A state of the art media box

PIERRE LABIAUSSE

KTH Information and Communication Technology

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

A state of the art media box

Master thesis

Pierre LABIAUSSE

Master of Science Thesis

Communication Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

1st March 2013

Examiner: Professor Gerald Q. MAGUIRE Jr.
Supervisor: Gaël MARRONNIER (*Simstream S.A.*)

Abstract

Today media centers are often cluttered with multiple devices each controlled by their own remote control. It is often hard and/or painful to manage and utilize these devices, especially for inexperienced users. Simstream wants to build an innovative smart-TV that as much as possible centralizes functions and controls. Operating the system should be intuitive and simple, yet experienced users should have access to more advanced operations.

This requires acquiring several inputs as well as integrating the communication devices that are necessary to control the attached external devices. Whenever possible, we want to efficiently process every input while minimizing latencies. As a result, we want all the frequent operation to be as quick and lightweight as possible in order to provide a high quality user experience even under high system loads.

This project takes advantage of the widespread availability of touchscreen mobile devices in order to provide an innovative means of control over the television, with remote control mobile applications running on an user's familiar device. A remote controller will also be sold together with the television, and this remote controller will also have a touchscreen, and will propose the same capabilities as the remote control mobile applications.

Finally, this platform will be open to third-party applications, and as a result this thesis project developed a software development kit which is designed to be easy and familiar enough for developers to adopt it and create applications with it. Applications will be developed together with an interface displayed on the remote controllers, in order to tailor the remote control interface to what is currently displayed on the television screen.

Sammanfattning

Idag är mediecentrer ofta belamrade med många enheter som är kontrollerade av sina egna fjärrkontroller. Det är ofta svårt och / eller smärtsamt att använda dessa enheterna, särskilt för oerfarna användare. Simstream vill bygga en innovativ smart TV som centraliserar funktioner och kontroller så mycket som möjligt. Att använda systemet ska vara intuitivt och enkelt, men mer erfarna användare ska också ha tillgång till mer avancerade funktioner.

Detta kräver att förvärva flera indata samt att integrera kommunikationsenheterna som är nödvändiga för att kontrollera de anslutna externa enheter. När det är möjligt vill vi behandla varje indata på ett effektivt sätt oh samtidigt minimera latenser. Det här betyder att en operation som utförs ofta skall vara så snabb och så lätt som möjligt, för att förbättra användarupplevelse även när systemet är hårt belastad.

Detta projekt drar fördel av den vidsprädda tillgången till pekskärma mobila enheter för att tillföra användaren en innovativ kontroll över sin TV, direkt från sin bekanta enhet.

Slutligen kommer denna plattformen att vara öppen för tredjepartsutvecklare, och som ett resultat har detta examensarbete utvecklat ett *software development kit* som är gjort för att vara enkelt och välbekant nog för att utvecklare ska kunna använda det och skapa applikationer med det.

Acknowledgements

I would like to thank everybody at Simstream, and especially my supervisors (Gaël MARRONNIER and Yann BARERA), for giving me the responsibility of developing this prototype, and for welcoming me into their company in a really friendly manner. It was a big project, spanning a lot of different areas in computer science, and I have learned a lot with it.

My gratitude also goes to my academic supervisor and examiner, Gerald Q. MAGUIRE Jr., for his numerous and valuable pieces of feedback. Without his help, insights, and commentaries on the different versions of my work, this thesis would have been far from what it is today.

Finally, I'm also deeply grateful for my family and friends, for their unconditional love and support throughout the years.

Contents

List of Figures

List of Tables

List of listings

List of Acronyms and Abbreviations

ACPI	Advanced Configuration and Power Interface
ADC	Analog to Digital Converter
API	Application Programming Interface
APU	Accelerated Processing Unit
AM	Applications Manager
CM	Command Manager
CPU	Central Processing Unit
FSM	Finite State Machine
GCC	GNU Compiler Collection
GUI	Graphical User Interface
IPC	Inter-Process Communication
JVM	Java Virtual Machine
LCD	Liquid Crystal Display
LED	Light Emitting Diode
NIC	Network Interface Controller
NFS	Network FileSystem
PID	Process Identifier
OS	Operating System
RTT	Round-Trip Time

SDK	Software Development Kit
STB	Set Top Box
SysV	UNIX System V
TV	Television
UART	Universal Asynchronous Receiver Transmitter port
VOD	Video on Demand
WM	Window Manager
WOL	Wake-On-Lan

Chapter 1

Introduction

1.1 Presentation of the project

Today we often find several devices connected to the television in a typical home (such as an external hard drive allowing the user to pause live-TV and record programs, DVD and Blu-Ray drives, or set-top boxes (STB) rented or sold by triple-play providers[1] to enable the user to get access to additional content). Every additional external device added to the media center may come with its own remote control. The collection of devices quickly becomes difficult to install and to use, especially for users who do not want to spend lots of time learning to manipulate each of these electronic devices. Using such an external device also typically supposes dedicating the whole screen to it, the underlying smart-TV effectively being used only to select an input channel to display on screen and then letting the external device operate, without further intervention. Moreover, using a smart-TV is rarely painless – especially for more advanced operations – due to a general lack of an interface that is easy to understand and to use.

Simstream wants to build a solution which would allow the user to easily access and operate the aggregate of these multiple devices via a unified interface. Media can be streamed from the collection of external devices – disk players, STBs, gaming consoles, etc. – to the system. The system needs not simply display one of these signals exclusively, but instead might display several streams at the same time, while transferring only one audio stream to the audio output. A typical scenario to illustrate a common use case would be making a call via the TV, while keeping an eye on a football match that is taking place in real-time. The ability to utilize several media streams simultaneously and in an integrated fashion is a major difference with traditional TV sets for which a selected signal goes directly

[1]Triple play services provide Internet access, telephony, and television – often including channels in addition to national channels – over a single broadband connection

to the display screen, and the TV's remote control only selects which input should be displayed and heard at any given time. Moreover, a smart-TV can provide its own set of functions, which potentially makes several external devices redundant – hence reducing the clutter surrounding the media center. The goal is to shift the media center from being a purely media player environment to a full-fledged media and communications platform to which information flows seamlessly to and from external devices (including those not directly linked to audio-visual activities, such as home-automation devices). This provides the media center with additional capabilities coming from the computer world, exploiting the convergence trend between media playing, computing, and communication (see the preface of [1]). Computers are gaining more and more TV-like capabilities (a perfect example for this phenomenon would be a home theater personal computer which consists of a low-end computer running dedicated softwares providing multimedia capabilities, such as XBMC [2]). On the other hand, connected TVs have more and more computational capabilities, even if they are not often easily exploited (see section 3.3).

In order to provide a better way to interact with a smart-TV than a regular remote control, Simstream wants to use a touchscreen tablet as a remote control. This remote control comes with its own set of challenges: It has to be powerful enough to support the touchscreen and to handle communications with the TV, yet be as lightweight as possible in order to run from a resource-constrained platform with severe power consumption limitations. The Contiki operating system could be used for such a remote control (see section 2.2). Moreover, mobile applications are proposed to allow users to control their TV from their own mobile devices.

The TV itself does not have as many constraints as a mobile device with respect to available resources, but the power consumption should still be kept as low as possible as it is an important criterion when choosing a home appliance (see [3] for the EU guidelines concerning the energy labelling of televisions). Additionally, the TV must be responsive to external events – such as an incoming calls – even when idle. This requires that when the system in idle mode, consuming next to no power, it must be able to quickly resume activity to handle the external activity. Advanced Configuration and Power Interface (ACPI) S3 mode (*suspend to RAM*) could be used for this purpose, as the system may remain responsive to external signals (e.g. via the local network with wake-on-LAN, see section 4.5.1), while allowing a fast switch to the *on* mode (as a full reboot is not performed). This S3 mode requires only that the RAM remains powered so that it can be automatically refreshed.

Finally this platform will provide support for third-party applications to extend its capabilities (e.g. adding home automation remote controls) with the help of a dedicated software development kit (SDK). This SDK must be straightforward to use for developers, as this is the key for its adoption by a community, while being powerful and versatile enough to enable developers to develop engaging applications that exploit the novel interactions provided by the remote control.

1.2 Objectives for this thesis

The objective of this thesis project is to develop a prototype for the system described above. It does not have to be comprehensive in terms of functionality, but must instead illustrate the intended objectives of the system. This project can be divided into three distinct sub-projects: the television's system itself, the remote controls (especially the iPhone remote control application), and the applications for the television. This last part is primarily centered on providing the necessary tools to develop applications, rather than developing the applications themselves. However, several applications were developed in order to illustrate the benefits of an innovative touchscreen remote control and to evaluate the quality of the software development tools. This thesis utilizes several different programming techniques, with an emphasis on the benefits of an event-driven approach to software development.

1.3 Structure of this thesis

Chapter 2 provides background information relevant to this thesis project, which can allow for a better understanding of the remaining of the thesis for the reader. Chapter 3 presents the current state of the IP-able, smart televisions, and set-top boxes. Chapter 4 details the architecture and the operations of the television system. Chapter 5 presents the applications for the television, their conception and the development tools used to create them. Chapter 6 introduces the remote controls principles and gives details on the implementation of the iPhone remote control application. Chapter 7 evaluates the system using efficiency and usability criteria. Finally Chapter 8 concludes this thesis and gives a brief overview of features that could be added to the prototype in the future.

Chapter 2

Background

This chapter aims to provide the reader with the terminology and a basic understanding of the technologies that are going to be used in this project. It builds on related work concerning these technologies, describes their advantages and disadvantages, and gives concrete examples of their utilization.

2.1 Threads versus Events

Threads and events are two major programming paradigms. Both of which have their advantages and disadvantages. This section introduces these two models (subsections 2.1.1 and 2.1.2), and presents several views opposing the two paradigms or trying to combine their strengths (subsection 2.1.3)

2.1.1 Threads

Multithreaded systems consist of multiple simultaneous threads of control within the same user space [4]. Threads are a familiar way of programming, as every thread represents a sequential flow of control, with a single point of execution and a reserved segment of stack address space per thread. A thread belongs to a process (often called "task"), and all threads within the same process share the same (virtual) memory space, which allows for efficient inter-thread communications. Concurrency is introduced by a scheduler, which is usually preemptive in a multi-threaded system, i.e., a thread can be interrupted while running. If a thread is preempted, a context switch occurs, i.e, the state of the current running thread (e.g. its registers) is stored and the state of the next thread to be run is loaded. The cause of this interrupt may be a higher-priority thread that becomes runnable (for priority-based schedulers) or that the time allocated for the thread (its "quantum") has run out (for time-slicing schedulers). The duration of the quantum is a tradeoff between efficiency and reactivity: If the quantum is too low, then the overhead of context switching becomes excessive, and if is too big, a running thread with a long computation would delay the processing of a user input in another thread, which could make the system appear as non responsive. A typical value for the quantum in modern systems' schedulers is 1ms which allows threads with long computations to make progress, while bounding the delay before other threads get a chance to run. Moreover, the majority of threads will block on I/O operations during their quantum and hence yield control of the CPU.

Although threads have been used for a long time, their functionality does not come for free. Threads are memory consuming, since each thread is allocated a dedicated stack address space when it is created, which is generally overprovisioned with respect to the thread's real needs. The creators of the Mach kernel addressed this issue by providing an optional continuation passing style for threads [5]: When yielding, threads can choose to register a callback function to be called on its next quantum, which allows the code to save the state of the thread by simply storing a callback funtion and its parameters and to discard the stack associated with the thread. Protothreads also make use of continuations, with some limitations (see section 2.3).

Threads also introduce the need for explicit synchronization and the difficulty of using threads correctly often leads to lots of subtle bugs (such as deadlocks and race conditions) which are extremely hard to locate and remove – this is due to the concurrency model and the indeterminism it introduces [6]. A resource shared by several threads (such as a variable) could be accessed concurrently by two or more threads. If no synchronization technique is used, two threads could read the value of the variable simultaneously, and update it based on its value. Such a scenario – illustrating a *race condition* – is likely to corrupt the value, as the second update of the variable overrides the first without taking it into account. To address this issue, developers introduce *critical sections* into their code. A critical section is a part of the code that can only be executed by one thread at a time. However, this technique adds a lot of boilerplate code and decreases the readability and the overall efficiency of a program, since threads have to *take turns* when accessing a common resource. Moreover, deadlocks may be introduced in the code, in the event of two threads that have a lock on two different resources, but that need access to the resource held by the other before they can exit their critical section. As a result of the need for synchronization, thread programming becomes harder and harder with the growth in the size of project, hence this approach often leads to buggy software.

2.1.2 Events

Event-driven systems [7] use a completely different way to provide concurrency. These systems are built on the *Handlers* design pattern, where events generated by applications are enqueued, and a dispatcher blocked in an infinite event loop takes an event from the queue and sends it to the appropriate event handler. The event handler then processes the event. Generally the handler either processes the complete event (in a run-to-completion fashion), or it does part of the work and sets up another event to complete the rest of the work. The latter approach is widely used in device drivers so as to minimize the time that interrupts are disabled.

This programming paradigm is different from the thread-based approach, as control is reversed when compared to traditional multi-threaded systems. Indeed, in an event-driven system, one cannot see any discernable flow of control, as the code is simply executed in response to events. Events control the applications. This approach can lead to scheduling improvements as scheduling of code execution is dynamically based on the application's actual requests for services. This model is used extensively for graphical user interfaces (GUIs). Java provides several event handler interfaces in the *java.awt.event* package (e.g. *ActionListener*, *MouseEventListener*). GUI programmers simply write "plug-in" pieces of code and register them with the appropriate GUI components, then when an event for which a handler is registered occurs (e.g. a keyboard button is pressed or released) the appropriate handler is called by the Java virtual machine (JVM) and passed an *Event* object containing the event type (here a *KeyEvent*) as well as some additional information (e.g. which key caused the event).

However, for longer running operations, the run-to-completion model of an event-driven system is limited, because it prevents the use of blocking abstractions. Indeed it is not possible for a function to block on a condition (e.g. waiting for an I/O operation to complete) in a non preemptive environment without delaying the handling of other events. As a consequence, event-driven programmers use finite state machines (FSMs) to control the flow of higher logic operations together with asynchronous function calls, using the so called *Hollywood principle* ("Don't call us, we'll call you"): An application requests an operation to be done by another component (by posting an event), and the other component informs the first component when it is done (by sending another event). As a result, the application does not block waiting for the operation to be completed (as it would in the synchronous model), but instead the first component is re-scheduled to be resumed when this operation has been completed. The main difference from the asynchronous model is that an application does not maintain its original stack, thus it has to keep track of the progress of the operation in some manner (hence the use of a FSM).

Event programming is conceptually harder because of the inversion of control it introduces, and the need for fragmenting blocking-operations in order to avoid delaying the processing of later events. However, event programming also has the benefit of having only one thread of control, i.e. one single point of execution, which makes debugging easier and avoid subtle (and often timing dependent) bugs that can avoid detection during the testing phase of an application.

2.1.3 Combining both paradigms

Both models have their advantages and disadvantages as well as their opponents and proponents. Ousterhout claims that *"Threads are a bad idea (for most purposes)"* [8], because of the inherent difficulty of using them without introducing bugs, and because of their poor performances (memory overhead due to the presence of multiple stacks, and time overhead due to frequent context switches). He advises the use of threads only when true CPU concurrency is needed, and to avoid using threads by keeping the code as single-threaded as possible. Dabek *et al.* have developed *libasync* [9], a C++ non blocking I/O library to make event-based programming more convenient. The library provides support for registering *callbacks*, i.e. functions to be invoked when a given event occurs. They have also proposed a solution to handle multiple processors (in *libasync-mp*) while avoiding synchronization problems. In their proposal, programmers associate a *color* with a callback, and the library guarantees that only one callback of a given color runs at any given time. This improvement is optional, as callbacks are given a default color which ensures that if the code is not explicitly designed to take advantage of a multi-processor environment, it will use only one processor; thus freeing the programmer from having to think about synchronization. Protothreads (see section 2.3) aim to provide a block/wait abstraction in an event-driven environment in order to facilitate state management.

In contrast, von Behren *et al.* claim that *"Events are a bad idea (for high-concurrency servers)"* [10]. They claim that concurrent requests on a typical server are largely independent and that their processing usually follows sequential steps, making threads a more natural abstraction for this environment. Threads provide a better way to control the flow of operations, by removing the need for manually saving and restoring state before and after function calls, and providing better handling of resources along unexpected paths (e.g. when an exception occurs). Moreover, compiler optimizations (such as dynamic stack sizes, temporary data removal before function calls, and compiler level detection or handling of data races) may decrease memory consumption and reduce the probability of synchronization related bugs related to threads.

Finally, some people have tried to reconcile the two paradigms. Lauer and Needham state that "Message-Oriented systems" (i.e. event driven) and

"Procedure-Oriented systems" (i.e. thread based) are duals of each other, and that they can yield similar performance results [11]. However, this statement is based upon specific implementations, different optimization possibilities, and how the problem at hand is more adapted to one model or the other. Li and Zdancewic propose integrating the best of the two worlds in a hybrid system [12], taking advantage of the expressiveness of threads as well as the flexibility and performance of events. Their application level framework (written in Haskell) provides a common interface for both models by relying on an intermediate representation that abstracts threads and their system calls – conceptually located above – as well as event handlers processing events from an easy to customize event loop. This allows a programmer to abstract away details of the underlying continuation passing style used, which is very similar to what is done by protothreads (section 2.3).

2.2 The Contiki operating system

Contiki [13] is a lightweight operating system designed to run on sensor nodes (or "motes") belonging to wireless sensor networks. Wireless sensor networks can be composed of a large number of such nodes, which communicate between themselves and with central sinks, to monitor phenomena by reporting different physical parameters (e.g. temperature, noise levels, humidity, vibration, etc.) as measured by their on-board sensors [14]. Contiki is designed to run on low-cost memory-constrained platforms, where power consumption is a major concern. Once the node is deployed, it needs to function without direct human interaction for potentially long periods of time.

Contiki is written in C and provides both an event-driven kernel and support for multi-threaded programs through a library. As described in [15, 16], porting Contiki to a new platform is quite simple, especially if the targeted platform is close to an existing port. Contiki also supports protothreads (see section 2.3), and is able to load and unload individual libraries and applications, instead of having to modify the binary for the entire system as in most embedded operating systems (such as for tinyOS [17]).

Apart from CPU multiplexing and support for loadable programs, Contiki does not provide any OS abstraction and applications are responsible for managing hardware resources themselves. This provides more flexibility at the application level, and allows the introduction of application specific optimizations of the usage of hardware resources [18, 19].

