The 2009 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 11-15, 2009 St. Louis, USA

Using Real-time Awareness to Manage Performance of Java Clients on Mobile Robots

Andrew McKenzie, Shameka Dawson, Quinton Alexander, and Monica Anderson

***Abstract*— In this paper, we propose an extension to existing mobile robot development environments that explicitly declares the frequency requirements for client controller threads. This extension enables better use of robot resources by running modules only as fast as needed. Developers are forced to consider the frequency requirements and their impacts, which should result in better code. Physical experiments employed a K-team Koala robot. Preliminary results showing the effect of explicitly defining frequency are also presented.**

I. Introduction

Robot controllers provide the logic and intelligence for autonomous robot systems. Mobile robot controller development is decidedly different from creating controllers for traditional manipulators. Manipulators utilize specifically designed software to manage real-time scheduling and kinematic constraints. Developers are forced to work within this framework to ensure appropriate operation. Without such fail safes, manipulators can be very dangerous when failure to process data as designed results in incorrect robot motion that can damage the robot or cause harm to people. Mobile robot controller development is different. Mobile robot controllers are often built on top of a hardware API, leaving the software architecture design to developers.

Unfortunately, controller development for mobile robots can be deceptively easy. Behavior-based controllers often utilize finite state machines to manage many levels of complexity to create autonomous robots that appear to think for themselves. The standard design and programming practices of defining frequency requirements for controllers is not as important since the size and power of most mobile robots do not pose a threat to humans.

In both cases, the proper operation of robot hardware and software controllers assumes specific frequency requirements and resource availability to work properly. However, many mobile robot controller architectures do not enforce this requirement. This oversight is apparent when surveying current mobile robot architectures. According to a survey of nine popular robotic architectures [1], either real-time features (components that manage the frequency of processes) do not exist or must be purchased. Therefore, it appears that the assumption is that correct frequency is a function of resource availability. If enough resources are available then the controller will run properly. However, the assumption of correct frequency and resource availability on all but the most sophisticated systems may be incorrect.

The incorporation of off-the-shelf real-time software into mobile robots is not the ultimate solution. Managing resource constrained real-time systems require a specially trained engineer. IEEE Engineer reports that there is a growing shortage of embedded systems engineers [2]. This shortage is due in part to the lack of training in the engineering curriculum. Not only is the expertise thin, but the cost of off-the-shelf real-time software may be prohibitive.

Rather than advocating that all mobile robots use a real-time core system, we propose a real-time aware extension to existing mobile robot architectures that forces developers to declare frequency expectations for each control module. Missed deadlines (when a task does not complete in a certain timeframe) are reported both during testing and experiments allowing developers to understand the ramifications of code changes in terms of existing resources.

Two purposes can be served by making this fundamental change to robot architectures. First, explicitly determining the frequency requirements for a module can result in better resource utilization. Processes that do not need to run often can relinquish resources to those that do. Second, developers can understand the relative resources needed by individual components and can adjust code or hardware to meet the demand.

Clearly, precise system based timing control is unavailable unless the operating system supports it. However, hiding temporal details from developers may appear to result in easy programming but may cause many hours of sorting through unpredictable results. Inexperienced developers can exacerbate the problem by developing inefficient code.

In this paper, we describe a real-time aware extension to mobile robot architectures that allows developers to declare frequency requirements as part of the client interface. This extension works in the absence of a real-time core or other special software. Section II presents previous results. Section III describes the extensions and how they can fit into an interface. Section IV presents an implementation of the extension on top of Player [3] within the Java client library. Experimental setup is detailed in Section V. Section VI shows experimental results regarding changes in resource utilization and its effect on performance for a path-planning task. Section VII presents the analysis. Conclusions and future directions are in Section VIII.

This work was supported in part by the following NSF grants: IIS-0846976 and CCF-0829827.

