TEKNOWLEDGE CORPORATION

ANTS VISUALIZER

DOCUMENTATION OF THE ANTS CHALLENGE PROBLEM

By

ALEXANDER EGYED

Created: 14 January 2003

ANTS VISUALIZER

The ANTs environment consists of numerous distributed software agents and hardware components. System development in such an environment requires a means of observing and validating the potentially hidden activities of these components. Existing visualization and debugging tools provide some mechanisms for observing behaviors and detecting faults in individual components, but the fast-paced nature of ANTs agents makes these conventional user interfaces (visualizations) and debugging techniques less effective. This chapter will discuss several techniques for visualizing and debugging complex, real-time, agent-based systems. These techniques vary in their level of invasiveness and general applicability.

OVERVIEW

User interfaces are a critical means of visualizing and verifying the correct behavior of a system. This section discusses visualization strategies for agent-based systems. The primary goal of these visualizations is to support the developers and testers in observing and debugging agent-based systems, although they have also proved useful for explaining domain and solution concepts to third parties. Two visualization strategies are discussed next: (1) infrastructure visualization and (2) agent visualization.

ANTs agents negotiate over the optimal use of radar nodes to track an unknown number of targets within a given two-dimensional space (the altitude dimension is removed from the ANTs challenge problem). The sensors are key in this setting since their measurements are the raw data that is used to interpret targets locations. Two limitations exist in the ANTs challenge problem that constitute key complexities:

- only one of the three sensors on a radar node (each covering approximately one third of the angular space around a node) can take measurements at any given point in time. Radars thus have 'blind spots' through which targets can move without being detected.

- only a limited bandwidth for agent communication exists. This limitation prevents all sensor measurements from being shared among all agents.

ANTs agents are deployed in a distributed setting where no single agent has complete knowledge of the overall system state and activities. ANTs agents thus have a limited knowledge of their surrounding world. Therefore their goal is to draw conclusions based on the limited knowledge they have available, including knowledge they are able to acquire through communication with other agents. Agent-level visualization is thus about how the agents perceive the world, and is usually agent-specific, since different solutions likely differ in how they make and respond to those perceptions.

The alternative to agent visualization is infrastructure visualization, which captures global knowledge about the world by operating external to the agents. This world knowledge may be comprised of a combination of things the agents have access to (e.g., sensors) or not have access to (e.g., target location). Since there is only one world and since all agents are part of the same world, only one infrastructure visualization is needed.

Infrastructure visualization is generally applicable and independent of individual agent implementations, but this independence has other limitations. Infrastructure visualization cannot observe agents directly. Instead it instruments the underlying infrastructure used by the agents, and then monitors that usage to track activities. Infrastructure visualization thus captures raw resource usage, such as sensor measurements being taken, but it does not attempt to provide rationale for why agents behave the way they do. Instead, infrastructure visualization is useful in evaluating the quality of agents as a measure of some solution-independent constraint. For example, the goal of the ANTs agents is to optimally track all targets at all times. This is a global constraint that can be evaluated using knowledge of current target locations and sensor orientations.

In summary, agent visualization provides rationale for agent behavior but it is hard to evaluate their effectiveness in satisfying the global goal. Agent visualization is useful in determining whether agents behave optimally given the limited knowledge of the world they have. Alternatively, infrastructure visualization is a means of judging the quality of agents and it provides a mechanism for comparing different solutions along a common set of metrics. It is also a means of detecting flaws in agent behavior based on some global properties they violate. A combination of infrastructure and agent visualizations is desirable to detect inconsistencies between the actual state of the world (the infrastructure) and the presumed state of the world (the agent).

ANTS INFRASTRUCTURE VISUALIZATION

The ANTs infrastructure consists of hardware components such as sensors, targets, and communication channels, and it consists of software components such as agents, data interpretation components (trackers) and support libraries. Additionally, a simulation environment called Radsim exists, which simulates the behavior of the hardware components for testing purposes. The simulated software components have interfaces that are (almost) identical to the real hardware components. Agents are thus executable on both with only minor alterations.

The ANTs infrastructure originally did not fully support visualization; only Radsim was capable of visualizing some simulated components (e.g., target movement, sensor modes and orientation). It was therefore necessary to instrument the ANTs infrastructure, both hardware and simulated, to capture all data necessary for visualization. This was achieved by instrumenting the interfaces to and from the individual components. For example, the agents gained access to the sensors and communication medium through a library of Java classes. Instrumentation code was added to those Java classes to intercept and forward required data to the visualization tool itself. The visualization tool then interpreted this data and visualized it in a meaningful fashion. The following describes briefly what hardware and software components were instrumented and why.

Radars/Sensors

Sensors measure amplitude and frequency values that give clues about the location, movement direction, and speed of targets. Only one type of sensor was available but it was capable of operating in four different modes. Each mode implied restrictions on the type of measurement taken (amplitude and/or frequency) and the duration required to take them (1-3 seconds). The purpose of the instrumentation was to capture information about the current state of all sensors (their location, orientation, and other attributes), the times and kinds of measurements, and the actual measurement values. Radars are illustrated in Figure 1 as gray circles with colored cones where the cones depict the orientation of the sensors (the active sensor head) and the colors of the cones depict the mode being used to take measurements. Amplitude measurements taken by sensors are depicted using ellipses, where the sensed location of the target is along the circumference of the ellipse. Frequency

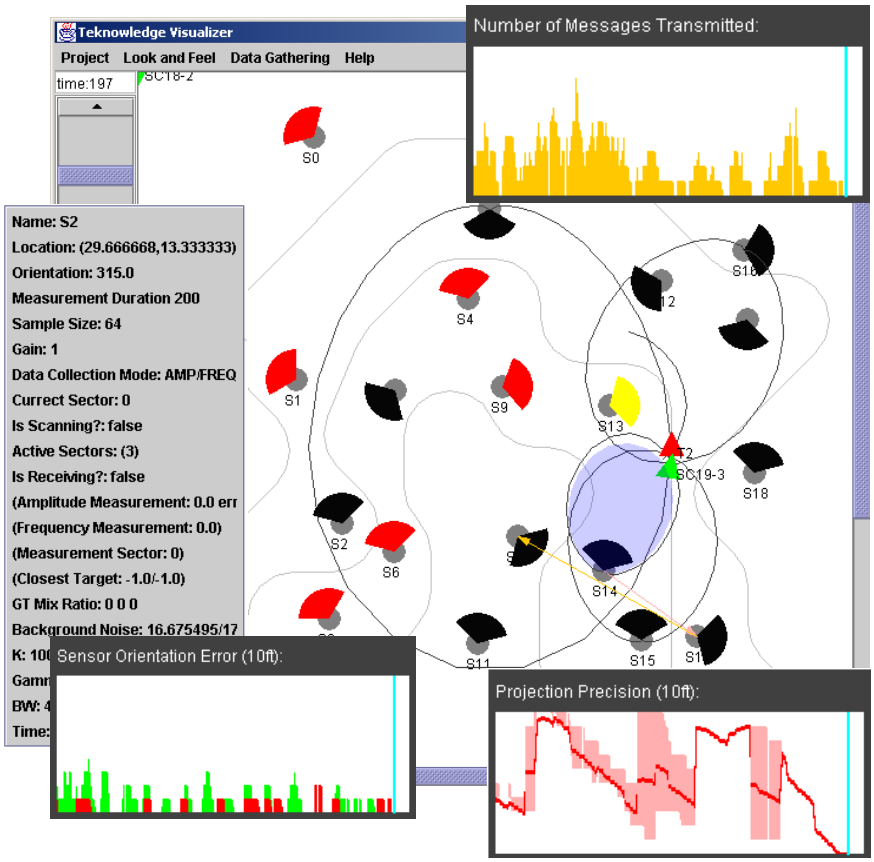


Figure 1. Teknowledge Visualizer illustrates Infrastructure Usage and operates in Real-Time and Playback Mode

measurements are depicted as cones where the sensed location of the target is within 15ft of the cone's direction .

Targets (Ground Truth)

Recall that the purpose of the agents' negotiation was to use available resources to detect and track the various targets moving through the environment. An important component of evaluating the progress of this task is to know where the targets actually are, including their individual locations, movement directions, and speed, which can be compared with the agents' results. Instrumentation was added to obtain real time target location, direction, and speed. This was straightforward in the simulated infrastructure; since the target was itself simulated it was straightforward to get the required information. Instrumenting the hardware infrastructure was much more elaborate because accurate data needed to be generated from the model trains which served as targets. To do this, synchronization points were added to the train tracks, and whenever the train would pass a sync point, the time of this event was recorded. Given that the track layout, location of sync points and train speed were known in advance, it was then possible to create a prediction algorithm that interpolated train location, direction, and speed. A more advanced version of the prediction algorithm also used a wireless mouse to monitor train movement, where the ``distance traveled'' by the mouse was used to determine location on the track given a fixed starting point. Targets are depicted in Figure 1 as red triangles were the sharp end of the triangle indicates target direction.

Communication Channels

The infrastructure supported several types of communication medium to determine the system's performance under different conditions. Instrumentation captured the utilization of the individual communication channels and details about the messages sent across them, such as message origin, destination, and length. Message traffic is depicted in Figure 1 in form of arrows between radar nodes.

Trackers

The tracker was capable of projecting presumed target locations, directions, and speeds based on sensor measurements. Instrumentation of the trackers captured the data given as input (amplitude and frequency measurements of sensors) and the data produced as output (target location, direction, and speed). The projected targets are depicted in Figure 1 in the form of green arrows. Input measurements given to trackers are depicted in the form of colored measurement cones (blue ellipses for amplitude measurements and red cones for frequencies).

Quality Criteria or Indicators

Although the ANTs infrastructure visualization does not visualize agent-specific information, which is discussed later, it can be used to define quality criteria and indicators that define good agent behavior. For example, any given radar should be active and searching in the right direction when a target is close. If it is inactive or not searching in the right direction then it may have missed an opportunity to provide measurement data for the tracker.

A special feature of the ANTs infrastructure visualizer is its ability to visualize infrastructure usage in real-time, where changes to the infrastructure are displayed as they are occurring. The instrumentation of the infrastructure utilized a fast network to send data to the visualizer. In cases where the network usage of the instrumentation could have adversely affected agent negotiation (e.g., RF), an alternative network was used. The visualizer was also capable of storing instrumentation data for later playback.

INSTALLATION

The visualizer package is distributed in form of two zipped files, called "visualizer.zip" and "radsim2.zip." Although the core of the visualizer files are packaged into "visualizer.zip", its functioning requires an instrumented simulation environment. The distributed "radsim2.zip" is such an instrumented simulator. To install, please close all running applications related to ANTS; especially previous versions of RadSim2, Agents, and the Visualizer. The visualizer was implemented in Java and should execute on any compatible platform. The visualizer was tested on Windows NT, Windows 2000, Windows XP, and Red Hat Linux only. It requires about 300KB of disc space.
Unzip those files into separate directories and adjust the CLASSPATH variable (i.e., if the visualizer is installed into c:\ants\visualizer then set its CLASSPATH to c:\ants).

Visualizer Command Syntax:

```
Visualizer/Application port config-file [GT-address GT-port]

    [port] listening port for visualizer; use 9876
    [config-file] config file (same as RadSim2) containing sensor,
              target, and world data
    [GT address] optional IP address of GT Server; use 10.0.0.51
```

```
[GT port] optional port number of GT Server; use 2000
```

Default Execution on Mitre Hardware:
`java Visualizer/Application 9876 /…/config.txt 10.0.0.51 2000`
where the last two parameters correspond to the location of the GT Server.

Default Execution on RadSim2:
`java Visualizer/Application 9876 /…/config.txt`
(RadSim2 does not require a GTServer)

Please use JDK 1.3 or better to run the visualizer.

Note: The Mitre Hardware provides a GTServer for real-time target information. This server is not available with RadSim. Do not provide a GT address and port if the Visualizer is used with RadSim.

RadSim2 Command Syntax:

```
Radsim [-c config-file]
    [config-file] config file containing sensor, target, and world
              data
```

Default Execution :
`java Radsim –c /…/config.txt`

SETUP OPTION 1: Execute the Visualizer on the same machine as RadSim: Run the visualizer first, then RadSim2, and then the agent.

SETUP OPTION 2: Execute the Visualizer on a different machine: The Radsim instrumentation is configured to expect the Visualizer on the same machine (localhost). If the visualizer is on a different host than Radsim then edit the Log.java file in /…/RadSim2/source/runtime/util/Log.java and change the IP address of the command Instrumentation.init("localhost", 9876) to your host. Please leave the port number as is. If the port number is changed then the visualizer's port parameter needs to be updated accordingly. Next, recompile the Instrumentation.java file. Finally, run the visualizer, then RadSim2, and then the agent.

WARNING: If you have a previous version of the Visualizer or RadSim2 installed then you **must ensure that their CLASSPATH entries are disabled while executing this software**. It is always recommended to uninstall earlier versions.

HOW TO START THE VISUALIZER WITH RADSIM

This section describes how to execute and operate the Visualizer together with the provided, instrumented RadSim2. Note that, in order to benefit from the visualization, agent(s) need to interact with RadSim. The visualizer then depicts, in real time, the infrastructure usage of the agent(s) on RadSim.

