ISSN 0280-5316
ISRN LUTFD2/TFRT--5876--SE

Resource Management for Mobile Robots

Mikael Kralmark

Department of Automatic Control
Lund University
August 2010

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
August 2010

Document Number
ISRN LUTFD/TFRT--5876--SE

Author(s)
Mikael Kralmark

Supervisor
Mikael Lindberg Automatic Control Lund, Sweden
Karl-Erik Årzén Automatic Control Lund, Sweden (Examiner)

Sponsoring organization

Title and subtitle
Resource Management for Mobile Robots. (Resurshantering för mobila robotar)

Abstract
As more functionality and complex systems are combined in limited settings the need for resource management is becoming increasingly important. Resources such as computing capacity, battery power and communication channels need to be divided between different applications and hardware to achieve the highest global performance. Traditionally the computing capacity has been handled by the system scheduler, but in dynamic systems with complex usage scenarios the system analysis that needs to be done off-line in order to guarantee stability does not only grow more unfeasible, but since such an analysis must be done considering the worst case scenario, this may provide an unnecessarily pessimistic result. In the case of mobile robotics, a key feature is the ability for the robot to adapt to a changing environment, the usage scenarios may vary from one time to another, the task to be performed could vary and the surroundings of the robot may change. Heavy computing load may cause specific parts of the system to overheat, thereby limiting the performance. To increase the performance and to make the system more flexible in terms of new hardware configuration, an adaptive resource management framework is needed. An adaptive resource management framework that does the system analysis on-line would reduce the configuration time, and increase the performance and flexibility.
In this thesis, an existing resource managemet framwork has been ported to a mobile robot patform. The thesis also proposes a method for controlling the CPU temperature, by using the system utilization and adaptive resource allocation as a mean to keep the CPU temperature bounded.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
40

Recipient's notes

Security classification

Contents

1. Introduction

1.1 Problem formulation and objectives

As more functionality and complex systems are combined in limited settings the need for resource management is becoming increasingly important. Resources such as computing capacity, battery power and communication channels need to be divided between different applications and hardware to achieve the highest global performance as illustrated in Figure 1.1.
Traditionally the computing capacity has been handled by the system scheduler, but in dynamic systems with complex usage scenarios the system analysis that needs to be done off-line in order to guarantee stability does not only grow more unfeasible, but since such an analysis must be done considering the worst case scenario, this may provide an unnecessarily pessimistic result.
In the case of mobile robotics, a key feature is the ability for the robot to adapt to a changing environment, the usage scenarios may vary from one time to another, the task to be performed could vary and the surroundings of the robot may change. Heavy computing load may cause specific parts of the system to overheat, thereby limiting the performance.
If, e.g. large amount of data is to be collected from the robot while it is running, this needs to be handled without the robot misbehaving due to insufficient CPU bandwidth. Also, the ability to mount new hardware without having to do extensive system analysis would provide a more flexible system.
To increase the performance and to make the system more flexible in terms of new hardware configuration, an adaptive resource management framework is needed. An adaptive resource management framework that does the system analysis on-line would reduce the configuration time, and increase the performance and flexibility.
In this thesis the Resource Manager (RM) that is being developed within the ACTORS project (www.actors-project.eu) has been ported to a mobile robot (Figure 3.1). By using feedback control the RM handles the scheduling on-line, thereby increasing the global performance of the system.
Even though the RM at the time of writing this, is not fully implemented and some important features are missing, the basic functionality of such a framework can be tested and one might be able to draw conclusions of what kind of system would be useful for adaptive resource management in robotics.
Most parts of the system has been developed prior to this thesis so the most effort has been made on getting all parts to work together, both to configure the system to run the RM and to configure the RM to run on this specific system, and also to adapt the robot software to be able to work with the RM.
Since the Actors RM is just partly implemented, only the existing parts have been tested here, and some “ad hoc” solutions have been implemented to get everything to work. The RM has also been extended with a CPU thermal controller that uses adaptive resource allocation as a mean to keep the CPU temperature bounded.

1.2 Outline

In Chapter 2, some tools that have been used are presented, and the different parts of the system are described in Chapter 3. In Section 3.1 the robot is described, and Section 3.2 describes the Earliest Deadline First kernel that has been used. Section 3.3 gives an introduction to the Resource Management Framework, both a summary of the internal structure and its main features, and the requirements made on client applications. The robot control application is described in Section 3.4, and in Section 3.5 is a description of a hardware monitoring system that has been used for the thermal control.
Chapter 4 explains how the different parts of the system have been adjusted to fit together. The upgrades made to the system in order to run the resource management framework is covered in section 4.1. How the recourse management framework has been adapted to be able to run on this system is described in Section 4.2. Section 4.3 describes a client application that has been used in this project. In Section 4.4 is a description of how the robot software has been adapted to be able to interact with the resource manager. Section 4.5 describes how the temperature control of the CPU has been implemented.
Chapter 5 covers the experimental setup, and also presents some results, and Chapter 6 concludes with some discussion.

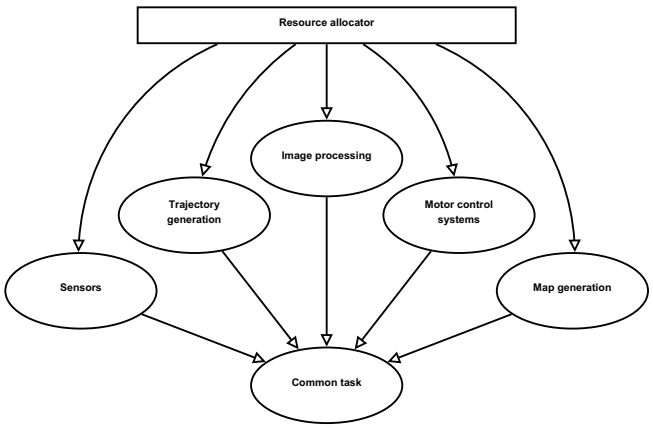


Figure 1.1 An example of how different parts of a system must work together to achieve a common goal

2. Tools

Listed below are some software tools that has been used in this project.

D-Bus

The D-Bus message bus system is a simple way for applications to talk to each other. The low-level reference implementation uses an XML representation for messages so the only required dependency is an XML parser. This makes it independent of the platform and of the implementing method for client applications. Most Linux systems already makes heavy use of D-Bus for communication between desktop applications and different D-Bus bindings is present in most Linux repositorys. More information on how to use D-Bus can be found on www.freedesktop.org/wiki/Software/dbus.

Qt and Qt D-Bus bindings

The Qt C++ libraries are a powerful cross-platform application and UI framework. Especially important to this project are its C++ to D-Bus bindings, that enables generation of D-Bus messages from C++ methods, and connecting methods to D-Bus signals. (qt.nokia.com)

C/C++

C++ has been the implementing method for all code in this project. C++ combines an object oriented environment with the powerful C code standard.

Matlab

MathWorks Matlab has been used to analyse and present results.

3. System description

3.1 Pioneer P3-DX

The robot used in this project is a Pioneer P3-DX from Mobile Robots [MobileRobots Inc, 2006]. It has an internal computer [Versalogic Corporation, 2007] currently running a Debian Linux operating system and it provides the possibility to mount different kinds of hardware and to run numerous different types of software. In its present configuration the robot has two SICK laser scanners (ref) to map the area around it, but other features such as stereoscopic vision and gripper could be mounted, which makes the robot very flexible. This robot is used in several projects at Lund University, involving e.g. more accurate position control with visual feedback from wheel position and there are also thoughts of equipping it with a projector making it able to present visual information, and also to generate 3D maps using structural light in combination with a camera.

Player

The communication between the internal computer and the robot micro controller is done through Player which is a free software tool for robot and sensor applications. This tool is used by more high-level software to control the motors or read sensor values from the robot (playerstage.sourceforge.net).

Figure 3.1 The Pioneer P3-DX robot. [conscious robots, 2007]

3.2 SCHED_EDF kernel

EDF (Earliest Deadline First) scheduling is a dynamic scheduling algorithm used for real time tasks. It places processes in a scheduling queue were the process that is closest to its deadline is allowed to execute. The theoretical scheduleability condition for the EDF scheduler is stated as:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{3.1}$$

where C_i is the worst case execution time and T_i is the period of task i, which is assumed to be equal to the deadline. This means that the EDF scheduler can guarantee that all deadlines are met provided that the total CPU utilization U is not more than 100% [Årzén, 2008]. EDF scheduling ability is not part of the linux mainline kernel, but an implementation of the EDF scheduler for linux has been developed within the ACTORS project [Scordino and Trimarchi, 2009]. With this kernel a new system call has been introduced, giving EDF scheduling policy to tasks. The assignment of scheduling parameters is done through the linux control groups virtual file system (cgroups). The cgroups system uses a hierarchical structure where the tasks placed at top level are given the highest priority. EDF tasks can run simultaneously with the ordinary scheduler, but are given higher priority, providing that the relation in Equation 3.1 still holds. Three new subfiles have been added to the cgroup system.

- `cpu.edf_period_us`
- `cpu.edf_runtime_us`
- `cpu.edf_reservation_data`

The `cpu.edf_period_us` and `cpu.edf_runtime_us` files are used as period and budget for the EDF scheduling, corresponding to T and C in Equation 3.1, and the `cpu.edf_reservation_data` file is used for feedback of how the task is using the reservation it is given.