A McKenzie, S Dawson, Q Alexander, and M Anderson are with the Department of Computer Science, University of Alabama, Tuscaloosa, Alabama 35487 USA (email: {awmckenzie, dawso003, qaalexander}@crimson.ua.edu; anderson@cs.ua.edu).

II. Related Work

Several robot development architectures are available for mobile robots. However, none of these environments provide either temporal awareness or real-time features. A complete treatment of robot development environments and their features is available in [1].

Subsumption-based robotic controllers build autonomy out of layers of behaviors [4]. Basic layers are complete processing units that take input and provide appropriate output. We are particularly interested in properties that enable temporal decompositions. Player [3] is an open source robot architecture designed to operate with a wide range of hardware components. The Player architecture specifically avoids defining decompositions and therefore does not provide a framework for adding frequency for behaviors.

Other robotic architectures do provide support for behavioral decompositions [5] - [7]. For example, ARIA programs [5] are composed of individual behaviors. Behaviors are each given a priority that is applied to the importance of actuator output. Priority does not affect resource assignment or utilization. Because the target hardware architecture only processes commands every 100 ms, all behaviors are run every 100 ms. Other threads can be added by the developer outside of the framework for processing tasks that have different frequency requirements.

It is instructive to examine the real-time architecture described in [8]. The architecture is based on port-based objects. Port-based objects are fully contained software components that contain both state and methods. They also provide input, output and resource ports. Input ports describe data needed by an object and output ports describe data produced by an object. An input port can only be connected to exactly one output port. Outputs of the same type must be merged into one unambiguous output. Each object is self contained and has its own frequency. Task scheduling is determined by the developer and can be modified to either increase system stability or reduce computational resource consumption.

State variables, not queues are used for inter-object communication. This model assumes the most recent data is always available so that tasks don't block while waiting for new data to be produced. However, a mechanism is required to insure mutually exclusive access. The paper suggests using a local copy of the data for processing to avoid the priority inversion associated with semaphores.

Player does create an abstraction of input and output ports referred to as interfaces. These interfaces define the data used by modules and allow sharing of data between modules. However, the inputs are implemented as queues and no temporal information collected about data is used at the client level. Outputs are mapped to a single device. There is no framework based facility to combine commands into a single output.

III. Extension Features

The proposed extension makes use of the subsumption-based decomposition of controllers. Robot controllers typically decompose behaviors into independent layers that process data and produce new data or robot actuation. Input data is generated from the robot's sensors or produced from other behaviors. Data is available to behaviors that use it. Behaviors process available data, creating new data or actuator commands destined for the robot. Multiple behaviors that can produce commands for the same actuator (for instance, wheel motors) manage resulting actuator output by a priority system.

We propose two extensions within the client development architecture: 1) specify and execute each behavior on its own scheduling period and 2) manage data production and consumption within the framework. The initial goals of these changes are to: 1) manage resource utilization, 2) provide behavior-based runtime profile information, and 3) facilitate shared data memory model. A secondary goal considers the application of these existing frameworks to a real-time operating system (RTOS). These extensions provide a mechanism within a RTOS to indicate thread priority, specify a schedule, and manage memory.

The extensions that we propose are comprised of three states to the individual behaviors (Figure 1). The Initialization state registers the behaviors with their frequency and data sharing requirements. It then verifies and subscribes to consumed data sources, which then creates shared data locations for produced data. Finally, it registers the produced data availability. In the Execution state, the monitor thread executes behaviors according to the user specified period and reports behavior execution into the next period. It then monitors the update of data sources and reports stale data and creates threads to manage multiple sources of data. Lastly, the Termination state deletes shared data and summarizes misses and data feed frequency.

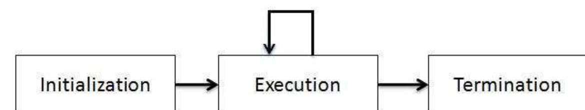

Fig. 1. A state diagram of individual behavior states.