CONFIG FILE

The visualizer takes as input a config file that defines the size of the simulated world, sensor information, and target track information. The format of the visualizer config file is identical with the format of the RadSim config file. Figure 2 depicts an example config files that was used during the ANTS final demo in Fairfax, VA in October 2002. Although the config file for RadSim accepts a wide variety of data, the config file for the visualizer is limited to the following three types of information:

- ROOMSIZE: defines the size of the simulated world. It is used by the visualizer to define boundaries for its graphics

- SENSOR: defines name, location, orientation, and noise level for all individual sensors. It is used by the visualizer to depict sensors in its graphics

```
ROOMSIZE 60 70

SENSOR S0 4.500000 11.500000 105.000000      7328.1 2.0 40.0 16.233160 1391.9 2.0 40.0 17.322033 3000.0 2.0 40.0 14.282719
SENSOR S1 20.333333 10.333333 120.000000      4510.4 2.0 40.0 18.664583 4513.7 2.0 40.0 16.689229 4445.9 2.0 40.0 14.621914
SENSOR S2 29.666667 13.333333 315.000000      10017.8 2.0 40.0 16.675496 5346.6 2.0 40.0 17.349499 3786.5 2.0 40.0 15.953537
SENSOR S3 35.916667 12.500000 90.000000      3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0
SENSOR S4 15.000000 21.500000 75.000000      6887.8 2.0 40.0 16.222170 5066.3 2.0 40.0 19.286450 3566.6 2.0 40.0 16.395081
SENSOR S5 21.000000 16.750000 195.000000      5279.1 2.0 40.0 13.907772 5141.4 2.0 40.0 14.878189 2920.1 2.0 40.0 14.409727
SENSOR S6 31.500000 16.666667 75.000000      1768.1 2.0 40.0 15.956859 3000.0 2.0 40.0 21.255993 2954.1 2.0 40.0 14.159673
SENSOR S7 39.666667 15.666667 285.000000      3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0
SENSOR S8 9.166667 22.916667 240.000000      4111.6 2.0 40.0 16.766363 13969.0  2.0 40.0 14.735120 2708.0 2.0 40.0 14.276213
SENSOR S9 20.750000 23.750000 340.000000      4251.1 2.0 40.0 17.267288 3279.0 2.0 40.0 14.987150 5092.8 2.0 40.0 14.431488
SENSOR S10 30.500000 24.750000 165.000000      2296.0 2.0 40.0 19.709475 3462.2 2.0 40.0 16.876560 35650.6 2.0 40.0 16.786581
SENSOR S11 37.500000 22.166667 165.000000      3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0
SENSOR S12 13.833333 34.083333 180.000000      5651.4 2.0 40.0 16.529898 3968.4 2.0 40.0 18.176639 2256.4 2.0 40.0 20.961947
SENSOR S13 22.000000 30.750000 105.000000      3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0
SENSOR S14 32.750000 30.333333 165.000000      3968.4 2.0 40.0 20.192915 782.0 2.0 40.0 20.299242 17678.2 2.0 40.0 21.289469
SENSOR S15 37.250000 32.833333 300.000000      5651.5 2.0 40.0 19.258065 3194.3 2.0 40.0 13.634188 5752.6 2.0 40.0 14.770008
SENSOR S16 11.833333 39.500000 330.000000      11497.4 2.0 40.0 15.662804 5632.1 2.0 40.0 13.751654 3715.8 2.0 40.0 16.137972
SENSOR S17 16.416667 39.750000 225.000000      3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0
SENSOR S18 26.333333 40.166667 285.000000      3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0 3000.0 2.0 40.0 17.0
SENSOR S19 37.000000 36.416667 315.000000      4301.6 2.0 40.0 14.305803 3000.0 2.0 40.0 15.148045 5376.5 2.0 40.0 16.520046

MODOBJ M0 60.000000 45.000000 1.0 0 0 50.000000 45.000000 48.851950 45.228361 47.878680 45.878680 46.905410 46.528998
45.757359 46.757359 44.609309 46.528998 43.636039 45.878680 38.332738 40.575379 37.359468 39.925061 36.211418 39.696699
33.711418 39.696699 32.563367 39.468338 31.590097 38.818019 29.822330 37.050253 28.849060 36.399934 27.701010 36.171573
26.552960 36.399934 25.579690 37.050253 18.508622 44.121320 17.535352 44.771639 16.387302 45.000000 15.239251 44.771639
14.265981 44.121320 13.615663 43.148050 13.387302 42.000000 13.387302 39.500000 13.158940 38.351950 12.508622 37.378680
5.437554 30.307612 4.787236 29.334342 4.558875 28.186292 4.558875 18.186292 4.787236 17.038241 5.437554 16.064971 14.276389
7.226136 15.249659 6.575818 16.397709 6.347457 18.897709 6.347457 20.045760 6.575818 21.019030 7.226136 22.786797 8.993903
23.760067 9.644222 24.908117 9.872583 26.056167 9.644222 27.029437 8.993903 30.564971 5.458369 31.538241 4.808051 32.686292
4.579690 45.186292 4.579690 46.334342 4.808051 47.307612 5.458369 48.280882 6.108688 49.428932 6.337049 59.428932 6.337049
MODOBJ M1 60.000000 40.000000 1.0 0 0 47.500000 40.000000 46.351950 39.771639 45.378680 39.121320 41.843146 35.585786
40.869876 34.935468 39.721825 34.707107 22.221825 34.707107 21.073775 34.478745 20.100505 33.828427 19.450187 32.855157
19.221825 31.707107 19.221825 29.207107 18.993464 28.059056 18.343146 27.085786 17.369876 26.435468 16.221825 26.207107
15.388492 26.207107 14.240442 25.978745 13.267172 25.328427 12.616853 24.355157 12.388492 23.207107 12.388492 20.707107
12.616853 19.559056 13.267172 18.585786 16.802706 15.050253 18.046328 14.219290 22.665726 12.305873 23.813776 12.077512
24.961827 12.305873 25.935097 12.956191 27.702864 14.723958 28.676134 15.374277 29.824184 15.602638 30.972234 15.374277
31.945504 14.723958 32.595823 13.750688 32.824184 12.602638 32.824184 10.102638 33.052545 8.954588 33.702864 7.981318
34.676134 7.330999 35.824184 7.102638 38.324184 7.102638 39.472234 7.330999 40.445504 7.981318 41.418774 8.631636 42.566825
8.859997 45.066825 8.859997 46.214875 9.088359 47.188145 9.738677 48.161415 10.388995 49.309465 10.617357 59.309465 10.617357
MODOBJ M2 60.000000 35.000000 1.0 0 0 50.000000 35.000000 48.851950 34.771639 47.878680 34.121320 40.807612 27.050253
39.834342 26.399934 38.686292 26.171573 33.686292 26.171573 32.538241 26.399934 31.564971 27.050253 29.797204 28.818019
28.823934 29.468338 27.675884 29.696699 26.527834 29.468338 25.554564 28.818019 24.904245 27.844749 24.675884 26.696699
24.675884 25.863366 24.447522 24.715316 23.797204 23.742045 22.823934 23.091727 21.850664 22.441409 21.200346 21.468139
20.971984 20.320089 21.200346 19.172038 21.850664 18.198768 22.823934 17.548450 23.971984 17.320089 27.305318 17.320089
28.453368 17.548450 29.426638 18.198768 30.399908 18.849087 31.547958 19.077448 34.047958 19.077448 35.196009
```

Figure 2. Config File Example as used in ANTS Final Demo, October 2002

- MODOBJ: defines the track of the target(s) in terms of its initial location, speed, and navigational points. It is used by the visualizer to indicate in advance the known track of all targets.

In order to maintain compatibility with RadSim, the visualizer config file may contain other information. The visualizer simply ignores config file entries that are not of the above types.

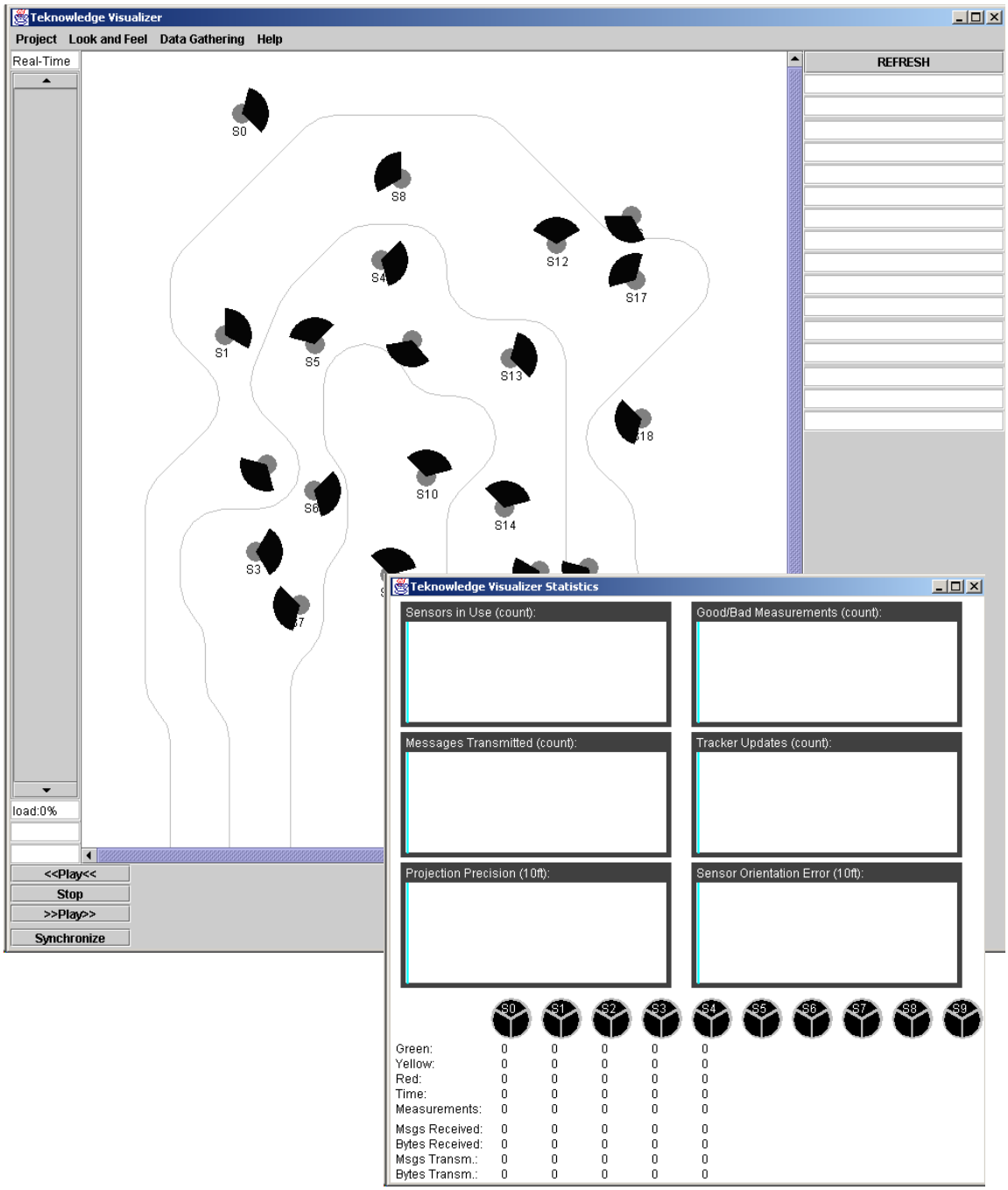


Figure 3. Visualizer after Startup (main window on top; statistics window on bottom)

START THE VISUALIZER

The window in Figure 3 is the default startup window of the Visualizer. In the main panel (top center), the visualizer displays known sensor and track information as defined in the config file. The scrollable size of the center panel depends on the defined room size in the config file. Sensors are depicted in gray circles with an attached cone to indicate sensor orientation. Initially the cone is black which indicates that the sensor is not scanning. Target tracks are depicted in light gray, thin lines. Targets are expected to move only along those lines.