3.3 ACTORS Resource Manager

Overview

The Actors RM works at a global level, collecting information about all applications that intend to run on the system and decides about the distribution of resources [Årzén *et al.*, 2009]. The applications can be any type of applications as long as they meet the requirements described below. The actual scheduling is done by the Linux scheduler, and the RM uses Linux control groups described in Section 3.2 to create reservations, so a Linux system is necessary to run the RM. Communication with the RM is done through D-Bus (Section 2) which makes it independent of the way clients are implemented, e.g. in Java, C++ etc.
Client applications that intend to work with the RM register once they have started and unregister when they have finished executing. Once an application is registered it will be given one or more virtual processors that will be placed on the physical cores, a virtual processor is represented as a C++ class with two main attributes, α and Δ, corresponding to the CPU share and time granularity the application will be given by the system scheduler.
Figure 3.2 shows an overview of how the RM handles the communication between client applications and the system scheduler.

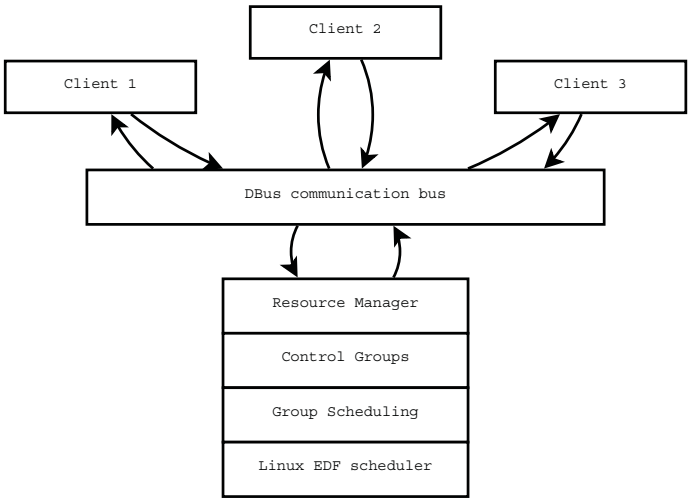


Figure 3.2 Overview of the communication between three clients, the RM and the EDF scheduler.

The RM will create new folders for the actors clients at the top level of the cgroups system, giving these tasks the highest priority and leaving other, non Actors tasks to use the bandwidth that is left free. 10% of the CPU will be reserved to the RM's control loop and to initial reservations, and 80% to Actors tasks, so 10% of the CPU is left for non Actors tasks. To decide how an application should run, the RM solves a linear optimization problem, and to do this the RM needs the following information from the applications.

Application importance The application importance value is used to decide which applications that should be granted more resources on an overloaded system.

Category The client applications will belong to one of three different categories, low, medium or high. This value is only used when creating the initial reservation before the application is granted a share of the CPU.

Threadgroups An application can consist of a number of threads grouped together in one or more threadgroups. Each threadgroup is given a virtual processor with a fixed period and a budget that will be controlled by the feedback controller of the RM.

Service levels The service levels structure is one of the most important features in the ACTORS RM. The service levels defines the different ways a client application can execute, with different resource demands and different performance. Each client application announces its service levels in the registration process with a D-Bus message, but in the RM a service level is a struct defined in the following way.

Listing 3.1 The service level struct

```
struct ServiceLevel
{
  unsigned int qualityOfService;
  unsigned int totalBandwidth;
  unsigned int granularity;
  unsigned int bWDistributionDataSpecifier;
  unsigned int bWDistributionCount;
  BandwidthDistributionMap bWDistribution;
};
```

- qualityOfService

  The integer `qualityOfService` defines the quality of this service level, 0 - 100 where 100 means highest quality. This value together with the application importance value is used when solving the optimization problem for assigning service levels to clients. The RM will maximize the quality of service delivered by all registered clients limited by the available bandwidth on the system. What good quality of service means is not defined, it could mean high performance or save battery power.

- totalBandwidth

  `totalBandwidth` is an integer specifying the the total bandwidth the client need at this service level, e.g. 200 means that the application needs in total 2 complete cores.

- granularity

  This value indicates the time granularity for all VPs of the client at this service level. If e.g. an application needs 20% of one core every 100 ms the `totalBandwidth` should be 20 and `granularity` should be 100.

- bWDistributionDataSpecifier

  Specifies how the values in the following BWDistributionMap should be interpreted.

  - 0: no data specified by application
  - 1: absolute values
  - 2: relative values

- bWDistributionCount

  Number of entries in the bWDistribution map.

- bWDistribution

  Maps the different threadgrops to a value of required bandwidth. The values should be interpreted according to the value specified by `bWDistributionDataSpecifier`.

Assigning service levels

When assigning service levels to client applications the RM will try to maximize the total quality of service delivered by the system. This is done by solving an optimization problem defined as:

$$\textit{Maximize}:$$

$$Z = \sum_{a=1}^{nApps} I_a \sum_{s=0}^{nSL_a} Q_{a,s} x_{a,s} \tag{3.2}$$

$$\textit{Subject to linear constraints}:$$

$$\sum_{a=1}^{nApps} \sum_{s=0}^{nSL_a} B_{a,s} x_{a,s} \leq L \tag{3.3}$$

$$\textit{With bounds on variables}$$

$$x_{a,s} = 0 \textit{ or } 1 \tag{3.4}$$

$$\sum_{s=0}^{nSL_a} x_{a,s} = 1 \tag{3.5}$$

$$\textit{Where}:$$

$$\begin{aligned}
nApps &= \textit{The number of registered applications.} \\
nSL_a &= \textit{The number of service levels for application } a. \\
Q_{a,s} &= \textit{Quality of service for application } a \textit{ at service level } s. \\
I_a &= \textit{The application importance value.} \\
B_{a,s} &= \textit{Bandwidth required for application } a \textit{ at service level } s. \\
L &= \textit{The upper limit of bandwidth for actors tasks.} \\
x_{a,s} &= \textit{A binary value that is true } (1) \textit{ if application } a \\
&\quad \textit{should be at service level } s, \textit{ and false } (0) \textit{ otherwise.}
\end{aligned}$$

Equation 3.2 maximizes the sum of quality of service, delivered by the individual applications, weighted by the application importance. It is bounded by Equation 3.3 that states that the sum of required bandwidth for the applications must not exceed the total bandwidth reserved to ACTORS tasks.
The solution to this problem is the binary array x. The bounds in Equation 3.4 and 3.5 state that one and only one of $x_{a,0} \ \ldots \ x_{a,nSL_a}$ will be true, i.e. application a will be given one service level.

Assigning physical cores

Once the RM has decided what service levels to use for the client applications there is another optimization problem to solve, how to place the virtual processors on the physical cores. The RM will keep 10% of one core to initial reservations and to the RM control loop, and leave 10% on each core free for non Actors applications so there will be 80% reserved to Actors applications on the first core and 90% on the other, if any, cores. Each threadgroup will have its own virtual processor and this can not be split on different cores.
Figure 3.3 shows an example of how this assignment of virtual processors could be done.

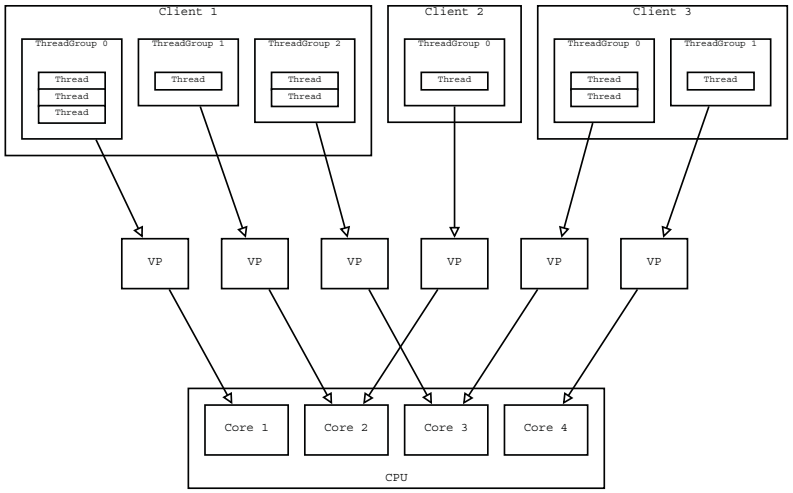


Figure 3.3 Overview of how the threadgroups can be placed on the physical cores.

Application requirements

Applications that intend to work with by the RM needs to be "ACTORS-aware", which means that applications must register with the RM at startup and follow instructions during runtime. If the application has multiple service levels the RM can give instructions to change service level, it is then assumed that the application do so. The reserved bandwidth will change, so if the application keeps running in the same service level this would cause it to either e.g. missing deadlines due to insufficient bandwidth or to decrease the global performance when not using the bandwidth it is given and thereby limiting the resources available to other applications.

Registration The registration process consists of a number of calls to the RM over D-Bus. This gives the RM the necessary information of the applications existence, initial CPU-category and its different service levels and treads. An application can consist of only one service level, but it still has to go through the complete registration process to be accepted by the Resource Manager. When all information has been provided a finalizing commit call is done to register the application. To complete the registration, six methods have to be called in the order they are listed below.

Method

int registerApp(**string** applicationId)

This method will inform the RM of the clients existence and the name of the application. No reservation is created since this is just the first of the six necessary calls to the RM. The application name will be mapped against an internal list to determine the application importance. If the application has a name that does not have a pre determined importance it will be given a default value.

Parameters
`string applicationId`: The name of the application to register.

Returns
Integer value, 0 on success, error code otherwise.

Method

int announceCPUCategory(**int** category)

With this call the applications CPU category is made known to the RM.

Parameters
`int category`: The category to be used for creating the initial reservation.

