Adaptivity & Control of
Resources in Embedded Systems

D1f

COOPERATION

Interface Specification

Responsible: **TU Kaiserslautern** (TUKL)

Karl-Erik Årzén (ULUND), **Pascal Faure** (AKA), **Gerhard Fohler** (TUKL),
Marco Mattavelli (EPFL), **Alexander Neundorf** (TUKL),
Vanessa Romero (ULUND), **Stefan Schorr** (TUKL)

Project Acronym: **ACTORS**
Project full title: **Adaptivity and Control of Resources in Embedded Systems**
Proposal/Contract no: **ICT-216586**
Project Document Number: **D1f 2.0 (Final Release)**
Project Document Date: **2011-01-29**
Workpackage Contributing to the Project Document: **WP1**
Deliverable Type and Security: **R-PU**

Contents

Chapter 1

Introduction

This is the final version of deliverable D1f; it is based on the updated intermediate release. It contains the updated details of the Resource Manager's interface and also all other parts have been updated to reflect the final status.

For detailed description of the internals of the Resource Manager refer to deliverable D3b. The instructions on how to compile, install and run the Resource Manager can be found in deliverable D3c.

This deliverable is the result of task 1.5 "Requirements/QoS Interface", which is part of work package 1 "Specification and design of actor components and networks of actors" of the ACTORS project.

The deliverable presents the interface of the ACTORS resource manager from the point of view of the applications using it. The resource manager is introduced in deliverable D3a "State Abstractions", which is also available as a final version now. The purpose of the resource manager is to distribute available resources in such a way among applications that an overall maximum service quality of the system is achieved. In order to have flexibility in assigning different amounts of resources to applications, the application must be flexible, i.e. they must be able to adapt to different availability of resources. This deliverable shows how applications can support different service levels, and how this can be represented using the application interface of the resource manager. Deliverable D4b "RBS Specification" on the other hand presents the interface of the resource manager to the other side, the operating system. The results of this deliverable are necessary for all tasks of work package 3 since they are all closely related to the resource manager, and for task 2.2. In task 2.2 a run time system for CAL applications has been developed. This run time system has to provide the possibility to run system actors. System actors are the components which will actually implement the interface from the application side, i.e. they will be the ones who connect to the resource manager.

We also present the implementation of the application interface of the resource manager here in Chapters two and three. Chapter four gives an introduction into our Adaptive MPEG-4 RVC Simple Profile Decoder. It is a server based application that supports the application interface of the resource manager by offering multiple service levels.

We chose MPEG4 Simple Profile instead of SVC here for two reasons: even if Simple Profile MPEG streams don't have built-in scalability features it can still be useful to provide different service levels, i.e. reduced CPU requirements when decoding them, and right now we actually have a Simple Profile decoder written in CAL available.

The fifth chapter shows how a different kind of application, a control application can support the application interface of the resource manager, and by that shows that the proposed interface should be flexible and powerful enough for very different types of applications.

This is followed by Chapter six which concludes this deliverable with a brief summary.

Appendix A describes the extended D-Bus introspection format. Finally, Appendix B presents the Kst interface which allows to monitor the exact properties of each virtual processor in real-time.

Chapter 2

The Application Interface of the Resource Manager

2.1 Overview

The description of the DBus interface of the resource manager in this deliverable updates the documentation in the final version of the deliverable D3a.

In an ACTORS system there will be an operating system with support for resource reservations, one global resource manager running in user-space, and a number of applications. Applications participating in the resource management provided by ACTORS register with the resource manager and publish their service levels and associated resource requirements. It is then the task of the resource manager to determine the service levels for the applications and a distribution of resources, mainly CPU time among the applications which leads to a satisfying behaviour of all applications in the system. In general we expect that applications providing more detailed information about their behaviour will get a reservation which fits their needs more closely than applications providing less detailed information, which includes applications not participating in the resource management at all. The interface shall be generic enough to support not only applications written using CAL, but also in any other language.

Applications are assumed to have discrete service levels as explained in e.g. D3a. The resource requirements published by the applications can e.g. specify a CPU bandwidth requirement as a percentage of the total available CPU bandwidth.

2.2 D-Bus

The resource manager is an application running in user-space on a Linux system. It communicates with the set of running ACTORS-aware applications. There are different options for implementing the communication that could have been used, such as shared memory, pipes, or local sockets, remote procedure call methods like ONC RPC [1], Java RMI [2] or CORBA [3], but we decided to use D-Bus [4] instead. D-Bus is a message bus system, which enables applications on one computer to talk to each other. It uses a star topology with a central D-Bus daemon to which all applications connect. To talk to another application they send a message to the D-Bus daemon which forwards it to the receiver application, this constitutes the so-called bus. There can be multiple of those buses running on a system, both system-wide buses, i.e. for all users, and "local" buses for users or specific uses. In the last few years D-Bus has established itself as the standard way for communicating between applications on Linux systems. On all recent Linux distributions by default

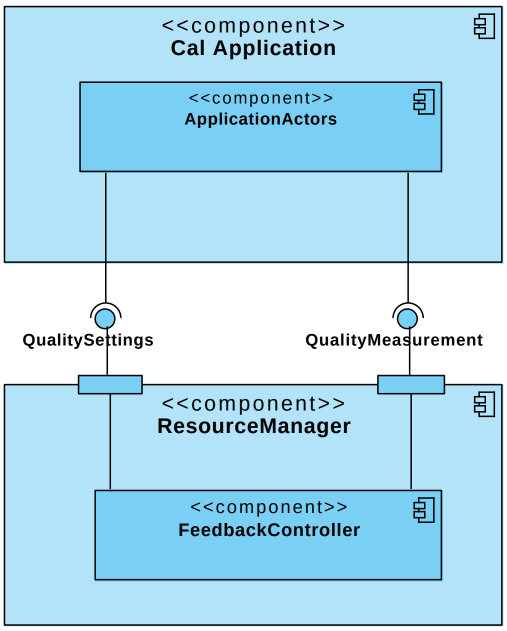


Figure 2.1: Resource Manager Applications Interface

a system bus and a per-user session bus are running. The system bus is used for system wide events, which can e.g. originate from the kernel and in this way be sent to any user application. The per-session bus is used e.g. as the main communication medium of the two major desktop environments for Linux, KDE 4 [5] and GNOME [6]. This widespread deployment and the availability of bindings for many programming languages make it the obvious candidate for implementing the interface for ACTORS. The D-Bus library is dual licensed under GPL version 2 [7] and the Academic Free License 2.1 [8], so it can be used freely both for free software projects as well as for closed software projects.

Compared to shared memory it has a higher overhead since the messages are serialized, read and written multiple times until they are available in the receiver. Since we expect that changes of the resource reservations and service levels of the application will happen infrequently, this shouldn't be a problem.

Using D-Bus cannot really be compared to using pipes or local sockets, since it is actually the same. Internally D-Bus uses local sockets for communication. So using pipes or local sockets directly for the interface would have meant that we would have to create an ACTORS-specific protocol to use on top of these sockets, and in the end we would probably end up with something like a reduced version of D-Bus, but with less features, less testing and therewith more bugs.

Compared to CORBA, D-Bus is easier to work with and it is much less complex, and it comes already with every Linux installation. Java RMI is not a candidate because it is bound to Java as implementation language. ONC RPC is network-centric and less flexible, e.g. it doesn't support sending signals as D-Bus does.

In ACTORS we use a system-wide bus (the default D-Bus system bus), since this is the straight forward way to get the communication between the clients and the resource manager working. A possible DBUS scenario for the ACTORS resource manager can be seen in Fig. 2.2.

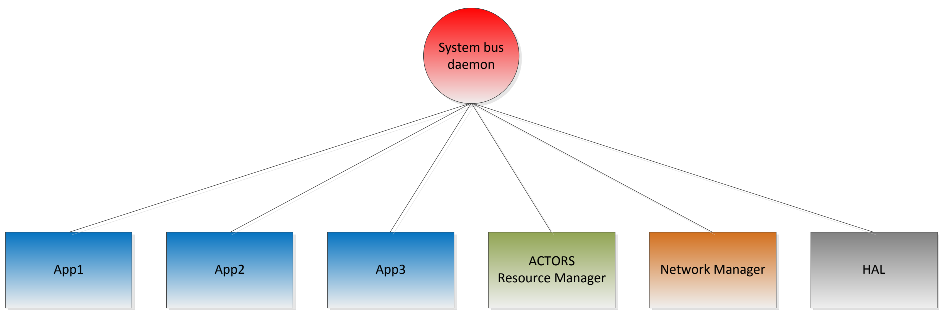


Figure 2.2: D-Bus bus setup in ACTORS

Chapter 3

Updated Interface

3.1 Interface Specification Format

The resource manager provides the application interface via D-Bus. D-Bus provides introspection functionality so that the interface of D-Bus service providers can be discovered at run-time. Introspection is provided by D-Bus objects by implementing the standard interface `org.freedesktop.DBus.Introspectable`. This interface has just one method, `org.freedesktop.DBus.Introspectable.Introspect (out STRING xml_data)`. This method returns the string `xml_data`, which contains the description of the interfaces of this object. The XML format is documented in [4] and there is a formal DTD[1] [9] for it, so the XML interface description can be verified for correctness.

