TRI: Bridging the Gap between Wireless Sensor Networks and Autonomous Agents

Todd Sullivan Dr. Yi Shang
Undergraduate Honors Thesis
University of Missouri-Columbia

Abstract

Wireless Sensor Networks (WSNs) offer vast amounts of real-time data about environments. These real-time data streams are an important resource for the ever-increasing number of autonomous agents. Robots can use WSN information to effectively extend their senses and gather data about regions that are not visible or immediately accessible. Additionally, software agents can aggregate WSN data for monitoring purposes such as environmental monitoring and intrusion detection. Despite advances in the field of WSNs, the development of most applications currently requires sensor-specific programming techniques. The research community, as well as industry, needs an efficient, convenient method for accessing WSN data through existing infrastructure such as intranets and the internet. This research project presents TRI, the TinyOS Robot Integration server. TRI is a multithreaded server that provides developers with WSN data management and agent-agent communication channels through a TCP/IP connection and a human-readable message protocol. The TRI server hides the details of retrieving data from and managing a WSN. Thus, developers with standard TCP/IP socket experience can incorporate WSNs into their projects. This research also presents TRI applications executing on a Sony AIBO that responds to its environment by its onboard sensors and the extra sensory data from a WSN.

1 Introduction

The field of Wireless Sensor Networks (WSNs) is a rapidly developing research field in computer science and computer engineering. As wireless sensors continue to shrink in size and increase in capabilities, many real-world applications in monitoring, tracking, and detection are becoming feasible. WSNs are ripe with information that external agents can use to learn more about their environment and thus make better decisions.

Despite advances in the field of WSNs, the development of most applications currently requires sensor specific programming techniques. These techniques include programming for specific operating systems such as TinyOS [1] and using APIs for solutions such as TinyDB [2] and TASK [3]. These programming specifics are a burden on application developers, especially when developing applications for external agents such as robots. To overcome these barriers, the research community needs a solution that removes these sensor-specific details and provides access to WSNs through common TCP/IP connections.

This paper presents *TRI*, a multithreaded server that provides developers with WSN data management and agent-agent communication channels through a TCP/IP connection and a human-readable message protocol. The main contributions of this paper are:

- The presentation of the TRI server including its design and implementation.
- The demonstration of a TRI application using a Sony AIBO that responds to its environment by its onboard sensors and the extra sensory data from a WSN.

2 Background

A sensor network can consist of hundreds or thousands of motes. These motes each run an operating system designed for low power consumption with limited processing and storage capabilities. The most popular motes for WSNs are Mica and Mica2, which run TinyOS. On top of TinyOS, TinyDB allows users to query the sensor network as if it were a database, while TASK builds upon TinyDB to allow for easy deployment of a sensor network for monitoring environments.

2.1 TinyOS

TinyOS is an application-specific operating system designed for low-power operation and event-centric concurrent applications. Developers create programs for motes using the NesC programming language, which is similar to C. The operating system uses a component-based programming model where developers wire components together to create an operating system for each specific application. TinyOS provides a core set of interfaces for accessing the LEDs, nonvolatile storage, timers, and various sensors.

Developing TinyOS applications requires programming experience in embedded devices. Patience is also required due to the nature of embedded programming and the lack of output devices that show the current state of variables and programs. These details are cumbersome for developers that are not interested in developing embedded device applications and are instead only interested in receiving sensory data from the WSN.

2.2 TinyDB

TinyDB is an application built upon TinyOS. TinyDB frees developers from programming within the TinyOS environment. Instead, the TinyDB mote application runs on all of the motes in a WSN network. Developers must write a Java program to access mote data or execute a command line program to receive specific data from the WSN.

TinyDB allows access to sensor data through TinySQL, which is a query language similar to SQL. TinySQL queries are of the form:

```
SELECT <aggregates>, <attributes>
[FROM {sensors | <buffer>}]
[WHERE <predicates>]
[GROUP BY <exprs>]
[SAMPLE PERIOD <const> | ONCE]
[INTO <buffer>]
[TRIGGER ACTION <command>]
```

An example of a query that retrieves each mote ID and light reading every second is: *SELECT nodeid, light FROM sensors SAMPLE PERIOD 1024;*.

While TinyDB is a much better solution for developers wishing to receive data from sensor networks, it is still cumbersome since developers must write a program in Java that uses the TinyDB Java library to manage queries to the WSN. Ideally, a developer should be able to connect a program such as an autonomous agent to a server that handles all query processing details. Instead of making each agent manage the process of requesting and receiving data from TinyDB, each agent should simply be able to connect through a TCP/IP connection to a server that manages TinyDB tasks.

2.3 TASK

TASK, the Tiny Application Sensor Kit, is a suite of tools designed to ease deployment of sensor network applications for non-sophisticated users. TASK consists of TinyDB, the TASK Server, PostgreSQL, TASK client tools, and the TASK Field Tool. The sensor network runs the TinyDB mote application. The Task Server acts as a manager between the WSN and the internet. The TASK Server uses the PostgreSQL database to store query and mote deployment information. The TASK client tools allow users to visualize sensor readings, create TinyDB queries that execute on the sensor network, and record mote deployment data such as locations and names of motes in the deployment area. The TASK Field Tool is a PDA application that helps users diagnose problems while in the field.

TASK is primarily for environmental monitoring. Its focus on ease of use allows researchers and WSN end users in areas such as agriculture to leverage the potential of WSNs for monitoring purposes. TRI's design learns from the results of TASK development. Instead of focusing on providing an easy-to-use tool for environmental monitoring, TRI focuses on providing a server application for agent communication and agent access to a WSN.

