Resource management and prioritization in an embedded Linux system

Fredrik Johnsson
Olle Svensson

LUNDS UNIVERSITET

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--9999--SE
ISSN 0280–5316

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

© 2014 by Fredrik Johnsson
Olle Svensson. All rights reserved.
Printed in Sweden by Media-Tryck.
Lund 2014

Abstract

The problem of limited resources on an Axis camera is handled by a two part solution where a resource manager (RM) distributes the available resources and where services adapt their service level (SL) in order to let the services finish their jobs on time. This is done in a game theoretic approach where the services are players, varying their SL in order to get a good match between given resources and their SL. This SL-adaptation scheme is then implemented on the streaming service on the camera and on some test services performing simple math operations. The resource manager is incorporated into systemd which uses CGroups [*title*] to implement the distribution of resources.

Acknowledgements

We want to thank our

Contents

1

Introduction

1.1 Axis

Axis is a company founded and based in Lund and it is the global market leader in network video solutions.

1.2 Problem formulation

It is becoming more common to have multiple resource intensive services running on Axis cameras. At the same time, the demands are increasing on reliable and consistent video frame rate and quality. This creates a problem where different services are competing for resources like CPU and RAM. This can result in poor performance of the camera under scenarios where the it is under heavier load than usual, one example being when answering calls from the network. The solution that we will evaluate is an application of the work carried out at the University of Lund [Martina Maggio, 2013], resulting in a resource manager called Game Theoretic Resource Manager (GTRM) and a library that lets services implement the SL adaptation and communication with the RM. The goal is to see if it is possible to run this resource manager on an Axis camera running Linux with the streaming application competing with some test applications for resources. In this test the cameras image quality will be its SL and the inverse of its frame rate the measurement for the time a job takes to finish, where a jobs deadline will be set to 1/deadline.

1.3 Related Work

Regarding the resource management, similar approaches have been developed in the past, but the resource manager and SL adaptation being evaluated here is unique in that it lets the applications themselves determine their SL, as opposed to being set by a central unit. An example of a RM that determines both resources and SL for all applications is one developed by the ACTORS project [E. Bini, 2011].

1.4 Outline of the report

2

Background

2.1 Game Theoretic Resource Manager

The resource management is based on the performance of the applications that are running. By calculating the difference between the applications deadline and its execution time one gets the performance, also called matching function, of the application. Ideally the matching function should be zero. When zero, the application has just enough resources to meet its deadline running with some SL. When positive, too much resources are assigned to the application, indicating that the job is done before deadline, and when negative, too little resources are assigned indicating that the application has missed its deadline.

The resource managing consists of two parts:

Service Level adaptation

The Service Level (SL) defines the quality of service of the application. In the case of the streaming application this is the quality of the image, but it can mean different things to different applications. The main property of the SL is that an increase in SL gives an increase in the required resources for the application. The idea is to change the SL to optimize the utilization of the amount of resources available. A high/low performance will be responded to by an increase/decrease in SL. This will make sure that the application is always presenting valid result in time but with quality as a trade-off. This adaptation is done by the application itself.

Resource Manager

The resource manager(RM) will also measure the performance of the applications. It will try to distribute the resource in the best possible way to the applications. The resources are modelled as "virtual platforms", and is basically a percentage of the total available resources. For example the amount of time an application is allowed to use the CPU in proportion to the other applications. Here "resources" could refer to something other than CPU, such as memory or network-resources, depending on different aspects of the system. The main property of the performance is that if an application is given more/less resources it should be able to execute at a higher/lower SL or have a shorter/longer execution time.

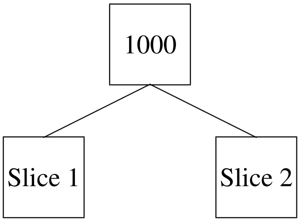
The theory of decoupling the resource manager from the service level adaptation was developed at the Department of Automation at Lunds University. The resulting resource manager is referred to as Game Theoretic Resource Manager, GTRM.

The idea behind this way of using a resource manager (RM) and service level (SL) adaptation, separating it from the norm, is to let each application adapt its own SL continuously independent from the RM loop. One of the main benefits of this decoupling is that the task of adjusting SL is given to the applications that have the most knowledge of how to tune their parameters in order to adjust their needed resources, and at what cost in quality.

2.2 systemd and cgroups

Systemd [sysd] is a system management daemon for Linux and it is the first process that starts during boot and thus it is given the PID number 1. Systemd implements a lot of features for increased performance and system management. It also has different features for management of resources, using cgroups [*title*], which makes it interesting for our project.

