Artificial Intelligence Software for the Autonomous Interoffice Delivery Robot

A Thesis Presented to

The Faculty of the Computer Science Program

California State University Channel Islands

In (Partial) Fulfillment

of the Requirements for the Degree

Masters of Science in Computer Science

by

Derek Antonio Rodrigues

July 2009

© 2009

Derek Antonio Rodrigues

ALL RIGHTS RESERVED

APPROVED FOR THE COMPUTER SCIENCE PROGRAM

____

Advisor: Dr. Andrzej Bieszczad Date

____

Dr. William Wolfe Date

____

Dr. Peter Smith Date

APPROVED FOR THE UNIVERSITY

____

Dr. Gary A. Berg Date

Artificial Intelligence Software for the Autonomous Interoffice Delivery Robot

by

Derek Antonio Rodrigues

Computer Science Program

California State University Channel Islands

Abstract

The Autonomous Interoffice Delivery Robot (AIDeR) is a robot conceived by Advanced Motion Controls (AMC) for the purpose of demonstrating their servo drive hardware. The AIDeR's function is to take orders for the pickup and delivery of various items in an office environment, autonomously navigating the environment to carry out these orders. This thesis presents the design and implementation details of the artificial intelligence software layer that is responsible for high-level navigation, scheduling, and processing of delivery orders.

Acknowledgements

The author would like to thank the thesis supervisor, Dr. Andrzej Bieszczad for his invaluable guidance, support, and management of the project. In addition, the author thanks his fellow students and AIDeR project team-members Ludovic Hilde, Douglas Whitesell, Andrew Wright, and Robert Kiffe for their respective contributions to the project. Finally, the author thanks his wife Karen Rodrigues and his children for their unwavering patience and support.

TABLE OF CONTENT

CHAPTER 1: INTRODUCTION **11**
1.1 INTRODUCTION TO THE AIDER PROJECT 11
1.2 DESCRIPTION OF THE AIDER 11
1.3 HIGH-LEVEL REQUIREMENTS 12
1.4 SOFTWARE SYSTEM 13
1.5 OVERVIEW OF THE THESIS DOCUMENT 14
1.6 KEY TERMS 15
CHAPTER 2: FUNCTIONAL REQUIREMENTS **16**
2.1 OVERVIEW 16
2.2 JOB 16
2.3 JOB MANAGEMENT 17
2.3.1 *Priority Queue* *17*
2.3.2 *Job List* *19*
2.3.3 *Job Interface* *20*
2.4 JOB PROCESSING 20
2.4.1 *Operating State* *20*
2.4.2 *Job Execution* *21*
2.4.3 *Event Notification* *22*
2.4.4 *Navigation* *22*
CHAPTER 3: DETAILED DESIGN **24**
3.1 OVERVIEW 24
3.2 EXAMINATION OF SOFTWARE TOOLS USED IN THE PROJECT 24
3.2.1 *Eclipse IDE with CDT* *24*
3.2.2 *C++* *24*
3.2.3 *Subversion (SVN)* *25*
3.2.4 *Google Groups* *25*
3.2.5 *VirtualBox* *25*
3.2.6 *Gentoo GNU/Linux 2.6.26* *25*
3.3 SUBSYSTEMS 25
3.3.1 *Job Processor* *27*
3.3.2 *Job Manager* *31*
3.3.3 *Protocol* *33*
3.3.4 *Control* *35*
3.3.5 *Search* *37*
3.3.6 *Command Task* *39*
3.3.7 *Command CGI* *39*
CHAPTER 4: NAVIGATION **41**
4.1 OVERVIEW 41
4.2 PATH PLANNING 41
4.3 NAVIGATION 45
4.4 MAP 46
CHAPTER 5: INTER-PROCESS COMMUNICATION **49**
5.1 OVERVIEW 49
5.2 CONTROL TASK 50
5.3 USER INTERFACE 52

5.4 MESSAGE PROTOCOL 53
5.4.1 *Enumerations* *54*
5.4.2 *Data Types* *54*
5.4.3 *Messages* *56*

CHAPTER 6: TESTING AND EXPERIMENTS 61
6.1 OVERVIEW 61
6.2 SETUP 62
6.3 TEAR DOWN 62
6.4 DEMO MAP AREA # 1 63
6.5 DEMO MAP AREA # 2 64
6.6 TEST PAGES 65

CHAPTER 7: CONCLUSIONS 68
7.1 SUMMARY 68
7.2 FUTURE WORK 69

TABLE OF FIGURES

Figure 1 - Concept drawing of the AIDeR 12
Figure 2 - Software System Layers 14
Figure 3- System overview 16
Figure 4 - Job state machine 19
Figure 5 - Job Processor state diagram 21
Figure 6 - Command Task's primary subsystems 26
Figure 7 - Job Processor class diagram 27
Table 1 - JobProcessor processing loop pseudo-code 28
Table 2 – Job Processing Agent pseudo-code 30
Figure 8 - Job Manager class diagram 31
Figure 9 - Protocol class diagram 33
Figure 10 - Control class diagram 35
Figure 11 - Search class diagram 37
Figure 12 – Graph for first demo map in CSUCI's Bell Tower 42
Table 3 - A* algorithm pseudo-code 44
Figure 13- Command Task IPC interfaces 50
Figure 14- Message protocol through the Command CGI 53
Figure 15 - Message format over the TCP channel 53
Figure 16 - Demo Map # 1 in the CSUCI Bell Tower 63
Figure 17 - Demo Map # 2 in the CSUCI Bell Tower 64
Figure 18 - DeliveryJob.html CGI test page 66
Figure 19 - RemoveJob.html CGI test page 66
Figure 20 - SendCommand.html CGI test page 67

Chapter 1: Introduction

1.1 Introduction to the AIDeR Project

For my graduate research project, I was given the opportunity to participate in a multi-team effort to design and implement the software system that would run a robot named the Autonomous Interoffice Delivery Robot, or AIDeR for short. The terms robot and AIDeR are used interchangeably throughout this document to refer to the AIDeR.

The AIDeR was provided by a company called Advanced Motion Controls (AMC) that plan to use the robot for demonstrating their servo-control hardware. The robot required sophisticated software that would combine embedded system software, artificial intelligence and task scheduling software, and a user-interface.

My responsibility was to design and implement the artificial intelligence layer of the software system. This thesis presents the design and implementation details including the rationale motivating the design and implementation decisions.

1.2 Description of the AIDer

Before proceeding further, a description of the AIDeR is warranted. The AIDeR is a robot that consists of the following hardware components:

- A front-mounted scanning-laser rangefinder for navigation and obstacle detection.
- Ultrasonic rangefinders on the left, right, and rear for navigation and obstacle detection.
- A display screen, keyboard, and pointing device for interacting with users directly at the robot
- 6 wheels powered by servo drives, capable of fine-motor movement, and 360 degree rotation
- Platform that may be lowered and raised to receive or deliver payloads
- Computer running the Gentoo GNU/Linux 2.6.26 operating system
- Batteries to provide power for up to 16 hours of operation
- 802.11b Wireless network interface card to support interacting with users remotely

The following is a concept drawing depicting the robot:

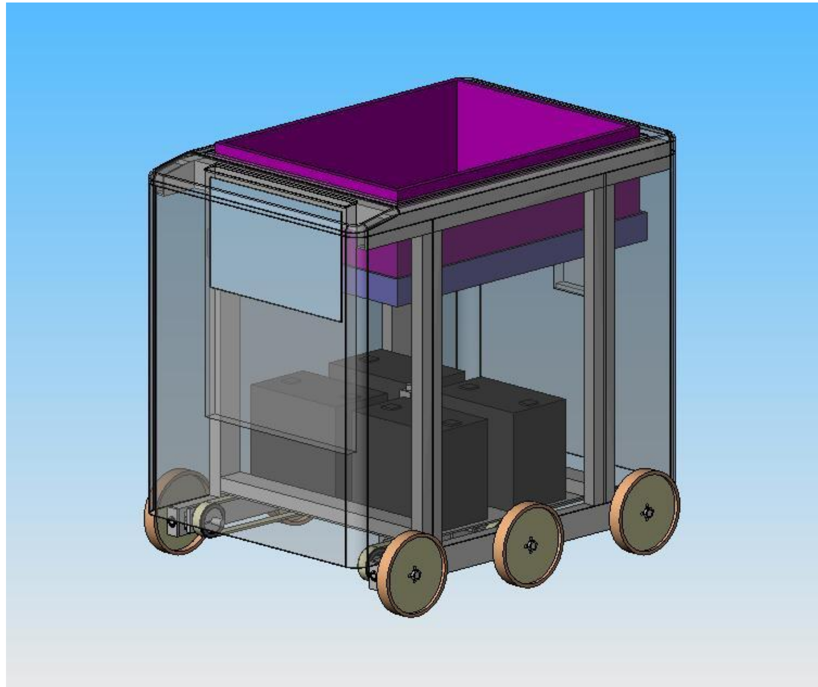

Figure 1 - Concept drawing of the AIDeR

1.3 High-Level Requirements

AIDeR was provided as a fully constructed robot with operational hardware. The robot's chassis, propulsion, and steering systems as well as the battery and charging system was designed and implemented by ME 428/481 students at California Polytechnic State University, with this work completed in March of 2006.

However, the robot still required software that would breathe life into its hardware components. Specifically, the software would need to satisfy the following high-level functional requirements:

- Accept delivery and pickup job requests from users and maintain these in a queue. Local and remote user interfaces are provided to allow users to submit jobs from the robot's user input devices and over the Web respectively.
- Manage the scheduling of jobs based on a combination of user priority and a job priority level classification. The scheduling algorithm would need to account for potential job starvation.
- Use path-planning to find the optimal routes between destinations in jobs.
- Execute jobs in the queue, navigating the environment as necessary to carry out a job. The environment may consist of a series of interconnected hallways and offices.
- The robot must travel within approximately six inches of a right wall whenever possible, and must drive at a safe speed.

- The robot must detect and avoid obstacles, and must alert pedestrians to its presence.
- The robot must monitor its battery power level and return to the charging station when the power level drops below a critical threshold.
- Perform the above functions autonomously.
- Use the robot's computing resources judiciously, giving priority to critical functions such as safety and local navigation.

1.4 Software System

Early in the design, it became apparent that the responsibility for the high-level requirements should be distributed across the following distinct software layers:

1. User Interface
    - Local and remote user interfaces from which users are able to add/remove jobs, view the queue of jobs, and the current status of the robot.
2. Artificial Intelligence
    - Manages a queue of delivery/pickup jobs
    - Manages the execution of jobs
    - Performs path planning
3. Hardware Abstraction
    - Operates and monitors the robot's hardware.
    - Uses the robot's hardware to drive the robot safely in the environment. This includes obstacle detection/avoidance, travelling against a right wall etc.
    - Reports status and error conditions to the Artificial Intelligence layer, including navigation status.

This thesis will focus on the Artificial Intelligence layer. The Artificial Intelligence and Hardware Abstraction layers were given the monikers of Command Task and Control Task respectively, terms which will be used throughout this document.

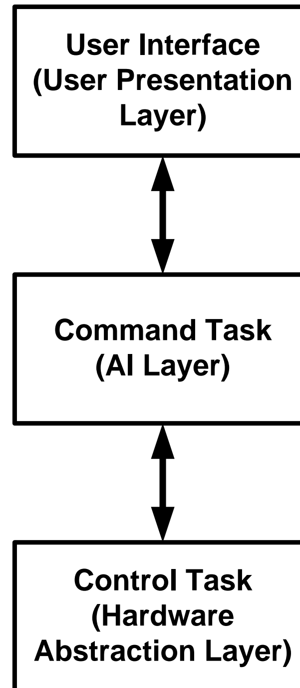

Figure 2 - Software System Layers

1.5 Overview of the Thesis Document

The second chapter presents the detailed functional requirements of the Command Task. The functional requirements will provide a detailed description of what precisely is expected from the Command Task to address the high-level requirements. The requirements are presented iteratively starting with the concrete definition of terms such as a 'Job'. The functional requirements provide the framework for the detailed design and implementation.

The third chapter provides an overview of the design details of the Command Task. The design details 'flow' from the functional requirements. The metaphors presented in the second chapter, such as "Job Manager" and "Job Processor" correspond with subsystems and classes presented in the implementation details.

The fourth chapter discusses the navigation implementation details at a higher level, with less focus on the C++ classes. The chapter discusses how the Command Task performs path planning, and how the resulting navigation plans are executed in conjunction with the Control Task. It is necessary to devote a chapter to this topic because it is one of the primary functions of the Command Task, is reasonably complex, and also an interesting topic in Computer Science.

In the fifth chapter, the inter-process communication mechanisms used by the Command Task are discussed. The Command Task's position as a middleware layer between the User Interface and Control Task required the use of different communication techniques. The implementation of these techniques resulted in a significant portion of the development effort, and are worthy of their own chapter.

The sixth chapter discusses the testing procedures and experiments conducted with the robot in 'demonstration' areas.

The seventh and final chapter will include a summary of the project's results. What remains unsolved and what could be added to further enhance the project will also be discussed.

1.6 Key Terms

- **Autonomous Interoffice Delivery Robot (AIDeR)** – hereinafter referred to as AIDer or robot.
- **Command Task** – the artificial intelligence software layer. Also, the name of the process that implements the Command Task logic.
- **Control Task** – the hardware abstraction software layer. Also, the name of the process that implements the Control Task logic.
- **Command CGI** – a subsystem of the Command Task that receives and processes web requests from the User Interface.
- **User Interface** – the user interface software layer. This term refers to both the user interface software that is run on and displayed in a web browser and the software that is run on and displayed in the robot's display screen.
- **User** – an authorized end-user that may interact with the robot via the User Interface
- **Job** – a task submitted by a User (via the User Interface) that is to be executed by the Command Task on the robot.
- **Job Priority Queue** – a prioritized collection of Jobs submitted by Users that are scheduled for execution by the Command Task
- **Job List** – a collection of Jobs that have already been executed by the Command Task
- **Environment** – an indoor location in which the robot may operate. Typified by a series of interconnected hallways and offices.
- **Landmark Type** – a unique set of physical characteristics as recognized by the Control Task that define a type of location in the Environment. For example, a 'right hallway' landmark type may exist in multiple places on the same map.
- **Landmark** – a unique location and point of interest in the Environment that can be described by a Landmark Type e.g. the right hallway entrance in front of Bob's office.