3 TRI Server Model

The TRI server allows agents to gather data from a WSN, read past data from the server's database, and communicate with other agents using a human-readable message format. The sensor network's motes run TinyDB, which the server uses to execute queries. The server stores agent information and query data in a MySQL database. The server is multithreaded and has no programming-related limit on the number of agents that can connect, communicate, and execute queries. Appendix A includes the applicable commands and responses for the TRI server.

3.1 Agent-Agent Communication

When agents connect to the server, they must register themselves. After registering, agents can receive a list of other agents connected to the server. Agents can communicate through 1 to 1 messages and 1 to N messages. Agents can also receive notices when other agents join or exit the server.

Agents communicate with one another through the send command, which has the format *send!#!agent_name!#!message* where *agent_name* is the registered name of the recipient and *message* is the message to be sent. The server parses the command as a string, but the message section can be any format that the agents mutually agree upon. For example, an agent can send binary data, XML data, or comma-separated data in the *message* section.

3.2 Agent-WSN Communication

Through the server, agents can create, start, stop, and listen to queries on the TinyDB network. Additionally, agents can choose to have queries log data to the MySQL database, monitor which queries are running on the TinyDB network, view which other agents are listening to each query, and listen to old query data from the MySQL database that is replayed in real-time.

3.3 Query Logging

The server records all agent and query activity. Recorded activities include agent login times, query data, query start and stop times, agent listen start and stop times for each query, and query details such as the query's TinySQL, description, and creator. Query data is only logged if at least one agent requests the server to log a specific query's results.

Agents can turn query logging on and off when starting a query or at any time while a query is running. Agents can choose whether queries continue to execute and record data after all listening agents stop listening. Thus, the server can continuously record a query's data to the database while agents come and go as listeners.

3.4 Real-Time Replay

One advantage of recording query data to a database is that agents can retrieve the histories and make decisions based on these histories. Another advantage of storing query data is that the server can replay the data back to an agent as if it is happening in real time. This is most useful when testing algorithms with a consistent set of data.

For example, suppose that a developer is working with an autonomous robot and is testing an algorithm. The developer needs to test the algorithm for a specific type of input from the sensor network. With the TRI server, the developer can create the desired effects in the sensor network once and have the readings stored in the database. When testing the algorithm, the server will replay the entire timeframe back to the robot as if the event was happening again. Since the data that the robot receives is the same every time, the developer can quickly adjust the algorithm for testing purposes.

An additional advantage of the real-time replay feature is that developers can easily create demonstrations for presentations that provide the robot with the correct feedback each time. This is useful when presenting research to an audience and when explaining an algorithm or concept to a group. In the latter case, the presenter or teacher can pause the replay at any time and demonstrate the topic in steps.

4 Implementation

The TRI server is a Java application that requires JDK 1.4.1 since the TinyDB Java library does not work with JDK 1.5. The server consists of six main parts: AbstractServer, triServer, QueryManager, Query, AgentHandler, and DBLogger. Figure 4a depicts these six implementation layers.

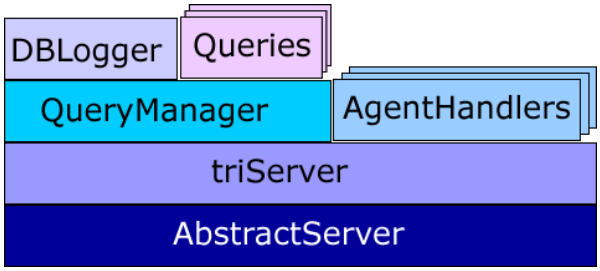

Figure 4a: The six implementation layers.
Each layer is built from the layers below it.

4.1 AbstractServer

The AbstractServer is a multithreaded server that listens for incoming requests and efficiently manages a group of connections by maintaining a pool of threads. The AbstractServer manages the group of connections through a request queue, which handles each request when a thread from the pool is available. This component has a minimum and maximum possible thread count for the pool. The thread count restrictions are specified when instantiating an AbstractServer object.

The AbstractServer performs all of the tasks involved in managing the connections for the server. This allows the layers above the AbstractServer to focus on application-specific details. As the name suggests, the AbstractServer is an abstract class that triServer extends.

The AbstractServer requires triServer to supply a request handler class for handling requests from connections. Within the constructor of triServer, the AbstractServer receives the AgentHandler class as the request handler. Thus, whenever the AbstractServer receives a new connection or message from a connection, the information is passed to the associated AgentHandler. The AbstractServer uses the thread pool to execute the AgentHandler instances.

4.2 triServer

The triServer acts as the bridge between the AgentHandlers and the QueryManager. The triServer maintains a map of agent names to AgentHandlers and provides the shared functions for the QueryManager and AgentHandlers. These shared functions pertain to adding agents to the system, sending preprocessed query data to agents, sending messages to agents, and removing agents.

The AgentHandlers initiate the majority of the code execution within the triServer. Since the AgentHandlers are managed by a pool of threads, the majority of the functions and data structures within the triServer are synchronized to avoid access from multiple threads at the same time. Additionally, the triServer contains the main function and is thus the class to compile and execute when starting the TRI server.

4.3 QueryManager

The QueryManager manages the queries as Query objects, initiates database activity through the DBLogger, and uses the TinyDB Java library to control the WSN. The QueryManager contains a map of query names to Query objects. The layer also maintains a generator for TinyDB IDs so that new queries into the WSN do not override older running queries.