The visualizer is initially in the 'real-time' mode (note the word 'real-time in the upper left). While in the real-time mode, the visualizer listens to the port specified during startup (default 9876) for current activities in RadSim2, trackers, agents, and other components. Information it receives through this port is instantly updated in the visualizer. Information received can also be captured into a file using the 'Data Gathering' menu item. Of course, the visualizer can playback captured data at a later time. In the playback mode, the left side of the visualizer window provides a timeline to quickly navigate backward and forward along the captured data. The timeline is of importance since captured and visualized data tends to be only accurate for brief moments in time. The timeline thus allows to observe changes to element states over time from the moment the data capture started until the moment where it ended. The 'Lock and Feel' window allows to switch between Motif and Windows-

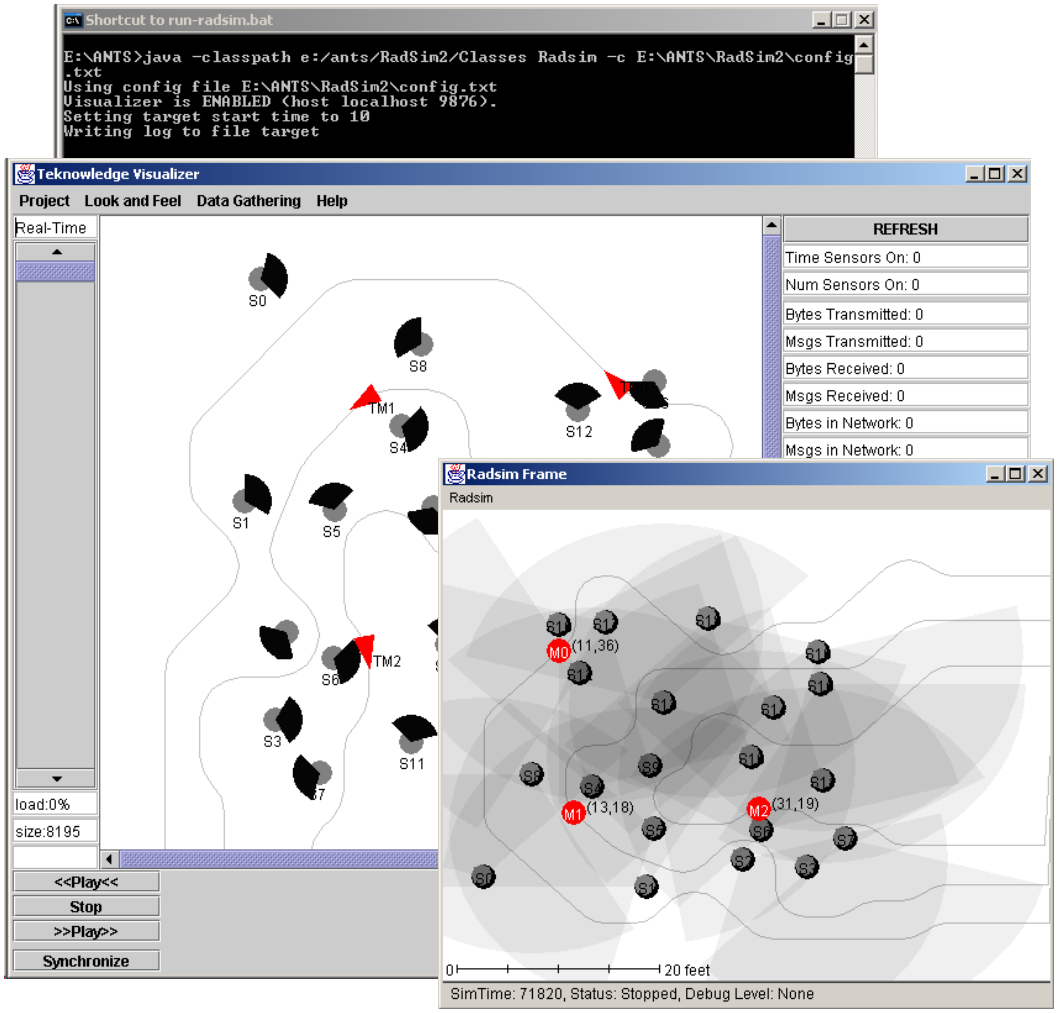


Figure 4. Instrumented RadSim interacting with the Visualizer

compatible user interfaces. The project menu provides support for loading previously captured data files and exiting the visualizer. Finally, the help menu displays the about box.

Aside from the visualizer main window (top part of Figure 3), the visualizer also displays statistical data (bottom part of Figure 3). The statistical window summarizes quality information (top) and it displays information about individual sensors (bottom).

START RADSIM2

Please start RadSim2 on the same host as the visualizer. If it is desired to run the visualizer on a different host then please follow the setup instructions given earlier. Figure 4 (top left) displays the command line window for RadSim2. If the string "Visualizer is ENABLED" appears then RadSim2 connected successfully to the Visualizer. In this case, infrastructure changes expressed through RadSim2 are instantly transmitted to and depicted by the Visualizer (see RadSim2 depicted in bottom of Figure 4 and the Visualizer in the middle). If RadSim2 and Visualizer are not configured correctly or if the Visualizer was not started prior to starting RadSim2, then the command line window of RadSim2 will display the string "Visualizer is DISABLED." In either case, enabled or disabled, the displayed string will be followed by a short description of host name and port that indicates the location (host = IP address of visualization machine; port = port the visualizer is listening to) the visualizer was expected. In case there are multiple potential locations for the visualizer then the system can be configured such that it iterates through all of them. In this case, the system will stop once it finds a visualizer or runs out of potential locations. We revisit this issue later for a more in-depth discussion on this kind of configuration.

START THE AGENT

The agent(s) use the RadSim library to observe targets with available resources (e.g., sensors). Thus, no explicit instrumentation is required to visualize agents since the agent(s) activities can be observed indirectly through their effect on the instrumented RadSim. Once the Test Agent is started successfully, the Visualizer will display, in real time, how the agent(s) affect the simulating components (see Figure 5).

HOW TO START THE VISUALIZER WITH HARDWARE

This section describes how to execute and operate the Visualizer together with the instrumented hardware infrastructure. In this setting, the visualization operates in much the same manner as in RadSim. The primary difference between the RadSim infrastructure and the hardware infrastructure is that the hardware infrastructure is typically already running before the visualizer is started. It is generally infeasible to shutdown and restart the hardware infrastructure frequently. In order to maintain a consistent handling of the visualizer in both the hardware and software infrastructure, we instrumented the hardware infrastructure such that it is activities will only commence with the startup of the agent(s). Thus, the instrumentation of the hardware infrastructure is dormant while no agent is running. The startup sequence of the visualization in context of the hardware infrastructure is such that the hardware infrastructure is started first, followed by the visualizer and, lastly, the agent(s). Visualizer and agents can be shut down and restarted repeatedly.

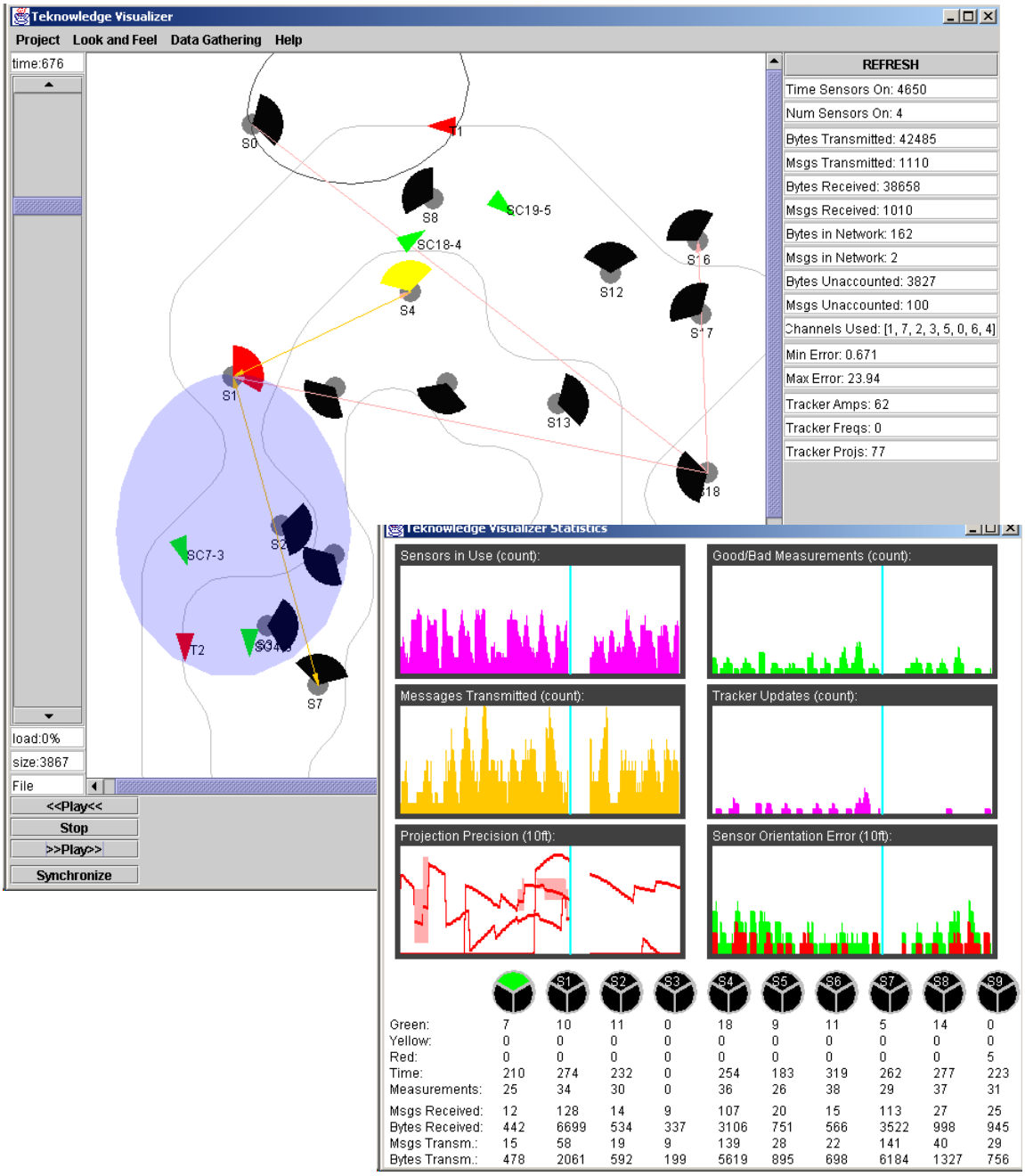


Figure 5. UMass Agent Behavior Depicted in Visualizer

The major downside of this solution is that the current state of the hardware infrastructure is unknown at the time the visualizer is started. Since the visualizer assumes the infrastructure to be in a 'default' state, the initial visualization may be incorrect. However, the startup of the agent(s) always re-initializes the hardware infrastructure. Since the visualizer is active before the start of the agent(s) it monitors this initialization after which, it is guaranteed that the hardware infrastructure and the visualizer are consistent.

The config file used in the hardware infrastructure is not identical, albeit very similar, to the config file used for RadSim (software infrastructure). The visualizer recognizes this difference and is capable of interpreting both of them. Please consult the instructions for the hardware infrastructure

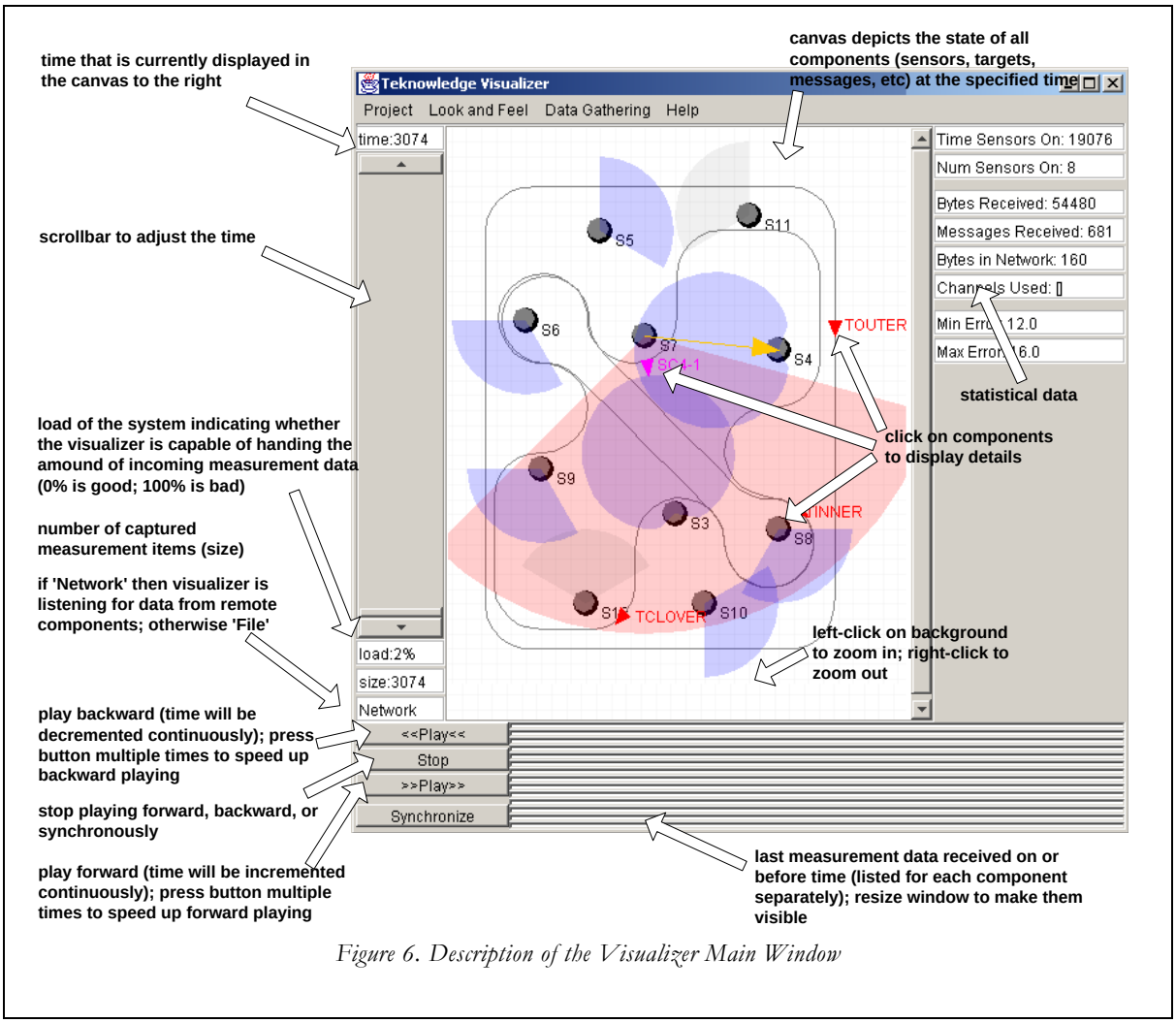


Figure 6. Description of the Visualizer Main Window

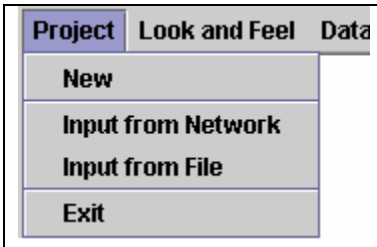
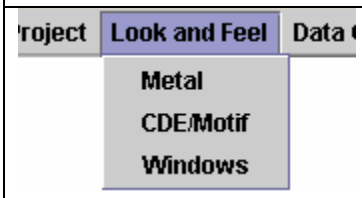
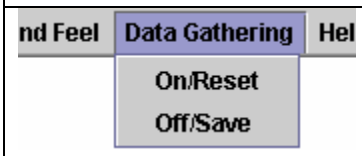
for more information about its config files. The agents used on the hardware infrastructure are also often not identical with the ones for RadSim. Since the visualizer does not instrument agents directly but only their effects on the hardware infrastruture, this difference is not significant.