Returns
Integer value, 0 on success, error code otherwise.

Method

int announceServiceLevels(**int** initialServiceLevel,
int nbrOfServiceLevels,
list<ServiceLevel> ServiceLevels)

This call will let the RM know what service levels the application has, and what service level it will start to run in.

Parameters
`int initialServiceLevel`: The index into the following serviceLevels list, at which the application is running initially.

`int nbrOfServiceLevels`: The number of entries in the following serviceLevels list.

`list<ServiceLevel> ServiceLevels`: A list of serviceLevels, as specified in Listing 3.1.

Returns
Integer value, 0 on success, error code otherwise.

Method

int createThreadGroup(**int** groupId)

Creates an empty threadgroup to be filled with threads. The groupId is an integer value that can be chosen freely by the application, the groupIds must be be unique per application only.

Parameters
`int groupId`: The id of the group to create.

Returns
Integer value, 0 on success, error code otherwise.

Method

int addThreadsToGroup(**int** groupId,
int nbrOfThreads,
list<int> threadIds)

This call adds the thread IDs of the threads in the application.

Parameters
`int groupId`: The id of the group i which to place the following threads.

`int nbrOfThreads`: The number of entries in the following list of threadIds.

`list<int> threadIds`: A list containing the Ids of the threads to be placed in this group.

Returns
Integer value, 0 on success, error code otherwise.

Method

int commit()

This call will make the client active. The RM will now solve the optimization problems to assign service levels and physical cores to the registered application.

Returns
Integer value, 0 on success, error code otherwise.

Running Once the application is registered it could send feedback to the RM about how it is performing with the CPU time it is given. The application can report its happiness which is a number in the range of 0 - 100, where 100 means the application is performing at its best. The RM can measure the bandwidth used by an application so the happiness value can be used to find out if there is another resource demand, such as data flow from another applications or access to an IO device that is not met.
The RM can send signals to application when they should change service level. This is done through a D-Bus signal `changeServiceLevel` and the applications are expected to be connected to and listen to this signal. Figure 3.4 shows an overview of how the communication between two applications and the RM is done.

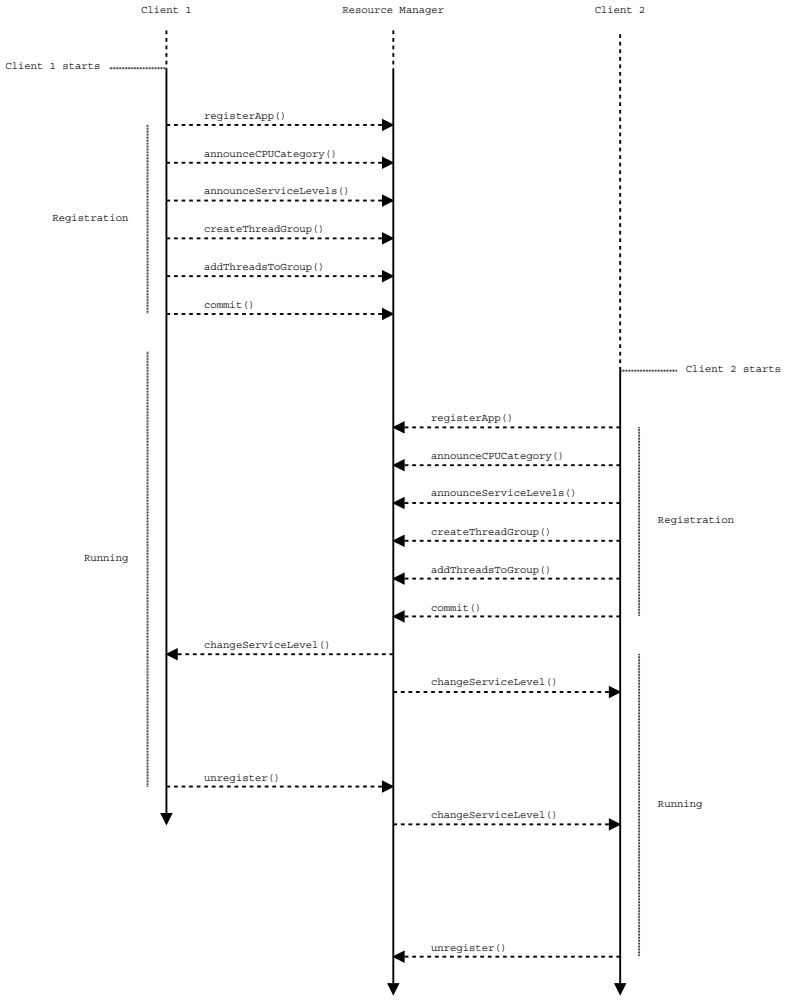


Figure 3.4 Timeline of two clients running with the RM.

Current state

What described above is the way the RM is intended to work, some parts are however not yet implemented. Below is a list of things that is not working yet:

Relative values of resource demands The relative values of the `bWDistributionDataSpecifier` in the service levels (see Listing 3.1) are not implemented, so for an application to run as is should one must specify the absolute values of the resource demands. All values given in the `totalBandwidth` and `bWDistribution` will be interpreted as absolute values, regardless of the value specified in `bWDistributionDataSpecifier`.

Assigning physical cores to applications The first optimization problem that assign service levels to applications takes the total bandwidth of the system as limit of utilization. But consider the case of three applications registering on a system with two cores, each application with a resource demand of 50%. The total bandwidth for Actors applications is 170% (90 + 80) and the sum of resource demands for the applications is 150% (50 + 50 + 50) so the optimization problem assigning service levels will say that the problem has a feasible solution. However there is no way to place these applications on the physical cores, one application each on the two cores will give 30% left on one core and 40% left on the other core so there is not sufficient space to run the third application on one of these cores. In the current state the RM will crash with a segmentation fault at this point.

Handling of happiness value The handling of clients happiness values is not implemented. The happiness signal can be sent by an application, and the RM will receive it, but no changes action will be taken.

3.4 Robot software, HAM

Overview

The software used to control the robot is called HAM and was developed in a project at Lund University, aiming to study the interaction between humans and robot through body language [Topp, 2008]. The name HAM is a short for "Human Augmented Mapping", and it enables the robot to go to specific places or follow users around while mapping its surroundings using two laser scanners. This system contains the features needed to study the effects and requirements for adaptive resource management. The use cases can not be determined in advance and it provides a dynamic execution where off-line analysis and scheduling could be insufficient.

Internal design

The HAM application consists of a number of threads performing different tasks. it also creates a Qt-based Graphical User Interface where the user can interact with the robot. The different threads are described below. The threads `Follower` and `Wanderer` are the threads that define the behaviour of the robot and only one of these can be active at one time. Figure 3.5 show how the threads are connected, and how the data is transmitted through the application.

DataHandler The DataHandler thread reads values from the laser scanners and saves the raw data in arrays containing information about the distance to objects in 360 degrees around the robot. It also calculates the robot position from the motion of the wheels. The robots battery power is also measured by this thread, if the robot is about to run out of battery this could cause it to misbehave, and if this is the case the motors are turned of and the robot is left standing still.

Tracker This thread keeps track of the objects in the surroundings of the robot, it is the thread consuming by far the greatest part of cpu capacity, but it is also essential for the performance of the system. It takes the raw data arrays created by the DataHandler and finds different objects that could be of interest, such as a person to follow or obstacles to avoid. The laser scanners are located close to the ground and persons are identified as two objects of the same size with about the same distance from the robot and the right distance between them. This is considered being human legs and once the Tracker has found a pair of legs it will keep track of the person.

Slammer The acronym SLAM stands for "Simultaneous Localization And Mapping". This thread creates a map and displays it in the GUI. It also keeps track of how the robot is positioned in the room.

MotionController Controls the motion of the robot. Other threads can access methods such as `setGoal()` and the `MotionController` will make sure that there is sufficient battery power and move the robot to the goal set by this other thread.

Follower This thread collects information about the map created by Slammer and about person objects created by the Tracker and sends motion commands to the MotionController to make the robot follow users around without hitting any obstacles.

Wanderer This thread collects information about the map created by the Slammer and lets the robot wander randomly on its own.

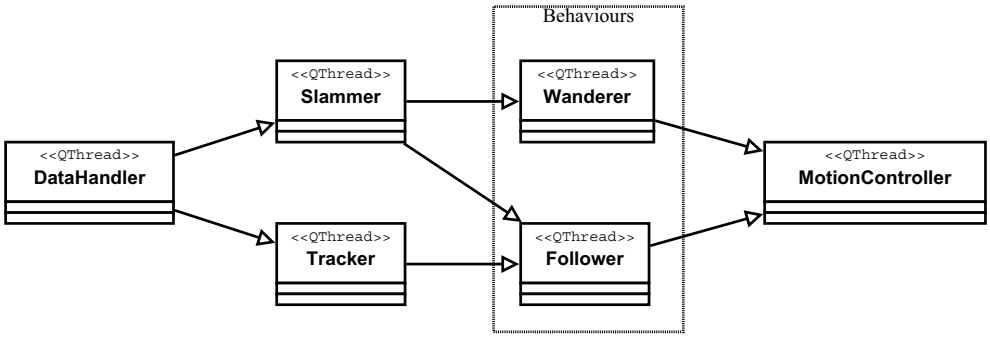


Figure 3.5 Diagram showing the flow of data in the HAM application

Modes of operation

There are different ways for the robot to operate, the two different behaviors `Follower` and `Wanderer`, but also a mode where only the `DataHandler`, `Tracker` and `Slammer` threads are active to create a map. These threads are active at all times, but the threads controlling the robots behaviour are active one at a time together with the three map creating threads and `Motioncontroller`.