The QueryManager provides the functionality for creating, starting, stopping, modifying, and listening to queries. The QueryManager also provides status information such as the number of running queries and query listeners. The QueryManager acts as the bridge between the DBLogger and the Query objects.

4.4 Query

The Query object manages a single query that is active in the TinyDB network. Each object includes information about the specific query such as its name, creator, description, start time, and TinyDBQuery object. The Query object maintains a list of listeners and sends query data to each listener when the TinyDB Java library sends a query result to the object's addResult method. The Query object also sends query data to the DBLogger if the query's database logging is set to true.

When removing listeners through the *stopquery* command, the Query object checks to see if the query should terminate. The object will tell the QueryManager to destroy the Query object if there are no additional listeners and the query is either not logging to the database or the last agent to leave sent the kill database flag. Appendix A outlines the kill database flag under the *stopquery* command. If a listener removes itself using the *stoplistentoquery* command then the Query object does not perform these checks.

If the Query object is a real-time replay instead of a TinyDB query then the Query object manages the histories and serves each history to the listeners at the appropriate time. The object achieves this through an inner class called resultsTask, which includes a timer that determines when to send each result to listeners. The timer determines which results to send by using each result's epoch value and keeping track of time relative to the when the query data was originally receiving data while recording the query to the database. TinyDB provides the epoch value from the motes for each set of query data. The server includes the epoch value and receiving time on the server when recording query data to the database.

4.5 AgentHandler

The AgentHandler object manages a single connection with an agent. The AgentHandler sends messages from the server, WSN, and other agents to its associated agent. The primary function of the AgentHandler is the handleCommand routine, which parses a message from the agent and performs the appropriate command.

The AgentHandler parses each message by splitting the string into segments based on the delimiter "!#!". In the case of a *send* command, the handler reconstructions the message with the delimiters intact so that the message is not mangled when sent to the receiving agent. If the message has any errors in syntax or the command does not execute properly, the AgentHandler will send the agent an appropriate error message as outlined in Appendix A.

4.6 DBLogger

The DBLogger manages the MySQL connection, writes all data to the database, and loads queries from the database into Query objects. When creating a new query, the DBLogger creates a new table in the database for the query's data. All tables in the database use the InnoDB engine. All methods within DBLogger return status codes so that other layers can report errors to the agents. The DBLogger records all activity by using prepared statements and result sets.

5 Application: Sony AIBO

This section demonstrates an application of the TRI server using a Sony AIBO. The AIBO programs are written in C++ and use the Tekkotsu framework [4]. In each demonstration, the AIBO reacts to the light level from each mote in the WSN. Due to resource constraints, the test environment includes one Sony AIBO and a WSN consisting of two Mica2 motes that sense sound, light, temperature, and pressure levels.

The AIBO programs use a shared triManager program for communication with the TRI server. The triManager program contains functions for saving agent data to memory and nonvolatile storage, sending commands to the TRI server, and registering TRI queries. The first TRI AIBO program to execute instantiates the triManager.

The AIBO programs are Head Movement, Sleep-Sit-Stand, and Walk to Light. In Head Movement, the AIBO lies down and moves its head in response to the average light from the WSN. In the brightest light reading, the AIBO's head is at its maximum upright position. In the darkest setting, the AIBO's head moves to the lowest position between its front legs.

Sleep-Sit-Stand is similar to Head Movement. In the darkest setting, the AIBO lies on the ground with its head in-between its front legs. As the environment's light level increases, the AIBO begins to sit up and then stand. The AIBO chooses its target position from the

full range of positions based on the average light reading.

In Walk to Light, the AIBO walks to the mote that senses the highest light intensity. The program only works with two sensing motes. The program assumes that bright pink designates mote 1 and bright green designates mote 2. When searching for the target mote, the AIBO turns counterclockwise until it sees the correct color. Once the program finds the target, the AIBO positions itself in front of the object.

5.1 Implementation

While TinyDB supports aggregation of values across the motes in the WSN, the AIBO programs do not use this feature. The AIBO programs do not use TinyDB's aggregation features when determining the average light reading from the WSN because TinyDB's aggregation is unreliable when the WSN contains only two motes. Due to interference, TinyDB does not always receive data from all motes during each epoch.

In the case of determining the average light value, if only one mote has data available, then that mote's data is the average. This causes problems, for example, when one of the motes has a light reading of 0 and the other has a light reading of 1,000. Instead of always returning an average of 500, the average fluctuates each epoch between the values 0, 500, and 1,000.

To remedy this issue, the AIBO instead receives each mote's light reading for each epoch. During each iteration, the AIBO computes the average based on the last received values from each mote. This technique significantly wastes bandwidth and computations when the number of motes in the WSN is large, but for the two-mote implementation, the technique suffices.

Additionally, all of the AIBO programs are templated classes that receive an integer called rate. Rate is the number of milliseconds to use in the sampling period section of the TinySQL statements. Thus, one can easily run the demonstrations at various sampling rates.

5.1.1 triManager

The triManager maintains agent data through a standard template library map of strings to strings. The triManager's initialization phase loads the file *tri.cfg*, which contains name-value pairs separated by a colon on each line, and parses it to create the map of agent data. *tri.cfg* must define the agent's name, and the TRI server's IP and port. If the configuration file does not define the required agent data then the triManager uses the default values *Scooter*, *192.168.1.2*, and *11223* respectively. All programs that use the triManager can define additional agent data through the use of the *ad*, *saveAgentData*, and *removeAgentData* functions.