A. Resource Utilization

In a real-time system, different tasks are scheduled to execute at individualized rates. These rates are chosen primarily to insure system stability but the rates at which data sensors can report new data may also be considered. System stability should be a consideration for mobile robots. If we apply these tenets to mobile robot controllers, behaviors that affect the stability of the system should run more frequently.

Determining the correct execution frequency for behaviors may require empirical testing. However, we can assume that tasks that affect system safety such as obstacle avoidance and stall recovery should run more frequently than mapping. From another perspective, mapping typically requires more system resources since it involves processing and storing larger amounts of data. Obstacle avoidance usually only

checks a few key values before determining if a controller command should be issued. Running mapping more frequently may cause the preemption of safety behaviors.

We propose that each behavior have a frequency associated with it. Behaviors associated with safety can run more often while resource intensive or less critical behaviors can execute less frequently. If input data is produced less frequently due to low sensor sampling rates, it may be possible to reduce the behavior's period without affecting overall system performance.

B. Memory Management

We propose the incorporation of shared data management similar to that proposed in [8]. All shared data requirements should be defined during the initialization stage. This allows global data allocation outside of the execution cycle. In addition, queues are not used so queue management functions that can be sensitive to the size of the queue are not needed.

Data that is not shared between behaviors is the purview of the client developer. Developers are free to allocate and deallocate memory during execution within a behavior. However, allocation and deallocation of memory can be costly in terms of time and is generally avoided in real-time systems. Impact of such choices may result in missed deadlines due to longer than expected processing times. Data regarding processing times may hint at problems due to time intensive requests.

To maintain data coherency and serialized access, each behavior allocates a local buffer for data during initialization. Global data is copied to the local store at the beginning of every execution cycle. The global data store is locked from writing during the copy but any number of consumers have read access. Producers update the global store at the end of the execution cycle during which all consumers are locked out. An aggressive schedule that requires behaviors to run frequently could cause lockout issues. This situation is apparent through reporting of missed deadlines or stale data.

C. Reporting

Missed deadlines and data availability are reported during execution and aggregated upon termination to provide the developer a summary of the temporal performance of the system. Developers can use this information to modify an overly aggressive schedule or to streamline code where possible. These metrics are not intended to eliminate the need for detailed profiling. It is possible that these metrics may indicate the need for more investigation via traditional profiling tools.

IV. Implementation

Player was modified to include the proposed extensions. Player is an open source robot architecture that interfaces abstracted clients to various hardware platforms. Clients are written in C, C++, Python or Java and can run locally or remotely via sockets. The Player server typically runs locally and contains drivers that translate generic commands to specific hardware interfaces.

Player was selected for two reasons. It does not provide a framework for behaviors. The developers intentionally decided not enforce a controller methodology. Instead the choice of controller framework is left to the framework user. Player also explicitly defines data sources as interfaces. Interfaces are an abstraction of data produced or consumed by behaviors. The explicit definition makes passing data between modules possible.

Although the proposed extensions are beneficial to any language implementation, Java presents a unique opportunity. It has been suggested that programs written in Java take less time to code and/or contain fewer errors [9]. Unfortunately, the features that make Java a particularly productive language make its use in embedded or real-time systems particularly challenging. Garbage collection reclaims unreferenced memory allowing programmers to allocate memory without creating large memory leaks. However, garbage collection preempts other threads contributing to unpredictable processing times. In addition, as an interpreted language, processing times can be longer than C or C++.