3.5 Hardware monitoring and thermal control

To handle hardware resources such as battery power some sort of hardware monitoring system is needed.

Lm-sensors

Lm-sensors is a free hardware monitoring package for linux (www.lm-sensors.org), it provides a configuration utility for finding supported hardware sensors and device drivers. The Pioneer robot has a National Semiconductor LM83 chip [National Semiconductor Corporation, 1999] and the kernel driver lm83 [Delvare, 2010] is used to read the temperature. Many motherboards have voltage, temperature and fan rotation speed sensors, but the LM83 chip is a temperature-only chip. The temperature reading is updated every other second, reading the temperature more often will do no harm, but will return old values. The sensor accuracy is 3°C and the resolution is 1°C. There is no hysteresis mechanism present on this chip.

CPU thermal model

To identify the dynamics of the CPU temperature a step response test has been performed, the result is shown in Figure 3.6. According to this the dynamics between utilization U and core temperature T can be roughly modelled as a first order system with time delay according to Equation 3.6.

$$\frac{T(s)}{U(s)} = \frac{K_p}{Ts+1}e^{-\tau s} \tag{3.6}$$

The system gain K_p, time constant T and dead time τ that can be determined from step response analysis.

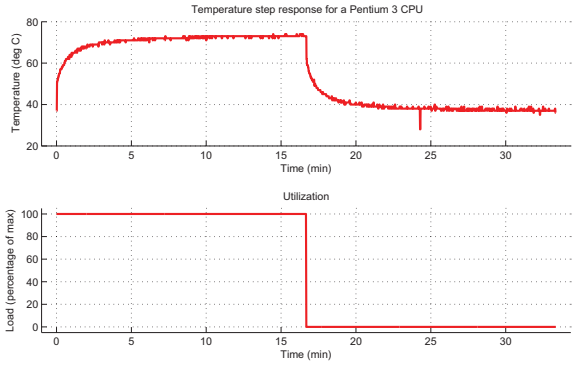


Figure 3.6 Result of a step response experiment from utilization to temperature performed on the processor of the robot.

4. Implementation

4.1 Adapting the robot platform to run the RM

In order for the robot to run the RM some changes have been made to the system to meet the dependencies.

SCHED_EDF kernel

The SCHED_EDF kernel described in Section 3.2 has been compiled and installed on the robot. The configuration file for the previous working kernel was used with the modifications from [Scordino and Trimarchi, 2009].
After leaving the graphical configuration editor the configuration entries `CONFIG_CGROUP_FREEZER` and `CONFIG_CGROUP_NS` had to be manually set to `n`.

Dependencies

To be able to build and run the RM the following dependencies have to be met:

- A C++ compiler
- CMake >= 2.6.0
- D-Bus libraries
- the Ruby scripting language >= 1.8.0
- Gnu Linear Programming Kit, glpk

These libraries are all present in the standard Debian repositories. However, the versions required is not in the repositories for Debian 4.0 which was installed on the robot. To avoid having local versions of e.g. CMake that would perhaps not be compatible with the rest of the system, a full system upgrade to Debian 5.0 has been done. Once this was done the Player libraries had to be recompiled to run on the new system.

4.2 Adapting the RM to the robot platform

To get the RM to run on the robot-platform some fixes had to be made. Since the Actors-RM is not yet complete, some parts are buggy and some parts are designed and tested in TrueTime (www.control.lth.se/truetime) but not implemented. The focus in this project when adapting the RM to the robot platform has been to fix the bugs and implement the missing parts in such a way that the RM can run on this specific platform, the version of the RM that is currently on the robot will e.g. not work well on a system with more than one CPU.

Bugs

The RM uses two main data structures to keep track of the registered clients, a map `m_clients` containing a representation of the clients with their D-Bus connection-ID as the key, and a struct `m_logicData` containing vectors with data to solve the

optimization problems. Assuming that each client creates a D-Bus-connection and resumes directly with the registration process the `m_clients`-map will be organized from old to new since new connections will have a higher number. But new clients are put at position 0 in the `m_logicData`-struct creating an order from new to old giving new clients the data belonging to the previous registered client and so on. Also the RM could not handle multiple registrations at the same time, which is desirable due to the design of the robots control applications. Another property that was not implemented was the ability for a client to unregister and register again without closing the DBus connection. All these issues have been fixed by letting the `m_logicData` struct be ordered as the `m_clients`. The RM also had some memory handling bugs that caused the RM to segfault when erasing entries in a map, and there was also an indexing error that made it impossible for an application to have more service levels than threadgroups. These bugs have been fixed.

Solving the bandwidth distribution problem

The RM solves a bandwidth distribution problem to place the virtual processors on the physical cores. This is done by spreading the virtual processors evenly on the cores, and right now it is not possible for a client to have more virtual processors than there are physical cores on the computer. The robot has a single core setting which limits the clients to have only one threadgroup each. However, the bandwidth distribution problem in this case is easy to solve, all virtual processors should be placed on the only core available. On top of this, the method that updates the available CPU bandwidth was written assuming that a client could only have one virtual processor on each core, so this method had to be fixed also.

4.3 Load-simulation client

To test and debug the RM a simple client application has been implemented. It does not perform any real tasks, but just consumes CPU time according to different service levels that are specified at startup. The client application can also be used to simulate load on the running system when testing the RM on the robot. The application is started with

```
./main <c> <n> <low> <high> <output file>
```

This will register a client with the category *c* and *n* service levels with resource demands spread equidistant between *low* and *high*. Output data such as in which service level the client is running is saved to the *output file*.
This application was used with some modifications when making the changes described in Section 4.2. A script was written to start a number of these applications simultaneously, with different number of threadgroups and service levels, letting them run for a while and then unregister and register again, with, or without closing the DBus connection. In this way the RM could be thoroughly tested and the bugs could be forced to appear and could then be identified and fixed.

4.4 Adapting HAM to the RM

Comunication with the RM

RmInterface Since all applications that intend to work with the RM should go through the same registration process all calls to the RM have been collected in a C++ class that handles the generation of D-Bus messages from ordinary C++ method calls. This class is called RmInterface and is built on the Qt D-Bus bindings. The reason for using Qt is that HAM already uses Qt to handle the GUI events, but also because of the complex structure of the service levels list. D-Bus low level bindings only recognise basic data types such as integers and chars, so to avoid having to write XML messages representing arrays, maps etc. by hand which would be very time consuming, high level bindings must be used. The methods in the RmInterface are the same as presented in the RM D-Bus API in Section 3.3, and the client applications can use them as ordinary internal methods.

changeServiceLevel To make the different pars of HAM listen to instructions from the RM a method `changeServiceLevel` has been implemented. This method is connected to the D-Bus signal `changeServiceLevel` from the RM. When the signal is sent, all methods that are connected to this signal will be invoked, i.e. all client applications will receive the signal. The RM sends the new service level together with the D-Bus connection id as an identifier specifying what client that is to change its service level, so each client must compare this identifier with their own id before changing service level. The method is presented below.
The internal variable `SL_index` is then used by the application to change its behaviour.

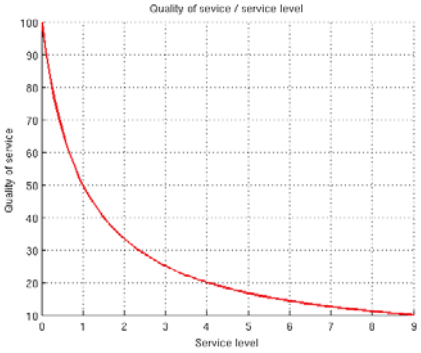
```
void changeServiceLevel(QString receivedID, int new_SL)
{
  if (receivedID.compare(myConnectionID) == 0)
  {
    SL_index = new_SL;
  }
}
```

Assigning service levels

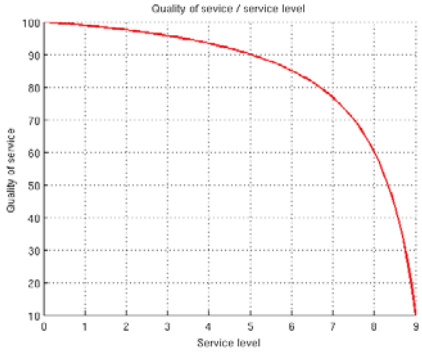
That the applications are Actors aware is a demand for being able to work with the RM, but to make fully use of the RM the application should also be adaptive, i.e. have more than one service level. In the HAM application all threads are periodic, and the different service levels are set to have different period, and thereby different resource demands. When designing a service level two parameters are of particular importance, the resource demand at the specific level, and the quality of service delivered by the application at this level.

Assigning quality of service When setting up the service levels for the applications it is useful to know how the RM will handle the service level assignment. The RM will maximize the overall quality of service delivered by the system, weighted by the importance value and bounded by the availiable bandwidth (Equations 3.2 - 3.3). Consider setting the quality of service according to Figure 4.1(a) - 4.1(c), showing how quality of service could be assigned to the service levels of an application in

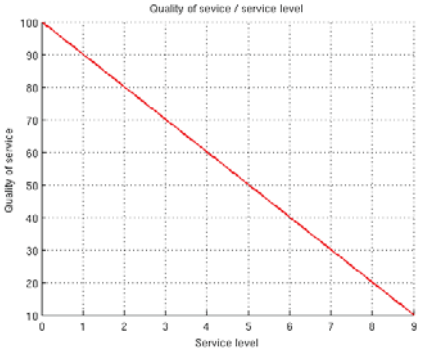
three different ways. Here the service levels are labeled 0 to 9 where 0 is the service level corresponding to highest performance and 9 to the lowest.
In Figure 4.1(a) the quality of service drops rapidly when going from level 0 to 1, lowering the overall quality of service by a large amount. But once an application is at level 1 the "cost" of going to level 2 will be less than changing the service level of another application with quality of service assigned in the same way. This will result in that if the available resources should decrease, or a new application with higher importance should register, an application designed in this way will go to level 9 before any other application goes to level 1. Figure 4.1(b) shows the opposite situation. The loss of quality of service increases as an application goes to a higher (lower performance) level, resulting in that all applications designed in this way will go to level 1 before any application goes to level 2.