NAVIGATING THE VISUALIZER

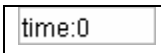
As was introduced earlier, the visualizer consists of a main window and a statistical window. Figure 6 depicts the Visualizer main window and highlights the major graphical elements embedded in it. The top part of the main window contains a menu with the following elements:

Project Look and Feel Data Gathering Help

Each menu item encapsulates a range of menu choices. The *Project* menu contains the menu items *New*, *Input from Network*, *Input from File*, and *Exit*; the *Look and Feel* menu contains the menu items *Metal*, *CDE/Motif*, and *Windows*; the *Data Gathering* menu contains the menu items *On/Reset*, and *Off/Save*; and, finally, the *Help* menu contains the menu item *About*. The following describes the menu items:

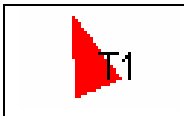
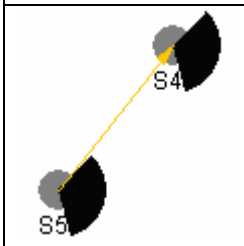
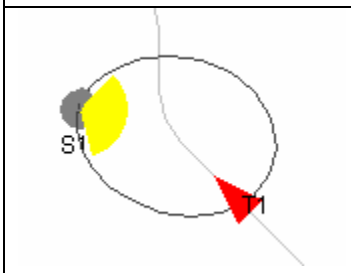
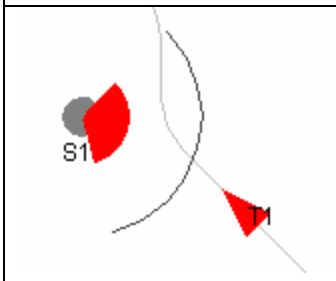
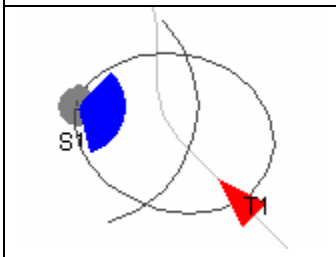
	• New: reinitializes the visualizer according to the config file and purges gathered data from the memory. • Input from Network: instructs the visualizer to expect data to arrive from the instrumentation port. The visualizer then displays instrumentation data in real-time as it arrives from either the software infrastructure (RadSim) or the hardware infrastructure. • Input from File: instructs the visualizer to stop listening to the instrumentation port and instead load a pre-existing data file. Data files are generated during data capture. • Exit: closes the visualizer.
	Toggles the user interface of the visualizer between the three modes *Metal*, *CDE/Motif*, and *Windows*. Neither choice affects the functionality of the visualizer but allows users to switch the user interface into a more familiar form. The default interface is *Windows*.
	Data Gathering allows instrumented data to be saved to a permanent data file. This menu item works only while the visualizer is listening to input from the network (see above). By pressing *On/Reset* the user is prompted a dialog box to specify a data file name. Once *Save* is pressed in this dialog box, data is saved into the file as it arrives from either the instrumented software infrastructure or hardware infrastructure. Pressing *Off/Save* stops the data capturing. It must be noted that pressing *On/Reset* first captures the current state of sensors before writing instrumented data. This ensures the correct context of gathered data.
	The Help menu currently only allows to display the *About* box with usage and copyright information. It is planned in a future release to add on-line help.

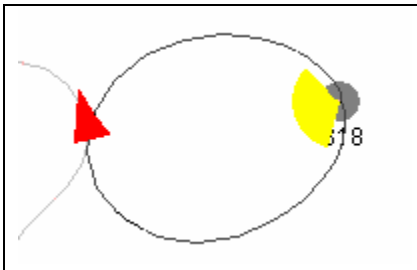
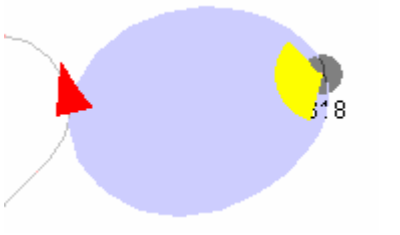
The main window of the visualizer uses a variety of graphical information to describe the current state and context of the software and hardware infrastructure and to control its visualization. The following table discusses this:

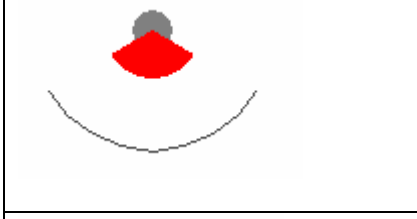
	Time information is displayed to indicate the time of the current visualization state. Generally, the time starts at zero after switching on the data gathering mode or after (re) initialization (e.g., new menu item). The time is indicated in 100ms intervals.

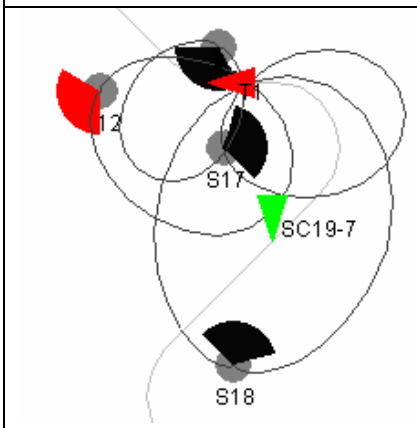
	Location and orientation of sensors is pre-defined in the config file specified during the start-up of the visualizer. The location includes x and y coordinates; the orientation includes the degree the '0' sensor head is offset from the nominal axis. Since sensors have exactly three sensor heads and since only one sensor head can be active at any given point in time, the graphical depiction of the sensor uses a cone to visualize the currently active sensor head. If the cone is black then the sensor is switched off (= does not gather data). The sensor may be colored which has the following meaning: • blue: sensor is collecting amplitude and frequency data • yellow: sensor is collecting amplitude data only • red: sensor is collecting frequency only in duration mode • green: sensor is collecting frequency only in interval mode The sensor name is depicted underneath the sensor.
load:0%	Load information indicates the CPU load of the visualizer in gathering data from the instrumentation port, depicting it , and potentially saving it to a file. If the load reaches close to 100% then the visualizer is unable to depict every piece of data; it will instead skip time intervals.
size:3867	Size information indicates the amount of gathered data. Size will be at zero initially and it will increase only if data capture is switched on (see Data Capture menu).
	Data that is captured by the visualizer (*Data Capture On/Reset* menu item) or data that is loaded into the visualizer from a file (*Load from File* menu item) can be re-played in several ways. Pressing the >>*Play*>> button plays the data forward until the end is reached. Pressing the <<*Play*<< button plays the data backward until the beginning is reached. The speed of playing the data forward or backward can be increased by pressing the buttons multiple times. The *stop* button stops data playing (it reduces the speed to 0). The play buttons essentially changes the current *time* and the visualizer displays the infrastructure state of that time. It is also possible to use the vertical scroll bar to change the time. Simply click the up or down arrows or drag-and-drop the time lever.

Synchronize	This button only works while data is captured by the visualizer in real-time from the infrastructure (*Data Capture On/Reset* menu item). By default, the visualizer will display the data in real-time while it is being captured but the user may choose to stop this or revisit an earlier time. In such a case, the data capture will proceed and the *synchronize* button will resume the real-time visualization once pressed.
Time Sensors On: 122 Num Sensors On: 10 Bytes Transmitted: 696 Msgs Transmitted: 16 Bytes Received: 644 Msgs Received: 15 Bytes in Network: 0 Msgs in Network: 0 Bytes Unaccounted: 52 Msgs Unaccounted: 1 Channels Used: [1, 7, 2, 3, 5, 0, 6, 4] Min Error: 0.900 Max Error: 0.900 Tracker Amps: 1 Tracker Freqs: 0 Tracker Projs: 1	The visualizer generates a large amount of statistical information. Most if it is visualized in a separate statistical window discussed later but some of it is visualized in the main window. The statistical information is reset after switching on the data gathering mode, after loading a data file, or after (re) initialization (e.g., new menu item). The *Time Sensors On* accumulates the total time sensors are switched on. If two sensors are on at the same time then the total time is twice the time elapsed. The measurement unit is 100ms. The *Num Sensors On* gives the number of currently active (measuring) sensors. The *Bytes Transmitted/Received* is an index of the network load in bytes. Differences between transmitted and received bytes may be attributed to data still being in transmission or to data loss (e.g., during RF). *Msg Transmitted/Received* is a similar index for the number of messages in transmission. *Bytes and Msg in Network* is an assumption on the amount of data still being in transmission. *Bytes and Msg Unaccounted* is thus an indication of data loss. *Channels Used* lists all communication channels (network channels) used. *Min and Max Error* is an index of the quality of agents in tracking their targets. It describes in feet how close a tracker projects the location of the actual target. In case multiple trackers are tracking the same target, the min error indicates the tracker that is closest to the actual target whereas the max target indicates the tracker that is furthest away. Since it is unknown which trackers track what targets, it is assumed that trackers track the targets they are closest to. This assumption is not valid always. Trackers generate their prediction based on sensor measurements (amplitude or frequency). *Tracker Amps and Freqs* indicates the number of amplitude and frequency measurements that are given as input to the tracker(s). *Tracker Projs* indicates the number of target projections that tracker(s) are able to generate.
REFRESH	Refreshes the user interface
	Light gray lines indicate target tracks. Tracks are specified in the config file and given to the visualizer during start up.

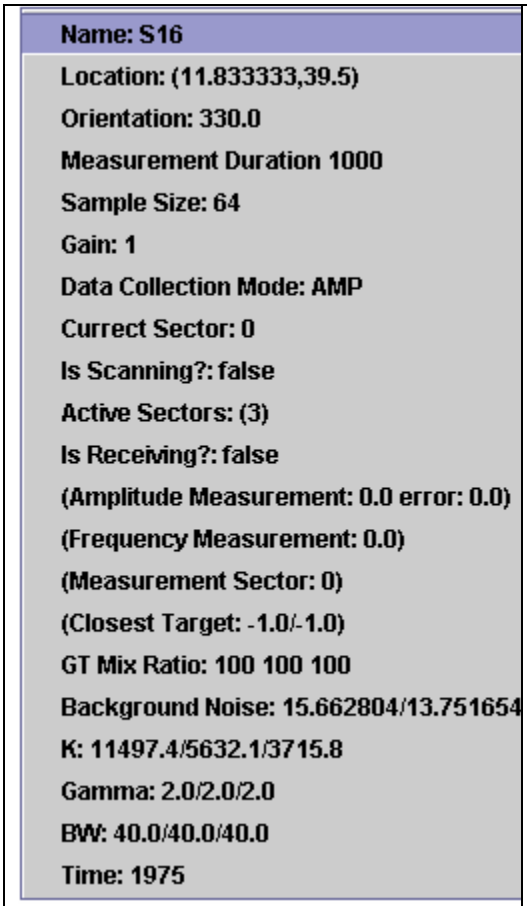
	Red triangle indicates the true target position. The sharp end of the triangle points in the direction the target is moving.
	Green triangle indicates the projected target position generated by the tracker. The sharp end of the triangle points in the direction the tracker believes the target is moving. If the projected target name starts with the letters "SC" then the tracker from the University of South Carolina is used. Otherwise the projected target name starts with the letters "BAE" for the BAE tracker.
	Communication between sensor nodes is indicated in form of arrows where the arrow head points out the destination of the message.
	A sensor makes amplitude measurements while in the blue or yellow state (see above). An amplitude measurement in turn can be translated into an ellipsoid shape that corresponds to the possible locations the target is at (assuming no noise). In other words, the target must be at any point along the circumference of the ellipse that corresponds to the measured amplitude. An amplitude measurement is graphically depicted in form of an ellipse.
	A sensor makes frequency measurements while in the blue, red, and green state (see above). A frequency measurement indicates the movement of a target in the general direction the sensor head is facing to. The value of the frequency measurement is an indication on how fast the target is moving and whether or not it is approaching. A frequency measurement is graphically depicted in form of an arc.
	A sensor makes amplitude and frequency measurements only while in the blue state (see above). The combined amplitude and frequency measurement is graphically depicted in form of an ellipse and an arch.

<table>
<tr>
<td> 
  </td>
<td>An amplitude measurement is an observation made by a sensor. It is in the discretion of the agent to decide whether or not to forward this measurement to a tracker to track a target. If an agent forwards an amplitude measurement to a tracker then this is graphically indicated in form of an light-blue, filled ellipse. The drawing of the filled ellipse is such that it does not erase the graphical information underneath it. Alpha rendering is used.