Cgroups, abbreviated from control groups, can be used to set the amount of resources, such as CPU or memory, of a process or a group of processes via a virtual file system. This file system forms a tree where the resources of a parent folder are shared by its children. The division of the resources among the children are determined by the amount of "shares" the children has been given.



Each application can be run as a "service" by specifying a service file which defines many different parameters and options. In this file we can specify which program or programs should be associated with which service and for example how much CPU shall be given to this service. The service file can then be placed in a certain folder in the cgroup file hierarchy. Different folders are used to represent different cgroup controllers or a different combination of controllers. Depending on which controllers are enabled different features are available, such as limiting CPU- and memory requirement. Services can be grouped into different slices and share properties depending on which slice they belong to. One can for example set how much CPU-time that shall be given to the applications in the slice and decide how the applications will divide it amongst themselves.

2.3 Video Streaming

The video streaming is based upon the GStreamer multimedia framework. W

2.4 Equipment

During the project we used two different cameras, the M1033 and the P3367, both manufactured by Axis. We first started using the M1033 because it came with systemd but we later switched to the P3367 because it ran a later version of systemd.

Axis M1033

This is a small camera, connected to the network either wired or wireless. It supports multiple H.264 streams and Motion JPEG running at a maximum resolution of 800x600 at 30 FPS. It has two way audio streaming, which means it can both record and play audio clips.

Axis P3367

The Axis P3367 is a fixed dome network camera capable of multiple H.264 streams as well as Motion JPEG streams. It supports various frame rates and resolutions up to 5 MP at 12 FPS and of course HDTV, 1080p at 30 FPS, and has two way audio streaming capabilities. The power is supplied using Power over Ethernet, meaning it does not need a separate power supply, but is instead powered directly from the network cable. It features an ARTPEC-4 system-on-chip, developed by Axis, which contains a single-core CPU running at 400 MHz and a coprocessor dedicated to video analytics.

