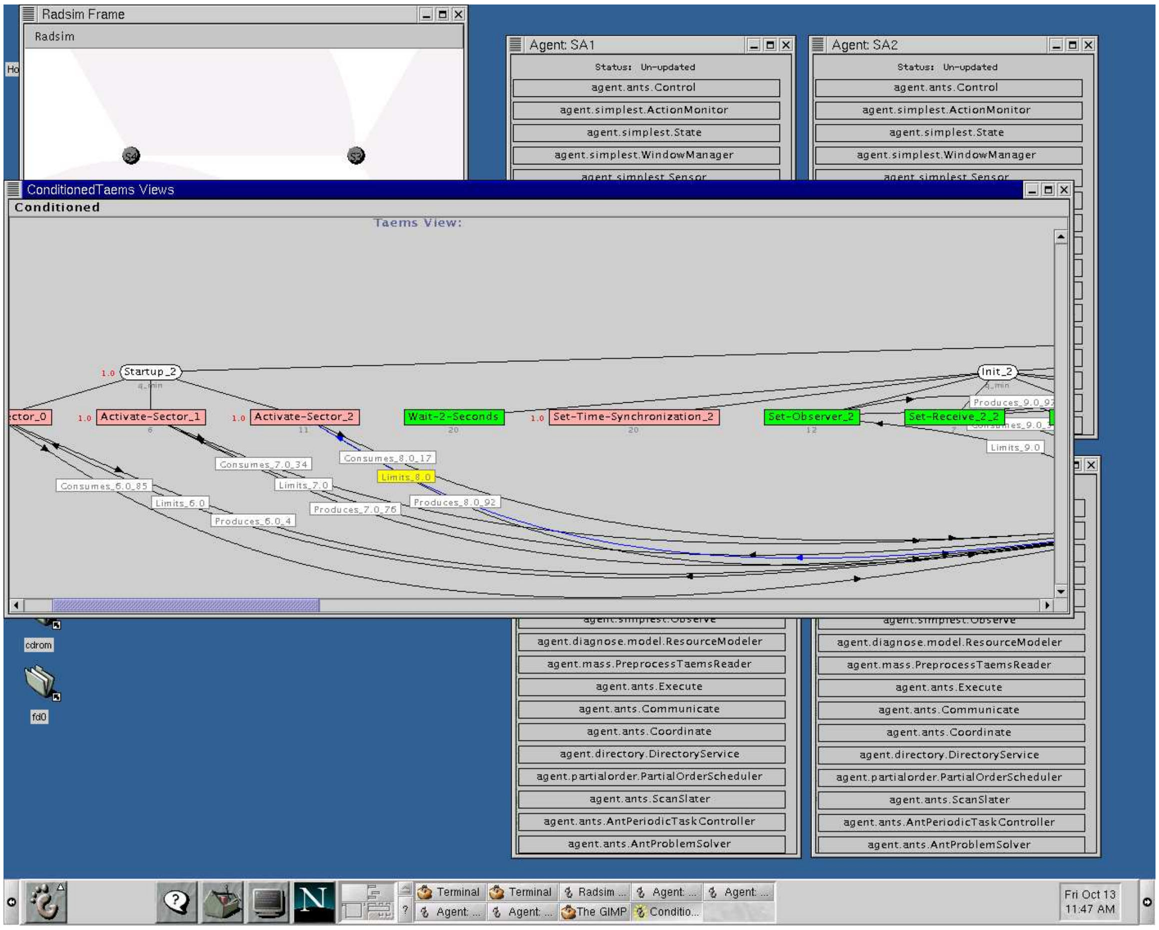


Figure 8: This screen shot shows the initialization of sensor SA2. The window in the foreground is a picture of the TÆMS task structure for initialization in execution. Methods can be seen as rectangles and tasks can be seen as ovals. The color of the method shows the status of its execution. Green (prints dark grey) rectangles indicate that the method is currently being executed and red (prints light grey) indicates a completed method. The small numbers under the methods indicates the execution order returned by the partial order scheduler. This example actually shows an occurrence of a partial order schedule slide. Because the Activate-Sector_2 method (item 11) locks the sensor resource and took longer than expected, the Set-Observer_2 method (item 12) was delayed. Notice that it is being executed at the same time that Wait-2-Seconds in being run which was item 20 in the execution order originally returned.