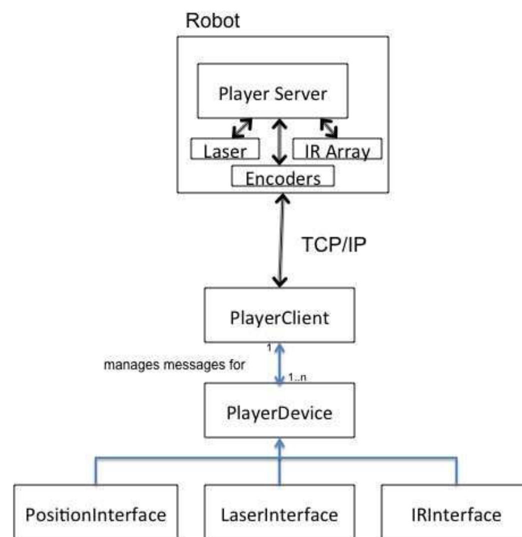

Fig. 2. Javaclient [10], a Java client library that interfaces to Player.

The extensions were implemented as part of the Java client library [10]. The client library manages communication with the Player server and exposes device specific functionality through interfaces (Figure 2). The extended version (Figure 3) moves robot interaction to a PLAYERDATATASK, which updates the global data stores with current data. In the original Javaclient, the library supported up to two threads, one for the client program and another for handling communication if requested. Other threads could be created and called from the client program. In the extended architecture, there are several new threads built into the framework. The MONITOR is created by the client program to manage behavior execution and Player communication, each in its own thread.

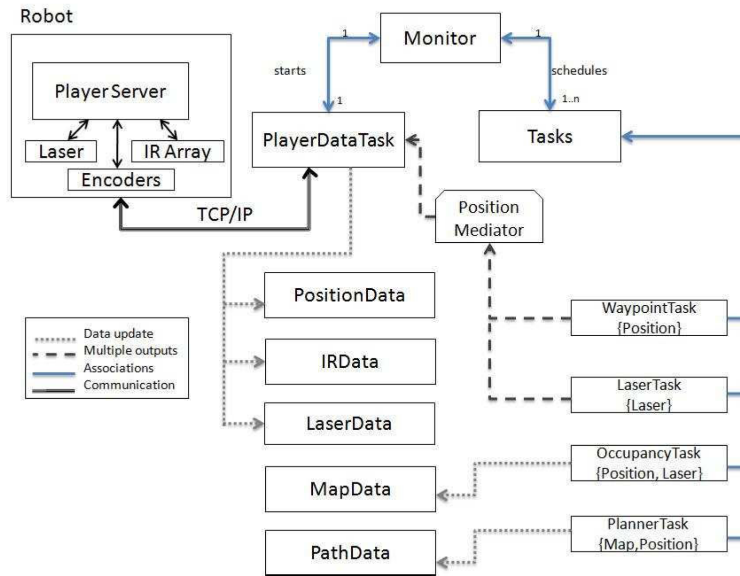

Fig. 3. Real-time aware framework implemented in Java in conjunction with the Player architecture.

The framework has five key components, a monitor and four task threads. The four task threads are:

* WAYPOINTPLANNER (Waypoint) is a high-level behavior that moves the robot through a series of waypoints. Although waypoints can be read in or set by another task, a set of pre-defined waypoints was used.
* LASEROBSTACLEAVOIDANCE (Obstacle) responds to obstacles sensed via the laser by slowing down the robot and turning away from the obstacle.
* OCCUPANCYGRIDMAPPING (Occupancy) uses the position and laser data reported by Player and maintains an occupancy grid map. The map tracks unknown, open and occupied space.
* DIJKSTRAPATHPLANNER (Planner) takes goal position requests and maintains a path from the current position to the closest goal. Multiple goals may be added and the requester must remove the goal when it is no longer valid (has been reached).

Other important modules include the POSITION MEDIATOR. It takes motion requests from LASEROBSTACLEAVOIDANCE and WAYPOINTPLANNER and determines what commands are ultimately sent to the robot. If LASEROBSTACLEAVOIDANCE requests control of the motors, its commands are executed since it is the higher priority task. Otherwise, commands from WAYPOINTPLANNER are executed.