The triManager manages the connection with the TRI server. This includes registering the agent, forwarding commands from AIBO programs, and parsing results from the TRI server. The triManager forwards messages from AIBO programs through a standard *printf* function. The triManager only forwards messages to the server if the agent is successfully registered with the server.

The triManager parses results from the TRI server by translating each message into a map of strings to strings. Appendix B includes the details of the map in relation to each possible response from the TRI server. After generating each result's map, the triManager forwards the map to all listeners. Thus, multiple AIBO programs executing on the same robot can concurrently use the triManager and access the WSN.

When sending query data to listeners, the triManager by default names the fields in the result row *field_0, field_1, field_2, ...* in accordance with the order that the fields are specified in the query's TinySQL. For ease of use, the triManager allows AIBO programs to register query fields through the registerQueryFields function. Registering a query's fields allows an AIBO program to designate names for each field. If an AIBO program register's a query's fields, then the query data's result maps will include the designated names instead of *field_0, field_1*, etc.

5.1.2 Head Movement

Head Movement, known as the triHeadLightAvgAIBOAgg class within Tekkotsu, can be visualized as a simple two-node state machine as shown in Figure 5.1.2a. During the initialization phase, the program loads the "sleeping" position, registers itself to receive sensory information about completed movements, and registers with the triManager. Once the AIBO is lying down, the program creates an object for controlling the head motors and registers the object with the Tekkotsu Motion Manager. The AIBO also creates a timer that fires at a rate that is specified by the class' type.

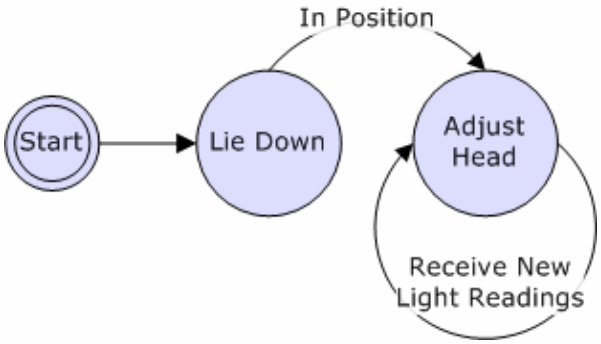

Figure 5.1.2a: Head Movement State Machine.

After registering with the triManager, the triManager will send a *registersuccess* message to the program when the AIBO is successfully registered with the TRI

server. Upon receiving the *registersuccess* message, the program sends its *startquery* message to the TRI server. When receiving query data from the TRI server, the program parses the data and records the new light reading for the respective mote.

The AIBO calculates the average light from the WSN each time the program's timer fires. If the average is different from the previous average, then the program calculates its new head position by translating the average light value into a percent, multiplying the percent by the distance between the maximum and minimum neck position, and then adding the result to the minimum neck position. Finally, the program obtains mutually exclusive access to the head motion object and sets the new joint value.

5.1.3 Sleep-Sit-Stand

Sleep-Sit-Stand, known as the triSleepToStandLightAvgAIBOAgg class within Tekkotsu, can be visualized as a five-node state machine as in Figure 5.1.3a. The initialization phase is almost identical to the Head Movement program. The only differences are that Sleep-Sit-Stand creates an object for controlling the entire robot's posture and loads the five predefined postures (sleep, sit, crouch, stand, excited) into memory. The states and their associated posture can be thought of as a ladder of average light values where sleep is on the bottom and excited is on the top.

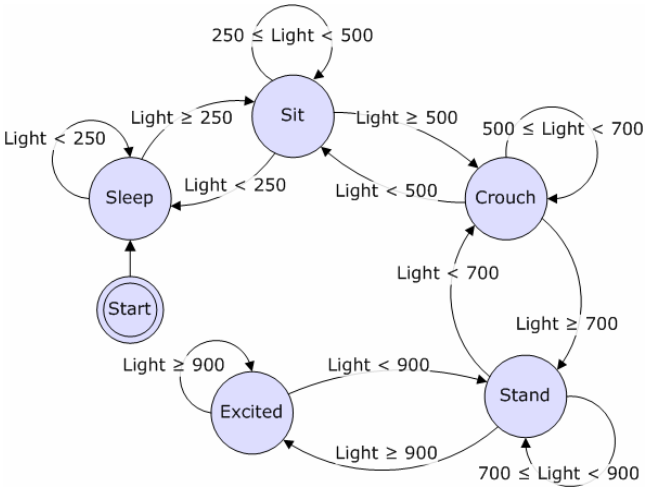

Figure 5.1.3a: Sleep-Sit-Stand State Machine.

When the program is in the sit, crouch, or stand state, the AIBO blends its current posture with the posture above it. The program calculates the percent weight that the two postures should have in the final blended posture by the following calculation:

$$Posture(N+1)\_Percent = \frac{Average\_Light - State(N)\_Min\_Light}{State(N+1)\_Min\_Light - State(N)\_Min\_Light}$$

$$Posture(N)\_Percent = 1 - Posture(N+1)\_Percent$$

After calculating the percents, the program blends the two postures with the current state's posture forming *Posture(N)_Percent* of the final posture and the following state's posture forming *Posture(N+1)_Percent* of the final posture. Thus, during the three interior states of the ladder the AIBO ranges from 100% Posture(N) to 100% Posture(N+1). For example, if the program is in the sit state and the average light value is 450 then *Posture(N+1)_Percent* equals 80% and *Posture(N)_Percent* equals 20%. In this case, the blended posture is 80% Posture(N+1) and 20% Posture(N).