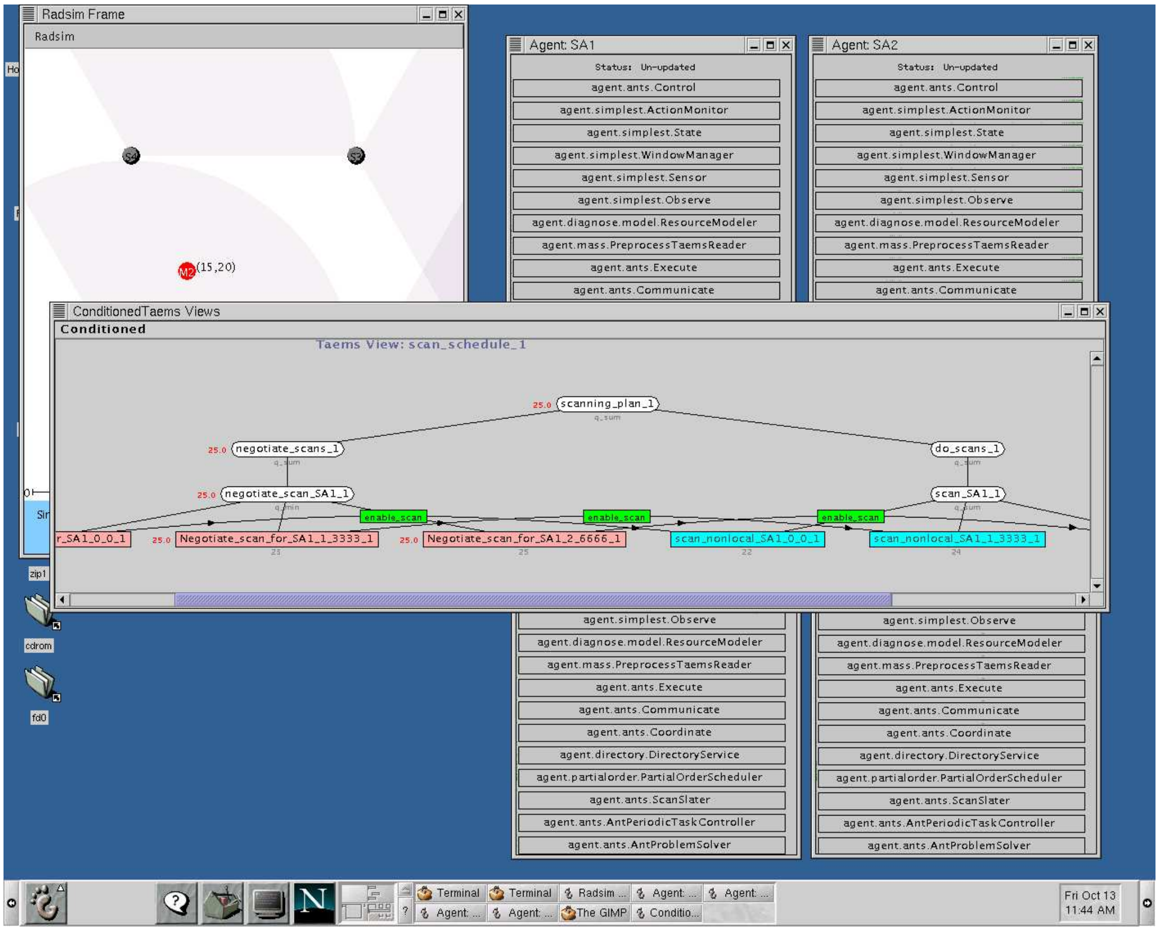


Figure 9: This is a screen shot of the simulator during a scan schedule negotiation. The sector manager(agent SA1) is negotiating with itself to perform three periodic scan tasks. Notice that negotiation for the commitments is represented, scheduled and executed as a standard method. Finishing negotiation enables the activity of scanning in the remote agent.