The MONITOR starts tasks at the pre-specified intervals and reports when a task does not meet its deadline. The tasks run independently but there are relationships according to how data is generated and consumed. For instance, a map of the environment is generated in one thread and that map is used by another thread to identify waypoints. The laser data is needed by both LASEROBSTACLEAVOIDANCE and OCCUPANCYGRIDMAPPING. However, the LASEROBSTACLEAVOIDANCE thread does not rely on the robot's position data.

Behaviors are created by subclassing TASKS. The TASKS superclass contains functionality needed for initialization and registration of data store use for the behaviors (shown in Figure 4). TASKS provides methods for registering production or consumption of data with the monitor. During execution, the TASKS superclass method replicates consumed data to a local store. The MONITOR class handles scheduling execution according to the specified schedule type: continuous, timed or one-time. Timed tasks are executed according to their specified period. The MONITOR class also reports deadline misses and when a task is running with no new data.

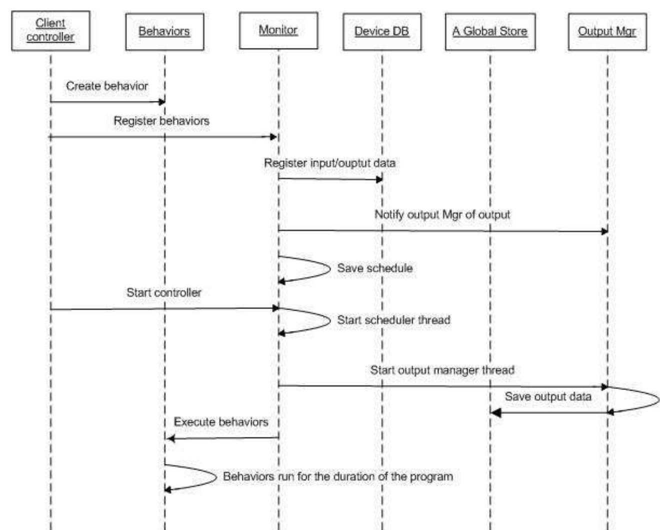

Fig. 4. Sequence diagram showing the interaction and order of the initialization phase. Each behavior notifies the monitor of data that is produced and consumed.

Some of the issues with using Java in embedded systems have been addressed in Java Real-time Systems (RTS) [11]. It implements real-time features such as no heap real-time threads, thread priorities, scoped and immortal memory. Some flavors of Linux such as SuSE and Solaris have early implementations. The proposed framework extensions explicitly separate memory into global shared and thread local for application of immortal types and no heap threads. Also, behaviors as threads allow for the application of different priorities to different tasks.

V. Experimental Setup

Experiments were performed to measure the usefulness in applying different frequency requirements to behaviors. These frequencies were supplied by the user via the command line. A waypoint navigation controller was used to measure the effects of frequency on task performance.

Fig. 5. K-Team Koala Robot equipped with an ETX-Nano computer for control and a Hokuyo URG laser ranger.

Physical experiments employed a K-Team Koala robot (Figure 5). The robot was equipped with an Acces I/O ETX-Nano computer which has an Intel Core Duo 1.66GHz processor, 2GB of RAM and an 8GB compact flash card for storage. Instead of using the on-board IR proximity sensors that the Koala is augmented with, a Hokuyo URG laser range finder was utilized. Player handles the interface to the robot and its devices. The Player server as well as the logic or control code was executed on-board the robot. Because of the high power requirements of the Acces I/O ETX-Nano, the robot was tethered with a power cord and an ethernet cat5 cable for remote communication. The testing environment was a 9.6m x 6.2m room (Figure 6).