Using this DTD it is possible to describe which methods are available and which arguments they support. This is enough information to document the "wire-format", but it is not expressive enough to be able to understand the semantics of the interface, e.g. composite type arguments are anonymous and this is not sufficient for code generators, so more information has to be added. The Telepathy project [10] extended the XML format to allow for more information in the interface specification [11]. This extended D-Bus introspection format will be used to specify the resource manager interface. Details about it can be found in the appendix.

3.2 Formal Resource Manager Interface Specification

Here follows the specification of the D-Bus interface of the ACTORS resource manager using the extended D-Bus introspection format. It should meet the requirements described above and in D3a.

```
<?xml version="1.0" ?>
<!-- DOCTYPE node PUBLIC
     "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
     "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd" -->

<node xmlns:tp="http://telepathy.freedesktop.org/wiki/DbusSpec#extensions-v0">
**  <interface name="eu.actorsproject.ResourceManagerInterface">

     <tp:mapping name="BandwidthDistributionMap">
        <tp:member type="u" name="Key"/>
        <tp:member type="u" name="Value"/>
     </tp:mapping>

     <tp:struct name="ServiceLevel" array-name="ServiceLevelList">
        <tp:member type="u" name="qualityOfService">
        <tp:member type="u" name="totalBandwidth">
```

[1]Document Type Definition, see e.g. http://www.w3schools.com/DTD/dtd_intro.asp

```
        <tp:member type="u" name="granularity">
        <tp:member type="u" name="bWDistributionDataSpecifier">
        <tp:member type="u" name="bWDistributionsCount">
        <tp:member type="a{uu}" prevtype="BandwidthDistributionMap"
                   name="bWDistribution">
     </tp:struct>

    <signal name="changeServiceLevel">
      <arg name="connectionId" type="s" direction="in">
      <arg name="newLevel" type="i" direction="in">
    </signal>

    <method name="reportHappiness" tp:name-for-bindings ="reportHappiness">
      <arg name="happiness" type="u" direction="in">
    </method>

    <method name="registerApp">
      <arg type="i" direction="out">
      <arg name="applicationId" type="s" direction="in">
    </method>

    <method name="unregister">
    </method>

    <method name="createThreadGroup">
       <arg type="i" direction="out">
       <arg name="groupId" type="u" direction="in" >
    </method>

    <method name="addThreadsToGroup">
       <arg type="i" direction="out">
       <arg name="groupId" type="u" direction="in" >
       <arg name="threadIdCount" type="u" direction="in" >
       <arg name="threadIds" type="au" direction="in" >
    </method>

    <method name="announceServiceLevels">
       <arg type="i" direction="out">
       <arg name="initialServiceLevel" type="u" direction="in" >
       <arg name="serviceLevelCount" type="u" direction="in" >
       <arg name="serviceLevels" type="a(uua{uu})" prevtype="ServiceLevel[]"
                 direction="in">
    </method>

    <method name="commit">
       <arg type="i" direction="out">
    </method>

  </interface>
</node>
```

Since the XML specification is quite verbose, below the interface can be found translated to a pseudo-C++-like language, which is more concise (but doesn't compile). We use an interface similar to this automatically generated from a code generator in the resource manager and potentially also in the client applications.

```
struct ServiceLevel
{
   unsigned int qualityOfService;
   unsigned int totalBandwidth;
   unsigned int granularity;
   unsigned int bWDistributionDataSpecifier;
   unsigned int bWDistributionsCount;
   map<unsigned int, unsigned int> bWDistribution;
};



class eu.actorsproject.ResourceManagerInterface
{
   public:
      int registerApp(string applicationId);
      void unregister();
      int commit();
      int createThreadGroup(unsigned int groupId);
      int addThreadsToGroup(unsigned int groupId,
                            unsigned int threadIdCount,
                            list<unsigned int> threadIds);
      int announceServiceLevels(string applicationId,
                                list<ServiceLevel> serviceLevels);
      void reportHappiness(string applicationId, unsigned int happiness);
   signals:
      void changeServiceLevel(string applicationId, unsigned int newLevel);
};
```

3.3 Documentation of the Interface eu.actorsproject.ResourceManagerInterface

This section contains the description of the interface methods of the resource manager. They are designed in such a way that the clients are purely passive, they don't have to provide any interface. They can make use of the resource manager, but they don't have to. It is also completely agnostic to whether the client applications are CAL applications or applications written in any other language.

The documentation uses the format shown below:

Method <method_name>(<parameter_name>: <type>) → <return_type>

The types are either standard D-Bus types, as presented in deliverable D1f in the appendix, or custom project-defined types. These custom types are composed also of the standard D-Bus types and documented after the methods. This is done in a similar way for the documentation of the method parameters. Here, if a parameter is a custom type, also its D-Bus type signature is shown:

`<parameter_name>:` `<custom_type><type>` <documentation>

Method registerApp (applicationId: s) → i

Each application which participates in the resource management has to register with the resource manager. The applicationId used here will be used later on by the resource manager to determine the importance value of this application.

Parameters

`applicationId:s` This string is the name of the application. It is used by the resource manager to map that client to the known applications with given importances.

Returns

`i` Integer return value, 0 on success, an error code otherwise.

Method unregister ( ) → nothing

This can be used by applications which have registered with the resource manager to close their connection with it again. It always succeeds.

Method announceServiceLevels (serviceLevels: ServiceLevel[]) → i

Called by client applications to make their supported service levels known to the resource manager. This information becomes active with a successfull call of the commit() method. Any previously announced service levels for this client are discarded then.

Parameters

`currentServiceLevel:u` The index into the following serviceLevels array of the service level, at which the application is running initially.

`serviceLevelCount:u` The number of elements in the following serviceLevels array.

`serviceLevels:ServiceLevel[](a(ua{uu}))` This is one of the most important data structures of the interface. This is the list of the service levels supported by the application. Each service level has an associated service indicator and the set of resource requirements for this level, e.g. the time granularity. All resource requirements should be hardware/platform independent.

Returns

`i` Integer return value, 0 on success, an error code otherwise.

Method createThreadGroup (groupId: u) → i

Applications can consist of multiple threads. The applications have to tell the resource manager which threads they consist of by sending their thread ids to the resource manager. Additionally these threads can be bundled to groups. This method has to be invoked in order to create a thread group.

Parameters

`groupId:u` The groupId is an integer id starting from zero to identify a group of threads. GroupIds have to be "dense", i.e., they must be a set of continuous numbers. These group ids must be unique only per application.

Returns

`i` Integer return value, 0 on success, an error code otherwise.

Method addThreadsToGroup (groupId: u, threadIdCount: u, threadIds: au) → i

This method adds a set of threads to ThreadGroup object. Each group of threads will be assigned to its own dedicated reservation. Of course it is also possible to have only one group of threads. This would also mean that these threads will not run concurrently on different cores. On the other hand it is also possible to have an own group for each thread, so in theory all threads could run in parallel given enough cores.

Parameters

`groupId:u` The groupId is an integer id starting from zero to identify a group of threads. GroupIds have to be "dense", i.e., they must be a set of continuous numbers. These group ids must be unique only per application.

`threadIdCount:u` The number of thread Ids in the array threadIds.

`threadIds:ua` The list of thread ids which should be added to this group. If the group doesn't exist yet it is created.

Returns

`i` Integer return value, 0 on success, an error code otherwise.

Method commit ( ) → i

Called by client applications after the client has announced all its information to the resource manager. The resource manager will then decide whether to accept that client or not. If it does, it will set up reservations etc. for it and move the application into this reservation. If it doesn't, the application should exit, or it is allowed to continue to run as a normal application without any guaranteed resources.