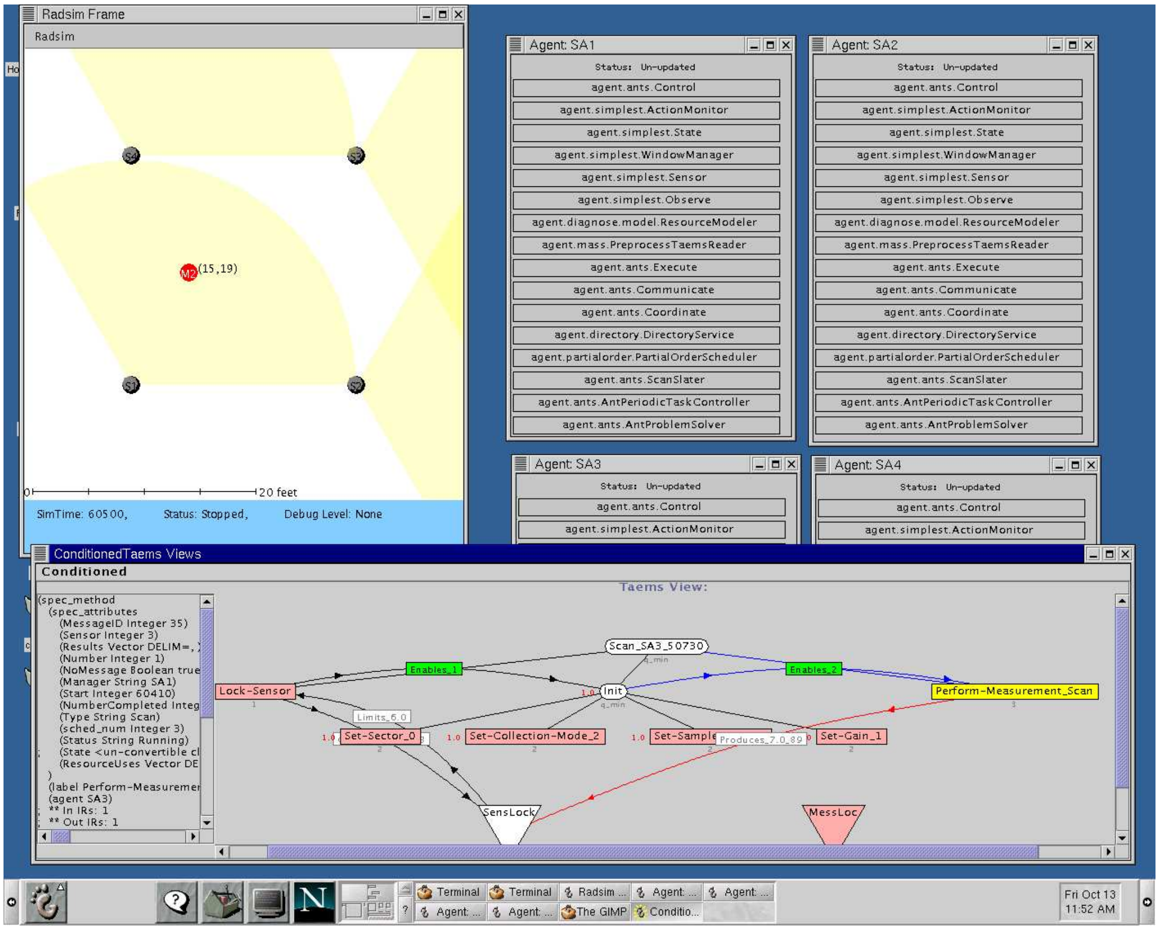


Figure 10: Here, the agents are actively scanning for a target. The simulator was set to start moving the target at time 60000 so the target has just began to move. The sensors are currently performing a scan, which is a periodic task, of the area and should find the target. The TÆMS iewer shows the execution of a scan task. Notice that the SensLock resource (represented as a triangle) is empty preventing other tasks from interfering with the measurement.