Chapter 2: Functional Requirements

2.1 Overview

This chapter presents the functional requirements of the Command Task. These functional requirements describe in concrete terms how the software is expected to behave, as well as providing the rationale for the implementation details that follow. The requirements are presented from the bottom up. That is, the most elementary topics are described first providing the basis for the more detailed topics.

The following diagram depicts the Command Task with respect to the other systems in the system architecture:

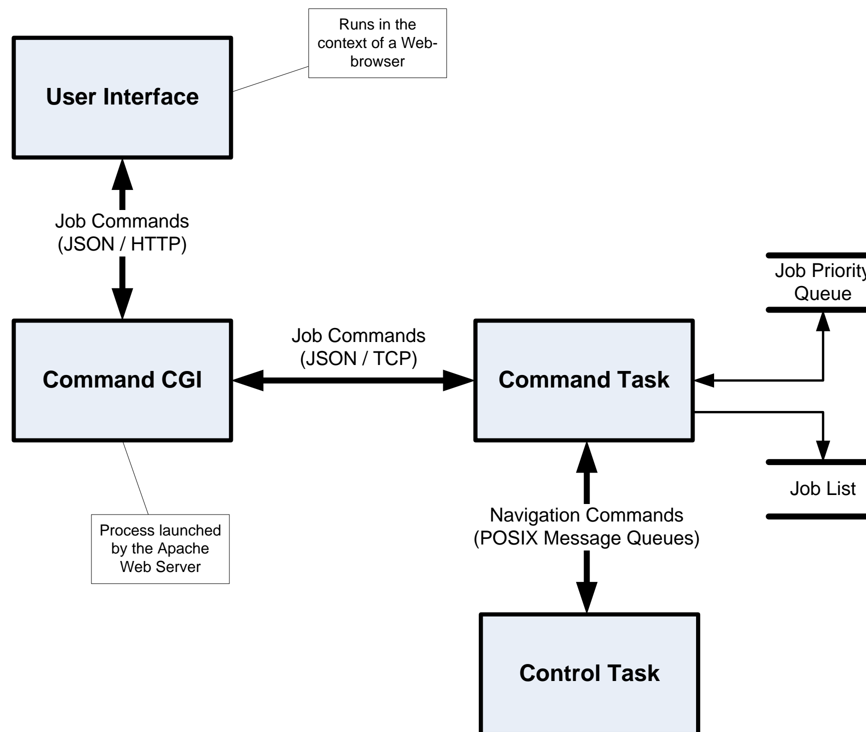

Figure 3- System overview

2.2 Job

A job is a sequence of instructions to be carried out by the robot. An instruction may be one of:

- Move from the current location to a specified location
- Wait for some condition to be met, where the possible conditions are:
    - Wait for a specified time period to elapse
    - Wait for user feedback
    - Wait for full power charge

Each instruction in a job has an associated timeout, which may be specified by the user or has an appropriate (configurable) default. If the instruction cannot be completed before the timeout, then the entire job is aborted. A movement instruction is completed when the robot has arrived at the destination location. A wait instruction is completed when the specified condition has been met.

A job is executed as an atomic operation and, once in progress, will not be interrupted or preempted unless the job is either cancelled or aborted (due to timeout).

The most common types of jobs will be a *delivery* job and a *movement* job.

A delivery job consists of the following sequence of steps:

1. Move to a starting location for item pickup
2. Wait for user acknowledgement that the item has been loaded
3. Move to a final location for item delivery
4. Wait for user acknowledgment that the item has been unloaded

The movement job will consist simply of moving to a location and possibly waiting for some condition to be met. For example, move from the current location to the charging station and wait for full charge.

2.3 Job Management

The Job Manager maintains newly submitted jobs in a *priority queue*. New jobs are added by the local and remote user interfaces which are described. Once a job is assigned, it is removed from the queue and added to the robot's *job list*.

2.3.1 Priority Queue

The priority queue is a global collection of all jobs that have not yet been assigned to the robot for processing. Jobs in the queue are sorted in descending order of job priority (as described below).

When a new job is added to the queue, it is assigned a *job priority*, a *job id*, and a *job state*. These properties are described in the following subsections.

Job Priority

The job priority is computed based on a combination of a given 'service level' and 'user level'.

The service level is a user-specified importance for the job that may be one of the following enumerations (with their corresponding integer values in parenthesis):

- Standard (1)
- Priority (2)
- Express (3)

The user level is the priority level of the user that creates the job and may be one of the following values:

- Casual (1)
- Regular (2)
- Power (3)

The resulting job priority is a product of the service level and user level. Jobs are arranged in the queue in descending order of job priority.

The following table enumerates the job priority resulting from all combinations of service level and user level:

	Casual	**Regular**	**Power**
Standard	1	2	3
Rush	2	4	6
Express	3	6	9

Once in the queue, to avoid starvation, a job's priority level will be increased by 2 for every hour it remains in the queue.

Job Id

The job id is a unique identifier for the job that is assigned by the system when the job is first added to the queue. The job id is immutable.

Job State

The job state reflects the current state of a job with respect to its processing by the robot:

- Unassigned – the initial state of a job when it is added to the queue
- InProgress – the job is currently assigned to a robot
- Aborted – the job failed to be completed
- Complete – all instructions in the job were completed

These are illustrated in the following diagram:

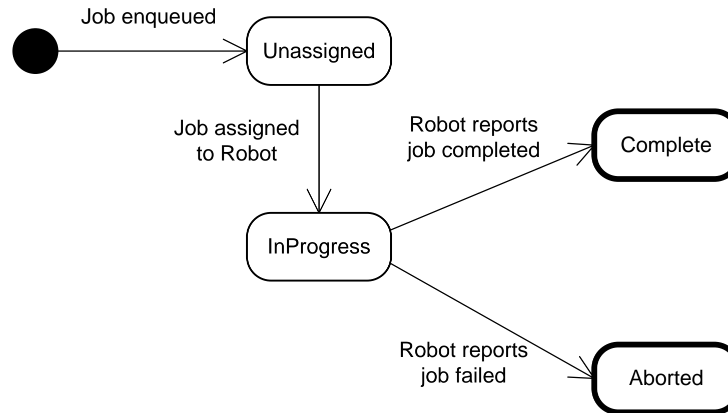

Figure 4 - Job state machine

Job Assignment

When the robot is ready for new work, it will request the assignment of a job from the Job Manager. Jobs will be assigned as follows.

The Job Manager will select from the priority queue as candidate jobs:

- the job at the front of the queue
- all jobs that have an equal priority to the front job
- the first job that has a lower priority than the front job

Of these candidate jobs, the Job Manager will give a temporary increase in priority level to the job whose first (or only) destination location is closest to the current position of the robot.

The Job Manager will then assign to the robot the candidate job with the highest priority, or default to the front job if all candidate jobs have an equal priority.

2.3.2 Job List

Jobs that have been assigned to the robot are maintained in a *job list*. The job list includes:

- the job currently being executed by the robot
- completed jobs – those jobs that have been completed successfully
- aborted jobs – those jobs that failed or were cancelled

2.3.3 Job Interface

The Job Manager exposes an interface to other systems (e.g. the UI, the Job Processor, other Job Managers, etc.).

The interface supports the following operations:

Operation	Description
Add Job	Submit a new job to the queue.
Remove Job	Cancel an existing job in the queue as identified by its job id.
Fetch Jobs	Obtain a list of all unassigned jobs in their current order of priority and a list of all assigned jobs (InProgress, Completed, and Aborted).
Fetch Status	Obtain miscellaneous status, location, and error information.
Get Locations	Get a list of all landmarks in the map.
User Feedback	Provide an acknowledgement when the Job Processor is waiting for user feedback.

2.4 Job Processing

The Job Processor:

- manages the execution of jobs and the operating state of the robot
- computes navigation paths for the traveling legs of jobs i.e. move instructions.
- interfaces with the Control module to:
  - deliver movement instructions, and
  - receive system status updates (e.g. positioning information, power charge level, etc.)
- tracks the current location of the robot on the map.

2.4.1 Operating State

The Job Processor maintains an operating state, which may be one of:

- Disabled - autonomous operation has been disabled. The robot will not execute new jobs until it is re-enabled.
- Offline – the current system time is outside of the configured operating hours of the robot. The robot will resume processing of jobs once the system time is again within operating hours.
- Waiting - the robot is idle and waiting for new jobs

- Busy - the robot is currently executing a job (e.g. charging)

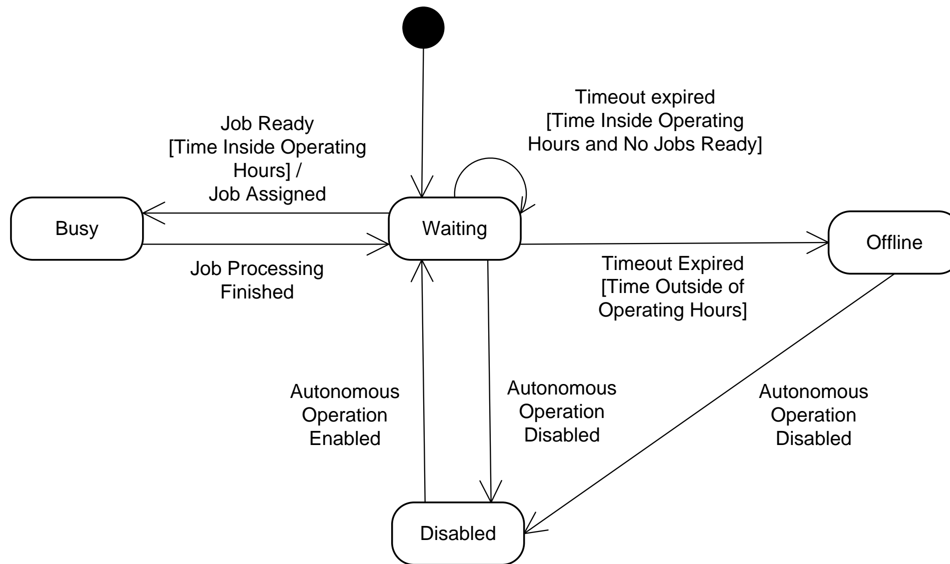

Figure 5 - Job Processor state diagram

2.4.2 Job Execution

When in the Waiting state, the Job Processor requests the assignment of a new job from the Job Manager.

Once a job is assigned, the Job Processor enters the Busy state and begins executing the job instructions sequentially. Instructions are executed as follows:

Instruction Type	Execution Steps	Exit Condition
Move	1. Job Processor computes the navigation plan from the current location to the destination location. 2. Processes navigation plan, passing steps to the Control module for execution one-at-a-time.	• Receive position update indicating that the robot has arrived at the final destination, OR • Receive notification that an error has occurred, OR • Timeout expired
Wait for condition	1. Wait for exit condition	• Specified condition (e.g. power-level full, user-feedback received, etc.) evaluates to true, OR • Timeout expired

If an instruction fails or times out, then the entire job fails and the job state is set to Aborted. If all the instructions in the job are completed successfully, then the job state is set to Completed. The final state of the job is reported to the Job Manager.

At the completion of each job, if the Job Processor determines that the current time is outside of normal operating hours, it immediately enters the Offline state and then creates and assigns itself the job of returning to its 'home' location. When the current time is again within normal operating hours, the Job Processor enters the Waiting state and begins processing jobs again. Otherwise, if at the completion of a job the current time is still within normal operating hours, the Job Processor enters the Waiting state and requests a new job from the Job Manager.

2.4.3 Event Notification

The Job Processor receives asynchronous notifications from the Control Task for the following events:

- Battery/power level – the battery charge level
- Position updated – an update on the robot's current location on the map. These updates are provided, at a minimum, upon arriving at a map landmark.
- Error and other status information

When the Job Processor receives notification that the power level is low, it submits a new high priority job to the Job Manager to return to the charging station.

The Job Processor uses these events to evaluate the condition of any wait instructions that may be in progress. For example, a 'position update' event may indicate that a move step has been completed, or a 'power-level full' event may indicate that a wait-for-condition instruction's exit condition has been met.

2.4.4 Navigation

Before delivering a movement instruction to the Control Task, the Job Processor computes a *navigation plan* that is the shortest path from the robot's current location to the target destination of the move.

The navigation plan consists of a series of static *landmark types* that the robot must reach on the way to the final destination (the *goal*). The Control Task will be able recognize these landmarks from pre-recorded sensor data and will be able to determine the robot's orientation with respect to each landmark.

The navigation plan does not concern itself with the finer navigation details such as obstacle avoidance and circumnavigation, driving through doorways, driving within X inches of the wall, etc. These are the responsibilities of the Control Task.

Each step in the navigation plan consists of a:

a. *landmark* - the current location,

b. *landmark type* – the landmark type as recognized by the Control Task,

c. *distance* – approximate distance to reach the destination landmark.

An example navigation plan:

1. From corner A, head at 90 degrees for approximately 900 centimeters to reach doorway B.
2. From doorway B, head at 270 degrees for approximately 600 centimeters to reach doorway C.

The navigation steps are executed one-at-a-time as follows:

- For each navigation step, the Job Processor dispatches the navigation step to the Control module and then waits for any of:
  - Position update indicating the next landmark type has been reached.
  - Failure notification – the Control module indicates that some error has occurred that is preventing it from reaching a landmark. An alarm is raised before the Job Processor moves on to the next job.
  - Timeout – the robot was unable to reach the destination in a reasonable time period (specified by the user or configurable default) and consequently the job has been aborted. An alarm is raised before the Job Processor moves on the next job.

If all steps in the navigation plan have been executed successfully, the Job Processor proceeds to the next instruction in the Job (if any).

Chapter 3: Detailed Design

3.1 Overview

This chapter provides an overview of the implementation details of the Command Task. The chapter is organized in terms of the software subsystems (i.e. modules) – the units of which the Command Task is composed. These units correspond to the functional objects introduced and described in the preceding chapter.

The Command Task was written in C++. The choice of language was due primarily to preference but also because it does not require an interpreter or other run-time software components that might hinder performance.