In these experiments, all trials were given the same four waypoints to reach (shown in Figure 6). From the start position (X_1), the robot would travel to each waypoint until it arrived at the finish position. The waypoints were chosen such that all behaviors needed to be employed to reach the ending position. Each task was given an interval value that defined both the task's deadline and period. For comparison, the robot was run with different interval values for the task threads (shown in Table I). The initial period values were based on the frequency of the URG laser, which is 100 ms. The period for each set thereafter was chosen based on

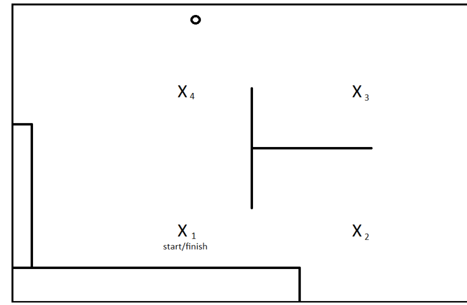

Fig. 6. Experiment room (9.6m x 6.2m) with the four chosen waypoints was equipped with ground truth positioning sensors.

observation. Each set of experiments had a total of ten runs. In three of the five experiment sets, all tasks were assigned the same period. In the remaining two sets, the period of the LaserObstacleAvoidance was varied to see its affect on the overall performance of the experiment. We chose to lower this period because of safety issues of both the robot and environment.

It was hypothesized that these experiments would help demonstrate the effect of task frequency on the overall system. We believe that the frequency of each task impacts the performance of the system independently. Tasks should not be permitted to run either too fast or too slow. For instance, if LaserObstacleAvoidance is run at too fast a period then it may cause the robot to hit an obstacle. Since OccupancyGridMapping uses both laser and position data, it is critical that it run at an appropriate rate so that the map is updated correctly. An incorrect map may cause DijkstraPathPlanner to plan an incorrect path and WaypointPlanner to move through an obstacle.

Trials were considered complete if they circled the obstacle approaching and passing each waypoint. If the robot either got stuck on an obstacle or did not reach all waypoints after six minutes, the trial was considered incomplete. Missed deadlines and run time for each trial was recorded.

VI. Experimental Results

Table I summarizes the results of the experiments. Each experiment set tested different intervals of periods to determine the effect of timing on the program's performance. Good performance was based on the following categories: run time, distance to the finish position and low standard deviation.

Figure 7 shows the percentage of missed deadlines for each individual task for each Set. It is shown that Set 1's individual tasks had the highest percentage of missed deadlines. The percentage of missed deadlines for all tasks was recorded for each experiment set (see Figure 8), which followed the same trend as missed deadlines per task.

The time that it took each run to finish as well as the robot's distance to the finish position were recorded for all runs. The average and standard deviation was then calculated for time (see Table I) and position (see Table II).

TABLE I

EXPERIMENTAL RESULTS.

Experiment Set	Behavior Periods (ms)				Trials	
	Waypoint	*Obstacle*	*Occupancy*	*Planner*	*Average Run Time (s)*	*Standard Deviation (s)*
1	10	10	10	10	266.95	6.64
2	100	10	100	100	267.34	10.39
3	50	50	50	50	265.89	11.19
4	100	50	100	100	276.97	5.17
5	100	100	100	100	272.68	6.28

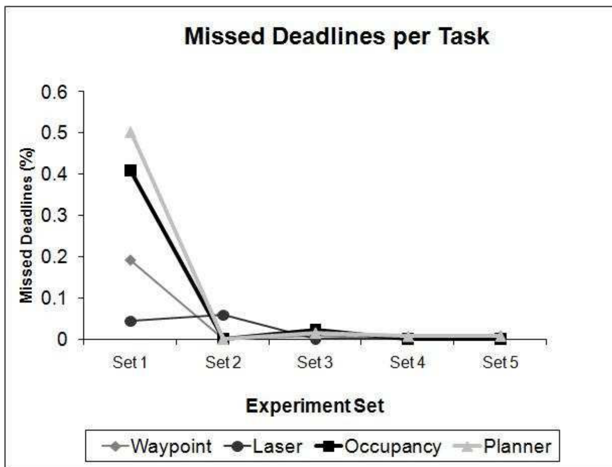

Fig. 7. Missed deadlines for each individual task.