Returns

`i` Integer return value, 0 on success, which means the application has been accepted, an error code otherwise, with means the application has been rejected and should terminate.

Method reportHappiness (happiness: u) → nothing

This function is called by the client to report how well it is currently achieving the assigned quality.

Parameters

`happiness:u` An integer value between 0 and 100 expressing how well the assigned quality is actually achieved. So if the application is told to run at the lowest level and is able to do this perfectly, it should report 100, if it is told to run at the highest level but can do only 50 % of that it should report a lower value, e.g. 50.

Signal changeServiceLevel (connectionId: s, newLevel: u)

This signal is emitted to notify an application about the service level it should run on.

Parameters

`connectionId:s` This is the connection identifier, for DBus usually something like ":1.51" (i.e. it is NOT the unambiguous name under which the application registered with registerApp()). This id can be retrieved e.g. using dbus_bus_get_unique_name().

`newLevel:u` The index of the service level at which the application should run on.

Struct ServiceLevel - (uuuuua{uu})

A struct representing one service level of an application, i.e. the quality and the required resources. It is used in announceServiceLevels(). In bindings that need a separate name, arrays of ServiceLevel should be called ServiceLevelList. The service level with the number zero refers to the service level providing the highest quality of service; applications running in service levels with larger numbers usually provide less quality.

Members

`qualityOfService:u` This one indicates the quality of this level. It is an integer value ranging from 0 (worst) to 100 (best).

`totalBandwidth:u` The total bandwidth the client will consume when running in this service level, e.g., 200 to indicate two complete CPUs.

`granularity:u` granularity contains the period in microseconds at which this client will have to be sampled on this service level.

`bWDistributionDataSpecifier:u` The bWDistributionDataSpecifier specifies how to treat the values stored in bwDistribution: 0: there is no data specified by the application, 1: the values should be treated as absolute values, 2: the values should be treated as relative values.

`bWDistributionsCount:u` bWDistributionsCount contains the number of items contained in the following map bWDistribution.

`bWDistribution:BandwidthDistributionMap[] (a{uu})` This is the map of bandwidth mappings for the individual virtual processors. Values have to fulfill 0 < x <= 100;

3.4 Information Flow

The overall flow of information from and to the resource manager is depicted in Fig. 3.1.

In the center the resource manager is located. An application which wants to have its resources managed by ACTORS has to actively register itself with the resource manager. This is done by connecting to the respective D-Bus bus and calling the `registerApp()` method of the resource manager. The next step is to announce its supported service levels by calling the method `announceServiceLevels()`. Each service level consists of a numerical value which gives an indication for the quality of this level, `ServiceLevel::QoS`, together with a set of resource requirements for each ThreadGroup in this level stored in the map `ServiceLevel::bwDistribution`.

When multiple applications have registered in this way with the resource manager, the resource manager will determine a distribution of resources which leads to the desired overall system behaviour. This includes determining the service level and the parameters of the reservation for each application. The applications will be notified via the `changeServiceLevel()` D-Bus signals about their assigned service levels. Here signals are used because otherwise the applications would have to provide a D-Bus interface themselves, which would increase the amount of modifications required to add support for the resource manager to applications.

When the application receives this notification, it shall adapt its inner algorithms in some way, so that it uses the specified resources while producing the promised quality. This may not always be successful, for instance due to varying computational demands from the work load, e.g. in an MPEG stream [12]. It is an important information for the resource manager whether the application is able to achieve the promised quality of the assigned service level. It is up to the applications to determine an integer value between 0 and 100 which expresses how well it currently achieves the designated service level, we'll call this number the *Happiness* of an application. Each application will periodically inform the resource manager by calling the `reportHappiness()` method about its current Happiness.

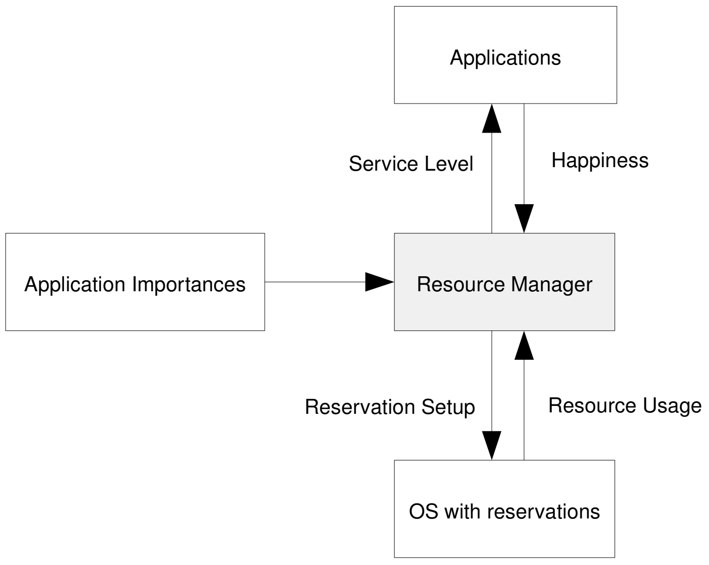


Figure 3.1: Information flow from and to the resource manager

Additionally to these Happiness numbers the resource manager will be provided with importance values for the applications. They are defined by the user or system integrator, and serve as hint to the resource manager which applications it should prefer and which can degrade first. Primary functions of the system like e.g. phone calls for a mobile phone should be assigned a higher importance than add-on functionality like e.g. an animated effects in the user interface. Practically the importance values could be provided using an XML-file which is read by the resource manager.

When the resource manager has determined a distribution of resources, it will set up the reservations accordingly. This will be done by specifying bandwidth and period or equivalent parameters as e.g. (α, Δ) as introduced in deliverable D3a, for the reservations and sending them to the operating system. Then the applications can run within these reservation. The operating system should also report back some resource usage information, i.e. whether deadlines are missed or to which percentage the reservations are used by the applications.

The resource manager will take its decisions based on this information together with the resource usage information from the operating system.

3.5 Example Session with the Resource Manager

Fig. 3.2 shows the communication between two applications *App1* and *App2* with the resource manager as a sequence diagram. At the beginning the resource manager is already running. Then App1 starts and connects to the resource manager, this is done in steps 1 to 7. After that the resource manager determines the distribution of resources, which should be easy since there is only one application. Then a second application starts and registers with the resource manager (step 8 to 12).

Therefore, in step 13 the resource manager notifies App1 by sending a D-Bus signal about a service level change since App2 got assigned some resources. At some point App1 reports its happiness back to the resource manager, as can be seen in step 14. This can happen once or multiple times as in the example. Then App1 and App2 run for some time without significant events happening, but informing the resource manager from time to time about their happiness. Then App1 creates new threads and adds them to an already existing ThreadGroup (step 19 and 20). After some time App1 terminates. The resource manager detects this (step 21) shortly before App2 also terminates (step 22).