With the goal of producing a quality software product that will perform its function correctly and serve as the basis for future work, I attempted to apply sound software development practices such as:

- Object-oriented analysis and design (OOAD).
- Use of mature software design patterns such as those presented in [12].
- Thread-safety and synchronization.
- Exceptions for error handling.
- C++ techniques such as those espoused in [10] and [11].

3.2 Examination of software tools used in the project

3.2.1 Eclipse IDE with CDT

I used the Eclipse Integrated Development Environment with the C/C++ Development Tools for all C++ software development and debugging. In addition, I provided make files so that all Command Task software components can be built using gnu make and without the use of the Eclipse IDE.

3.2.2 C++

- GNU C++ with the following libraries
- Standard Template Library (STL)
- POSIX
- JSONCPP– a free, public-domain C++ implementation of a JSON parser

3.2.3 Subversion (SVN)

Subversion is an open-source revision control system which is used for storing all source code for the AIDeR project. Separate folders are maintained in the repository for the different software components (i.e. Command Task, Control Task, etc.) In addition, periodic tags were taken to capture the source tree at significant project milestones.

3.2.4 Google Groups

Google Groups is an online collaboration forum. The AIDeR project team used a members-only Google Group as a discussion and communication portal for the project. This proved to be a very helpful collaboration tool.

3.2.5 VirtualBox

VirtualBox is a free x86 virtualization platform by Sun Microsystems. I used VirtualBox to host a virtual machine that was configured as much as possible like the configuration on the AIDeR. This machine was my development and test environment.

3.2.6 Gentoo GNU/Linux 2.6.26

The operating system installed on the robot and in the virtual machine used for development. See Appendix B at the end of this document for a more detailed description.

3.3 Subsystems

The Command Task source code is separated into multiple discrete subsystems. The subsystems are separated according to their respective functions and are compiled into static libraries. The declarations and implementations in each subsystem are scoped by unique namespaces. This enhances the modularity and readability of the code, and should provide for ease of future maintenance.

The Command Task is composed of the following subsystems:

- Job Processor
- Job Manager
- Protocol
- Control
- Search
- Command Task
- Command CGI
- Common – there are also a number ‘common’ subsystems that implement general functionality such as exception handling, thread synchronization, object serialization, TCP sockets, date and time, and string parsing. These common subsystems are not discussed in further detail.

The subsystems of the Command Task are illustrated in the following layered diagram. The italicized elements are components of the enclosing subsystem and not subsystems themselves.

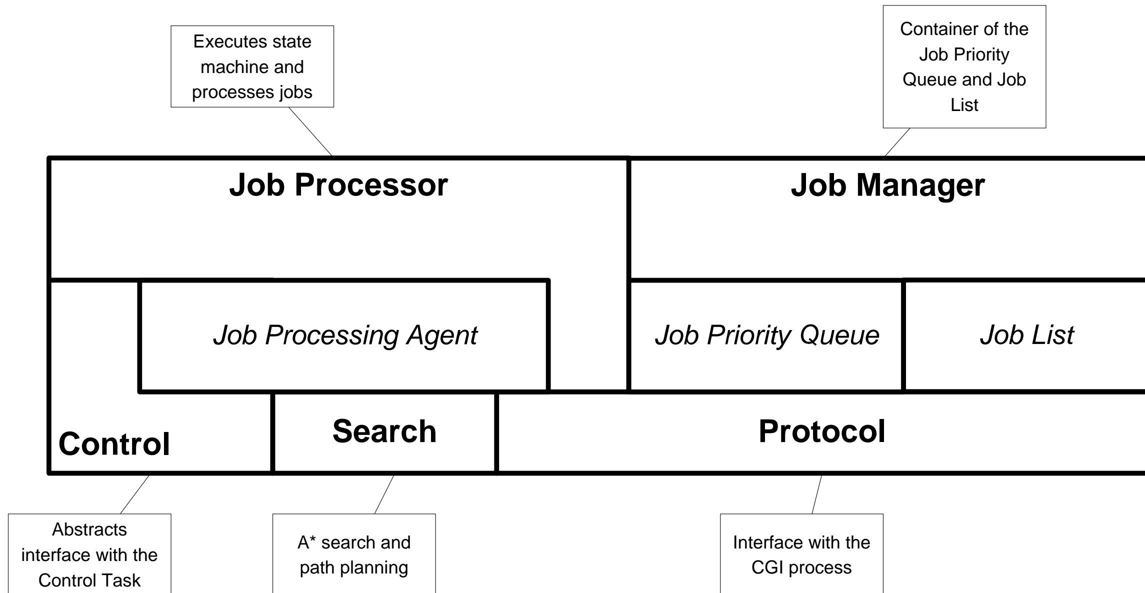

Figure 6 - Command Task's primary subsystems

The implementation details of the above subsystems are described in the following subsections. Each subsection begins with a static UML diagram presenting the classes in the given subsystem. For brevity, the UML diagrams omit trivial details such as class constructors, destructors, accessor methods, and method parameters.

3.3.1 Job Processor

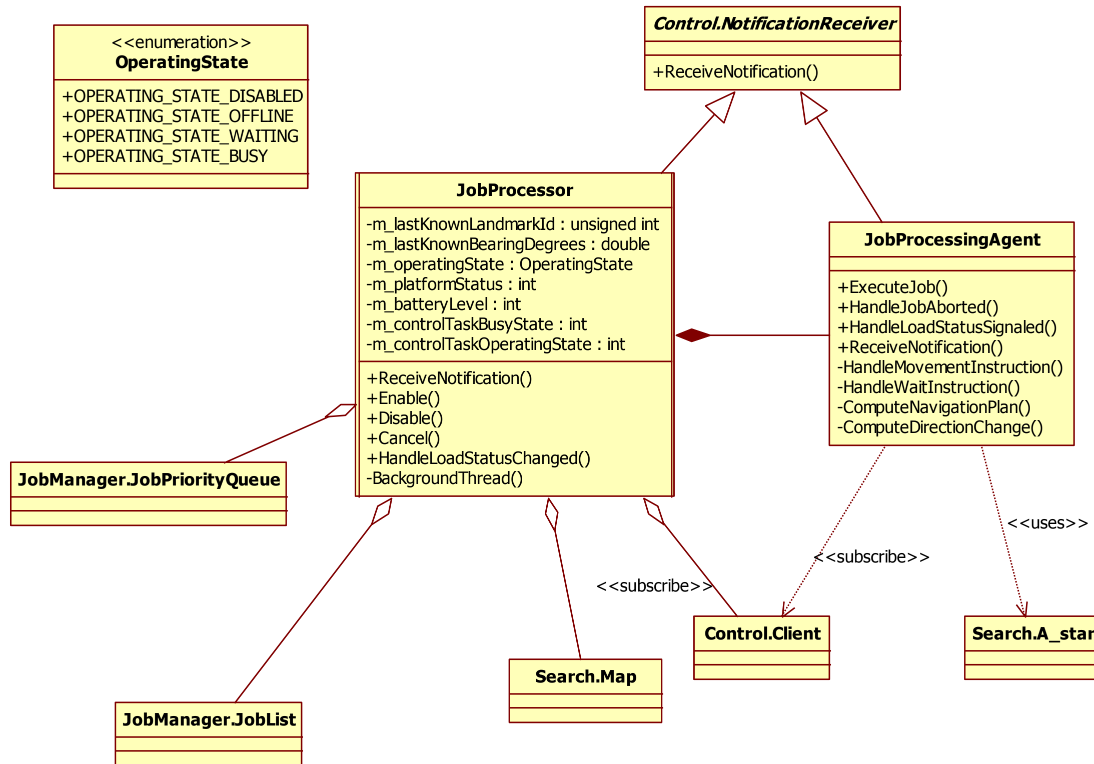

Figure 7 - Job Processor class diagram

The Job Processor class is the 'control center' of the Command Task. As described in section 2.3, it manages the operating state of the robot, executes jobs, and maintains status information including:

- Last known landmark
- Last known bearing (degrees)
- Current operating state
- Platform status
- Battery level
- Control task busy state
- Control task operating state

The Job Processor logic is executed in a continuous loop in which it polls the Job Priority Queue for new jobs (with a 30 second sleep-wait between each interval). The pseudo-code for the processing loop is as follows:

```
While ( Not Shutting Down )
    Wait for 30 seconds or shutdown.
    If ( current time is within operating hours )
        If ( current state is OFFLINE )
            Enter WAITING state.
        End If
    Else If ( current time is outside of operating hours )
        If ( current state is WAITING )
            Create and execute new job to return home.
        End If
        Enter OFFLINE state.
        Continue.
    End If
    Enter BUSY state.
    Retrieve next job from the queue (if any) and execute it.
    Report outcome of last job.
    Continue.
End While
```

Table 1 - JobProcessor processing loop pseudo-code

This loop is executed on a 'background' thread that is created, owned and managed by the Job Processor. The use of a background thread allows for a dedicated thread of execution for the processing of jobs and also frees up the 'foreground' thread (the process' primary thread) to perform other tasks. The background thread exits when the Job Processor object is destroyed.

As demonstrated in the above pseudo-code, if the Job Processor determines that the current time is outside of the configured window of operating hours, then it will immediately process a new job that will return the robot to the home landmark (where the charging station is presumed to be) and will enter the OFFLINE operating state. The background thread will continue to run in the OFFLINE state while outside of operating hours. Once the current time is once again within operating hours, the operating state is changed to WAITING and the Job Processor will resume processing new jobs.

When the Job Processor exits its wait cycle and is in the WAITING state, then it will attempt to retrieve the next Job from the queue. If there is at least one Job available, then the Job Processor constructs a Job Processing Agent object to execute the Job.

The Job Processing Agent is a class that models the intelligent agent metaphor. The sole function of the Job Processing Agent is to execute a given Job. The Job Processing Agent's lifetime is scoped by the execution of a Job, which it performs on the Job Processor's background thread. Once execution is complete, the Job Processing Agent is destroyed and the Job's final status is reported back to the Job Manager.

The PEAS (Performance, Environment, Actuators, Sensors) description for the Job Processing Agent is as follows:

Type:
• Goal-based agent • Learning - may consider recording the actual path traversal time for use in future path-cost evaluations. *Not currently implemented.*
Performance Measure:
• Finds the best path to the destination goal landmark, where best is defined as the path with the shortest total distance.
Environment:
• Partially observable - Receives indication of location at discrete landmarks • Deterministic – the next state is determined by the current state and the action of the agent. • Sequential – current decisions impact future decisions • Static –the map is constructed in advance and is not updated in response to changes in the environment. • Discrete – for each navigation step, either the Control Task does or does not reach the next landmark • Single agent – the agent will not be competing or cooperating with another agent
Actuators:
• Changes environment via navigation commands sent to the Control Task
Sensors:
• Receives location, status, and error updates from the Control Task • Receives job state updates from the Job Processor e.g. job has been cancelled

The Job Processing Agent processes jobs by sequentially iterating and executing the instructions in the job. The pseudo-code is as follows:

```
For each instruction in Job
    If ( instruction type is movement and destination is not current location )
        Compute navigation plan to destination using Search::A_star
        For each step in navigation plan
            Send navigation command using Control::Client
            Wait for asynchronous result of movement or timeout
        Next
    Else If ( instruction type is wait )
        Wait for condition (time period or user-feedback) or timeout
    End If
Next
```

Table 2 – Job Processing Agent pseudo-code

If an instruction is a *movement* type instruction, then the Job Processing Agent uses the Search::A_star class (see section 3.3.5) to construct a navigation plan from the current location to the destination location of the movement. The Job Processing Agent then steps through the navigation plan, sending navigation commands to the Control Task via the Control::Client class (see section 3.3.4 below). Since the interface to the Control Task is asynchronous, the Job Processing Agent must wait until it receives a notification that the Control Task has reached the next landmark before proceeding to the next step in the plan. It does so by entering a conditional wait (using the POSIX condition variable API's). It exits the wait when it either times out (each instruction has an associated timeout) or it receives notification from the Control Task regarding the move.

A *wait* type instruction is handled by simply entering a wait state for a specified time-period or for a signal of user-feedback (e.g. the user has acknowledged that a payload in the tray has been either added or removed).

The Job Processing Agent implements the Control::NotificationReceiver interface and subscribes to the Control::Client class to receive asynchronous notifications from the Control Task (see section 3.3.4 below). Because these notifications are delivered asynchronously, the Job Processing Agent is able to block its own thread of execution while it waits. If it times-out waiting for a notification, then it aborts the job. The Job Processing Agent listens for the following notifications:

- Detected landmark – signals the condition variable to unblock processing and proceed to the next navigation step or next instruction
- Unable to locate landmark - signals the condition variable to unblock processing. The Job Processing Agent consequently aborts the job.
- Battery level – signals the condition variable to unblock processing if the battery level is not low and we were waiting for full-charge.