2.3 Protothreads

As seen in section 2.1.2, the pure event-driven model blocks waiting for pending operations, as event handlers must run to completion before they yield the CPU. Event handlers should thus yield the CPU as soon as possible. Without the ability to wait on a condition (e.g. the completion of an I/O request), the event-driven programmer must resort to state machines to control the flow of high-level logical operations involving more than one event handler. While FSMs are powerful modeling tools, they lack support for explicit state as utilized in popular programming languages. This can make the state difficult to determine in the code, thus making debugging harder.

Protothreads [20] provide a block/wait abstraction in an event-driven environment. Protothreads are based on local continuations, which encapsulate the state of the execution of a program. This state consists of the location of the code that the program is currently executing (the continuation point) as well as the values of all the local variables. The local continuation of a protothread is *set* when it uses the block/wait abstraction (either conditional: *PT_WAIT_UNTIL*, or unconditional: *PT_YIELD*) and *restored* when the protothread execution resumes. Protothreads are currently implemented either with a GNU Compiler Collection (GCC) C language extension (namely the label-as-values extension), which limits their portability, or with ANSI C, which prevents the use of switch statements anywhere in the code. Both of these approaches currently do not preserve the values of automatic variables (i.e. variables allocated on the stack) across a blocking statement. In order to preserve variables, programmers using protothreads must use static local variables (allocated in the data section of memory), which in turn may cause problems for reentrant code. This limitation explains why the memory overhead of protothreads is so low (the size of a pointer), as only continuation points are currently preserved.

Protothreads are designed to work independently of the underlying scheduling method. In Contiki, a protothread is scheduled every time an event is dispatched to the process implemented by it. Protothreads can be considered as blocking event handlers, which allows the programmer to make the high-level logical flow of the program more apparent in the source code (as compared to a large state machine with either one or more subfunctions). In the majority of cases it is possible to entirely remove the FSM thus greatly reducing the number of states and state transitions when using protothreads.

However, protothreads do not pretend to be a silver bullet in the domain of event-driven applications. They can greatly reduce code complexity in a large number of cases (i.e. suppressing or reducing the need for an implicitly defined FSM and reducing the number of lines of code). But in some cases protothreads can become more cumbersome than the FSM they try to replace

[21]. If protothreads greatly simplify the coding style for sequential problems (e.g. making an I/O request, waiting for the result, and then doing something with it), this is only because only a limited number of valid sequences are possible. When multiple sequences of events can occur at a given point, the translation from a carefully designed state machine to its protothread equivalent may introduce deeply-nested *if-then-else* sequences with boolean flags keeping track of previous state changes, resulting in code that is difficult to debug.

2.4 Using a Linux Kernel as a base

While Contiki is a good fit for the remote controller, the TV itself is a more complex machine, and it lacks the computational, memory, and power constraints of a remote control unit. Moreover, the TV will integrate a large number of communication and I/O interfaces (Wi-Fi, Ethernet, USB) as well as – for the prototype anyway – a recent AMD CPU with integrated graphical capabilities. Porting Contiki to support all of this hardware would necessitate manually writing and maintaining dedicated drivers to interact with each piece of hardware. Conversely, utilizing an existing OS and tuning it to achieve high performance for our particular needs and devices frees us from this burden. For example, the Android system has been built on top of a Linux kernel, acting as a hardware abstraction layer, on which Google has built their own operating system [22, 23] (while modifying the underlying linux OS at the same time, in order to tailor it to their particular need).

An alternative would have been to integrate drivers from another operating system into a new OS, as Bryan Ford *et al.* have done when creating the Flux OSKit by aggregating OS components from both Linux and FreeBSD, in order to allow rapid bootstrapping of a new OS [24]. They used encapsulation for each legacy component ("glue code"), in order to feed the component the information that the component expects, and to convert the information received from the component into a common format. While this project is very interesting, as it allows the programmer to select the component that is better suited for a particular need (e.g. to make a performance/memory requirements tradeoff) or which is more mature, this project has not been active for quite some time, and the source code is no longer available. As a result, achieving similar results would require too much time to be practical.

Starting from a well-documented OS with extensive support on the Internet allows us to get a lot of functionality for free, while still being able to iteratively tune the implementation, for example by including high-performance APIs (such as the one presented in section 2.5).

2.5 The Netmap framework

The netmap API [25] aims to be capable of saturating a high speed network interface (typically an Ethernet interface operating at 10 Gbit/s or roughly millions of packets per second) by reducing per-packet processing costs *without* having to modify either the application code or the hardware interface. Being able to generate or to receive traffic at line rate is very interesting for network-centered applications (firewalls, traffic monitoring, streaming servers, etc.). However, the networking code architectures of modern mainstream operating systems are the same today as they were almost 30 years ago. This means that they are still designed to run under the constraints from the past (i.e. low bandwidth and scarce memory resources), and are not really adapted to today's conditions (i.e. high bandwidth and plenty of memory).

Netmap uses a shared memory region (between the kernel and user space applications) containing buffers and descriptors allocated once when the network device is initialized. This eliminates buffer handling costs, as well as the need for copying data between user and kernel address space. Applications fill available buffers with packets to send, greatly decreasing the cost per packet of a system call that would otherwise be required to send each packet. As a result, rather than one system call per packet, we only need one call to send a group of packets. Another optimization used by netmap is to take advantage of the multiple buffer architectures of recent network interface controllers (NICs) by providing applications with the possibility to attach themselves to all available rings or to just one, which allows them to exploit the available parallelism in the system.

Applications that do a lot of raw packet I/O can benefit greatly from this API, with the maximum number of packets processed per seconds increased by an order of magnitude (and with these results scaling with the number of processors and their clock frequency). Moreover, only minimal changes are required to adapt the application, and these changes can be avoided by using a *libpcap*-compatible API on top of netmap.

Chapter 3

Existing solutions

This chapter gives an overview of the current smart-TV market (either directly integrated into the television itself or by an external device attached to the television set). This chapter describes the shortcomings of both types of solutions and identifies some of the pitfalls to avoid.

3.1 Market state

Studies show that the connected-TV market is on the verge of exploding [26]. For example, while less than 20% of UK households have an IP-capable TV today (in comparison to connected consoles' 45% and pay-TV devices' 38% share of available IP-capable devices), this share could grow to reach 55% by 2015, surpassing the two main contenders. This rapid increase in the fraction of TVs that are IP-capable devices is driven by the desire of hardware developers to increase the share of connected-TVs with respect to global TV sales, as well as the growing ecosystem of content providers (Video on Demand (VOD) providers, Replay TV services, multimedia sharing platforms, etc.).

At the moment, most platforms are not open to developers, and this has lead to a very small ecosystem of applications on these platforms. VisionMobile [27] categorized TV-related applications into three categories: TV-applications only (usually a simple gateway to a content provider who streams directly to the TV), Mobile-applications only (such as Zeebox [28] and GetGlue [29]) which are intended to be used in parallel with a TV service in order to enhance watching TV (by providing additional content linked to the currently watched program or shortcuts to social networks), and the last category – applications that close the gap between TV and mobile devices, i.e. that can send content to the television or even have control over the television. The most interesting and the least exploited category is the last of these three. Even though sending content from a mobile device to the TV is becoming common (and basically corresponds to using the TV as an external display for the device), controlling the TV (or the STB) from a mobile device is not yet common, at least without mimicking a classic remote control on the device's touchscreen while presenting additional content.

A survey based upon a study of more than a thousand technology stakeholders and critics has shown that 48% of them are pessimistic with respect to a possible breakthrough of smart systems integration inside the home: *"By 2020, most initiatives to embed IP-enabled devices in the home have failed due to difficulties in gaining consumer trust and because of the complexities in using new services"* [30]. This shows that providing advanced functionality is interesting and starting to happen, but this functionality is unusable in practice because of bad integration or horrible user experience. Such a poor experience is not beneficial for the company nor for their clients. A guide for choosing a set-top box (STB) insists on the importance of the *"Babysitter test"* [31], i.e. a person without previous experience with the system should be able to immediately utilize the simpler functions. The real challenge is not providing new and exciting features, but rather striking a balance between features, simplicity, and usability. The goal is allowing people who are not comfortable with technology to use the system, while giving access to more advanced features to other users.

3.2 Set-top boxes

Many different STBs are available on the market at the moment, which makes finding the desired combination of services, channels, and supported formats difficult, especially considering that most of these systems are closed to third-party applications. A STB offers an affordable solution, as the customer only needs to buy the box to enjoy its functionality, while re-using the same screen, sound system, and so on. However, this also comes at the price of an additional HDMI port to connect to the TV, but this is not really an issue in the general case, as current TV displays have several HDMI inputs available, and if all of them are occupied, it is possible to plug a HDMI switch into the display. However, this solution adds yet another remote control, and a layer of indirection between the users and their TV. This problem can be mitigated by using a "universal" remote control to replace all the remote controls by a single one. However, this is not really a practical solution, since either every device is still independently controlled by the universal remote (which is more or less logically equivalent to having one remote per device), or the remote control allows for more elaborate sequences of operations (e.g. a *watch a DVD* action which sequentially turns on the DVD player, the TV screen, and the sound system, and then starts the DVD playing), at the cost of a longer and more complicated set-up for the remote control.

3.3 Smart-TVs

The main problem with current smart-TVs (as pointed out by VisionMobile in [27]) is that even if the devices are connected, they are not really ready to be connected, as they lack applications, have a poor GUIs, and provide poor navigability. Moreover, these smart-TVs are often only controllable by complex remote controllers, or simpler, “classic” ones, which leads to a complex series of manipulation for unadapted situations (such as entering text into a textfield on a TV). This leads to a under-utilization of these functions in practice or to a bad user experience for those few users who try to take advantage of these functions.

However, a connected-TV could be a good centralized solution, as it can integrate disk readers, a hard drive, and a more powerful processor which in turns decreases the clutter of the typical media center. Integrating more devices also facilitates making them interact more directly with each other, and controlling them using a common controller in order to provide a unified feel.

Chapter 4

The television system

This section describes the television system and the underlying programming concepts used by this system. The main objective of this system is to be stable and reactive, so that user interactions (via the remote control) are processed efficiently in order to deliver a high-quality user experience. Section 4.1 describes the programming methodology for this system and gives an architectural overview of this system, section 4.2, section 4.3, and section 4.4 present the core subsystems of the television (respectively the window manager, the command manager, and the application manager), and section 4.5 presents some auxiliary subsystems that provide additional services which are not vital for the system itself.

4.1 Methodology

This section presents the programming environment and concepts used to develop the television system. Subsection 4.1.1 presents the programming environment in which the system was developed, subsection 4.1.2 introduces the global programming concepts that drove the prototype development, subsection 4.1.3 gives an in-depth explanation of message queues, which are the backbone of this prototype, and subsection 4.1.4 concludes this section with an architectural overview of the prototype.

4.1.1 Programming environment

The programming environment for this project utilizes one development machine and two testing machines. One of these testing machines is a virtual machine hosted by the development machine. The reasons for utilizing a virtual machine are three-fold. First, it allows us to abstract away a lot of material details, such as painful system configurations to support new programs, content, or other files;

it especially facilitates audio processing. Indeed on the physical machine, the sound is extracted from the HDMI signal, then sent to an amplifier, which was not operational at the beginning of the project. Second and most importantly, it is a lot more practical to work on only one machine at a time, especially given that the testing machine is built into a quite large case, as the case must be wide enough to allow for good sound emission from the three speakers situated behind the front of the case. Consequently this testing machine is not situated near the development machine. Moreover, this allows the programmer to test different methods of configuring the underlying Linux system until one works, and subsequently try only the latter configuration on the physical prototype (which mitigates problems, such as conflicts between software packages).

Programming is done on a a set of machines which are each connected to a network filesystem (NFS), so modifications to one of the modules are instantaneously available on the testing machines, after a modified module has been recompiled.

4.1.2 Modules, stubs, and iterative development

Given the size of the project, a modular approach is really well suited as it enables a programmer to rapidly develop a prototype that may be iteratively improved later on. Modern software development is based on managing the inherent difficulty of bigger and bigger projects by decomposing the system into a collection of more focused subsystems. The goal of this decomposition is to obtain subsystems (which are called *modules* in this thesis) that have a *high cohesion* with themselves and a *loose coupling* with others.

High cohesion means that a module performs a very focused task. Such a module is easier to understand and to reuse, as the code remains small and pertains only to a limited set of operations within the system. Loose coupling means that modules do not need to know much about the implementation of other modules in order to interact with them. Ideally, modules should be entirely oblivious to implementation details of other modules, and they should communicate with them only through their well-defined public API. This principle is known as *information hiding* or *encapsultion* in the object-oriented paradigm.

Both high cohesion and low coupling allow for a more efficient development process, as each module can be developed in isolation and thus be easier to understand, as it is focused on a limited task. Once an API is defined for a module (i.e. a definition of the possible interactions between this module and other modules of the system), this module can be iteratively modified, improved, tested and upgraded. As long as this module respects the relevant APIs, other modules do not have to be adapted – or even recompiled – in order to suit the new version of this first module. Parnas states in [32] that modules should be organized around hiding and encapsulating each difficult or likely to change design decision. Doing so ensures that the developer may easily modify those decisions – even late in the development cycle – without having to refactor each and every module involved in the system. This promotes flexibility of the system, which is paramount when developing a prototype.

Given the limited time frame of this thesis project, an iterative programming approach has been used. Each module (see section 4.1.4 for an architectural overview of the project) supports only a minimal subset of functionality at first, and later more and more use cases are added to it. As the goal of this thesis project is not to create a finished product but rather working a prototype, iterative development is particularly well-suited, as the first iteration allows the programmer to test an interaction model for one particular action, while other similar actions can be added later on, whether on a per-need basis or if time is available.

Stubs are extensively used while developing a module. Stubs are placeholder pieces of code that can output pre-determined values when they are called. Many simply exist without actually performing any kind of work, but will be the locus for adding functionality later. This allows the development and testing of a module regardless of the state of progress of other modules that the first module depends on or provides services for.

4.1.3 Message queues: Making loosely coupled modules

As different modules do not share a common virtual memory space, inter-process communication (IPC) techniques must be used to build loosely-coupled modules, such as system message queues. This is exactly the same mechanisms as for events, i.e. a module sends a message to a message queue that the intended recipient of the message will read from. The recipient fetches a message from the queue, possibly filtering based upon the message type (with UNIX System V queues, see section 4.1.3.1) or by message priority (with POSIX message queues, see section 4.1.3.2), and acts on this message. Once the message is acted upon, the recipient fetches the next message, if any is available in the queue.

The same performance criterion as for events applies for the treatment of messages. While a message is being acted upon by a module, other messages sent to this module cannot be processed. As a consequence, if the processing of a message takes some time, then the delivery and the processing of other messages in the queue are delayed, which might cause noticeable performance issues if some of the delayed messages are requests for time-critical actions. Another issue, which is more of a concern for messages than for events is the maximum number of messages that can be enqueued. Indeed, the system imposes limitations on the maximum number of messages that can be enqueued, or the maximum amount of memory that a queue can utilize. If such a limit is reached, the subsequent sending of a message to the queue would fail. In this case, either the message would have to be dropped or the sender would have to try to send it again later. As a consequence, when using message queues, developers must ensure that the rate at which messages are processed from a queue remains greater – most of the time – than the rate at which messages are sent to this queue. On the other hand, no synchronization is necessary – and therefore fewer subtle bugs are introduced in the system – as modules only work on one task at a time.

4.1.3.1 UNIX Sytem V message queues: The *sys/msg.h* system library

UNIX System V (SysV) is one of the oldest commercial Unix OS. It was developed by AT&T in the early 1980s. However, SysV IPC facilities – semaphores, shared memory, and message queues – are still widely implemented in Unix systems today, even on POSIX systems. This is despite the fact that the POSIX committee has not standardized these techniques and that they have proposed alternative specifications for IPC.

As seen in Listing 1 SysV messages are structures composed of two elements: a message type (`mtype`), which is supposed to be a positive integer, and an array of one character. If the second member could only hold one character, it would be rather limited, however this declaration corresponds to a well known exploitation of pointer arithmetic, which is called a *flexible array member* in the C99 standard. If the last member of a structure is allocated more memory than explicitly stated in the structure's definition, this memory can be accessed for later uses, as long as the size of the second member is known at the time this memory is used. As a consequence, the second member of this structure can hold an arbitrary-sized array, or even a structure (which we refer to as the *contained structure*). This approach allows us to send an arbitrary amount of data associated with the message – as long as the total length of the message is less than the system-defined limit. Programs can filter messages they want to fetch from the queue, either by specifying a positive integer (which is the exclusive type of message they are interested in) or by specifying a negative integer (which means they are interested in messages whose type values are lower than the absolute specified value).

```c
struct msgbuf {
    long mtype;         /* message type */
    char mtext[1];      /* message text */
};
```

Listing 1: System V IPC messages

The second element of the structure must be suited for all possible kinds of messages. Indeed, the recipient (reader) of such a message needs to know the size of the contained structure in order to retrieve a message, and the message size must be known before the message type is. This is especially tricky if the recipient is interested in a range of message types which can have a *contained structure* other than a string with a pre-defined length. The contained structure must be "one size fits all" in this case. A simple solution to this problem is to define the contained structure as a structure containing one member for each message type. This member can be a primitive type, as well as a user-defined structure. The reader can now receive a message, knowing in advance the size of the contained structure, and once the reader knows the message type, it can access the relevant field in the contained structure, which is *a priori* the only one initialized by the message sender (the writer). Using this technique makes adding new message types easy, as we only need to add a field in the contained structure (if needed). The reader does not need any modification other than the addition of a handler for the new message type, as it can learn the size of the contained structure using the `sizeof` operator.

SysV message queues can be used as two-way communication channels between more than two modules. However, having two readers interested in the same message type leads to an undefined behavior concerning which of the modules will actually receive a given message. As a consequence, message queues are easier to use if only one process reads from a sysV queue or if it does not matter which reader reads from the sysV message queue.

A client-server approach can be used with several processes reading from the same sysV message queue(s). In the simplest approach one process acts as the server for a message queue, filtering messages with a range of message types. Other processes send requests with these message types, and the sending of an optional response is possible via the same queue using a specific message type, defined in the protocol (or possibly in the request). In this case, the requesting process can filter the message queue to pick out only this response, without diverting messages that were not intended for it.

Another way of using a sysV message queue with more than one reader is having each reader only interested in messages whose types are equal to the reader's process ID (PID). This approach is used in the sysV message queue between the application manager and the applications (see section 5.3). In order for the application manager to have an associated message type independent from its PID, it would be possible to have it interested in messages whose type value is 1. Indeed we can be sure that no application will ever be interested in this message type, as only the *init* process can have a PID equal to 1 in a POSIX system. However, due to the limitation discussed below, the application manager does not use the sysV message queue to receive messages from applications.