(a) The loss of quality of service decreases as the application goes to higher levels.



(b) The loss of quality of service increases as the application goes to higher levels.



(c) Linear relation between service level and quality of service.

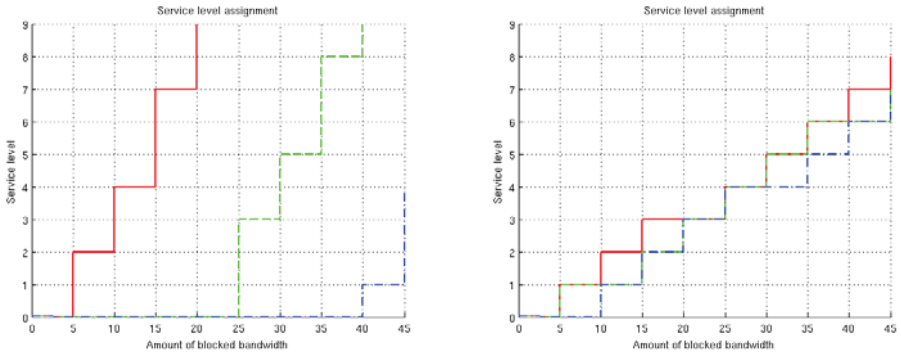
Figure 4.1 Three different ways to assign quality of service to service levels.

In the case shown in Figure 4.1(c) the cost is the same at all changes of service level, so in this case the application importance value will be used to determine the assignment of service levels. Figure 4.2 - 4.3 show how the RM will assign service levels to three applications depending on how much of the bandwidth that is blocked. All three applications have 10 service levels each, with resource demands according to:

Sevice level	0	1	2	3	4	5	6	7	8	9
Resource demand (%)	26	24	22	20	18	16	14	12	10	8

The quality of service is assigned to the service levels in different ways according to the curves in Figure 4.1(a) - 4.1(b). In this example it is assumed that there is one

core and that the available bandwidth for Actors applications is 80% as stated in Section 3.3. The scale of the x-axis shows how much of the 80% that for some reason is being blocked, and the figures show how the service level assignment would be done in this case.
In Figure 4.2(a) all three applications have their quality of service set according to Figure 4.1(a) and we can see that the application represented by the red line will drop to level 9 before the green application starts to follow. In Figure 4.2(b) all applications are assigned quality of service according to Figure 4.1(b) which instead causes the applications to change service levels together.



(a) Quality of service set according to Figure 4.1(a).

(b) Quality of service set according to Figure 4.1(b).

Figure 4.2 Three identical applications change their service levels when the available bandwidth is limited.

Figure 4.3 shows the case where the red application has its quality of service according to Figure 4.1(a) and the green and blue applications according to Figure 4.1(b). Here the cost of changing service level for the red application is very high when going from level 0 to 1 so the RM will keep it at level 0 as long as possible, but once it is forced to change level due to insufficient bandwidth it will go directly to level 9 and the green and blue application will go back to levels with higher performance.

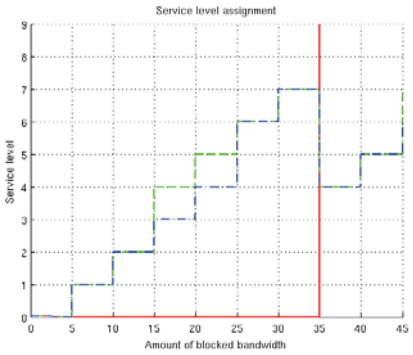


Figure 4.3 The application represented by the red line has its quality of service set according to Figure 4.1(a), and the green and blue applications has their quality of service according to Figure 4.1(b).

The results shown here is a special case since all three applications are identical in terms of their resource demands, the results are not directly applicable to the general case of applications, but the knowledge of how the RM will handle the quality of service is still useful when designing client applications. Different assignments of

quality of service will provide different results, even if the optimization problem will be more complex than shown here.
For the HAM application different ways to assign quality of service have been tested, and the results from this is presented in Figures 5.2 - 5.3.

Estimating the resource demands Since the RM is not yet able to handle relative resource demands or make estimates at runtime (Section 3.3) some estimate of the absolute values are required to be done off-line. This has been done by measuring the average computation time for the different threads and assigning the quote between the computation time and the period as the resource demand (Equation 4.1).

$$B_s = 100 * \frac{d}{p_s} \tag{4.1}$$

$$Where:$$

$$\begin{aligned} B_s &= required\ bandwidth\ (resource\ demand)\ at\ service\ level\ s \\ d &= duration\ (computation\ time) \\ p_s &= period\ of\ the\ application\ at\ service\ level\ s \end{aligned}$$

The period is the design parameter that can be chosen to achieve the desired behaviour. All threads have been assigned 5 quality levels with period according to Table 4.1.

Service level	0	1	2	3	4
Period (ms)	10	20	30	40	50

Table 4.1 All threads are assigned the same period.

As an estimate of the average computation time, measurements taken directly from the different threads are used. For the Tracker this value varies a lot, but for the other threads it seems that the duration has an upper limit. The values taken as the average duration are presented in Table 4.2.

Thread	Duration (ms)
DataHandler	1
Tracker	6
Slammer	0,15
MotionController	1
Follower	0,2
Wanderer	0,05

Table 4.2 The average computation time for the different threads

This results in an assignment of resource demands as presented in Table 4.3. The resource demands of some threads are very low, and the RM can only handle integer values and has a lower limit of 2%. If registering the Wanderer thread as an application, the resource demands would be 0 for service level 1 - 4, and this causes the EDF kernel to lock down, so the values in Table 4.3 have to be truncated to the nearest integer and if they are below 2%, they are set to 2%.

Thread \ Service level	0	1	2	3	4
DataHandler	10%	5%	3,33%	2,5%	2%
Tracker	60%	30%	20%	15%	12%
Slammer	1,5%	0,75%	0,5%	0,375%	0,3%
MotionController	10%	5%	3,33%	2,5%	2%
Follower	2%	1%	0,66%	0,5%	0,4%
Wanderer	0,5%	0,25%	0,166%	0,125%	0,1%

Table 4.3 The resource demands for the different threads. Calculated from Table 4.1 and 4.2 according to Equation 4.1.

Grouping threads in threadgroups

The HAM application contains 6 different threads (Section 3.4), which are listed here again for convenience:

- DataHandler
- Tracker
- Slammer
- MotionController
- Follower
- Wanderer

These threads can be grouped together in thread groups in different ways, as described in Section 3.3. One way would be to register HAM as one application with many threads, but another possibility is to register each thread as an application of its own. This would provide the possibility for each thread to have its own service levels and to send happiness values even if this handling is not yet implemented in the RM (Section 3.3). Another way to see it is that the application consists of two different parts, one part that creates a map with the `DataHandler`, `Tracker` and `Slammer` threads, and another part that defines the robots behaviour with one of `Follower` or `Wanderer` together with `MotionController`. These two parts could be registered as two individual clients with their threads placed into thread groups. Both ways have been tested, and the results are presented in Section 5.

4.5 Processor thermal control

As an attempt to make the RM handle the battery power as a resource a new thread has been introduced. This thread is a PI controller that controls the temperature of the CPU with the utilization as control signal. In this way one can set a limit to the CPU temperature and turn of the active cooling system, thereby limiting the power consumption [Lindberg, 2010]. Unfortunately the robot is not equipped with any way to control the fan, but the experiment still stands as an idea to how the RM could be extended. As described in Section 3.5 the temperature reading is updated by the sensor every other second. The periodic threads that will run have a shorter period which result in an aliasing effect as can be seen in Figure 4.5. For the case with shorter period the aliasing effect is still present, but is limited by the fact that the load is more evenly spread. To handle the measurement noise the signal is filtered through a FIR filter with a rectangular window of one minute before calculating the control error. Using the results from the step response test in section 3.5 and adding the filter dynamics, the system parameters in Equation 3.6 are determined to be $K_p = 0.37$, $T = 32.54\ s$ and $\tau = 31.1\ s$. Using this, a controller that runs every two seconds has been designed using an internal model control (IMC) approach [Rivera *et al.*, 1986]. The controller parameters are set to $K = 4.1792$ and $T_i = 48.09$. If the limit of utilization is changed the RM will do a new service level assignment with the value from the PI controller as the upper limit L in Equation 3.3.
The actual utilization on the system is then determined by the service level assignment and bandwidth controller of the RM. Figure 4.4 show how the service level assignment relates to the limit set by the controller in the case of running HAM with service levels and resource demands set according to Table 4.3. The measured value of the utilization from the RM is used when updating the integral state in the PI controller to prevent integrator windup.
Figure 4.6 show a diagram of the controller setup.