The Job Processor also receives asynchronous notifications from the Control Task, by implementing the Control::NotificationReceiver interface. The notifications that are handled by the Job Processor are:

- Platform status – this is simply recorded
- Control Task status – this is simply recorded
- Battery level – this is recorded and if battery level is low (level 2 or 3) then a new job is enqueued to return the robot home

3.3.2 Job Manager

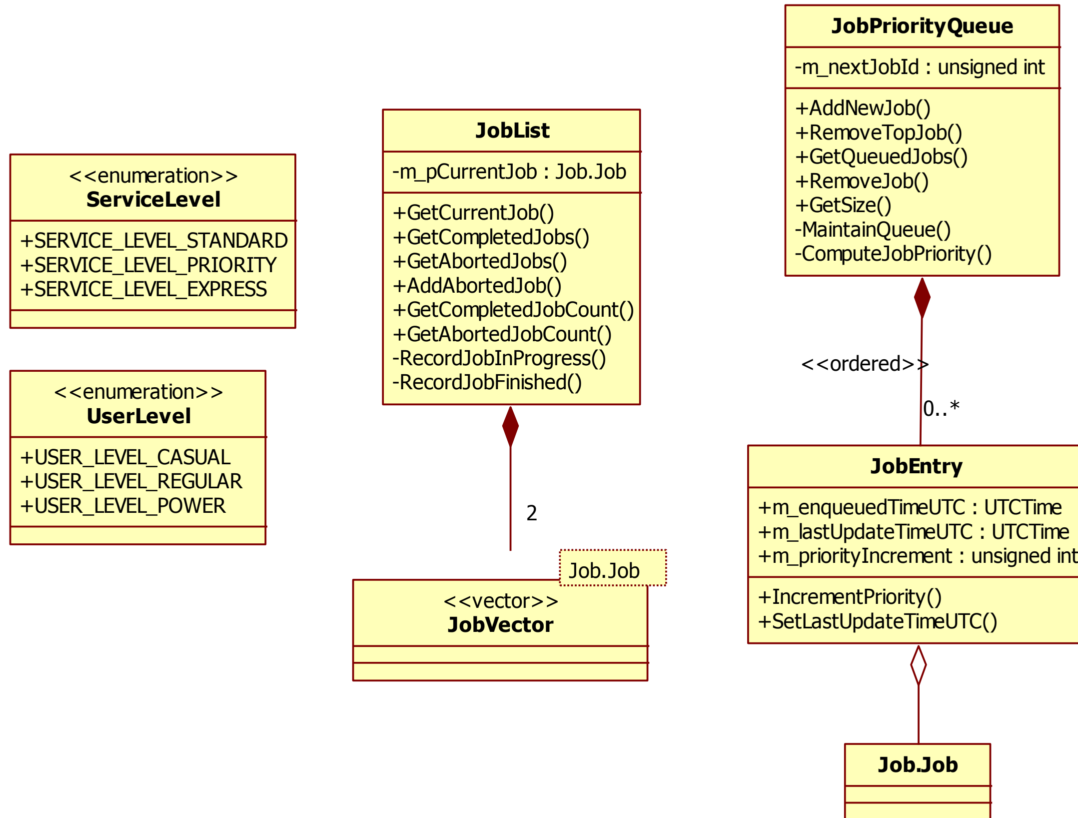

Figure 8 - Job Manager class diagram

The Job Manager subsystem contains the Job Priority Queue and the Job List container classes, as described in sections 2.3.1 and 2.3.2 and presented in further detail in the following sub-sections.

Job Priority Queue

The Job Priority Queue is implemented as an STL queue of jobs. Jobs in the queue are sorted in descending order of job priority. When a new job is added to the queue, it is assigned a job priority, a job id, and a job state.

Although the STL contains a 'priority_queue' implementation, this was not used because the priority of items in the queue can change based on time (i.e. due to the starvation avoidance requirement described in 2.3.1) and a queue was required that can dynamically resort its own elements.

Sorting of the queue is performed lazily:

a. At the time of insertion of a new job, and

b. When the next job is requested

The latter (b) is required to account for changes in priority due to time. The Job Priority Queue gives a priority boost of 2 to jobs for each successive hour they have remained in the queue. For example, a job that has been in the queue for 2 hours will get a priority boost of 4.

The sort is implemented using the STL sort( ) function which has an algorithmic complexity of O(n log n).

For each job, a timestamp is recorded at the time of entry into the queue and every time the job's priority is updated. This allows the Job Priority Queue to determine when it needs to update a given job's priority.

Future enhancements to the Job Priority Queue may consider:

- Collect and maintain queue statistics such as the high watermark, average queue length, average job wait time, etc.
- Consider using a background thread for queue maintenance
- Add a location-based priority boost during job selection
- Add a maximum queue length with alarming when the queue length crosses some threshold and becomes backlogged

Job List

The Job List contains two lists (implemented using STL vectors) of Jobs that have been processed:

- Completed – Jobs that were completed successfully
- Aborted – Jobs that were aborted either due to failure or were cancelled by the user

3.3.3 Protocol

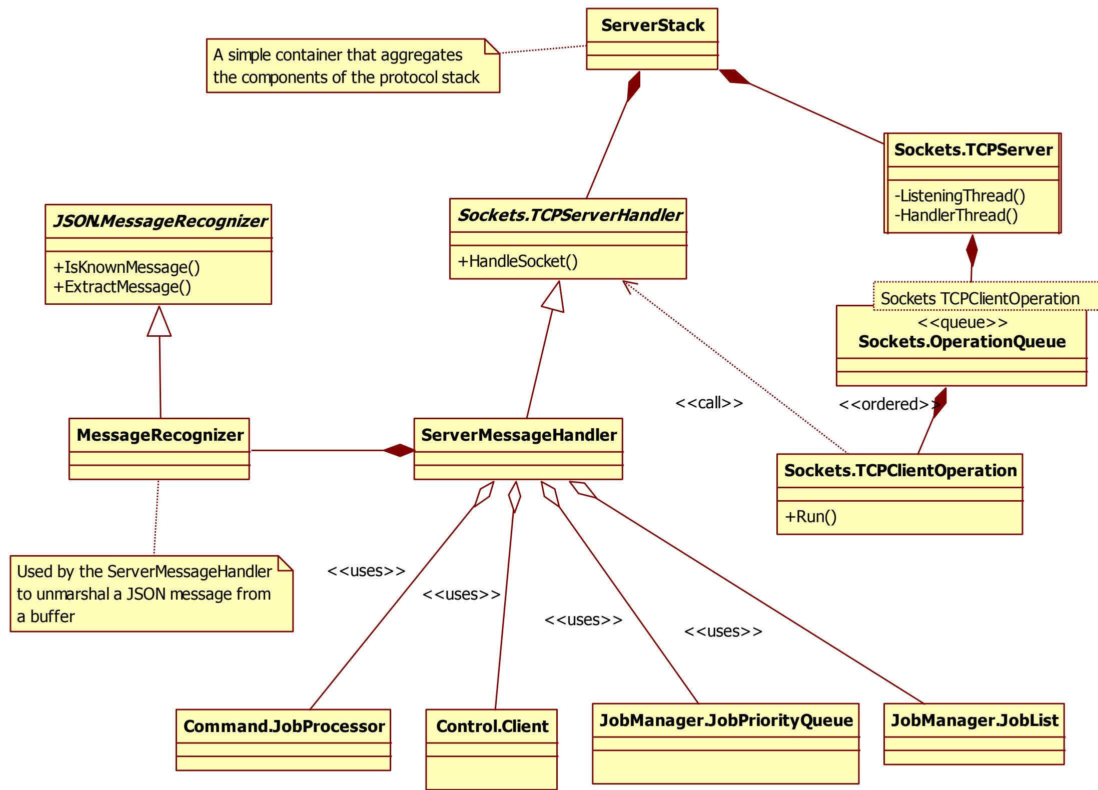

Figure 9 - Protocol class diagram

The protocol subsystem defines the TCP-based server stack and the messages that are required for inter-process communication with the Command CGI process. An overview of the classes is presented here and the topic is covered in further detail in Chapter 5:.

The protocol stack consists of the following classes:

- Sockets.TCPServerHandler – an interface providing a pure virtual HandleSocket() method which must be implemented by derived classes.
- Sockets.TCPClientOperation – contains a pair of Sockets.TCPClientSocket and a reference to a Sockets.TCPServerHandler. Has a Run() method, which when executed invokes the Sockets.TCPServerHandler's HandleSocket() method.
- Sockets.OperationQueue – a FIFO queue of Sockets.TCPClientOperations
- Sockets.TCPServer – opens a listening socket and spawns two background threads which perform the following respective functions: (a) listen for and accept new client sockets which are added to a Sockets.OperationQueue, and (b) process new socket operations in the queue. The processing thread invokes the Run() method of the operations it pops from the queue.

- ServerMessageHandler – implements the Sockets.TCPServerHandler interface. Reads JSON messages from the client socket, processes the messages, and returns a response. Contains references to the JobList, JobPriorityQueue, Search::Map, and the Control::Client, which it uses to service the various requests.
- MessageRecognizer – marshals and unmarshals JSON messages to/from a buffer. Used by the ServerMessageHandler.

The typical flow through the server stack is as follows:

1. The TCPServer listening thread blocks on an accept( ) call waiting for new connections.
2. When a client socket request is received, the client socket is packaged in a TCPClientOperation object and added to the end of the OperationQueue. The listening thread signals the processing thread (via a POSIX condition variable) and then re-enters the blocking accept( ) call.
3. The processing thread wakes from its blocking wait on the POSIX condition variable and reads the TCPClientOperation from the front of the queue. The processing thread invokes the operation's Run( ) method.
4. The TCPClientOperation, which contains the client socket and a reference to a TCPServerHandler, invokes the TCPServerHandler's HandleSocket( ) method and passes the client socket.
5. The ServerMessageHandler, which implements the TCPServerHandler HandleSocket interface, processes the client socket request as follows:
    a. It reads an 8-byte header from the client socket which indicates the length of the JSON message, and the type of the JSON message (a integer identifier).
    b. It then reads the message (of the expected length) into a buffer and 'extracts' the JSON message using the MessageRecognizer. The MessageRecognizer uses a JSON parser to deserialize the JSON message into a C++ object.
    c. The message is then processed based on its type. For instance, a new job is added to the Job Priority Queue.
    d. A response message is constructed and serialized as JSON into a buffer. The response may be a simple success code or could contain more detailed information such as the contents of the Job Priority Queue.
    e. The response buffer is written to the client socket.
    f. The client socket is then gracefully shutdown and closed.
    g. Done.
6. The ServerMessageHandler then returns control to the processing thread which will handle the next operation in the queue, or re-enter a blocking wait on the condition variable.

All messages defined in the protocol are implemented as C++ classes. There is a corresponding C++ class for each message. These messages classes are serialized to/from JSON by the MessageRecognizer, which uses the 3rd party JSONCPP library.

3.3.4 Control

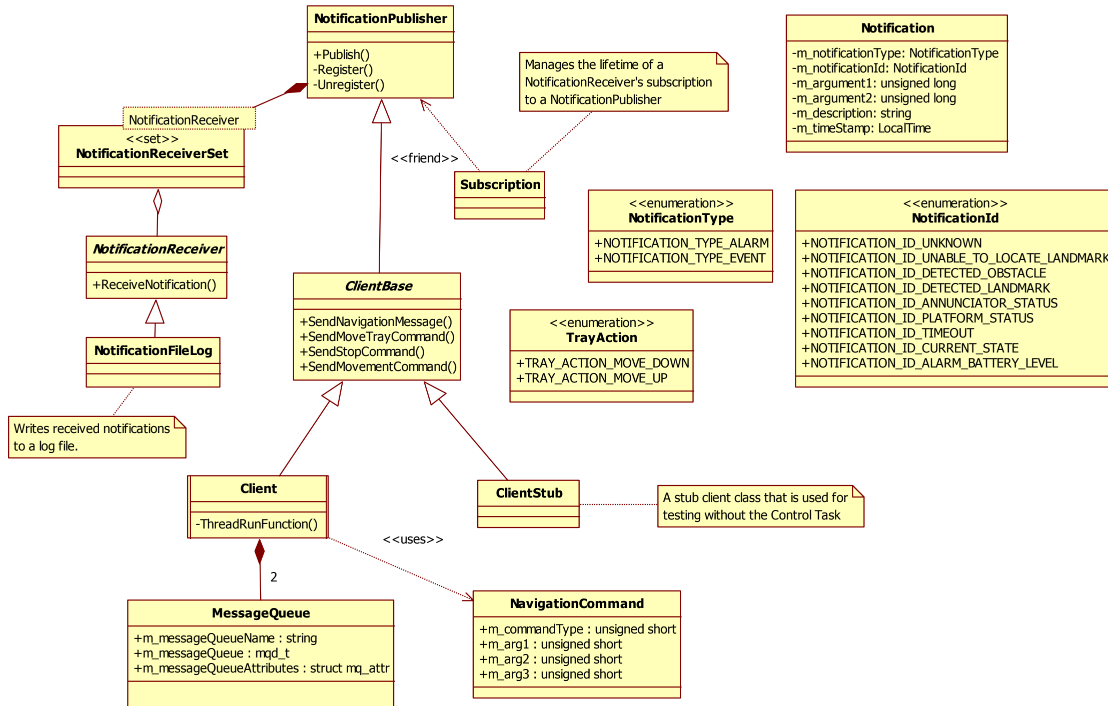

Figure 10 - Control class diagram

The Control subsystem is intended to provide an abstraction of the inter-process communication interface with the Control Task. An overview of the classes is presented here and the topic is covered in further detail in Chapter 5:.

The abstraction is accomplished with a Client class that provides asynchronous methods for sending commands to the Control Task and a NotificationReceiver interface for receiving notifications (events and alarms) from the Control Task. Notifications received from the Control Task are delivered using the ‘publisher/subscriber’ software design pattern, whereby consumers must implement an abstract subscriber interface and subscribe to a publisher to receive notifications. The publisher is the Client class which implements the NotificationPublisher interface. The subscriber interface is the NotificationReceiver class. The Client delivers notifications to all registered NotificationReceivers on its own background thread.

The Control subsystem consists of the following classes:

- Notification – contains the data that is received in a notification from the Control Task.
- NotificationReceiver – the pure virtual interface that must be implemented by a class that would like to receive Control Task notifications.
- NotificationPublisher – a base class that supports the registration and unregistration of zero or more NotificationReceiver’s that would like to receive Control Task notifications.

- ClientBase – the abstract interface for a Client class that declares the methods for sending commands. This base class is defined only to allow a drop-in substitution of a StubClient class, that is used for stubbing the interface to the Control Task. The ClientBase derives from NotificationPublisher.
- Client – Encapsulates the interface to the Control Task. Contains two POSIX message queue handles for sending and receiving messages respectively. Also owns and manages a background thread that receives Control Task notification messages and delivers these to all registered NotificationReceiver subscribers.

The typical usage case is as follows:

1. A consumer class* derives from the NotificationReceiver class, providing a thread-safe implementation of the ReceiveNotification() method.
2. The consumer subscribes to the Client class (a type of NotificationPublisher) to begin receiving notifications.
3. The consumer begins delivering commands to the Control Task using the Client class' interface.
4. The consumer handles asynchronous notifications from the Control Task delivered via the NotificationReceiver interface.

The Job Processor and Job Processing Agent both implement the NotificationReceiver interface and subscribe to the Client class to receive notifications.