The possibility to filter messages by their type is the main advantage of the sysV message queues. However, these queues come with an important drawback as well. Within the code, message queues are not file descriptors (even if they are similar in form), nor do they provide any asynchronous notification. As a consequence processes cannot simply react to messages coming from several sysV queues, or from a sysV message queue and a file descriptor. In this case, such a process would have to periodically poll the sysV queues, or dedicate a thread blocking on each sysV queues and one blocking on the file descriptors (using the select function described in section 4.3.1). Both these solutions are not really satisfying, as the first one introduce an additional delay in the message processing, and the second one introduce the need for threading and synchronization within the process, which is best avoided if possible. As a result, sysV message queues are less suitable to processes that have to react to several message queues and file descriptors than their POSIX alternative (described in the next section).

4.1.3.2 POSIX message queues: The *mqueue.h* system library

In contrast with sysV message queues, POSIX message queues provide asynchronous notifications when a message is sent to a queue. A `sigevent_t` structure can be associated with a message queue. This structure is able to specify a signal to be sent or a callback function to be called when a message in posted in a queue. Even better, a POSIX message queue is implemented as a file descriptor in Linux systems, even if this is not specified in the POSIX standard. As a result, it is possible to monitor the POSIX message queues with I/O multiplexing system calls (such as the `select` system call, described in section 4.3.1).

POSIX message queues are priority-driven. Each message is associated with a priority, which can range from 0 (the highest priority) to `MQ_PRIO_MAX` (defined in *limit.h*). Messages with the highest priority are dequeued first, and messages with the same priority are retrieved in a FIFO fashion. However, messages do not have an associated type, which limits the usability of the POSIX message queues with several readers. The only realistic scenario in which several readers are waiting for messages on a POSIX queue would be a pool of workers waiting on messages from this queue and performing long duration processing on these messages. In this case, messages with the highest priorities in the queue would be processed by the first workers available.

For our project, POSIX queues are used as one of the possible entry points for each module. Each module is associated with one POSIX queue, so we do not have to concern ourselves with message destinations when several readers are involved.

4.1.4 Global architecture

The TV system is composed of three core modules: a window manager (**WM**, see section 4.2) controls the appearance, position, and size of the display of each running application, an application manager (**AM**, see section 4.4) manages the applications and their life-cycle, and a command manager (**CMD**, see section 4.3) handles the communications between the remote controls and the global system via various network interfaces. Figure 4.1 shows the relationships between these three core modules.

Each module can send a message to another module by using the latter module's POSIX entry message queue. The **WM** and **AM** do not need to communicate directly with each other. However, some indirect communications take place between these two modules, through applications and the X Window server: The **AM** launches an application, which requires the X Window server to map a window to the display screen, which in turn generates an event addressed to the **WM** (as explained in more depth in section 4.2). Finally, the application manager communicates with the applications using both POSIX and sysV message queues: the application manager sends messages to the applications via a sysV message queue shared between all applications, while the applications send messages to the application manger through the manager's POSIX entry message queue.

Finally, one auxiliary module is shown on Figure 4.1: the address discovery module. This module is responsible for communicating the TV's MAC and IP addresses to the remote controls. This module is described in section 4.5.1.

4.2 The window manager

A window manager is responsible for handling the appearance and behavior of a windowing system. It can for example draw a frame around an application's window, support drag and drop actions, specify the appearance of a window depending on whether it has the focus or not, and is able to perform various actions relevant to window management, such as minimizing and restoring windows or managing virtual desktops. A taxonomy of window managers has been given in [33]. Although this taxonomy is quite old it is still relevant today. Among window managers two famous examples are gdm and kdm, which are the default window managers for the desktop environments GNOME and KDE respectively.

The window manager for this project is quite simple in comparison with that of a traditional desktop environment. Subsection 4.2.1 presents the X Window System and some mechanisms used by window managers built on top of it, subsection 4.2.2 presents our own window manager, and subsection 4.2.3 lists

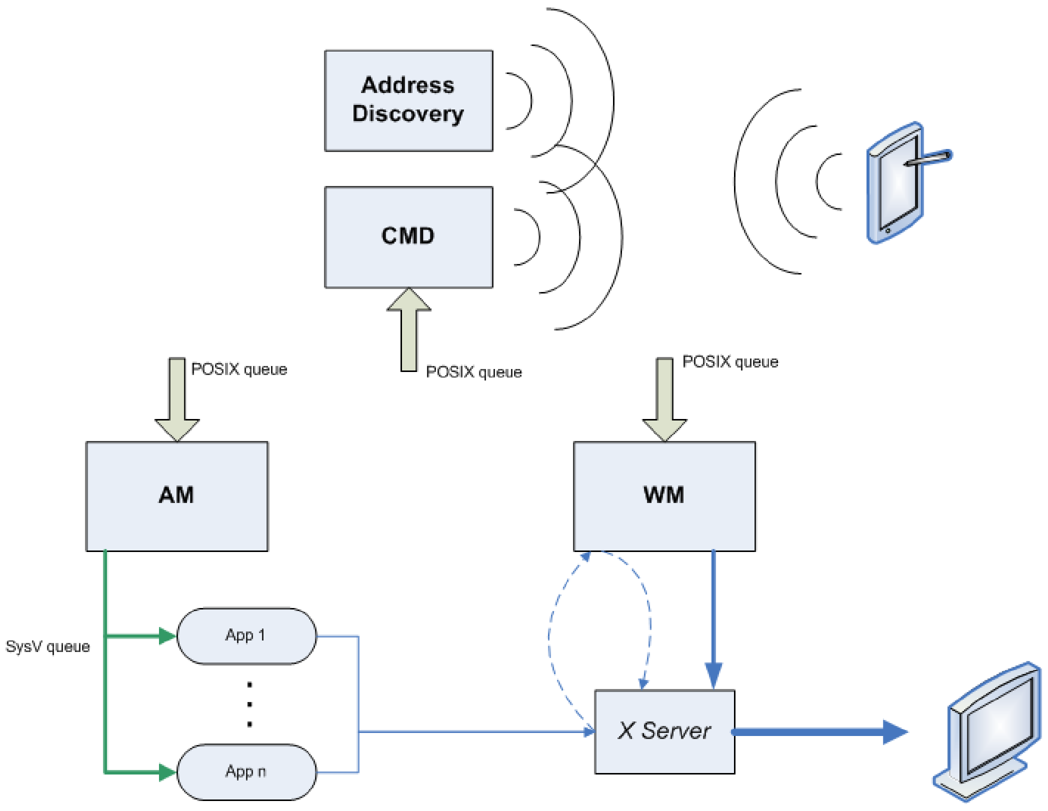


Figure 4.1: Global architecture of the television system

some further features to be added to the window manager in the future.

4.2.1 X window system and X window managers

The X window system [34] (commonly called “X11” as 11 is the current major version number) provides the GUI foundations for the majority of Unix-based operating systems (notable exceptions are Apple’s OS X and iOS, and Google’s Android). X11’s protocol is hardware-independant, so that applications that use it do not have to be recompiled to run in another setting. The protocol is based on a client-server model, designed for a network-transparent utilization. A program can use a remote X server’s display without having to run on top of the same operating system or architecture as the remote X11 server’s host.

A window manager program for the X environment is one of the clients of the X server. One of the X Window system’s design goals is to give programmers complete freedom in designing the user interface of a system, independently from the managed applications (giving the window manager the right to move and resize applications’ displays without asking for their permission), and from the

X server itself (by not imposing policies or pre-defined facilities, but by providing mechanisms – “hooks” – to realize basic operations instead).

The basic way to program a window manager using Xlib [35] – the low-level C library to interface with the X protocol – is to request the server to be notified when an application wants to map or reconfigure a window by selecting the `SubstructureRedirectMask`. If the `SubstructureRedirectMask` is selected by an application (i.e. a window manager), the request to display a window on the screen sent by an application to the X server is ignored by the server, which generates a `MapRequest` event. The action of the window manager upon receiving this event is entirely implementation-specific, a window manager could even discard every such request, preventing all applications from being displayed. If the window manager wishes the requesting application to be displayed, it sends a `XMapWindow` request to the X server directly. In the example presented in Figure 4.2, the window manager can move and/or resize the window before requesting the X server to map – i.e. display – it.

Similarly, an application that wishes to modify the position, the parameters, and/or the size of one of its windows must send a request to the X server, which in turn generates a `ResizeRequest`. In practice, this case is rarer, as facilities to do such modifications are often offered directly by the window manager (e.g. with a frame added around the application’s window), and not by the application itself. Only one application can select the `SubstructureRedirectMask` at a time, which prevents a conflict between two simultaneously running window managers.

Although the window manager is called a client in the X terminology, it acts more as a server, since every request sent by applications is acted upon by the window manager and not by the X server.

4.2.2 Our window manager

The television window manager is quite simple in comparison with window managers on traditional desktop systems. It is a tiling window manager, i.e. two windows never overlap each other (as opposed to stacking window managers for which part of a window can appear on top of another window). Moreover, windows are assigned pre-defined places and sizes depending on the current screen mode and the number of currently displayed applications.

The window manager was developed in C, directly calling Xlib. The first version of the window manager has two screen modes: the *full-screen* mode (**FS**), in which only one application is displayed, filling the entire screen, and the *big-quarters* mode **BQ**, in which four slots are available for applications (see Figure 4.3).

Applications that have mapped a display onto the screen (“*clients*”) are stored in a linked list, and four pointers are used to point to some of them (the quarter

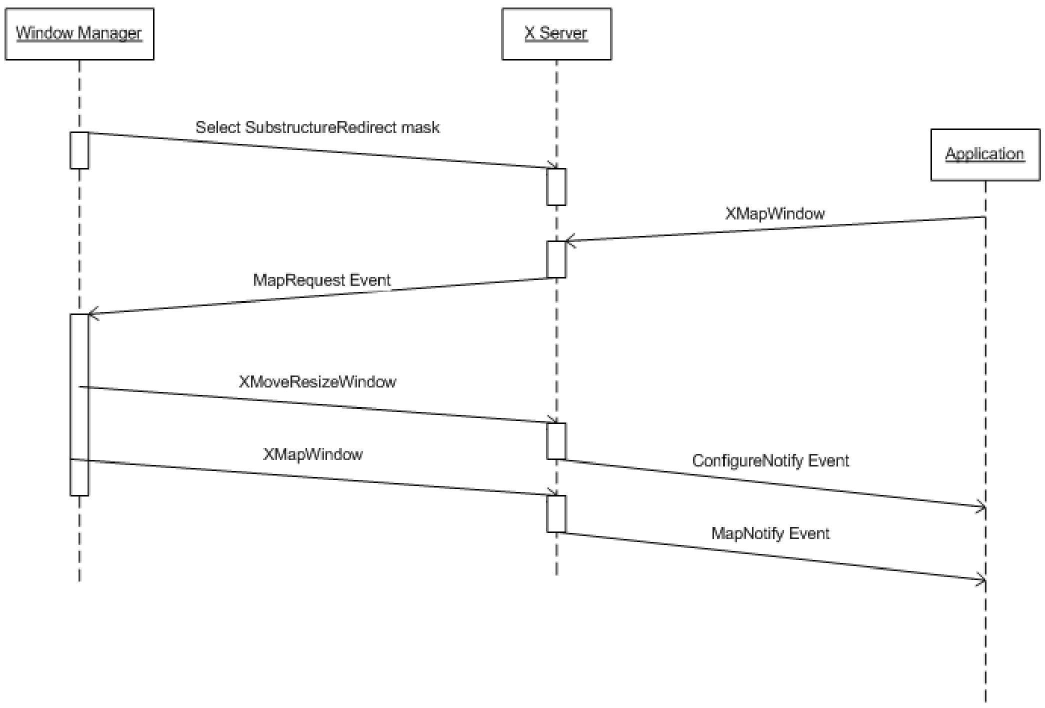


Figure 4.2: Basic mechanism of a window manager running on top of a X Server

clients, which appear on a quarter slot while the TV is in **BQ** mode). It is possible to have more clients than available display slots. Changing the placement of an application in **BQ** mode consists only in swapping its current pointer with the target application's pointer.

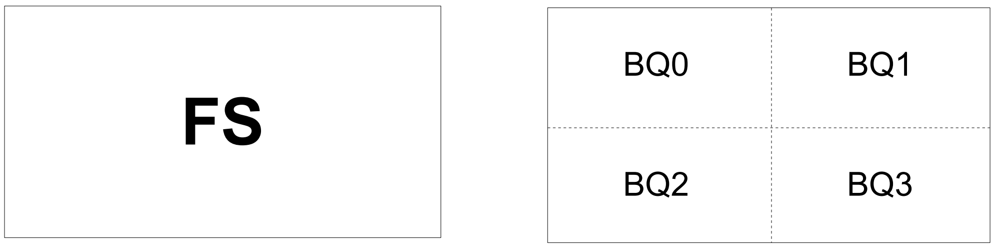
One of the quarter windows is the *focused* window. The *focused* window has two purposes: It is kept on screen when switching from **BQ** mode to **FS** mode, and this is the window currently controlled from the corresponding application's control interface on the remote control (see section 6.2.1). In **FS** mode, the window appearing on the screen is automatically the *focused* window.

4.2.3 Remaining work

The following paragraphs describe some of the work that remains to be done for the WM.

4.2.3.1 Double buffering

If pixels are drawn directly via Xlib (as it is the case when drawing part of the current user interface), it is necessary to take precautions when changing them. A practical approach to changing the picture drawn by the pixels consists of filling

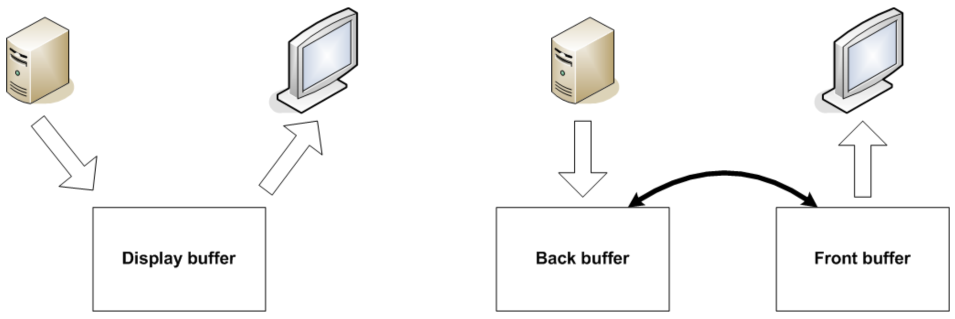


(a) Full-screen mode (b) Big-quarters mode

Figure 4.3: Screen types available in the window manager

the display buffer with black, onto which the new picture is drawn. However, this method is prone to *flickering*, which is caused by pixels that appear successively colored, then black, then colored to the user. This is likely to occur, since the screen might display the buffer while it is blackened. The displayed screen is the output of the contents of a display buffer and is typically updated at 50, 60, or 120 Hz.

A simple solution to this problem is to use *double buffering*. Instead of drawing directly on the front buffer (i.e. the buffer that is being displayed), the program draws on a back buffer. Once the back buffer is ready, the front and the back buffers are swapped, i.e. the front buffer becomes the back buffer and vice-versa (see Figure 4.4). Note that this operation does not involve data copying, and is therefore "cheap".



(a) Without double buffering (b) With double buffering

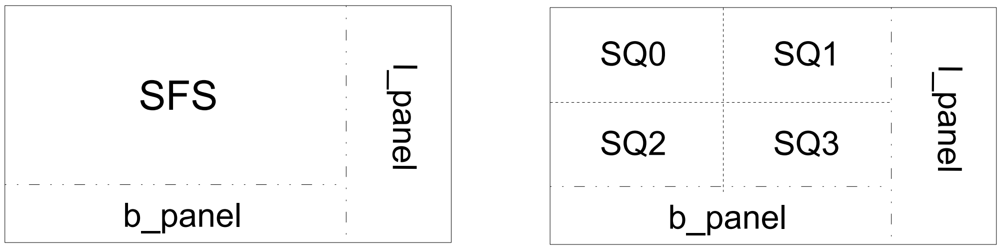
Figure 4.4: Double buffering

This solution has one major drawback: The swapping could occur while the front buffer is being sent to the monitor. In that case, another artifact called *tearing* becomes visible, that occurs when part of the screen represents data from the old buffer, while the rest of the screen represents the date of the new buffer. However, the user interface is relatively static (with respect to games with a high framerate), so double buffering could be sufficient for our needs.

Additional techniques are of course available to handle the tearing problem. It is possible to synchronize the buffer swapping with the screen refresh (a technique called *vertical synchronization* or *vsync*). This prevents tearing but may halve the refresh rate, since if the next buffer to be displayed is not entirely ready when the monitor refreshes, the previous buffer remains displayed and will be replaced only during the next screen refresh. Another possibility is using *triple buffering* or *multiple buffering*, which consists in using two or more back buffers and one front buffer, and swapping the front buffer with the most recently completed back buffer when the monitor is ready for a new refresh cycle. This prevents tearing, while not limiting the internal framerate of a program. In this configuration, a program can always write to a buffer without having to wait. Finally, an entirely different solution is called *frameless rendering* [36], which immediately replaces a random pixel on the screen with its most recently updated value.

4.2.3.2 Additional screen modes: small quarters and small full-screen

Two additional screen modes are to be added in a later version of the window manager: the *small-quarters* screen mode (**SQ**) and small full-screen (**SFS**). These additional mode are very similar to **BQ** and **FS** respectively, in that the same slots are available, albeit smaller, and there are two additional panels on the right: *l_panel* ("lateral panel"), and on the bottom of the screen: *b_panel* (see Figure 4.5).



(a) Small full-screen mode (b) Small-quarters mode

Figure 4.5: Additional screen types for the window manager

4.3 The command manager

The CM is the module which provides a gateway between a remote control – be it the dedicated physical remote controller or a mobile application (see Chapter 6) – and the system. This module provides an abstraction over the discrepancies between the communication channels used by the different remote controls (i.e. over TCP sockets for the mobile applications and via a dedicated receiver for the dedicated remote control). This module takes care of the technical details needed to communicate with the remote controls, leaving the rest of the system free to interact with them in a higher-level fashion, independently of the kind of remote control being used. Moreover, this allows for a better flexibility: Only the CM needs to be modified when a new communication technique is necessary or more convenient, while the other modules can remain unchanged.

The CM also handles the scenario in which several remote controls are detected (for example, if a user switches from using the dedicated remote control to using the remote control application running on his or her mobile device). As only one remote control may operate at a given time – at least for the moment – the CM closes the connection with the former remote control and maintains a connection with the new remote control. The remote control is responsible for synchronizing itself with the TV's state.