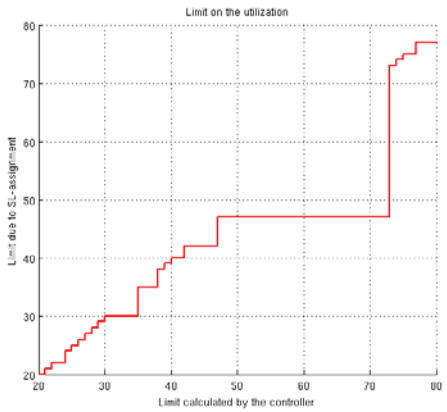
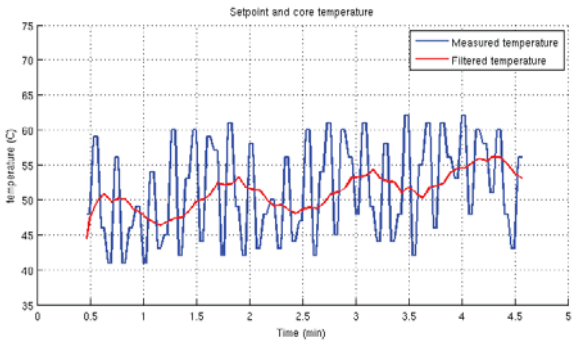
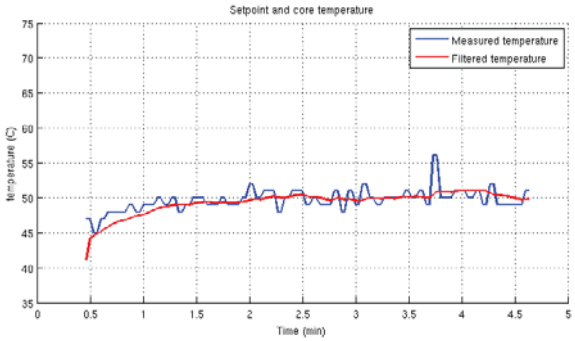


Figure 4.4 The limit set by the controller results in a lower utilization on the system



(a) Time granularity 100 ms.



(b) Time granularity 10 ms.

Figure 4.5 Measured and filtered temperature when running an application with constant load but different time granularity.

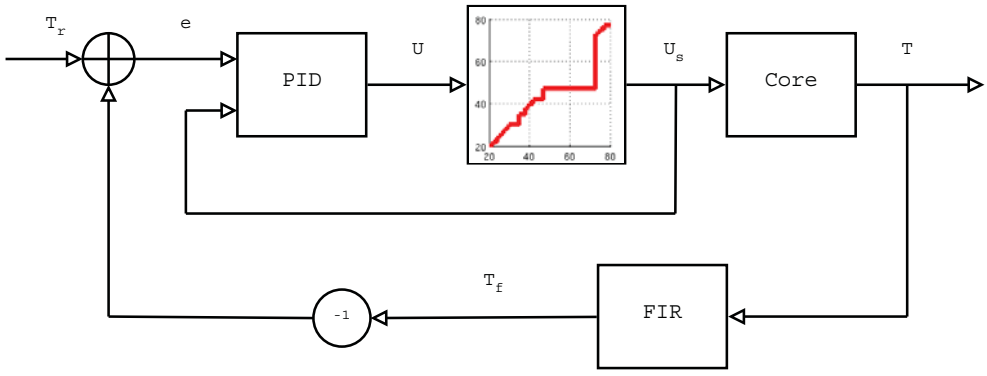


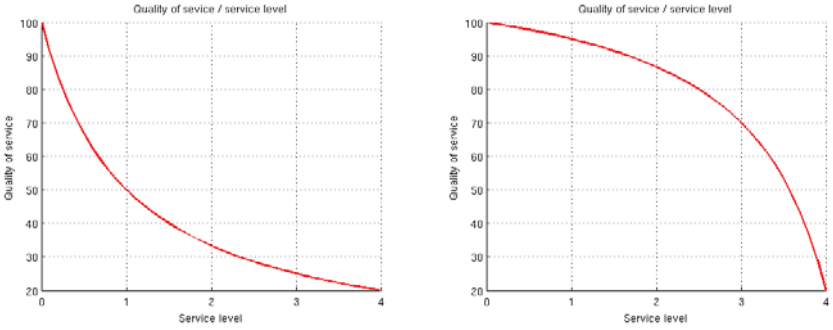
Figure 4.6 Diagram of the controller setup, T_r = temperature setpoint, e = control error, U = controller output, U_s = saturated output, T = measured temperature, T_f = filtered temperature.

5. Experimental setup and results

The tests performed here aim to show the results of different ways of assigning quality of service and different ways of putting threads in threadgroups. Both to evaluate the behaviour of HAM in these cases and to see that the RM acts the way it is supposed to. All tests in Section 5.1 - 5.2 are performed in the same way, HAM registers in follower mode with threadgroups and service levels assigned in different ways. The client application described in Section 4.3 then registers with the RM with only one service level, which forces the RM to give it the bandwidth it demands. The resource demand of this application goes from 5 to 60% in steps of 5%, with 10 seconds interval. This will cause HAM to change service levels in different ways depending on the way the service levels are assigned.

5.1 Assigning quality of service

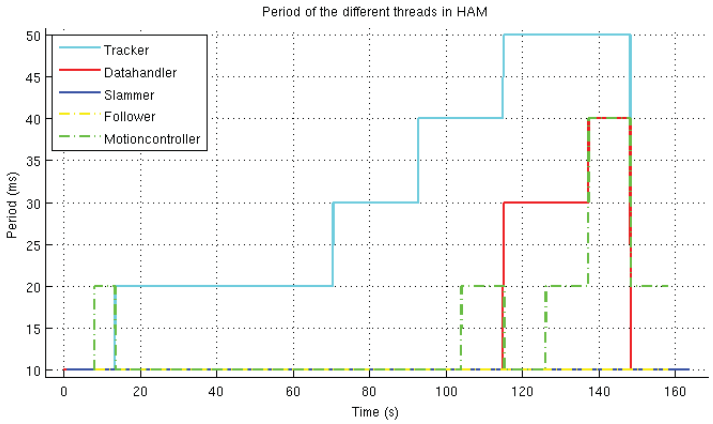
To see how the RM handles the service level assignment among many applications all threads have been registered as individual clients in this test. The quality of service are assigned according to Figure 5.1(a) - 5.1(b), and the theoretical result presented in Figure 4.2(a) - 4.2(b) gives an idea of what to expect.
In Figure 5.2 the quality of service has been assigned to the threads according to Figure 5.1(a) and in Figure 5.3 it is set according to Figure 5.1(b). The figures showing the period of the threads are produced using data from HAM, so this shows that HAM receives and handles the `changeServiceLevel` signal correctly. When comparing Figure 5.2 and Figure 5.3 we can see that the RM assign service levels more equal among the threads in Figure 5.3, but all differences in these figures can't be explained by the difference in quality of service. At first when only the `Tracker`, `DataHandler` and `Slammer` are active there is enough resources to run all these threads at service level 0, but when the `Follower` and `Motioncontroller` are activated the RM can choose to change the service level of either `Tracker`, `DataHandler` or `Motioincontroller`. Either way this is done will result in the same overall quality of service. In this case the decision of which client that should change service level is simply based on which client that happens to come up first in the implementation of the optimization problem solver, and this is the reason why the `Motioincontroller` changes level in Figure 5.2(a) but not in Figure 5.3(a).
It can also bee seen that the `Slammer` and `Follower` threads does not change service level, and this is due to the fact that they have the same resource demand for all their service levels, so there is nothing to gain from changing level.



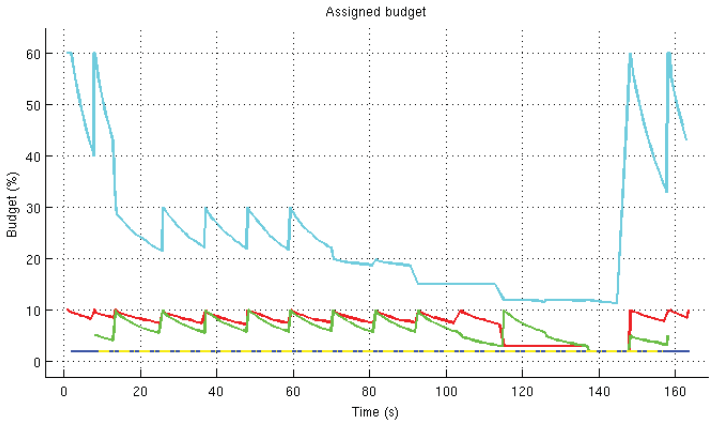
(a) The loss of quality of service decreases as the application goes to higher levels.

(b) The loss of quality of service increases as the application goes to higher levels.

Figure 5.1 Two ways to assign quality of service to the 5 different service levels.

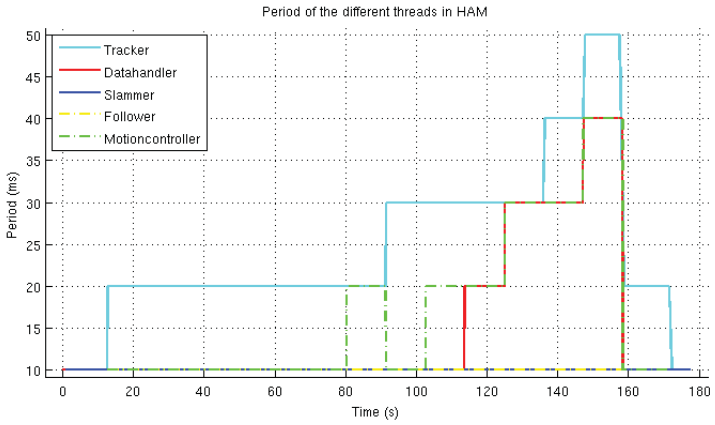


(a) Period of the different threads in HAM.