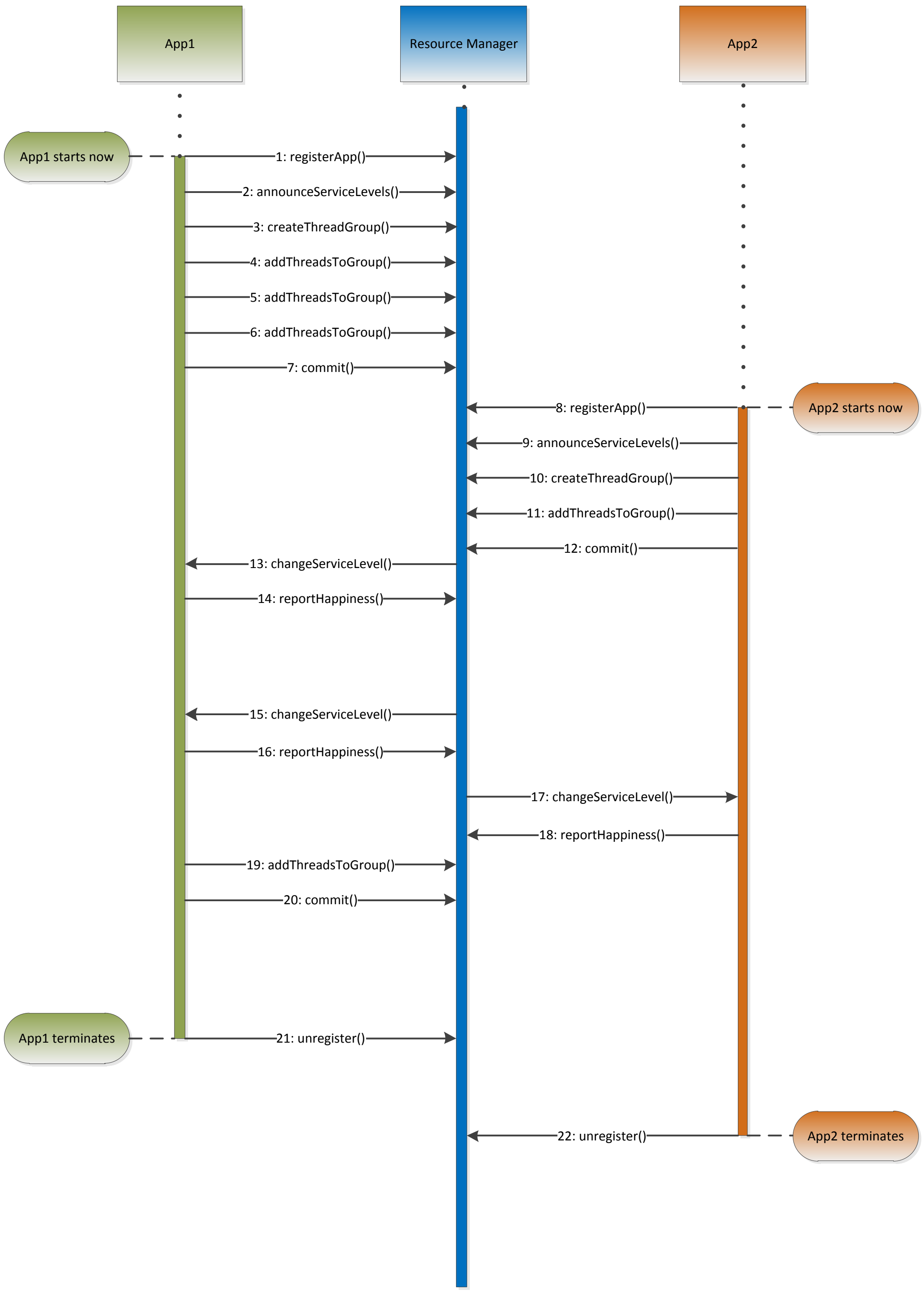


Figure 3.2: Example communication sequence between the ACTORS resource manager and two applications

Chapter 4

Example Application: An Adaptive MPEG-4 RVC Simple Profile Decoder

4.1 Overview

An example application which can have multiple service levels is a MPEG video decoder. Different service levels of a video decoder can e.g. concern the frame rate, the resolution of the decoded image or the number of skipped frames.

We added support for multiple service levels to a Simple Profile MPEG-4 decoder. Our decoder is a served based implementation, as shown in Figure 4.1. The video decoder is a client application registered with the ACTORS Resource Manager running on SCHED_EDF. It is a CAL based program, in charge of decoding and displaying the MPEG-4 video stream. The video itself originates from a web cam which is connected to the server. This web cam sends its video stream to the server application which handles the TCP/IP streaming to the client application.

The Simple Profile of MPEG-4 doesn't include the features supporting adaptivity present in higher MPEG-4 profiles, e.g. it doesn't support B-, SP- and SI-frames, streams in multiple solutions, feedback to the encoder or other advanced features of higher profiles [13].

Nevertheless we investigated options on how to make also a Simple Profile MPEG-4 decoder adaptive. There are multiple options, they are introduced in the following sections. To implement these or one of these options, we implemented a system actor, which announces the service levels to the resource manager. The

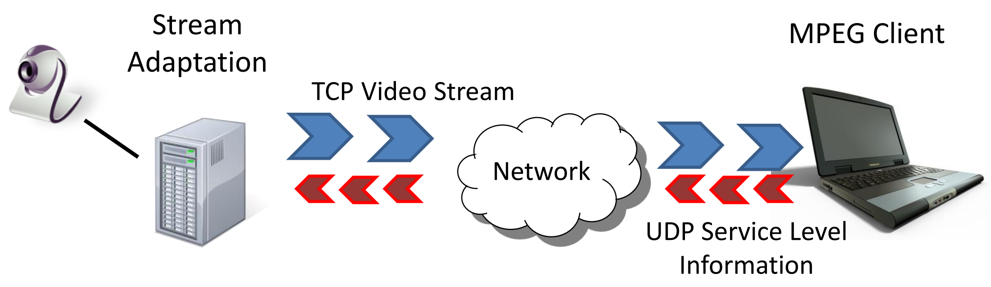


Figure 4.1: Overview of the Adaptive Simple Profile MPEG4 decoder

same actor also receives the decisions from the resource manager on which service level to run and ”translates“ this to tokens which are sent to the appropriate actors. Additionally, we implemented a small UDP protocol that allows the client application to inform the streaming server about its current service level. This allows the server to react to the changing resource availability on the client side. There exist different configuration files for the decoder to adjust for usage on e.g. quad-core and dual-core devices.

Frame Skipping

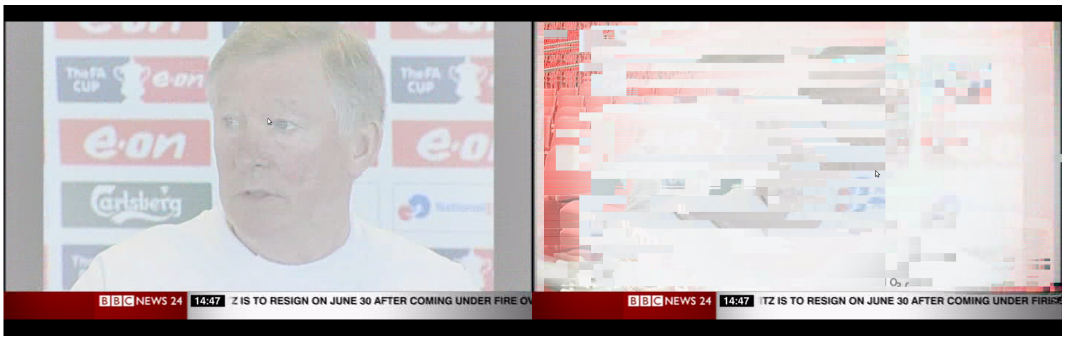
Quality Aware Frame Skipping (QAFS) [14] is a technique focused on MPEG2 video streams. The main idea is to extract meta data from the video stream and use this information to take sensible decision about which frames to skip in the case that not enough CPU time is available to decode all frames. This method requires gathering information from multiple frames, in order to be able to select one or more of those for skipping. In short, a Group Of Pictures (GOP) is considered, and then frames which contribute the least to the video quality are discarded first. This means B-frames are the first “victims” of this algorithm, and just after all B-frames in a GOP are discarded, P-frames are considered.

The MPEG decoder – since it is a Simple Profile decoder – doesn't support B-frames. This means that the QAFS algorithm cannot simply be applied to this decoder.

Macro Block Skipping

Alternatively to skipping whole frames, we considered skipping the decoding of macro blocks within a frame. An advantage compared to skipping whole frames is that it is not necessary to buffer multiple frames, introduced by the need to buffer a whole GOP in QAFS. If a macro block is skipped, the content from the previous frame will be displayed in the area this macro block encoded.

However, our approaches revealed that while an implementation of a macro block skipping algorithm is quite straight forward, the resulting MPEG-4 video quality is unacceptable, as can be seen in Figure 4.2. Removing whole macro blocks immediately results in nasty block artifacts that annoy the audience and finally lead to completely destroyed pictures.


BBC NEWS 24
14:47
Z IS TO RESIGN ON JUNE 30 AFTER COMING UNDER FIRE O
BBC NEWS 24
14:47
TZ IS TO RESIGN ON JUNE 30 AFTER COMING UNDER FIR

(a) Original scene. (b) After applying macro block skipping.

Figure 4.2: Macro block skipping resulting in poor video quality.

DCT Coefficient Skipping

Another alternative is to decode only the lower frequency DCT coefficients and skip the higher frequency ones. Different service levels are achieved by adjusting the number of DCT coefficients to skip. This way quite fine grained service levels are achievable. This results in saving of the decoding time on the client side but produce an image which is less sharp or has minimally visible square artifacts, but is still much better than with macro block skipping see 4.3.

That is why we have chosen the DCT coefficient skipping algorithm for our final implementation. The next section will explain the different service levels that our decoder offers.

