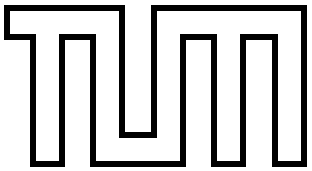


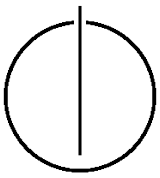
FAKULTÄT FÜR INFORMATIK

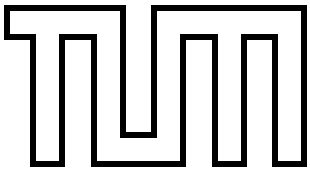
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Cryogenic
Enabling Power-Aware Applications on Linux

Alejandra Morales Ruiz





FAKULTÄT FÜR INFORMATIK

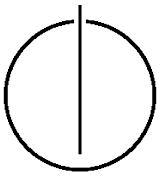
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Cryogenic
Enabling Power-Aware Applications on Linux

Cryogenic
Ein Linux Kernel-Modul für Kooperatives Energiesparen

Author:	Alejandra Morales Ruiz
Supervisor:	Dr. Christian Grothoff
Date:	February 17, 2014



Ich versichere, dass ich dieses Master-Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, February 17, 2014 Alejandra Morales Ruiz

Acknowledgments

I want to thank Christian Grothoff for giving me the opportunity to write this thesis as well as for the support and advice given throughout its completion.

I also thank the people at the Chair for Robotics and Embedded Systems, especially Reinhard Lafrenz and Steffen Wittmeier, who allowed me to access their laboratory and provided me with the necessary equipment to perform the energy measurements.

Thanks to Danny Hughes and Wilfried Daniels, from the *Katholieke Universiteit Leuven*, for their advice and contributions to the experimentation and the subsequent results of this work.

I would also like to express my gratitude to the whole community of Linux developers for sharing their knowledge and experience on the Internet, which has helped me not only during this thesis, but during all my studies.

Finally, I would like to thank my parents and brothers, who always supported and encouraged me to finish my studies abroad, and my partner, Ángel, because this thesis would have never been possible without him.

Abstract

As a means of reducing power consumption, hardware devices are capable to enter into sleep-states that have low power consumption. Waking up from those states in order to return to work is typically a rather energy-intensive activity. Some existing applications have non-urgent tasks that currently force hardware to wake up needlessly or prevent it from going to sleep. It would be better if such non-urgent activities could be scheduled to execute when the respective devices are active to maximize the duration of sleep-states. This requires cooperation between applications and the kernel in order to determine when the execution of a task will not be expensive in terms of power consumption.

This work presents the design and implementation of Cryogenic, a POSIX-compatible API that enables clustering tasks based on the hardware activity state. Specifically, Cryogenic's API allows applications to defer their execution until other tasks use the device they want to use. As a result, two actions that contribute to reduce the device energy consumption are achieved: reduce the number of hardware wake-ups and maximize the idle periods.

The energy measurements enacted at the end of this thesis demonstrate that, for the specific setup and conditions present during our experimentation, Cryogenic is capable to achieve savings between 1% and 10% for a USB WiFi device.

Although we ideally target mobile platforms, Cryogenic has been developed by means a new Linux module that integrates with the existing POSIX event loop system calls. This allows to use Cryogenic on many different platforms as long as they use a GNU/Linux distribution as the main operating system. An evidence of this can be found in this thesis, where we demonstrate the power savings on a single-board computer.

Contents

1. Introduction

1.1. Motivation

Energy represents a critical resource nowadays for many kind of devices, as it may set limitations in their operation. Depending on their characteristics, these limitations are different for every kind of device or system.

For mobile devices the limitation we come across is lifetime. The usage of smartphones and tablets has substantially grown over the last years, and this growth comes along with a higher performance demand. Processors are more and more powerful and internal memories have more capacity, which allow users to install a large number of applications on their devices. Some of these applications consume energy even after if the user is not actively interacting with them, as they could continue to perform tasks in the background.

An example of this behaviour is Google Latitude. This application was a feature of Google Maps that allowed users to update their current location and share it with their contacts. The task ran in background in order to automatically update all positions, without the necessity to specifically request the synchronization. This way users were able to see their own location and the location of their friends within Google Maps.

The goal of this work is to allow application developers to enable such features while minimizing the resulting increase in power consumption. This thesis will describe mechanisms for applications to manage the impact of background tasks on power consumption.

1.2. Contribution

Modern hardware devices have the capability to enter into sleep-states after a period of idleness as a means of reducing power consumption [11]. Naturally, resuming the activity of suspended devices takes time and energy as well [7, 10, 14]. In this thesis, we are concerned with the energy consumption of background tasks that prevent hardware from suspending or even force hardware to resume from suspension.

The contribution of this work is a mechanism that permits developers to implement power-aware applications based on the hardware activity state. Such power-awareness is achieved through the enabling the deferment of non-urgent tasks until other applications make use of the same device they need to use.

As a result, these tasks will wait until the corresponding device is already active and they may be optimistically executed in a clustered way. This behaviour is the central idea of Cryogenic and the key to reach our goal: maximize the idle periods of every single device and reduce the number of hardware wake-ups. Thus, we enable developers to balance between the responsiveness of an application and the amount of energy it consumes.

For that purpose, we have created a Linux kernel module that serves as a simple API for the developer of power-aware applications. With the Cryogenic module we achieve two

main objectives:

- *Ease of installation*: In order to use Cryogenic, a Linux user only needs to install our new module following the usual procedure. There is no need to rebuild the kernel during the installation neither during updates. To benefit from Cryogenic, the user will still have to use applications that have been implemented using its API; however, existing legacy applications will continue to run without problems.
- *Application migration*: The Cryogenic API makes it easy for developers to defer background tasks to a time where they will not have a significant impact on power management operations by the hardware. The developer simply needs to decide which tasks are non-urgent, and add a few lines of code in order to make use of Cryogenic's API and permit the deferment of these tasks.

1.3. Document organization

This thesis is organized in chapters as follows: in Chapter 2 we will discuss about other existing mechanisms thought to reduce power consumption. In Chapter 3 we will detail Cryogenic's design and implementation and introduce the usage of the resulting API. We will also describe the integration of Cryogenic in an existing software. In chapter 4 we will describe the methodology used to perform the energy measurements and present the results of our tests. Finally, in Chapter 5, we will present the conclusions and future steps of this work.

2. Related work

In this chapter we will discuss a set of existing mechanisms whose design and implementation are focused on achieving the same goal as Cryogenic, that is to say, reduce power consumption. This section will allow us to see that there are many considerations to take into account when talking about power consumption in a system, since each one of these mechanisms intends to save energy in a different fashion.

2.1. Cinder

Cinder [15, 16] is an operating system designed to fulfill some specific power-related needs of modern mobile devices. Based on HiStar, it allows to track how applications make use of energy and provides command over this usage. In this section we will explain the main characteristics of Cinder's design, starting with an overview of HiStar, the operating system Cinder is built on. We will then discuss its main operational features and compare it to Cryogenic's design.

2.1.1. HiStar

The HiStar [17] operating system is designed to minimize the amount of code that must be trusted. It controls how information flows through the system by means of a label mechanism that allows users to specify fine-grained security policies.

The HiStar kernel is organized around six first-class objects: *segments*, *threads*, *address spaces*, *devices*, *containers*, and *gates*. Whereas the former four objects are similar to conventional kernels, the latter two add new protection features. Containers provide hierarchical control over object allocation and deallocation by holding hard links to objects. Once an object has no path from it to the root container it is deallocated and garbage collected. The root container is a designated one that can never be deallocated. Gates provide protected control transfer, allowing a thread to jump to a named point in another address space. The rest of the kernel abstractions (files, processes, etc.) are built from theses objects.

Among other fields, every object has an immutable *label* that determines which threads can observe and which can modify the object. A label consists of a set of *categories* owned by threads. Threads can create new read or write categories and place them in the labels of the objects they create. Then, to observe an object, a thread must own all the read categories in an object's label and, likewise, it must own all the write categories in the label if it wants to write the object. Threads can also grant ownership of any category to other threads, so as to share privilege. Apart from these permissions there exist a third option that allows threads to read an object as long as they do not communicate with the network or other processes.

By means of containers HiStar provides a way to account and revoke storage resources. This hierarchy does not suffice to deal with resources where control over consumption

rate is as important as quantity, which is the case of energy. This shortcoming is thus the motivation that led to the creation of two new kernel objects: *reserves* and *taps*. Both objects are presented in the following section.

2.1.2. Design of Cinder

A *reserve* represents a right to use a given quantity of energy. All threads are associated with a reserve from which they draw energy and, like all other kernel objects, every reserve has a label that controls which threads can manipulate it. Once an application has consumed an amount of energy, this quantity is subtracted from the corresponding reserve, and the kernel ensures that any application does not perform actions for which its reserve does not have enough energy. Reserves allow threads to subdivide their available energy in order to delegate it to other threads and they track the energy consumed from them by applications. This provides accounting information and permits applications to be made power-concerned.

A *tap* is an special-purpose thread whose job is to transfer energy between reserves. While reserves provide quantities of energy that may be consumed by threads, taps control the rate at which these quantities could be consumed. A tap is composed by a rate, a source and a sink reserve, and a label that determines the privileges needed to transfer energy from the source to the sink reserve.

Reserves and taps therefore form a graph of energy consumption rights. The system battery is represented as the root reserve, whose energy is subdivided and given to the rest of the reserves.

Any application may have the possibility of consuming energy at a high rate through the association with a reserve fed by a high rate tap. This can create a problem if the application draws less energy than provided by the tap, since the reserve will fill with energy that other applications will not be able to use. In order to solve this situation, it is possible for reserves to get unused resources back by means of proportional taps. This way, a reserve being fed by a tap has a proportional tap in the opposite direction that transfers a portion of its energy back to the parent reserve per unit time.

These proportional taps do not prevent applications from hoarding energy, as a thread could create a new reserve without proportional taps and transfer energy to it, accumulating over time enough energy to starve the rest of the system. Cinder prevents hoarding by imposing a decay of sources across all reserves. In other words, every reserve has an implicit proportional backwards tap to the battery. The decay is computed using a half-life, returning to the battery the 50% of energy in each reserve every half-life period of time.

Threads are also able to share their resources in order to avoid resource inversion, where a thread with enough energy in its reserve cannot run because a thread holding a lock has run out of energy. In this case, the thread blocked on the lock can donate energy to the thread holding it, permitting the former to run.

2.1.3. Discussion

Cinder design is thought to provide three basic mechanisms to perform energy management: *isolation*, *delegation* and *subdivision*. *Isolation* is the mechanism that prevents applications from consuming inordinate amounts of energy or starving other applications. *Delega-*

tion allows an application to lend its energy to other applications, making both, the donor and the recipient, able to consume the delegated resources. *Subdivision* allows any application to partition its available energy. Combining the three mechanisms, Cinder allows an application to give another principal a partial share of its energy, being totally sure that the rest will remain for its own use. This idea is the basis of Cinder's operation.

There also exist specific situations where Cinder has hardware-related considerations. Hungry-devices may have high initial energy costs. For example, experiments have shown that the overhead of powering the radio up on a mobile phone dominates the total amount of energy consumed for flows lasting less than 10 seconds. Cinder can solve this problem with reserves and taps. The networking stack (netd) has a reserve that stores energy to power the radio up. Any network system call made when the device is off blocks if the sum of netd's reserve and its own one is not enough to power it on. Blocked threads delegate a share of their energy to the netd's reserve and once there is enough energy to power the radio on, Cinder permits the threads to proceed. When the device is on, additional operations are billed in proportion to the active periods.

This is similar to the main idea of Cryogenic's design: to cluster tasks that want to make use of the same device. We can see then that, due to its delegation mechanism, Cinder could imitate the behaviour of Cryogenic with other devices, creating specific reserves that save energy to wake them up. It could even distinguish between urgent and non-urgent assigning lower rate taps to those that are non-urgent.

Cinder also worries about the user's experience and expectations, since it accounts the available energy and is able to ration it depending on this amount and the user's desired performance or expected lifetime. For example, Cinder could manage to make background applications consume less energy than foreground applications, or divert energy to an specific application that the user considers more priority when battery life is low. Cryogenic does not bear in mind such considerations, and only defers the execution of tasks that are considered non-urgent by the application's developer.

Regarding the implementation, we believe it is important to point the different efforts needed to implement both systems. Cinder is an operating system built on the top of another operating system. As explained in previous sections, an extension of HiStar had to be implemented to create reserves and taps, and this implies modifying some parts of the existing kernel as well, in order to integrate these new created objects. This represents a higher amount of work than Cryogenic, which is an extension of the Linux kernel implemented by means of a kernel module. Thus, all the new code is added on the module and there is no need to modify the existing kernel. Furthermore, any Linux user willing to use Cryogenic only has to install the new module, without the need to install a different operating system.

The migration of applications represents a quite different effort for each mechanism as well. In the case of Cryogenic, the needed modifications are simple and incremental. The developer must decide which are the non-urgent tasks and wrap their execution with Cryogenic's API. This means just adding a few lines of code around the code that executes these tasks. In the case of Cinder, the developer has to adapt the application to a new operating system and design the reserve and the tap for the application as well as for every single task he wants to assign an isolated budget of energy to. This could be difficult work since the work of reserves and taps must be consistent in order to avoid problems presented in the previous section, like hoarding energy or resource inversion.

To sum up, it is easy to notice that, although similar behaviours could be achieved with these mechanisms, Cinder and Cryogenic approaches are different. Whereas Cinder bases its ability to control energy consumption on fairly rationing the actual existing resources (the battery) among applications, Cryogenic does not address the issue of the amount of available power nor how much energy an application consumes. Its only concern is on when devices must be used in order to reduce energy consumption by maximizing the duration of idle periods. Roughly speaking, we could say that Cinder cares about how much energy an application consumes, that is to say, accounting; Cryogenic instead cares about saving or, in other words, creating opportunities for reduced consumption.

2.2. big.LITTLE Processing

big.LITTLE Processing [5, 9, 13] is an energy saving mechanism for mobile platforms developed by ARM[1]. With this mechanism different types of CPU in terms of performance and energy efficiency are paired together and software execution is dynamically transitioned to the appropriate CPU depending on performance needs. big.LITTLE intends to take advantage of the different usage patterns observed on smartphones and tablets: periods of high processing activity, such as gaming or web browsing, alternate with longer periods of low intensity tasks, like e-mail or audio. We will present the main hardware characteristics and the existing software transition methods. We will finally present the similarities and differences with Cryogenic.

2.2.1. Hardware design

The key of big.LITTLE processing is the combination of two processors, the big and the LITTLE ones, that are architecturally identical. Therefore, all instructions will execute in an architecturally consistent way whether they do it on the big or on the LITTLE processor,but with different performance.

The significant difference in energy consumption between both processors are due to differences in their micro-architectures. The main difference is that the LITTLE processor has a pipeline length between 8 and 10 stages, whereas the pipeline of the big one has a length between 15 and 24 stages; this is critical as the energy consumed by the execution of an instruction is related to the number of pipeline stages it must traverse [12].

2.2.2. Software models

big.LITTLE Migration

In this model, the kernel scheduler is unaware of the big and LITTLE cores and it is a power management software that resides in the kernel space who controls the migration of software context between cores. When the LITTLE processor is executing at its highest operating point and more performance is demanded, a migration is executed and both, the operating system and the applications, move to the big processor.

[1] ARM Holdings: `www.arm.com`

There exist two types of migration: *CPU Migration* and *Cluster Migration*. In the *Cluster migration* the entire context is migrated from all LITTLE CPUs to the same number of big CPUs and vice versa. Therefore, only one cluster can be used at the same time. This is inefficient when the load on a single CPU is high, but the load on other CPUs in the cluster is low. This is solved with the *CPU migration*, where each LITTLE CPU is logically paired with a big CPU and the migration software can execute the context switch between CPUs to match the current performance demand. In this case, only one CPU per processor pair can be used at once. In both types of migration, processors or clusters that are not being used can be powered off.

big.LITTLE MP

In this case, the performance requirements of every task that is currently running is what determines whether a big processor must be powered on or not. If there are demanding tasks, a big processor is powered on to execute them while low tasks can keep executing on a LITTLE processor. As in the previous model, any processors that are not being used can be powered off.

MP then permits to run applications on the processing resource that is most appropriate to their requirements. This way, applications that require significant processing performance or have time critical results can be executed on the big processor, whereas applications that do not have these constraints can run on the LITTLE processor.

2.2.3. Discussion

The first characteristic of big.LITTLE Processing that draws our attention is that, in contrast to Cryogenic, it is a hardware-based mechanism to reduce power consumption. It is true that it has software-related issues, as a manager is needed to perform the OS and application transition between CPUs when using migration, and MP needs cooperation between the scheduler and processes to assign the execution of tasks to the most appropriate CPU; but in the end any user must acquire a device with this type of processor to obtain power saving benefits.

The deployment complexity of big.LITTLE is different depending on the software model chosen. With migration, neither the operating system nor applications require any modifications or changes to adapt to the system, and any application is able to migrate its execution to the big or LITTLE processors. When using MP, extra-work is required: the kernel scheduler needs changes to fully support big.LITTLE, since multiprocessor support for the Linux kernel assumes that all processors have the same performance capabilities. Hence, Cryogenic's deployment needs a greater deal of effort compared to big.LITTLE's deployment with migration, but the effort is similar when the model is MP because of the necessity of both mechanisms to extend the kernel operation.

Despite these differences, we could consider that both mechanisms address a similar issue, especially when using MP as the software model for big.LITTLE. With MP, the scheduler assigns the execution of tasks that do not have heavy processing requirements to the LITTLE processor, consuming as a result less energy to run them. Similarly, Cryogenic defers the execution of non-urgent tasks to prevent them from waking up devices that

are currently sleeping, reducing the energy consumed for their execution. Thus, both systems allow to determine a set of tasks that offer the chance to reduce energy consumption through their execution under special conditions.

The way these sets are determined is different for each method. In the case of Cryogenic, it is the application developer who must choose the non-urgent tasks that may defer their execution. In big.LITTLE instead, the real processing requirements determine whether an application can keep its execution on the LITTLE processor or it must migrate to the big one. We can see then that Cryogenic has a subjective aspect that could compromise the responsiveness of the system if the developer does not determine the urgent and non-urgent tasks in a proper way.

We also wonder if we could combine both mechanisms. At first glance we find it possible, since the OS and software migration does not interfere with Cryogenic operation. Nevertheless an undesired behaviour could occur with the big.LITTLE migration. Given a situation where Cryogenic forces many tasks to defer their execution, these tasks might be executed in a clustered way as long as they are waiting for the same device to be awake. This could provoke an extra-demand of processing performance and, consequently, a migration from the LITTLE to the big processor. Therefore, the energy saved due to Cryogenic may be consumed during the execution of the deferred tasks on the big processor and we would not obtain the expected benefit.

2.3. Windows 7 Kernel Improvements

The processor activity, especially the one related to periodic tasks from applications or drivers, has a great influence on the power consumption of a system. Modern processors enter into a low-power state during periods of idle activity between the execution of instructions, but many technologies need a minimum time of idleness to obtain actual power savings. If the idle period is too short, the power required to enter and exit the low-power state could be greater than the power saved. The following techniques are improvements introduced in the Windows 7 kernel with the aim of managing and reducing this power consumption that results from periodic software activity.