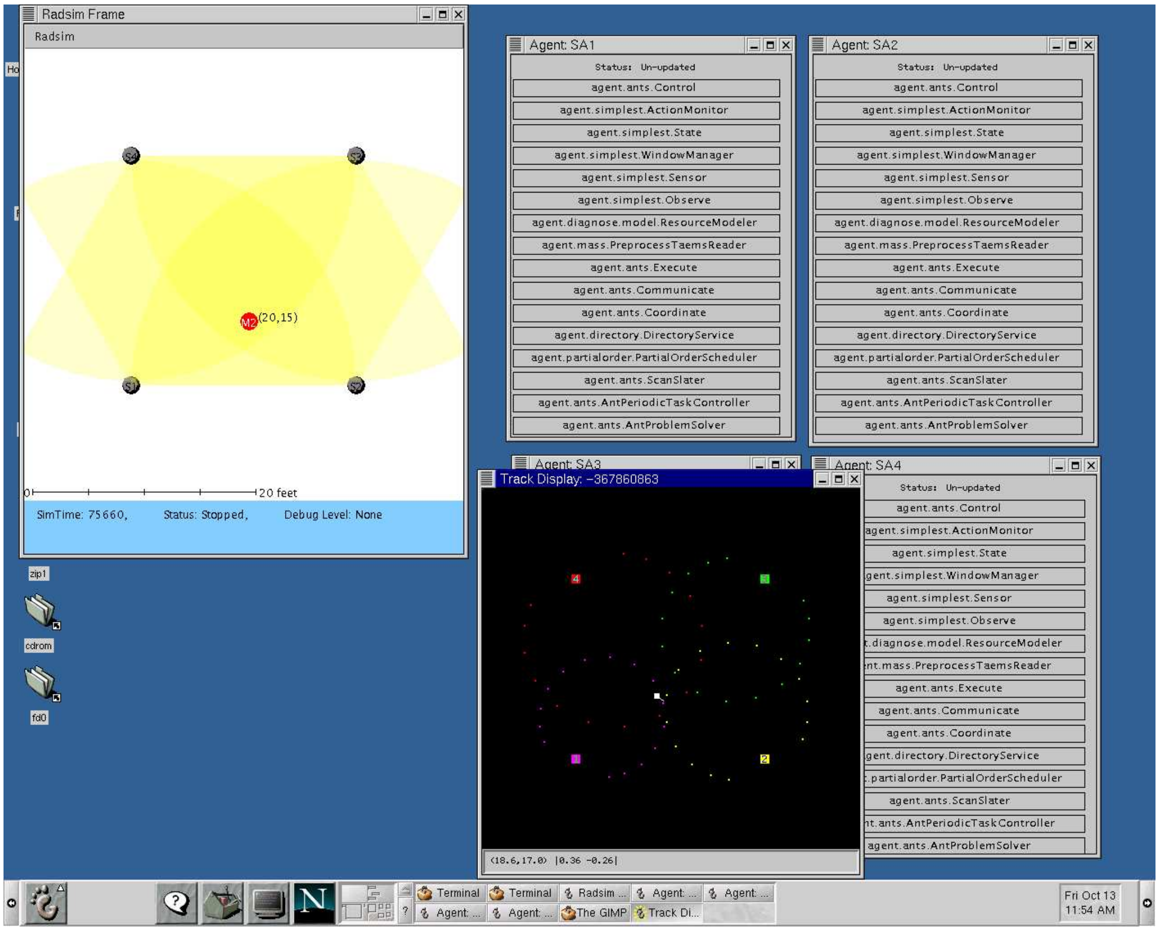


Figure 11: After locating the target, the track manager tasks sensors to track it. In this simple scenario, all of the agents have been asked to track. Notice that all of the agents have their sensors on at the same time. Synchronization is required in order to get a high quality location prediction. The window in the lower center of the screen shows that estimated location of the target along with the amplitude rings used to create that estimate.