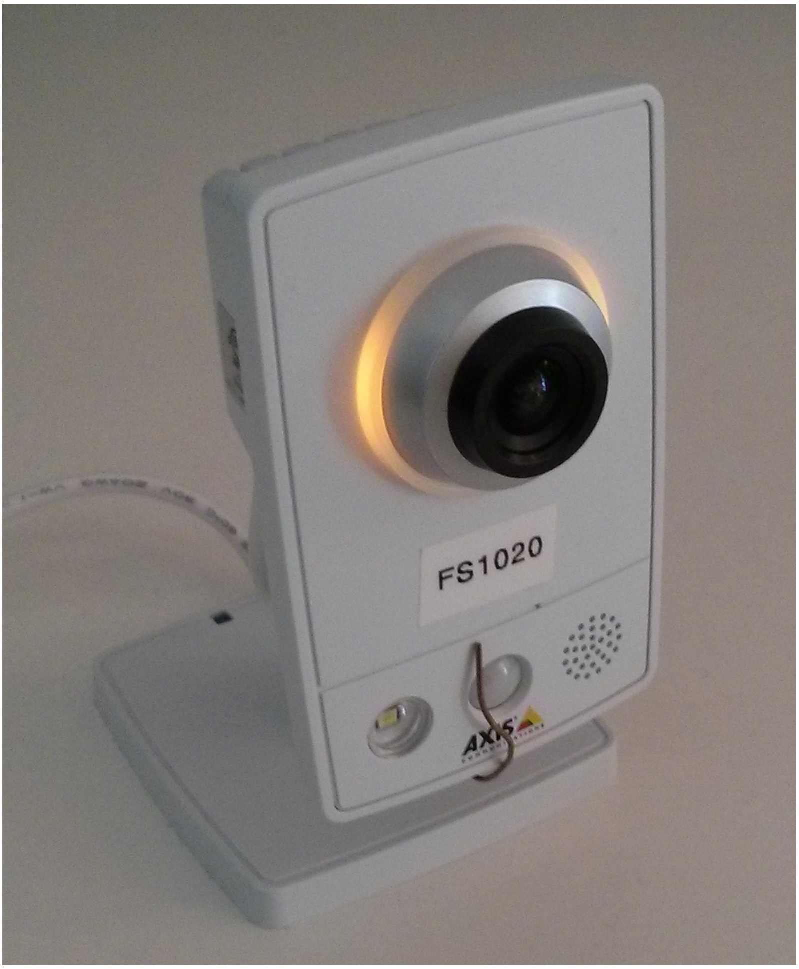


Figure 2.1 Axis M1033

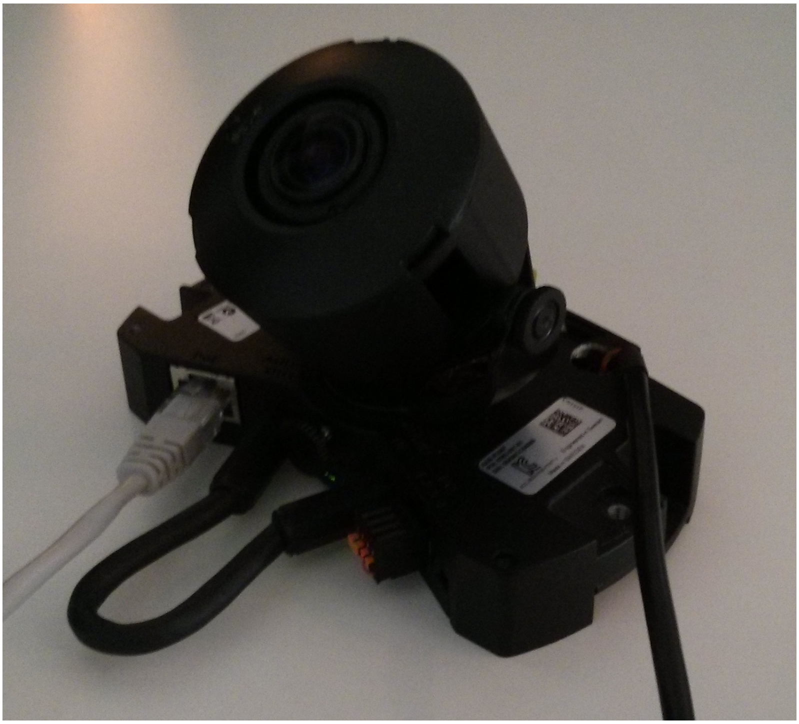


Figure 2.2 Axis P3367, without its casing

3

Implementation

All code were written in C and cross-compiled using Axis' compiler for the corresponding platform e.g. camera. For the resource allocation different service files and slices were specified, and the resulting plots were generated with Octave.

3.1 Constraints

We decided that creating service levels for all the applications running on the system would not be a realistic approach. This is because there are many different applications, some of which may not even be developed at Axis, and we cannot expect people to modify them to implement the performance measurements and service level features needed. Instead we implemented the service level part only in the video streaming application and on some test application that only does some random computations to stress the system.

3.2 Resource Management

The resource management is implemented as a part of systemd, with all of the resource management source code integrated into systemds. If an application has a poor matching function it sends this information to systemd via UNIX-sockets. Sockets represent an endpoint of communication and from these we can obtain a socket descriptor. These are used in the same way as file descriptors by the applications [socket]. Our interprocess communication (IPC) consists of a socket in systemd and one for each application that we are monitoring. From these sockets we can read or write messages between the applications and systemd. This was implemented by more or less copying the “Notify” feature of systemd which is used by certain applications that want to for example notify systemd that they have started or other status changes [sysd-notify].

The whole chain from the application to systemd is pretty long but most of it is managed in “sd-event” which serves as a wrapper around many different IPC events and messages. Our data transferred via sockets is picked up with the help of epoll, which is handled in sd-event. From this we can be notified and read data from the socket descriptor as soon it is available, in an interrupt based fashion. Finally we execute our dispatch function, which we register when the socket communication is established, and contain all the code that we want to execute when we receive a message to our socket. The function first extracts the data from the message, which is the PID of the sending application, its performance, whether it is satisfied or not and the weight attribute. We then update a hashmap which contains all our applications that are managed by GTRM. This data is then used when we calculate the virtual platforms and set the CPUShares in systemds “main-loop”.