While in the sleeping state, the program moves the AIBO's head using the same algorithm as Head Movement. When in the excited state, the AIBO raises its head to the side, opens its mouth, and flaps its ears in accordance to the average light value in a similar fashion to the Head Movement program. While transferring from one posture to another posture that is more than one state away, the AIBO loads the sequence of postures one after another to create a fluid transition. The algorithm for moving between these postures is not immediately visible because the algorithm must take into account the unique activities that occur at each state.

If the program is in the sleep state and the average light value increases to more than 250, then the program has to move to the "top" of the sleep state before reaching the sit state. The "top" of the sleep state is when the AIBO's neck joint is at its maximum value, which makes the AIBO's head rest on its shoulders. Similarly, when moving from the excited state to the stand state the program must move to the "bottom" of the excited state before reaching the stand state. The "bottom" of the excited state is when the AIBO's mouth is closed, its head is straight, and its ears are not flapping. The "top" of the sleep state and the "bottom" of the excited state are easy to reach because they are the hard-coded postures for the two states that are held in memory.

When the program climbs up the ladder, the program simply transitions by loading each state's posture in order until the target state is reached. When the program moves down the ladder, the program transitions by loading each state's posture in order until the state above the target state is reached. At this point, the program considers itself at the target state and runs the target state's routine for posture adjustment.

The case moving down is different because the states sit, crouch, and stand continuously blend their respective state's posture with the posture for the state above. If the program included the target state's posture while loading states moving down the ladder then the program would reach the "bottom" of the target state and then move upwards into a blend between the target state's posture and the poster for the state above. The only special case when moving down is when moving to the sleeping state. In this special case, the postures load in sequence and include the sleeping posture.

5.1.4 Walk to Light

Walk to Light, known as the triWalkToLight class within Tekkotsu, can be visualized as the state machine in Figure 5.1.4a. The initialization phase includes all of the steps in Head Movement. Additionally, the initialization includes creating and registering a walking motion object and registering the program for vision events.

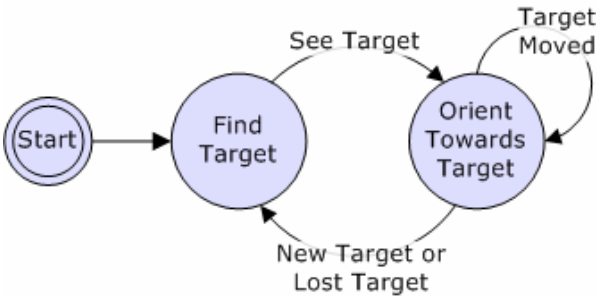

Figure 5.1.4a: Walk to Light State Machine.

The program reaches the find target state by receiving a new target or by losing sight of its current target. The program receives a new target when the current target mote's light value is less than the other mote's light value. The program can lose sight of the current target if the target moves faster than the AIBO can turn to face it.

Once the program is in the find target state, the program uses the walking motion object to make the AIBO turn counterclockwise. The AIBO continually turns until the program receives a motion event for the color that corresponds to the target. The program moves into the orientation state upon seeing the target.

During the orientation state, the program positions the AIBO in front of the target. Two separate processes accomplish this orientation. The first process points the AIBO's nose towards the center of the object while the second process moves the AIBO's legs in the proper direction so that the AIBO's nose will point straight ahead.

The Tekkotsu visual system notifies the program of the target's sighting several times each second. Upon notification, the program determines the center of the object in the camera and updates the head motion's target head pan and head tilt values such that the resulting position will leave the center of the object in the center of the camera. After creating the head motion, the program stores the calculated pan and tilt values and the percentage of the camera that the object occupies in memory (AP).

While in the orientation state, a timer fires at a rate specified by the class' type. The program adjusts the walking motion during each timer firing. The walking motion object includes methods for setting a direction to walk based on a forward velocity, strafing (left) velocity, and turning (counterclockwise) velocity. These forward and strafing velocities are in millimeters per second while the turning velocity is in radians per second.

Conveniently, the head motion object uses radians per second for the pan and tilt values. Thus, the turning velocity of the walking motion object is simply the last recorded pan value. The program calculates the forward velocity based on the last calculated AP and pan values. If the AP is greater than 65% then the AIBO is too close to the target and the program sets the forward velocity to -50. If the AP is less than 45% then the AIBO is far away from the target and the program chooses a positive value for the forward velocity. The velocity chosen depends on the pan value. If the pan value is in-between -10% and 10% then the object is straight ahead and the program sets the forward velocity to 100. Otherwise, the program sets the forward velocity to 50 because the AIBO needs to turn more than it needs to walk forward. If the AP is in-between 45% and 65% then the AIBO is a comfortable distance from the target and the program sets the forward velocity to zero. Additionally, the program always sets the strafing velocity to zero.

5.1.5 Real-Time Replays

The real-time replays of the AIBO demonstrations do not involve any special programming on the AIBO. Instead, the user manually initiates a real-time replay. First, the user must record the replay by starting the desired query and setting it to log data to the database. After recording data, the user can initiate a real-time replay of an AIBO demonstration by telnetting into the TRI server, registering as an agent, starting a replay using the *replayquery* command, and then starting the desired AIBO demonstration program.

5.2 Results

5.2.1 Head Movement

The Head Movement application works as intended. As shown in Figure 5.2.1a, the AIBO lies down and moves its head in accordance with the light. In the figure, the average light percent is at approximately 50% and the AIBO's neck is half way in-between its minimum and maximum values.