The top picture shows an amplitude measurement made by a sensor. The bottom picture shows that amplitude measurement being forwarded to a tracker.</td>
</tr>
<tr>
<td> 
  </td>
<td>A frequency measurement is an observation made by a sensor. It is in the discretion of the agent to decide whether or not to forward this measurement to a tracker to track a target. If an agent forwards a frequency measurement to a tracker then this is graphically indicated in form of an light-green, filled cone. The drawing of the filled cone is such that it does not erase the graphical information underneath it. Alpha rendering is used.

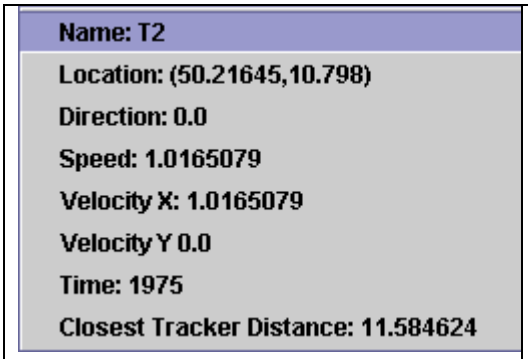
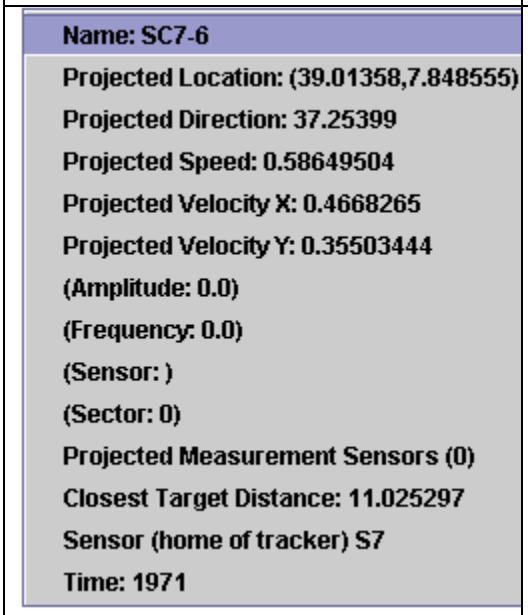
The top picture shows a frequency measurement made by a sensor. The bottom picture shows that frequency measurement being forwarded to a tracker.</td>
</tr>
<tr>
<td> 
</td>
<td>The visualizer is often presented with a large amount of information that is depicted concurrently. The figure on the left shows a target that is being tracker by four sensors. All sensor made amplitude measurements. Presumably the tracker was not updated with these latest measurements since its projection of the actual location of the target is depicted further down.</td>
</tr>
</table>

The visualizer cannot depict all gathered data graphically. In order to give the user the ability to further investigate details, the various tracker, target, and sensor icons can be clicked upon on the screen. A pop-up window appears which lists the known state of the selected item precisely. See below for a description:

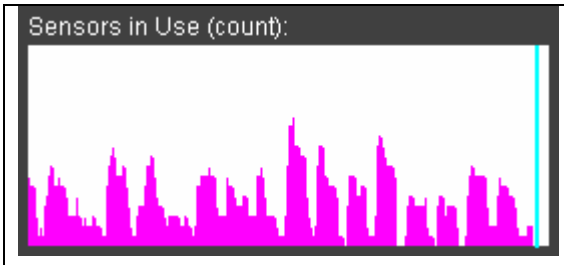
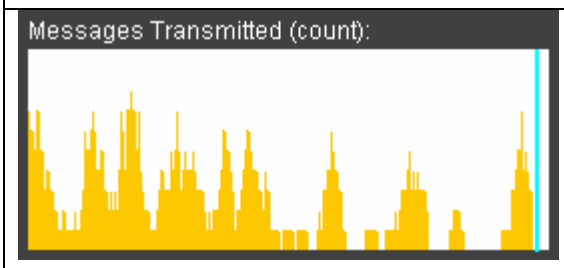
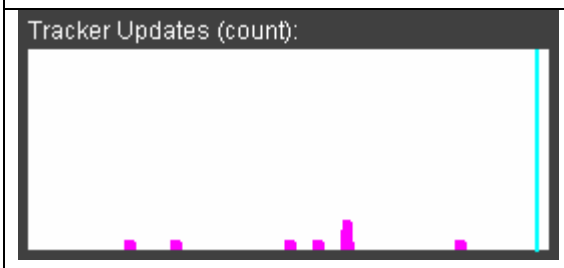
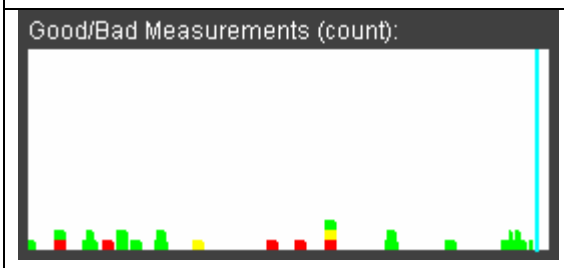
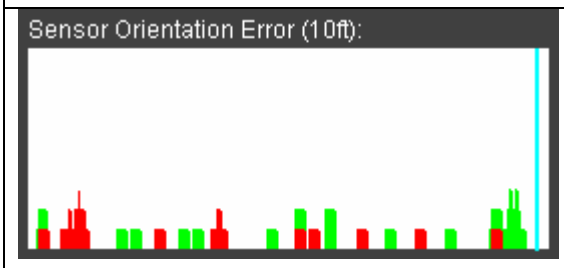


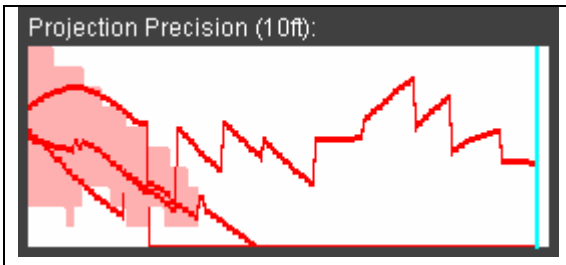
Sensor

- Name: name of sensor (from config file)
- Location: x/y coordinates of sensor (from config file)
- Orientation: angle of sensor head '0' relative to nominal axis (from config file)
- Measurement Duration, Sample Size and Gain
- Data Collection Mode: depicted graphically in form of blue, yellow, red, and green cones.
- Current Section: only one of the three sector heads can be active (0, 1, 2) at any given time.
- Is Scanning? equivalent to 'is taking measurements'
- Is Receiving? indicates if messages are currently arriving at sensor. This is graphically depicted in form of arrows.
- Amplitude and Frequency Measurements: (values)
- Measurement Section: Sector at which measurements where taken (note: due to delays the active sector may already have changed before the measurements arrive)
- Closest Target: x/y distance to the closest target (note: only available if target location is known)
- GT Mix Ratio: ratio or 'real' sensor data versus 'hypothetical' one. The section about controlling the environment will be discussed later in more detail.
- Background Noise/K/Gamma/BW: Noise information about each sensor head (sector) (from config file).
- Time: Last status update

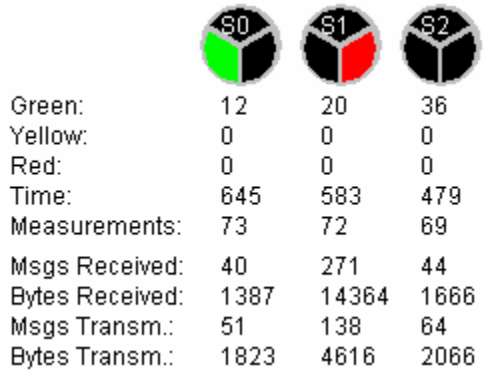
	**Target** • Name: name of target (from config file) • Location/Direction/Speed: information about target • Velocity X/Y: derived from direction and speed • Closest Tracker Distance: Distance in feet to the closest x/y position a tracker was able to guess. • Time: Last status update.
	**Tracker** • Name: name of tracker. Name comprises letters "SC" or "BAE" depending on tracker origin (SC… University of South Carolina; BAE…BAE), a number before the dash indicating the sensor node the tracker is executing upon, and a number after the dash indicating how may trackers where spawned in the past by this sensor node. • Projected Location/Direction/Speed: information about presumed location of target • Amplitude/Frequency: Amplitude or frequency measurement given as input, if any. • Sensor/Sector: Origin of measurement in terms of sensor and sensor head (sector) • Closest Target Distance: distance in feet to the closest x/y position of a target. Note that it is assumed that the tracker tracks the target it is closest to. This assumption is not valid always. • Sensor: sensor node the tracker is executing upon. This information is also embedded in the tracker name. • Time: Last status update.

Aside from the main window in Figure 6, the visualizer also has a statistical window. The following describes the various sections of the statistical window:

	Sensor use over time is graphically depicted in form of a bar chart. The peaks and valleys in the chart indicates the time of high and low sensor usage. This information is useful in determining overall sensor usage which is a key resource.
	Number of messages transmitted over time is graphically depicted in form of another bar chart. The peaks in the chart indicate high message traffic in the communication network. This information is useful in identifying potential network load/overload problems.
	Number and frequency of tracker updates over time. The more (good) data is given to the tracker the better its predictions. Similarly, it is little useful to have times of plenty (lot of data given to tracker) while also having times of starvation (no/little data given to tracker).
	Sensor measurements are potentially noisy. If "Controlling the Environment" is enabled (see later) then it can be determined how noisy measured data really is over time. If the measured data has little noise (<20%) then it is depicted in green; if the noise is between 20% and 50% then it is depicted in yellow. Noise above 50% is depicted in red. This information can be useful in determining agent robustness in the face of low/high nose.
	In almost all circumstances, it is preferable to have sensor 'look' at the closest target; not looking at a target while one is close is a waste of opportunity (every measurement is important). This statistics indicates how often sensors were told to look in the right direction when a target was close. This information can be useful in determining how well an agent understands its environment.



Projection precision is an indication how well the target is tracked over time. The peaks indicate the tracker is close; the valleys indicate the tracker is far. Tracking precision is indicated for at most 10ft. Thus, if a peak is at the top then the tracker is exactly on the target; otherwise, if a valley is at the bottom then it is more than 10ft away from the target. Multiple lines indicate that multiple targets are being tracked. Furthermore, shaded areas indicate that multiple trackers tracked the same target.



The statistical window also provides a brief summary of individual sensor nodes in terms of the quality of measurements (see good/bad measurements above), the total time duration the sensor was measuring, the total number of measurements taken, the number and bytes of messages received and transmitted.

The circle on top depicts the sensor and its three sensor heads graphically. The green, yellow, and red color are graphical indications of the quality of measurements.

The statistical information is reset (1) after switching on the data gathering mode, (2) after loading a data file, or (3) after (re) initialization (e.g., new menu item).

DATA GATHERING

Upon startup, the visualizer only displays data in real time and it does not gather data. To also gather data for later replay, the menu item *Data Gathering::On/Reset* has to be pressed. This resets the visualizer (e.g., eliminate old tracker information) and starts gathering incoming data. The size of gathered data can be seen in the text field labeled 'size.' At any time during the data gathering process can data be saved. The menu *Data Gathering::Off/Save* prompts a dialog box for defining a file name. Once saved, data can be retrieved via menu item *Project::Input from File*. Note that the import of files overwrites current data in the visualizer. Also, it stops the data gathering process via the network. This is indicated through the text field 'network' switching to 'file.'

INSTRUMENTATION OF INFRASTRUCTURE

The visualizer requires an instrumented infrastructure to gather data. There exist several versions of instrumented software and hardware infrastructures to date. If it is desired to newly instrument an infrastructure then this section describes the process.

SOFTWARE INFRASTRUCTURE (RADSIM)

Remove the file Log.java from RadSim. Create the sub-directory structure 'runtime/util/' underneath 'source/' and copy the instrumented Log.java and Instrumentation.java there. Please request the instrumented Log.java and Instrumentation.java from Alexander Egyed (aegyed@teknowledge.com). Edit the following files and insert the code between '`TEKNOWLEDGE INSTRUMENTATION BEGIN`' and '`TEKNOWLEDGE INSTRUMENTATION END`'.