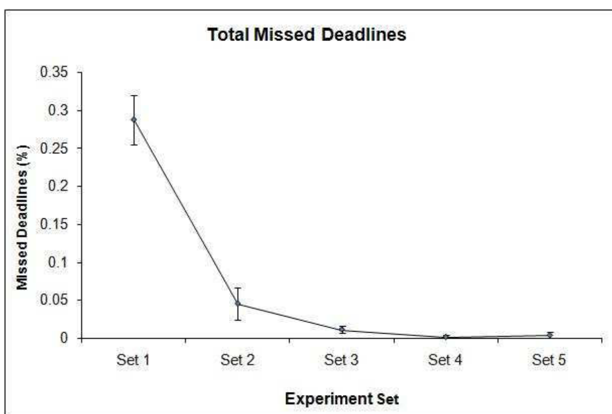

Fig. 8. Percentage of missed deadlines for each experiment set.

TABLE II

DISTANCE FROM FINISH POSITION.

Set	*Average Distance (m)*	*Standard Deviation (m)*	*Min Distance (m)*	*Max Distance (m)*
1	2.19	0.17	2.04	2.45
2	2.21	0.23	2.02	2.59
3	2.18	0.21	1.96	2.45
4	2.14	0.43	1.80	2.86
5	1.93	0.17	1.70	2.12

VII. ANALYSIS

Figure 7 shows that as the individual tasks' periods increase, the tasks missed deadline rate decreases. In Sets 1, 4 and 5, the tasks' with the highest missed deadline rate are DIJKSTRAPATHPLANNER followed by OCCUPANCYGRIDMAPPING, WAYPOINTPLANNER and finally LASEROBSTACLEAVOIDANCE. The computational requirements of the tasks follow in that order. Figure 8 shows that as the tasks' periods increase, the overall missed deadline rate decreases. When the tasks' periods are low, the system has less computation to do then compared to when the periods are high.

Table I shows that although Set 3 has the fastest average run time, it also has the highest standard deviation. The run time values and distance from the finish position are almost identical (Tables I - II). T-tests were used to determine if the results are statistically significant. The t-test results for distance from the finish show that the data sets were not statistically different. The total run times from Sets 1 and 4 were statistically different as well as Sets 1 and 5. Since the only difference between the sets were the frequencies of behaviors, the results suggest that task timing does affect the overall system performance. It is also worth noting that the t-test results for run time values of Sets 2 and 4, Sets 3 and 4, and Sets 4 and 5 had confidence intervals of 82%, 85% and 79% respectively. Therefore those sets are on the verge of being considered statistically significant. Note that the only difference in Sets 2 and 4 and Sets 4 and 5 is the period assigned to LASEROBSTACLEAVOIDANCE; the other tasks all have a period of 100 ms. It is also worth noting the standard deviation for Sets 2 and 3 are almost double of Sets 1, 4 and 5.

Experiments were also run with periods larger than 100 ms. These experiments were problematic because the robot would hit the wall as well as obstacles. Experiments were also run with periods of 5 ms and less, which resulted in the robot becoming non-responsive and locking up which would require a reboot of the system. This supports our hypothesis that a system should not be run too fast or too slow. In control theory [12], the sampling rate needs to be at least two times the fastest input signal. In our system, the Hokuyo laser sends out its data at a rate of 10Hz. Therefore, any behavior in our control system that depends upon the laser information (LASEROBSTACLEAVOIDANCE and OCCUPANCYGRIDMAPPING) should have a period of at most 50 ms. WAYPOINTPLANNER uses position information that updates

on average at 16Hz, suggesting that WAYPOINTPLANNER may need to run with the faster period. In contrast, even though the position information is updated from the robot at 16Hz, the grid size is only 0.5 meters. With the top speed of the robot being 0.1 meters per second, the occupancy cell only changes at most once every 5 seconds. This suggests that available resources may be increased by slowing down the DIJKSTRAPATHPLANNER. This may be significant given the resources needed to replan, even in a large resolution grid.