(b) Bandwidth assigned to the different threads by the RM.

Figure 5.2 Period of the threads, and budget assigned by the RM when quality of service is set according to Figure 5.1(a).



(a) Period of the different threads in HAM.

(b) Bandwidth assigned to the different threads by the RM.

Figure 5.3 Period of the threads, and budget assigned by the RM when quality of service is set according to Figure 5.1(b).

5.2 Threadgroups

In this test the quality of service is set according to Figure 5.1(b), and the threads have been put in threadgroups as shown in Figure 5.4. Here the different clients that has a connection to the RM is `Mapper` and `Behaviour`, and the change of period is distributed internally to the different threads of these clients. In this way the period of `Slammer` and `Follower` is changed together with `DataHandler` and `MotionController` respectively. The RM controls the bandwidth of the three different virtual processors, and Figure 5.5(b) and 5.6(b) shows the assigned budget. In both Figure 5.5 and Figure 5.6 the robot is set in follower mode, and the threadgroups and service level assignment are the same, but in Figure 5.5 the robot has been standing still during the whole test. This was achieved simply by not walking in front of the robot, so that the tracker could not find anything to track. In Figure 5.6 the robot has been led around a room with a lot of chairs, tables and other things that the tracker will find interesting. The result is that the resource demands of the `Tracker` for one service level varies between 10 and at least 30% in the different cases as can be seen when comparing the budget assigned to the `Tracker` in Figure 5.5(b) and 5.6(b). This problem could be solved be splitting the `Tracker` thread into smaller parts that e.g. tracks one object each. These parts could then register when needed and the estimate of the resource demand would be more accurate.

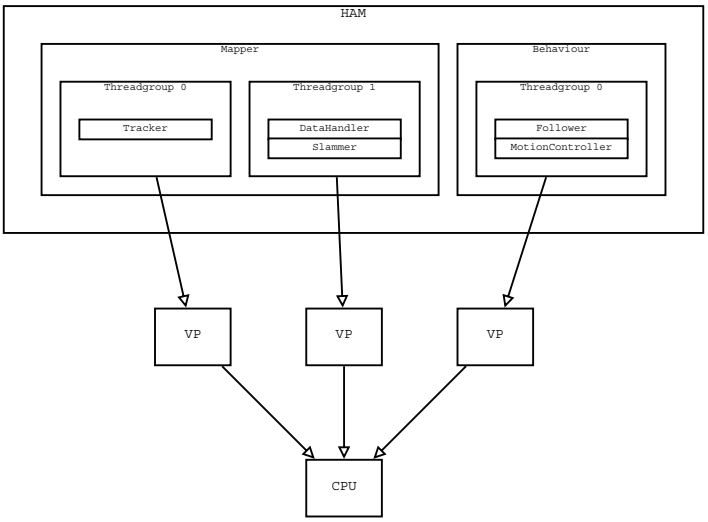
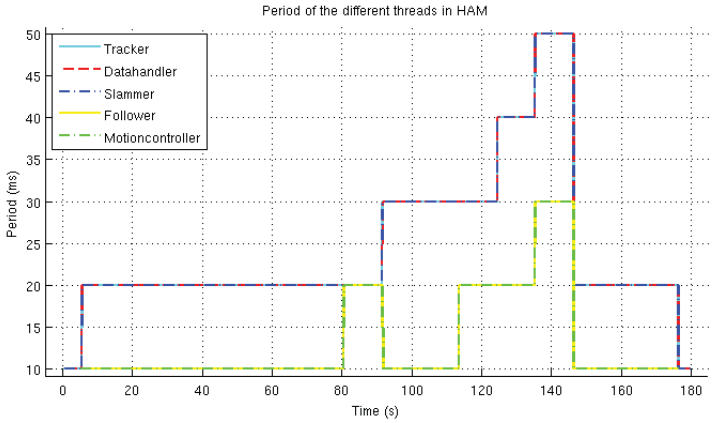
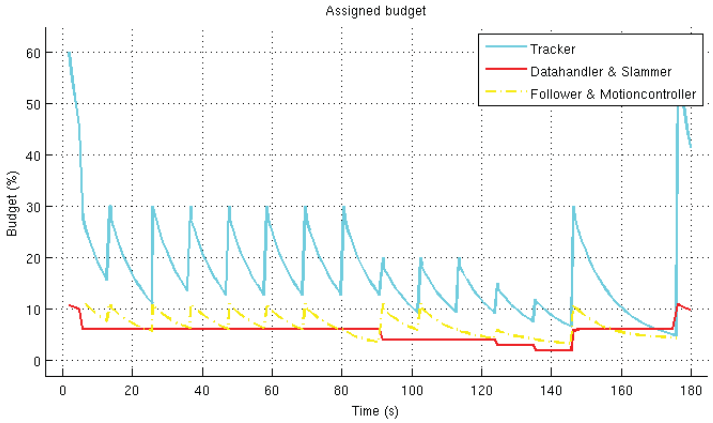


Figure 5.4 The threads of HAM are put in different threadgroups

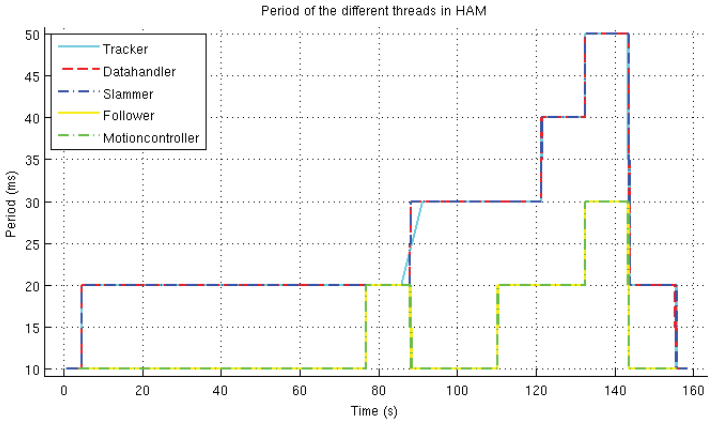


(a) Period of the different threads in HAM.

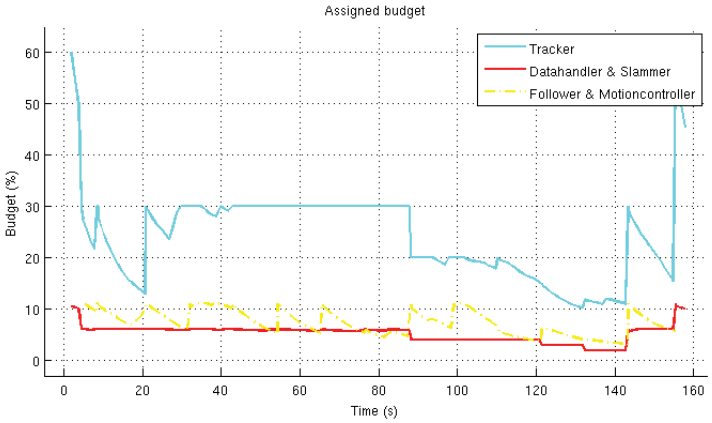


(b) Bandwidth assigned to the different threadgroups by the RM.

Figure 5.5 Period of the threads, and budget assigned by the RM when the robot is standing still.



(a) Period of the different threads in HAM.



(b) Bandwidth assigned to the different threadgroups by the RM.

Figure 5.6 Period of the threads, and budget assigned by the RM when the robot is moving around.

5.3 Temperature control

To test the performance of the controller two experiments have been performed.
In the first experiment the temperature setpoint was set to 55°C for 10 minutes and then to 45°C for 10 minutes. A modified version of the client application described in Section 4.3 with service levels according to Table 5.1 and only one thread was running on the system. The application has random execution time and after about 200 seconds it decreases its use of bandwidth about 10%. The result is presented in Figure 5.7. During the first 10 minutes the temperature limit of 55°C does not affect the execution of the application which is running in service level 0. The slow response from utilization to temperature that can be seen after 200 seconds is due to the FIR filter, and it is clear that the control signal (Utilization limit) does not wind up even if the temperature does not reach the setpoint. After 10 minutes when the temperature setpoint changes to 45°C the temperature limit is violated and the utilization limit is pushed down below the current measured utilization. This results in a service level change for the running application that goes to service level 2. When the temperature has stabilized the application can go back to service level 1 and stay at this level.

Application name	SL	QoS [%]	Resource demand [%]	Period [ms]
A1	0	100	60	20
	1	90	30	20
	2	75	20	20

Table 5.1 Service level table for application A1

For the second experiment, two applications were running on the system. The service levels were set according to Table 5.2 and application A1 were given higher importance. Also here the applications consists of only one thread each. The setpoint temperature was kept at 50°C through out the experiment and the result is presented in Figure 5.8. During the first 300 seconds application A1 are running alone, and the temperature limit is not violated. When application A2 starts to execute the temperature builds up, and about 700 seconds in to the experiment the temperature limit is violated which forces application A2 to change service level. Since application A1 has higher importance it is left to run in service level 0 through out the experiment.