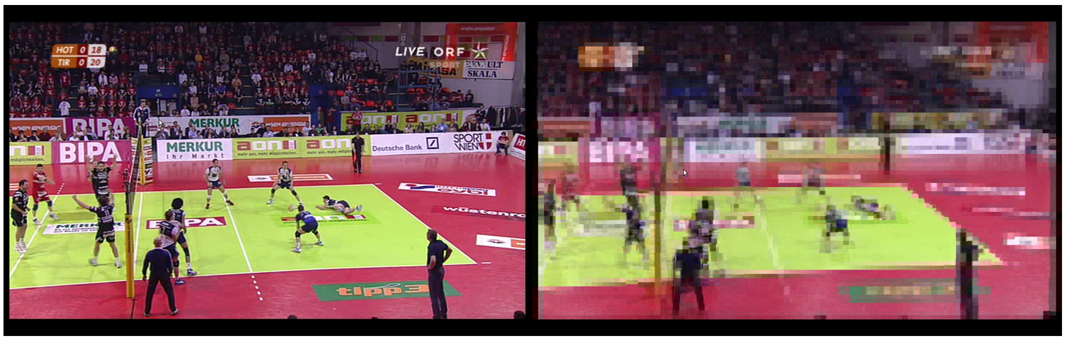

(a) Original scene. (b) After applying DCT coefficient skipping.

Figure 4.3: DCT coefficient skipping resulting in reduced video quality.

4.2 The Service Levels of the Adaptive MPEG4 RVC SP Decoder

We defined the following three service levels for our adaptive MPEG4 RVC SP decoder:

Service Level 0

The decoder works with maximum quality, nothing is skipped and the original video is untouched and send to the client. It has maximum CPU bandwidth demands, i.e. the application cannot consume more CPU time than in this mode.

Service Level 1

This service level consumes a bit less CPU than the service level 0. and still produces acceptable quality. The original video stream is modified like this: the algorithm tries not to touch the I frames. Only in P frames, starting from the ending of a GOP, some macro blocks are skipped.

With a modified algorithm like this the quality of the produced video frames will be lower than at level 0. We assume that the quality at this level will be 60 to 80 % of the maximum quality, depending on the original video streams properties.

Service Level 2

Running at this level the decoder should consume significantly less CPU but result in a significantly reduced video quality. The output stream bandwidth is set to a minimum to allow for video playback even under severely constrained resource availability. One interesting feature of this “mode of operation” is that it allows to keep a video stream alive and still provide a basic video playback when a normal best effort approach would fail to deliver any video information.

Chapter 5

Example Application: A Feedback Controller

5.1 Introduction

As another example of how a CAL application of a completely different type, but still with multiple quality levels can interact with the resource manager a feedback controller is used. The example is intentionally kept very simple in order to emphasize the interaction with the resource manager.

The aim of the control system is to control the position of a rolling ball on a beam. The so called ball and beam process, hence, consists of a horizontal beam and a motor that controls the beam angle. The measured signals from the process are the beam angle relative to the horizontal plane and the position of the ball. The process is shown in Fig. 5.1.

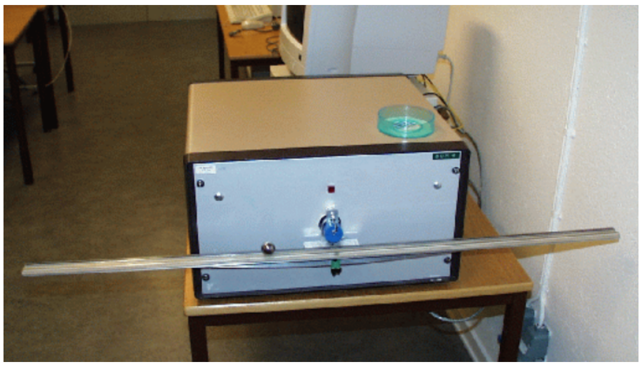


Figure 5.1: The Ball and Beam process

The dynamic model from the motor to the ball position consists of two transfer function blocks connected in series, in which the beam angle appears as an intermediate output signal, see Fig. 5.2. Processes of this type are often controlled by cascade controllers where an inner controller controls the dynamics of the first block and an outer controller controls the overall dynamics. Often PID controllers are used both for the inner and outer controllers. The cascade control structure is shown in Fig. 5.3. In cascade structures the output (i.e. control signal) of the outer control loop is used as the reference signal for the inner controller. Cascade control structures, often with more than two layers, are very common in industrial practice, e.g., in automotive combustion engine control.

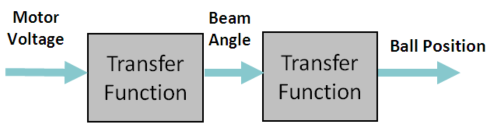


Figure 5.2: Ball and Beam Model Structure

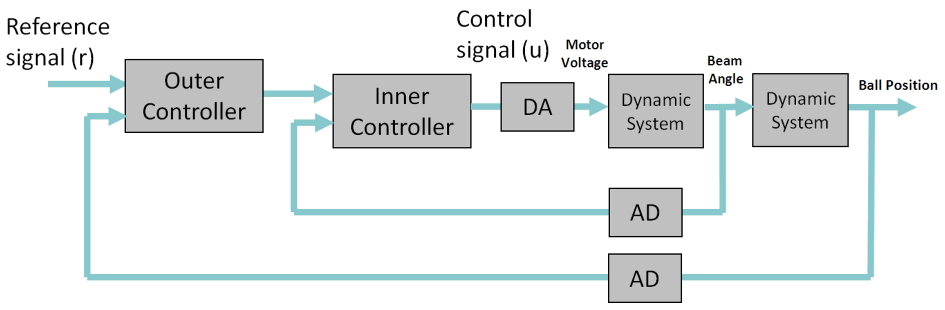


Figure 5.3: Cascade control structure for the Ball and Beam process

The PID controller is the most common controller type in industry. The "text-book" version of the PID-controller can be described by the equation

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int^t e(s)\, ds + T_d\,\frac{de(t)}{dt}\right)$$

where the error e is the difference between the reference signal r (the set point) and the process output y (the measured variable). K is the *gain* or *proportional gain*, T_i the *integration time*, and T_d the *derivative time* of the controller.

However, some modifications are necessary in order to get good control performance. A pure derivative cannot, and should not be, implemented, because it will give a very large amplification of measurement noise. The gain of the derivative must thus be limited. This can be done by approximating the transfer function sT_d as follows:

$$sT_d \approx \frac{sT_d}{1 + sT_d/N}$$

The transfer function on the right approximates the derivative well at low frequencies but the gain is limited to N at high frequencies.

It is also advantageous not to let the derivative act on the reference signal and to let only a fraction b of the reference signal participate in the proportional part. The PID-algorithm then becomes

$$U(s) = K\left(bR(s) - Y(s) + \frac{1}{sT_i}\Big(R(s) - Y(s)\Big) - \frac{sT_d}{1 + sT_d/N}\,Y(s)\right) \tag{5.1}$$

where U, R, and Y denote the Laplace transforms of u, r, and y.

A controller with integral action combined with an actuator that saturates can give some undesirable effects. If the control error is so large that the integrator saturates the actuator, the feedback path will be broken, because the actuator will remain saturated even if the process output changes. The integrator, being an unstable system, may then integrate up to a very large value. When the error is finally reduced, the integral may be so large that it takes considerable time until the integral assumes a normal value again. This effect is called *integrator windup*.

There are several ways to avoid integrator windup. A common method for *antiwindup* is illustrated by the block diagram in Figure 5.4. In this system an extra feedback path is provided by using the output of the actuator model and forming an error signal e_s as the difference between the estimated actuator output u and the controller output v and feeding this error back to the integrator through the gain $1/T_t$. The error signal e_s is zero when the actuator is not saturated. When the actuator is saturated the extra feedback path tries to make the error signal e_s equal to zero. This means that the integrator is reset, so that the controller output is at the saturation limit. The integrator is reset to an appropriate value with the time constant T_t, which is called the tracking-time constant.