2.3.1. Timer Coalescing

The Windows kernel scheduler is driven by a timer interrupt platform that has a default period. On every of these timer interrupts the kernel checks whether scheduled timers have expired and, if so, it performs a callback to the function associated with the timer. Timer Coalescing [2, 3] allows applications and drivers to set a tolerable delay for the expiration of their scheduled timers. The kernel then uses this delay to adjust the time when the timer expires and makes it coincide with the expiration of other software timers. This behaviour is shown with two examples.

In Figure 2.1 we can see the regular non-coalescing behaviour. As already explained, on every timer interrupt the expiration of timers is checked and then the appropriate callbacks are performed. This is noticeable since a callback for every expired timer is issued right after the timer interrupt that follows the expiration.

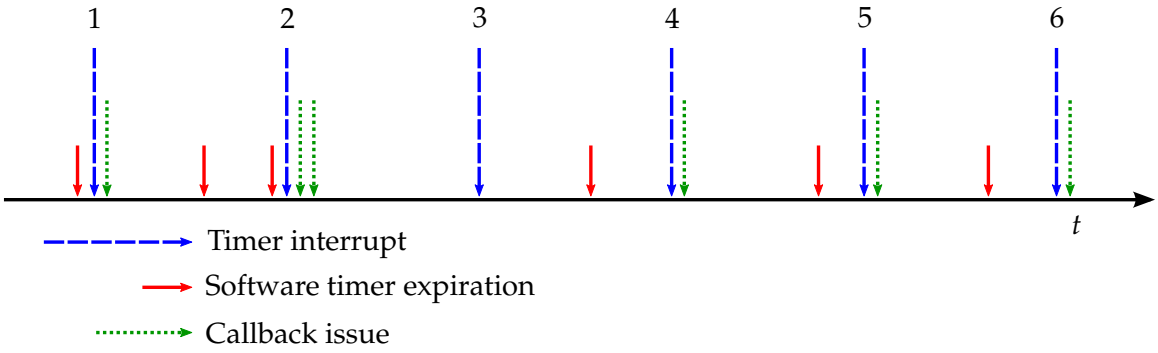


Figure 2.1.: Service of expired timers without coalescing

Figure 2.2 shows the benefits of Timer Coalescing. The expiration scheduled before the first timer interrupt, marked with dashes in the picture, has a delay that allows to defer it and, as a result, the service callback can be issued during the next timer interrupt along with two other callbacks. The same happens with the expiration prior to to the fourth timer interrupt, that can be served later so that it coincides with the service of another timer expiration.

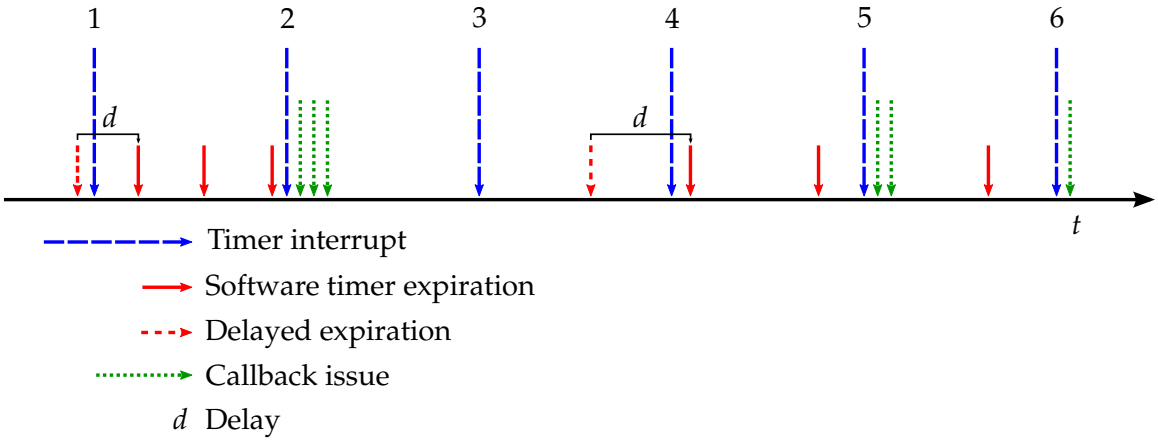


Figure 2.2.: Service of expired timers with coalescing

Since the callbacks are performed grouped into a single period of processing, the idle periods between interrupts become longer and the number of times the processor must exit and enter the low-power state is reduced. This way, the overall system power consumption is reduced.

Developers can take advantage of Timer Coalescing in different ways depending on whether they are developing an application or a driver. For applications, a new user-mode function is provided which allows to set the period when a timer should periodically expire as well as its tolerable delay[2]. Similarly, a new kernel-mode function that enables to set these fields is provided for drivers[3]. Developers can also specify a tolerable delay for

[2] http://msdn.microsoft.com/en-us/library/windows/desktop/dd405521(v=vs.85).aspx
[3] http://msdn.microsoft.com/en-us/library/windows/hardware/ff553249(v=vs.85).aspx

their periodic timers when they use the Windows Driver Framework[4]. In this case it is not a function but a field on a structure what they must use to set it[5].

2.3.2. Intelligent Timer Tick Distribution

In systems with multiple logical processors, the timer interrupts are mirrored on every processor and then the callbacks for the corresponding expired timers are performed. The Intelligent Timer Tick Distribution (ITTD) [3] is a mechanism that reduces the amount of timer interrupts in this kind of systems. To achieve this, application processors are not woken up from low-power states unless software timers are expiring or hardware interrupts other than the timer interrupts occur. Application processors (AP) are any processors in the system that are not the base service processor (BSP).

Further examples illustrate the differences between the regular processor operation and the operation when using ITTD with and without coalescing. Figure 2.3 shows the normal behaviour on multiprocessor systems. Timer interrupts arrive to both processors and expired timers are served on the corresponding one.

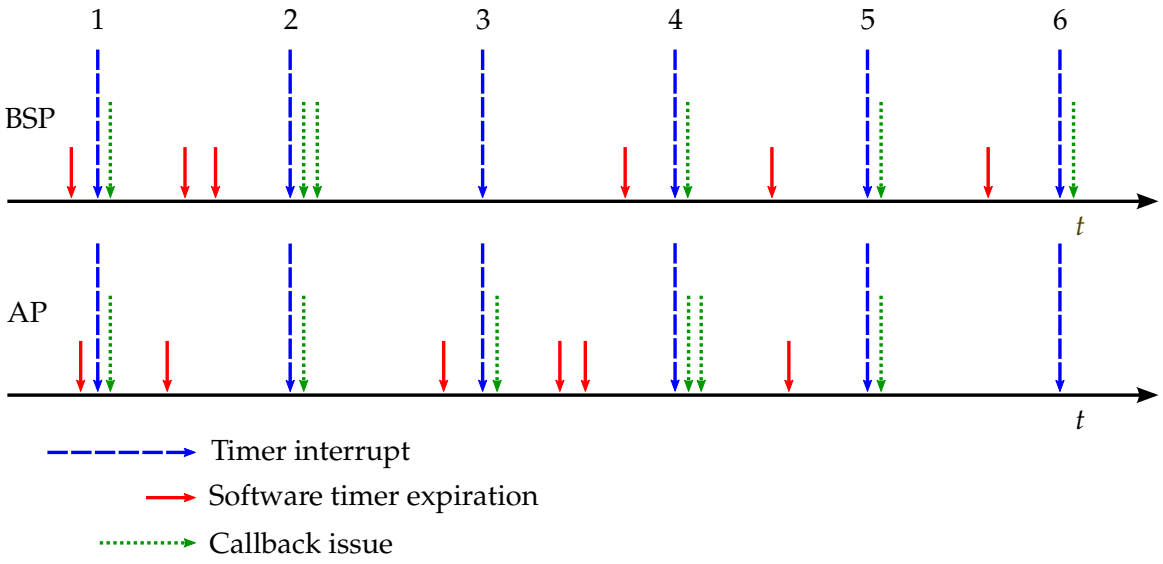


Figure 2.3.: Multiple logical processors without Coalescing nor ITTD

In Figure 2.4 we can see the behaviour of a system with ITTD. It is easy to notice that the only difference with the previous figure is the last timer interrupt on the AP, which in this case is not issued because there are no pending callbacks. This example illustrates that, although ITTD could be used on its own, their benefits may not be significant if expirations are not properly scheduled for this aim.

[4] http://msdn.microsoft.com/en-us/library/windows/hardware/ff557565(v=vs.85).aspx

[5] http://msdn.microsoft.com/en-us/library/windows/hardware/ff552519(v=vs.85).aspx

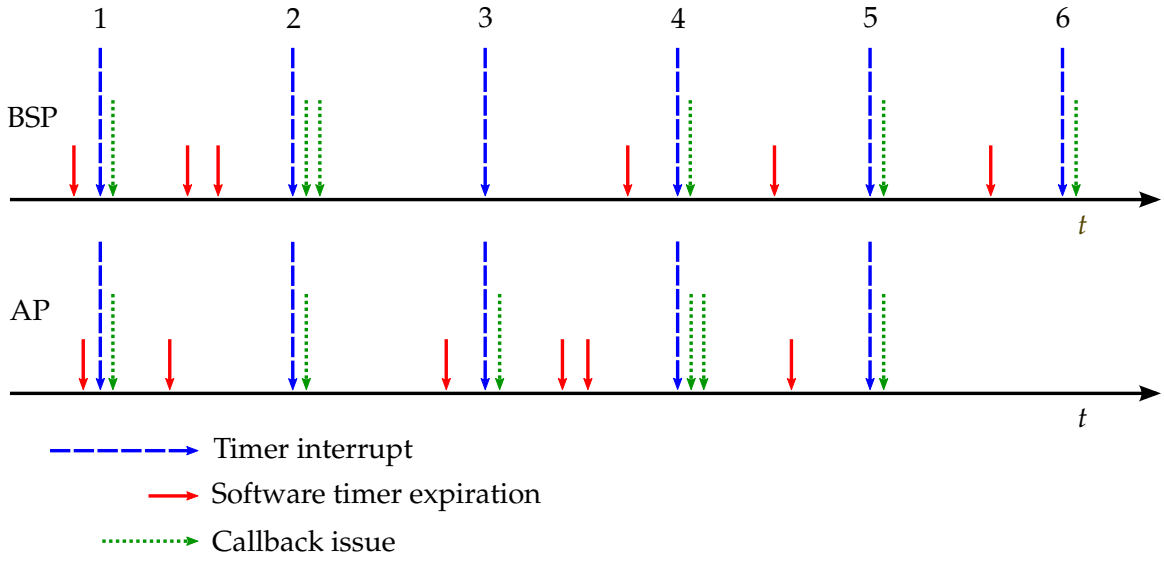


Figure 2.4.: Multiple logical processors with ITTD

The actual benefit of ITTD appears when it is used along with Timer Coalescing. After Timer Coalescing has grouped callback issues, it is more likely to find timer interrupts that have no work to do and thus ITTD can remove them. This is shown in Figure 2.5, where Timer Coalescing allows to cluster the execution of callbacks and ITTD disables three timer interrupts with no expired timers to serve: the second, the fourth and the sixth.

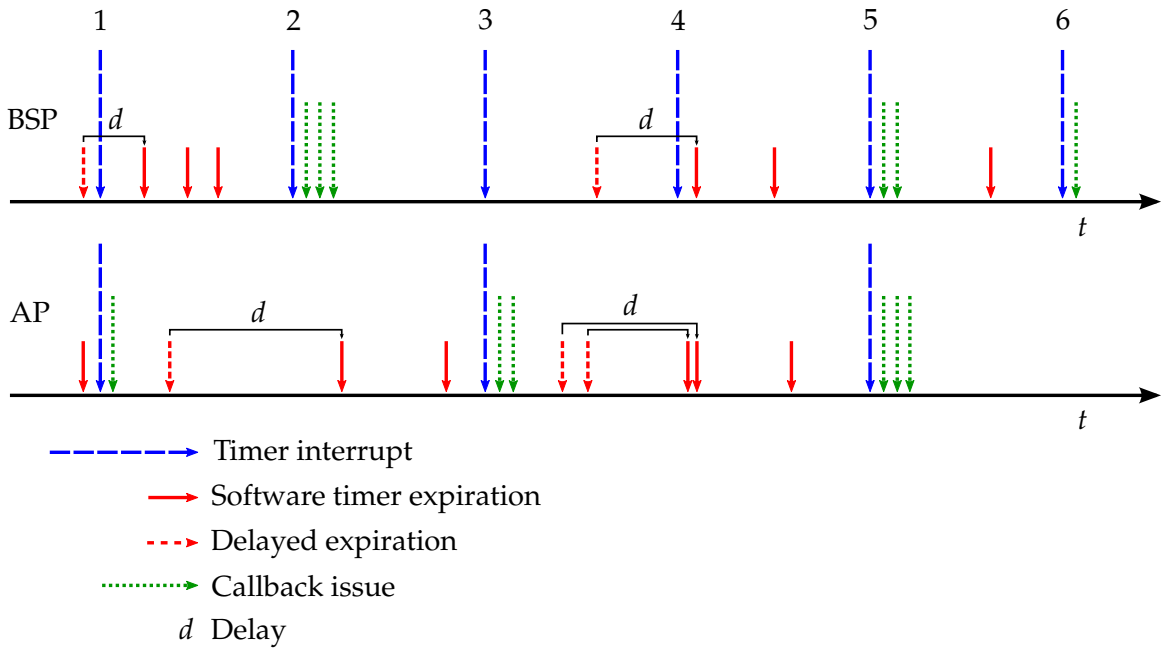


Figure 2.5.: Multiple logical processors with Coalescing and ITTD

The final results for the BSP are identical to the example of the previous section since

ITTD does not affect its operation, but for APs the difference is significant, as the idle periods may now be longer than the default timer interrupt platform period.

2.3.3. Discussion

Of all the mechanisms we have written about in this chapter, these improvements made in the Windows 7 kernel are the most similar to Cryogenic. The fundamentals are the same: try to have the longest idle periods as possible so that the number of hardware wake-ups is reduced. Nevertheless, the scope of Cryogenic is different: it focuses on devices like hard disks and network cards instead of processors.

Although coalescing is not the final goal of Cryogenic, it is a behaviour likely to happen, since a set of tasks that are waiting for the same device may be executed in a clustered way once the device is woken up. In fact, this would be a rather desirable situation and an evidence of good system performance in terms of power consumption.

The deployment work is similar for both systems. From the OS point of view, Cryogenic needs a new module to extend the kernel operation and support the new features. The Windows 7 kernel also needed modifications, since new functions must be provided to support Timer Coalescing and the timer interrupt platform must be able not to issue interrupts when there are no pending callbacks, which is the effect of ITTD. From the developer point of view, in both cases they have to adapt their applications to the new APIs by adding or modifying some lines of code. In the case of ITTD, its operation is totally transparent to the developer and does not add any extra-work.

Both systems focus their efforts on a set of tasks that give the opportunity to save energy through their deferment: periodic events that may force hardware to resume its activity at a given time when their execution is not essential. Application developers of both platforms undertake to choose such tasks and set appropriate times for delays and timeouts so that the responsiveness of the system is not affected.

Despite of focusing on the Windows 7 kernel, there are other platforms that also use similar techniques to the ones explained in this section to save energy. For example, Apple introduced timer coalescing on Mavericks[6] in order to reduce background work while the machine is running on battery power [8]. The Linux Kernel introduced a new configuration parameter[7] as of version 2.6.21 that allowed CPUs in lower-power states to remain in this state longer. This method was named Tickless Kernel [4] and has effects similar to ITTD.

[6] OS X Mavericks: `http://www.apple.com/osx/preview/`
[7] NO_HZ: `http://lxr.hpcs.cs.tsukuba.ac.jp/linux/kernel/time/Kconfig`

3. Design & Implementation

In this chapter we present the design and implementation of Cryogenic. To start with, we introduce our approach and illustrate the behaviour of a system running Cryogenic. Next, we present the kernel module that supports Cryogenic's operation. Finally, we show the developers perspective and how they can use the resulting API to benefit from Cryogenic.

3.1. Approach

In Section 1.2 we presented the main goals of Cryogenic: reduce the number of hardware wake-ups so that the overhead power consumption they provoke is avoided and extend the duration of idle and sleep periods in order to reduce the overall power consumption. In this section, we illustrate with two examples the behaviour of a normal system and the achievement of these goals after running Cryogenic on the same scenario.

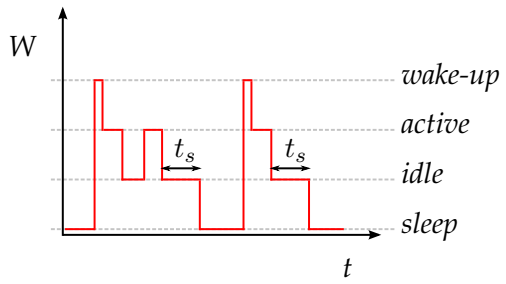


Figure 3.1.: Energy consumption model

The examples follow the model presented in Figure 3.1. Devices can operate in four states that differ in their energy consumption rate. These states are: *active*, *idle*, *sleep* and *wake-up*. When a device is actually working, it remains in the active state until it has completed its tasks. When this happens, the device enters the idle state, that has a lower consumption, and then two state transitions are possible: it can become active again if it is requested to perform more tasks or otherwise it enters the sleep state, which is the one with the lowest consumption rate. This can only happen if the device is idle enough time to reach the sleep timeout t_s. Once the device is sleeping it can be requested to work again and thus it must change from sleep to active. This transition is represented by means of the *wake-up* state, that has the highest consumption rate.

Although in this model the increase of consumption from one state to the next one with higher consumption is proportional for all transitions, this is only a representation and the real increases will heavily depend on every hardware device. Similarly, the sleep timeout may be different from one device to another.

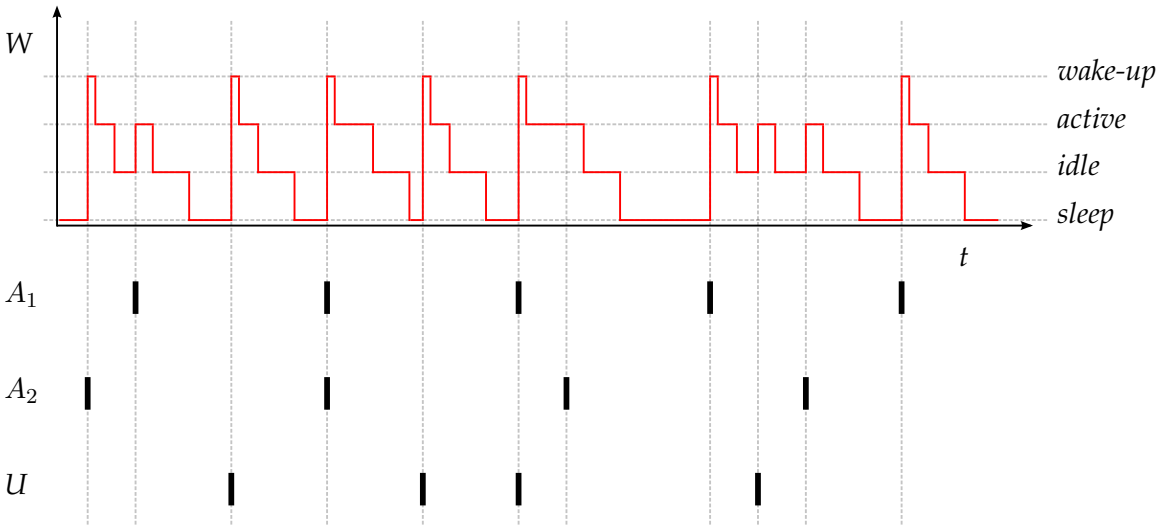


Figure 3.2.: Current execution

Figure 3.2 shows a possible scenario where Cryogenic would be beneficial. On the one hand, there are two applications, A_1 and A_2, that execute tasks periodically, each one with a different period. On the other hand, there are tasks executed due to the user interaction, U, that have no predictable pattern. Assuming that all tasks want to make use of the same device, e.g. the network card, the figure illustrates the resulting power consumption for this device. As we can see, every time tasks need to use the device when it is currently sleeping, there is an overhead power consumption caused by the transition from sleeping to active. Then comes the active period and once tasks are finished, the device becomes idle. In this example, the frequency of execution forces the device to switch from idle to active many times, preventing it from going to sleep.