MobileObject.java:

```
public void drawSelf (Graphics g, int screen_width, int screen_height) {
        Color redColor = Color.red;
        Color whiteColor = Color.white;
        Color blackColor = Color.black;
        ...
        g.setColor (redColor);
        g.fillOval ( x - rad, y - rad, 2 * rad, 2 * rad);
        g.setColor (whiteColor);

//TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.targetState(env.clock.simtime * env.clock.ticksize, "T"+name, (float) x_coord,
        (float)y_coord, (float)mdirection, (float)speed);
//TEKNOWLEDGE INSTRUMENTATION END

        // BCH
        FontMetrics fm = g.getFontMetrics();
        g.drawString(name, x - fm.stringWidth(name)/2, y + fm.getAscent()/2);
        // g.drawString (name, x - 10, y + 5);
        g.setColor (blackColor);
        g.drawString ("(" + (int) x_coord + "," + (int) y_coord + ")", x + 10, y);
    }
```

Sensor.java:

```
private void end_scanning() {
...

//System.out.println("Amp = " + aval + ", Freq = " + fval);
if ((aval > (AMP_THRESHOLD – sectorNoise[currentSector])) &&  //Make sure its big enough
    ((aval / max_amp) > alignment_val) &&  //Make sure its "aligned"
    (fval > freq_val) )
    {
freq_val = fval;
    }
    }
}
    }
    else
freq_val = -1;


// TEKNOWLEDGE INSTRUMENTATION BEGIN
float closestTargetDistance = Float.MAX_VALUE;
float closestTargetOrientationOffset = Float.MAX_VALUE;
for (int targ = 0; targ < env.mobile_objects.size(); targ++) {
    MobileObject mobj = (MobileObject) env.mobile_objects.get(targ);
double dx = xS – mobj.x_coord;
double dy = yS – mobj.y_coord;
double R = Math.sqrt(dx * dx + dy * dy);
    if (mobj.isMoving() &&  closestTargetDistance > ((float)R) )
    {
double diff = Math.abs((180/Math.PI*atan(-dy, -dx)) – (orientation[currentSector]));
double phi = Math.min(diff, 360 – diff);
closestTargetDistance = (float) R;
closestTargetOrientationOffset = (float) phi;
    }
}
System.out.println("Measurement:                    "+amp_val+"/"+freq_val+"                    ---
        "+closestTargetDistance+"/"+closestTargetOrientationOffset);
Instrumentation.sensorMeasurement(-1, "s" + nodeID, amp_val, freq_val, currentSector, amp_val,
        freq_val, closestTargetDistance, closestTargetOrientationOffset);
// TEKNOWLEDGE INSTRUMENTATION END

// Build the results string:
```

```
    if (! do_take_amp_measurement() )
amp_val = -1;

    String scanResults = amp_val + " "
+ freq_val + " "
+ scanTimestamp + " "
+ freqDuration + " "
+ dataCollectionMode + " "
+ currentSector + " "
+ sampleSize + " "
+ gain + " ";
...
}

// TEKNOWLEDGE INSTRUMENTATION BEGIN
public double atan(double y, double x)
{
   double retval;
   retval = Math.atan2(y, x);
   return (retval < 0 ? 2 * Math.PI + retval : retval);
}

public double toDegrees(double radians)
{
   return 180 / Math.PI * radians;
}
// TEKNOWLEDGE INSTRUMENTATION END
```

HARDWARE INFRASTRUCTURE

Replace the file Log.java in 'runtime/util/' with an instrumented Log.java. Also add the file Instrumentation.java there. Please request the instrumented Log.java and Instrumentation.java from Alexander Egyed (aegyed@teknowledge.com). Edit the following files and insert the code between '`TEKNOWLEDGE INSTRUMENTATION BEGIN`' and '`TEKNOWLEDGE INSTRUMENTATION END`'.

MTIRadarServiceThread.java:

```
...
package runtime.driver;
...
//TEKNOWLEDGE INSTRUMENTATION BEGIN
import java.net.*;
import java.io.*;
import track.*;
//TEKNOWLEDGE INSTRUMENTATION END
...
public class MTIRadarServiceThread extends ServiceThread {
...
  private LinkedList _observers;

// TEKNOWLEDGE INSTRUMENTATION BEGIN
NodeState _nodeState = null;
// TEKNOWLEDGE INSTRUMENTATION END
...
  public MTIRadarServiceThread(int ID) throws AddressUnavailableException {
    super("MTIRadarServiceThread", ID);

    _driver = MTIRadarDriver.get();
    _stats = new Statistics();
    _active = new boolean[MTIRadar.SECTOR_COUNT];
    _observers = new LinkedList();
    _measurement = new MTIRadarMeasurement();

// TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.connectGTServer(LocalNode.get().getID());
_nodeState = Instrumentation.parseConfigFileForNodeState(LocalNode.get().getID());
if (Instrumentation.isGTServerConnected() && _nodeState != null)
        System.out.println("Ground    Truth    Feedback    enabled    >>>ID"+LocalNode.get().getID()+"
        "+_nodeState);
// TEKNOWLEDGE INSTRUMENTATION END
  } // constructor
...
  public void run() {
...
    for (;;) {
```

```
      // if no measurements, block till a command arrives
      if (_measurementCount == 0 || _observers.isEmpty())
        processMessage(receive());
      // otherwise, try to measure
      else {

        startMeasurement();

//TEKNOWLEDGE INSTRUMENTATION BEGIN
Log.radarBeginScanning("s" + LocalNode.get().getID(), LocalNode.getLocalTime());
//TEKNOWLEDGE INSTRUMENTATION END

        // block and process messages while data /won't/ be AV
        start = LocalNode.getLocalTime();
        stop = start + _driver.getMinTime();
        for (current = start;
             current < stop;
             current = LocalNode.getLocalTime())
          if ((message = receive(stop - current)) != null)
            processMessage(message);

        /*
          keep looking for the measurement until the maximum time plus some
          extra value pass by.  The extra time exists in case time constants
          are off.  After that, the measurement definately exists or the
          command failed.
        */
        stop = start + _driver.getMaxTime() + 500; // half second
        while (!_driver.dataReady() && current < stop) {
          if ((message = receive(POLL_INTERVAL)) != null)
            processMessage(message);
          current = LocalNode.getLocalTime();
        } // while

        finishMeasurement();

//TEKNOWLEDGE INSTRUMENTATION BEGIN
Log.radarEndScanning("s" + LocalNode.get().getID(), LocalNode.getLocalTime());
//TEKNOWLEDGE INSTRUMENTATION END
        // decrement mesurement count
        if (_measurementCount != MTIRadar.MEASUREMENTS_INFINITE && _measurementCount != 0)
          _measurementCount--;
      } // else
    } // for
  } // run
...
// TEKNOWLEDGE INSTRUMENTATION BEGIN
        public double atan(double y, double x)
        {
            double retval;
            retval = Math.atan2(y, x);
            return (retval < 0 ? 2 * Math.PI + retval : retval);
        } // atan

        public double toDegrees(double radians)
        {
            return 180 / Math.PI * radians;
        } // toDegrees
// TEKNOWLEDGE INSTRUMENTATION END

  private void finishMeasurement() {
    float aval[], amp, freq;
    byte[] buffer;
    Message message;
    ListIterator iter;
    Address observer;

    int i, j;
    /*
      if no data was returned, set both measurements to -1.  Otherwise, process
      and return the real data.
    */
    aval = _driver.getData();
    if (aval == null) {
      amp = -1;
      freq = -1;
    } // if
    else {
      switch (_measurement.mode) {
```

```
      case MTIRadar.MODE_AMP:
         amp = aval[0];
         freq = -1;
         break;

      case MTIRadar.MODE_FREQ_DURATION:
         amp = -1;
         freq = aval[0];
         break;

      case MTIRadar.MODE_FREQ_INTERVAL:
         amp = -1;
         freq = aval[0] / 10;
         break;

      case MTIRadar.MODE_AMP_FREQ_INTERVAL:
         amp = aval[0];
         freq = aval[1] / 10;
         break;

      case MTIRadar.MODE_FFT:
         MTIRadarDriver.doFFT(aval, _measurement.sampleSize);
         amp = aval[0];
         freq = aval[1];
         break;

      default:
         Debug.die("MTIRadarServiceThread: unknown mode: " + _mode);
         // shush compiler
         amp = freq = -1;
      } // switch
    } // else

// TEKNOWLEDGE INSTRUMENTATION BEGIN
         float correctAmp = _nodeState.background[_current];
         float correctFreq = 0.0f;
             float perfectAmp = -1.0f, perfectFreq = -1.0f;
             float oldAmp = amp;
             float oldFreq = freq;
         _nodeState.currentSector = _current;
         if (Instrumentation.isGTServerConnected() && _nodeState != null)
         {
             Enumeration e = Instrumentation.getGTServerTargets();
             while (e.hasMoreElements())
             {
                 TargetState targetState = (TargetState) e.nextElement();
                 double                                  speed                                                   =
         Math.sqrt(targetState.Vx*targetState.Vx+targetState.Vy*targetState.Vy);
                 if (speed > 0.1)
                 {
                     double dx = _nodeState.x - targetState.x;
                     double dy = _nodeState.y - targetState.y;
                     double R = Math.sqrt(dx * dx + dy * dy);
                     double RDOT_TO_F = 19; // was 18.9;
                     double rdot = ((targetState.x - _nodeState.x) * targetState.Vx + (targetState.y -
         _nodeState.y) * targetState.Vy) / R;
                     double  diff  =  Math.abs(toDegrees(atan(-dy,  -dx))  -  (_nodeState.orientation  +
         _current * 120));
                     double phi = Math.min(diff, 360 - diff);
                     double thisFreq = Math.abs(RDOT_TO_F * rdot);
             //System.out.println("FREQ: "+thisFreq+" "+rdot+" "+targetState.Vx+"/"+targetState.Vy);
                     double    thisAmp    =    (_nodeState.K[_current]    *    Math.exp(-1*(phi    *
         phi)/(_nodeState.BW[_current]*_nodeState.BW[_current])))             /             Math.pow(R,
         _nodeState.Gamma[_current])+_nodeState.background[_current];
                     if (thisAmp <30.0f)
                         thisFreq = 0.0f;

                     if (thisAmp>correctAmp)
                     {
                         correctAmp = (float) thisAmp;
                         correctFreq = (float) thisFreq;
                     }
                         if (closestTargetDistance > ((float)R))
                         {
                         closestTargetDistance = (float) R;
                         closestTargetOrientationOffset = (float) phi;
                         }
                 }
             }

                 perfectAmp = correctAmp;
```

```
                perfectFreq = correctFreq;

            //System.out.print("SensorMeasurement:  "+amp+"  /  "+freq  +  "  ===>  "+correctAmp+"  /
        "+correctFreq+" = ");
            //only mix if GT server connection exists
            switch (_measurement.mode)
            {
            case MTIRadar.MODE_AMP:
                amp       =       amp*(1.0f-Instrumentation.getGTServerMixRatio()[_current])       +
        correctAmp*Instrumentation.getGTServerMixRatio()[_current];
                freq = -1;
                break;
            case MTIRadar.MODE_FREQ_DURATION:
                amp = -1;
                freq      =       freq*(1.0f-Instrumentation.getGTServerMixRatio()[_current])      +
        correctFreq*Instrumentation.getGTServerMixRatio()[_current];
                break;
            case MTIRadar.MODE_FREQ_INTERVAL:
                amp = -1;
                freq      =       freq*(1.0f-Instrumentation.getGTServerMixRatio()[_current])      +
        correctFreq*Instrumentation.getGTServerMixRatio()[_current];
                break;
            case MTIRadar.MODE_AMP_FREQ_INTERVAL:
                amp       =       amp*(1.0f-Instrumentation.getGTServerMixRatio()[_current])       +
        correctAmp*Instrumentation.getGTServerMixRatio()[_current];
                freq      =       freq*(1.0f-Instrumentation.getGTServerMixRatio()[_current])      +
        correctFreq*Instrumentation.getGTServerMixRatio()[_current];
                break;
            case MTIRadar.MODE_FFT:
                amp       =       amp*(1.0f-Instrumentation.getGTServerMixRatio()[_current])       +
        correctAmp*Instrumentation.getGTServerMixRatio()[_current];
                freq      =       freq*(1.0f-Instrumentation.getGTServerMixRatio()[_current])      +
        correctFreq*Instrumentation.getGTServerMixRatio()[_current];
                break;
            default:
                Debug.die("MTIRadarServiceThread: unknown mode: " + _mode);
            }
            System.out.print("SensorMeasurement: ");
            System.out.println(""+amp+" / "+freq);
        }

Instrumentation.sensorMeasurement(-1, "s" + LocalNode.get().getID(), amp, freq, _current, perfectAmp,
        perfectFreq);
// TEKNOWLEDGE INSTRUMENTATION END

         // generate measurement
    _measurement.amplitude = amp / _gain;
    _measurement.frequency = freq;

    /*
      send message to all observers.  The message is created first so that
      unnecessary copies are not made
    */
    _measurement.timeMeasured = LocalNode.getLocalTime();
    buffer = _measurement.toBytes();
    message = new Message(getAddress(),
                   null,
                   Message.TYPE_MTI_RADAR_MEASUREMENT,
                   buffer);
    iter = _observers.listIterator();
    while (iter.hasNext()) {
      observer = (Address)iter.next();
      message.setDestination(observer);
      transmit(message);

    } // while
  } // finishMeasurement
...
```

TRACKER

Instrumentations exist for both the SC Tracker from the University of South Carolina and the BAE Tracker from BAE. The following shows how to instrument the tracker using the example of the SC Tracker. Note that the instrumentation of the BAE Tracker is very similar. Edit the following

file and insert the code between 'TEKNOWLEDGE INSTRUMENTATION BEGIN' and 'TEKNOWLEDGE INSTRUMENTATION END'.