3.3.5 Search

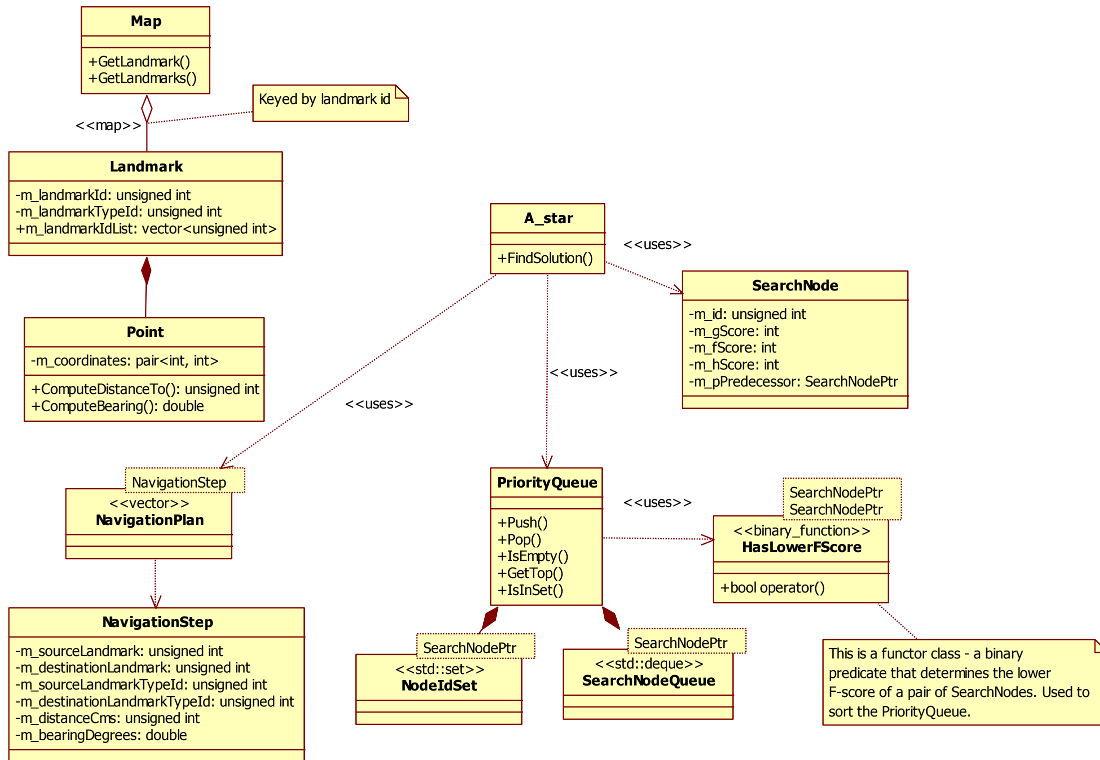

Figure 11 - Search class diagram

The search subsystem encapsulates the path planning logic in the Command Task. The external interface is the A_star class which, given a current location and destination landmark, will compute the navigation plan using the A* search algorithm. This is discussed in further detail in Chapter 4:.

The search subsystem consists of the following classes:

Point - A class representing a Cartesian coordinate and providing methods to:

- calculate the straight-line distance between two points – using the Pythagorean theorem: $a^2 = b^2 + c^2$
- calculate the angle between two points - $(\text{atan}(y_2\text{-}y_1, x_2\text{-}x_1) * 180) / PI$

Landmark - A class representing a ‘point-of-interest’ in the robot’s environment. The landmark consists of:

- Landmark id – a unique identifier for this landmark
- Landmark type id – the id assigned to this type of landmark as defined by the Control Task (e.g. right hallway entrance).
- Point – the Cartesian coordinates of the landmark in the map

- Neighboring landmarks

Map - A collection of all landmarks in the map space. The map class parses the set of landmarks from a text file and exposes methods for obtaining and iterating the landmarks.

Search Node - The Search Node represents a node in the search space. Contains the following attributes:

- f-score
- g-score
- h-score
- predecessor node

Priority Queue - An implementation of a priority queue. This container is required for the 'open set' variable in the algorithm i.e. to be able to retrieve the next search node with the highest f-score. The interface of the priority queue resembles a regular queue, supporting the following operations:

- Push
- Pop
- IsEmpty
- GetTop
- IsInSet

The class uses a lazy sorting approaching, delaying sorting of the search nodes in the collection until GetTop is invoked.

Navigation Plan - The Navigation Plan is the output of the search computation. It is an ordered sequence container that is a collection of Navigation Steps, beginning from the start node to the goal node. Each Navigation Step contains the following information:

- Source Landmark Id
- Destination Landmark Id
- Source Landmark Type Id
- Destination Landmark Type Id
- Distance Centimeters – distance between the source and destination. This is used as the 'guidance' distance communicated to the Control Task to begin looking for the next landmark.
- Bearing Degrees – direction of travel between the source and destination. The bearing is used to detect direction changes between steps.

3.3.6 Command Task

The Command Task subsystem builds the binary executable that is the Command Task. This subsystem includes the application entry point (main) and a Command Processor class.

The Command Processor constructs all the required objects that make up the 'application stack'. This includes:

- Configuration File – reads and parses configuration parameters from a file
- Control Client
- Search Map
- Job Processor
- Notification File Log
- Protocol Server Stack
- Job Manager (Job List and Job Priority Queue)

In addition, the Command Processor reads input from standard in to support various commands useful for debugging.

The Command Task may be run in one 3 modes as determined by a command-line parameter:

- -normal – constructs the application stack for normal operation on the robot and interaction with the Control Task
- -stubbed – identical to the –normal mode, however uses a stubbed interface to the Control Task. This is useful for running the Command Task for debugging and/or testing purposes.
- -interactive – does not construct any objects. Allows for piecemeal or selective construction of objects via the standard input command processing.

3.3.7 Command CGI

The Command CGI subsystem builds the binary executable that is the Command CGI. This executable uses many of the same static libraries used by the Command Task for TCP and JSON support.

The Command CGI process is an intermediary between the HTTP-based User Interface client and the Command Task that facilitates the transfer of JSON messages over the two disparate transport channels (HTTP and 'raw' TCP). The CGI process is not however, a blind relay. It serializes and de-serializes the JSON messages, effectively validating the content before transmission between the two parties.

The main() function performs the following:

1. Reads the HTTP query string and content length from the QUERY_STRING and CONTENT_LENGTH environment variables respectively.
2. Reads the HTTP request content (if any) from standard input.
3. The received content, expected to be a JSON message, is de-serialized into a C++ message object.
4. Establishes a TCP connection to the Command Task process' TCP server.
5. The message object is serialized again and sent to the Command Task using the client socket.
6. The response is read from the client socket and de-serialized into a response message object. The response message object is then serialized to JSON again and streamed to standard out. The standard output is the HTTP response content that is sent back to the HTTP client that originated the request.

In the event of any exception (such as a JSON parsing error, TCP connection failure, etc), the error content is returned to the HTTP client.

See section Chapter 5: for more details on the CGI and messaging interface.

Chapter 4: Navigation

4.1 Overview

In this chapter, we discuss the means by which the Command Task accomplishes navigation of the robot in its target environment (i.e. an office or factory). Navigation is required to carry out those jobs that require movement to deliver a payload from one location to another.

Navigation in the Command Task is a two-step process that consists of (a) path planning, and (b) plan execution, processes which are performed in serial.

Path planning is the phase in which the Command Task uses a path-planning algorithm to produce a navigation plan. The navigation plan is an ordered sequence of navigation steps that will take the robot from its current location to a target destination. Once the navigation plan has been computed, the Command Task must then execute the plan step-by-step to move the robot to its target location.

The details of these processes are described in the following subsections.

4.2 Path Planning

Before executing any movements the Command Task must plan how it is going to get from its current location to its target destination.

All destinations are identified in terms of landmarks - unique points-of-interest in the environment. Landmarks may represent hallway intersections, entries to doorways, or other significant locations in the environment. These landmarks are pre-determined locations that are encoded into a static map. The map is represented as a directed graph with vertices representing the landmarks and edges representing the traversable paths between them. The edges are weighted by the distance between the landmarks (in centimeters). The graph must be directed because of the requirement that the robot must only traverse hallways with the wall on its right-hand side.

The following is a graph corresponding to a hallway intersection in CSUCI's Bell Tower which served as the first demo map:

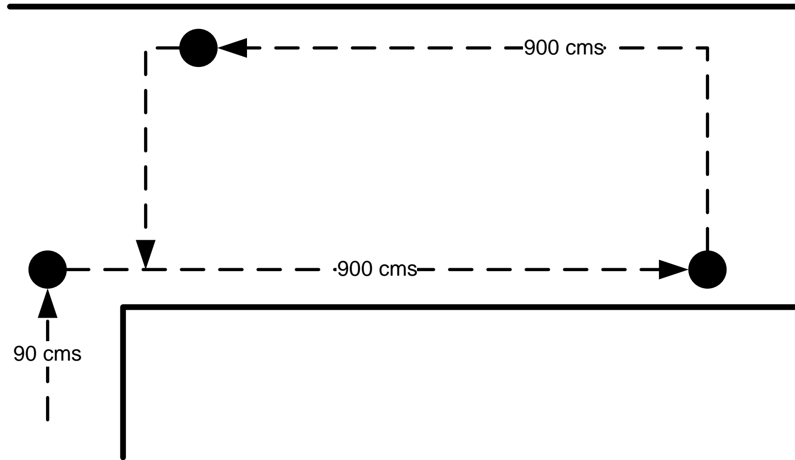

Figure 12 – Graph for first demo map in CSUCI's Bell Tower

Given its current location, the Command Task must determine an optimal route to the destination landmark of its next goal location. It must do so using an algorithm that is optimal with respect to both time and space. These requirements stem from the goal of minimizing resource conflicts with the Control Task, which has real-time operational and safety functions.

The Command Task uses the A* search algorithm for path planning. A* is in the category of 'informed' best-first search strategies. I selected this algorithm because:

1. It meets the performance requirements. According to Norvig and Russell, the A* algorithm is "complete, optimal, and optimally efficient".
2. The algorithm is the most "widely-known form of best first search" and consequently, there is an abundance of documentation and implementation examples.

The A* algorithm evaluates each node in the current set of potential successors using the following function:

$$f(n) = g(n) + h(n)$$

Where:

$g(n)$ = the cost to reach this node

$h(n)$ = the heuristic function which returns the cost to reach goal node from this node

The choice of heuristic function is domain specific. The Command Task uses as its heuristic function: h_{SLD} - the straight-line distance between the landmark node being considered and the goal landmark node. Russell and Norvig suggest this heuristic as "an admissible heuristic". It is an admissible heuristic because it does not overestimate the cost of reaching the goal, which maintains the optimality of A*.

Given two nodes (the landmark being considered in the search path and the goal landmark), the Command Task computes h_{SLD} using the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

I based my implementation of the A* algorithm on the following pseudo-code (adapted from a Wikipedia article on A*):

```
closedset := the empty set
openset := set containing the initial node
g_score[start] := 0
while openset is not empty
{
  pX := the node in openset having the lowest f_score[] value
  if x = goal
    return
  remove x from openset
  add x to closedset
  foreach y in neighbor_nodes(x)
  {
    if y in closedset
      next
    tentative_g_score := g_score[x] + dist_between(x,y)
    tentative_is_better := false
    if y not in openset
    {
      add y to openset
      h_score[y] := heuristic_estimate_of_distance_to_goal_from(y)
      tentative_is_better := true
    }
    elseif tentative_g_score < g_score[y]
      tentative_is_better := true
    if tentative_is_better = true
    {
      came_from[y] := x
      g_score[y] := tentative_g_score
      f_score[y] := g_score[y] + h_score[y]
    }
  }
}
```

Table 3 - A* algorithm pseudo-code

The implementation is defined in a static A_star::FindSolution( ) method which has the following inputs:

- [in] startLandmarkId : unsigned int
- [in] goalLandmarkId : unsigned int
- [in] searchMap : Search::Map
- [out] navigationPlan : NavigationPlan

The output of the search is a *navigation plan* with an ordered sequence of *navigation steps*. Since the search path produces a tree-like structure (with successor nodes pointing back to their predecessor), the navigation plan must be constructed by walking back up the tree starting from the goal node to the starting node, and then reversing these steps to create a plan in the right order. Each step in the resulting navigation plan will contain:

- Source landmark id
- Destination landmark id
- Source landmark type (as understood by the Control Task)
- Destination landmark type (as understood by the Control Task)
- Distance between the source and destination in cms
- Vector difference in degrees between landmarks – this is computed using the arctangent function and is necessary for the Command Task to be able to detect changes in direction between steps.

Nodes in the search tree consist of the following attributes

- gScore
- fScore
- hScore
- predecessor – a pointer to the predecessor node (used to be able to reconstruct the search path).

Since nodes are non-trivial and are copied between data structures (i.e. open set, closed set), they are implemented using object reference counting, whereby only one instance of a given node exists in memory and copies of reference counted pointers are stored and copied between the various containers.

Future work may consider the following strategies to optimize use of the algorithm:

- Use the Memory Bounded and Recursive Best-First Search (RBFS) adaptations of the A* algorithm.
- Perform navigation computation 'in the background' while the robot is idle or busy performing some other task. For example, while waiting for the robot to reach a landmark, we could begin computation of the navigation plan for the next Job.
- Distribute computation across the both the robot and one or more remote computers using RPC.

4.3 Navigation

The navigation function of the Command Task is concerned with executing the plan computed by the path planning function described in the preceding subsection.

Navigation in the Command Task is concerned only with 'discrete' navigation between landmarks. The navigation is discrete because all navigation steps begin at a landmark and end at a landmark. The Command Task does not concern itself with the details of how the Control Task moves between the landmarks e.g. obstacle detection and avoidance, staying within 6 inches of the wall, etc. which are the responsibilities of the Control Task.