This section presents the *select* system call (subsection 4.3.1), describes the operations performed by the CM (subsection 4.3.2), and presents several modifications that would be desirable in a future version of the CM (subsection 4.3.3).

4.3.1 Multiplexing inputs: the select() function

In an event-driven system (and more specifically in a message-driven one), a process needs to wait for an event to arrive and then perform a task on it. This process could periodically check for the existence of an outstanding event, waiting for a given amount of time between checks if there is no outstanding event. This method – called "busy wait" – is very simple, but it is highly inefficient. Indeed, such a process consumes a lot of unnecessary CPU time simply by waking up to see that no work has to be done and yielding the CPU.

A better solution is for processes to block while waiting for an event, and not wake up until an event is there to be processed. The *select* and *poll* system calls are both designed to monitor a given number of file descriptors, blocking the process until at least one of them is ready to be used without blocking. The select call has been chosen for this project, as both functions are very similar in interface and performance.

Listing 2 is an excerpt from the select function's manual page. The four first functions are tools for manipulating file descriptor sets (i.e. the `fd_set` structures). They allow the programmer to remove a file descriptor from a set (`FD_CLR`), check if a file descriptor is in a set (`FD_ISSET`), add a file descriptor to a set (`FD_SET`), and empty a set (`FD_ZERO`).

The select function itself takes three file descriptor sets as parameters, whose purpose is two-fold. First, these sets specify which file descriptors must be

```
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);

int select(int n,
           fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

Listing 2: Excerpt from the select function's manual page

monitored by the select function to determine when these file descriptors have data available (for descriptors in the `readfds` set), can be written to without blocking (`writefds` set), and/or when an exceptional condition (e.g. out-of-band data arrives on a socket) occurs on a descriptor (`exceptfds` set). Second, when the select function returns, these parameters hold the file descriptors that can be read from or written to without blocking (*a priori*). As a consequence, the file descriptors that were inside the sets before the function call do not necessary remain in those sets, and must be kept in separate sets if they need to be used again (e.g. if the select call is part of an infinite event loop). The first parameter must be set to the highest-numbered file descriptor present in the three sets plus one, and the last parameter allows the definition of an optional timeout for the function (i.e. the delay after which the function will return even if no file descriptor may be used without blocking).

The select call returns either a strictly positive integer, that corresponds to the number of file descriptors left in the three sets together, 0 if the timeout delay has expired, or -1 in case an error has occurred. In the latter case, as for a great number of system calls, the special variable errno is set with an integer corresponding to the error type that has occurred.

Once the select call returns, the process is awakened, and if the returned value is strictly positive, it can uses the `FD_ISSET` for each of the file descriptors originally in the sets to determine which one(s) can be used without blocking. However, some unusual circumstances may cause a file descriptor to be marked as ready to be used without blocking by the select function whereas this is not the case. As a consequence it is best to set the `O_NONBLOCK` option for the file descriptors, preventing the process from performing operations that would block on the file descriptor, and to be prepared to handle a possible `EWOULDBLOCK` exception that would be raised in such a case.

Both poll and select have performance issues when used with large sets of file descriptors, as they operate in O(n) time with respect to the number of file descriptors. The *epoll* function has been implemented in Linux in reaction to this problem, while the *kqueue* function [37] has been introduced to FreeBSD for the same reason, as well as to provide an additional support for other kinds of notifications, such as signals and asynchronous I/O events. Both these functions execute in O(1) time with respect to the number of file descriptors, and as a consequence are much more scalable than poll and select. However, poll and select perform as well as epoll – or even slightly better – when the number of file descriptors is low (as reported in [38]).

4.3.2 Operations of the command manager

Listing 3 shows the skeleton of the CM. This is a typical example of using the select function in order to multiplex I/O entries in a process. After the data structures and message queues have been initialized, the process enters an infinite while loop blocking on the select call until a file descriptor contains some data (i.e. the file descriptor can be read from without blocking). Two file descriptors are used for this loop. The `inset` is used as the select function set of the file descriptor which we are interested in reading from (as we are not interested in writing without blocking, or in possible exceptions from the file descriptors, the corresponding two other sets are set to NULL). The `watchset` keeps track of the file descriptors, and is used to reinitialize `inset` before each call to select.

The CM module has three possible entry points. Either a message arrives on the module POSIX entry queue, the server socket is ready to accept a new connection, or there is some data available on the client socket. Both the server socket and the POSIX message queue are initialized and have their file descriptors added to the watched set before the infinite loop, and the client socket is added to this set after the server socket has successfully accepted a connection from a remote control.

Once the select function returns, we know that at least one of these file descriptor can be read from without blocking, as no timeout is set for the select function. As a consequence, we check whether each file descriptor is in `inset`. If the client socket has available data, we read this data from the socket with the `recv` function and transmit the request to the appropriate module by sending it to their POSIX entry queue (see Table 6.2 for the possible requests). If there is an outstanding message on the POSIX queue, we prefix it with the length of the message and send it to the remote control – if one is connected – over the client socket. As a consequence, it is not possible for other modules to send messages for the CM, as all messages are transmitted to the remote control. However, if we ever need to do so, we could use a dedicated priority level for such messages, which the CM could examine to see if this is a message to transmit or a request for itself. Finally, if the server socket is marked as ready, there is an outstanding client connection and we can call `accept` on the socket without blocking to obtain

```
FD_ZERO(&watchset);
FD_SET(queueFD, &watchset);
FD_SET(serverSocket, &watchset);
while(1) {
        inset = watchset;

        rv = select(maxFD + 1, &inset, NULL, NULL, NULL);

        /* Client socket has available data */
        if(FD_ISSET(clientSocket, &inset)) {
                /* ... */
        }

        /* POSIX entry queue has incoming message */
        if(FD_ISSET(queueFD, &inset)) {
                /* ... */
        }

        /* Outstanding client connection to accept */
        if(FD_ISSET(serverSocket, &inset)) {
                /* ... */
        }
}
```

Listing 3: Skeleton of the command manager

a client socket associated with the remote control. This socket is then added to the watched set, after closing and removing a previous client socket if needed.

4.3.3 Remaining work

The following paragraphs describe some of the work that remains to be done for the CM.

4.3.3.1 Include support for a dedicated remote control

A few modifications will be necessary in order to support additional types of communication, such as communication with the included remote control (see section 6.1). Additional file descriptors will have to be added to the select call, and additional functions will have to be implemented if those file descriptors are ready to be used in a non-blocking fashion. However, the same methods will be used to transmit remote control messages to the correct modules and to receive messages to transmit to the remote controls, so that code can be shared for all kinds of remote control.

4.3.3.2 Adding size information on TCP packets

A TCP connection is not message-oriented but rather is stream-oriented. This means that data accumulates on the socket until the socket is read from, even if there were data from a previous request already available. Currently we assume that when data is available on the socket, that this data fits in a 1024 bytes buffer, and that this data corresponds to one request. However, if two messages are sent almost simultaneously from the remote control to the TV, it may happen that these two requests are aggregated together in the TCP stack, and therefore are received as one message when the CM fetches the data from the TCP stack. A simple solution for this problem is to prefix each message from the remote control with the length of the message (using a pre-determined number of bytes), read the length of the message, and then read the appropriate number of bytes from the TCP stack to actually receive the message. This method is used for messages from the TV to the remote control, as the TV needs to send messages than can be bigger than 1024 bytes, such as application logos.

4.4 The application manager

The Application Manager (AM) manages applications running on the television (applications themselves are described in Chapter 5). This module was developed in C++, unlike the WM and the CM, as the object-oriented paradigm is only well-suited in the context of the AM.

This module is responsible of launching applications, keeping track of the applications and transmitting them request from the remote controller while they are running, and terminating applications on demand. Figure 4.6 is a class diagram for this module. the `AppManager` class has two possible entry points: The AM module's POSIX entry queue, and a file descriptor through which the `SIGCHLD` signals are received (as described in section 4.4.2.3). This object transmits requests from the remote control to the applications based on the applications' PID and manages the `RunningApp` objects. These objects are responsible for keeping a synchronized view of the running application's remote control interface (see section 5.1). Finally, the `AppRepository` is responsible for locating the applications' supporting files (e.g. executable file and application icon) based on their name.

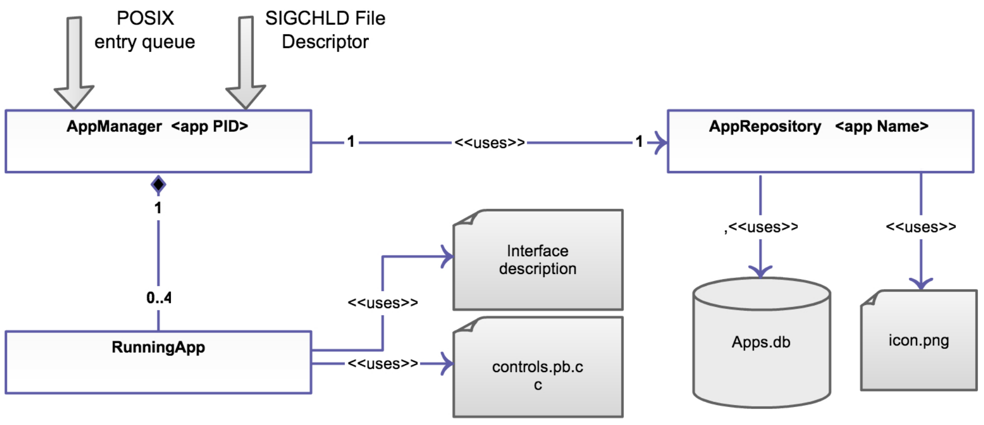


Figure 4.6: Class diagram for the Application Manager

4.4.1 Storing information about applications

SQLite – a self-contained and serverless SQL database engine [39] – is used to store the required information about applications (such as their name, an icon, and the path to their executable). SQLite is designed to be simple to administer, operate, and embed in a large program. As a consequence of these design goals, it is a small and fast SQL solution. It is a better solution than *ad-hoc* data files (i.e. files containing a record on each line, with a user-defined format, and directly manipulated with the `fopen()`, `fread()`, and `fwrite()` functions), as SQLite provides the expressiveness of SQL queries and Atomicity, Consistency, Isolation, Durability (ACID) guarantees of a SQL database engine. On the other hand, it is better than client-server SQL engines (such as MySQL) for our purpose, because there is no need for scalability with respect to either the number of records (as the number of installed applications will be small with respect to the database's capabilities) or the number of concurrent accessors (as only the `AppRepository` should need to write to this database).

Even if we could avoid using an application database for the moment, since all the information stored about an applications (e.g. its name and paths to its main method and icon) can be found within the file system. However, using a database from the start will make extending the application manager easier, especially if we need to store new information (such as a list of authorized system actions for each application, as is done in the mobile application markets). Moreover, it is quicker to perform a simple `SELECT` request on a database than to scan the filesystem to reconstitute the same information every time we need to access it.

4.4.2 Life cycle of applications

This section details the life cycle of applications, from the moment they are launched (subsection 4.4.2.1) to the moment they are exited (subsection 4.4.2.2) and the moment they disappear from the system (subsection 4.4.2.3).

4.4.2.1 Launching an application

Each application runs in an independent process, which is a child process of the AM. When the AM is asked to launch an application (with the *launchApp:app_name* request, see section 6.2.3.1) it *forks* and *executes* the application in the resulting child process. The AM then sends the PID of the application to the remote control, in order for the remote control to be able to use this PID when sending commands to the application. Using the application PID instead of its name allows the AM to run several instances of the same application simultaneously. If this behavior should not be possible for some application, then the AM could enforce a limit of at most one running instance for such an application.

4.4.2.2 Closing an application

When an application needs to be closed, it is sent a `appWillExit` command. The application can perform customized tasks upon receiving this message (such as saving its state). Ideally, these tasks should not take long, and the application should exit by itself after. If an application is still running after a pre-determined delay, i.e. if the AM has not been notified of the application process' closing, then it sends a `SIGKILL` signal to the application process. This signal cannot be blocked or ignored by the process, and causes the process to terminate immediately.

This method has a disadvantage: If an application does not need to perform any action before exiting, it should not have to handle the `appWillExit` command. As a consequence, an default behavior upon receiving this command (i.e. immediate termination) should be provided for all applications. If the application does not overwrite this behavior, the AM does not have to wait for the closing delay nor will it need to send a `SIGKILL` signal.

4.4.2.3 Reaping zombie processes

When a child process exits, a `SIGCHLD` signal is sent to the parent process. Since all child processes have some data to deliver to their parent process (such as their exit status), they linger in the system until the parent process reads this information using one of the system calls of the *wait* family. As the `SIGCHLD` signal is ignored by default, a child process could remain in the system until its parent process itself has exited. In this case, the child process is called a “zombie process”, as it does not accomplish any work, but remains in the system without dying. When the parent process exits, the zombie process becomes orphaned and is adopted by the init process. As init calls wait periodically, the zombie process will disappear from the system shortly after having been adopted. Zombie processes are not really a threat for the system if their parent processes are relatively short-lived. However, the AM is very long-lived, as it starts and stops with the system itself. As a consequence, zombie processes could accumulate and clutter the process table, possibly resulting in the impossibility to spawn new processes.

Two POSIX methods can be used to avoid this accumulation of zombie processes (by "reaping" them). First, we can handle the `SIGCHLD` signal in the parent process, by calling one of the wait functions in order to retrieve the zombie's exit value, letting it die. Second, the parent process can explicitly ignore the `SIGCHLD` signal, while specifying that it is not interested in its child's return values (by using the `sigaction` system call with the `SA_NOCLDWAIT` flag). Both these methods prevent the appearance of zombie processes in the system, but they also have some drawbacks. The first method imposes a limit of using only async-signal methods in the callback function for handling the signal, and may introduce synchronization problems worse than those encountered using threads. The second method prevents obtaining the exit status of the child processes or information about its resource usage.

Linux implementations provide a non-POSIX utility that is very useful for handling signals: `signalfd`. With this function, it is possible to be notified of the arrival of signals through a file descriptor. This allows a process to process the signal at a time of our choosing, rather than having to do it asynchronously at the signal's arrival. The file descriptor obtained with this function is added to the process's main select, and the signal can be treated as another entry point of the module, without any synchronization issue. If several child processes exit before the `SIGCHLD` signal notification is removed from the file descriptor, the notifications do not stack. As a consequence, it is impossible for the parent process to know how many `SIGCHLD` signals were received. The parent process must call the wait function in a loop until there is no more pending exited children (i.e. the wait function returns 0, if there is no longer a pending exited child, or fails with errno set to `ECHILD` if the process does not have any more children). When reaping the application processes in the AM, we use the `wait3` function, which is not defined in the POSIX standards, but comes from the BSD world. This function allows the retrieval of information concerning the resource consumption of a child process (e.g. the CPU time used or the number of IPC messages sent/received). This information could be used later to compute statistics or to detect possible problems with applications.

4.5 Auxiliary modules

This section presents auxiliary modules, which provide services which are non-essential for the system itself. Subsection 4.5.1 presents the address discovery module, responsible for automatically setting up a connection with the remote controller, and subsection 4.5.2 introduces watcher modules.

4.5.1 Address discovery module and Wake-On-Lan

The TV setup must remain as simple as possible, and we must ensure that no unnecessary configuration steps are needed. As a result we cannot assume a fixed IP address to the TV and hardcode this address to the remote control mobile application, as this address might conflict with another equipment that is already using this address or with an address assigned by a DHCP server (which is typically present in a home LAN).

IP-multicast over UDP is used to discover the TV's addresses (its IP address and its MAC address). The TV's MAC address can be used to switch the TV on, as described in the last paragraph of this section. Multicasting a UDP datagram to a multicast group means sending this datagram to the address of the group. In order to receive packets from a multicast group, potential receivers must *join* this group, i.e. they must inform the network of their interest in this group. Upon sending a packet to a multicast address, this packet will be forwarded to all interested parties. However, no guarantee is provided concerning correct delivery of such a packet to all potential receivers, as UDP is an unreliable transport protocol.

A multicast address is is the `224/4` address range (i.e. addresses ranging from `224.0.0.0` to `239.255.255.255`). A guideline for address assignment in this range is found in RFC 5771 [40]. Following this guideline, we should use an address in the Administratively Scoped Block (`239/8`, i.e. addresses ranging from `239.0.0.0` to `239.255.255.255`), reserved for addresses which are for *"local use within a domain"*. Within this scope, a guideline for address assignment is given in RFC 2365 [41]. This RFC defines the `239.255/16` range (from `239.255.0.0` to `239.255.255.255`) as the IPv4 Local Scope. The multicast address `239.255.123.124` and port `12345` has been arbitrarily chosen within this scope as the multicast group address used for the discovery of the TV's addresses.

In the context of address discovery, we are not using multicasting to support one-to-many or many-to-many communications, but rather as an IP address that can be used independently of the underlying LAN topology and constraints. When the remote control mobile application needs to discover the "true" addresses of the TV, it sends a pre-defined string to the address discovery group address. If the television is switched on, then the TV's address discovery module is running and it will have joined this multicast group. If no nothing happens to the UDP datagrams, this module receives the string and replies to it (using the remote control's IP address contained in conjunction with the received UDP datagram). The response of the TV only needs to contain the TV's MAC address, as the IP address of the TV is contained in the response UDP datagram.

Wake-On-Lan (WOL) is an ethernet standard allowing computers to be switched on remotely by network messages. It consists of broadcasting a "magic packet" over a local network. This magic packet is composed of the hexadecimal value for 255 repeated six times (i.e. `FF FF FF FF FF FF`) followed by sixteen repetitions of the MAC address of the target computer. The magic packet can be sent over any network and transport protocol, but we follow the typical practice by sending it in a UDP datagram to the UDP destination port 7 (the *Echo* port). The NIC on the target computer is able to recognize this packet, even if the computer itself is switched off or sleeping, and the controller triggers the awakening of the computer. WOL is also possible over Wi-Fi (Wake on Wireless LAN), but few wireless NICs offer this functionality.

4.5.2 Watcher modules

In order to monitor the state of the system, watcher modules can be introduced, to check if the core systems are still reactive. One such watcher module was developed to monitor the number of messages on the POSIX message queues. As described in section 4.1.3, there is a limit on the number of messages that can be enqueued on a message queue, and if this limit is reached messages will be lost, which cannot be allowed to happen. However, neither the POSIX queues or the sysV queues provide any facility to be notified of a message's arrival (the POSIX queues do provide a notification when a message arrives on an empty queue though). Moreover, a watcher module cannot block on a message queue waiting for a message to arrive without consuming this message, so this is not suitable. As a consequence, it is not possible to closely monitor the number of messages on a message queue without heavily intruding in the code of the core modules using these queues. So this watcher module can only monitor the number of messages on the POSIX queues by periodically fetching this number (the delay has been set at 1 second in the prototype). The `mq_getattr` function is used for this purpose. The maximum number of messages found in a queue is also logged for evaluation purposes (see Chapter 7). This cannot provide a fine-grained evaluation of the behavior of messages on a queue (such as the rate of arrival or the average latency between a message arrival ad its consumption), since we expect messages to be processed in a time much shorter than 1 second. However, this is sufficient to detect serious problems in a core module: If a POSIX entry queue holds more than a given number of messages at any given time, we can assume that the corresponding module has crashed or is blocked while processing a message.

In either case we should investigate how to resolve the situation.

Chapter 5

Applications

This chapter describe the television applications. These applications are composed of two parts: the actual application, running on the television, and the application interface, which is displayed on the remote control. The application interfaces are described in section 5.1, the application themselves are described in section 5.2, and the method of communication between an application's interface and the application is presented is section 5.3. Finally, section 5.4 presents an example of such an application and section 5.5 describe some of the work that remains to be done for the applications.

5.1 Application interfaces

The main criterion that differentiates our prototype from the current smart-TV market is the way users interact with the TV, and especially with the applications running on it. A TV set cannot offer the same interactions with application as a computer or a mobile phone, since we lack the ability to directly interact with objects on screen, be it with a mouse or with a touchscreen. However, we can provide richer interactions than just using four directional arrows and a "OK" button. With the recent rise of touch screens on mobile devices, we believe that it is possible to provide a user interface that is more direct, adapts to what is shown on screen, and is already familiar to users (especially if they are interacting with the TV from their own mobile device).

Together with the actual application, the developer must describe the application's interface. An application interface consists of at least one *page*. A page fills much of the control panel on the remote control (see section 6.2.1). If several pages constitute the application's interface, the developer can request any of these pages to be displayed (pages are associated with a name in the interface's description), directly from the application's code. Such a request is transmitted to the AM, which can update the state of the application before transmitting the request to the CM which in turn sends the request to the remote control. Within a page can be different kinds of objects: button, textfields, labels, sliders, at most one list, and at most one one-dimensional or one two-dimensional swipe zone. The description tool allows the developer to customize each object (e.g. define its name, position, size, label, etc.).

The interface must be described in a XML file for the moment (an example of such an interface description can be found in Listing 5 in section 5.4). A graphical editor will later be added to the SDK. Such an editor would make the creation of interfaces easier, as it would allow dragging and dropping objects to their desired position. This editor would also provide immediate feedback to the developer on what the interface really looks like, without having to send it to the remote control. Another improvement to make with regard to the interface tool is to support different screen sizes. In the current implementation, the position and size of elements are limited to absolute pixel-per-pixel measures, which prevents designing interfaces that can look their best on a range of screen sizes. In order to provide such support, layout systems are likely to be added to a subsequent version of the SDK.

In order to minimize the storage space used by the interface description as well the required time to sent this file over the network to the remote control, we convert the XML document to a binary format using a python script. Google's protocol buffers [42] are used for this purpose. In order to use these protocol buffers, we must describe the structure of the data in a `.proto` file. In such a file, we describe the messages we will be sending with the protocol. Messages can contain repeated, optional, or required fields, which in turn can be primitive data, other messages, or enumerations. This `.proto` file can then be processed by the `protoc` tool in order to produce classes for C++, Java, or Python. These classes are used to create, access, serialize, and parse messages. As protocol buffer messages are serialized in binary form, storing interfaces as such messages requires less memory than storing them using their XML form, and therefore they take less time to transmit to the remote control. Moreover, they are also a lot faster to parse compared to an XML file.

5.2 Programming applications

In order to interest third-party programmers in developing applications for our system, it must be easy for them to do so. As a consequence, they must be provided with efficient tools, as well as a familiar developing environment. As a consequence, we have chosen to base our applications on the Qt framework [43], with applications themselves being written in C++. The Qt framework consists of a lot of modules, the two most important being *QtCore*, which provides the core non-GUI classes, and *QtGui*, which contains most of the GUI components. *Phonon* [44] was used in addition to these two modules to provide multimedia support. Phonon can use different backend multimedia frameworks such as VLC [45], which is used by the TV for the moment. Qt's main objective is to offer an entirely cross-platform development framework, and as a consequence it provides a lot of tools to perform various tasks independently from the underlying system (e.g. parsing XML documents, doing network communication, and accessing SQL databases).

A central feature of Qt is the signals and slots mechanism, which corresponds to a framework-supported *Observer* design pattern. With this design pattern, observers are automatically notified when objects they observe are modified. This is one of the major design pattern for GUI programming, since ideally the object that handles the appearance is separated from the object that handles the data. However, if the data is modified, the appearance might have to be changed as well. Each class that inherits from `QObject` either directly or indirectly can define signals and slots. Slots are declared as normal functions, and might be used as such. Signals are declared as functions with a void return type. Both are declared in the C++ header, after the slot modifier and the signal modifier, respectively. Signals cannot be used as normal functions, but they can be *emitted* by their defining class. When a class emits a signal, all slots connected to it are called (in the order in which they have been connected to the signal), and if these slots require parameters, the parameters of the signal being emitted are passed to them. When the last connected slot returns, the emit call returns as well. The `QObject::connect` static method is used to connect a signal to a slot, if possible (i.e. if the parameters of the signal and those of the slot are compatible). This ensures a compilation-time type check, as opposed to a run-time check. On the downside, using slots and signals in a class requires the class to be pre-compiled in order to create auxiliary *moc* files, supporting this mechanism. After this step, commands which are not defined in C++ (such as the slot and signal modifiers in the header) are hidden by the pre-processor in order to enable compilation by a normal C++ compiler. Qt provide a tool to auto-generate makefiles for Qt projects, which makes these additional steps easier to perform.

5.3 Transmitting commands to applications

In order to provide third-party developers with a familiar environment, every GUI component of the application interface corresponds to a `QObject` which emits a signal when the user has used the component. However, this is quite different from "normal" Qt application development, since in this case the developer would place the GUI components directly into the application code, and these components would be handled by the Qt framework itself. In our case, the components are handled by the remote control after they have been described in an independent interface description. The remote control cannot interact directly with the Qt framework for the application, since it has to communicate with the command manager. Table 5.1 shows a list of possible commands sent by the remote control in response to a user interaction with the control panel of an application. The *Sender* column of this table is not part of the command itself. A command from the remote controller is composed of two to three fields separated by a pipe character.

Table 5.1: Inputs of the SDK

Sender	Object type	|	Object Name	|	Optional parameter
Remote control	button		button name		
	text		textfield name		new text
	slider	|	slider name	|	new value
	listLabel		page name		selected label
	swipe		page name		direction
Poam	appWillExit				

These commands are inserted into the *command* field of a command request, containing the PID of the target application (see Table 6.2 on page 62 for the full command request). The last line of Table 5.1 (the command for terminating an application) does not result from a direct interaction with the remote control. The remote control sends the corresponding request to the AM (with the *killApp:app_pid* command), and the AM transmits the *appWillExit* command to the application. This is necessary, because the AM does not examine command requests before transmitting them to the applications (if they exist in the system). However, the AM must be involved in order to terminate an application (as described in section 4.4.2.2). If the corresponding request came as a command, then the AM would have to parse all commands before transmitting them, which would add some parsing cost to each application request sent by the remote control. Moreover, this would be conceptually wrong, since application request are only intended for applications, while the request for terminating an application is intended for the AM, which then becomes responsible for actually terminating the application.

Applications for the TV use a shared library that handles the interaction with the system modules. This library defines the objects corresponding to GUI component (such as the `Button` object, which emits a *clicked* signal when the user has tapped it on the remote controller), a `PopApp` object, and a `Controller` object, which receives the commands transmitted by the AM. A class diagram for this library is provided in Figure 5.1.

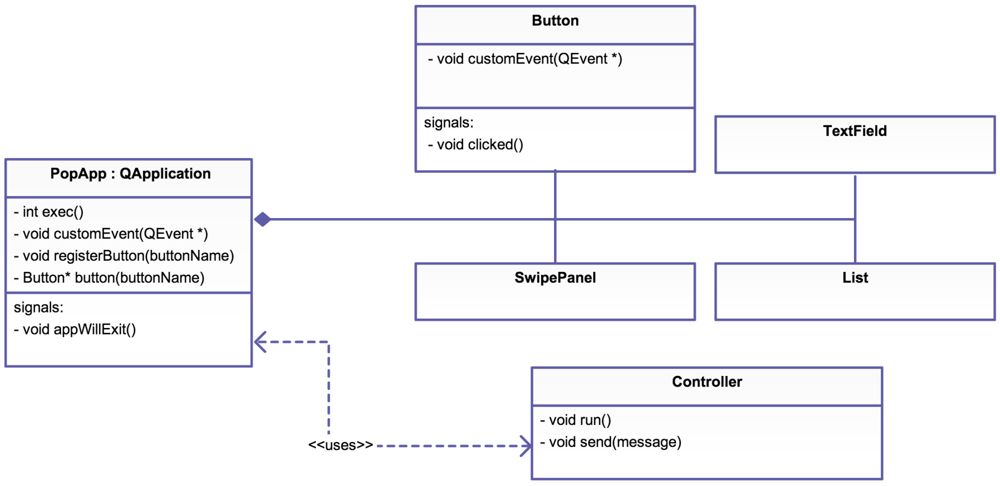


Figure 5.1: Class diagram for the applications' shared library

The most important class in this library for developers is the `PopApp` class. This class extends the `QApplication` class, which is the central class of a Qt application. One and only one `QApplication` object is used by any GUI Qt application, and this object manages the control flow and settings of this application. The `PopApp` class extends these responsibilities by providing lists of GUI component objects, and methods to register and access them (only the methods concerning button handling are shown in the class diagram). The GUI components must be registered with `PopApp` in order to be used. This registration process is the responsibility of the developer – even though we could provide a tool that auto-generates a *main* method which automatically performs the registration of GUI components defined in an application interface description. The developer can potentially register GUI components that are not in the interface description, but these components would never be used. Finally, the `PopApp` class is also used to communicate with the application interface displayed on the remote control (for example, in order to change the page that is currently displayed).

The `Controller` is responsible for communications with the rest of the system, and is completely transparent for the developer. When the developer calls the *exex* method on the `PopApp` object (as he or she would do with a `QApplication` object), the *run* method of the controller is executed on an independent thread. This method is a while loop blocking on the sysV queue between the AM and the applications. As described in section 4.1.3.1, the controller only fetches messages from the queue whose type value is equal to the application process's PID. These messages uniquely identify a GUI component, either by name – provided by the developer in the description for buttons, textfields, labels, and sliders – or by the page name for lists and swipe panels. Upon reception of such a message, the controller sends a customized `QEvent` to the target component object (or directly to `PopApp` itself for the *appWillExit* command). This relies on the asynchronous event system provided by the Qt framework. A reference to the object is obtained via the access methods of `PopApp`, and the objects handle the event – with their *customEvent* method – by emitting the corresponding signal. Listing 4 shows the similarity between a classic Qt application and an application for the TV. A button named "buttonExit" has been defined in the corresponding application's interface description.