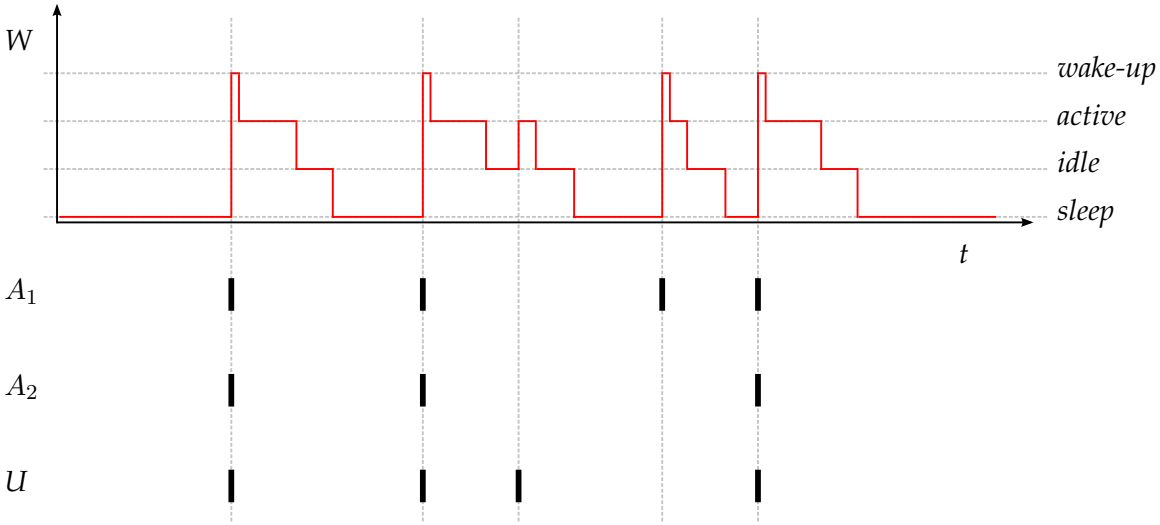


Figure 3.3.: Execution with Cryogenic

With Cryogenic, the execution of tasks that belong to A_1 and A_2 could be deferred until other tasks interact with the device. This way, the network card is already active when A_1 or A_2 need to use it and the peak consumption caused by the state transition is eliminated. Figure 3.3 illustrates this behaviour. Indeed we can notice that the number of wake-ups has been reduced from 7 to 4. We can also see that the user interaction, which obviously we can not manage, acts as the main trigger for waiting tasks. As a result, most tasks are executed right after the user's tasks in a clustered way and this allows sleep periods to become longer.

3.2. Architecture

The whole implementation of Cryogenic is embedded in a kernel module that works as a character device driver [1, Ch. 1]. When Cryogenic is loaded, a set of character devices is created and a subset of system calls is defined. The system calls handle the character devices through the device nodes created under `/dev/cryogenic/`. This is the API that developers will use later to manage the interaction between applications and hardware devices.

3.2.1. Device representation

Every device present in a system is represented in the kernel by means of the `struct device`[1], which is a low-level representation that contains basic information needed to build the device model [1, Ch. 14]. The `struct device` is usually embedded in other high-level structures that track additional information of each specific subsystem. This way, the SCSI subsystem provides the `struct scsi_device`[2] to represent its devices and the network subsystem represents its devices through the `struct net_device`[3]. Both structures have an instance of the `struct device` as one of their fields.

For character and block devices, a device node is created in the `/dev/` directory as well, and it is internally represented as an instance of the `struct cdev`[4].

Figure 3.4 is an scheme that illustrates which structures and device nodes would represent any hardware device allowed to operate under Cryogenic's management, which are storage devices attached to the SCSI bus and network devices, such as ethernet and wireless LAN cards. There are more structures involved in the device model, but these are the most important ones for the work done in this thesis.

[1] `http://lxr.free-electrons.com/source/include/linux/device.h`
[2] `http://lxr.free-electrons.com/source/include/scsi/scsi_device.h`
[3] `http://lxr.free-electrons.com/source/include/linux/netdevice.h`
[4] `http://lxr.free-electrons.com/source/include/linux/cdev.h`

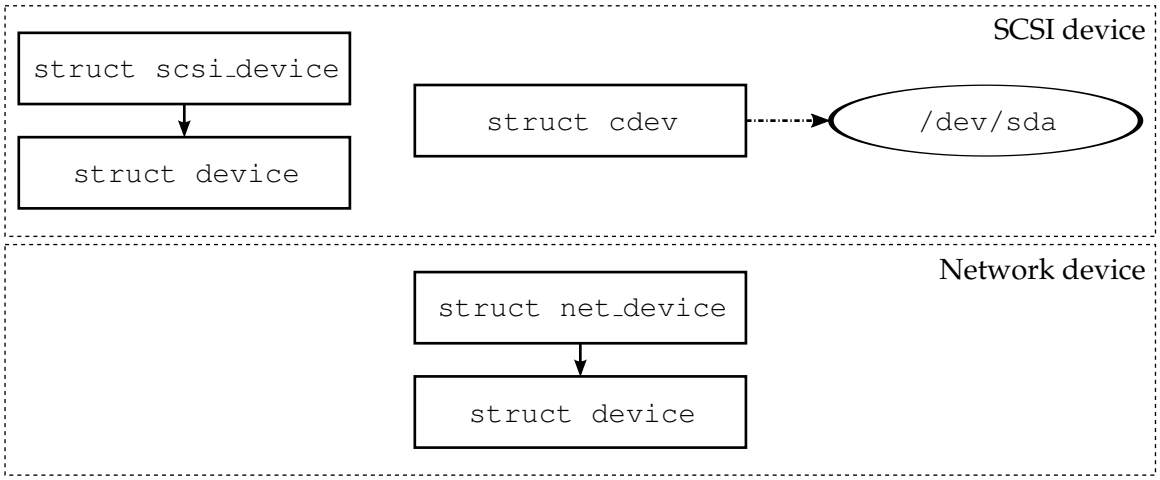


Figure 3.4.: Internal representation of SCSI and network devices

Structures are represented by boxes and device nodes are ellipses. A normal arrow between two boxes represents that the target one is embedded inside the origin. The dashed arrow between a box and an ellipse represents the relationship between a device node and the `struct cdev` that represents it.

We can notice that, since they are not block or character devices, network devices do not have any device node present in the filesystem.

Cryogenic creates a new character device for each of these hardware devices. The corresponding device nodes are put together under the new location `/dev/cryogenic/`. These new character devices will determine whether it is inexpensive or not that an application interacts with the hardware device they are associated to at a given time. The way they do this task will be presented later.

Cryogenic also defines a new structure for each hardware device. This structure keeps track of all information needed to hold its operation. Figure 3.5 illustrates the definition of `struct pm_device`.

```
1  struct pm_device {
2      int minor;
3      const char *name;
4      struct cdev pm_cdev;
5      struct device *dev;
6      wait_queue_head_t event_queue;
7      int unplugged;
8      request_fn_proc *scsi_request_fn_address;
9      unsigned char serial_number[MAX_SERIAL_NUMBER_SIZE];
10     int scsi_cdev_open;
11     struct net_device_ops my_ops;
12     const struct net_device_ops *old_ops;
13 };
```

Figure 3.5.: Definition of `struct pm_device`

Here is a brief explanation of every field of the structure. The work these fields do will be presented in the following sections.

`int minor`
: The minor number assigned to the character device `pm_cdev`.

`const char *name`
: A string that identifies the device. For SCSI devices, this field points to the device SCSI address[5]. For network devices, it points to the name the kernel gives by default to every network interface, e.g. *eth0*, *eth1*, *wlan0*, etc.

`struct cdev pm_cdev`
: Structure that represents the device node created under `/dev/cryogenic/`.

`struct device *dev`
: Pointer to the structure that represents the hardware device.

`wait_queue_head_t event_queue`
: Queue where deferred tasks will wait until they are allowed to proceed.

`unplugged`
: Flag that determines whether a device has been unplugged from the system.

The following fields are only used by SCSI devices.

`char serial_number[MAX_SERIAL_NUMBER_SIZE]`
: It is the device unique serial number assigned by the manufacturer. It has a maximum size of 20 bytes.

`request_fn_proc *scsi_request_fn_address`
: This field points to the `request_fn` function of the device. This function is called every time and I/O request must be performed on the device.

`int scsi_cdev_open`
: This counter indicates how many times the character device associated to this SCSI device is currently open.

The following fields are only used by network devices.

`const struct net_device_ops *old_ops`
: Pointer to the `struct net_dev_ops` of the device. This structure contains callbacks needed to manage a network interface.

`struct net_device_ops my_ops`
: An instance of `struct net_device_ops`.

[5] `http://www.netfibre.com/?p=392`

In Figure 3.6 we can see the representation of SCSI and network devices after Cryogenic has been loaded on the system.

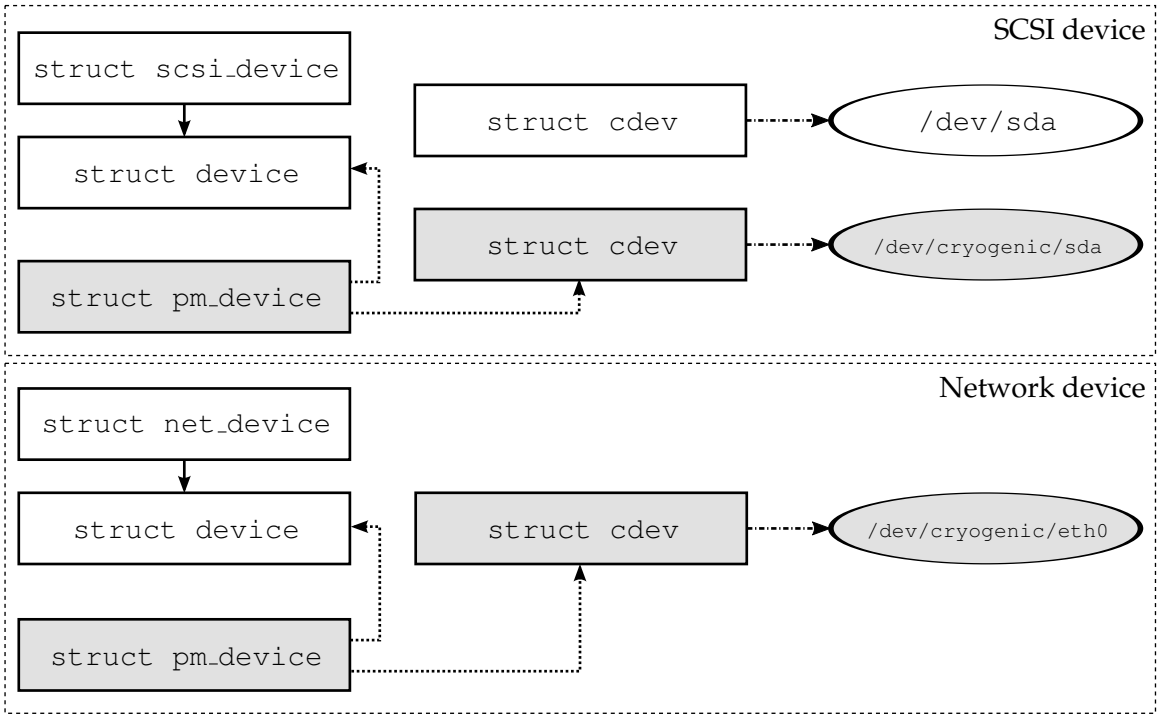


Figure 3.6.: Internal representation of SCSI and network devices after loading Cryogenic

The structures and nodes added by Cryogenic are the shadowed ones. A dashed arrow between two boxes means that the origin structure has a field that points to the target structure.

3.2.2. Device search

When Cryogenic is loaded, the first task that must be performed is the search of devices. For that purpose, the module first allocates a region of minor numbers [1, Ch. 3] between 0 and 9. Once this region has been allocated, there is no way to dynamically reallocate it, which means that a maximum number of 10 devices can be managed by Cryogenic at once.

The module then allocates enough memory to save information for the 10 possible devices and assigns it to the variable declared in Figure 3.7, which will store instances of the `struct pm_device`, presented in Section 3.2.1. Thus, this array is the main variable that keeps track of all devices whose usage is being managed by Cryogenic. At this point, the search has not been performed, so all the instances are empty and they are marked as free by setting the `minor` field to -1.

```
1 static struct pm_device *pm_devices;
```

Figure 3.7.: Declaration of `pm_devices`

Then comes the actual search, that starts with SCSI devices. Cryogenic looks over the SCSI bus, that is represented on the system as an instance of `struct bus_type`[6], and for every attached device whose driver is called "sd", creates the corresponding character device and fills a position of `pm_devices` with its information. Selecting only those devices managed by the sd driver is important, since there are other devices attached to the SCSI bus, e.g. DVD units, that we are not interested in.

The procedure to find network devices is similar to the SCSI case. The module looks over all the devices present on the `init_net`[7] namespace[8] and then selects only ethernet and wireless LAN devices. This time, the criteria to filter is the name of the device, which must be an string starting with "eth" or "wlan".

After this work is done, the new device nodes are already present in the directory `/dev/cryogenic/`. Nodes corresponding to SCSI devices are named after the device serial number. For network devices, the default interface name is used for the device node as well. Figure 3.8 shows the list of files in a system that has attached three hard drives, two ethernet cards and a wireless LAN card.

```
1 # ls -l /dev/cryogenic/
2 total 0
3 crw------- 1 root root 247, 0 Dec  9 16:03 9VP26KSV
4 crw------- 1 root root 247, 2 Dec  9 16:03 eth0
5 crw------- 1 root root 247, 3 Dec  9 16:03 eth1
6 crw------- 1 root root 247, 5 Dec  9 16:04 WD-WCAU46069319
7 crw------- 1 root root 247, 1 Dec  9 16:03 WD-WCAV90469334
8 crw------- 1 root root 247, 4 Dec  9 16:05 wlan0
```

Figure 3.8.: List of files in `/dev/cryogenic/`

In this example we can observe the major and minor numbers assigned to the devices. 247 is the major number, the same for all devices since it identifies the driver, i.e. the module. After the comma comes the minor number, that is different for each device and uniquely identifies them inside the module.

All these character devices belong to the class "cryogenic" that is created at loading time as well. A class is a high-level view of devices that abstracts low-level details and allows to group them depending on what they do [1, Ch. 14]. Classes are shown under `/sys/class/`. In Figure 3.9 we can see the list of files in the `/sys/class/cryogenic/` directory of the system presented in the previous example.

[6] `http://lxr.free-electrons.com/source/include/linux/device.h`
[7] `http://lxr.free-electrons.com/source/include/net/net_namespace.h`
[8] `http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/`

```
# ls -l /sys/class/cryogenic/
cryogenic!9VP26KSV
cryogenic!eth0
cryogenic!eth1
cryogenic!WD-WCAU46069319
cryogenic!WD-WCAV90469334
cryogenic!wlan0
```

Figure 3.9.: List of files in `/sys/class/cryogenic/`

If more than ten devices are attached to the system at loading time, Cryogenic will store information only for the ten former and will notify the user that the rest could not be added.

3.2.3. Hotplugging

Cryogenic is able to detect when a hardware device is plugged or unplugged and dynamically add or remove the corresponding devices and structures. The implementation of this functionality is very similar in both subsystems. The main idea is the interception of uevents, which are messages sent by the kernel to the userspace with information about changes in the state of a device. The issue of uevents is managed by udev[9]. Therefore, the kernel must be compiled with the uevent option enabled in the configuration[10]. Otherwise, the hotplugging will not work.

For SCSI devices, the interception is done by means of the SCSI bus. Among its fields, the `struct bus_type` has a callback function whose declaration is shown in Figure 3.10. This function is called when a device is added, removed or a few other things that generate uevents[11]. The first parameter corresponds to the device whose state has changed and the second carries the uevent message.

```
int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
```

Figure 3.10.: Declaration of the `uevent` callback

For network devices, the work is done through the net class, that is internally represented by means of the `struct class`[12]. This structure has a uevent callback function as well as the SCSI bus that is called every time there is a change of state on a device that belongs to the class.

The way we access the SCSI bus is different to the way we access the net class. As we already explained, the SCSI bus is represented as an instance of the `struct bus_type` called `scsi_bus_type`[13]. The variable is declared global and it is an exported symbol, which means that we can just declare it as extern in our module and access the bus as we

[9] `https://wiki.debian.org/udev`
[10] DM_UEVENT `http://lxr.hpcs.cs.tsukuba.ac.jp/linux/drivers/md/Kconfig`
[11] `http://lxr.free-electrons.com/source/include/linux/device.h`
[12] `https://www.kernel.org/doc/htmldocs/device-drivers/API-struct-class.html`
[13] `http://lxr.free-electrons.com/source/drivers/scsi/scsi_sysfs.c`

would do with any other variable. The net class instead is not global and it is not an exported symbol, so we have to access it through the pointer to the net class that all network devices have. For this reason, it is a requirement that at least one network device is present in the system when Cryogenic is loaded in order to enable the network hotplugging. The loopback interface typically satisfies this requirement.

When Cryogenic is loaded, the addresses of the callbacks are changed so that they point to interceptor functions implemented in the module. The interceptors then find out which type of action has issued the uevent and call auxiliary functions that perform the device addition or removal.

When it is a "remove" action on a network device or a SCSI device that is not being used, the module sets the `unplugged` flag to 1, enacts the device removal as presented in Section 3.2.7 and marks the `pm_device` structure as free. If it is a SCSI device with open descriptors, the module sets the `unplugged` flag to 1 and wakes up waiting tasks, but keeps the rest of the fields of `pm_device` untouched. This way, we prevent other devices from using this slot and allow the system to use the same structure if the hardware device is reconnected later.

If it is an "add" action on a network device, Cryogenic looks for a free slot in `pm_devices`, sets the information of the new device and creates the character device. If there is no free position in the array, a message is written to the system log and the device is not added. If it is a SCSI device, the module first checks whether it has been reconnected and, if so, it sets the flag `unplugged` to 0 and resets other fields of `pm_device` that may change after the reconnection. Note that the open character device that prevented the device from being removed does not change, allowing the userspace application that was using it to continue working without the necessity of closing and reopening it.