Figure 5.2.1a: Head Movement Video Screenshot.

The program's response time is dependent on the sampling period for the light readings. The standard sampling periods in the out-of-the-box application are 128, 256, 512, and 1024 milliseconds. Thus, when the sampling period is 1024 milliseconds, the AIBO's head adjusts with a delay of approximately $1024 + \varepsilon$ milliseconds, where ε tends to be an unnoticeable amount of milliseconds.

5.2.2 Sleep-Sit-Stand

The Sleep-Sit-Stand program successfully moves the AIBO through the different postures and blends of postures. Figure 5.5.2a shows the AIBO in a blend between sitting and crouching. The program has the same response times as the Head Movement program. Additionally, the Sleep-Sit-Stand program can cause the AIBO to shut down if the AIBO is on a rough or sticky surface.

Figure 5.2.2a: Sleep-Sit-Stand Video Screenshot.

When the AIBO reaches the standing and excited positions, the program straightens the AIBO's legs. This process does not involve lifting each paw. Instead, the AIBO effectively drags its feet along the floor. Thus, when on a rough or sticky surface the AIBO's motors may not be able to drag the feet to the proper position. If the motors cannot reach their target position, they will exceed their power output safety limits and the AIBO will immediately shut down. In tests, the motors tend to reach their safety limits within 15 to 30 seconds of a sticky situation.

5.2.3 Walk to Light

The Walk to Light program's integration with the TRI server works flawlessly. The demonstration has similar reaction delays as the other demonstrations. Similarly to the Sleep-Sit-Stand program, the demonstration issues lie entirely in the limitations of the AIBO and Tekkotsu framework.

Figure 5.2.3a: Walk to Light Video Screenshot.
The two targets (the pink and green boxes) each contain one light-sensing mote. When the second light, which is outside of the picture (above the AIBO), is on, the pink target is brightest. Otherwise, the green target receives the most light from the visible lamp in the image.

The Walk to Light program relies on the AIBO's and Tekkotsu's vision system for locating the target object. The vision system is extremely sensitive to lighting conditions. The best lighting conditions are achieved when using multiple white light sources that eliminate most shadows.

Figure 5.2.3a depicts the test setup for the Walk to Light demonstration. Due to resource limitations, the setup is not the ideal configuration. The vision system returns more accurate results when the walls and floor are solid black. In the test setup, the vision system often found hints of pink objects in the brown walls.

Additionally, the size of the target objects have a significant effect on performance. In the test setup, the AIBO had a significantly easier time locating the green target in the bottom left of Figure 5.2.3a than the pink target in the top right of the figure. The green target was easier to locate because larger objects are determined to have a more consistent size in Tekkotsu's built-in vision system.

5.2.4 Real-Time Replays

Figures 5.2.4a and 5.2.4b show the real-time replays of the Sleep-Sit-Stand and Walk to Light programs, respectively. In each figure, the ghost-like AIBO is the AIBO during the replay of the original WSN data. Each figure includes a screenshot from the original recording and an overlay from the recording of the replay. As seen in the figures, the AIBO's responses are almost identical. The only differences are due to different starting positions on the test setup's floor.

Figure 5.2.4a: Sleep-Sit-Stand Original/Replay Overlay.

Figure 5.2.4b: Walk to Light Original/Replay Overlay.

6 Conclusion

TRI allows developers to easily integrate WSNs into their projects without requiring knowledge of the underlying systems for sensing, retrieving, and storing data. The TRI server is accessible for any program that can access the internet or an intranet and establish TCP/IP connections. The AIBO demonstrations provide a basic template for autonomous agent integration with the TRI server and show basic applications where an autonomous agent uses WSN data to make decisions.

The TRI and AIBO demonstration source code is available at http://*research.daysignmedia.com/ug/*. The source code is available under the GNU Lesser General Public License Version 2.1. The previously listed website also includes videos of the AIBO demonstrations, additional commentary, and tutorials for installing and modifying the TRI server and AIBO demonstrations.

7 Acknowledgements

I would like to thank Dr. Yi Shang for inviting me into the Undergraduate Honors Research Program and for providing the AIBO, motes, laptop, and router for the project. Additional funding was also provided by the College of Engineering. The project's development would also not be possible without the contributions from all of the individuals of the Tekkotsu framework, TinyOS, and TinyDB.

References

[1] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An Operating System for Wireless Sensor Networks," *Book chapter in "Ambient Intelligence",* edited by W. Weber, J. Rabaey, and E. Aarts, Springer, April 2005.

[2] S. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong, "TinyDB: An Acqusitional Query Processing System for Sensor Networks," *ACM TODS*, 2005.

[3] P. Buonadonna, D. Gay, J. M. Hellerstein, W. Hong, and S. Madden, "TASK: Sensor Network in a Box," *European Workshop on Sensor Networks (EWSN)*, 2005.

[4] D. S. Touretzky and E. J. Tira-Thompson, "Tekkotsu: A framework for AIBO cognitive robotics" *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-2005)*. Menlo Park, CA: AAAI Press.

Appendix A - TRI Server Commands and Responses

Client to Server

register!#!agent_name

Used to register an agent name on the server.

agent_name can be any string that does not contain "!#!".

If an agent does not register, then it can only use the commands *register* and *exit*.

TRI Response:	Upon successful registration, the TRI server returns *registersuccess*.
Error Status:	1 = The agent is already registered. 2 = The command is malformed. It either does not contain an *agent_name* or contains extra arguments (separated by "!#!"). 3 = Another agent is already using the specified *agent_name*.

exit

Disconnects the agent from the TRI server.