```
{
    QApplication app(argc, argv);

    QWidget window;

    QPushButton quit("Quit", &window);
    QObject::connect(
        &quit,
        SIGNAL(clicked()),
        &app,
        SLOT(closeAllWindows()) );

    window.show();
    return app.exec();
}
```

Classic Qt application

```
{
    PopApp app(argc, argv);

    QWidget window;

    app.registerButton("buttonExit");
    QObject::connect(
        app->button("buttonExit"),
        SIGNAL(clicked()),
        &app,
        SLOT(closeAllWindows()) );

    window.show();
    return app.exec();
}
```

TV application

Listing 4: Similarities between Qt applications TV applications.

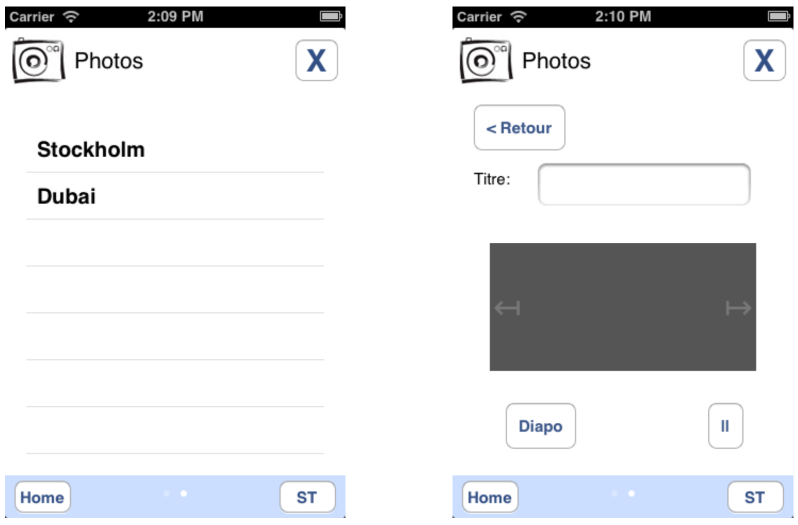
5.4 Application example: Photo viewer

The *Photos* application uses all the GUI components currently available in the SDK, except for the slider. Its interface is composed of two pages: The first page presents a list of picture galleries available on the television, while the second page is used to browse through pictures from a gallery. Galleries correspond to folders in a pre-defined location within the television's filesystem, but this behavior should be improved in a future version of the application by using a SQLite database in order to provide the user with a more flexible application (e.g. a picture could potentially belong to two or more galleries at a time). Moreover, this application will also be responsible for importing pictures of the user (stored in a plugged-in external storage device) to the television in a future version.

Listing 5 shows the interface description of the application. The list of available picture galleries in the first interface page (see Figure 5.2(a)) has been hard-coded as it is not yet possible to send this information to the interface from the application code. Similarly, the textfield which is designed to display the picture title and allow the user to modify it in the second page (see Figure 5.2(b)) cannot yet display the picture title. The grey area on this second page is a swipe zone that allows users to manually change the displayed picture by swiping his or her finger in this zone. The swipe zone is one-dimensional in this example, but it could be made two-dimensional by changing its parameter in the interface description.