Finally, the original uevent functions, whose addresses had been previously saved, are called in order not to compromise the system operation.

3.2.4. Task information

In order to support the work done by the system calls, a new structure that keeps track of tasks execution is provided by Cryogenic. Figure 3.11 illustrates its definition.

```
1 struct pm_private {
2     int first_poll;
3     int timer_added;
4     unsigned long min_delay;
5     struct timer_list timer;
6     struct pm_device *pm_dev;
7 };
```

Figure 3.11.: Definition of `struct pm_private`

Next we describe the structure's fields.

`int first_poll`
: This field is a flag that indicates whether it is the first time that poll has been called for

a specific task (`first_poll = 1`) or the task has returned to poll after its execution has been deferred (`first_poll = 0`).

`unsigned long min_delay`
: It is the minimum time the task must wait until its execution can be resumed. It is set in jiffies[14] and it is an absolute time.

`struct timer_list timer`
: This is the timer that controls the timeout. An excerpt of the definition of the `struct timer_list` is shown in Figure 3.12. The rest of fields have been omitted since they are not meant to be accessed from outside the timer code itself [1, Ch. 7]. The field `expires` is the timeout, that is to say, the maximum time that a task can wait until its execution can be resumed. As with the minimum delay, it is set in jiffies and it is an absolute time. The pointer `function` must be set to a function that will be called whenever the timer expires with `data` as an argument.

`int timer_added`
: This flag determines if the timer that controls the task timeout has been added or not. If the timer has been added it is set to 1, otherwise its value is 0.

`struct pm_device *pm_dev`
: Pointer to the `struct pm_device` that corresponds to the device the task wants to use.

```
1 struct timer_list {
2     /* ... */
3     unsigned long expires;
4     void (*function)(unsigned long);
5     unsigned long data;
6     /* ... */
7 };
```

Figure 3.12.: Definition of `struct timer_list`

3.2.5. System calls

As a device driver, Cryogenic must provide the necessary system calls to manage the character devices it creates. The set of system calls provided is illustrated in Figure 3.13, which shows the declaration of the `struct file_operations` of our module [1, Ch. 3].

[14] `http://www.makelinux.net/books/lkd2/ch10lev1sec3`

```
static struct file_operations pm_fops = {
    .owner = THIS_MODULE,
    .open = pm_open,
    .release = pm_release,
    .poll = pm_poll,
    .unlocked_ioctl = pm_ioctl
};
```

Figure 3.13.: Declaration of pm_fops

Next we present the work done in each of these system calls.

open

The declaration of the open system call is illustrated in Figure 3.14. This function creates an instance of the struct pm_private and assigns it to the private_data field of filp, which is meant to preserve state information across system calls [1, Ch. 3].

```
static int pm_open(struct inode *inode, struct file *filp)
```

Figure 3.14.: Declaration of pm_open

Then, some of the fields of the structure are set: pm_dev points to the corresponding device, timer_added is set to 0 and timer is initialised. Before the initialisation, the function and the data fields are set, respectively, to timeout_wake_up and to the created pm_private instance. Therefore, when the timer expires timeout_wake_up will be called with pm_private as a parameter. The function will be presented in following sections. Finally, a variable that keeps track of the references to the module is increased by 1. If the device is a SCSI device, the field scsi_cdev_open is increased by 1 as well.

release

The release system call first checks whether there is an scheduled timer for the task and, if so, it deletes it to prevent it from expiring. Then, the work done by pm_open must be undone. Thus, pm_release frees the memory that was allocated for the pm_private structure and decreases the value of the reference counter.

If it a SCSI device, its specific reference counter must also be decreased by 1. If the counter reaches 0 and unplugged is set, it means that the hardware device had been unplugged while the character device was still open and the device could not be removed. Therefore, the device is removed here and the slot is marked as free.

The declaration of this system call is illustrated in Figure 3.15.

```
1 static int pm_release(struct inode *inode, struct file *filp)
```

Figure 3.15.: Declaration of pm_release

ioctl

The ioctl [1, Ch. 6] system call is used to set the minimum delay and the timeout for a specific task. The declaration of this system call is illustrated in Figure 3.16.

```
1 static long pm_ioctl(struct file *filp, unsigned int cmd, unsigned long
     arg)
```

Figure 3.16.: Declaration of pm_ioctl

The parameter cmd is the ioctl command[15]. This parameter is used to identify the work that must be done in the system call. In our case, pm_ioctl is only used to set the delay and the timeout, and thus Cryogenic only defines one command, illustrated in Figure 3.17

```
1 #define PM_IOC_MAGIC 'k'
2 #define SET_DELAY_AND_TIMEOUT _IOW(PM_IOC_MAGIC, 1, struct pm_times)
```

Figure 3.17.: Definition of the ioctl command

Four different macros are available to perform the creation of ioctl commands. Since we are writing data on the module, we use the _IOW macro. The magic number is set to 'k' because it is one of the unused blocks defined in the ioctl documentation. The next parameter is used to identify the command among all the possible commands of a device driver. Since Cryogenic only defines one command, we assign it the number 1. Finally, the last parameter notifies the type of the data that is going into the kernel: struct pm_times. This structure is provided by Cryogenic and its definition is illustrated in Figure 3.18.

```
1 struct pm_times {
2     unsigned long delay_msecs;
3     unsigned long timeout_msecs;
4 };
```

Figure 3.18.: Definition of pm_times

[15] http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/Documentation/ioctl/ioctl-number.txt

The parameter `arg` is the address of an instance of this structure that is created in userspace. Thus, `ioctl` just needs to cast it to `struct pm_times *` in order to access its fields.

When `pm_ioctl` is called, we get the `pm_private` structure of the task through the parameter `filp`. Before setting the delay and the timeout, we need to check if a timer had been previously added. In that case, we delete it and set the corresponding flag to 0. This is done in order to prevent Cryogenic from adding more than one timer for the same task at the same time.

Then the delay and timeout passed as parameters are set on the corresponding fields of `pm_private`. Both values are relative times passed in milliseconds, but must be saved as absolute times in jiffies, so the appropriate conversions are done before setting the values [1, Ch. 7].

It is important to notice that `pm_ioctl` does not add any timer, it only sets the field `expires` of the timer that later, in `pm_poll`, will be added. For this reason, the `first_poll` flag must be set here to 1, since the call to `pm_poll` will be issued right after the call to `pm_ioctl`.

poll

The `poll` [1, Ch. 6] system call determines whether a task is allowed to perform an I/O operation or otherwise it must wait some event to happen. The declaration of `pm_poll` is illustrated in Figure 3.19.

```
1 static unsigned int pm_poll(struct file *filp, struct poll_table_struct
      *table)
```

Figure 3.19.: Declaration of `pm_poll`

Similarly to `pm_ioctl`, the first we need to do here is getting the `pm_private` structure by means of the parameter `filp`.

Firstly, Cryogenic checks if the `unplugged` flag is set. If it is, we must return a value indicating that the device is ready to perform an I/O operation. Otherwise, Cryogenic checks if the current time is greater than the task's delay. If it does not, the task is not allowed to proceed. If it does, we must check if the execution comes from a timeout that has expired or from an I/O operation performed by some other application. In both cases, `first_poll` will be set to 0 and then, `pm_poll` must allow the task to proceed and perform the I/O operation by returning the appropriate value. If `first_poll` is set to 1 the task is not allowed to proceed, since it means that this is the first time that `pm_poll` has been called and the task must wait for its timer to expire or for other tasks to perform an I/O operation on the device.

Finally, when a task is not allowed to proceed, `poll_wait` [1, Ch. 6] is called in order to queue the task on the `event_queue` of the corresponding device and a timer is added to control the timeout. The corresponding flag is set to 1 to indicate that it has been added. The flag `first_poll` is set to 0 to indicate that the next time `pm_poll` is called for this task

it will not be a direct call from the userspace, but a resumption of activity after waiting for an event.

3.2.6. Triggering events

Two possible events are meant to resume the activity of waiting tasks: an I/O operation on a SCSI or network device performed by other tasks or the expiration of their timer. When these events occur, an auxiliary function that calls the `wake_up` [1, Ch. 6] method over the corresponding `event_queue` is executed. In this section we present the different functions that perform this work.

I/O operation on SCSI devices

The I/O operations on block devices are managed through the `struct request_queue`. When a process needs to perform a read or a write operation, a request is created and queued on the `request_queue` associated to the device. When it is time to process requests, the `request_fn` method associated to the queue is called [1, Ch. 16]. This method is a callback whose declaration is illustrated in Figure 3.20.

```
1 typedef void (request_fn_proc) (struct request_queue *q);
2
3 struct request_queue {
4     /* ... */
5     request_fn_proc          *request_fn;
6     /* ... */
7 };
```

Figure 3.20.: Declaration of `request_fn`

In order to resume the activity of tasks that are waiting for a specific SCSI device, Cryogenic intercepts the calls to the `request_fn` method of this device. The interception is done through the substitution of the callback's addresses by the address of the interceptor function defined in our module. The field `scsi_request_fn_address` is used to save the original address.

The interceptor determines which SCSI device is being requested to perform the I/O operation and then wakes up all tasks present on its `event_queue`. Eventually, the original `request_fn` method is called to continue with the regular system execution.

If the requested device is not found because it has been unplugged, a message is written to the system log. Since the `request_fn` function is a void method, there is no need to return an error code.

I/O operation on network devices

The `struct net_device` has a field called `netdev_ops` that points to an instance of `struct net_device_ops`[16]. This structure defines several callbacks to manage the oper-

[16] `http://lxr.free-electrons.com/source/include/linux/netdevice.h`

ation of the network device. To enact the resumption of tasks that perform network I/O, Cryogenic intercepts the calls to the function `ndo_start_xmit`, declared in Figure 3.21.

```
1 struct net_device_ops {
2     /* ... */
3     netdev_tx_t             (*ndo_start_xmit) (struct sk_buff *skb,
          struct net_device *dev);
4     /* ... */
5 };
```

Figure 3.21.: Declaration of `ndo_start_xmit`

As its name indicates, this function initiates the transmission of a packet. Its interception requires more work than the SCSI `request_fn` method, since the pointer `netdev_ops` is declared constant and thus, the address of the callback cannot be directly changed by the interceptor address. Instead, we need to create and identical copy of the structure pointed by `netdev_ops` that is stored on the field `my_ops` of `pm_device`. Then, the `ndo_start_xmit` callback of this new instance is modified to point to our interceptor and `netdev_ops` is made to point to the new instance. The address of the original instance is saved on the `old_ops` field.

From this point on, the work done is almost identical to the SCSI interceptor: the function determines which device is being requested to send a packet, wakes up the tasks queued on its `event_queue` and calls the original `ndo_start_xmit` function.

The `ndo_start_xmit` function returns a value according to the result of the transmission. Thus, if the device is not found, the interceptor returns `NETDEV_TX_BUSY`[17] to indicate that the driver could not properly take care of the packet. A notification message is written to the system log as well.

In contrast to the packet transmission, the packet reception is an event that comes from outside the system. This makes the system unable to decide or know when a packet must be received and therefore, the packet reception has not been considered as a triggering event in this thesis.

Timer expiration

As we presented in Section 3.2.4, the `struct timer_list` has a field called `function`. This pointer must point to a function defined in our module that will be called whenever the timer expires.

The name of this function is `timeout_wake_up` and its definition is illustrated in Figure 3.22.

```
1 static void timeout_wake_up(unsigned long private_data)
```

Figure 3.22.: Definition of `timeout_wake_up`

[17] `http://lxr.gwbnsh.net.cn/linux/Documentation/networking/netdevices.txt`

Through the pointer passed as a parameter, the function is able to get the `private_data` structure that contains the information of the task whose timer has expired. The expected behaviour here would be to wake up only this task. However, waking up this task would likely issue an I/O request on the network device that would eventually wake up all the tasks in the `event_queue`. Thus, we can just directly wake up all tasks in the queue.

3.2.7. Device removal

Either when a hardware device is unplugged or when Cryogenic is unloaded, one or more devices may be removed from the system and some actions must be enacted in order to keep the system's consistency and avoid future errors.

For each target device that is actually being removed, the first job consists of restoring the original address of the `request_fn` method or the `ndo_start_xmit` function. This is important since all hardware devices are still attached to the system after unloading Cryogenic and the structures illustrated in Figure 3.4 remain in the kernel and they must keep working properly. Then, all tasks that are waiting in the `event_queue` are woken up in order to delegate the error handling as we already explained. Lastly, the associated character device created by Cryogenic and its corresponding device node is destroyed, and the `pm_device` instance is set as free. Note that Cryogenic cannot be unloaded if any of its character devices are open.

In Section 3.2.3 we explained that a SCSI device that is unplugged while its character device is open is not removed from the system, but its tasks are woken up. As a result, when they return to `pm_poll`, the system call will return a value indicating that they are ready to proceed. The userspace application will then try to perform an I/O operation on a device that is not present on the system anymore. Thus, the error handling is delegated to the calls that actually manage the operation. Besides, if the hardware device is plugged again before closing the descriptor, the userspace application is able to continue its execution using the same descriptor.

The operation of network devices is subject to factors that are no guaranteed to remain unchanged after a possible reconnection, for example the type of IP address used (IPv4/IPv6) or the network they belong to. Thus, the case of a device being reconnected has not been considered for network devices and they are removed from Cryogenic after they are unplugged. If an associated character device was open, the userspace application must close it an open a new one to continue working.

3.3. Developer perspective

In this section we present the way developers should use Cryogenic's API in order to benefit from its capabilities. We illustrate with examples the code that must be added to modify the behaviour of any application and give a clearer idea of the execution flow.

3.3.1. UDP client

To start with, consider the C code in Program 3.1 that implements a simple UDP client.

Program 3.1: client.c

```c
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define BUFLEN 512
#define PORT 666
#define SRV_IP "131.159.74.67"

int main(int argc, char *argv[])
{
    struct sockaddr_in sock;
    char buf[BUFLEN];
    int sock_fd;
    int i;

    sock_fd = socket(PF_INET, SOCK_DGRAM, 0);
    if (sock_fd < 0) {
        perror("socket() failed\n");
        exit(1);
    }

    memset((char *) &sock, 0, sizeof(sock));
    sock.sin_family = AF_INET;
    sock.sin_port = htons(PORT);
    if (inet_aton(SRV_IP, &sock.sin_addr) == 0) {
        fprintf(stderr, "inet_aton() failed\n");
        exit(1);
    }

    struct timeval exec_t;
    struct tm *t;

    i = 1;
    while(1) {

        sprintf(buf, "%02d\0\n", i);

        struct sockaddr *saddr = (struct sockaddr *) &sock;
        if (sendto(sock_fd, buf, BUFLEN, 0, saddr, sizeof(sock)) < 0) {
            perror("sendto() failed\n");
            exit(1);
        }
        gettimeofday(&exec_t, NULL);
        t = localtime(&exec_t.tv_sec);
```

```
51          printf("Sent %02d [%02d:%02d:%02d.%03d]\n", i, t->tm_hour, t->
                tm_min, t->tm_sec, (int) exec_t.tv_usec/1000);
52          ++i;
53
54          sleep(5);
55      }
56
57      close(sock_fd);
58
59      return 0;
60  }
```

This program creates a UDP socket and sends a packet to a server every 5 seconds. Immediately after sending every packet, a message with the transmission number and a timestamp is printed to the standard output as we can see in Figure 3.23. It is easy to notice the period of 5 seconds between transmission looking at the timestamps.

```
1  # ./client
2  Sent 01 [16:17:56.322]
3  Sent 02 [16:18:01.323]
4  Sent 03 [16:18:06.323]
5  Sent 04 [16:18:11.323]
6  Sent 05 [16:18:16.324]
```

Figure 3.23.: Sample output of Program 3.1

Imagine now that we want to apply Cryogenic to give more flexibility to the transmission time. Figure 3.24 illustrates the behaviour of the UDP client before and after applying the modifications.

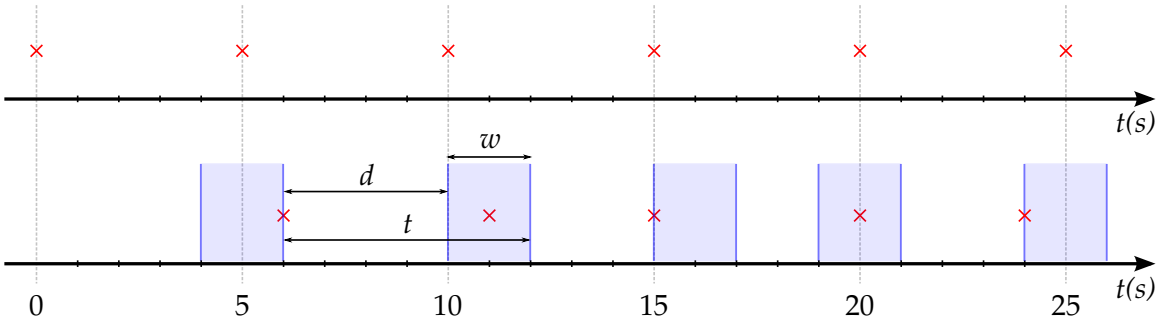


Figure 3.24.: Behaviour of Program 3.1 before and after Cryogenic

The first timeline presents the normal operation: a transmission, represented by a red cross, is issued every 5 seconds. The second timeline presents the operation under Cryogenic's management. After every transmission, the program waits 4 seconds and then, a window w of 2 seconds starts. This window is the margin we want to let the program to send the next packet and Cryogenic will determine the time inside the window when the packet will be actually transmitted. As a result, the transmissions will be issued between 4

and 6 seconds after the previous one. These are, respectively, the delay d and the timeout t that we will set later to the task.

In order to apply Cryogenic, the first thing we must do is adding the code that opens the character device associated to the interface that will be used to send the packets, which we must know beforehand. If it was, for instance, `eth0`, we would add the code illustrated in Figure 3.25 before the main loop. The resulting file descriptor will be used later to call `ioctl` and `select`.

```
int fd = open("/dev/cryogenic/eth0", O_RDWR);
if (fd < 0) {
    perror("open() failed");
    exit(1);
}
```

Figure 3.25.: Opening the character device

Next, we have to pass the delay and the timeout to the module. For that purpose, we must define and use the `struct pm_times`, presented in Section 3.2.5. Remember that these times must be passed in milliseconds and they are relative times. The code added is illustrated in Figure 3.26.

```
struct pm_times times;
times.delay_msecs = 4000;
times.timeout_msecs = 6000;
```

Figure 3.26.: Setting the delay and the timeout

Now it is time to call `ioctl` in order to actually pass the values to the module. The code illustrated in Figure 3.27 is added at the beginning of the main loop. Note that the `ioctl` command used is the one presented in Section 3.2.5 and thus we must define it here as well.

```
int r;
r = ioctl(fd, PM_SET_DELAY_AND_TIMEOUT, &times);
if (r < 0) {
    perror("ioctl() failed");
    exit(1);
}
```

Figure 3.27.: Calling `ioctl`

The following step is calling `select`. This system call receives one or more file descriptors and determines whether I/O becomes possible for any of them. The file descriptors are passed as parameters by means of three sets: a set to check the availability of data for

reading, a set to check the availability of space to write and a set for exceptional conditions that usually involve TCP sockets. After calling `select`, all file descriptors will be removed from the sets except those that are immediately ready to perform the corresponding I/O operation and a value that describes the operations that could be performed is returned.