TRI Response:	This command does not have a response.
Error Status:	This command cannot have an error-based response.

getagents

TRI Response:	Returns an *agentlist* with a comma-separated list of the connected agents' names of the form: *agentlist!#!agent_name,agent_name,agent_name,agent_name,...*
Error Status:	This command cannot have an error-based response.

getrunningqueries

TRI Response:	Returns a *runningquerylist* with a comma-separated list of the running querys' names of form: *runningquerylist!#!query_name,query_name,query_name,...*
Error Status:	This command cannot have an error-based response, but can return an empty list that looks like: *runningquerylist!#!*

getquerylisteners!#!query_name

TRI Response:	Returns a *querylistenerslist* with a comma-separated list of the query's listeners' agent names: *querylistenerslist!#!agent_name,agent_name,agent_name,agent_name,...*
Error Status:	1 = The command is malformed. You either did not specify a *query_name* or the command contains extra arguments (separated by "!#!"). This command can return an empty list that looks like: *querylistenerslist!#!*

send!#!agent_name!#!message

Sends the *message* to the agent with the specified *agent_name*.

The *message* field can contain any set of characters, binary data, etc. A *message* is not mangled by the TRI server if it contains "!#!" in the string, binary format, or any other format that the agent wishes to use.

TRI Response:	This command does not have a response.
Error Status:	1 = The command is malformed. It is missing the *agent name* and/or *message*. 2 = The agent specified by *agent_name* is not connected to the TRI server.

sendall!#!message	
Sends the *message* to all agents connected to the TRI server **including** the sending agent.	
The *message* field can contain any set of characters, binary data, etc. A *message* is not mangled by the TRI server if it contains "!#!" in the string, binary format, or any other format that the agent wishes to use.	
TRI Response:	This command does not have a response, but the agent receives its own message in the format: *fromagent!#!agent_name!#!message*
Error Status:	1 = The command is malformed. It is missing the *message*.

sendallbutself!#!message	
Sends the *message* to all agents connected to the TRI server **excluding** the sending agent.	
The *message* field can contain any set of characters, binary data, etc. A *message* is not mangled by the TRI server if it contains "!#!" in the string, binary format, or any other format that the agent wishes to use.	
TRI Response:	This command does not have a response.
Error Status:	1 = The command is malformed. It is missing the *message*.

notifyonagentregister	
Turns on agent register notification for the sending agent. Whenever a new agent registers with the TRI server, agents with this notification turned on receive an *agentregister!#!agent_name* response.	
TRI Response:	Returns a *notifyonagentregistersuccess* response.
Error Status:	This command cannot have an error-based response

NOnotifyonagentregister	
Turns off agent register notification for the sending agent.	
TRI Response:	Returns a *NOnotifyonagentregistersuccess* response.
Error Status:	This command cannot have an error-based response

notifyonagentexit	
Turns on agent exit notification for the sending agent. Whenever an agent exits the TRI server, agents with this notification turned on receive an *agentexit!#!agent_name* response.	
TRI Response:	Returns a *notifyonagentexitsuccess* response.
Error Status:	This command cannot have an error-based response

NOnotifyonagentexit	
Turns off agent exit notification for the sending agent.	
TRI Response:	Returns a *NOnotifyonagentexitsuccess* response.
Error Status:	This command cannot have an error-based response

createquery!#!query_name!#!query_description!#!SQL	
Creates a query in the database with the specified *query_name*, *query_description*, and *SQL*.	
After creating a query, the query can execute on the WSN by using the *startquery* command.	
TRI Response:	Returns a *createquerysuccess!#!query_name* response.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters. 2 = Query creation failed. This is not necessarily bad. If the parameters were of the correct format and the database server was accessible, then this means that the query specified by *createquery* already exists in the database.

startquery!#!query_name!#!type	
Starts the query specified by *query_name*.	
type can be *logonly*, *listenonly*, *logandlisten*, or *lastlogreplay*.	
TRI Response:	Returns a *startquerysuccess!#!query_name* response.
	Also returns a *listentoquerysuccess!#!query_name* response if the *type* specifies that the agent wants to listen to the query.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters. 2 = The query specified by *query_name* does not exist in the database. 3 = The query was successfully started, but the TRI server failed to add the agent as a listener.

stopquery!#!query_name!#!optional_kill_even_if_logging_to_database	
Stops the query designated by *query_name*.	
optional_kill_even_if_logging_to_database can be any set of characters. For example, if you want to use the database kill flag, you can send *stopquery!#!query_name!#!1*.	
If you do not want to use the database kill flag, then the query will continue to exist if it is logging to the database. To not use the kill flag, send *stopquery!#!query_name*.	
TRI Response:	Returns a *stopquerysuccess!#!query_name* response.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters. 2 = The TRI server failed to stop the query. This is most likely because the query is not running.

listentoquery!#!query_name	
Stops the query designated by *query_name*.	
optional_kill_even_if_logging_to_database can be any set of characters. For example, if you want to use the database kill flag, you can send *stopquery!#!query_name!#!1*.	
If you do not want to use the database kill flag, then the query will continue to exist if it is logging to the database. To not use the kill flag, send *stopquery!#!query_name*.	
TRI Response:	Returns a *listentoquerysuccess!#!query_name* response.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters. 2 = The TRI server failed to stop the query. This is most likely because the query is not running.