```xml
<interface>
  <page name="home">
    <list posX="20" posY="40" width="280" height="380">
      <cell label="Stockholm" />
      <cell label="Dubai" />
    </list>
   </page>
  <page name="gallery">
    <swipezone type="horizontal" posX ="35" posY="150" width="250" height="120" />
    <button name="return" label="&lt; Retour" posX="20" posY="20" />
    <label name="lbl_title" text="Titre:" posX="20" posY="80" />
    <textfield name="txtTitle" posX="80" posY="75" width="200" />
    <button name="launch" label="Diapo" posX="50" posY="300" />
    <button name="pause" label="||" posX="240" posY="300" />
  </page>
</interface>
```

Listing 5: XML interface description for the *Photos* application



(a) First interface page (b) Second interface page

Figure 5.2: Screenshots of the *Photos* application

5.5 Future improvements

A lot of work remains to be done on the application development tools. The description of application interfaces must be made easier by providing a graphical editor and automatic layout managers, in order to make the developers able to describe interfaces that look their best on multiple size of screens. The development of applications themselves can also be improved by generating automatically a main method which registers the GUI components declared in the interface description with the applications' shared library.

Moreover, the behavior of GUI components must be improved by providing requests from the application to code to the GUI component for most of them. At the moment, it is only possible to change the currently displayed interface page on the remote control and update the value of the slider components. Additional components will also be created for the SDK. However, both implementing new components and provide additional control over them from the application code will rely on mechanisms that are already present and exploited in the system, so it should be easily done.

Chapter 6

The remote controllers

Users will operate the television via a remote control of their choice. This remote controller could be the one shipped with the television (as described in section 6.1) or their own smartphone, via a dedicated mobile application (see section 6.2). The term *remote* will be used from this point forward in the thesis to refer to both the included device and mobile control applications. The characteristics of the remote will lead to a good or bad user experience. As a result, we must minimize the latency from the moment the user initiates an action (e.g. by tapping on a button) to the moment where the result of the action appears both on the TV screen (e.g. launching an application) and on the remote itself (e.g. making the button appear depressed and/or changing the information presented on the screen). Another major criteria is the navigability of the remote interface. Unlike the majority of remotes which are used to navigate an interface shown on the TV screen itself (e.g. with the arrow buttons), our remote will have its own GUI. This allows us to show the command interface only on the remote, rather than on the TV screen where it would hide (or entirely replace) the displayed media. This also allows a more direct interaction with the interface, as the user does not have to select the correct item using only the arrow buttons, and then press the “OK” button on the remote, or use one of a number of function buttons – which generally give the user no indication of what they are for (for example the red, green, blue, and yellow buttons on many STB remote controls).

With our remote, the user can directly press the button he or she wants and we can dynamically change the labels on buttons to reflect what action they will invoke. As described in section 5.1, applications are developed together with an interface description which allows application developers for the TV to build their own user interface, adapted to their application. This customized interface can improve the user’s experience, as users do not have to guess what action a button will perform in a given context.

6.1 Included remote

This section describes the work accomplished in order to build a dedicated remote controller for the television. This device has to be cheap to build, as it will be shipped with every television, and it must be very battery-efficient, because users are not used to recharge or to change the batteries in a remote controller very often. Subsection 6.1.1 presents several screen technologies that are both innovative and power-efficient and subsection 6.1.2 presents the prototyping platform that has been used during this project.

6.1.1 The display screen

The technology chosen for the display screen is very important, as it is likely to be one of the most battery-consuming elements of the remote control. For example, it is possible to greatly reduce power consumption of the screen by choosing a reflective display instead of an emissive one. A reflective display uses an external lighting source, and operates by reflecting or absorbing light, as opposed to emissive displays which necessitate backlighting by a light emitting source in order to work. Three potential passive display technologies are electronic paper (see section 6.1.1.1), interferometric modulator displays (see section 6.1.1.2), and liquid crystal displays.

6.1.1.1 Electronic paper

Electronic paper was developed at Xerox in the 1970s using dipolar spheres (two extremities of the sphere carry an opposite electrical charge) which are free to rotate. One "side" of the sphere is black and the other is white. Each sphere is suspended between a pair of electrodes. When a charge is applied to the electrodes, the polarity of this charge makes either the black face or the white rotate to face the electrode, which causes external light to be absorbed or reflected by the sphere. The electrodes were made with a transparent material, such as indium tin oxide.

This technology has evolved toward electrophoretic displays, invented by Joseph Jacobson and Barrett Comiskey [46]. These two were among the co-founders of the E Ink corporation, which is currently the main manufacturer of electronic paper. The technology consists of spheres containing positively-charged black particles and negatively-charged white particles, each of the spheres is wedged between a pair of electrodes. If the surface electrode is charged positively, then the white particles accumulate at the surface of the sphere and reflect the light (thus creating a white pixel). On the contrary, if the surface electrode is negatively charged, then white particles are pushed away from the surface, and the black ones take their place, absorbing the light (thus creating a black pixel).Electronic paper can also display colors in addition to a monochrome grayscale. This is done by using a matrix of color filter arrays (consisting of four sub-pixels, which are red, green, blue, and white).

This technology has many advantages. First of all, it has very low energy requirements, as it is a bi-stable technology. This means that if the image displayed on the screen is static, no power is consumed to maintain it. Only when the picture changes does the display screen utilize energy. This form of display is more relaxing for the eye than a backlit screen, and it allows for far greater visibility under direct sunlight and/or when viewed from a wide angle. However, the refresh rate is quite low (typically 120ms for a monochrome display, and up to 980ms for a color one [47]). This prevents the display of smooth and detailed animations in color, but does not prevent the display of a relatively static user interface, especially if transitions are done in black and white, and colors are added once the transition is over.

6.1.1.2 Interferometric modulator displays

Mirasol displays (which is the trademarked name for interferometric modulator displays) constitute a really promising technology developed by Qualcomm [48]. They utilize the principles of constructive and destructive interferences of optical waves. When light hits a structure consisting of one thin partially-refelective membrane and a reflective membrane situated below. As a result, the light reflected by the bottom membrane is slightly out of phase with the light reflected by the top one. The phase difference depends on the height of the optical cavity created by the two membranes (on the order of a hundred nanometers). The phase difference will have a constructive effect for some wavelengths (i.e. colors), and a destructive effect for others, thus determining which color will be reflected by the cavity. If the bottom membrane is close enough to the top one, constructive interferences are located in the ultraviolet wavelengths, which are invisible to the naked eye, hence appearing black.

This technology is really promising, as it provides several advantages over the electronic paper, while still being a bi-stable technology (once the bottom membrane is in place, no additional energy is necessary to maintain it there). First of all, making use of interferences to produce colors instead of filters (which operate by absorbing light) exploits the available light much more efficiently. But the main advantage concerns the refresh rates possible with these displays: As the bottom membranes of the different cells must only be displaced by a couple hundred nanometers to switch between reflecting a color or “reflecting” black, Mirasol displays can support refresh rates on the order of tens of milliseconds, which is far better than the nearly one second of color e-paper, and is comparable to video frame rates.

However, last June Qualcomm decided to stop production of these displays, and instead to license the technology to others. This change might make these displays unavailable for a long time (depending upon if and when someone licenses and manufactures these displays).

6.1.2 Prototyping equipment

An Arduino Uno board is used for prototyping the included remote control. It is an open-source single-board microcontroller popular in the field of electronic hobbyists, since it is easy to use and a wide range of tutorials are available for it on the Internet (see for example [49], a 50 chapters and counting tutorial series on Arduino). The Uno board is composed of a USB interface (to transfer code from a computer to the board and to power the board), an additional power socket, 14 digital pins, and 6 analog pins. The board uses the ATMega328 micro controller, which belongs to the Atmel AVR family. This micro controller embeds 32 kB of flash memory (0.5 kB of which is taken by the boot loader), as well as 1 kB of EEPROM (additional memory that may be utilized by programs to store persistant data into).

Digital pins can be set as input or output pins. An input pin can be read to determine if the pin is in a HIGH (pin connected to 5V) or LOW (pin connected to GND) state. An output pin can source up to 40 mA to the rest of the circuit. Analog pins can be used as general purpose I/O pins, but they are also 10-bit analog to digital converters (ADCs), which means that they can measure an input voltage and compare it to a reference voltage, returning a value between 0 and 1023, i.e. $2^{10} - 1$ (5V is the default reference, but any voltage between 0 and 5V can be applied to the AREF pin to be used as reference).

A pre-built board adapted to the Arduino (called a “shield”) supporting a touchscreen LCD screen is used for the prototype. This shield is presented more in more detail in section 6.1.3.3.

6.1.3 Developing on the Arduino

This section presents several basic aspects of programming an Arduino board (section 6.1.3.1), how to make the Arduino and a computer interact over the USB interface (section 6.1.3.2), and how the screen shield operates (section 6.1.3.3).

6.1.3.1 Basics

The Arduino programming language is based on Processing, which is a programming language designed to be simple enough to be used by artists without pre-existing computer science background to perform digital graphics creation. AVR-C code can also be introduced in an Arduino program. An Arduino program (also called "sketch") is then transformed into AVR-C and compiled into binary code before being transmitted to the micro-controller.

Every Arduino sketch must include a `setup()` method, which is called once when the sketch is launched on the board and should be used to perform initializations (such as setting pins as inputs or outputs), and a `loop()` method, which is called repeatedly during the lifetime of an application. Classical control structures (e.g. if, while, for) as well as familiar variable types (int, long, arrays, etc.) are available while writing sketches. Additional methods are provided in order to act on the input and output pins. Examples of these methods include the `pinMode` method, which sets a pin in a INPUT or OUTPUT mode (both constants are providing by the language), the `digitalWrite` method which sets an output pin to a HIGH or LOW state, and the `digitalRead` method which returns the state of an input pin. Some other useful methods are provided, such as `millis`, which returns the number of milliseconds passed since the beginning of the program and `delay` which makes the program block for the given number of milliseconds.

Figure 6.1 presents the code and the electronic circuit of a basic example: making a light emitting diode (LED) blinks (the LED is switched on for 1 second and switched off for 1 second). This sketch was copied from the Arduino *getting starting* guide [50], and was released into the public domain.

```
int ledPin = 13;

void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

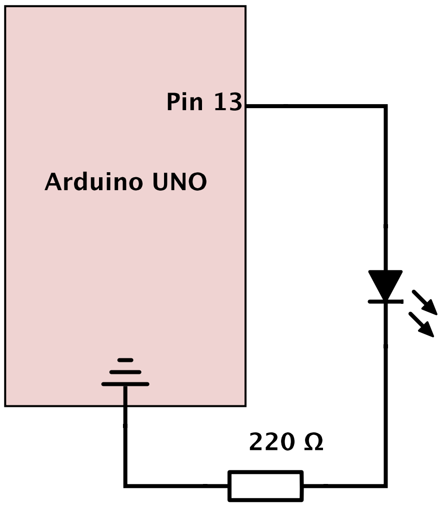


Figure 6.1: Basic Arduino sketch and electronic circuit

While this code works fine and does make the LED blinks, it is flawed in that we cannot perform any other work while the LED is blinking. This is due to the use of the delay function, which blocks the code until the specified delay has passed. In order to make the LED blink while still being able to perform other work, the millis method should be used to check if more than one second has passed since the last LED state change, and if so switch the LED on or off. By using this method, the loop method does not block, and is repeatedly called by the Arduino system, which allows additional work to be performed within this loop.

The boot loader included in the flash memory of the Atmega micro controller is responsible for handling the transfer of an Arduino sketch to the board and for launching the last uploaded program when the transfer is complete (or when the board is first powered on).

6.1.3.2 Serial communication with a computer over USB

The Arduino board pins 0 and 1 act as a serial port, called a universal asynchronous receiver transmitter port (UART). Pin 0 (receive - RX) and pin 1 (transmit - TX) respectively receive and send data. Serial communication means that bits are sent one after the other (as opposed to parallel communication, in which a data bus is used to send several bits in parallel). As only one wire is used to send data, there is no clock signal available for the data recipient to indicate when it is appropriate to read a bit from the wire. As a result, the sender and the recipient must agree upon the rate at which bits will be sent on the line (i.e., agree upon a baud rate). There is no need for an RSR-232 serial port on the PC, since a USB port can utilize a driver to support serial communication.

In an Arduino sketch, the `serialEvent` method may be implemented in order to handle serial communication. This method is called in-between every two calls to the `loop` method, allowing for reasonable handling of serial communication. The programming language offers methods for handling these communications, such as `Serial.begin` to initialize a serial line for a given baud rate, `Serial.available` to check if some data is ready to be read from the serial buffer, and `Serial.read` and `Serial.write`, used for reading from and writing to the serial port.

It is possible to use the `SoftwareSerial` library in order to simulate a hardware UART port using pins other than pins 0 and 1. However, if more than one software serial port is used, only one of them may receive data at any given time. Moreover, the serial communication system provided with a software serial port is half-duplex, i.e. it can only be used to send or to received at a given time.

On the PC side, it is possible to communicate with the Arduino by using the integrated development environment provided for writing sketches for the Arduino. Although this is a simple method to check if a sketch behaves as it should on the board, it does not allow for automatic communication between the board and the PC. In order to test such an interaction, a small program has been written in order to control which of three LEDs is switched on in an electronic circuit via a command line interface on the PC. This command line interface was developed in C using the `terms.h` library. This library is useful for creating terminal I/O interfaces.

6.1.3.3 Using the screen

The screen shield used for the remote controller prototype embeds a LCD screen driven by a Solomon Systech SSD1289 controller and a touchscreen driven by a Texas Instrument ADS7843 controller. As shown in Table 6.1, this shield uses a lot of pins to communicate with the Uno board. However, some of these pins are used to support the microSD (TF) card reader included on the screen shield, and these are not needed for our prototype. As a result, some of these pins can be utilized to perform other tasks (such as radio communication).

Pins 9 to 13 as used for communicating with a SPI (or four-wire) interface. Both the microSD (TF) card reader and the touch controller use this interface,. The target SPI device is identified by the state of the chip select pins (pins 9 and 10). Since the microSD (TF) card reader is not used, this pin is actually free to use on the Arduino. Moreover, since the touch controller is the only device using the SPI interface, the input pin 1O on the shield could be tied to HIGH, thus freeing another pin on the Arduino board. As a result, while using the touchscreen and not the microSD (TF) card reader we can have up to 3 free pins on the Arduino board. This is enough pins to add a device that uses three pins or less. A device using an Inter-Integrated Circuit (I^2C or two-wire interface) would be suitable, and we could even chain other devices to share this interface.

Pin A5 effectively frees 8 pins on the Arduino board, by converting a 8-bit data bus (output from the Arduino board) to a 16-bit data bus (received by the screen). This is done with a shift register: While pin A5 is in the LOW state, output from the 8-bit data bus pins are stored in a shift register. When pin 15 is switched to the HIGH state, output from the data bus are added after the 8 bits stored in the shift register, and the resulting 16 bits are sent to the screen controller. This method uses half as many pins to send data to the screen at a cost of an additional delay in sending this data.

Two libraries written by Henning Karlsen were used for the screen shield: UTFT [51] which provides methods to manipulate the screen (with methods such as `drawPixel` and `drawBitmap`) and UTouch [52] which provides support for the touchscreen (witch methods such as `getX` and `getY`).

Table 6.1: Pin layout of the touchscreen shield

LCD shield pin	Name	Description
0~7	Data bus	8-bit data bus to the screen
9	TF_CS	Chip select pin of serial interface (SPI)
10	Touch select (TCS)	Chip select pin of serial interface (SPI)
11	MOSI (DIN)	Master-Out Slave-In (SPI)
12	MISO (DOUT)	Master-In Slave-Out (SPI)
13	SCK (CLK)	Serial clock (SPI)
A0	Chip select (CS)	Chip Select for the screen
A1	Register select (RS)	Data bus interpreted as data or command
A2	Write (WR)	Read or write from the screen
A3	Reset (RST)	Screen reset pin
A4	Touch_IRQ (IRQ)	Triggered when a touch is detected
A5	Latch signal (ALE)	8-bit to 16-bit data bus

6.2 Mobile remote control applications

Mobile remote control applications take advantage of the widespread availability of smartphones. Users can simply use their smartphone to control their television, hence offering a familiar platform. For simplicity's sake, only an iPhone application has been developed at the moment but an equivalent application will be developed for Android. Several additional functions could be provided for the mobile application users, such as displaying pictures from their telephone on the TV screen or allowing them to remotely stream media hosted on their TV to their smartphone.

Section 6.2.1 gives an overview of the iPhone application's GUI, section 6.2.2 details the connection phase of the application connection to the TV, and section 6.2.3 presents some important technical details about the application.

6.2.1 Overview of the application

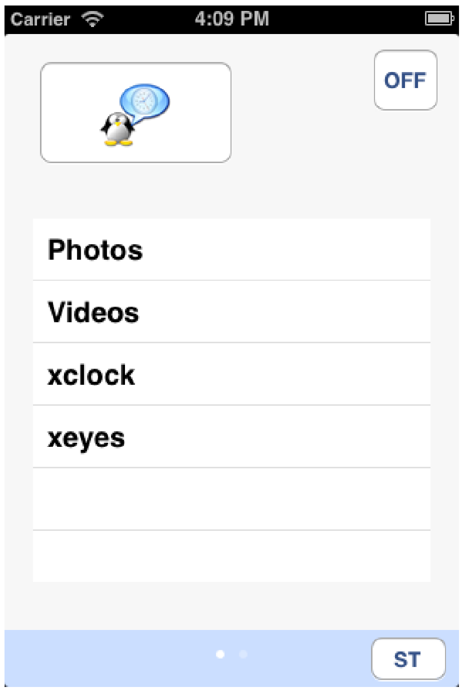
The main view of the application contains an horizontal scroll view, i.e. several subviews conceptually placed side by side, but from which the user can only see one at a time. This scroll view contains at least one subview: the *home panel*. This view displays general information – such as the current screen mode – and offers general actions on the TV – such as launching an application. As seen in Figure 6.2, the home panel is a bit different depending on the screen mode the TV is currently in (FS or BQ, see section 4.2.2). This panel contains a list of applications installed on the TV as well as a button to switch the TV off. When the TV is in FS mode a single button is displayed on the upper-left corner. This button shows the icon of the application currently displayed if any. When the TV is in BQ mode, four buttons are displayed instead, corresponding to the four quarter slots on the TV. Tapping on a button with an icon – i.e. corresponding to a slot in which an application is displayed – displays the corresponding application's *control panel*.

Since at most four applications can be displayed at the same time, there can be up to four different control panels on the remote control application. These panel are located to the right of the home panel within the scroll view, and the position of the control panel in the scroll view depends on the application's slot on the TV: If the application is in the upper-left slot, the corresponding control panel is located directly to the right of the home panel. If an application slot is empty, the scroll view scrolls directly to the next displayed control panel.

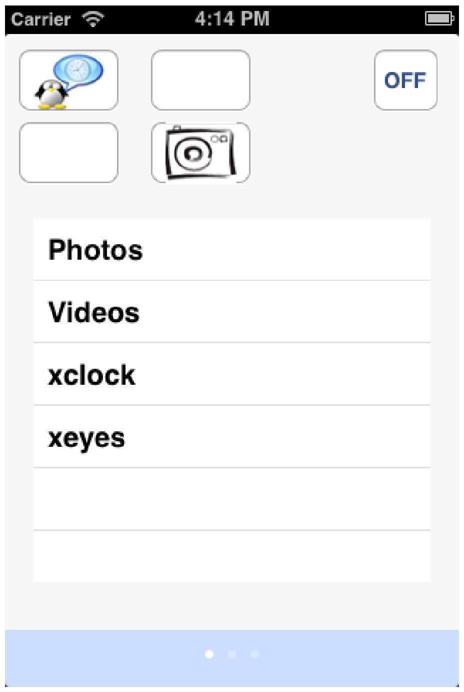
Finally on the bottom of the screen is a fixed toolbar, which is always displayed independently of the currently displayed subview in the scroll view. This toolbar shows the current number of subviews and the current location within the scroll view, a shortcut to the home panel on the left (if needed), and a button to change the screen mode. As shown in Figure 6.2, this button (labelled "ST") is not always displayed while the home panel is. Indeed, if the TV is currently in the BQ mode, the user cannot switch to the FS mode, since we would not know which application should be displayed on the TV. When an application control panel is displayed, the user can switch to the FS mode, and the corresponding application is kept on the screen.

6.2.2 Connecting the remote control application to the TV

When the remote control application is launched, it has to connect to the TV. In this section we assume that the smartphone has a WLAN interface and is connected to the same home network that the TV is connected to and that this network is a single subnet. As explained in section 4.5.1, the application cannot know in advance the IP address of the TV or if the TV is even switched on. As a consequence, the application first tries to contact the address discovery module of the TV. If this module is responsive, this means that the TV is switched on, and the application can learn the TV's MAC and IP addresses. If the address discovery module is not responsive, this means that the TV is switched off. If the remote control already knows the MAC address of the TV, it can send the TV a magic packet to switch it on. However, if the TV's MAC address is not known by the application, then it cannot do anything other than alert the user. In this case, the user has to switch on the TV by another means (e.g. with the



(a) Home panel while in FS mode



(b) Home panel while in BQ mode

Figure 6.2: Remote control application's home panel

included remote control, or with a physical button on the TV itself). Once the TV is switched on, the remote control application can connect to it and learn about its MAC address in order to be able to switch it on later. Figure 6.3 is an activity diagram summarizing these steps. When it learns the TV's addresses they are store in the microprocessor's FLASH storage area for later use.

6.2.3 Technical details of the application

This section presents several some of the details about the mobile remote application. A class diagram is shown in Figure 6.4. This diagram presents the architectural organization of the remote control application's control panels. For the remainder of this section, both the objects' class name (i.e. `Object`) and their "real" name (i.e. object) will be used interchangeably to refer to the object. For simplicity's sake, the classes related to the home panel, the main view, and the application delegate have been omitted, as they correspond to a classic implementation of an interface-builder based iOS application. Section 6.2.3.1 details the launching of an application on the TV from the mobile device, section 6.2.3.2 presents the method used by the mobile application to communicate

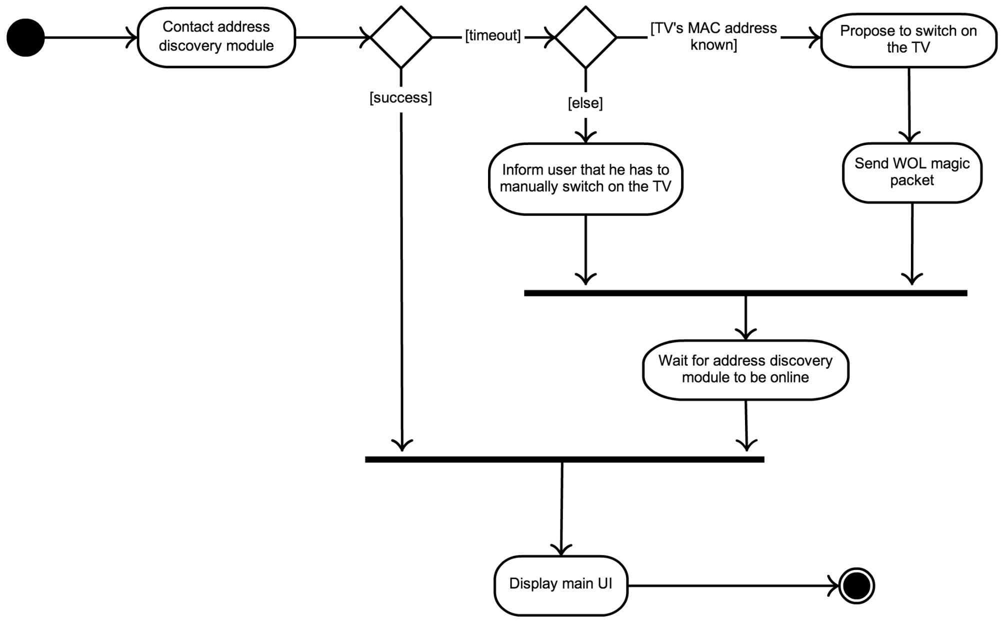

Contact address discovery module
[timeout]
[TV's MAC address known]
Propose to switch on the TV
[success]
[else]
Inform user that he has to manually switch on the TV
Send WOL magic packet
Wait for address discovery module to be online
Display main UI

Figure 6.3: Remote control application's connection phase

with the TV, and section 6.2.3.3 details how user interactions are transmitted within the remote control application.