The `select` system call will internally execute `pm_poll`, presented in Section 3.2.5. Therefore, whenever `select` determines that a file descriptor is ready for reading or writing, it will be actually determining that the hardware device associated to that character device is already active and Cryogenic allows the task to proceed its execution.

Thus, we need to create a set and add the file descriptor of the open character device. We could use the read or the write set but not the set for exceptional conditions since the implementation of `pm_poll` does not support it. Figure 3.28 illustrates how to do this task.

```
1 fd_set wr;
2 FD_ZERO(&wr);
3 FD_SET(fd, &wr);
```

Figure 3.28.: Declaring and initialising the sets of file descriptors

After the declaration, the macro `FD_ZERO` clears the sets and `FD_SET` adds `fd` to the set that monitors the availability of write operations. Now the call to `select` can eventually be performed right before sending the packet. The code is shown in Figure 3.29.

```
1 r = select(fd+1, NULL, &wr, NULL, NULL);
2 if (r < 0) {
3     perror("select() failed");
4     exit(1);
5 }
```

Figure 3.29.: Calling `select`

The first parameter must be an integer one more than the maximum file descriptor in any of the sets. Since we only have one descriptor, we just need to increase the descriptor of the open device by one. The last parameter is a timeout that determines the longest time that `select` may wait before returning. Since it is set to NULL, `select` will block indefinitely until the descriptor becomes ready.

Eventually, it has to be checked with the `FD_ISSET` macro that the file descriptor is still in the set in order to effectively send the packet. The code that sends the packet should be wrapped by and `if` statement, as Figure 3.30 shows.

```
if (FD_ISSET(fd, &wr)) {

    /* ... */

}
```

Figure 3.30.: Checking the presence of the file descriptor in the set

Lastly, the file descriptor must be closed before the end of the main function. Figure 3.31 illustrates the code that performs this operation.

```
close(fd);
```

Figure 3.31.: Closing the character device

After adding these excerpts of code and deleting the call to `sleep`, the program is ready to run under the management of Cryogenic. The entire code with all the necessary includes, defines and declarations can be consulted in Section B.1.2.

To summarize, in every iteration of the loop, the call to `ioctl` sets the delay and the timeout and `select` is called with the descriptor of the character device passed as a parameter. The execution will block there until the descriptor becomes available for writing, that will happen between 4 and 6 seconds after the call and actually means that the network device is active and sending the packet will not be expensive in terms of energy consumption or, otherwise, the device has not been activated by others and the timeout has expired forcing the packet to be sent anyway. Figure 3.32 illustrates the output of the program.

```
# ./client-cryo
Sent 01 [16:24:01.852]
Sent 02 [16:24:06.860]
Sent 03 [16:24:11.566]
Sent 04 [16:24:17.580]
Sent 05 [16:24:23.596]
Sent 06 [16:24:29.612]
Sent 07 [16:24:33.908]
Sent 08 [16:24:39.916]
Sent 09 [16:24:45.932]
Sent 10 [16:24:50.940]
```

Figure 3.32.: Sample output of the program in Section B.1.2

This section is just an example to introduce the usage of the API. Developers may want their tasks to have a different behaviour that fits better to the requirements and the operation of their applications. Therefore, they need to properly calculate the delay and the timeout for each transmission.

3.3.2. Filesystem synchronization

In this section we illustrate the usage of Cryogenic with SCSI devices and give a simple example where the behaviour we want to achieve with Cryogenic requires the calculation of a different delay and timeout for each task.

Consider the code in Program 3.2 that has a similar behaviour to the one in the previous section.

Program 3.2: sync.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include <time.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <sys/ioctl.h>
#include <sys/select.h>
#include <sys/time.h>

#define BUFLEN 512

int main(int argc, char *argv[])
{
    char buf[BUFLEN];
    int i;

    int fd_file = open(argv[1], O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if (fd_file < 0) {
        perror("open() failed");
        exit(1);
    }

    struct timeval exec_t;
    struct tm *t;

    i = 1;
    while(1) {

        int b = sprintf(buf, "%d\n", i);

        if (write(fd_file, buf, b) < 0) {
            perror("write() failed\n");
            exit(1);
        }
        sync();
        gettimeofday(&exec_t, NULL);
        t = localtime(&exec_t.tv_sec);
        printf("Written %02d [%02d:%02d:%02d.%03d]\n", i, t->tm_hour,
```

```
43                  t->tm_min, t->tm_sec, (int) exec_t.tv_usec/1000);
44          ++i;
45
46          sleep(5);
47      }
48
49      close(fd_file);
50
51      return 0;
52  }
```

This program opens a file passed as a parameter and, every 5 seconds, it writes to the file and then synchronizes the filesytem to actually write the changes to the hard drive. After every synchronization a message with the writing number and a timestamp is printed to the standard output.

Now we want to apply Cryogenic so that we define a tolerance window to send the packet that will not be placed depending on the previous synchronization time, but on the period. Figure 3.33 illustrates this behaviour.

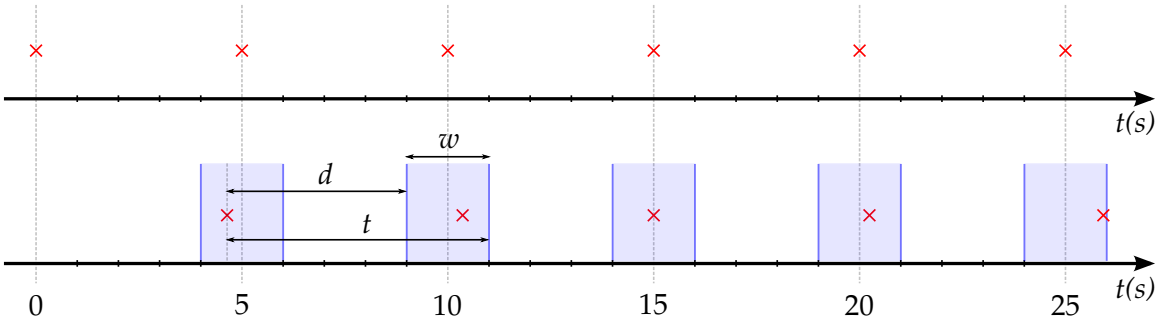


Figure 3.33.: Behaviour of Program 3.2 before and after Cryogenic

The delay and the timeout for every synchronization are calculated in function of the time of the previous one in order to make the middle of the window coincide with every period of 5 seconds. This way, given a situation where the timeout of the task always expires because no other tasks are using the device, the position of the windows will not be affected in contrast to the program in the previous section, where the windows would be increasingly displaced.

Add the code that opens the character device as we presented in the previous section. Remember that now the file we must use is named after the serial number of the hard drive where the file that is modified is stored.

Next we define a new variable for the tolerance and set it to 2 seconds. Then, we can declare the variable of type `pm_times` and set the delay and the timeout, respectively, to 5 seconds minus half the tolerance and 5 seconds plus half the tolerance, as it is illustrated in Figure 3.34.

```
unsigned long tolerance = 2000;
struct pm_times times;
times.delay_msecs = 5000 - tolerance/2.0;
times.timeout_msecs = 5000 + tolerance/2.0;
```

Figure 3.34.: Setting the delay and the timeout

These values will be only valid for the first synchronization. From now on we need to calculate them in every iteration, so we need a variable that stores the time when the synchronizations start. Add the code in Figure 3.35 right before the loop starts.

```
struct timeval start_t;
gettimeofday(&start_t, NULL);
```

Figure 3.35.: Getting the start time

The next steps are identical to the UDP client example: call `ioctl`, declare a set of descriptors, add the descriptor of the character device, call `select` and wrap the code that writes to the file and calls `sync` with the `if` statement that checks the presence of the descriptor in the set.

Before the loop ends, we need to set the delay and the timeout for the next synchronization. The code that performs this task is illustrated in Figure 3.36, that also includes the code that prints the new calculated values to the standard output.

```
unsigned long start_t_msecs = (start_t.tv_sec*1000) + (start_t.tv_usec
    /1000);
unsigned long exec_t_msecs = (exec_t.tv_sec*1000) + (exec_t.tv_usec
    /1000);
times.delay_msecs = start_t_msecs + 5000*i - exec_t_msecs - tolerance
    /2.0;
times.timeout_msecs = times.delay_msecs + tolerance;
printf(" - New delay: %lu\n", times.delay_msecs);
printf(" - New timeout: %lu\n", times.timeout_msecs);
```

Figure 3.36.: Calculating the new delay and timeout

The values are calculated the following way:

$$t_{delay_i} = t_{start} + 5000 \times i - t_{exec_{i-1}} - \frac{w}{2} \tag{3.1}$$

$$t_{timeout_i} = t_{delay_i} + w \tag{3.2}$$

Figure 3.37 clarifies this equations. The addition of t_{start}, which is the time when the main loop starts, and $5000 \times i$ corresponds to the time of period i. Since the delay and

timeout are passed as relative times, we need to subtract the time when the $(i-1)$-th synchronization is issued from the previous addition, obtaining as a result the time area between $t_{exec_{i-1}}$ and i. Finally, to obtain the delay for the current synchronization we just need to subtract from this area half the duration of the window, and the timeout is the result of adding half the window to this area or, easier, adding the window to t_{delay_i}.

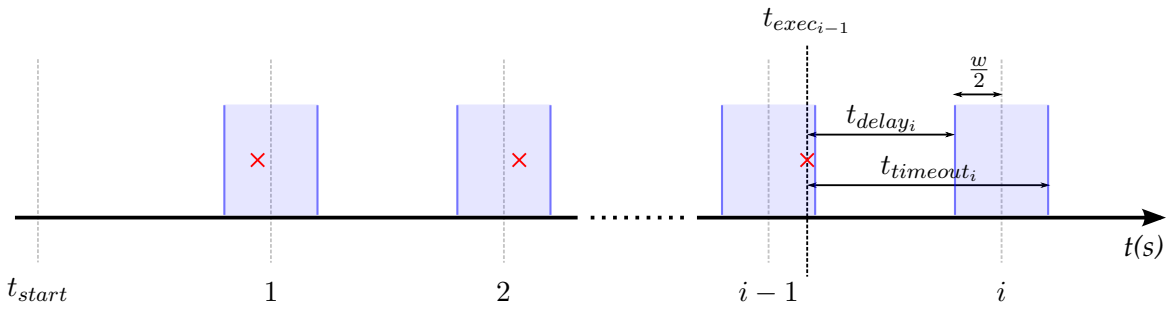


Figure 3.37.: Times involved in the calculation of the delay and the timeout

After deleting the call to `sleep` the code is ready to be executed. The entire program with all the modifications can be found in Section B.2.1.

Figure 3.38 illustrates the output of the program. We can observe the delay and the timeout for the next synchronization after every writing and how it is respected: the synchronizations are never issued before the delay or after the timeout. It is also noticeable that great values come always after an early synchronization, and small values come after late synchronizations. This allows to make the window coincide with the initial period of 5 seconds, as we explained before.

```
# ./sync-cryo file.txt
Written 01 [18:24:39.397]
 - New delay: 3241
 - New timeout: 5241
Written 02 [18:24:43.491]
 - New delay: 4147
 - New timeout: 6147
Written 03 [18:24:49.600]
 - New delay: 3038
 - New timeout: 5038
Written 04 [18:24:53.823]
 - New delay: 3815
 - New timeout: 5815
Written 05 [18:24:58.076]
 - New delay: 4562
 - New timeout: 6562
Written 06 [18:25:04.244]
 - New delay: 3394
 - New timeout: 5394
```

Figure 3.38.: Sample output of the program in Section B.2.1

3.3.3. The GNUnet neighbour discovery

The examples we have presented so far were specially created in order to illustrate sample scenarios where Cryogenic could be beneficial and introduce the usage of its API. In this section we want to test the ease of migration of an existing application that was not originally thought to use Cryogenic. For this purpose, we provide and example where it is applied on a real software, and evaluate then the effort needed to achieve the integration of Cryogenic into the existing code.

The aforementioned software is GNUnet[18], an official GNU package that provides a framework for secure, decentralized peer-to-peer networking. GNUnet allows anonymous censorship-resistant file-sharing, and provides link encryption, peer discovery, resource allocation and communication over several transport protocols.

Ir order to connect with the others, peers in the GNUnet overlay network need to get address information. Among other methods, GNUnet provides UDP neighbour discovery in LAN to obtain this information. Every 5 minutes, the broadcast addresses of each IPv4 interface found are gathered and a HELLO message is sent to these addresses. Similarly, a multicast request is created for each IPv6 interface and the HELLO message is sent to the multicast groups.

The behaviour of this functionality gives the opportunity to apply Cryogenic in order to reduce the power consumed by each network device. In order to allow the addition and the correct operation of Cryogenic's API, a previous modification is enacted: each broadcast or multicast message is sent from a single task. This is a requirement, since we need to open the corresponding file in `/dev/cryogenic/` for each interface, and set the delay and timeout and call `select` for each transmission.

Upon this, we can add to the file `plugin_transport_udp_broadcasting.c`, located under the `transport/` directory in GNUnet's source tree[19], the code that belongs to Cryogenic. This file contains all the necessary structures and functions that perform the neighbour discovery. We pay special attention to the following ones that are being modified:

`struct BroadcastAddress`

> This structure contains a pointer to an address and a pointer to an instance of `Plugin`, a structure that encapsulates information about the UDP transport protocol state. An instance of this structure is created for each interface found.

`static int iface_proc()`

> This function is called for every interface found. If it is an IPv4 interface, it gets its broadcast address and calls the function that issues the transmission. If it is an IPv6 interface, it creates the multicast request and calls the function that issues the transmission as well.

`static void udp_ipv4_broadcast_send()`

> This function issues the transmission of a broadcast message through an IPv4 interface and queues the following transmission.

[18] `https://gnunet.org/`
[19] `https://gnunet.org/svn/gnunet/src/`

`static int udp_ipv6_broadcast_send()`

This function issues the transmission of a multicast message through an IPv6 interface and queues the following transmission.

`void stop_broadcast()`

This function disables the broadcasting functionality.

The code that is being added follows the same structure as the UDP client and the filesystem synchronization examples, but looks slightly different, since we use the GNUnet's API to call some system calls, as well as to handle some errors.

The first step is adding to the `BroadcastAddress` structure two fields that save, respectively, the file descriptor of the corresponding character device and the delay and timeout for the transmissions. The new fields are illustrated in Figure 3.39. The structure `GNUNET_DISK_FileHandle` is provided by the GNUnet's API in order to handle open files on different operating systems. The second new field is an instance of the structure supplied by Cryogenic to handle the times, which we presented in Section 3.2.5.

```
#if LINUX
    /**
     * Cryogenic handle.
     */
    struct GNUNET_DISK_FileHandle *cryogenic_fd;

    /**
     * Time out for cryogenic.
     */
    struct pm_times cryogenic_times;
#endif
```

Figure 3.39.: New fields in `struct BroadcastAddress`

Next, the device node corresponding to each interface found should be opened. This is done in the `iface_proc` function, as Figure 3.40 illustrates. This code is added right before calling the functions `udp_ipv4_broadcast_send` and `udp_ipv6_broadcast_send`.

The `GNUNET_DISK_file_open` function is a GNUnet's API function that opens a file. It receives as parameters the name of the file to open and two flags that determine the opening mode and the permissions, and returns an instance of the `GNUNET_DISK_FileHandle` structure.

The interface name whose character device has to be opened is one of the parameters that `iface_proc` receives, and it is handled by means of other GNUnet's functions in order to build the entire path to the corresponding file.

```
#if LINUX
    /*
     * setup Cryogenic FD for ipv4 broadcasting
     */
    char *filename;

    GNUNET_asprintf (&filename,
                     "/dev/cryogenic/%s",
                     name);
    if (0 == ACCESS (name, R_OK))
    {
        ba->cryogenic_fd =
        GNUNET_DISK_file_open (filename,
                               GNUNET_DISK_OPEN_WRITE,
                               GNUNET_DISK_PERM_NONE);
    }
    GNUNET_free (filename);
#endif
```

Figure 3.40.: Opening the character device for an IPv4 interface

The following steps are calculating and setting the delay and the timeout for the transmission and calling `select`. The code illustated in Figure 3.41 is added to the functions that issue the transmission of a message.

In order to calculate the delay and the timeout we make use of the broadcast interval, which is already stored in the `Plugin` structure that is passed to the functions as a parameter. As we want to place the new tolerance window in a symmetrical position with respect to the current transmission period, the delay is set to the broadcast interval minus the 50%, and the timeout is set to the broadcast interval plus the 50%.

GNUnet does not define any function in its API to call `ioctl`, thus the call is performed using the usual API. If `ioctl` fails, the necessary actions to continue with the regular system operation are enacted.

In contrast to `ioctl`, the call to `select` is performed through a GNUnet's API function: `GNUNET_SCHEDULER_add_write_file`. The call to this function blocks until the file descriptor passed as the second parameter becomes ready for writing. When this happens, the function passed as the third parameter is called.

As we can see, we are calling the same function again, which will issue the transmission in the first place and then, set the times and call `select` for the next one.

The code in Figure 3.41 corresponds to an IPv4 interface. For an IPv6 interface the code would be identical but replacing all the appearances of `udp_ipv4_broadcast_send` with the corresponding one for IPv6. In both cases, the code between lines 1 and 26 is added before the call to `GNUNET_SCHEDULER_add_delayed`, which appears in line 27. This function calls the function passed in the second parameter after the interval of time passed in the first parameter. This is the current behaviour of the broadcasting and this line will be executed only in case of an error.

```
#if LINUX
    /*
     * Cryogenic
     */
    if (NULL != baddr->cryogenic_fd)
    {
        baddr->cryogenic_times.delay_msecs =
            (plugin->broadcast_interval.rel_value_us/1000.0)*0.5;
        baddr->cryogenic_times.timeout_msecs =
            (plugin->broadcast_interval.rel_value_us/1000.0)*1.5;

        if (ioctl(baddr->cryogenic_fd->fd,
                  PM_SET_DELAY_AND_TIMEOUT,
                  &baddr->cryogenic_times) < 0)
        {
            GNUNET_log_strerror (GNUNET_ERROR_TYPE_WARNING, "ioctl");
            baddr->broadcast_task =
               GNUNET_SCHEDULER_add_delayed (plugin->broadcast_interval,
                    &udp_ipv4_broadcast_send, baddr);
        }
        else
            GNUNET_SCHEDULER_add_write_file (
               GNUNET_TIME_UNIT_FOREVER_REL, baddr->cryogenic_fd,
                                             &udp_ipv4_broadcast_send,
                                             baddr);
    }
    else
#endif
    baddr->broadcast_task =
        GNUNET_SCHEDULER_add_delayed (plugin->broadcast_interval,
                                      &udp_ipv4_broadcast_send, baddr);
```

Figure 3.41.: Setting the delay and the timeout and calling `select` for and IPv4 interface

Finally, all the open file descriptors should be closed before the neighbour discovery ends. Figure 3.42 illustrates the codes that performs this task, added after the main `while` loop in function `stop_broadcast`. The pointer `p` points to a `BroadcastAddress` and it is got previously in the function through the `Plugin` passed as a parameter.