3.3 Service Level

There are two ways of implementing SL, one simpler and one more advanced. The first one simply multiplies the current SL with the matching funtion and multiplies the result with a constant scale factor. This will decrease the service level if the performance i negative and increase it if it is positive. The scaling factor (called epsilon) also makes sure that we don’t get too much overshoot.

$$sl_i(t+1) = sl_i(t) + \varepsilon * (f_i(t) * sl_i(t))$$

The way also includes how much the virtual platform of the application has changed. This is calculated as the performance multiplier, PM.

$$PM_i = (1 + f_i) * (vp_i(i+1)/vp(i))$$

After calculating the performance multiplier the new service level can be calculated like this

$$sl_i(t+1) = sl_i(t) + (\varepsilon * sl_i(t) * PM_i)$$

3.4 Resource Allocation

Using slices we can set a minimum amount of available resources for the applications in the slice. If the applications under a slice/service don't use all of the resources given to them, the unused resources are free to be used by other slices. Under each slice there can be sub-slices or services dividing the resources further, building a hierarchy in the cgroups folder. The slices in the pie chart above, represent two different sets of applications. The static green slice consists of applications that won't be managed by GTRM and simply will share the resources under this slice according to a predefined setting. Good choices of applications to put here would be ones that doesn't vary much in their resource requirements or ones that have very strict hard deadlines. The second slice, the red GTRM slice, consists of applications that will be managed by the GTRM and that might implement SL adaptation. Good choices of applications to put here would be ones that do vary much in their resource requirements and/or can vary it's quality in some acceptable way to adapt to it's available resources.

All applications that are managed by the GTRM are run as services under the GTRM-slice. A service can, just as a slice, reserve a minimum percentage of allocated resources from it's parent slice.

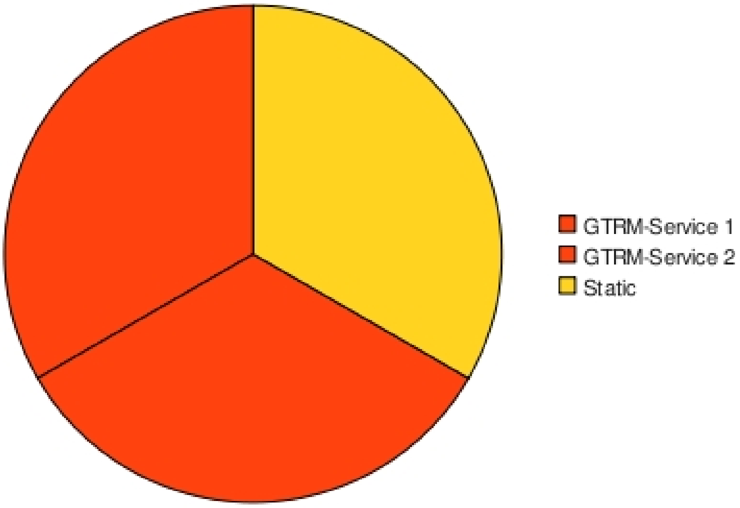


Figure 3.1 Pie chart describing how we split up our resources

In the Pie chart above the static slice would be guaranteed a minimum of ⅓ of the total resources while Service1 and Service2 would be given half of the ⅔ reserved by its parent slice, guaranteeing them a minimum of ⅓ of the total available resources each.

The resources of the applications running in the GTRM-slice will be managed by our GTRM. Some of the applications will have SL capabilities implemented and some will not. The reason for this is mainly because of the scope of our project. We are mainly concerned about using the SL to maintain a steady framerate but ideally all applications running in the GTRM-slice should, if it makes sense to the application, have a SL implemented to put the theory into practice. An extension of the project would be to have the entire system managed by the GTRM and have all applications manage their SL. The weight is the parameter that sets how big part of the adaptation that is made by changing the service level contra changing the amount of resources. By using the weight parameter we can make up for the fact that we can’t change SL of some applications and the GTRM will only manage their resources instead.

4

Use cases

For implementation and testing reasons we came up with the following use-cases. In all the use cases we assume that the static slice is under heavy load so that no extra resources are given to the GTRM-slice.

4.1 Normal mode

1. The system runs under good conditions, meaning we have enough resources for all applications and the GTRM-slice can run at maximum SL without any problems.
2. The GTRM and SL adaptation will not drag down performance compared to the old system.

4.2 Balancing a high load caused by video streaming

The applications mentioned here are all on the GTRM-slice.

1. The camera will film something that causes a high load, for example, a PTZ-camera is moving around or an intense scenery is being filmed.

2. The applications with the worst performance will adapt and lower their SL (e.g. quality) assuming they have weights setup to do so.
3. The GTRM will increase the resources given to the applications with the worst performance. These resources are taken from other, better performing, processes on the slice that in turn will lower their SL to accommodate for the change in CPU.
4. When the scenery is “calmer” we will have an overall increase on the SL and the virtual platform will be redistributed.
5. The frame rate will be about the same during the entire procedure.

4.3 Balancing a high load caused by other applications

In this case the static slice starts out not being under full load.