6.2.3.1 Launching applications

In order to launch an application, a user must select the application from the application list displayed on the home panel. This application list is obtained from the TV, with a *listApps* request (see Table 6.2). If the TV is in FS mode, the *launchApp:app_name* request is sent to the TV right away (see Table 6.2). If the TV is in BQ mode, the user is invited to choose which slot to place the application in (the buttons corresponding to the slots on the home panel begin blinking). Once a slot is selected, a *BQ0-3* request is sent to the TV, to indicate to the WM which slot gets the focus (see section 4.2.2), followed by the *launchApp* request.

The AM launches the application on the TV and sends back the resulting application's PID and the application's interface description in the protocol buffer's binary model as a response to the request (see section 6.2.3.2). The home panel controller creates a new `ControlPanel` using this information. The ControlPanel sends a *getIcon* request to the TV, which is sent in binary form as well. Launching an application requires two to three requests, sent with a very short delay between each other, which might cause a problem on the TV side. Indeed, these requests could be aggregated together (as explained in 4.3.3.2) as

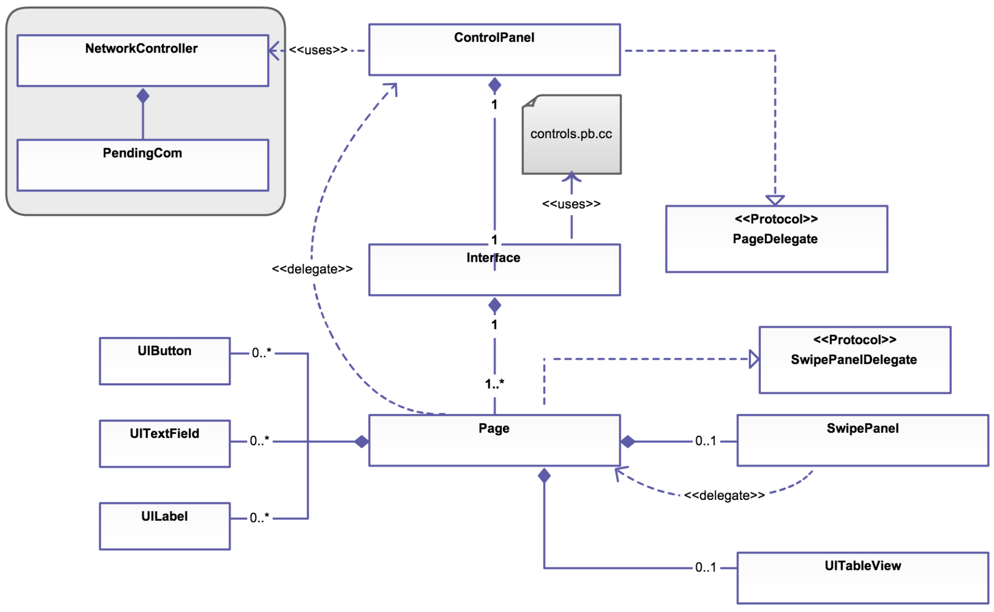


Figure 6.4: Control Panel and Network Controller class diagram

no size information is yet added to the requests. Subsequently, these requests are forwarded to the appropriate module, filling up their respective POSIX entry queues. The first problem will be solved in the next version of the CM, while the second is not really an issue, since the second request for the AM is sent after the first one has been processed. However, it would be more efficient to send only one request instead of three in later versions of the mobile application.

Once the control panel is created, it handles the creation of the `Interface` for the application with the interface binary data received from the TV. Parsing this data is done with the help of the *controls.pb.cc* file, generated by the `protoc` tool (see section 5.1). This file contains class definitions for the data structures forming applications' interfaces (e.g. pages, buttons). If the interface data is successfully parsed, the `Interface` creates one `Page` subview per page item, as defined in the interface description. Each page may contain buttons, textfields, labels, and at most one swipe panel and one table view as described in section 5.1.

6.2.3.2 The network controller

The `NetworkController` is the object handling communications with the TV, once the TV's addresses are known. Although only one instance of this object is created during the lifetime of the remote control application, it is not a *singleton* object *per se*, as defined by the Singleton design pattern. This controller is instantiated by the main view controller of the remote control application when this main view controller is created (i.e. after the initial connection phase has been successful). A reference to this instance of the network controller is then passed to objects that need to use it as a parameter of their initializing function (i.e. the equivalent of a Java constructor), following the *Dependancy Injection* design pattern.

The network controller uses the NSStream classes, provided by iOS. These classes are meant to be used in a event-oriented fashion. Once a stream is set up – or in this case two streams, an input stream for messages from the application to the TV, and an output stream from the TV to the mobile device – the `handleEvent` function of the stream's delegate (the network controller itself in our case) is called when an event has happened on the stream. One possible event for a stream is `NSStreamEventHasBytesAvailable`, which indicates that the stream contains some data. Within the `handleEvent` is typically a switch to choose which action to perform on the stream depending on the type of event that happened on it.

The network controller offers two request modes: requests that expect a response (called *pending*) and a impler one-way request. When the network controller is asked to send a request to the TV, it prefixes the request with an ID. This ID is a 4-digit number, with trailing zeros if needed, in order to make parsing of the request easier on the TV side. The request ID is incremented after each request is sent, until it reaches 9998, when it is reset to 0000. ID 9999 is reserved for messages which come from the TV to the mobile device and which are not a response to a previous request. Such a "reversed" request can be for example the request to display an interface page for a given application. In the case of a one-way request, the request ID is not needed, but it is still placed in the message to facilitate parsing. The request is also prefixed by its destination module (provided as a parameter of the request methods). A list of requests that can be sent by the remote control application to the TV is shown in Table 6.2. These requests are composed of three to four fields separated by columns. The two fields indicate the target module of the request and the request ID, and both these fields use a fix number of characters, in order to accelerate the processing of this request by the CM. The other fields do not have a fixed size, and the column is used by the target module to parse the request and the optional parameter. This is done to improve code readability, but we could get a performance increase by using a binary protocol, such as Google's protocol buffers.

When sending a pending request, the network controller is provided with two block objects [53]. These block objects are a feature introduced in iOS 4 as an effective method for creating callback methods. They are *closures*, i.e. they are functions associated with an encapsulated state (the state from which they have been declared). As a consequence, blocks can access variables outside of their normal lexical scope, even when used in a different object than the one they

Table 6.2: Requests sent by the remote control to the TV

Target module	:	Request ID	:	Request	:	Optional parameter
poam	:	request ID (0000-9998)	:	listApps	:	
				launchApp		application name
				killApp		application PID
				getIcon		application name
				command		appPid_*command*
powm				FS		
				BQ		
				BQ0-4		

were created in. More precisely, blocks capture every variable accessible from the scope from which they have been *declared*. The first block is the action to be performed when a response arrives for a pending request. The second block is executed if the pending request timeouts (after a fixed pre-determined period).

A `PendingCom` object, containing the ID of the request, a timer, and the success and failure blocks is created when a pending request is sent. The network controller examines the ID of incoming messages. If the ID corresponds to the ID of a pending block, then the success block is executed with the data contained inside the message. If the ID is 9999, the message is a request sent on the TV's initiative, and this request is transmitted to the appropriate message handler. Finally, if the ID does correspond to one of the pending communications and is not 9999, this probably means that the originating pending communication has expired. In this case, the incoming message is simply ignored, as the failure block associated with the pending communication should already have been executed.

When a message arrives on the input stream of the controller, the controller reads the first 8 bytes of the data, which contains the size of the message. It then retrieves the message. This mechanism prevents messages colliding (as described in section 4.3.3.2), and allows the reception of large messages (such as the binary data of an application icon), *without* allocating a large memory space every time some data is available on the input stream. This would be a waste of memory in the majority of cases, as the received messages are generally small.

6.2.3.3 From the application interface to the application and back again

The *delegate* model, introduced by Apple, is an illustration of the inversion of control involved in event-drive systems. A delegate is a class that implements required methods of a *protocol* (the Objective-C equivalent of a Java interface). A host object has a reference to a delegate (which can remain null if no delegate has been set). Whenever the host object needs to notify its delegate of something (e.g. a GUI input), it calls the corresponding protocol method on its delegate. Delegates are often used by off-the-shelf components that perform independent and generic work, and call the developer's code only when customized actions are needed. An example of this model is the implementation of the application interface on the control panel.

When a GUI component is used, for example when an item is selected on the list, the `UITableView` class calls the `tableView:didSelectRowAtIndexPath:` method on its delegate, which has to *adopt* (i.e. implement the required method of) the `UITableViewDelegate` protocol. The delegate is responsible for performing the specialized action that is supposed to take place for this implementation (in our case, informing the TV that an item has been selected with a one-way request). This model allows for a separation between generic and specialized actions, and leads to more re-usable components, as their behavior can be transparently adapted, without requiring knowledge of the inner-workings of the component itself.

A `Page` is the delegate for all the GUI objects created following the interface description. When the user performs an action on one of them, the page informs its own delegate (the control panel), with a message to send to the TV. This message is already written by the page, containing relevant information provided by the protocol method (i.e. what text does the textfield contain after the user has changed it). The ControlPanel delegation is not really needed here, but is used for convenience: the network controller does not have to be injected as a reference to every page through the interface, which itself does not need it.

Chapter 7

Evaluation of the prototype

This chapter evaluates the prototype. The most important criteria for this prototype (besides stability) is to bound the latency observed by the user between the moment he or she initiates an action until this action is observed on the TV and on the remote controller. We also need to ensure that messages can always be enqueued in the message queues in the television system.

Section 7.1 presents the methodology used to evaluate the prototype, and section 7.2 details the results obtained during this evaluation.

7.1 Methodology

Every message sent by the remote control transits through a single place in the remote controller and in the television. This makes timing of latency very easy. On the remote control iPhone application, every message is sent by the `NetworkController`, and on the television every message is received by the CM.

However, as described in section 6.2.3.2, messages from the remote control application can be sent in either a one-way mode (in which no response is expected from the TV) or in a pending mode (in which a response is expected, and a `PendingCom` object is created). For timing purposes, every message that would be sent using the one-way mode is instead sent in the pending mode by the network controller, to enable us to measure latencies for every type of messages. As a result, small modifications have been necessary in the television system, in order to make each module respond to every request, even if the response contains only the request ID of the message and this response will be discarded by the network controller after the timing has been observed, (it will simply be discarded since no success block had been registered in the `PendingCom`).

In the remote control iPhone application, the `timeIntervalSinceNow` method of the `NSDate` class has been used. This method returns a `NSTimeInterval`.

This `NSTimeInterval` is a double and is documented by Apple to have a sub-millisecond precision. The `PendingCom` class has been slightly modified to get the current NSDate when created and to log the NSTimeInterval corresponding to the time difference between this NSDate and the NSDate corresponding to the moment when the response corresponding to the request represented by the `PendingCom` is received.

In the CM on the television side, the `gettimeofday` function has been used to get a `timeval` structure containing the number of seconds and microseconds since the epoch (i.e. January 1st, 1970 at midnight). This function has an hardware-dependant precision, but a sub-millisecond precision should be obtained on recent standard PCs, as in this case this function relies on a dedicated hardware clock. However, the CM was not designed to match responses to requests depending on their request ID as was the network controller in the remote control. As a result, slight modifications have been necessary to allow for these latency timings on the TV side. The received request is stored in a linked list of structures containing the request ID and the timestamp corresponding to the time when this request was received. When a response is sent to the remote control application by one of the TV modules, the CM must determine the request ID of this response, match it with the corresponding structure in the linked list, and log the time difference between the two timestamps. These additional steps on the CM side should not introduce too much additional delay, as the request is inserted in constant time in the linked list, and retrieved in O(n) time with respect to the number of pending requests (which should always be low).

7.2 Results

Figure 7.1 and Figure 7.2 present the latencies observed during a short utilization of the television. During this period, 70 requests were sent to the television from the remote control. These requests spanned all available requests (a list of the possible requests is provided in Table 6.2, on page 62). Moreover, an instance of the *Video* application was running and a HD video was playing for most of the time (from request ID 11 to request ID 25 and request ID 30 to request ID 67). This is particularly important for two reasons. First, we must ensure that the video quality is not altered when a request is processed by the television system (e.g. the video must not become jerky and/or have distorted audio). Second and foremost, the *Video* application sends one message per second to the corresponding application interface on the remote controller in order to update the position of the slider on this interface. This adds some load to the system, even if the messages updating the slider are not timed (as they are requests sent from the TV to the remote control).

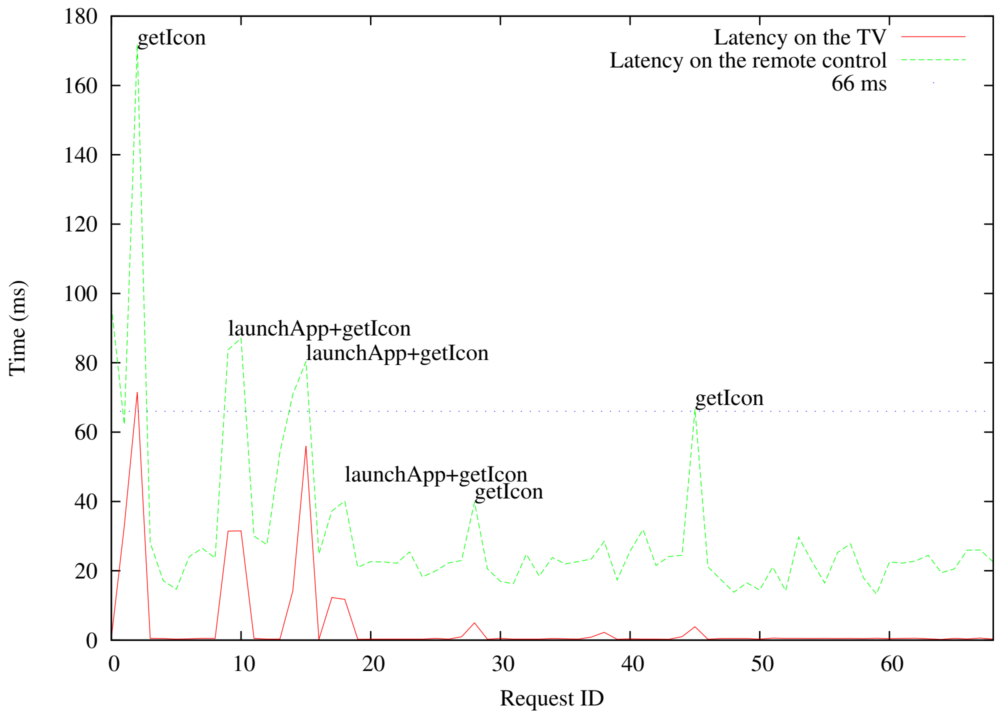


Figure 7.1: Observed latencies on the TV and on the remote controller

A latency of $1/15^{th}$ of a second (i.e. 66 ms) has been chosen as an arbitrary ideal latency under which the result of a user interaction should be observed on the TV and on the remote control. It is important to note that timings from the remote control application's perspective relate to the time it takes to receive a response to a given request. As a result, these durations should be divided by two to obtain the approximate time at which an operation has been processed on the TV, hence providing visual feedback to the user. In the majority of cases, this visual feedback on the TV is more important than the visual feedback observed on the remote control, and in all cases, this early visual feedback makes the operations seem more reactive. Thus delays of up to 132 ms as seen by the remote control could be deemed satisfactory with our arbitrary ideal latency. As seen in Figure 7.1, mainly two requests have latency issues: the *launchApp* request and the *getIcon* request, both from the remote controller's perspective and from the TV's perspective.

From the TV's perspective, both these operations involve fetching a file (the interface description for the *launchApp* request and the icon file for the *getIcon* request) in the TV's filesystem and reading the binary content of this file into memory. This latency could be avoided by caching the content of these files in memory (within the `AppRepository` object). This caching would have to be

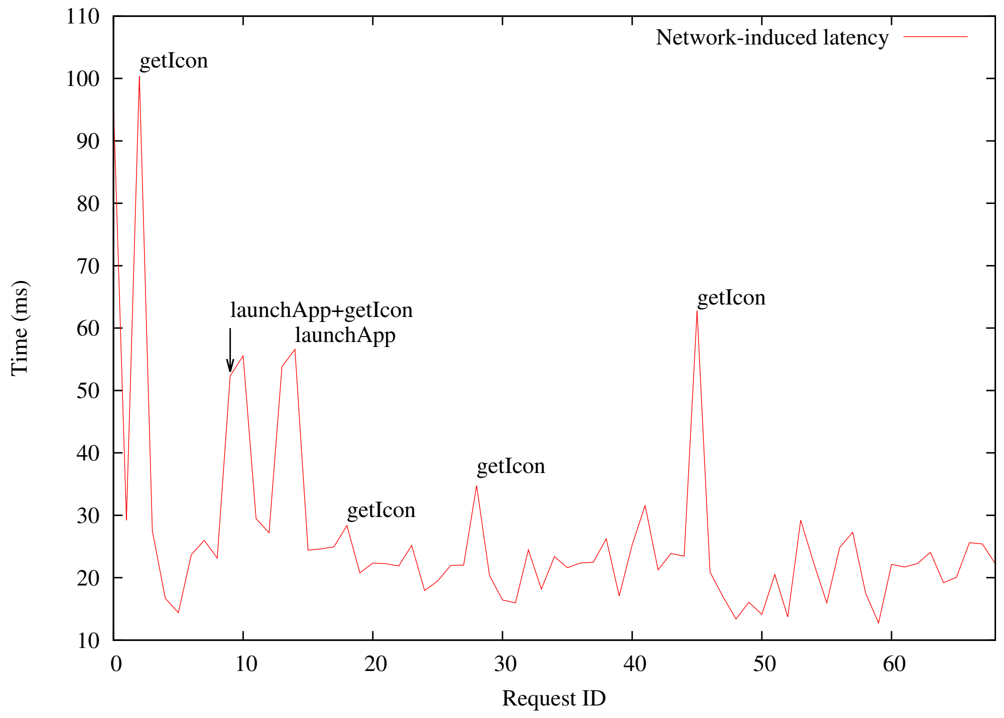


Figure 7.2: Network-induced latencies

adapted depending on the number of applications installed on the TV, since even if the icon files' size is limited to 16 kB and the interface descriptions' size should be small (due to the binary format used), caching these files in memory for a large number of applications could become costly in terms of memory. As a result, lazy caching (i.e. caching the information when it is requested) will probably be implemented in a future version of the AM for those applications that are launched most often. Moreover, these operations involve blocking system calls in order to retrieve the file contents from the filesystem, and the processing of these operations does not handle these blocking operations as such for the moment. As a result, if the system call blocks, the processing of the request blocks as well, which might delay the processing of subsequent requests. Of course, the blocking operations should be handled better in a future version of the AM, so as not to block waiting for a system call to return but rather to process a subsequent request in the meantime.