```
#if LINUX
    GNUNET_DISK_file_close(p->cryogenic_fd);
#endif
```

Figure 3.42.: Closing the character device descriptor

As we already mentioned, the code added to GNUnet follows the same structure and the same flow of events as the previous examples. This is because the key of the behaviour of

any application running under Cryogenic's management is the delay and the timeout, and their calculation is the only part of the code that completely depends on the developer. This makes it possible to apply Cryogenic to any real software by using localized modifications in all cases, regardless of the overall system complexity.

Besides, the implementation of Cryogenic by means of a module and its redefinition of the POSIX system calls allows the developer to use existing APIs in most cases. This is an advantage, since the developer does not need to learn a new API, but has to learn how to make Cryogenic work properly.

4. Experimentation

In this chapter we present the experiments performed in order to evaluate Cryogenic's operation. We present first the methodology used and provide guidelines to do the correct setup. Upon this, we present the test programs and illustrate the results obtained.

4.1. Methodology

The first decision concerning the methodology followed to carry out the energy measurements was the usage of a Raspberry Pi[1] as the base system. The Raspberry Pi is a single-board computer composed of a Broadcom system on a chip (SoC), a SD Card used for booting and persistent storage, two USB and one ethernet ports, video and audio outputs and other low-level peripherals. The Broadcom SoC is composed of an 700MHz ARM processor and 512MB of RAM. The power is usually supplied through a MicroUSB cable. Figure 4.1 shows a scheme of the Raspberry Pi.

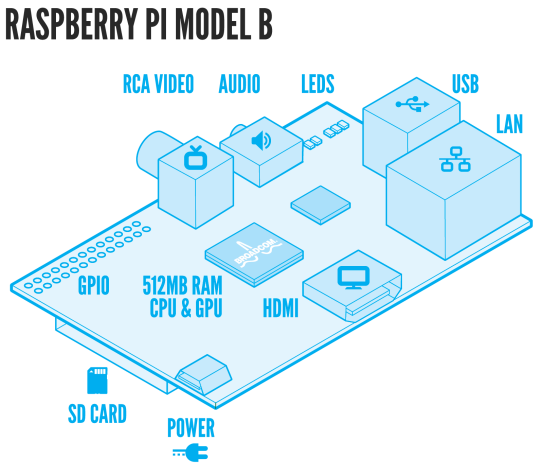


Figure 4.1.: Raspberry Pi scheme

The main reason that led to this decision was the low baseline power consumption of a Raspberry Pi, which is around 3.5W and tiny compared to a desktop or a laptop. This is important, since this way we have a chance of observing the reduction in power consumption. Other advantages are the possibility of supplying power to the Pi through the pin header, that allows us to control the input voltage or current supplied, and obviously the possibility of using a GNU/Linux distribution as the main operating system.

[1] http://www.raspberrypi.org/

In order to measure the energy consumed by the Raspberry Pi, we supply a constant input voltage V_i of 5V to the Pi and use an oscilloscope to measure the current draw $I(t)$ over the platform while the tests programs are running. Then, the power is calculated applying the following formula:

$$P(t) = V_i \times I(t) \tag{4.1}$$

Finally, we integrate the power with respect to time to obtain the total energy consumption:

$$E = \int_a^b P(t)\, dt \tag{4.2}$$

Figure 4.2 illustrates the circuit used to perform the measurements. Power is supplied using a DC power supply, and two probes are required: a current probe, represented by a dotted line, and a voltage probe, represented by a dashed line. In particular, for our experimentation we used a *TTi CPX400D Dual 420 watt* power supply, a *Tektronix MSO 2024* mixed signal oscilloscope, a *Tektronix A622 AC/DC* current probe and the Raspberry Pi was model B.

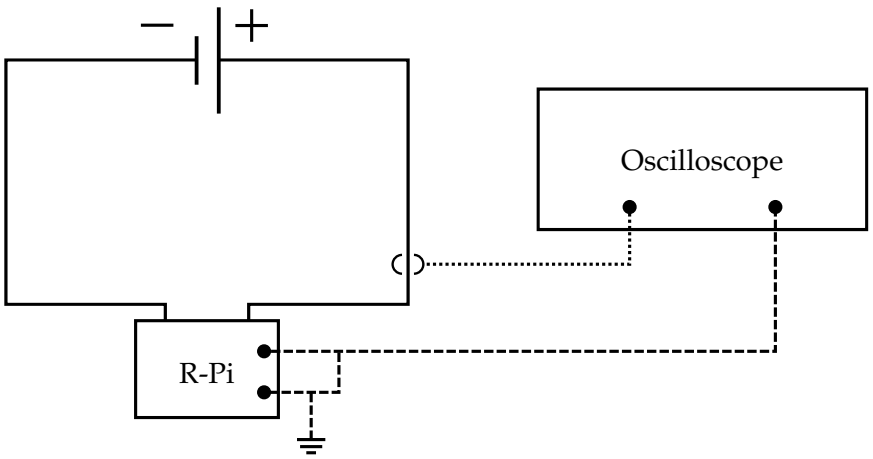


Figure 4.2.: Circuit diagram for measurements

The voltage probe measures the voltage on a specific General Purpose Input/Output (GPIO) pin, which is previously configured as an output pin. When the execution of the test programs start, the pin is set to its high value and provides a voltage of 3.3V. When the execution finishes, it is set to its low value again and the output voltage drops to 0V. This way we determine when the test programs start and end their execution.

4.2. Set Up

In this section we present how to set up the Raspberry Pi in order to successfully load Cryogenic and run the test programs. We also illustrate how to assemble the different components of the circuit in Figure 4.2.

4.2.1. Raspberry Pi

For our experimentation on the Raspberry Pi, we used the last version of the Debian-based distribution Raspbian, which at the time of writing is *wheezy* and can be downloaded from `www.raspberrypi.org/Downloads`. As usual, the image must be written to a 4GB or larger SD card using the UNIX tool `dd`.

Module Compilation

Cryogenic must be compiled before loading it on the Raspberry Pi. There are to ways of doing this: compiling directly on the Pi or cross compiling on another GNU/Linux system. Compiling on the Pi requires the installation of the corresponding Linux headers. Although we managed to compile Cryogenic, further errors appeared at loading time and we never succeeded to run Cryogenic after compiling it on the Pi. Thus, in this thesis we document how to cross compile Cryogenic and we would strongly recommend to do it this way.

First, download the latest Raspberry Pi compiler and set an environment variable that points to its location:

```
# git clone https://github.com/raspberrypi/tools
# export CCPREFIX=/home/me/tools/arm-bcm2708/arm-bcm2708-linux-gnueabi/bin/arm-bcm2708-
    linux-gnueabi-
```

Now, download the Raspberry Pi kernel source. Since *wheezy* uses the kernel version 3.6.11+, we must download it from the 3.6 stable code branch:

```
# mkdir raspbian
# cd raspbian
# git init
# git fetch git://github.com/raspberrypi/linux.git rpi-3.6.y:refs/remotes/origin/rpi-3.6.y
# git checkout rpi-3.6.y
```

Then set the following environment variable that points to the location of the source:

```
# export KERNEL_SRC=/home/me/raspbian
```

The next step is to compile the entire kernel. Although we will not use the resulting image, we need to build `Module.symvers` and other intermediaries that will allow us to compile the module.

Run the following command to ensure that we have a clean directory tree:

```
# make mrproper
```

Then copy the current configuration file of the Pi to the source location:

```
# scp pi@rpiaddress:/proc/config.gz .
# zcat config.gz > .config
```

Finally, write the configuration and compile the kernel:

```
# make ARCH=arm CROSS_COMPILE=${CCPREFIX} oldconfig
# make ARCH=arm CROSS_COMPILE=${CCPREFIX}
```

At this point, go to the directory where Cryogenic source file is located and create the `Makefile` to compile it:

```
obj-m += pm-rpi.o

all:
    make ARCH=arm CROSS_COMPILE=${CCPREFIX} -C $(KERNEL_SRC) M=$(PWD)
       modules

clean:
    make -C $(KERNEL_SRC) M=$(PWD) clean
```

Run `make` and the object file `pm-rpi.ko` will be created. This file must be copied to the Raspberry Pi.

Module Loading

Once the `pm-rpi.ko` file is present on the Pi, we can dynamically load or unload it running the following commands:

```
# insmod pm-rpi.ko
# rmmod pm-rpi.ko
```

To automatically load Cryogenic at boot time, edit the file `/etc/modules`, which contains a list of modules that must be loaded at boot time, and append the following line:

```
pm-rpi
```

Next, create a directory under `/lib/modules/3.6.11+/kernel/drivers/` called `pm-rpi/` and place here the object file `pm-rpi.ko`.

Finally, run the following command to rebuild the dependencies:

```
# depmod -a
```

Reboot the system and Cryogenic will be automatically loaded.

C Library for Broadcom BCM2835 Installation

In order to manage the GPIO pins, the bcm2835 library must be installed on the Raspberry Pi. During our experimentation we used the version 1.32, that can be downloaded running the following command:

```
wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.32.tar.gz
```

Now, we only need to extract the files, run the configuration script, compile and install:

```
tar zxvf bcm2835-1.xx.tar.gz
cd bcm2835-1.xx
./configure
make
sudo make check
sudo make install
```

4.2.2. Circuit assembly

The assembly of the circuit in Figure 4.2 is achieved through the pin header embedded in the Raspberry Pi. Figure 4.3 shows the pin header layout.

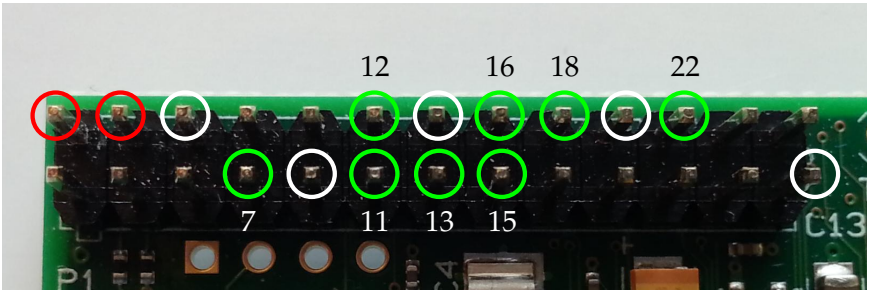


Figure 4.3.: Pin header layout

The pins surrounded by red circles are power pins of 5V, the ones surrounded by white circles are ground pins, and the ones surrounded by green circles are GPIO pins. The rest are not important for our experimentation.

Thus, we should connect the positive pole of the power supply to one of the 5V pins, and the negative pole to one of the ground pins. It is important to be aware of the polarity and supply no more than 5V. Otherwise, the Pi could be damaged since there is no over-voltage protection on the board.

Next, we need to connect the voltage probe to one of the GPIO pins, whose behaviour can be controlled through software. The ground clamp of the probe should also be connected to one of the ground pins. The number next to each GPIO pin in Figure 4.3 is its identifier inside the pin header and is the number that we must use to control the behaviour of each specific pin.

Finally, place the current probe around one of the cables that connect the Pi to the power supply, taking care to match the current direction indicator with the actual direction in the circuit. There is no danger of damage, but the signal of the sample would be inverted in case of placing it wrongly.

4.3. Test programs

Four shellscripts have been created in order to simulate different scenarios where the operation of Cryogenic may vary. These scripts execute four instances of different versions of the UDP client presented in Section 3.3.1, let them run for 60 seconds and then terminate their execution.

The versions of the UDP client are the following:

`client.c [period (ms)]`

Sends a UDP packet at every interval of time defined by the input parameter, which defines a period.

`client-rand.c [period (ms)]`

Sends a UDP packet at a random time between the last transmission and two times the period defined by the input parameter.

`client-cryo.c [interface] [period (ms)] [tolerance (ms)]`

Sends a UDP packet at every interval of time defined by the period plus/minus half the tolerance. The actual time is determined by Cryogenic. The interface parameter is the name of the active interface that is used to send the packets.

For all versions, we selected prime numbers for the period and the tolerance input parameters. This way, we make the appearance of collisions to be unlikely.

Next, we describe the shellscripts:

`test1.sh [interface]`

Runs two instances of client.c and two instances of client-cryo.c.

`test1r.sh [interface]`

Runs one instance of `client.c`, one instance of `client-rand.c` and two instances of `client-cryo.c`.

`test2.sh`

Runs four instances of `client.c`.

`test2r.sh`

Runs one instance of `client-rand.c` and three instances of `client.c`.

These scripts also undertake to call the C programs that set the corresponding GPIO pin to its high and low values when they start and finish their execution. These C programs, as well as the UDP client versions and the scripts can be consulted in the Appendix of this thesis.

4.4. Results

In this section we present the results of our research. Each experiment consists of a batch run of the scripts presented in Section 4.3 in which we vary different factors, like the frequency of packet transmission or the size of the packets sent.

In order to obtain results statistically significant, each script is executed ten times for every experiment, and then the mean and the standard deviation of each set of results are calculated.

While all these experiments were running, an HDMI monitor and a wired USB keyboard were plugged to the Pi. The network device used was an *Edimax EW-7811Un* wireless USB adapter.

4.4.1. Baseline consumption

A first batch run enacted by the authors of [6] let us see that the baseline power consumption of the Raspberry Pi was rather high compared to the specific power consumption of the WiFi device. As a result, the percentage of consumption decrease was low and did not reflect the real savings.

To solve this situation, we decided to measure first the baseline consumption of the Pi with the WiFi device plugged but not connected to any wireless network. Then, for each experiment, we subtract this baseline to the total energy consumed and calculate the savings for the WiFi device only. Table 4.1 illustrates the results of these measurements. The script `baseline.sh`, whose code is included in Section B.3.1, just sets the high value for the GPIO pin, waits for 60 seconds and sets the low value for the pin again.

#	Baseline
1	112.869877 J
2	112.934779 J
3	113.864178 J
4	114.061811 J
5	114.026117 J
6	114.005366 J
7	112.675130 J
8	112.943434 J
9	113.727830 J
10	113.918938 J
Mean	**113.502746 J**
Standard Deviation	0.569063 J

Table 4.1.: Baseline consumption results

4.4.2. Experiment 1

For this experiment, we connected the WiFi device to the network *eduroam*[2] and set the packet payload to 512 bytes. The period of packet transmission for every client and the tolerance for the ones that use Cryogenic is shown in Table 4.2, expressed in milliseconds. As we already mentioned, prime numbers are used in order to avoid collisions and the tolerance window is the 50% of the period.

no Randomization				Randomization			
Cryogenic[1]		no Cryogenic[1r]		Cryogenic[2]		no Cryogenic[2r]	
client	5003	client	5003	client-rand	5003	client	5003
client	3533	client	3533	client	3533	client	3533
client-cryo	3001 1511	client	3001	client-cryo	3001 1511	client	3001
client-cryo	2503 1249	client	2503	client-cryo	2503 1249	client	2503

Table 4.2.: Transmission period and tolerance in milliseconds for Experiment 1

[2] `https://www.eduroam.org/`

The superscripts on the header of the table denote which of the shellscripts presented in Section 4.3 is executed in each case: *1* corresponds to `test1.sh`, *1r* to `test1r.sh`, *2* to `test2.sh` and *2r* to `test2r.sh`. This notation is applied throughout this section.

The results obtained for these values are listed in Table 4.3. The row labeled as "Device consumption" corresponds to the subtraction of the baseline, calculated in the previous section, to the mean. The "Savings" row is the percentage of decrease of the Cryogenic test programs with respect to the non-Cryogenic ones, distinguishing between the no randomized and the randomized cases.

	no Randomization		Randomization	
#	Cryogenic^{1}	no Cryogenic^{1r}	Cryogenic^{2}	no Cryogenic^{2r}
1	118.231250 J	119.751021 J	119.356147 J	118.056000 J
2	117.313994 J	119.984405 J	117.449686 J	119.808370 J
3	116.243912 J	117.578811 J	117.572424 J	117.117942 J
4	117.970088 J	119.250669 J	118.407992 J	118.908278 J
5	119.608451 J	120.404512 J	119.870179 J	120.257686 J
6	119.873042 J	119.942403 J	118.956864 J	120.526507 J
7	117.779464 J	119.748845 J	118.428992 J	119.524538 J
8	119.651749 J	120.710930 J	119.631208 J	120.879302 J
9	120.822664 J	121.485627 J	120.808594 J	125.580573 J
10	120.891067 J	121.814166 J	122.029147 J	125.100955 J
Mean	118.838568 J	120.067139 J	119.251123 J	120.576015 J
Device consumption	5.335822 J	6.564393 J	J 5.748377 J	7.073269 J
Standard deviation	1.555452 J	1.186711 J	1.422768 J	2.758853 J
Savings	**18.72%**		**18.73%**	

Table 4.3.: Results of Experiment 1

As expected the test programs that use Cryogenic consume less energy than the others, almost 20% in both cases. Nevertheless, taking into account the consumption values for the WiFi device, we consider that the standard deviation obtained is high for all the sets, especially for the non-Cryogenic test program that uses randomization, and this makes these results less precise.

A possible explanation for such different values when running the same script is the usage of *eduroam*, a wireless network that may have a large numbers of users generating traffic that could cause interference. In order to work around this issue, we used a different network for the following experiments.

4.4.3. Experiment 2

In this experiment, we connected the Pi to a protected wireless network specifically created for the experimentation. This way, we guaranteed that any other user was able to connect and cause interferences to the network.

The rest of parameters remained untouched: the transmission periods and tolerances were the ones displayed in Table 4.2 and the payload size was 512 bytes. Table 4.4 illustrates the results for the new testing configuration.

Looking at these results, the first thing we notice is that the consumption specific to the WiFi device has increased between 2 and 3 joules. Several factors may explain this increase. For instance, throughout the experimentation we noticed that the overall consumption usually increases when the Pi has been on for a long period of time. There also exist other factors related to the new network, like the router speed or the encryption used, that might cause different packet overheads.

	no Randomization		Randomization	
#	Cryogenic[1]	no Cryogenic[1r]	Cryogenic[2]	no Cryogenic[2r]
1	121.903248 J	121.550464 J	120.515872 J	121.008448 J
2	121.400528 J	121.399952 J	122.224240 J	121.673120 J
3	121.979152 J	120.989200 J	121.806688 J	121.534880 J
4	121.531632 J	121.297056 J	120.440608 J	122.748688 J
5	122.751792 J	123.730816 J	121.679616 J	123.309792 J
6	122.269840 J	122.510320 J	121.645088 J	122.855616 J
7	122.249616 J	122.806528 J	123.776688 J	123.280640 J
8	122.462464 J	123.320656 J	121.963504 J	122.951008 J
9	121.602608 J	122.018160 J	122.188176 J	123.209472 J
10	122.392544 J	122.096432 J	121.166000 J	122.385648 J
Mean	122.054342 J	122.171958 J	121.740648 J	122.495731 J
Device consumption	8.551596 J	8.669212 J	8.237902 J	8.992985 J
Standard deviation	0.445371 J	0.909376 J	0.952592 J	0.817383 J
Savings	**1.36%**		**8.40%**	

Table 4.4.: Results of Experiment 2

Although the energy consumption has increased, the savings in these experiments are more moderate, especially in the case of no randomization, where the decrease is smaller than 2%. However, for the not randomized case, the standard deviation of the non-Cryogenic test programs is almost two times the deviation of the Cryogenic tests, and this could have affected to the final result.

In spite of this difference, it is important to note that the standard deviation is now smaller than 1 joule in all cases. This fact, along with the overall consumption increase, makes these results to be more precise than the results of the first experiment.

Since Cryogenic may defer the transmission of some packets, we wanted to check that the savings obtained were not due to the transmission of fewer packets. Therefore, we decided to calculate the energy consumption per packet sent. Table 4.5 illustrates the results.

For each test program, the table displays the WiFi device consumption and the number of packets sent in every run. With these values, we calculate the consumption per packet and obtain the mean and the standard deviation of each set. Again, The "Savings/Packet" row is the percentage of decrease of the Cryogenic results with respect to the non-Cryogenic results.

As we suspected, the test programs that use Cryogenic sent fewer packets than the rest. However, the usual behaviour is that a Cryogenic client sends only one less packet than a non-Cryogenic one, and this is not a consequence of the deferment, but a consequence of the client design. It is easy to notice this issue just comparing the code of both versions:

`client.c` sends a packet right after the main loops starts, waits a period, sends the following packet and so on; in contrast, `client-cryo.c` assigns the delay and the timeout and calls `select` at the beginning of the loop, forcing the first transmission to wait.

	no Randomization					
#	Cryogenic[1]			no Cryogenic[1r]		
1	8.400502 J	71 pkt	0.118317 J	8.047718 J	73 pkt	0.110243 J
2	7.897782 J	71 pkt	0.111236 J	7.897206 J	73 pkt	0.108181 J
3	8.476406 J	73 pkt	0.116115 J	7.486454 J	73 pkt	0.102554 J
4	8.028886 J	71 pkt	0.113083 J	7.794310 J	73 pkt	0.106771 J
5	9.249046 J	71 pkt	0.130268 J	10.228070 J	73 pkt	0.140111 J
6	8.767094 J	71 pkt	0.123480 J	9.007574 J	73 pkt	0.123391 J
7	8.746870 J	71 pkt	0.123195 J	9.303782 J	73 pkt	0.127449 J
8	8.959718 J	71 pkt	0.126193 J	9.817910 J	73 pkt	0.134492 J
9	8.099862 J	71 pkt	0.114083 J	8.515414 J	73 pkt	0.116650 J
10	8.889798 J	71 pkt	0.125208 J	8.593686 J	73 pkt	0.117722 J
		Mean	0.120118 J		Mean	0.118756 J
	Standard deviation		0.006418 J	Standard deviation		0.012457 J
Savings/Packet	**-1.15%**					
	Randomization					
#	Cryogenic[2]			no Cryogenic[2r]		
1	7.013126 J	76 pkt	0.092278 J	7.505702 J	78 pkt	0.096227 J
2	8.721494 J	76 pkt	0.114757 J	8.170374 J	78 pkt	0.104748 J
3	8.303942 J	76 pkt	0.109262 J	8.032134 J	78 pkt	0.102976 J
4	6.937862 J	76 pkt	0.091288 J	9.245942 J	78 pkt	0.118538 J
5	8.176870 J	76 pkt	0.107590 J	9.807046 J	78 pkt	0.125731 J
6	8.142342 J	76 pkt	0.107136 J	9.352870 J	78 pkt	0.119909 J
7	10.273942 J	76 pkt	0.135183 J	9.777894 J	78 pkt	0.125358 J
8	8.460758 J	76 pkt	0.111326 J	9.448262 J	78 pkt	0.121132 J
9	8.685430 J	76 pkt	0.114282 J	9.706726 J	78 pkt	0.124445 J
10	7.663254 J	76 pkt	0.100832 J	8.882902 J	78 pkt	0.113883 J
		Mean	0.108393 J		Mean	0.115295 J
	Standard deviation		0.012534 J	Standard deviation		0.010479 J
Savings/Packet	**5.99%**					

Table 4.5.: Consumption per packet for Experiment 2

Regarding the consumption per packet, the results are contradictory since we got a smaller consumption for Cryogenic packets when using randomization but greater when not. As we pointed for the overall consumption in the not randomized case, the standard deviation for for the non-Cryogenic result is almost two times the deviation of the Cryogenic one, which may explain this unexpected result.

In order to see how Cryogenic modifies the transmission time of some packets, in this experiment we plotted the packets sent by the test programs that are not randomized. The result is displayed in Figure 4.4.

We can see how, for both test programs, the transmission of the packets that belong to

client 1 and 2 coincide, since the periods are the same and none of them use Cryogenic. In contrast, the transmissions of client 3 and 4 do not coincide, since these use Cryogenic. It is easy to notice how packets defer or anticipate their transmission so that they coincide with the transmission of other packets.

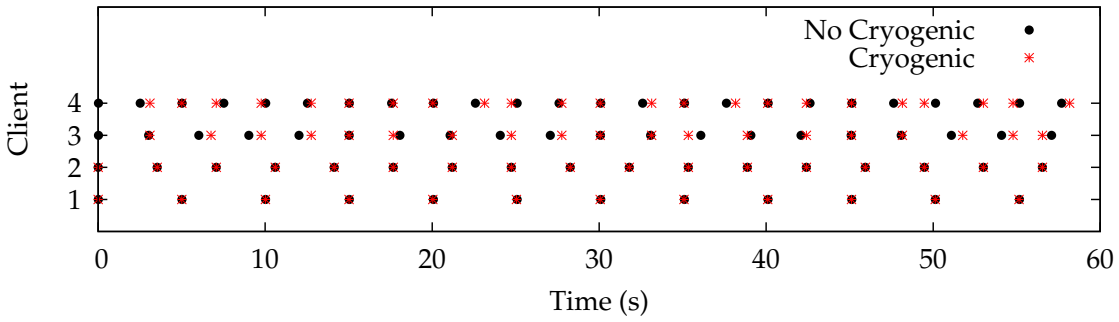


Figure 4.4.: Packet transmission times for not randomized test programs

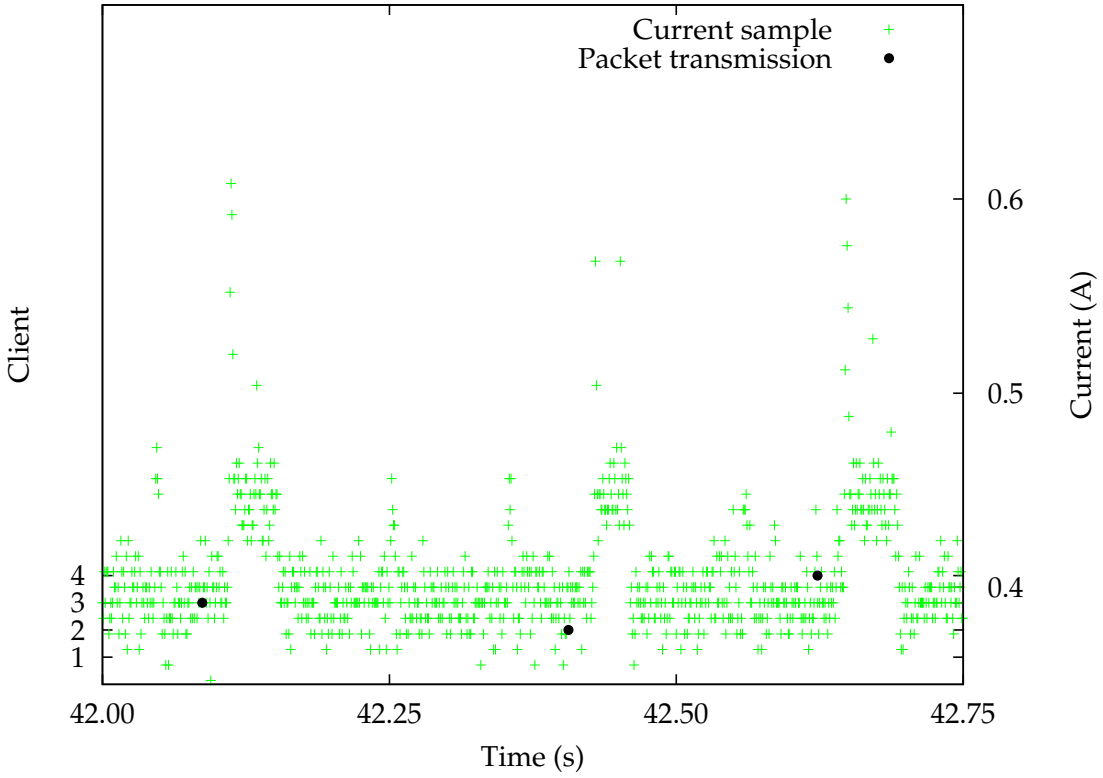


Figure 4.5.: Current draw of non-Cryogenic test program

Due to the large number of current samples taken and the short transmission periods, it would be difficult to see the impact of these modified transmissions on the energy con-

sumption if we plotted the current draw for the whole experiment duration. For this reason, Figure 4.5 illustrates the current draw between seconds 42 and 42.75 for non-Cryogenic test program.

We can see the transmission of three packets, separated some milliseconds, corresponding from left to right to client 3, 2 and 4. Each transmission provokes a peak in the current draw, that will be translated in a peak of energy.

When running the Cryogenic test program, the packet sent by client 3 defers its transmission, and the transmission of the packet that belongs to client 4 is anticipated, so that both packets and the packet that belongs to client 2, which does not use Cryogenic, are sent almost at the same time. This behaviour is illustrated in Figure 4.6.

As we can see, the number of current peaks decreases from three to only one, consumings as a result less energy than the non-Cryogenic test program in the same period of time.

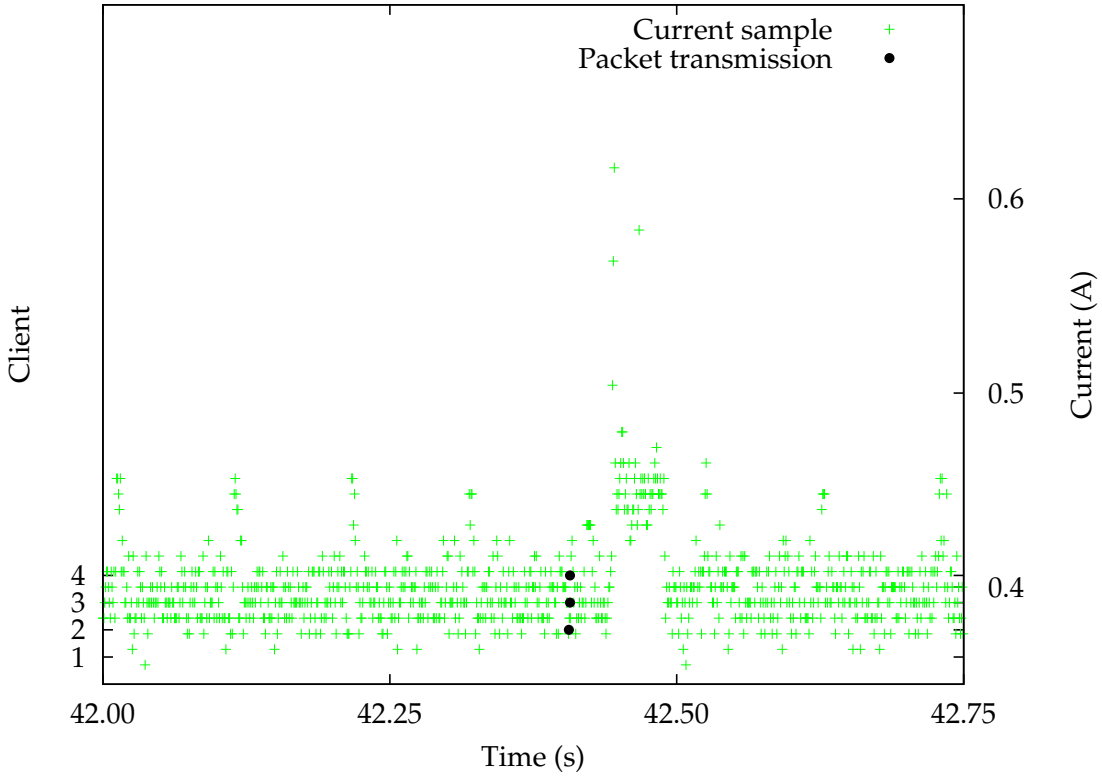


Figure 4.6.: Current draw of Cryogenic test program

4.4.4. Experiment 3

In order to boost the energy consumption of the WiFi device, in this experiment we modified the period of packet transmission so that packets were sent at double frequency. Table 4.6 illustrates the new period and tolerance values. Note that we keep using prime numbers and the tolerance is still 50% of the period.

no Randomization					Randomization				
Cryogenic[1]			no Cryogenic[1r]		Cryogenic[2]			no Cryogenic[2r]	
client	2503		client	2503	client-rand	2503		client	2503
client	1747		client	1747	client	1747		client	1747
client-cryo	1499	751	client	1499	client-cryo	1499	751	client	1499
client-cryo	1249	631	client	1249	client-cryo	1249	631	client	1249

Table 4.6.: Transmission period and tolerance in milliseconds for Experiment 3

The results of Experiment 3 are illustrated in Table 4.7. From this experiment on, both the overall savings and the savings per packet are displayed in the same table and only the relevant data to interpret the result is included.

	no Randomization		Randomization	
	Cryogenic[1]	no Cryogenic[1r]	Cryogenic[2]	no Cryogenic[2r]
Total consumption	122.831509 J	123.508019 J	123.456717 J	123.825835 J
Device consumption	9.328763 J	10.005273 J	9.953971 J	10.323089 J
Standard deviation	0.454201 J	0.347781 J	0.423930 J	0.478646 J
Savings	**6.76%**		**3.58%**	
Consumption/Packet	0.064243 J	0.067833 J	0.066630 J	0.068227 J
Standard deviation	0.002991 J	0.002348 J	0.002904 J	0.003110 J
Savings/Packet	**5.29%**		**2.34%**	

Table 4.7.: Results of Experiment 3

Although we expected to obtain a consumption two times greater than in Experiment 2, the increase is smaller than a 20% in all cases. This issue, along with the unexpected consumption increase between Experiment 1 and Experiment 2, makes us think that a factor not related to the network made the consumption in Experiment 2 to be greater than usual.

On the other hand, it is easy to notice that, so far, this is the experiment with best results in terms of standard deviation, which confirms our suppositions about the unpredictable behaviour of *eduroam*. Again, the consumption is smaller for the test programs that use Cryogenic, and the same happens for the consumption per packet.

4.4.5. Experiment 4

The target of this experiment was to test how the size of the packets sent affects to the consumption. Thus, we increased the payload size from 512 to 1000 bytes and kept the period untouched. The results are illustrated in Table 4.8.

As we can see, the consumption not only has not increased, but has slightly decreased. Nevertheless, we consider this decrease is not significant, since it is caused because of the division of this experiment in two different days. We already pointed that the overall consumption usually increases when the Pi has been on for a long time, which was the case for runs 1 to 6. Runs 7 to 10 were performed on a different day, when the Pi had been

on for less time. This explains the consumption decrease, as well as the worse values for standard deviation.

	no Randomization		Randomization	
	Cryogenic[1]	no Cryogenic[1r]	Cryogenic[2]	no Cryogenic[2r]
Total consumption	122.641411 J	122.945605 J	122.614053 J	123.674467 J
Device consumption	9.138665 J	9.442859 J	9.111307 J	10.171721 J
Standard deviation	1.708826 J	1.151604 J	1.138469 J	1.308511 J
Savings	**3.22%**		**10.43%**	
Consumption/Packet	0.063025 J	0.063935 J	0.061150 J	0.067326 J
Standard deviation	0.011785 J	0.007803 J	0.007641 J	0.008721 J
Savings/Packet	**1.42%**		**9.17%**	

Table 4.8.: Results of Experiment 4

Thus, we can conclude that the increase of the payload size did not significantly affect to the energy consumed. Obviously, we cannot generalize this conclusion to any increase, since the transmission of packets with size greater than the network's MTU would cause the fragmentation of these packets, sending as a result more packets.

In spite of obtaining values slightly greater for the standard deviation in this experiment, the consumption is still smaller for the test programs that use Cryogenic. We can also observe that the consumption per packet is smaller for the Cryogenic clients.

Furthermore, we notice that the values obtained are similar to the values of consumption per packet obtained in Experiment 3, in contrast to the values obtained in Experiment 2, which are are almost two times greater.

4.4.6. Experiment 5

For this last experiment we preserved the payload size and halved again the transmission period. Table 4.9 shows the new used values.

no Randomization					Randomization				
Cryogenic[1]			no Cryogenic[1r]		Cryogenic[2]			no Cryogenic[2r]	
client	1249		client	1249	client-rand	1249		client	1249
client	877		client	877	client	877		client	877
client-cryo	751	373	client	751	client-cryo	751	373	client	751
client-cryo	631	313	client	631	client-cryo	631	313	client	631

Table 4.9.: Transmission period and tolerance in milliseconds for Experiment 5

The results of this experiment are displayed in Table 4.10. This time, we can observe how increasing the transmission frequency a 100% caused an increase of energy consumption by about 100% as well. The standard deviation slightly increased compared to Experiment 4, but the increase is smaller than the consumption increase, so we consider they are rather significant results.

	no Randomization		Randomization	
	Cryogenic[1]	no Cryogenic[1r]	Cryogenic[2]	no Cryogenic[2r]
Total consumption	133.103653 J	133.881750 J	132.757357 J	133.363406 J
Device consumption	19.600907 J	20.379004 J	19.254611 J	19.860660 J
Standard deviation	0.698003 J	1.194147 J	1.078104 J	0.670387 J
Savings	**3.82%**		**3.05%**	
Consumption/Packet	0.067543 J	0.069791 J	0.065247 J	0.066871 J
Standard deviation	0.002417 J	0.004090 J	0.003635 J	0.002257 J
Savings/Packet	**3.22%**		**2.43%**	

Table 4.10.: Results of Experiment 5

As before, the energy consumption is smaller for the programs that use Cryogenic, and the same happens for the consumption per packet. The final consumption per packet is also similar to the values obtained in Experiments 3 and 4.

5. Conclusions & Future Work

Throughout the experimentation we were able to notice that several factors may affect the energy measurements. We already pointed that using different networks or performing experiments at different times with respect to the moment when the Pi was powered on may cause variations on the results. The WiFi device model used for the experiments may also have an impact, as well as the current probe tolerance, its initial calibration accuracy, or the oscilloscope's sampling frequency.

However, there is a clear trend to reduce the energy consumed by the WiFi device. Leaving apart Experiment 1, performed with a different network that led to imprecise results, we achieved savings between 1% and 10% for the consumption specific to the WiFi device. We also made sure that these savings were not a consequence of doing less work by measuring the consumption per packet sent. The results are similar: between 1% and 9% of savings.

The reduction achieved is moderate, but the modifications applied on the original application are simple and localized, as we presented in Section 3.3. Therefore, the effort needed to benefit from Cryogenic is fair compared to the savings obtained. Besides, this reduction also accomplishes our initial expectations, since our goal was the achievement of savings while performing the same amount of work. It is important to point as well that the Cryogenic test programs were not completely flexible to obtain savings, since only half the packets sent were actually using Cryogenic.

As a further step, we plan to perform more energy measurements using a GSM radio, a common device nowadays for smartphones. We are also willing to integrate Cryogenic with more existing applications and to submit the patch to the Linux Kernel Mailing List, in order to achieve the inclusion of Cryogenic in the mainline Linux kernel.

Appendix

A. Module

This chapter presents the source code of Cryogenic.