Tracker.java:

```
package track;

import java.io.*;
import java.util.*;
import java.awt.Point;
//TEKNOWLEDGE INSTRUMENTATION BEGIN
import control.*;
import runtime.util.*;
//TEKNOWLEDGE INSTRUMENTATION END

...
public class Tracker {

...
    public MotionModel motionModel = null;
    public int nLastUpdate = 0;

//TEKNOWLEDGE INSTRUMENTATION BEGIN
/**
 * Unique tracker name for the visualizer
 */
private static int trackerNumber = 1;
private String trackerName=null;
//TEKNOWLEDGE INSTRUMENTATION END

...
    public Tracker() {

//TEKNOWLEDGE INSTRUMENTATION BEGIN
trackerName = "SC"+Node.getID()+"-"+trackerNumber++;
Instrumentation.initTracker(-1, trackerName, "S"+Node.getID());
//TEKNOWLEDGE INSTRUMENTATION END
...

    }
...
    public Tracker(int width, int height, float gridSize) {
        String filename;

//TEKNOWLEDGE INSTRUMENTATION BEGIN
trackerName = "SC"+Node.getID()+"-"+trackerNumber++;
Instrumentation.initTracker(-1, trackerName, "S"+Node.getID());
 //TEKNOWLEDGE INSTRUMENTATION END
...

    } // constructor
...
    public void setNodeStates(Object[] states) {
        int i;
        double base;

        if (states == null)
            throw new IllegalArgumentException("bad states: null");
        for (i = 0; i < states.length; i++)
            if (states[i] == null || !(states[i] instanceof NodeState))
             throw new IllegalArgumentException("bad state " + i + ": " +states[i]);

//TEKNOWLEDGE INSTRUMENTATION BEGIN
String[] sensors = new String[states.length];
for (i=0; i<states.length; i++)
{
        NodeState nodeState = (NodeState) states[i];
        sensors[i] = new String(""+nodeState.ID);
}
Instrumentation.trackerSetMeasurementSensors(-1, trackerName, sensors);
//TEKNOWLEDGE INSTRUMENTATION END

        _nodeStates = states;


        // determine base and modified measurement weights
        base = Math.sqrt(states.length) / states.length;
        _locationWeight = LOCATION_WEIGHT * base;
        _velocityWeight = VELOCITY_WEIGHT * base;
```

```
...
    } // setNodeStates

...
    public void newAmpFreq(int measuredTime, int id, float measuredAmp, float measuredFreq, int
        measuredSector, int measuredGain) {
        NodeState node = null;
        node = get(id);
        if(node == null)
            return;

        int oldSector = node.currentSector;
        node.currentSector = measuredSector;
        this.newAmp(id,measuredTime,measuredAmp);
        this.newFreq(id,measuredTime,measuredFreq);
        node.currentSector = oldSector;

//TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.trackerNewAmplitude(-1, trackerName, measuredAmp, id, measuredSector);
//TEKNOWLEDGE INSTRUMENTATION END

//TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.trackerNewFrequency(-1, trackerName, measuredFreq, id, measuredSector);
//TEKNOWLEDGE INSTRUMENTATION END
    } // newData
...
    private void outputCurrentPredictions(TimeFrameData tf, int time1){
...
        TargetState target = new TargetState(x,y,velx,vely,time1);//tf.nPredTime);
        target.velFromFreq.x = velxFromFreq;
        target.velFromFreq.y = velyFromFreq;

        this._track.addElement(target);
        if(this.vTimeFrameData.size() > 3){
            TimeFrameData outTf = (TimeFrameData)vTimeFrameData.elementAt(this.vTimeFrameData.size()-
        3);
            Tracker.outputFile.println(outTf);
        }

//TEKNOWLEDGE INSTRUMENTATION BEGIN
double speed = Math.sqrt( (double) (velx * velx + vely * vely) );
double direction = 0.0;
if (speed > 0)
   direction = 180.0 / Math.PI * Math.acos( ((double)velx) / speed );
Instrumentation.trackerNewProjection(-1,  trackerName,  (float)  x,  (float)  y,  (float)direction,
        (float)speed );
//TEKNOWLEDGE INSTRUMENTATION END

        //cleaning the vector for memory
        try{
            if(vTimeFrameData.size() > 10){
            for(int i = 0; i < 5; i++)
                vTimeFrameData.removeElementAt(0);
            ((TimeFrameData)(vTimeFrameData.elementAt(0))).prevTimeFrame = null;
            }
        }catch(Exception e){};
    }
...
        public void newAmp(int id, int time, float amp) {
            NodeState node;
            boolean newData = false;
            node = get(id);
            if(node == null){
                System.out.println("Node doesnt exist in newAmp of tracker ...");
                return;
            }
            node.amp = Math.max(0, amp - node.background[node.currentSector]);
            node.lastAmp = time;

            dataFile.println("AMP\t" + id + "\t" + time + "\t" +(time+node.clockSkew)+"\t"+ amp
                        +"\t"+node.currentSector+"\t"+node.background[node.currentSector]);
            dataFile.flush();
            if (node.amp > 0) {
//TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.trackerNewAmplitude(-1, trackerName, amp, node.ID, node.currentSector);
//TEKNOWLEDGE INSTRUMENTATION END

                vMeasurements.addElement(new AmpFreqMeas("AMP",
                                node.ID,node.lastAmp,node.amp,
```

```
                                node.currentSector));
            } // if
        } // newAmp


        public void newAmp(int id, int time, float amp, int sector) {
            NodeState node;
            boolean newData = false;
            node = get(id);
            if(node == null){
                System.out.println("Node doesnt exist in newAmp of tracker ...");
                return;
            }
            node.amp = Math.max(0, amp - node.background[node.currentSector]);
            node.lastAmp = time;

            dataFile.println("AMP\t" + id + "\t" + time + "\t" + amp
                        +"\t"+node.currentSector+"\t"+node.background[node.currentSector]);
            dataFile.flush();
            if (node.amp > 0) {
//TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.trackerNewAmplitude(-1, trackerName, amp, node.ID, node.currentSector);
//TEKNOWLEDGE INSTRUMENTATION END

                vMeasurements.addElement(new AmpFreqMeas("AMP",
                                node.ID,time,node.amp,
                                sector));
            } // if
        } // newAmp

...
        public void newFreq(int id, int time, float freq) {
            NodeState node = null;
            boolean newData = false;
            node = get(id);
            if(node == null){
                System.out.println("Node doesnt exist in newFreq of tracker ...");
                return;
            }

            dataFile.println("FREQ\t" + id + "\t" + time + "\t" + freq
                    +"\t"+node.currentSector+"\t"+node.background[node.currentSector]);
            dataFile.flush();

            if (freq > 0) {
//TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.trackerNewFrequency(-1, trackerName, freq, node.ID, node.currentSector);
//TEKNOWLEDGE INSTRUMENTATION END

                node.freq = freq;
                node.lastFreq = time ;
                vMeasurements.addElement(new AmpFreqMeas("FREQ",
                                node.ID,node.lastFreq,node.freq,
                                node.currentSector));
            } // if
        } // newFreq

        public void newFreq(int id, int time, float freq, int sector) {
            NodeState node = null;
            boolean newData = false;
            node = get(id);
            if(node == null){
                System.out.println("Node doesnt exist in newFreq of tracker ...");
                return;
            }

            dataFile.println("FREQ\t" + id + "\t" + time + "\t" + freq
                    +"\t"+node.currentSector+"\t"+node.background[node.currentSector]);
            dataFile.flush();

            if (freq > 0) {
//TEKNOWLEDGE INSTRUMENTATION BEGIN
Instrumentation.trackerNewFrequency(-1, trackerName, freq, node.ID, node.currentSector);
//TEKNOWLEDGE INSTRUMENTATION END

                node.freq = freq;
                node.lastFreq = time;
                vMeasurements.addElement(new AmpFreqMeas("FREQ",
                                node.ID,node.lastFreq,node.freq,
                                sector));
            } // if
```

```
        } // newFreq
} // Tracker
```

DATA PROTOCOL FOR EXCHANGING INSTRUMENTED DATA

Data about the software and hardware infrastructure is gathered through instrumentation code that is inserted into the infrastructure. This instrumentation code sends observed data to the visualizer using the data protocol discussed below. The data is send it UTF format which typically is a string that starts with an unique name and terminates with the dollar symbol ('$').

VERSION time version $ time…long, version…integer Defines the protocol version.
INIT_ROOM time width height $ time…long, width…float, height…float Initializes a room for experimentation at sets its size.
TERM_ROOM time $ Terminates a room.
SENSOR_ACTIVATE_RECEIVER time name $ time…long, name…string Update that the sensor is started listening to the receiver of the communication channel.
SENSOR_DEACTIVATE_RECEIVER time name $ time…long, name…string Update that the sensor is stopped listening to the receiver of the communication channel.
SENSOR_SET_RECEIVER_CHANNEL time name channel $ time…long, name…string, channel…integer Update what receiver channel the sensor is listening to.
SENSOR_SET_TRANSMITTER_CHANNEL time name channel $ time…long, name…string, channel…integer Update what transmitter channel the sensor is writing to.
SENSOR_MESSAGE_RECEIVED time name source length contents $ time…long, name…string, source…string, length…integer, message…string Update that a message was received by sensor 'name'. The message was sent by 'source' and has a certain length and contents.
SENSOR_MESSAGE_TRANSMITTED time name destination length contents $ time…long, name…string, source…string, length…integer, message…string Update that a message was transmitted by sensor 'name'. The message was sent to 'destination' and has a certain length and contents.
SENSOR_ACTIVATE_SECTOR time name sector $ time…long, name…string, sector…integer Update that the sensor has switched to a given sector.
SENSOR_DEACTIVATE_SECTOR time name sector $ time…long, name…string, sector…integer Update that the sensor has switched away from a given sector.
SENSOR_BEGIN_SCANNING time name sector $ time…long, name…string, sector…integer Update that the sensor has started taking measurements at a given sector.
SENSOR_END_SCANNING time name sector $ time…long, name…string, sector…integer Update that the sensor has stopped taking measurements at a given sector.
SENSOR_SET_LOCATION time name x y $

time…long, name…string, x…long, y…long Update on the current location of the sensor.
SENSOR_SET_ORIENTATION time name orientation $ time…long, name…string, orientation…float Update on the current orientation of the sensor relative a nominal axis.
SENSOR_SET_CURRENT_SECTOR time name sector $ time...long, name…string, sector…integer Update on the currently selected sensor head of the sensor.
SENSOR_SET_DATA_COLLECTION_MODE time name mode $ time…long, name…string, mode…integer Update on the current data collection model of the sensor.
SENSOR_SET_MEASUREMENT_DURATION time name duration $ time…long, name…string, duration…integer Update on the current duration of measurements taken at the sensor.
SENSOR_SET_SAMPLE_SIZE time name size $ time…long, name…string, size…integer Update on the current size of measurements taken at the sensor.
SENSOR_SET_GAIN time name gain $ time…long, name…string, gain…integer Update on the current gain of the sensor.
SENSOR_GT_MIX_RATIO time name head ratio $ time…long, name…string, head…string, ratio…integer Update on the current mix ratio of 'real' measurements versus 'perfect' measurements. The head is a string indicating the sensor head number. If the string is 'ALL' then the update applies for all sensor heads. If the ratio is 0 then the measurements are real. The ratio is at most 100 which indicates that the measurements are perfect.
INIT_SENSOR_CALIBRATION time name K0 Gamma0 BW0 N0 K1 Gamma1 BW1 N1 K2 Gamma2 BW2 N2 $ time…long, name…string, K0…float, Gamma0…float, BW0…float, N0…float, K1…float, Gamma1…float, BW1…float, N1…float, K2…float, Gamma2…float, BW2…float, N2…float Initializes a sensor's calibration values.
INIT_SENSOR time name x y orientation, recChannel transChannel duration size gain mode currentSector activeSectors $ Initializes a sensor with all relevant state information. See previous entries on individual state information.
TERM_SENSOR time name $ time…long, name…string Terminates a sensor.
TARGET_NEW_STATE time name x y direction speed $ time…long, name…string, x…float, y…float, direction…float, speed…float Update on the location, direction, and speed of a target.
TARGET_SET_LOCATION time name x y $ time…long, name…string, x…float, y…float Update on the location of a target.
TARGET_SET_DIRECTION time name direction $ time…long, name…string, direction…float Update on the direction of a target.
TARGET_SET_SPEED time name speed $ time…long, name…string, speed…float Update on the speed of a target.
INIT_TARGET time name x y direction speed $ time…long, name…string, x…float, y…float, direction…float, speed…float Initializes a target.
TERM_TARGET time name $ time…long, name…string Terminates a target.
TRACKER_NEW_AMPLITUDE time name amplitude origin sector $

time…long, name…string, amplitude…float, origin…string, sector…integer Update on amplitude measurement given to the tracker. Origin and sector describe sensor node and sensor head where measurement was taken.
TRACKER_NEW_FREQUENCY time name frequency origin sector $ time…long, name…string, frequency…float, origin…string, sector…integer Update on frequency measurement given to the tracker. Origin and sector describe sensor node and sensor head where measurement was taken.
TRACKER_NEW_PROJECTION time name x y direction speed $ time…long, name…string, x…float, y…float, direction…float, speed…float Update on the location, direction, and speed where a tracker presumes a target to be.
TRACKER_SET_MEASUREMENT_SENSORS time name measurementSensors $ time…long, name…string, measurementSensors Update on the sensors that supply measurements to the tracker.
INIT_TRACKER time name $ time…long, name…string Initializes a tracker.
TERM_TRACKER time name $ time…long, name…string Terminates a tracker.
INIT_AGENT time name $ time…long, name…string Initializes an agent.
TERM_AGENT time name $ time…long, name…string Terminates an agent.
END_INSTRUMENTATION $ Terminates an experiment.

DATA FORMAT FOR STORING INSTRUMENTED DATA

The visualizer can also gather data and store it persistently into a file. The data format also uses UTF strings and conforms exactly to the data protocol discussed in the previous section.