Given a navigation plan, the Command Task must execute each step using the Control Task's navigation interface (described in Chapter 5:), which exposes the following high-level navigation commands:

- Travel Against the Wall – move the robot along the right wall of a hallway for a specified distance
- Enter Right Hallway – at the entrance to a hallway intersection, enter the right hallway
- Enter Left Hallway – at the entrance to a hallway intersection, enter the left hallway
- Enter Front Hallway – at the entrance to a hallway intersection, proceed forward across the intersection, entering the hallway ahead of the robot
- U-Turn – perform a u-turn such that the robot will be positioned on the opposite side of the hallway and pointing in the opposite direction
- Stop

The navigation steps provided in the navigation plan must be 'translated' into one or more of the above high-level navigation commands. This is accomplished by determining:

- If the current landmark type indicates we are at a hallway intersection which mandates use one of the hallway navigation commands
- If a change in direction is required – turn left, turn right, continue forward, or turn around. Turning left or right are only allowed at hallway intersections.
- Distance to next landmark

The change in direction is computed from the *bearing* property of each navigation plan step (the bearing is computed using the arctangent function). The bearing is 'normalized' into one of the following values:

- NONE (No direction change) - if direction change is <= 5 degrees.
- TURN_AROUND – if direction change is 180 degrees (+- 5 degrees).
- LEFT - if direction change is < 180 degrees to the left
- RIGHT – if direction change is < 180 degrees to the right

The change in direction is processed first according to the following contextually-based rules:

- If we are at hallway intersection and
  - direction change is LEFT, then issue Enter Left Hallway command.
  - direction change is RIGHT, then issue Enter Right Hallway command.
  - direction change is FORWARD, then issue Enter Front Hallway command.
  - direction change is TURN_AROUND, then issue U-turn command.
- Otherwise, if
  - direction change is TURN_AROUND, then issue U-turn command.
  - direction change is LEFT, RIGHT, or FORWARD, then do nothing.

After processing the direction change, a Travel-Against-Wall command is issued to drive the robot to the next landmark.

After submitting a navigation command, the Command Task must wait for the Control Task to execute it before proceeding to the next step. The Control Task is expected to deliver a notification indicating it has either arrived at or cannot find the given landmark. In the case of a failure indicating the Control Task cannot find the landmark or if the Command Task times out waiting for a notification, then the navigation plan is aborted and the Command Task enters a failure state that requires manual intervention by a human operator. Manual intervention is required to tell the Command Task where it is and determine why the last navigation command failed. Such failures should be unexpected if the map has been constructed correctly.

4.4 Map

As previously described, the map is represented by a static graph in which the landmarks are vertices and the paths between them are edges. The map graph is represented in a dictionary structure in which there is an entry for each landmark and the value contains a description of the landmark including a list of the adjacent landmarks.

The following are detailed instructions for constructing a map. Construction of the map is non-trivial and is nuanced by the Control Task navigation commands and the landmark type definitions available. Before constructing the map, the list of landmark types (defined by the Control Task) must be available.

Instructions:

1. Draw a spatial representation of the map space – the area in which the robot is to operate.
2. Locate and identify all the *landmark types* in the map space – these are all the locations that the robot would identify as landmark types such as a right hallway entrance, doorway, etc.
3. Demarcate these landmark types on the drawing as nodes and write the landmark type id (as defined by the Control Task). Place an arrow indicating the robot's direction of travel with respect to the landmark. This is important because the robot won't recognize the same landmark type when approaching from a different direction.

4. Draw an edge between pairs of adjacent landmark nodes that meet all of the following conditions:
    - the landmarks ‘flow’ in the same direction
    - there is a right hand-side wall that the robot may travel along, or a hallway intersection between the landmarks
    - there are no intermediate landmarks
5. Once you are finished with step 4, the map should look like a directed graph with landmarks as nodes and edges representing the connecting passages (typically hallways). In its current form, the map may consist of multiple disconnected graphs, overlapping graphs, or if you’re lucky, just a single connected graph.
6. The next step is to connect disconnected graphs such that every landmark is reachable. ‘U-turn’ edges are used to connect disconnected graphs. Executing a u-turn will cause the robot to cross from one-side of a hallway to the other, placing the robot in the opposite direction and on the opposite wall. The constraints are:
    - The u-turn must begin at a landmark node and terminate at an edge.
    - The destination edge must be heading in the opposite direction -- by definition this should be true since the robot can only travel with a wall on its right hand side (hallways excluded).
7. Prune any disconnected graphs and edges/nodes that aren’t desired or required.
8. Write the approximate distance in cms of each edge in the graph. This should be reasonably accurate since this distance is used in the path-planning algorithm and also provided as a guideline distance to the Control Task as it looking to identify the next landmark.
9. Add an x and y axis such that the entire graph appears in quadrant I.
10. Add the coordinates (x, y) in cm’s for each landmark with respect to the x and y axis.
11. Assign a unique integer identifier (> 0) to each node that identifies the landmark (not the landmark type).
12. Pick one node to be the ‘home’ landmark – the place the robot will return to when it goes offline (outside of operating hours) or requires a battery charge.
13. Validate the map. Traverse the map as if you were the robot, using only the following movement commands to travel between landmarks:
    a. Travel along wall (wall on right hand-side)
    b. Enter right hallway
    c. Enter left hallway
    d. Enter front hallway
    e. U-Turn

You should be able to travel between any two landmarks using these commands.

14. Finally, encode the map as a text file. For each landmark node, add a line to the text file with:
    - Landmark Id (the unique identifier you assigned to each landmark)
    - Landmark Type Id – the landmark type (as defined by the Control Task)

- The landmark coordinates
- The list of adjacent (neighbor) landmark ids
- A 1 or 0 indicating if the landmark is at a hallway intersection.

Each line should have the following format:

<LandmarkId>;<LandmarkTypeId>;<Coordinate>;<NeighborList>;<IsIntersection>

E.g. 1;1;(100,450);{2,3,7};1

The BNF notation for the map file content is as follows:

```
Line               := <Comment>|<LandmarkDefinition>
Comment            := #[*]
LandmarkDefinition :=<LandmarkId>; <LandmarkTypeId>; <Coordinate>; <NeighborList>
LandmarkId         := unsigned int > 0
LandmarkTypeId     := unsigned int > 0
Coordinate         := (<x>,<y>)
x                  := int
y                  := int
NeighborList       := {[<NeighborIds>]}
NeighborIds        := <LandmarkId>[,<NeighborIds>]
IsIntersection     := 0|1
```

Chapter 5: Inter-process Communication

5.1 Overview

Inter-process Communication (IPC) plays a critical role in the Command Task, facilitating communication with the User Interface and Control Task. Without it, the Command Task would be of little use.

IPC is required for:

- Receiving job related commands from the User Interface
- Issuing navigation commands to and receiving alarm and event information from the Control Task

The two channels of communication are used for very different purposes and not surprisingly have very different design and technology requirements.

The following diagram provides an overview of the IPC interfaces used by the Control Task:

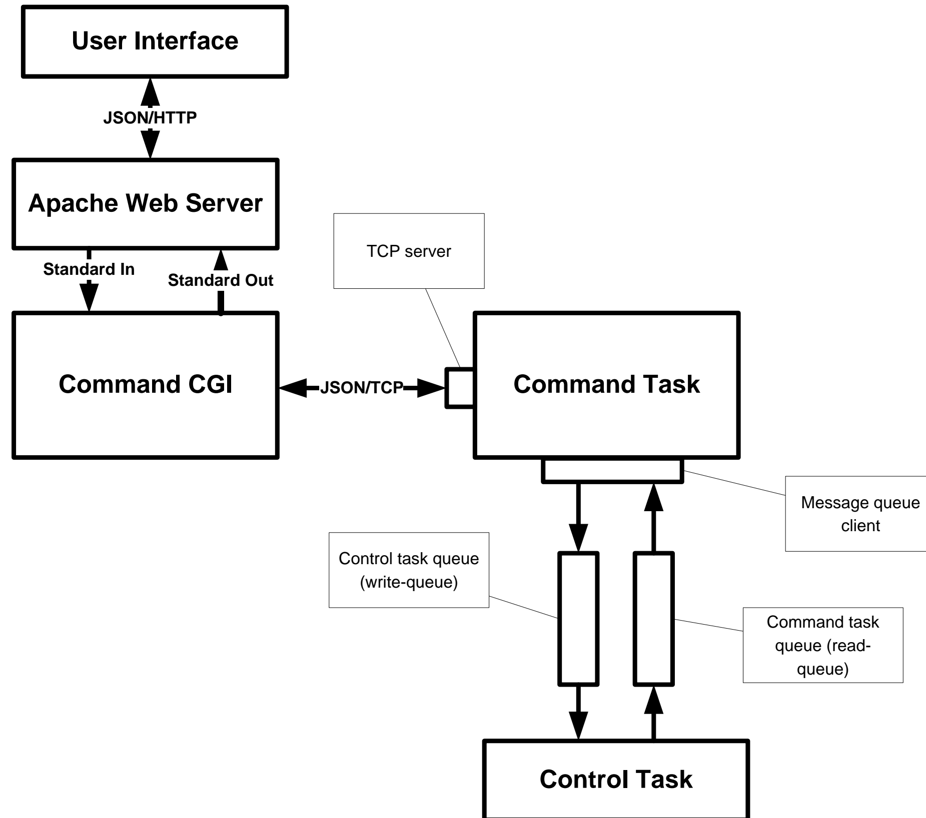

Figure 13- Command Task IPC interfaces

In this chapter we will discuss the nature of each interface.

5.2 Control Task

IPC with the Control Task is accomplished via an asynchronous 'inter-task' messaging protocol using POSIX message queues as the underlying transport mechanism.

The inter-task message protocol, which is defined by the Control Task and exposed via a common header file, defines message structures that support:

1. Sending from 1 to 15 navigation commands in a single message. A navigation command consists of a command type and 3 arguments, the values of which depend on the command type.

The following command types are currently supported:

- Move Forward
- Rotate
- Travel Along Wall
- Stop

- Enter Left Hallway
- Enter Right Hallway
- Enter Front Hallway
- Move Tray
- Make U-Turn

2. Receiving alarm or event data. The alarm/event message consists of:

- Timestamp
- Event type and id
- Text description
- Two arguments, which vary depending on the notification type

The following alarms/events are currently supported:

- Alarms
  - Unable to locate landmark
  - Detected obstacle
  - Timeout
  - Battery level
- Events
  - Detected landmark
  - Annunciator status
  - Platform status
  - Current state

Two named POSIX message queues are used to support fully duplex communication between the Control and Command Tasks:

- "/CONTROLT" – the outbound queue for navigation messages sent to the Control Task
- "/COMMANDT" – the inbound queue for alarm and event messages received from the Control Task

The Command Task assumes the message queues have already been created at run-time.

The implementation in the Command Task abstracts the Control Task API as a C++ client library (described in section 3.3.4). The client provides an asynchronous interface to send navigation messages and receive responses. The Client hides the details of the message queues and the messaging protocol.

The Client uses the publish/subscribe software design pattern to communicate alarms and events it receives from the Control Task. In this pattern, objects subscribe to a publisher to receive asynchronous message notifications. In this case, the publisher is the client and the messages are alarms and events.

The Client accomplishes this mechanism by spawning a 'background' thread to monitor the receiving message queue. The Client's background thread polls the receive message queue at an interval of every 2 seconds. If no message is available, the thread re-enters the wait cycle. Otherwise, the receiving thread reads the message from the queue, and publishes it to all subscribed receivers. The pattern fits well with the asynchronous nature of the underlying message queue transport mechanism.

The Client sends navigation messages on the 'foreground' thread. That is, the message is pushed into the message queue on the caller's thread. If the message is successfully pushed into the message queue, the Client returns immediately.

5.3 User Interface

The Command Task's IPC with the User Interface is required to support a browser-based application implemented using Web 2.0 technologies such as JavaScript, AJAX, HTTP, etc.

Because the Command Task is not and does have an HTTP server, it must rely on a 3rd party Web Server to manage the direct HTTP communication with the User Interface client, which is running in a browser application on an end-user's machine. Including an HTTP server in the Command Task is certainly possible but was not feasible in the context of this thesis. The Apache Web Server is a reliable and industry proven HTTP server and was readily available on the Linux OS distribution installed on the robot. This required however, some means of directing communication received by the Apache Web Server to the Command Task process. The Command Task process itself could not run using any existing CGI or scripting technologies (e.g. Ruby, Python, Perl, JSP, etc) because it needs to run continuously and not only in context of servicing an HTTP request.

To solve this problem, a separate Command CGI application is used. This process handles the HTTP request from the client and then communicates with the Command Task using an 'internal' communication channel. TCP is used as the underlying transport for this internal channel; however another mechanism like pipes could also have been used.

The Command CGI application functions as a hidden proxy between the User Interface and the Command Task. From the perspective of the User Interface, it communicates directly with the Command Task.

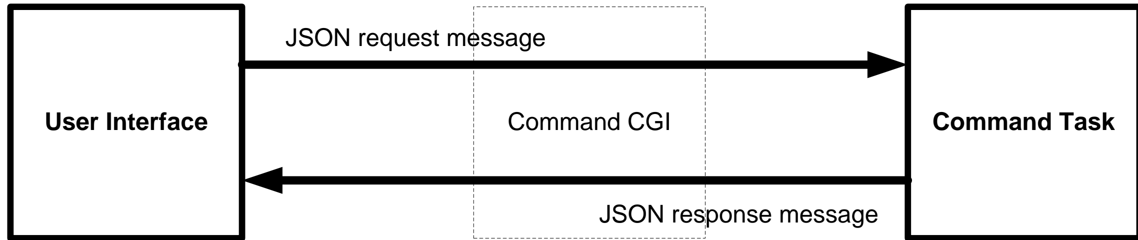

Figure 14- Message protocol through the Command CGI

The application message protocol (described below) is agnostic of the underlying transport. The message protocol is delivered to the Command CGI application through the Apache Web Server using HTTP which takes care of transport level details such as:

- identifying the message type (via the URL), and
- the message length (via the HTTP Content-Length header).

However, these details are not natively supported by the TCP channel between the Command CGI and Command Task. Therefore, all messages exchanged over the TCP channel require a message header. The header is an 8-byte prefix containing a 4-byte message type (a unique integer identifier) and a 4-byte byte-count.

Figure 15 - Message format over the TCP channel

5.4 Message Protocol

The Command Task message protocol consists of a set of object messages that are exchanged between the User Interface and Command Task. The messages are encoded using the JavaScript Object Notation (JSON) format.

JSON is a "lightweight data inter-change format" that is natively supported by the JavaScript language (as of Standard ECMA-262 3rd Edition - December 1999). JSON shares many of the advantages of XML but in a more compact form.