VIII. CONCLUSION

Many robot architectures view correct frequency as a function of resource availability. By not properly addressing timing concerns mobile robot systems may not fully utilize system resources. Furthermore, ignorance of timing details can lead a developer to produce unstable systems. This is partially because our intuition is that only the critical tasks of a robot control program need to run fast, which is wrong. One must apply the laws of Control Theory to robotics, requiring all behaviors and their input dependencies to be analyzed instead of just the critical tasks.

We presented a means of extending a robot software platform to incorporate temporal awareness. The system, written in Java and added to the Player interface, allows individual program components to be assigned a schedule frequency. Although it cannot override the underlying OS, this system can report missed deadlines and enforce subtask hierarchy and priority through application of scheduling intervals. We presented the details of the system and its implementation. Experiments demonstrated how manipulating the timing of subtasks affects overall task performance. The results of testing validate the hypothesis and show how important proper timing is to resource utilization in mobile robot systems.

In future work, the architecture will be extended to Java RTS, which is required to run on a RTOS such as Solaris. The JVM will also provide run time information that can be used to adjust the timing behaviors automatically. This extension will require a better understanding of appropriate frequencies and their relationship to code features. This includes which tasks depend on specific input devices, such as the URG Laser, as well as the dependencies between the different tasks. Then Digital Control System theory can be used to find the required task periods. The hardware will be varied to show the effects of different hardware on timing and resource requirements. The architecture will also be written in a more real-time friendly language such as C. The next step will be to map the entire architecture to a RTOS, in turn making it actually real-time instead of real-time aware. Human studies that evaluate programmer proficiency may shed light on features of the architecture that are useful in teaching embedded programming techniques.

IX. ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the following NSF grants: IIS-0846976 and CCF-0829827.

REFERENCES

[1] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007.

[2] M. Anderson, "Help wanted: Embedded engineers why the united states is losing its edge in embedded systems...," *Today's Engineer*, February 2008.

[3] B. Gerkey, R. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. Mataric, "Most valuable player: A robot device server for distributed control," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), Maui Hawaii*, 2001.

[4] R. Brooks, "Intelligence without Representation," *Artificial Intelligence*, vol. 47, no. 1-3, pp. 139–159, 1991.

[5] K. Konolige, "Saphira robot control architecture," Technical report, SRI International, Menlo Park, CA, April 2002, Tech. Rep.

[6] T. Balch. (2001) TeamBots software and documentation.

[7] M. Scheutz, "ADE: STEPS TOWARD A DISTRIBUTED DEVELOPMENT AND RUNTIME ENVIRONMENT FOR COMPLEX ROBOTIC AGENT ARCHITECTURES," *Applied Artificial Intelligence*, vol. 20, no. 2, pp. 275–304, 2006.

[8] D. Stewart, D. Schmitz, and P. Khosla, "The Chimera II real-time operating system for advanced sensor-basedcontrol applications," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 22, no. 6, pp. 1282–1295, 1992.

[9] G. Phipps, "Comparing observed bug and productivity rates for Java and C++," *Software—Practice & Experience*, vol. 29, no. 4, pp. 345–358, 1999.

[10] R. Rusu, R. Robotin, G. Lazea, and C. Marcu, "Towards Open Architectures for Mobile Robots: ZeeRO," *Proceedings of the Automation, Quality and Testing, and Robotics International Conference (AQTR 2006), May*, 2006.

[11] G. Bollella and J. Gosling, "The Real-Time Specification for Java," *COMPUTER*, pp. 47–54, 2000.

[12] J. Dorsey, *Continuous and Discrete Control Systems*. New York: McGraw-Hill College, 2001.

[13] S. Bennett, *Real-Time Computer Control: An Introduction*. New York: Prentice Hall International, 1994.