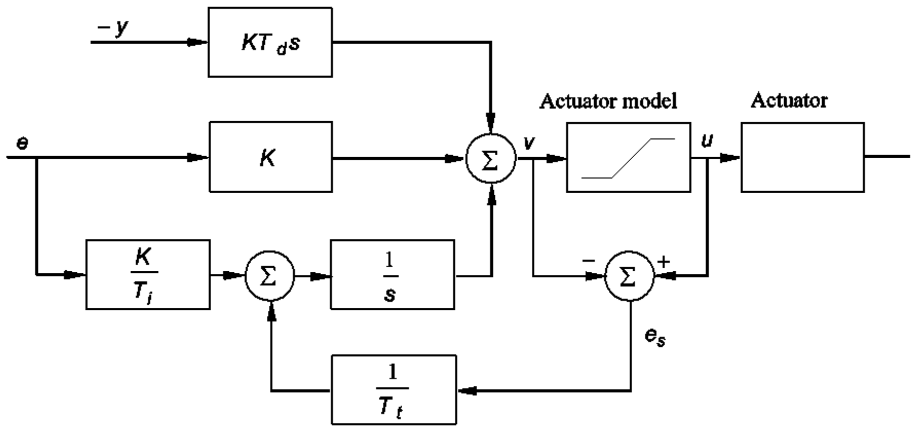


Figure 5.4: PID controller with anti-windup. The actuator output is estimated from a mathematical model of the actuator.

In order to implement a digital PID controller the PID algorithm must be discretized. This can be done in several ways. The following is one common way of doing this. The proportional part

$$P(t) = K\Big(br(t) - y(t)\Big)$$

requires no approximation because it is a purely static part. The integral term

$$I(t) = \frac{K}{T_i}\int^t e(s)\,ds$$

is approximated by a forward difference approximation, that is,

$$I(kh+h) = I(kh) + \frac{Kh}{T_i}\,e(kh)$$

where h is the sampling period. The derivative part given by

$$\frac{T_d}{N}\,\frac{dD}{dt} + D = -KT_d\,\frac{dy}{dt}$$

is approximated by taking backward differences. This gives

$$D(kh) = \frac{T_d}{T_d+Nh}\,D(kh-h) - \frac{KT_dN}{T_d+Nh}\,\Big(y(kh) - y(kh-h)\Big)$$

The control signal is given as

$$u(kh) = P(kh) + I(kh) + D(kh) \tag{5.2}$$

In order to implement tracking anti-windup the update equation for the integral part is modified slightly. The resulting pseudo-code for a PID controller looks as follows:

```
y = yIn.get();
e = r - y;
D = ad * D - bd * (y - yold);
v = K*(b*r - y) + I + D;
u = sat(v,umax,umin)}
uOut.set(u);
I = I + (K*h/Ti)*e + (h/Tt)*(u - v);
yold = y
```

Here `ad` and `bd` are pre-calculated parameters representing the parameters in the discretization of the derivative part. The saturation function, `sat`, limits `v` between the end values, i.e. it corresponds to the internal actuator model.

A too long delay between the input and the output has a negative effect on control performance. The code above is therefore structured so that the computational delay between the input of the measurement signal (the sampling) and output of the control signal (the actuation) is as small as possible. Therefore, the update of the state variables `I` and `yold` are performed after the actuation. To split the code in two parts, commonly called `CalculateOutput` and `UpdateState` is very common in control implementation. It also influences the execution order in cascade controller structures. In order to minimize the input-output latency for a cascade controller it is important to execute the parts of the algorithm in the following order:

```
Sampling
Outer.CalculateOutput
Inner.CalculateOutput
Actuation
Inner.UpdateState
Outer.UpdateState
```

5.2 Basic CAL Model

A first CAL model of the cascade controller for the Ball and Beam process is shown in Fig. 5.5. The model contains two instances of a PID CAL actor, one clock system actor, two input system actors, and one output system actor. The PID actor contains two actions, one in which the `CalculateOutput` part of the algorithm is performed and one in which the `UpdateState` part is performed. The state variables in the controller and the variables that have to be saved between the invocations of the two actions are represented as state variables in the actor. In the example we assume that the reference value for the outer controller is constant and contained within the corresponding actor.

The clock actor periodically generates a trigger token. The input and output actors are the interface between the CAL application and the external IO. This interface can be implemented in a variety of ways. The semantics is, however, that, an input actor should produce a current sample each time it receives an input trigger token and that the output actor should send the value contained in the input token to the actuator.

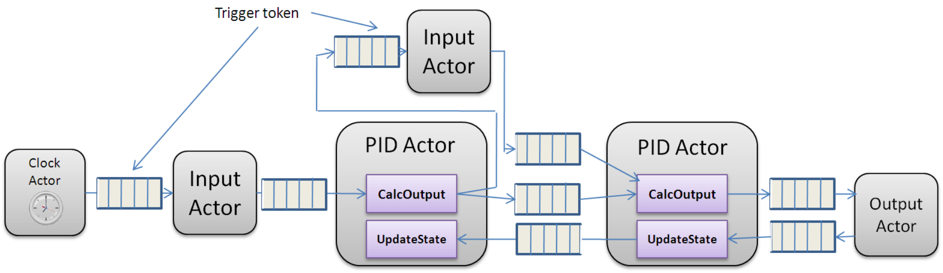


Figure 5.5: CAL Model of the cascade controller.

5.3 The Quality Function

As described in Deliverable D3a control applications naturally have a mapping from resource usage to quality. A controller is typically designed for a nominal desired sampling period. By increasing the sampling period the consumed computing resources decrease and also the control performance. This relationship between sampling period and performance is often linear or quadratic. If, in addition to changing the sampling period, also the controller parameters are updated in accordance with the new sampling period, the performance decrease becomes smaller.

A natural candidate as a performance measure is a continuous-time quadratic cost function defined as

$$J = \lim_{T\to\infty} \frac{1}{T} \int_0^T \begin{bmatrix} x(t) \\ u(t) \end{bmatrix}^T Q \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt$$

where Q is a positive semi-definite matrix, $x(t)$ is the state variable vector, and $u(t)$ is the control signal vector. An advantage with the quadratic cost function definition of control performance is that for a large class of systems it is possible to calculate the expected value of the cost function analytically off-line using, e.g., the Jitterbug tool [15]. The output of this tool would then typically be a graph that relates the control cost (the inverse of the performance) with the sampling period.

The cost could be expressed as a function of the sampling period, h either as

$$J(h) = \alpha + \beta h^2$$

or as

$$J(h) = \alpha + \gamma h$$

One could combine this with a maximum value of h for which the achieved performance is considered acceptable, i.e., the resource manager should ensure that the control application always receives at least the corresponding amount of resources.

The cost function can be used as a quality sensor. In this case the cost is calculated on-line by the control application and compared to the nominal value for the current service level. The difference between the nominal cost at the current service level and the measured cost can then be used as a basis for calculating a happiness value that is periodically sent back to the resource manager.

5.4 Extended CAL model

In order to be practically useful the simple model in Fig. 5.5 has to be extended. For example, it should be possible on-line to change the controller parameters from, e.g., some user interface. This requires that the PID actors are extended with extra inports and actions through which new controller parameters can be sent. To simplify the presentation we will assume that CAL supports structured tokens, i.e. a new set of controller parameters are sent as a single token to the PID actor. When a parameter token is available a separate action, `UpdatePars`, is triggered that updates the controller parameters. The situation is shown in Fig. 5.6. The

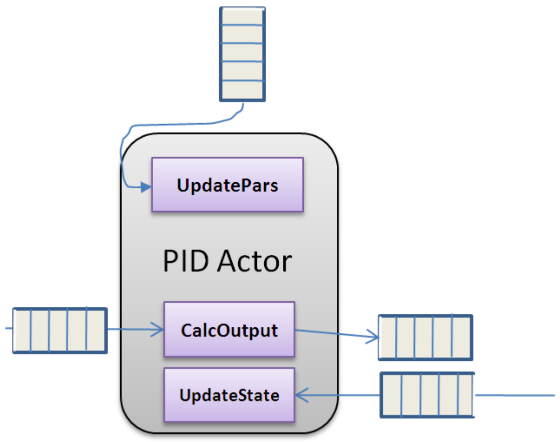


Figure 5.6: PID actor with parameter update action.

same approach could be used to handle on-line changes of the reference signal.

The service level information of the applications would be registered with the resource manager at the initialization of the application by executing a special system actor. This system actor would call the interface methods `registerApp()` and `announceServiceLevels()` with the associated parameters as arguments. An example of where it is included in the CAL model is shown in Fig. 5.7. The One-Shot

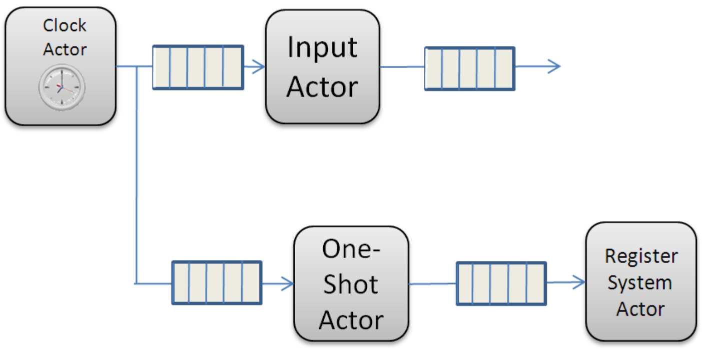


Figure 5.7: System actor for announcing the service level information.

actor is an ordinary CAL actor that only emits an output trigger token the first time it receives an input token. The Register System Actor (also referred to as the DBus actor in Deliverable D5b) contains a call to the Resource Manager interface. The service level information that would be transferred to the resource manager for a typical control application are shown in Chapter 5 in D3a.