A unique request object message is defined for each function that the User Interface requires. The following request messages are defined:

- Create Job
- Fetch Jobs
- Remove Job
- Get Locations
- Fetch Status

- User Feedback

Each request message has a corresponding response message. All responses include, at a minimum, the following attributes:

- responseCode : int – a server response code. 0 = Success. Any other value indicates failure.
- responseText : string – server response message. This may be a status message in the case of success or an error message in the case of failure.

Example:

```
response:
{
        responseCode:0,
        responseText:"Success"
        ...
}
```

The following sections describe the message protocol in terms of Remote Procedure Call (RPC) interfaces. Each interface has input parameters and output parameters defined which correspond to the attributes of the request message and response message respectively.

5.4.1 Enumerations

The following table lists the enumeration data types used in the interface.

Enumeration Name	**Possible Values**
ServiceLevel	• Standard = 1 • Priority = 2 • Express = 3
UserLevel	• Casual = 1 • Regular = 2 • Power = 3
InstructionType	• Move = 1 • WaitForCondition = 2
WaitCondition	• UserAcknowledgment = 1 • FullPower = 2 • TimePeriodElapsed = 3
JobState	• Unassigned = 1 • InProgress = 2 • Aborted = 3 • Complete = 4
OperatingState	• Busy • Waiting • Offline • Disabled

5.4.2 Data Types

The following table lists the complex data types used in the interface.

Attribute Name	**Attribute Type**	**Optional**	**Description**
Instruction			
Type	InstructionType enum	No	Instruction type.
timeoutSecs	int	Yes	Instruction timeout in secs after which Job is aborted. Defaults to a server configurable default value.
destinationLocationId	int	Yes	Destination location id. Present when instruction type = *Move* only.
waitCondition	WaitCondition enum	Yes	Wait condition. Applicable when instruction type = *WaitForCondition* only.
waitTimePeriod	int	Yes	Wait timeout. Present when instruction type = *WaitForCondition* only and waitCondition = *TimePeriodElapsed*.
Coordinate			
X	int	No	
Y	int	No	
Location			
Id	int	No	Unique identifier for map location. One of the values returned by the GetLocations method (see 0.0.0).
coordinates	Coordinate	No	Location map coordinates
Name	string	Yes	Optional friendly name for map location.
Job			
Id	int	Yes	Job id. Present in server response only.
State	JobState enum	Yes	Job state. Present in server response only.
statusMessage	string	Yes	May contain error and/or other diagnostic information about this job. Present in server response only.
startedTimeStamp	datetime string	Yes.	Time job was started. Present in server response only.
finishedTimeStamp	datetime string	Yes	Time job finished (successful or otherwise). Present in server response only.
instructions	Instruction[]	No	Ordered list of Instructions.

5.4.3 Messages

GetLocations

Returns a list of map locations that are used to identify destination locations in job assignments.

URI Stem – /cgi-bin/command_cgi?mid=9

Attribute Name	**Type**	**Optional**	**Description**
Input Parameters			
None			
Output Parameters			
locations	Location[]	No	List of locations.

Example:

```
response:
{
        responseCode:0,
        responseText:"Success",
        locations:
        [
                {
                        id:0,
                        name:"Bob's Office",
                        coordinates:
                        {
                           x:200,
                           y:100
                     }
                },
                ...
        ]
}
```

CreateJob

Creates and adds a new job to the job queue. Returns a unique server-assigned job identifier.

URI Stem – /cgi-bin/command_cgi?mid=3

Attribute Name	**Type**	**Optional**	**Description**
Input Parameters			
userId	String	No	User id.

serviceLevel	Int	No	Job service level.
userLevel	Int	No	User level.
Job	Job	No	The new job to create.
Output Parameters			
jobId	Int	Yes	Unique job id for the new job. Present only if the job is successfully created.

Example:

```
request:
{
        userId:"Bob123",
        serviceLevel:1,
        userLevel:1,
        job:
        {
                instructions:
                [
                        {
                                type:1,
                                destinationLocationId:5,
                                timeoutSecs:12000
                        },
                        {
                                type:2,
                                waitCondition:1,
                                timeoutSecs:300
                        }
                ]
        }
}

response:
{
        responseCode:0,
        responseText:"Job successfully created",
        jobId:1000
}
```

RemoveJob

Removes an existing job from the job queue as identified by its jobId.

URI Stem – /cgi-bin/command_cgi?mid=7

Attribute Name	**Type**	**Optional**	**Description**
Input Parameters			
userId	String	No	User id.
jobId	Int	No	Unique job id for job to remove.

Output Parameters			
None			

Example:

```
request:
{
        userId:"Bob123",
        jobId:1000
}

response:
{
        responseCode:0,
        responseText:"Job successfully removed"
}
```

FetchJobs

Returns a list of all jobs.

URI Stem – /cgi-bin/command_cgi?mid=5

Attribute Name	**Type**	**Optional**	**Description**
Input Parameters			
None			
Output Parameters			
unassignedJobs	Job[]	No	List of jobs in the queue that have not yet been assigned to a robot. Jobs are in descending order of their priority in the server queue.
assignedJobs	Job[]	No	List of jobs that have been assigned to a robot and are either in progress, have been completed, or have been aborted.

Example:

```
response:
{
        responseCode:0,
        responseText:"Success",
        unassignedJobs:
        [
                {
                        id:1000,
                        state:1,
                        statusMessage:"",
                        instructions:
                        [
                                ...
                        ]
                }
                ...
        ],
        assignedJobs:
```

```
		[
			{
				id:1001,
				state:4,
				statusMessage:"Job completed successfully",
				instructions:
				[
					...
				]
			}
			...
		],
	}
```

FetchStatus

Returns robot status and location details include control task alarms.

URI Stem – /cgi-bin/command_cgi?mid=11

Attribute Name	**Type**	**Optional**	**Description**
Input Parameters			
None			
Output Parameters			
startTimeHour	short	No	Id of last known location.
startTimeMinute	short	No	
endTimeHour	short	No	
endTimeMinute	short	No	
currentOperatingStatus	string	No	
homeLandmarkId	int	No	
lastLandmarkId	int	No	
lastHeadingDegrees	double	No	
currentJobId	int	No	
destinationLandmarkId	int	No	
pendingJobsCount	int	No	
completedJobsCount	int	No	
abortedJobsCount	int	No	
platformStatus	int	No	
batteryLevel	int	No	

controlTaskBusyState	int	No	
controlTaskOperatingStat	int	No	
notificationLog	string	No	
traceLog	string	No	
exceptionLog	string	No	

UserFeedbackInfo

This message signals the Command Task that a payload has been added or removed from the tray. It is used to unblock the Command Task when it is executing a job instruction that requires it to 'wait for user feedback'.

URI Stem – /cgi-bin/command_cgi?mid=13

Attribute Name	**Type**	**Optional**	**Description**
Input Parameters			
None			
Output Parameters			
None			

Chapter 6: Testing and Experiments

6.1 Overview

This chapter describes the procedures used for testing the Command Task as well as the specific experiments conducted to test the Command Task on the robot in several test environments.

Testing of the Command Task included feature testing and debugging both in the development environment and on the robot. Feature testing was approached from the perspective of a quality assurance department – verifying the functional requirements are met and testing various scenarios and corner cases. Debugging included using debug trace and debugging tools (such as the debugging tools supported by the Eclipse IDE).

Initial testing and debugging was conducted exclusively in the development environment (the Linux virtual machine hosted by Virtual Box). This was sufficient to test almost all functionality as the Linux operating system matched that of the robots and included all components required by the Command Task (i.e. Apache Web Server) except for the Control Task.

A Control Task interface 'stubbing' mechanism allowed for execution of various scenarios in the development environment. The stubbing mechanism simply accepted navigation commands and always successfully acknowledged arrival at the requested landmark. This proved to be very helpful as test scenarios could be verified before integration on the robot and most issues could be reproduced without the need for the robot hardware.

Testing of the CGI interface was facilitated by various test pages – simple HTML forms that supported the addition and removal of jobs, fetching status, getting the map locations, etc. These pages exercised all legs of the CGI interface (see Section 5.4.3) and were also useful in testing the robot in general, as they could be used to add and remove jobs. In addition, an internal-only test page was developed to support sending low-level navigation commands to the Control Task outside of the context of a job.

The following sections describe the setup and tear down processes followed when testing on the robot and the experiments tested with the robot to verify the Command Task functionality.

6.2 Setup

The following describes the steps that were routinely followed to prepare for testing and debugging of the Command Task on the robot. These steps assume that the Command Task deliverables are readily available on the tester's laptop computer, and that the Control Task and its dependencies (control_task, joystick, create_queues) are already installed on the robot.

1. Attach the three servo drive power cables (at the rear end of the robot). These are purposely left detached during non-operation to avoid power drain.
2. Toggle the two power switches to the on position.
3. Detach the power source plug (which would otherwise tether the robot to the wall).
4. Depress the motherboard power switch to power on the motherboard and boot up the operating system.
5. Once the operating system has finished booting, make a note of the assigned IP address in start-up trace outputted to the robot's display screen
6. Use WinSCP to establish a file sharing session with the robot. Copy all deliverables into a working directory on the robot (typically the user's directory in /home/). See Appendix A for installation instructions and a list of the deliverables.
7. Push the robot to the desired starting location in the environment. The robot should be positioned at the desired initial landmark (consistent with the bearing and home landmark identified in the configuration.ini file).
8. Use PuTTY or some other terminal emulator to establish a remote session to the robot.
9. Enter sudo -i at the prompt and browse to the working directory.
10. Run ./create_queues to initialize the message queues.
11. Run ./control_task to run the Control Task process.
12. Now use PuTTY to establish another remote session to the robot (keeping the first active).
13. Run ./command_task -normal to run the Command Task process.
14. The robot is now ready for testing with the Command Task.

6.3 Tear Down

The following steps are followed to take the robot offline and move it into a non-operational state.

1. Terminate the Command Task process. This is usually performed by entering the control-c keystroke in the remote terminal in which the command_task was run.
2. Terminate the Control Task process. This is usually performed by entering the control-c keystroke in the remote terminal in which the control_task was run.
3. Using WinSCP, copy the notifications.log and trace.log file from the remote directory to the tester's laptop. These may be inspected offline for debugging.
4. Run the joystick driver process by entering ./joyctrl in one of the remote terminal sessions. Click the button at the base of the joystick. This frees the wheels from the drives, allowing the robot to be pushed. The joystick program will terminate.

5. Push the robot back to the storage location, where the power cord is.
6. Enter the halt command into one of the remote terminal sessions to shutdown the operating system.
7. Close the remote terminal sessions and the WinSCP session.
8. Remove the power cables from the servo drives.
9. Once the operating system has been shutdown, toggle the two power switches to the off position.
10. Plug the robot back into the power source.

6.4 Demo Map Area # 1

Initial integration testing of the Command Task and Control Task on the robot was conducted in a 'demo' area located in the CSUCI Bell Tower. This map, while small and seemingly uninteresting, exercised much of the Control Task's local navigation logic as well as the inter-process interaction between the Command and Control Task and proved to be an excellent initial test environment.

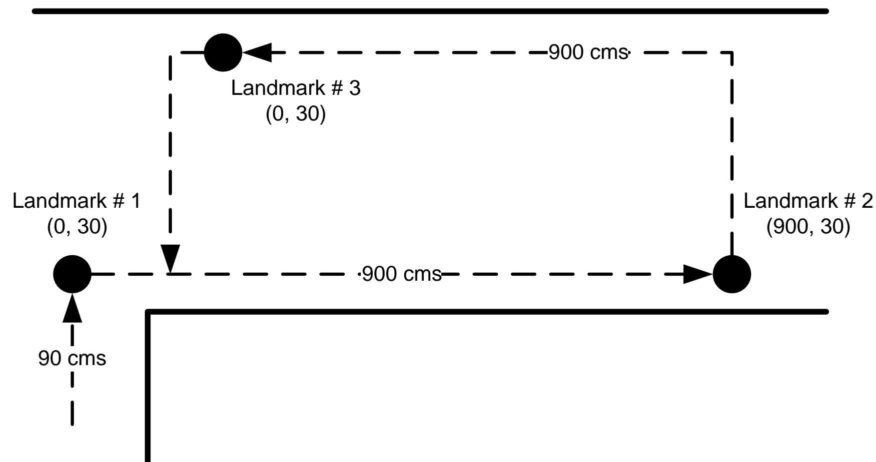

Figure 16 - Demo Map # 1 in the CSUCI Bell Tower

As illustrated in the above diagram, the map consisted of 3 distinct landmarks characterized by 3 landmark types:

- Landmark # 1 - A left hallway entrance with landmark type # 1
- Landmark # 2 – A right hallway entrance with landmark type # 2
- Landmark # 3 – A doorway entrance with landmark type # 3

The x coordinates shown in the graph correspond to the approximate distance in centimeters between the landmarks. The y coordinates were somewhat arbitrary and did not matter in this map since there are no movements along the y-axis.

The map encoding is as follows (see Section 4.4 for map encoding details):

1;1;(0,30);{2};1

2;2;(900,30);{3};0

3;3;(0,30);{2};0

In this map, landmark # 2 and landmark # 3 are neighbors and can be reached by executing a u-turn followed by a travel-against-the-wall. Landmark # 1 cannot be reached from landmarks #2 or # 3. However it can be used as a starting point to reach landmark # 2.

Initial testing in this map area exercised the inter-process communication protocol between the Command and Control Tasks. Jobs executed in this area were jobs consisting of a single movement instruction such as: move from landmark # 1 to landmark # 3.

6.5 Demo Map Area # 2

Demo Map Area # 1 proved to be an invaluable starting point, however this did not fully exercise the path-finding and job processing logic of the Command Task. While thorough testing of the path-finding and job processing was performed in the virtual machine development environment using various test maps, a more complex area was needed to integration test the Command Task using the actual robot.

The demo area was expanded to include another portion of the CSUCI Bell Tower hallway with several additional landmarks, adding for the possibility of more complex navigation plans and jobs.