Application name	SL	QoS [%]	Resource demand [%]	Period [ms]
A1	0	100	40	20
	1	90	30	20
	2	75	20	20
A2	0	100	20	20
	1	85	10	40
	2	35	5	80

Table 5.2 Service level table for applications A1 and A2

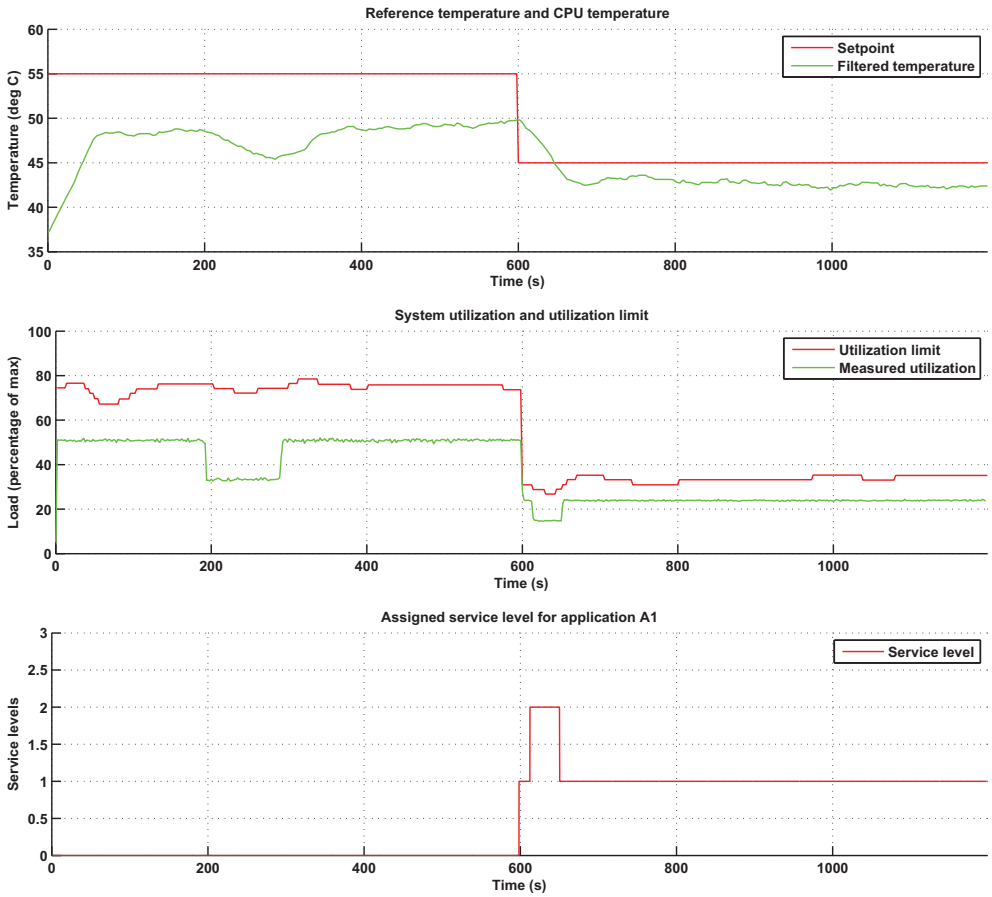


Figure 5.7 Performance results under normal conditions. One application running on the system subject to changes in system temperature limit.

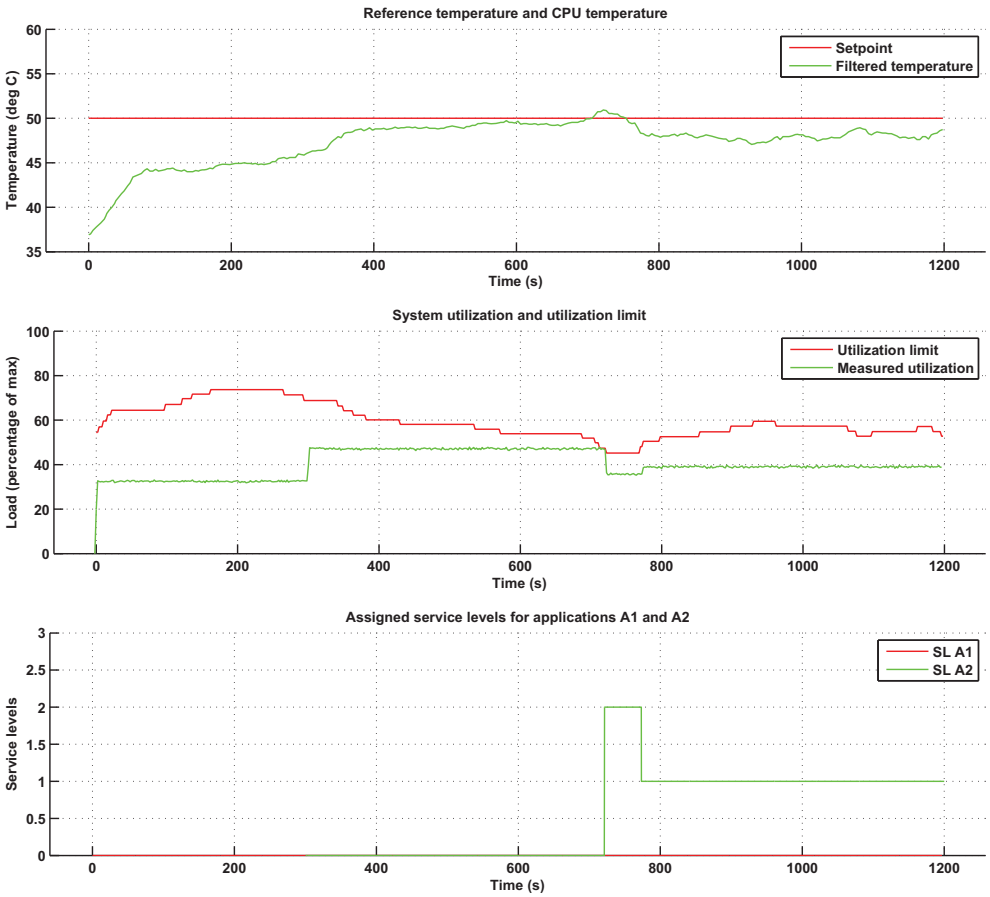


Figure 5.8 Performance results under overloaded conditions. Two applications running on the system subject to system temperature constraints.

6. Conclusions and future work

The quality levels that have been set to HAM does not provide a change of performance that can be measured, one could even question if a period of 10 or 50 ms really makes any difference to the performance. The robot is not very fast, and the surroundings does not change that fast that the robot is likely to miss obstacles when having a period of 50 ms. The quality levels have been assigned in this way as an effort to make the robot consume more CPU time than available to be able to see if the RM handles this the way it is expected to. In this case where the CPU time is not limited in a way that effects the robot (when running HAM prior to the changes made in this project it consumed below 10% of the CPU) a far more interesting resource to consider would be the battery power. The robot can only run for about 40 minutes without charging the batteries and this is a limitation. More appropriate quality levels could involve turning off the back laser when the robot is moving forward in known areas, and other ways to change the use of hardware to achieve high performance with low power consumption.
This raises the question of how to define high performance and quality of service. What "good quality of service" is, is not defined by the RM, but by the individual client applications. As shown in Section 5.1, the way of assigning quality of service has a great impact on the service level assignment, and it would perhaps be necessary to specify some guidelines on how the quality of service should be assigned.
The quality of service could also be affected by external factors. If good quality of service means keeping the performance of the positioning within some error bounds while consuming as little battery power as possible, the desired way to operate will change e.g. if the robot travels in a large room with few obstacles or if it travels in a tight space. In a large room the data would not need to be updated as often and a higher motor speed could be accepted. One way to achieve this change of way to operate could be by using the happiness value, but another way would be to re-register the service levels, with different values of quality of service, letting the clients estimate their performance at runtime.
Once the RM is fully implemented it will have greater potential than what has been demonstrated in this project. Also if the control system of the robot was designed with respect to the RM from the start, some of its features that is normally handled internally by the control application could instead be handled by the RM, thereby giving more information to use when optimizing the performance of the entire system. Producer-consumer situations could be considered as resource management problems and the happiness value could be used to inform the RM of the clients performance. One way to handle the happiness values is simply:

- Not happy $\rightarrow$ change to a lower level
- Too happy $\rightarrow$ change to a higher level

In other words, if one part of the system does not get new data at the desired rate this would cause a low happiness value and therefore shifting it to a lower level, leaving more resources to other applications.

7. Bibliography

Årzén, K.-E. (2008): “Real-time control systems.”

Årzén, K.-E., V. R. Segovia, A. Cervin, A. Neundorf, G. Fohler, and E. Bini (2009): “Db3.”

conscious robots (2007): http://www.conscious-robots.com/en/reviews/robots/mobilerobots-pioneer-3-p3-dx-8.html.

Delvare, J. (2010): “Kernel driver lm83.” http://www.mjmwired.net/kernel/Documentation/hwmon/lm83.

Lindberg, M. (2010): “Adaptive resource management for uncertain execution platforms.”

MobileRobots Inc (2006): *Pioneer 3 Operations Manual*. Amherst, NH, US.

National Semiconductor Corporation (1999): *LM83 Triple-Diod Input and Logical Digital Temperature Sensor with Two-Wire Interface, DS101058*.

Rivera, D. E., M. Morari, and S. Skogestad (1986): “Internal model control 4: Pid controller design.” In *Industrial and Engineering Chemestry Research*, pp. 252–265.

Scordino, C. and M. Trimarchi (2009): “D4e.”

Topp, E. A. (2008): “Human-robot interaction and mapping with a service robot: Human augmented mapping.”

Versalogic Corporation (2007): *Model VSBC-8 Reference manual*. Eugene, OR, US.