1. The static slice is giving extra resources to the GTRM-slice, making the applications on the GTRM-slice have a higher SL than they normally would.
2. The resource demand of the applications on the static slice starts to grow.
3. The SL of the applications on the GTRM-slice will adapt and lower their SL:s (e.g. quality).
4. The static slice is done with the more demanding tasks and starts giving extra resources to the GTRM-slice again. Now we will see an increase in the SL and a redistribution of CPU resources to the applications on the GTRM-slice.
5. The frame rate will be about the same during the entire procedure.
6. The applications in the static slice will run without any issues.

4.4 GTRM-slice not running

1. No applications are running in the GTRM-slice.
2. The static slice is allowed to use the entire amount of resources if necessary.
3. Some applications on the GTRM-slice starts to run.
4. The GTRM-slice will adjust its SL and virtual platforms as in 'use case 3' .

4.5 Applications with different weights

1. Applications with different weights are running in the GTRM-slice. All applications have a good enough performance which means that no adaptation or resource management is running.
2. One of the applications performance goes bad.
3. Resource management and SL adaptation for all applications starts.
4. The applications with the higher weights adapts mainly by increasing their cpu, which typically goes faster than adjusting the SL, making them reach a good performance faster. At the same time the applications with lower weights changes more slowly toward a better performance and adapts mainly by lowering their SL.
5. The system reaches a stable point.

4.6 System reaches stable point with some bad performances

1. A number of applications are running with good performances and no adaptation or resource management is being made.

2. A new application is started with a default SL.
3. Resource manager and SL adaptation is started.
4. The system reaches a stable point where not all applications have a good performance.
5. The applications that supports SL adaptation will have lowered this as much as possible.
6. The GTRM loop will continue to run. The SL adaptation in the applications will not run if the SL is at minimum and the performance is below the defined boundary or if the SL is at maximum and the performance is above the boundary.

5

Testing

The end result will be a working prototype that can demonstrate that the system works and what results that can be expected. These are the main aspects that we want to test.

- Can we keep the FPS we want even during high load of the system.
- Can we adapt so that other applications can perform well during high loads as well.

The tests output will consist of logs showing CPU-time for applications along with their service levels, performance and virtual platforms. The CPU-time can differ between how much resources are assigned and how much is actually used and it is of interest to see how much they vary if any. The FPS and video/sound quality are other outputs we can use. There are different things we can do and combine to stress the system in a realistic perspective. These things could be seen as testing the use cases and can be combined with each other.

- The first and simplest case is a camera that is still and filming a scenery with minor changes. Here we would like maximal quality of the images since the scene itself does not require a lot of resources

- We want to test more complex scenery, for example scenery with a lot of different things going on at the same time. This will stress the video streaming and we want to make sure that if necessary the quality will be reduced in favor for a steady FPS.

- A moving PTZ-camera will cause a high load when it is moving around. This will cause the whole picture to be redrawn and not partly as when the image is still, and will be the ultimate stress test for the streaming part.

- We will also stress the CPU and memory by creating different test applications that will be pointless but demanding in terms of resources. Just for the sake of seeing how things turn out during intense loads. Some of the applications will have a linear relationship between computational time and the SL while others will have a non linear one. This matters since the SL adaptation assumes a linear relationship between the SL and the computational time. If the steps in SL are small enough the linear relationship could still be assumed even for non linear ones.

To demonstrate how cleverly we can manage our resources, we could compare the old system with our version. We have thought of two realistic and demonstratively ways. The first would be to show how the FPS is decreased when we introduce a high load to the old system, and how we can maintain it in the new system. The high load could as an example be a SSL- key generation.

In the second example we could show a scene with both video and audio recording while stressing the camera. In the old system we would have great video quality but very poor sound performance, e.g. impossible to hear anything. But with our version the video quality would be reduced but instead one could hear what is going on as well. To use this as a conclusive experiment, we would also have to make quantitative measures of the audio, for example signal to noise ratio to see how much audio quality would improve.

The end result would also be presented as nice diagrams and graphs presenting various data that we are interested in, such as resources, bitrate and FPS.

6

Results

7

Conclusion

Bibliography

E. Bini G. C. Buttazzo, J. E. S. R. G. G. F. K. .-E. A. R. V. C. S. (2011). "Resource management on multicore systems: the actors approach". *IEEE Micro* **vol. 31**:3, 72–81.

Martina Maggio Enrico Bini, G. C. K.-E. r. (2013). "A game-theoretic resource manager for rt applications".

title. URL: `url`.