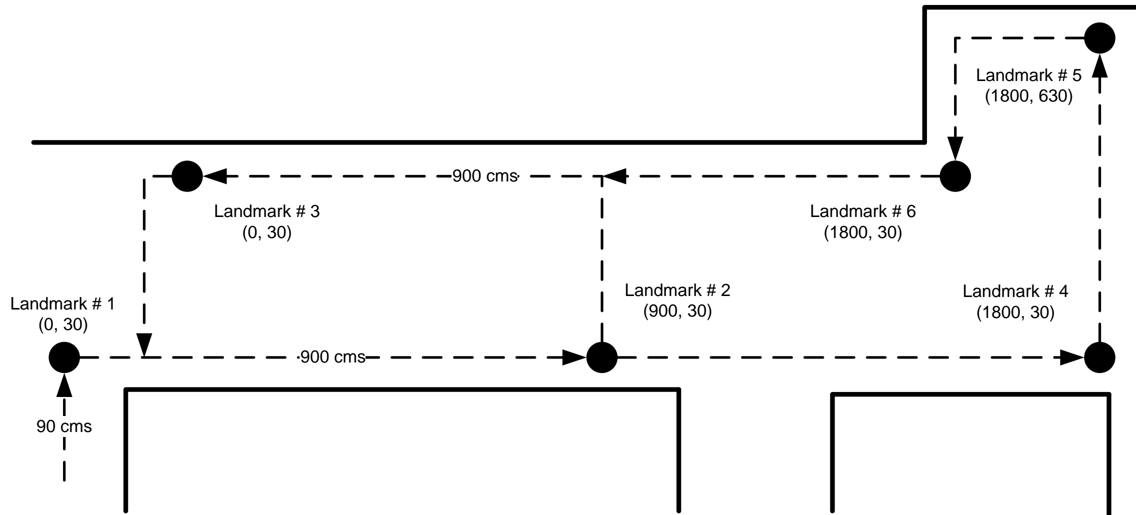

Figure 17 - Demo Map # 2 in the CSUCI Bell Tower

The map encoding is as follows:

1;1;(0,30);{2};1

2;2;(900,30);{3,4};1

3;3;(0,30);{2};0

4;4;(1800,30);{3,5};1

5;5;(1800,630);{6};0

6;6;(1800,30);{3,4,5};1

Note that the y coordinates for landmarks in the same hallway are identical e.g. landmark # 4 and landmark # 6 share the same coordinates. In the context of the real world environment, landmark # 4 and # 6 are really the same location but just being approached from different angles. The static map views these as distinct landmarks.

Successful tests conducted in this map included multiple successive job requests such as the following:

1. Move from landmark # 1 to landmark # 4 and wait for user feedback (payload loaded).
2. Move from landmark # 4 to landmark # 6 and wait for user feedback (payload unloaded).

In some tests, the robot veered slightly off course due to a slope in the hallway, and failed to recognize a landmark. This resulted in the job being aborted which required the robot to be manually moved back to a landmark. This underscored the need for some type of fault tolerance and recovery mechanism, which is beyond the scope of this thesis.

6.6 Test Pages

The following figures are illustrate a few of the non-trivial test pages used to verify the CGI interface and also used for general purpose testing of the Command Task.

These pages contain forms that use JavaScript to submit a JSON formatted request message to the CGI interface and then display the corresponding response (also JSON) in a text box.

Pickup Landmark:

1

Dropoff Landmark:

1

HTTP result:

Server Response:

Add New Job

Figure 18 - DeliveryJob.html CGI test page

Job Id:

1

HTTP result:

Server Response:

Remove Job

Figure 19 - RemoveJob.html CGI test page

Command Type:

Move Forward

1

Argument 1 (Timeout = 0..600 seconds):

0

Argument 2 (Distance cms / Radius degrees / Radius cms / Down = 0 / Up = 1):

0

Argument 3 (Speed / Landmark Id / Clockwise = 1 / Counterclockwise = 0):

0

HTTP result:

Server Response:

Send Control Command

Figure 20 - SendCommand.html CGI test page

Chapter 7: Conclusions

7.1 Summary

The goal of this thesis project is the completed development of an Artificial Intelligence software layer that would fit into the overall system architecture, and would meet the high level requirements documented in section 1.3. Specifically, the software would need to accept delivery and pickup job requests from users and maintain these in a queue, manage the scheduling of jobs, use path-planning to find the optimal routes between destinations in jobs, and execute jobs in the queue, navigating the environment as necessary to carry out a job. As a result of the design, implementation, and testing of the Command Task software, these requirements were successfully accomplished and demonstrated in a demo map area using the robot.

The Command Task, as a middle-ware layer in the system architecture, was implemented using a variety of technologies – from the use of message queues to communicate with the Control Task, an implementation of the A* search algorithm for path planning, an intelligent agent for job execution, to the JSON-based TCP and CGI interfaces that are used for interaction with the local and Web-based user interfaces.

Sound software engineering principles were applied to the design and development of the Command Task, which should provide a solid foundation for future maintenance and iteration of the software, leading to its eventual deployment to production. Specifically, software design specifications were written and reviewed in advance of development, the use of mature software design patterns were used, and the source code written using quality-driven industry techniques.

There is plenty of opportunity for further enhancements and new features to the Command Task platform (see Section 7.2 below). Certainly, some refinement is necessary before the Command Task would be ready for use in a production environment. For example, a failure tolerance and recovery mechanism is needed to reduce or eliminate the need for human intervention when there are local navigation failures such as the failure to recognize a landmark or an obstacle that cannot be circumnavigated. In addition, the process of constructing the static map is tedious and has room for refinement to reduce the effort required by the administrator.

In conclusion, I am very grateful for the opportunity to have contributed to this project and collaborated with an excellent team to create a working product. In addition to the learning experience, it was particularly rewarding to see the physical robot executing delivery jobs in response to a job requests sent from a Web browser.

7.2 Future Work

The following is a list of suggestions that may be considered for implementation in future work on the Command Task software. Some of these are functionality that was identified during the initial design but deferred due to scope. These suggestions are grouped according to their general area of functionality.

Jobs

- Consider recycling terminated and aborted jobs, so that users could recall tasks that are routine. Perhaps allow for recurrence of jobs with some frequency e.g. regular mail pickup every weekday morning.
- Allow users to schedule jobs for execution at a later date or time, so the Job Manager would not consider them until at least the specified date/time.
- Preempt the current job when the robot is en route to a location with an empty bin and gets a higher priority job to complete. Add an administrative setting that could control the commitments; for example, the system might be required to finish a job that has already started even with an empty bin. That could avoid confusion of users, who would not know why the robot is not fulfilling their requests. If dynamic changes are allowed, then in theory user starvation may occur. That suggests that waiting time must be a part of the computation as well.

Scheduling

- Evaluate queued jobs for overall (power consumption) efficiencies. I.e. estimate the power requirements for job and use them in the job prioritization algorithm.
- Evaluate constraint-based optimization, optimization of elevators, other options to determine if there is a better algorithm for job queue prioritization and selection.

Navigation and Path-Planning

- Consider navigation plan computation as a distributed task shared between the robot and a remote machine, to offload some of the computational burden from the robot’s hardware.
- Currently, the A* search heuristic function uses the straight-line distance from the current node to the goal. Consider factoring an ‘estimated travel time’ into the heuristic function. Initially, this estimate could be populated by an educated guess or based on trial runs. However, after time, this could be based on an average of recent actual traversal times.
- When the Control Task reports failure to reach a landmark, the Job Processor will abort the job and requires manual intervention to recover. The Command Task could incorporate intelligence to reason about its current location and attempt to discover where it is without human intervention. This may include interfacing with the Control Task to obtain data about the environment. This same reasoning mechanism could be used to allow the robot to discover where it is at any location in the environment.
- The Command Task could incorporate learning into navigation plan execution to handle transient failures such as obstacles in the way. The Command Task could compute an alternate search path that avoids the current ‘problem’ location.

- Currently, map construction and the subsequent verification by testing, can be a lengthy and very tedious process. Some level of machine learning may be considered to help automate this process, whereby the robot would be 'walked' through the environment and would discover and dynamically compile the map.

References

1. *"Autonomous Interoffice Delivery Robot: Software Development of the Control Task"*, Master's Thesis, by Ludovic Hilde, California State University, Channel Islands, 2009.
2. *"AIDeR Local User-Interface"*, Capstone Project, by Andrew Wright, California State University, Channel Islands, 2009.
3. *"AIDeR Remote User-Interface"*, Capstone Project, by Robert Kiffe, California State University, Channel Islands, 2009.
4. *"Artificial Intelligence: A Modern Approach"* 2nd Edition, by Stuart J. Russell, Peter Norvig, Prentice Hall, New Jersey, 2003.
5. *"Constructing Intelligent Agents Using Java"* 2nd Edition, by Joseph P. Bigus, Jennifer Bigus, Wiley, New York, 2001.
6. *"Hybrid Control for Robot Navigation: A Hierarchical Q-Learning Algorithm"*, by Chunlin Chen, Han-Xiong Li, and Daoyi Dong, IEEE Robotics and Automation Magazine, 2008.
7. *"JavaScript Object Notation"* (for notation specification and definition), http://www.json.org
8. *"A*"* (for search algorithm pseudo-code), http://en.wikipedia.org/wiki/A*
9. *"AIDeR Autonomous Interoffice Delivery Robot Phase II Design Document"*, by Kevin Steinberg, California Polytechnic University, 2006.
10. *"Effective C++: 55 Specific Ways to Improve Your Programs and Designs"*, 3rd Edition, by Scott Meyers, Addison-Wesley Professional, 2005.
11. *"More Effective C++: 35 New Ways to Improve Your Programs and Designs",* by Scott Meyers, Addison-Wesley Professional, 1996.
12. *"Design Patterns: Elements of Reusable Object-Oriented Software"*, by Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Addison-Wesley Professional, 1994.
13. *"json-cpp library",*(an open-source C++ implementation of a JSON parser), by Baptiste Lepilleur, http://sourceforge.net/projects/jsoncpp/

Appendix A – Installation Instructions

Get all content from the command\bin folder in SVN e.g. from aider\trunk\command\bin.

At present, this includes:

- command_cgi
- command_task
- configuration.ini
- DemoMap.txt
- ExtendedDemoMap.txt
- DeliveryJob.html
- FetchJobs.html
- FetchStatus.html
- GetLocations.html
- Index.html
- RemoveJob.html
- SendCommand.html
- SimpleJob.html
- UserFeedback.html

You can also build the binaries yourself:

```
svn checkout svn+ssh://<yourid>@oak.cs.csuci.edu/var/svn/aider
cd aider/trunk/command
make
```

Install the CGI application and CGI test pages (from the aider/trunk/command directory):

```
sudo cp ./bin/command_cgi /var/www/localhost/cgi-bin
sudo chmod 775 /var/www/localhost/cgi-bin/command_cgi

sudo cp ./bin/DeliveryJob.html /var/www/localhost/htdocs
sudo cp ./bin/FetchJobs.html /var/www/localhost/htdocs
sudo cp ./bin/FetchStatus.html /var/www/localhost/htdocs
sudo cp ./bin/GetLocations.html /var/www/localhost/htdocs
sudo cp ./bin/Index.html /var/www/localhost/htdocs
sudo cp ./bin/RemoveJob.html /var/www/localhost/htdocs
sudo cp ./bin/SendCommand.html /var/www/localhost/htdocs
sudo cp ./bin/SimpleJob.html /var/www/localhost/htdocs
sudo cp ./bin/UserFeedback.html /var/www/localhost/htdocs
```

Installing the command_task (from the aider/trunk/command/bin directory):

1. First:

```
sudo chmod 775 ./command_task
```

2. The command_task depends on the following files, which it expects to find in the same directory as the executable:

- configuration.ini = configuration file
- ExtendedDemoMap.txt = the map

3. The command task will create two log files

- trace.log - all debug trace
- notifications.log - all notification msgs received from the control_task

Running on the robot:

1. Initialize the message queues

```
./create_queues
```

2. Run the control task

```
./control_task
```

3. Run the command task (from the aider/trunk/command/bin directory)

```
./command_task -normal
```

Running on your VM:

Run (from the aider/trunk/command/bin directory)

```
./command_task -stubbed
```

Using the command_task:

Browse to the test pages (e.g. http://<yourip>/NewJob.html) to add jobs, fetch jobs, etc. or use the terminal interface (described below).

Command input interface:

The command task accepts and processes terminal input command.

The following commands are supported (all case-insensitive):

```
SendCCMsg <CommandType> <Arg1> <Arg2> <Arg3>   = Send message to the
control task
Movf <DistanceCms> [<TimeoutSecs>]  = Move forward
Rot <Degrees> [<Clockwise=1|0> [<TimeoutSecs>]] = Rotate
Taw <LandmarkId> <DistanceCms> [<TimeoutSecs>]  = Travel against the
wall
Stop
Elh = Enter left hallway
Erh = Enter right hallway
```

```
Efh = Enter forward hallway
trayup
traydown
ShowMap
Enable = Enable job processing
Disable = Disable job processing
Cancel = Cancel the current (in-progress) job
LoadChanged = Signal the command task that the load status has changed
(i.e. loaded/unloaded)
NewJob <DestinationLandmarkId> = Add a new movement job to a given
destination
```

Appendix B - AIDer Operating System and Development Environment

The original AIDer software from California State Polytechnic University ran on RTLinux-Free 2.6.9, based on a custom patchset for the Linux kernel and a Red Hat userland. RTLinux-Free 2.6.9 included real-time scheduling capabilities, but is no longer actively maintained. As such, new software was chosen and installed.

The AIDer onboard computer runs Gentoo GNU/Linux 2.6.26. GNU/Linux is a monolithic kernel, Unix-like operating system. Linux 2.6.26 includes real-time scheduling capabilities and is actively maintained.

The Gentoo Linux distribution is a highly customizable and configurable Linux distribution, and was chosen as certain development software requirements were not chosen until well after Control Task development was under way. According to the Gentoo Philosophy (see http://www.gentoo.org/main/en/philosophy.xml), "The goal of Gentoo is to design tools and systems that allow a user to do that work as pleasantly and efficiently as possible, as they see fit."

The AIDer userland includes:

- glibc 2.8 and the GNU toolchain, including GCC 4.3.2
- X.org 7.2 and Xfce 4.4.3
- Apache 2 Web Server with mod_python and mod_ruby
- Mozilla Firefox 3
- Ruby 1.8
- Perl 5.8.8
- Python 2.5.4

For convenience, AIDer's operating system and development environment were replicated on a Sun VirtualBox virtual machine to allow the project's members to test their software without using AIDer itself.