stoplistentoquery!#!query_name	
Removes the agent from the query specified by *query_name*.	
TRI Response:	Returns a *stoplistentoquerysuccess!#!query_name* response.
Error Status:	1 = The command is malformed. It is missing its parameter or contains too many parameters. 2 = The TRI server failed to remove the agent from the listener list. This is most likely because the query is not running.

setlogquery!#!query_name!#!type	
Changes the log type of the query designated by *query_name*.	
type should be set to *0* for turning database logging off and *1* for turning database logging on.	
TRI Response:	Returns a *setlogquerysuccess!#!query_name* response.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters. 2 = The TRI server failed to set the query's database logging status. This is most likely because the query is not running.

createstartquery!#!query_name!#!query_description!#!SQL!#!type	
Creates the query if it does not exist and then starts the query. This is a combination of *createquery* and *startquery* and is used in all of the AIBO demonstrations instead of the separate queries.	
If the query is already running, then the TRI server adds the agent as a listener if requested by *type*.	
type can be *logonly*, *listenonly*, *logandlisten*, or *lastlogreplay*.	
TRI Response:	Returns a *createstartquerysuccess!#!query_name* response.
	Does not return the success message from *createquery* or *startquery*.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters.

reinjectquery!#!query_name	
Tells the TRI server to reinject the TinySQL query into the WSN. Sometimes after starting a query the WSN fails to return any results because interference stops the query injection from reaching the motes. If an agent is expecting data but is not receiving any data, this command might solve the issue.	
TRI Response:	Returns a *reinjectquerysuccess!#!query_name* response.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters. 2 = The TRI server failed to reinject the query. This is most likely because the query is not running.

replayquery!#!query_name!#!start_date!#!end_date	
Replays the query specified by *query_name* starting at the date specified by *start_date* and ending on the date specified by *end_date*.	
start_date and *end_date* are of the format YYYY-MM-DD HH:MM:SS	
This starts a real-time replay. Once the replay reaches the end of the sensor data in the date range, the replay starts over from the beginning. Note: This means that the replay starts from the beginning once it reaches the end of the date range's recordings, which is not necessarily the same as the *end_date*.	
TRI Response:	Returns a *replayquerysuccess!#!query_name* response.
Error Status:	1 = The command is malformed. It is missing a parameter or contains too many parameters. 2 = The TRI server failed to generate a replay because the query does not exist.

Server Responses	
connectsuccess	Notifyonagentexitsuccess
registersuccess	NOnotifyonagentexitsuccess
agentlist!#!agent_name,agent_name,...	createquerysuccess!#!query_name
agentregister!#!agent_name	startquerysuccess!#!query_name
agentexit!#!agent_name	createstartquerysuccess!#!query_name
runningquerylist!#!query_name,query_name,...	stopquerysuccess!#!query_name
querylistenerslist!#!agent_name,agent_name,,...	setlogquerysuccess!#!query_name
querydata!#!query_name!#!field_0!#!field_1...	listentoquerysuccess!#!query_name
fromagent!#!agent_name!#!message	stoplistentoquerysuccess!#!query_name
fromserver!#!message	reinjectquerysuccess!#!query_name
notifyonagentregistersuccess	replayquerysuccess!#!query_name
NOnotifyonagentregistersuccess	commanderror!#!command!#!status!#!message
commanderror is the response that the server sends whenever an error occurs. The *status* numbers are relative to the *command* and are specified in the **Client to Server** table. The *message* field includes an English message describing the error. The only *command* that is not included in the Client to Server table is *mustlogin*. The *mustlogin* error message is sent if the agent is not registered and sends the TRI server any command except *register* or *exit*.	

Appendix B – triManager's Server Response to Map Translation

The following table includes all of the mappings of the string to string map that the triManager sends to the AIBO applications whenever receiving a response from the TRI server.

Always Available Mappings	
command	The *command* mapping returns the first field of the response string. For example, *command* may return *connectsuccess*, *registersuccess*, *querydata*, *agentlist*, *commanderror*, etc.

Conditional Mappings		
Relative Command	Mapping	Description
querydata	query_name	Returns the name of the query that the result row corresponds to.
	num_of_fields	Returns the number of fields in the result row.
	epoch	Returns the epoch for the data in the WSN.
	query_error	Only exists if the field count does not match num_of_fields. If the mapping exists, it returns "invalid_field_num".
	field_0, field_1,…	The *field_x* mappings only exist if the agent does not use registerQuery-Fields.
	Names of the fields	The names of the fields exist if the agent uses registerQueryFields to name the fields. (This replaces the default mappings of field_0, field_1,…)
agentlist	agent_count	Returns the number of agent names in the list.
	agent_name_0, agent_name_1,…	Each agent_name_x returns an agent's name.
agentregister	agent_name	Returns the name of the agent that joined the TRI server.
agentexit	agent_name	Returns the name of the agent that left the TRI server.
commanderror	error	Returns the command that produced the error. For example, this mapping may return *createquery*, *startquery*, *listentoquery*, *stopquery*, *replayquery*, *send*, etc.
	error_number	Returns the error number for the error. This number is relative to the command that produced the error. The possible numbers for each command are in Appendix A.
	error_message	Returns an English language string of the error.
fromagent	agent_name	Returns the name of the agent that sent the message.
	message	Returns the message that the agent sent. This message can be in any format that the agents have agreed upon using.
fromserver	message	Returns a message from the TRI server. Rarely used.
Any kind of query success message	query_name	Returns the query name that the success message corresponds to.