Once the Resource Manager has distributed the resources it reports back to the application which service level it has been assigned. This is done through the `changeServiceLevel()` signal. This signal is catched by a special system actor that listens for this particular signal and when it arrives sends the amount of resources to an ordinary CAL actor that translates the service level to the corresponding sampling period and recalculates the controller parameters. The new sampling period is sent to the clock actor and the new parameters are sent to the two PID actors., see Fig. 5.8. Alternatively a single system actor could be used as the single interface

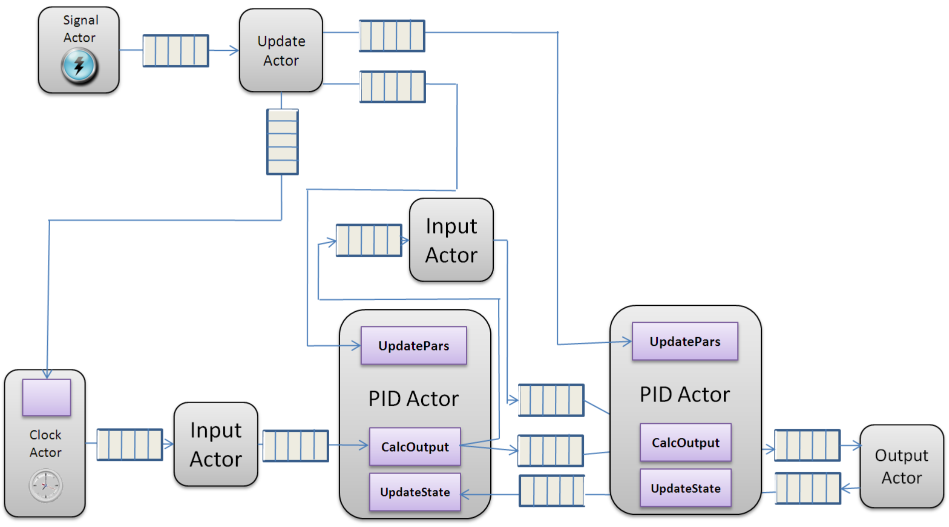


Figure 5.8: Receiving the new service level. The Signal actor is a system actor that receives the new service level. This is forwarded to the Update actor where the new controller parameters are computed. The new parameter sets are forwarded to the PID actors and the new sampling period is forwarded to the clock actor.

to the resource manager via the DBus interface. This is the solution used in the ACTORS demonstrators and in Deliverable D5d.

The quality measurement could be implemented by an ordinary CAL actor that based on the control signal and the measurement signal updates the cost function and every n'th time sends the value to another actor where the happiness value is calculated and finally sent over to the resource manager by a special system actor that internally calls the `reportHappiness()` method in the interface. The approach is illustrated in Fig. 5.9. Not shown in the figure is the connection between the Update Actor and the Cost Function Actor where the current sampling interval is propagated. This is necessary in order to be able to compute the cost function correctly. Also here an alternative solution is to use a single system actor as the interface to the resource manager. This is what has been used in the demonstrators.

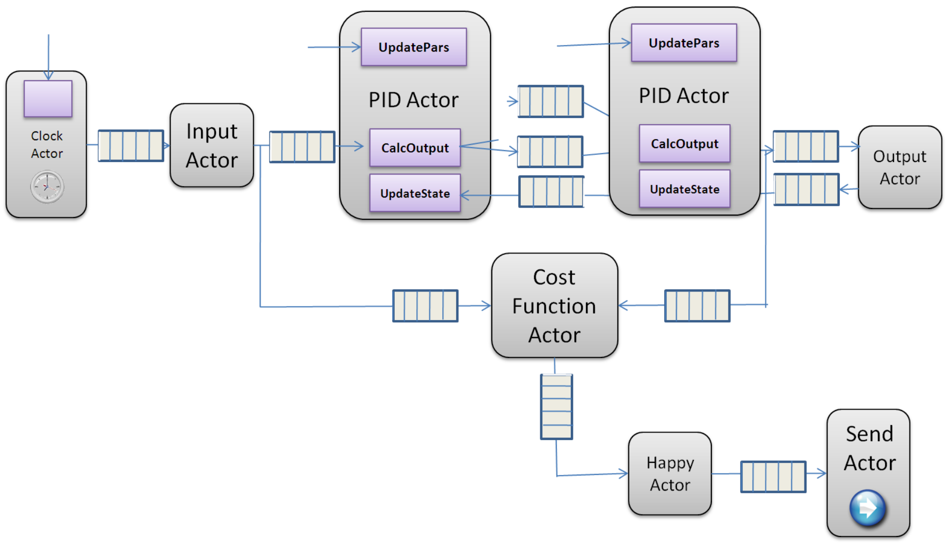


Figure 5.9: Calculation of obtained quality. A cost function actor updates the cost function. In the happy actor this value is converted to a happiness value that is sent to the resource manager in the system actor Send Actor.

Chapter 6

Conclusion

The focus of this deliverable is to explain and specify the details of the DBus interface between the Resource Manager and its client applications. For detailed description of the internals of the Resource Manager refer to deliverable D3b. The instructions on how to compile, install and run the Resource Manager can be found in deliverable D3c.

Chapter 2 gave a a general summary of the idea of the DBus interface. This was followed by Chapter 3 which presented a comprehensive overview of all available DBus method and their parameters. This included a description of the methods as well as an example communication sequence between the Resource Manager and two client applications. Chapter 4 presented our Adaptive MPEG-4 RVC Simple Profile Decoder and discussed the pros and cons of different ways to achieve adaptivity. The following chapter showed a totally different application scenario: a control application offering different service levels.

Finally, this deliverable is concluded by an Appendix that presents the extended DBus introspection format and the Kst Real-Time Plotting and Data Viewing Interface.

Bibliography

[1] Sun Microsystems, Inc., "RFC 1057, RPC: Remote Procedure Call Protocol Specification, Version 2." http://www.ietf.org/rfc/rfc1057.txt, 1988.

[2] Sun Microsystems, Inc., "Java Remote Method Invocation." http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[3] Object Management Group, Inc., "Common Object Request Broker Architecture (CORBA) Specification, Version 3.1." http://www.omg.org/technology/documents/corba_spec_catalog.htm.

[4] H. Pennington, A. Carlsson, and A. Larsson, "D-Bus Specification." http://dbus.freedesktop.org/doc/dbus-specification.html.

[5] "The K Desktop Environment." http://www.kde.org.

[6] "GNOME: The Free Software Desktop Project." http://www.gnome.org.

[7] Free Software Foundation, Inc., "GNU General Public License, version 2." http://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

[8] "The Academic Free License 2.1." http://open2.mirrors-r-us.net/licenses/afl-2.1.php.

[9] D. A. Wheeler, "DTD for D-Bus Introspection data." http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd, 2005.

[10] "Telepathy: Flexible communications framework." http://telepathy.freedesktop.org.

[11] "Specification D-Bus introspect format extensions, Version 0." http://telepathy.freedesktop.org/wiki/DbusSpec#extensions-v0.

[12] D. Isovic, G. Fohler, and L. Steffens, "Some misconceptions about temporal constraints of mpeg-2 video decoding," in *23rd IEEE Real-Time Systems Symposium*, 2003.

[13] I. E. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia.* Wiley, 1 ed., August 2003.

[14] D. Isovic and G. Fohler, "Quality aware MPEG-2 stream adaptation in resource constrained systems," in *16th Euromicro Conference on Real-time Systems (ECRTS 04)*, (Catania, Sicily, Italy), 2004.

[15] A. Cervin and B. Lincoln, "Jitterbug 1.1—Reference manual," Tech. Rep. ISRN LUTFD2/TFRT--7604--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, Jan. 2003.