RE-CONFIGURING VISUALIZER, RADSIM2, AND HARDWARE

By default, the instrumentation is configured such that it expects the visualizer on the same host as the software/hardware infrastructure. This configuration is useful in small environments that are set up for experimentation purposes. In larger environments, it is generally useful to have the visualizer running on a dedicated machine. In such a case, the instrumentation needs to be notified of the host name and port number of the visualization machine. The file Instrumentation.java defines a variable where to look for the visualizer. For convenience, the variable allows multiple, potential locations to be defined: see variable 'hosts' and 'ports' below:

Instrumentation.java:

```
package runtime.util;
...
public class Instrumentation extends Object
{
    static Socket _socket=null;
```

```
    static DataInputStream _instream=null;
    static DataOutputStream _outstream=null;
    static String _host;
    static int _port;
    static boolean _errorOccured = false;
    static long _lastTime = 0;
    static boolean _wasInitialized = false;

        static String[] _hosts = new String[]{"10.0.0.9","10.0.0.11","10.0.0.101","10.0.0.10"};
        static int[] _ports = new int[]{9876,9876,9876,9876};
        static int _countHosts = 4;
        static int _retries =5;

        public static String _GTServerHost = "10.0.0.51";                                   //connect  to
        GTServer for Sensor feedback
        public static int _GTServerPort = 2000;
        public static DataInputStream _GTInstream = null;
        public static DataOutputStream _GTOutstream = null;
        public static Hashtable _targets = new Hashtable();
        public static float[] _sensorGTMixRatio = {0.0f,0.0f,0.0f};
        public static Thread _GTServerFeedbackThread=null;
        public static int _nodeID = -1;

        static String _configFile = "/localhome/mitreants/Visualizer/config.txt";

    /** Creates new Instrumentation */
    public Instrumentation() {}

    public static synchronized void init()
    {
        _wasInitialized = true;

            int j=0;
            do
            {
                for (int i=0; i<_countHosts; i++)
                {
                    _host = _hosts[i];
                    _port = _ports[i];
                    if (connect(_hosts[i], _ports[i]))
                    {
                        System.out.println("Visualizer is ENABLED (host "+_host+" "+_port+").");
                         return;
                    }
                }
                j++;
                int delay = new Random().nextInt(200);
        System.out.println("Retry to connect to Visualizer (2) = "+delay);
                try { Thread.sleep(delay); } catch (Exception ex) {}
            }
            while (j < _retries);

            if (_outstream==null)
            {
        System.out.println("Visualizer is DISABLED (run Visualizer to enable visualization).");
            }
    }
...
```

Recompile Instrumentation.java after making changes.

DESIGN INFORMATION

This section gives on overview on the design of the visualizer in context of the agents and RadSim2. The purpose of the system is to have multiple, intelligent (software) agents negotiate over the best use of available resources (radars) to track a series of targets. The targets and sensors can be simulated via the RadSim2 components or they can be observed via real sensors and targets. The following design discusses the visualizer in context of RadSim2 only.

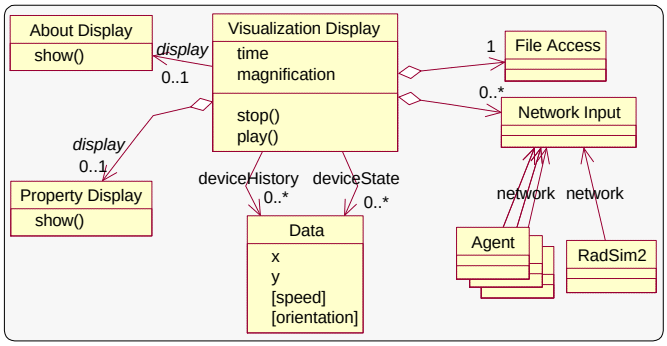


Figure 7. Design Class Structure of Visualizer only

The system's main components are Agent, RadSim2 (sensor and target simulator), and a real-time Visualizer (note: we omit the real hardware for brevity). These components communicate via a network where the Visualizer is on the receiving end and the other components are transmitting. There is also communication going on between Agents and RadSim2 that is omitted here. Figure 7 depicts the design structure of the Visualization component. The Visualizer has an input interface (class *Network Input*) to handle data coming from the network (Agents and RadSim2) and it has a storage facility for incoming data (class *Data*). A visualization component to display the incoming data (class *Visualization Display*), data properties (class *Property Display*), and the about box (classes *About Display*) make up the user interface. Data can be stored in a file (class *File Access*).

Figure 8 shows a refinement of Figure 7 depicting the real, implementation-level class structure of the Visualization component (implementation-level implies that there is a one-to-one mapping between the class structure and the actual code). A network is used to gather data in near real-time speed (class *DataInputStream*). Gathered data may come from one out of three different types of input devices: *Sensor*, *Target*, and *Tracker*. Sensors track targets and trackers use sensor data to estimate target location. All input devices have properties like a time stamp, a name, and a history of changes. Data received from devices is usually information about activities or state changes. Each data is stored in a new device instance and the class *Scenario* is responsible for keeping track of the current state of all devices (sensors, targets, and trackers) as well as their entire change history. The heart of the visualization component is the class *TrackFrame* which processes and visualizes the data. TrackFrame also has an user interface with elements like button, text area, or canvas to graphically display the devices and their states (i.e., device properties are visualized via icons and other graphical clues). A *PopupDialog* is used to display properties of a device.

The class structure in Figure 8 is already simplified since numerous methods and attributes are omitted. Also, the RadSim2 and Agent components are only indicated through single classes. Naturally, they are complex components themselves and consist of numerous classes. The notation used in Figure 7 and Figure 8 follows the UML standard for class structures [12]. Classes may have three types of properties relevant here: *attributes*, *methods*, and *relationships*. Attributes are variable declarations (e.g., *Target* defines the global variable 'x' of type float) and methods are computational functions (e.g., *TrackFrame* has a method called stopPlaying()). Relationships define how classes are connected. The UML relationships *generalization* (inheritance), *dependency*, and *association* are relevant here. A generalization defines inheritance and is indicated through a triangle head (e.g., *Tracker* inherits from *Device*). An association defines a semantic relationship between classes where the arrowhead indicates direction of calls (e.g., *Tracker* relates to 0, 1, or more *Sensors* to track a target; the arrowhead indicates that *Tracker* uses methods/attributes of *Sensor* but not vice versa). An association can be plain (solid line) or aggregate (solid line with diamond at one end). If an association is aggregate then the owner class created the owned class and the owned class may be expected to die

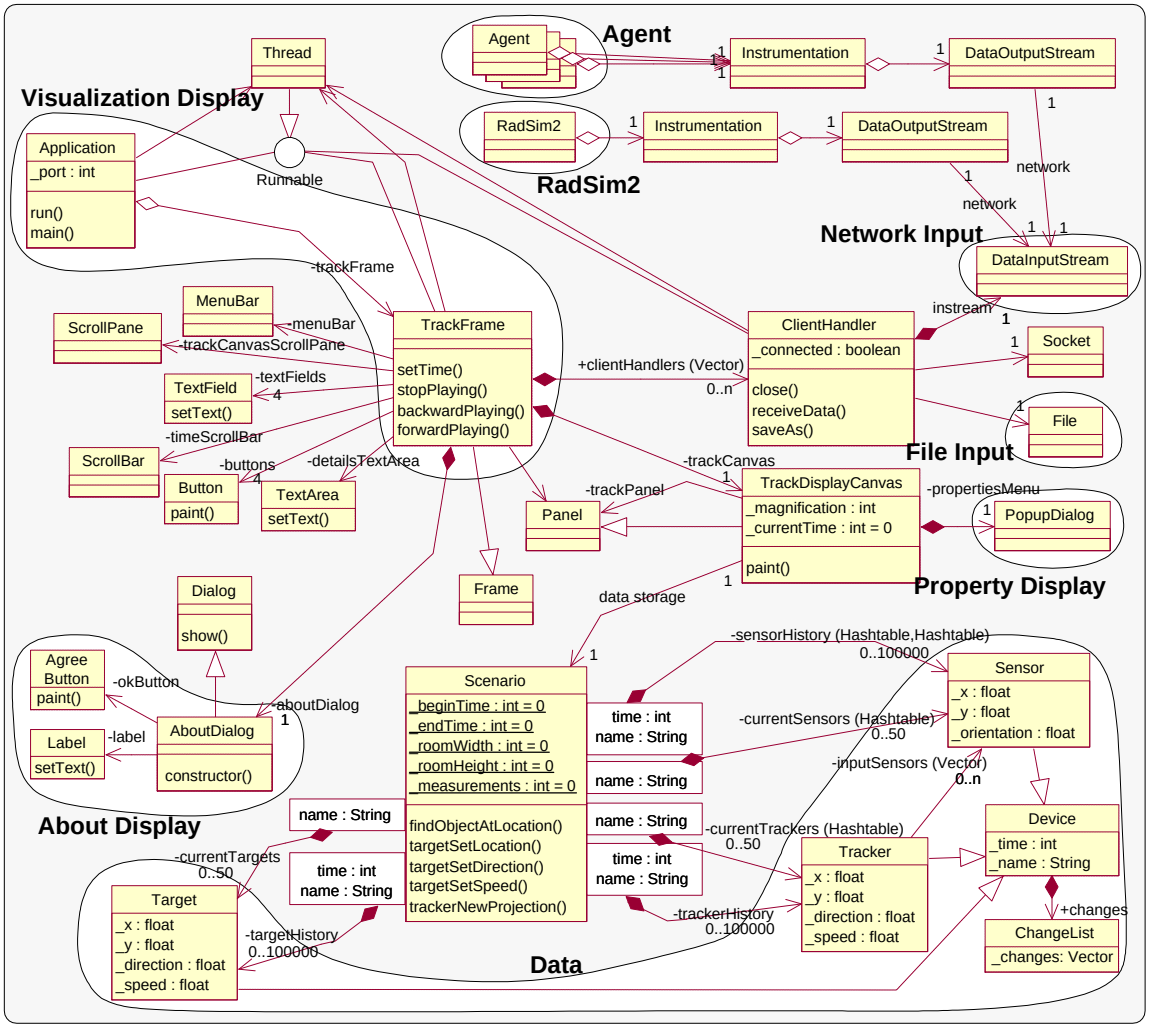


Figure 8. Implementation Class Structure of Visualizer only (shaded areas indicate clusters of design classes)

with the owner (diamond filled = owned class dies; diamond empty = owned class does not die). For instance, class *Scenario* owns instances of class *Sensor* and those instances will die once *Scenario* dies. Associations can be uni-directional (arrowhead) or bi-directional (no arrowheads); and they may have defined cardinalities. For instance, *Scenario* can access attributes/methods of *Sensor* but not vice versa (arrowhead) and *Scenario* may own 0 to 100000 instances of *Sensor*.

ANTS AGENT VISUALIZATION

Infrastructure visualization provides important functionality, not only because it is generally applicable to all programs developed using the infrastructure, but also because it allows different solutions to be evaluated using a common framework. What it does not provide, however, are solution-specific services. Different solutions to the same problem can and will use vastly different approaches, and general infrastructure-level visualization is unlikely to be able to capture and display

all the nuances and complexities which make these approaches interesting. Thus, while infrastructure visualization is good at describing how well an approach is working in general terms, agent-specific instrumentation and visualization is needed to capture, evaluate and debug how a specific approach functions.

To varying degrees, all of the teams involved in the ANTs project utilized instrumentation and visualization as part of their debugging cycle. Their effort is not discussed here. Please refer to documentation for agent visualizations to the documentation of agents teams.

CONTROLLING THE ENVIRONMENT

OVERVIEW

Implementing ANTs agents for an ideal world is already a non-trivial task, but the real-world environment of ANTs is further complicated by (1) unreliable, limited communication and (2) sensor measurement noise. Each sensor platform had built-in support for wireless RF (Radio) communication with the drawback of low bandwidth and a likelihood of message loss that increased with bandwidth usage. Additionally, the measured values produced by the sensors would deviate from their theoretical performance due to a combination of known and unknown external effects. Of course, these complexities are to be expected in any real-world environment. Nonetheless, it is hard to design and validate systems where the effects of noise are not well understood. In an effort to control noise for purposes of debugging and testing, a pair of mechanisms were added which could compensate for these effects.

The simulated infrastructure, Radsim, could mitigate both problems because it simulated the noise factors itself. This allowed developers to understand in what ways noise influenced agent behavior, and as such was an important tool when preparing agents for the hardware environment. However, like nearly every simulation, Radsim could not capture all of the complexities of the hardware system with perfect accuracy. For instance, the hardware infrastructure exhibited certain timing complexities (including non-determinism) that were not easily captured by Radsim. Consequently, it was also necessary to mitigate network and sensor noise in the hardware infrastructure, for validation purposes.

The RF network noise and bandwidth limitations were mitigated through the use of an alternative, high-bandwidth, reliable TCP network. Agents could then be tested and validated separately through both communication methods. The problem of sensor noise was somewhat harder to mitigate, but we found that the instrumentation technique used for visualization could serve dual purposes by providing "perfect" data that could be used to control the sensor noise problem. The train instrumentation provided target location information, which was then used to compute expected sensor measurements based on their theoretical model. These "noise-free" sensor measurements were then mixed with actual sensor measurements in real-time to provide measurements with a controlled amount of noise. These techniques proved useful in facilitating the transition of agent technologies from pure simulation to a hardware-only environment.