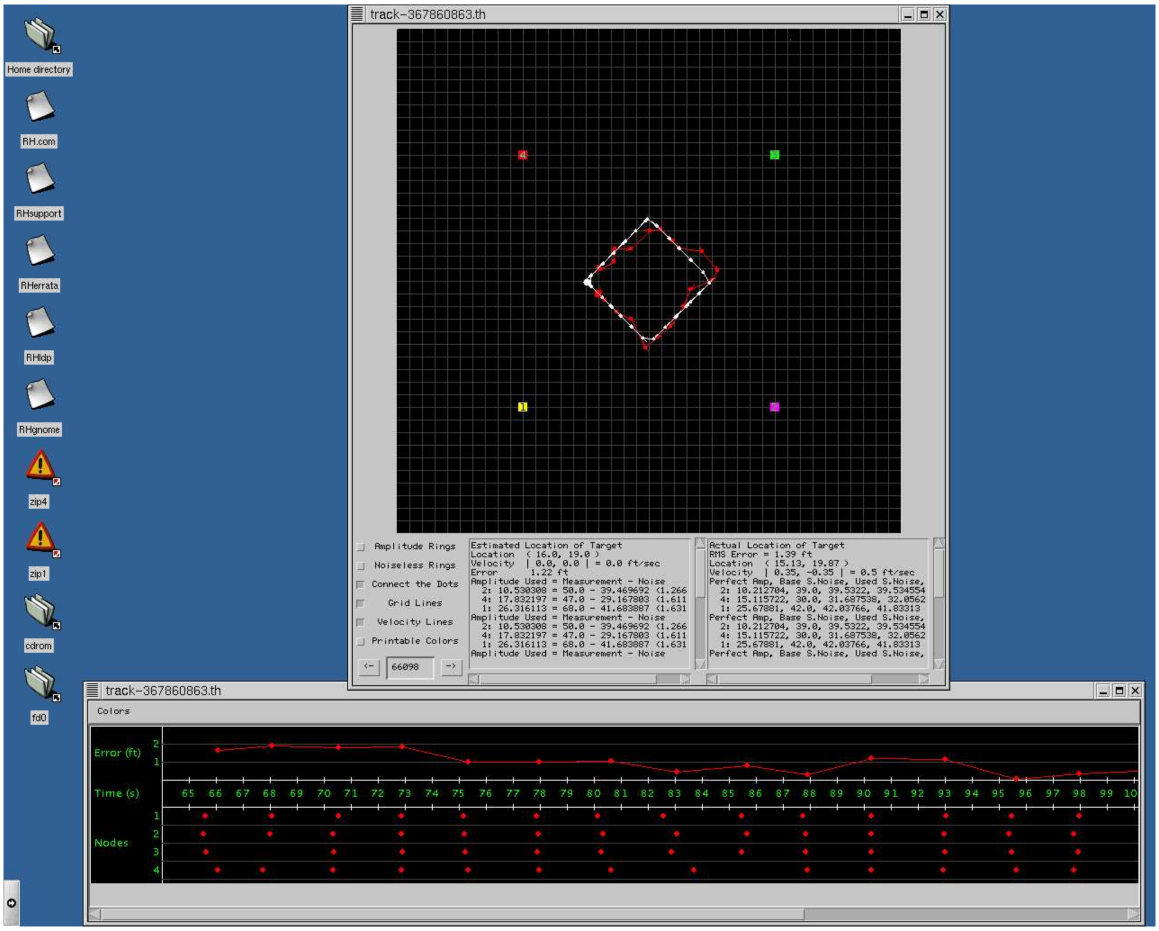


Figure 12: This screen shot shows our visualization tool for the run just competed. The upper window show the actual vs estimated path of the target. The lower window shows the RMS error of the track as well as the relative synchronization of the agents tracking the target.