Appendix A

The Extended D-Bus Introspection Format

A.1 The D-BUS Introspection DTD

Below you can find the current DTD for the D-Bus introspection format. It defines the basic XML format used for D-Bus interfaces.

```
<!-- DTD for D-BUS Introspection data -->
<!-- (C) 2005-02-02 David A. Wheeler; released under the D-BUS licenses,
         GNU GPL version 2 (or greater) and AFL 1.1 (or greater) -->

<!-- see D-BUS specification for documentation -->

<!ELEMENT node (interface*,node*)>
<!ATTLIST node name CDATA #REQUIRED>

<!ELEMENT interface (annotation*,method*,signal*,property*)>
<!ATTLIST interface name CDATA #REQUIRED>

<!ELEMENT method (annotation*,arg*)>
<!ATTLIST method name CDATA #REQUIRED>

<!ELEMENT arg EMPTY>
<!ATTLIST arg name CDATA #IMPLIED>
<!ATTLIST arg type CDATA #REQUIRED>
<!-- Method arguments SHOULD include "direction",
     while signal and error arguments SHOULD not (since there's no point).
     The DTD format can't express that subtlety. -->
<!ATTLIST arg direction (in|out) "in">

<!ELEMENT signal (arg,annotation)>
<!ATTLIST signal name CDATA #REQUIRED>

<!ELEMENT property (annotation)> <!-- AKA "attribute" -->
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property type CDATA #REQUIRED>
<!ATTLIST property access (read|write|readwrite) #REQUIRED>

<!ELEMENT annotation EMPTY> <!-- Generic metadata -->
<!ATTLIST annotation name CDATA #REQUIRED>
<!ATTLIST annotation value CDATA #REQUIRED>
% \end{verbatim}
```

A.2 The Extended D-Bus introspection format

Using the D-Bus DTD it is possible to describe which methods are available and which arguments they support. This is not expressive enough to be able to extract semantics from it, since e.g. the composite type arguments are still anonymous.

E.g. for code generators more information has to be added. The Telepathy [10] project extended the XML format to allow for more information in the interface specification. The basic and extended elements are documented below.

<node name="/org/freedesktop/sample_object"> A node describes a D-Bus object. A D-Bus object has a name and it can support multiple D-Bus interfaces.

<interface name="org.freedesktop.SampleInterface"> A D-Bus interface is similar to a class in C++ or Java. It can contain methods, signals and properties.

<method name="Frobate"> A method is a member function of an interface which can be called by other D-Bus clients. It can take input and output arguments.

<arg name="foo" type="i" direction="in"/> <arg> is a method argument. The attribute direction specifies whether it is an input or output argument. The type attribute specifies the type of the argument. D-Bus supports basic types as int or string and also composite types as struct, array and map. The complete list of types is in Table A.1.

<signal name="changeContinuous"> A <signal> is a message which is sent out by an interface and which can be received by interested other objects on the bus. It can have arguments the same way as methods.

<property name="Bar" type="y" access="readwrite"/> Properties are basically public member data, which can be read and modified.

<tp:docstring> Encloses HTML-formatted documentation for the parent tag.

<tp:enum name="ResourceId" value-prefix="Resource" type="u"> Allows definition of an enum type, i.e. an integer type with a restricted set of values, which are also named. This is then typically mapped to an unsigned integer type, e.g. UINT32. The attribute value-prefix should be used by code generators as a prefix for all values of this enum.

<tp:enumvalue suffix="None" value="0" /> <tp:enumvalue> is only valid inside enclosing <tp:enum> tags. It defines one of the valid values for this enum type. The attribute suffix will be appended to the value-prefix from the enclosing <tp:enum>.

<tp:mapping name="ResourceDemand"> This defines the name of a map from one type to another type. Used together with <tp:member>

<tp:struct name="QualityLevel" array-name="QualityLevelList"> This gives an struct defined using (...) a name. A separate name for an array of these structs can be specified. It is used together with <tp:member>

<tp:member type="u" tp:type="ResourceId" name="Key"/> When used inside <tp:mapping>, there should be always exactly two, one for the key and one for the value of the map. The names don't matter much then.

When used inside <tp:struct>, they define an individual member of that struct.

The tp:type attribute This attribute can be used everywhere additionally to the type attribute. It allows to define the type in more detail, e.g. to restrict the possible values of an integer to those of an enum, or to give members of a struct names.

Conventional Name	Code	Description
INVALID	NULL	Not a valid type
BYTE	'y'	8 bit unsigned integer
BOOLEAN	'b'	Boolean, 0 is FALSE, 1 is TRUE
INT16	'n'	16 bit signed integer
UINT16	'q'	16 bit unsigned integer
INT32	'i'	32 bit signed integer
UINT32	'u'	32 bit unsigned integer
INT64	'x'	64 bit signed integer
UINT64	't'	64 bit unsigned integer
DOUBLE	'd'	IEEE 754 double
STRING	's'	null-terminated UTF-8 string
OBJECT_PATH	'o'	Name of an object instance
SIGNATTURE	'g'	A type signature
ARRAY	'a'	An array of the type which follows
STRUCT	'(...)'	A struct consisting of the enclosed types
VARIANT	'v'	A variant type, the type is part of the value itself
DICT_ENTRY	'{...}'	A map or dictionary mapping from the first enclosed type to the second

Table A.1: D-Bus Type Signatures

There is no "official" formal specification of these extensions yet, so documents cannot formally be verified for correctness. Due to the deficiencies of the DTD language producing a formal specification for the format of D-Bus interfaces using XML Schema or RelaxNG would help in this regard.

Appendix B

The Kst Real-Time Plotting and Data Viewing Interface

B.1 Overview

During the development of the ACTORS resource manager we had the need to monitor the resource managers decisions and their influence on the client applications behavior. Especially the development and integration of the resource managers different logic types greatly benefited from this real-time interface. Inside the resource managers feedback algorithms, we integrated a small interface to follow the essential decisions the resource manager makes.

The data format this interface uses is directly compatible to the Kst data format. Kst is a freely available open source application under the terms of the GPL. It is the fastest real-time large-dataset viewing and plotting tool available and provides basic data analysis functionality.

B.2 Description

Whenever a client successfully registers with the resource manager, a new file called kstfile<id>.kst is created in the folder `/tmp/Actors` (where id is a consecutive number). At run-time, these files are filled with the properties of the clients. Each column in the file stands for a unique property, in total 7 properties are measured: It is the VP index, the average used budget, the hard reservations, the assigned budget, the average used bandwidth, the assigned bandwidth and the assigned period that are being monitored.

In order to separate the contents of the file, a small python script (`/trunk/resourcemanager/kst/split7.py`) can be used like this:

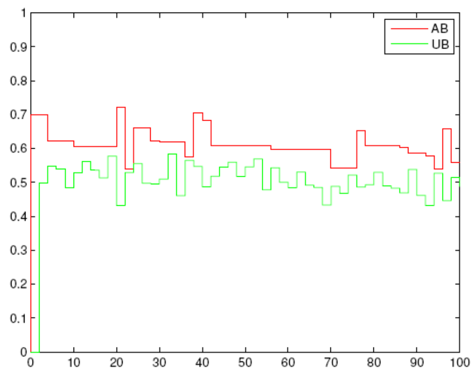
```
>../resourcemanager/kst/split7.py prefix kstfile0.kst kstfile1.kst
```

This script will then create the following files, each only containing information about a single virtual processor:

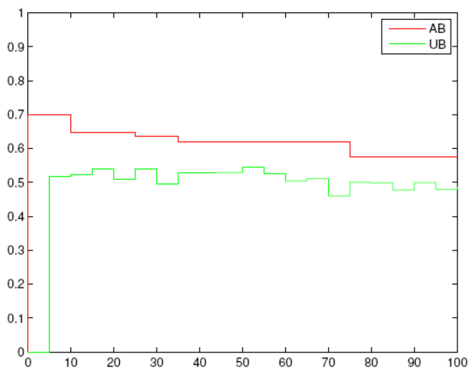
- prefix_0_vp_1
- prefix_0_vp_2
- prefix_0_vp_3
- prefix_0_vp_4
- prefix_1_vp_1

- prefix_1_vp_2
- prefix_1_vp_3
- prefix_1_vp_4

These files can directly be read by Kst. Figure B.1 shows such an example real-time measurement of 2 virtual processors using Kst.



(a) VP1: Assigned and used budget



(b) VP2: Assigned and used budget

Figure B.1: Example real-time measurement of 2 virtual processors using Kst.