From the remote controller's perspective, the observed latencies for these operations are also due to the network cost necessary to transmit a bigger amount of data. Indeed, all other responses contain only a string which is always smaller than a few tens of characters, with the exception of the response to the *listApp* response, which contains a comma-separated list of all the names of applications which are installed of the system. While the number of applications installed on the TV remains low, this is a viable strategy. However, if the number of installed applications grow, it will be necessary to cache this information in the remote control and only check if the cached information is still valid when the application starts.

Interface descriptions and application icons could also be cached in the remote control application's storage for a few popular applications (i.e. the most often applications launched by this remote control application). This would avoid the cost of sending the content of these files each time one of these applications is started. However, the remote control application is much more memory-constrained than the television system, so a lot more of these files could be potentially cached in memory on the TV's side than in the filesystem on the remote controller's sider.

As seen in Table 7.2, the observed latencies can also be caused by random network throughput fluctuations. Even though network latencies are consistently greater for *launchApp* and *getIcon* requests than for other kinds of requests, some requests can have a high latency independently of their type. For instance, the request with the request ID 13 is a *focus* request. This request has an observed latency of 54.131 ms from the remote control application's perspective, for a latency of only 0.297 ms from the TV's perspective. This means that a 53.834 ms latency was induced by the network, although the request string is only 12 characters long, and the response is only 13 characters long. For this experiment, the average network latency (rounded to two digits after the point in the thesis) was 26.70 ms. However, the standard deviation for this latency is 15.72 with a minimum round-trip time (RTT) of 12.737 ms and a maximum RTT of 100.404 ms.

The network throughput can degrade the observed latencies, especially for a LAN with a lot of traffic, or a poor Wi-Fi signal quality. A test was conducted with a very weak Wi-Fi signal and high network latencies. Results for this experiment are shown in Table 7.1. As seen in this table, even though the television system efficiently processes each received request, network-induced latencies greatly delay the visual feedback received by the user. However, network throughput fluctuations are not a parameter on which we can act, thus a special consideration must be given to requests which encounter a timeout in the remote controller (which could potentially desynchronize the state of the remote control and the state of the TV). During this experiment, 8 requests encountered a timeout on the remote controller. Even though the performances of the network is terrible in this example, the average latency observed in the remote control application is only slightly above 132 ms, which is twice the arbitrary ideal delay. This means that visual feedback is still provided to the user on the television under this latency limit in the majority of cases.

Finally, results obtained from the watcher module monitoring the POSIX queues correspond to what could be expected: The maximum number of messages observed on each of the POSIX queues was only rarely above 0, and this number has never reached 2 for any POSIX entry queue, unless the corresponding module was manually stopped for testing purposes. This indicates that messages are processed quicker than they are enqueued. Moreover, the fact that the observed maximum is only rarely above 0 indicates that messages are consumed almost as soon as they are enqueued, so when the watcher module observers the number of messages on each queue, the probability that a message is waiting to be retrieved on a queue is very low. This is comforting for the stability of the system as a whole because it means that the probability that a message queue overflows is infinitesimal, unless one of the core module of the system has crashed or has hunged on an operation. In this case, the problem of this module would be detected by the watcher module observing an abnormally high number of messages in the module's entry queue, and actions could be taken in order to solve this problem (e.g. by starting a new instance of the module and providing facilities to synchronize it with the rest of the system).

Table 7.1: Experiment with poor LAN conditions

Request kind	Number of requests	Avg. Processing time (ms)	Avg. latency (ms)	Avg. network latency (ms)
listApp	6	0.46	149.04	148.58
launchApp	34	3.93	166.46	162.53
killApp	29	0.56	121.09	120.53
command	46	0.52	115.90	115.38
getIcon	34	8.83	57.38	48.55
FS	13	0.34	228.46	228.13
BQ	14	0.35	178.46	178.11
Focus request	79	0.31	162.06	161.74
Total	255	2.00 (σ=8.34)	139.68 (σ=194.49)	137.68 (σ=195.03)

Chapter 8

Conclusion and Future work

8.1 Conclusion

The objective for this thesis project was to develop a prototype for an innovative smart television that overcomes the shortcomings observed in the products currently available on the market. These shortcomings revolve around a poor global user-experience induced by user interfaces that are difficult to navigate and hard to understand. Taking advantage of the widespread availability of touch screen devices already owned by the users (e.g. smartphones and tablets), this prototype proposes a novel way of interacting with the television. Rather than being confronted with a fixed set of buttons on a remote controller and navigating long chains of menus on the television screen, users are shown a control interface that is customized depending on the information displayed on the television. Moreover, navigating menus can be done solely using the remote controller, which prevents a main user interface or a succession of menus from being displayed on the user's television screen, potentially hiding the current media completely or partially. This project was divided into three sub-projects: the television system, the applications, and the remote controllers.

The use of two major principles in modern software engineering (high-cohesion and low-coupling) has made the development of the television system easier, quicker and more flexible, by decomposing this system into independent modules that communicate with each other using well-defined APIs. Modules communicate with each other by sending messages via message queues. Even if the need for inter-process communication was introduced by using independent modules (running on independent processes), the evaluation has shown that the delay caused by these additional steps was not high enough to be noticeable by the end-user. This architecture is particularly important for flexibility, which is paramount when developing a prototype. Each module was developed in isolation from other modules, receiving requests from manual commands and sending responses to placeholder stubs, and was integrated into the rest of the system only after it was deemed sufficiently functional. In the future, it will be possible to easily add modules into the system by following the same principle, without disrupting the system itself. Additionally, pre-existing modules can easily be refactored to be more efficient or secure, as long as they respect the documented APIs.

Applications for this prototype are composed of two parts: the actual application, running on the television, and the application's interface, displayed on the remote controller. This separation provides the user with an innovative way of controlling the applications: This user only sees controls that he or she might use on the application at any given time. This greatly simplifies the utilization of applications on our smart-television with respect to a classical remote controller. Indeed, such a remote control will always have too many buttons to chose from at a given point in time, but too few buttons to be flexible enough to accommodate every situation, especially with applications developed by third-party developers. With our interaction model, the user is presented with controls that are customized for each possible interaction with the applications: a button with a label in order to request an action, a textfield to enter some text, or a list of items from which he or she chooses an item. Moreover, if a user is controlling the television from his or her own mobile devices, these controls are the native controls of this mobile device, with which the user is already familiar.

This prototype is intended to be open to third-party developers. As a result, some tools have been developed to make the development of applications and the description of their interfaces easier. In particular, every programmer who is already comfortable with developing Qt applications should be able to quickly learn how to program for our prototype. Indeed, our prototype's applications are based on the Qt framework, and great care has been taken in developing the interaction mechanism between the applications and the television system so that a Qt application and an application for our television are very similar in nature.

Finally, the evaluation of this prototype has shown that the system is quite reactive, since the majority of user interactions are processed and a visual feedback is provided for them in less than an arbitrary delay of 66 ms. However, these results are heavily dependent on the characteristics of the LAN to which the television and the remote controller are connected.

8.2 Future work

Given the time constraints of this thesis project, the prototype that was developed only supports a limited set of functionality. A lot of work remains to be done in order to transform this prototype into a viable finished product. In particular, *cross-cutting* programming aspects such as security and overall stability of the system were not really treated in this project, even though they will be indispensable in the finished product.

The remainder of this section is separated into three subsections, a description of future work to be performed on each subproject: the television system (subsection 8.2.1), the applications (subsection 8.2.2), and the remote controls (subsection 8.2.3).

8.2.1 Remaining work on the television system

Even though the core modules of the television system should remain quite stable in the future, several improvements should be made to the system as a whole. The most important amelioration to be added is a way to select which application will be heard at a given time. For the moment, the first application that needs to emit audio will do so, and subsequent applications that need to emit audio as well cannot. While this might seem to be a good thing that only one application can emit sound at any given time, the user should be able to change which application he or she wishes to hear, which is not possible for the moment. Moreover, applications that emit sound notifications should be able to emit those sounds at the same time that another application's sound is heard. It might be interesting to utilize spatialized audio to enable the user to hear multiple audio streams at the same time while helping him or her locate which application is the source of a given audio stream.

The prototype will also be able to acquire several external audio/video streams (e.g. from a gaming console) and integrate these streams within the user interface of the television. This requires additional HDMI ports on the current system, as well as a dedicated module to handle these external streams. Such an improvement will allow the user to use an external device at the same time as other applications are running on the television.

Modules should be developed to centralize the handling of commonly recurring actions, and to provide a notification system for applications to profit from these actions. For example, such a module would detect that a user has plugged an USB stick into the television, and will notify running applications of this fact, in case these applications might want to make use of it.

8.2.2 Remaining work on the applications

In order to facilitate the application development for third-party developers, the current SDK must be improved. A graphical tool for creating and testing application interfaces will be developed, and we could propose an integrated development environment, in order to make application development for our system as easy and straightforward as possible.

More out-of-the-box GUI components will be added to the applications' interfaces, and the currently available components will be improved, in order to make them more flexible. Indeed, the majority of these components are static for the moment, and cannot be modified after they have been defined in the description. However, it will be possible to modify them from the application code in a future version of the SDK. For example this would allow changing a label on a button or dynamically adding items to a list.

Finally, a mechanism needs to be provided for installing applications on the system, so that users will be able to easily install additional applications on their television. This installation process will likely include a list of authorizations that an application must request in order to perform potentially dangerous and/or invasive actions (as is already done in the mobile application markets).

8.2.3 Remaining work on the remote controls

Even though the remote control application for the iPhone is complete with respect to the current functionality of the television system, the included remote control is far from being able to control the television. A lot of effort remains to be done on this device, in order to bring it to the level of the iPhone application. Additional remote control mobile applications must also be developed, for the Android smartphones, Android tablets, and iPad tablets. However, the iPhone application was sufficient for this thesis project to illustrate and evaluate the interaction system proposed by our prototype.

8.3 Required reflections

This project thesis major proposal is a novel interaction mechanism with a smart television. Since the number of smart televisions across typical households is expected to grow much larger in the next few years, such a proposal may have an important impact if used by the industry's major constructors. The general conclusion is that controlling such a widely available device should be as easy as possible, even for novice users. This would lead to a larger and quicker adoption of "advanced" features of smart televisions (i.e., features related to their IP capabilities), opening economical opportunities with additional platforms from which developers may propose content. This would also lead to an improvement of the user experience in utilizing smart televisions. A standardization of this interaction model may be needed in order to facilitate the adoption of this model, resulting in a technology that is more interesting for developers to learn, as it would allow them to produce applications that may run on several different smart televisions without additional efforts on their part.

Environmental and ethical aspects will have to be considered later for this project. Care will have to be taken in ensuring that the television's power consumption remains low, both while the television is running and while it is sleeping. European guidelines for energy labeling will provide a guideline in determining which energy consumption thresholds should be respected by the finished product. Ethical aspects (together with security concerns) will also be very important for the finished product, especially considering that a lot of personal user information may flow towards the television (if chosen by the user). As a result, we must ensure that manipulation of this information respects the user privacy, even facing malicious activities.

Bibliography

[1] M. Pagani. *Multimedia and Interactive Digital TV: Managing the Opportunities Created by Digital Convergence*. Irm Press, 2003.

[2] Xbmc. `http://xbmc.org/about/`. [Online, accessed September 2012].

[3] European Commission. Commission Delegated Regulation (EU) supplementing Directive 2010/30/EU of the European Parliament and of the Council with regard to energy labelling of televisions, 2010.

[4] Andrew D. Birrell. An introduction to programming with threads. Technical report, Digital Equipment Corporation Systems Research, 1989.

[5] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. *ACM SIGOPS Operating Systems Review*, 25(5):122–136, October 1991.

[6] E.A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[7] Stephen Ferg. *Event-Driven Programming: Introduction, Tutorial, History*. February 2006.

[8] John Ousterhout. Why threads are a bad idea (for most purposes). Usenix Annual Technical Conference, January 1996.

[9] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM Press, 2002.

[10] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[11] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, April 1979.

[12] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *ACM SIGPLAN Notices*, 42(6):189–199, June 2007.

[13] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462. IEEE Comput. Soc., December 2004.

[14] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. In *Computer Networks 38*, page 393–422. Elsevier, 2002.

[15] George Oikonomou and Iain Phillips. Experiences from porting the Contiki operating system to a popular hardware platform. In *Proceedings of the 2011 International Conference on Distributed Computing in Sensor Systems and Workshops*, pages 1–6. IEEE, June 2011.

[16] Alexandru Stan. *Porting the Core of the Contiki operating system to the TelosB and MicaZ platforms*. Bachelor thesis, International University, Bremen, 2007.

[17] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. In Werner Weber, Jan M. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer-Verlag, Berlin/Heidelberg, 2004.

[18] D.R. Engler and M.F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, Orcas Island, May 1995. IEEE Comput. Soc. Press.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, December 1995.

[20] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, November 2006.

[21] Miro Samek. Protothreads versus state machines. `http://embeddedgurus.com/state-space/2011/06/protothreads-versus-state-machines/`, 2011. [Online, accessed August 2012].

[22] Frank Maker and Yu-Hsuan Chan. A Survey on Android vs. Linux. Technical report, University of California, Davis, University of California, Davis, California, USA, March 2009.

[23] Dominique A. Heger. Mobile Devices - An Introduction to the Android Operating Environment - Design, Architecture, and Performance Implications. Technical report, DHTechnologies, 2011.

[24] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles SOSP '97*, pages 38–51, Saint Malo, France, 1997. ACM.

[25] Luigi Rizzo. Revisiting Network I/O APIs: The netmap Framework. *Queue*, 10(1):30:30–30:39, January 2012.

[26] Richard Lindsay-Davies. Connected TV in the UK: State of play and forecast. In *EPRA – Cullen International Workshop*, Brussels, Belgium, May 2012. Digital TV Group.

[27] Ben Hookway. Developing for TV: crossing the chasm between screens. `http://www.visionmobile.com/blog/2012/04/developing-for-tv-crossing-the-chasm-between-screens/`, April 2012. [Online, accessed August 2012].

[28] Zeebox. `http://zeebox.com/uk/about-us`. [Online, accessed August 2012].

[29] Getglue. `http://getglue.com/about`. [Online, accessed August 2012].

[30] Janna Anderson and Lee Rainie. The Future of Smart Systems. Technical report, Pew Internet & American Life Project, June 2012.

[31] David Pierce. Buying a set-top box: everything you need to know. www.theverge.com/2012/5/17/2997361/buying-set-top-box-buying-guide, May 2012. [Online, accessed August 2012].

[32] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems Into Modules. In *Classics in software engineering*, pages 139–150. Edward Nash Yourdon, yourdon press edition, 1979.

[33] Brad A. Myers. Window Interfaces: A Taxonomy of Window Manager User Interfaces. *IEEE Computer Graphics*, pages 65—84, 1988.

[34] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[35] James Gettys, Robert W. Scheifler, Chuck Adams, Vania Joloboff, Hideki Hiura, Bill McMahon, Ron Newman, Al Tabayoyon, Glenn Widener, and Shigeru Yamada. *Xlib - C Language X Interface*, volume X Version 11, Release 7.7 of *X Consortium Standard*.

[36] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques SIGGRAPH '94*, pages 175–176. ACM Press, 1994.

[37] Jonathan Lemon. Kqueue, A Generic and Scalable Event Notification Facility. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 141–153, 2001.

[38] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of 6th Annual Linux Symposium*, volume 1, pages 217–228, July 2004.

[39] SQLite. http://www.sqlite.org/. [Online, accessed October 2012].

[40] M. Cotton, L. Vegoda, and D. Meyer. IANA Guidelines for IPv4 Multicast Address Assignments, BCP 51, RFC 5771, March 2010.

[41] D. Meyer. Administratively Scoped IP Multicast, BCP 23, RFC 2365, July 1998.

[42] Protocol buffers. https://developers.google.com/protocol-buffers/. [Online, accessed October 2012].

[43] The Qt Framework. http://qt.digia.com/Product/. [Online, accessed November 2012].

[44] Phonon library. http://phonon.kde.org/. [Online, accessed November 2012].

[45] VideoLAN. VLC. http://www.videolan.org/vlc/. [Online, accessed November 2012].

[46] Joseph M. Jacobson and Barrett Comiskey. US Patent 5930026 - Nonemissive displays and piezoelectric power supplies therefor, July 1999.

[47] E Ink. E ink triton imaging film technical specifications. http://www.eink.com/sell_sheets/triton%20sell%20sheet.pdf. Online, accessed September 2012.

[48] Qualcomm. mirasol displays - imod technology overview. http://www.qualcomm.com/media/documents/files/technology-overview-whitepaper.pdf, July 2011. Online, accessed September 2012.

[49] John Boxall. Getting Started and Moving Forward with Arduino. http://tronixstuff.wordpress.com/tutorials/. [Online, accessed December 2012].

[50] Arduino learning guide. http://arduino.cc/en/Tutorial/HomePage. [Online, accessed December 2012].

[51] Henning Karlsen. UTFT library. http://www.henningkarlsen.com/electronics/library.php?id=51. [Online, accessed December 2012].

[52] Henning Karlsen. UTouch library. http://www.henningkarlsen.com/electronics/library.php?id=55. [Online, accessed December 2012].

[53] Blocks Programming Topics. http://developer.apple.com/library/ios/#documentation/cocoa/Conceptual/Blocks/Articles/00_Introduction.html. [Online, accessed November 2012].

TRITA-ICT-EX-2013:32