```
#include <linux/module.h>
#include <linux/cdev.h>
#include <linux/kallsyms.h>
#include <linux/netdevice.h>
#include <linux/poll.h>
#include <scsi/scsi_device.h>
#include <scsi/scsi_eh.h>

#define MAX_DEVICES 10
#define FREE_SLOT -1
#define DEVICE_NAME "cryogenic"
#define PM_IOC_MAGIC 'k'
#define SET_DELAY_AND_TIMEOUT _IOW(PM_IOC_MAGIC, 1, struct pm_times)
#define SCSI_TIMEOUT (2*HZ)
#define MAX_SERIAL_NUMBER_SIZE 21


struct pm_device {
    int minor;
    const char *name;
    struct cdev pm_cdev;
    struct device *dev;
    wait_queue_head_t event_queue;
    int unplugged;
    request_fn_proc *scsi_request_fn_address;
    unsigned char serial_number[MAX_SERIAL_NUMBER_SIZE];
    int scsi_cdev_open;
    struct net_device_ops my_ops;
    const struct net_device_ops *old_ops;
};


struct pm_private {
    int first_poll;
    unsigned long min_delay;
    struct timer_list timer;
    int timer_added;
    struct pm_device *pm_dev;
};


struct pm_times {
```

```
43      unsigned long delay_msecs;
44      unsigned long timeout_msecs;
45  };
46
47
48  static int major;
49  static int device_open;
50  static struct class *pm_class;
51  static struct pm_device *pm_devices;
52  static void *scsi_uevent_address;
53  static void *net_uevent_address;
54  static unsigned int scsi_inq_timeout = SCSI_TIMEOUT/HZ + 18;
55
56  static int pm_open(struct inode *, struct file *);
57  static int pm_release(struct inode *, struct file *);
58  static unsigned int pm_poll(struct file *, struct poll_table_struct *);
59  static long pm_ioctl(struct file *, unsigned int, unsigned long);
60
61  static int scsi_uevent_interceptor(struct device *dev,
62                                     struct kobj_uevent_env *env);
63  static int net_uevent_interceptor(struct device *dev,
64                                    struct kobj_uevent_env *env);
65  static void request_fn_interceptor(struct request_queue *);
66  static netdev_tx_t ndo_start_xmit_interceptor(struct sk_buff *,
67                                                struct net_device *);
68
69  static int create_device(struct pm_device *, struct device *, int);
70  static int remove_device(struct pm_device *);
71  static void clean_module(void);
72  static int assign_scsi_devices(struct device *, void *);
73  static int set_scsi_serial_number(struct scsi_device *,
74                                    unsigned char *);
75  static int for_each_net_device(int *);
76  static void wake_up_tasks(wait_queue_head_t *);
77  static void timeout_wake_up(unsigned long);
78  static void enable_hotplugging(void);
79  static void disable_hotplugging(void);
80  static void plug_device(struct device *);
81  static void unplug_device(struct device *);
82  static int scsi_device_reconnected(struct device *);
83
84  MODULE_AUTHOR("A. Morales Ruiz");
85  MODULE_LICENSE("GPL");
86
87  extern struct bus_type scsi_bus_type;
88
89  static struct file_operations pm_fops = {
90      .owner = THIS_MODULE,
91      .open = pm_open,
92      .release = pm_release,
93      .poll = pm_poll,
94      .unlocked_ioctl = pm_ioctl
```

```
95  };
96
97
98  /* ***** init/exit methods ***** */
99
100
101 /*
102  * Gets a range of minor numbers [0..MAX_DEVICES],
103  * creates the device class, allocates an array for
104  * MAX_DEVICES devices and marks all slots as free,
105  * creates existing devices, enables hotplugging
106  * and marks the device as not open
107  */
108 static int __init pm_init(void)
109 {
110     dev_t tmp_dev;
111     int err;
112     int i;
113     int n;
114
115     tmp_dev = 0;
116     err = alloc_chrdev_region(&tmp_dev, 0, MAX_DEVICES, DEVICE_NAME);
117     if (err < 0) {
118         printk(KERN_WARNING "Cryogenic could not be loaded
119                [alloc_chrdev_region() failed].\n");
120         return err;
121     }
122
123     major = MAJOR(tmp_dev);
124
125     pm_class = class_create(THIS_MODULE, DEVICE_NAME);
126     if (IS_ERR(pm_class)) {
127         clean_module();
128         printk(KERN_WARNING "Cryogenic could not be loaded
129                [class_create() failed].\n");
130         return PTR_ERR(pm_class);
131     }
132
133     pm_devices = (struct pm_device *)kzalloc(MAX_DEVICES*sizeof(struct
        pm_device), GFP_KERNEL);
134     if (pm_devices == NULL) {
135         clean_module();
136         printk(KERN_WARNING "Cryogenic could not be loaded
137                [kzalloc() failed].\n");
138         return -ENOMEM;
139     }
140     for (i = 0; i < MAX_DEVICES; ++i) {
141         pm_devices[i].minor = FREE_SLOT;
142         pm_devices[i].unplugged = FREE_SLOT;
143     }
144
145     n = 0;
```

```
146     err = bus_for_each_dev(&scsi_bus_type, NULL, (void *) &n,
147                            assign_scsi_devices);
148     if (err < 0) {
149         clean_module();
150         printk(KERN_WARNING "Cryogenic could not be loaded
151                [assign_scsi_devices() failed].\n");
152         return err;
153     }
154 
155     err = for_each_net_device(&n);
156     if (err < 0) {
157         clean_module();
158         printk(KERN_WARNING "Cryogenic could not be loaded
159                [for_each_net_device_() failed].\n");
160         return err;
161     }
162 
163     enable_hotplugging();
164 
165     device_open = 0;
166 
167     printk(KERN_INFO "Cryogenic was loaded [MAJOR number %d].\n",
168            major);
169 
170     return 0;
171 
172 }
173 
174 
175 /*
176  * Cleans all the module data and disables hotplugging
177  */
178 static void __exit pm_exit(void)
179 {
180     if (!device_open) {
181         clean_module();
182         disable_hotplugging();
183         printk(KERN_INFO "Cryogenic was unloaded.\n");
184     }
185     else printk(KERN_INFO "Cryogenic is in use.\n");
186 
187 }
188 
189 
190 /* ***** new system call methods ***** */
191 
192 
193 /*
194  * Allocates and initialises private_data and increases the counter
195  */
196 static int pm_open(struct inode *inode, struct file *filp)
197 {
```

```
198     struct pm_private *priv;
199     int minor = iminor(filp->f_dentry->d_inode);
200
201     priv = kzalloc (sizeof (struct pm_private), GFP_KERNEL);
202     if (priv == NULL) {
203         printk(KERN_WARNING "Cryogenic: Device could not be opened
204                [kzalloc() failed].\n");
205         return -ENOMEM;
206     }
207
208     filp->private_data = priv;
209     priv->pm_dev = &pm_devices[minor];
210     priv->timer.function = timeout_wake_up;
211     priv->timer.data = (unsigned long) priv;
212     priv->timer_added = 0;
213     init_timer(&priv->timer);
214
215     if (scsi_is_sdev_device(pm_devices[minor].dev))
216         pm_devices[minor].scsi_cdev_open++;
217     device_open++;
218     try_module_get(THIS_MODULE);
219
220     return 0;
221 }
222
223
224 /*
225  * Frees private_data and decreases the counter
226  */
227 static int pm_release(struct inode *inode, struct file *filp)
228 {
229     struct pm_private *priv = filp->private_data;
230     int minor = iminor(filp->f_dentry->d_inode);
231
232     if (priv->timer_added) {
233         del_timer(&priv->timer);
234         priv->timer_added = 0;
235     }
236     kfree(priv);
237
238     /* We check if it is a scsi device this way because
239      * if the device has been unplugged, the call to
240      * scsi_is_sdev_device may return NULL */
241     if (pm_devices[minor].scsi_request_fn_address != NULL) {
242         pm_devices[minor].scsi_cdev_open--;
243         if (pm_devices[minor].unplugged == 1 &&
244             pm_devices[minor].scsi_cdev_open == 0) {
245             remove_device(&pm_devices[minor]);
246             printk(KERN_INFO "Cryogenic: Device %s was removed.\n",
247                    pm_devices[minor].serial_number);
248             pm_devices[minor].minor = FREE_SLOT;
249             pm_devices[minor].unplugged = FREE_SLOT;
```

```
250             }
251         }
252         device_open--;
253         module_put(THIS_MODULE);
254 
255         return 0;
256     }
257 
258 
259     /*
260      * Returns whether a device can be used (> 0) or not (0)
261      * In case it can't be used, the call is queued and a timer is added
262      */
263     static unsigned int pm_poll(struct file *filp, struct poll_table_struct
        *table)
264     {
265         struct pm_private *priv = filp->private_data;
266         int minor = iminor(filp->f_dentry->d_inode);
267 
268         /*
269          * unplugged == 1 -> return that poll is ready, delegate error
             handling
270          *
271          * first_poll == 0 -> timeout expired or call from interceptor
272          * otherwise -> first time we call poll
273          */
274         if (pm_devices[minor].unplugged == 1 ||
275             ((jiffies >= priv->min_delay) && !priv->first_poll)) {
276             if (priv->timer_added) {
277                 del_timer(&priv->timer);
278                 priv->timer_added = 0;
279             }
280 
281             return POLLIN | POLLRDNORM | POLLOUT | POLLWRNORM;
282         }
283 
284         priv->first_poll = 0;
285         poll_wait(filp, &pm_devices[minor].event_queue, table);
286 
287         /* If timeout == MAX, do NOT add_timer */
288         if (!priv->timer_added && priv->timer.expires > jiffies) {
289             add_timer(&priv->timer);
290             priv->timer_added = 1;
291         }
292 
293         return 0;
294     }
295 
296 
297     /*
298      * Sets the min delay and the maximum timeout
299      * We assume time is given in milliseconds
```

```
 * jiffies = msec*HZ/1000
 */
static long pm_ioctl(struct file *filp, unsigned int cmd, unsigned long
    arg)
{
    unsigned long j;
    struct pm_times *times;
    struct pm_private *priv = filp->private_data;
    priv->first_poll = 1;
    if (priv->timer_added) {
        del_timer(&priv->timer);
        priv->timer_added = 0;
    }
    switch(cmd) {
    case SET_DELAY_AND_TIMEOUT:
        if (!access_ok(VERIFY_WRITE, (void *) arg, sizeof(struct
            pm_times)))
            return -EFAULT;
        times = (struct pm_times *) arg;
        if (times->delay_msecs > times->timeout_msecs)
            return -EINVAL;
        j = jiffies;
        priv->min_delay = j + (times->delay_msecs * HZ)/1000;
        priv->timer.expires = j + (times->timeout_msecs * HZ)/1000;
        break;
    default:
        return -ENOTTY;
    }

    return 0;
}


/* ***** interceptors ***** */


/*
 * Dynamically adds or removes scsi devices when
 * these are plugged/unplugged
 */
static int scsi_uevent_interceptor(struct device *dev,
                   struct kobj_uevent_env *env)
{
    typedef int uevent(struct device *dev, struct kobj_uevent_env *env);
    uevent *original_uevent;
    int res;

    if (scsi_is_sdev_device(dev) &&
        !strncmp(env->envp[0], "ACTION=add", 10)) {
            plug_device(dev);

    }
```

```
350     else if (scsi_is_sdev_device(dev) &&
351              !strncmp(env->envp[0], "ACTION=remove", 13))
352         unplug_device(dev);
353 
354     original_uevent = (uevent *) scsi_uevent_address;
355     res = original_uevent(dev, env);
356     return res;
357 }
358 
359 
360 /*
361  * Dynamically adds or removes network devices when
362  * these are plugged/unplugged
363  */
364 static int net_uevent_interceptor(struct device *dev,
365                   struct kobj_uevent_env *env)
366 {
367     typedef int uevent(struct device *dev, struct kobj_uevent_env *env);
368     uevent *original_uevent;
369     int res;
370 
371     if (env->envp_idx > 0 &&
372         !strncmp(env->envp[0], "ACTION=add", 10))
373         plug_device(dev);
374     else if (env->envp_idx > 0 &&
375              !strncmp(env->envp[0], "ACTION=remove", 13))
376         unplug_device(dev);
377 
378     original_uevent = (uevent *) net_uevent_address;
379     res = original_uevent(dev, env);
380     return res;
381 }
382 
383 
384 /*
385  * Intercepts the call to the request_fn method
386  * for every disk and calls its original method
387  */
388 static void request_fn_interceptor(struct request_queue *q)
389 {
390     int i;
391     request_fn_proc *address = NULL;
392     typedef void request(struct request_queue *);
393     request *original_request;
394 
395     for (i = 0; (i < MAX_DEVICES) && address == NULL; ++i) {
396         if (pm_devices[i].minor != FREE_SLOT) {
397             if (scsi_is_sdev_device(pm_devices[i].dev)) {
398                 struct scsi_device *sdev =
399                     to_scsi_device(pm_devices[i].dev);
400                 if (sdev->request_queue == q)
401                     address = pm_devices[i].scsi_request_fn_address;
```

```
402                 }
403             }
404         }
405
406         if (address == NULL)
407             printk(KERN_WARNING "Cryogenic: Requested device
408                    was unplugged.");
409         else {
410             --i;
411             wake_up_tasks(&pm_devices[i].event_queue);
412             original_request = (request *) address;
413             original_request(q);
414         }
415     }
416
417
418     /*
419      * Intercepts the call to the ndo_start_xmit method
420      * for every network device and calls its original method
421      */
422     static netdev_tx_t ndo_start_xmit_interceptor(struct sk_buff *skb,
423                                                   struct net_device *dev)
424     {
425         void *address = NULL;
426         typedef netdev_tx_t xmit(struct sk_buff *skb, struct net_device *dev
                );
427         xmit *original_xmit;
428
429         int i;
430         for (i = 0; (i < MAX_DEVICES) && address == NULL; ++i) {
431             if (pm_devices[i].minor != FREE_SLOT) {
432                 if (!scsi_is_sdev_device(pm_devices[i].dev)) {
433                     struct net_device *netdev =
434                         to_net_dev(pm_devices[i].dev);
435                     if (netdev == dev) {
436                         address = pm_devices[i].old_ops->ndo_start_xmit;
437                     }
438                 }
439             }
440         }
441
442         if (address == NULL) {
443             printk(KERN_WARNING "Cryogenic: Requested device
444                    was unplugged.");
445             return NETDEV_TX_BUSY;
446         }
447         --i;
448         wake_up_tasks(&pm_devices[i].event_queue);
449         original_xmit = (xmit *) address;
450         return original_xmit(skb, dev);
451     }
452
```

```
453
454 /* ***** other methods ***** */
455
456
457 /*
458  * Sets fields of the pm_device struct and creates
459  * a new character device
460  */
461 static int create_device(struct pm_device *pm_dev,
462              struct device *dev, int minor)
463 {
464     int err;
465     struct device *device;
466     const char *name;
467     dev_t current_dev = MKDEV(major, minor);
468
469     pm_dev->minor = minor;
470     pm_dev->name = dev_name(dev);
471     pm_dev->unplugged = 0;
472     pm_dev->scsi_cdev_open = 0;
473
474     if (scsi_is_sdev_device(dev)) {
475         struct scsi_device *sdev = to_scsi_device(dev);
476         if (set_scsi_serial_number(sdev, pm_dev->serial_number) == 1)
477             name = pm_dev->serial_number;
478         else {
479             /* The scsi inquiry that gets the serial number of the
480              * device may sometimes fail. In this case, we name the
481              * char device after the scsi address and set the
482              * seral_number field to the null character */
483             name = pm_dev->name;
484             pm_dev->serial_number[0] = '\0';
485         }
486     }
487     else
488         name = pm_dev->name;
489
490     cdev_init(&pm_dev->pm_cdev, &pm_fops);
491     pm_dev->pm_cdev.owner = THIS_MODULE;
492     err = cdev_add(&(pm_dev->pm_cdev), current_dev, 1);
493     if (err < 0) {
494         printk(KERN_WARNING "Cryogenic: Error [cdev_add() failed].\n");
495         return err;
496     }
497
498     device = device_create(pm_class, NULL, current_dev, NULL,
499                            "cryogenic/%s", name);
500     if (IS_ERR(device)) {
501         cdev_del(&(pm_dev->pm_cdev));
502         printk(KERN_WARNING "Cryogenic: Error
503                [device_create() failed].\n");
504         return PTR_ERR(device);
```

```
505     }
506
507     pm_dev->dev = dev;
508
509     /* Initialise event wait queue */
510     init_waitqueue_head(&pm_dev->event_queue);
511
512     if (scsi_is_sdev_device(dev)) {
513         /* Save the original request_fn address and set
514          * the interceptor address */
515         struct scsi_device *sdev = to_scsi_device(dev);
516         pm_dev->scsi_request_fn_address =
517             sdev->request_queue->request_fn;
518         sdev->request_queue->request_fn = &request_fn_interceptor;
519         /* Unused fields */
520         memset(&pm_dev->my_ops, 0, sizeof(struct net_device_ops));
521         pm_dev->old_ops = NULL;
522     }
523     else {
524         /* Save the original ndo_start_xmit address and
525          * set the interceptor address */
526         struct net_device *netdev = to_net_dev(dev);
527         const struct net_device_ops *ops = netdev->netdev_ops;
528         pm_dev->old_ops = ops;
529         pm_dev->my_ops = *ops;
530         pm_dev->my_ops.ndo_start_xmit = &ndo_start_xmit_interceptor;
531         netdev->netdev_ops = &pm_dev->my_ops;
532         /* Unused field */
533         pm_dev->scsi_request_fn_address = NULL;
534     }
535
536     return 0;
537 }
538
539
540 /*
541  * If the device can be removed: restores addresses,
542  * wakes ups waiting tasks and destroys the character
543  * device. Returns 1. Otherwise, wakes up waiting
544  * tasks and returns 0.
545  */
546 static int remove_device(struct pm_device *pm_dev)
547 {
548     /* We check if it is a scsi device this way because
549      * if the device has been unplugged, the call to
550      * scsi_is_sdev_device may return NULL */
551     if (pm_dev->scsi_request_fn_address != NULL) {
552         /* pm_dev->serial_number[0] == '\0' means that the
553          * char device is named after the scsi address and
554          * it is destroyed even though it is open */
555         if (pm_dev->scsi_cdev_open > 0 &&
556             pm_dev->serial_number[0] != '\0') {
```

```
557                 wake_up_tasks(&pm_dev->event_queue);
558                 return 0;
559             }
560             /* If device has not been unplugged, restore its address */
561             if (pm_dev->unplugged == 0) {
562                 struct scsi_device *sdev = to_scsi_device(pm_dev->dev);
563                 sdev->request_queue->request_fn =
564                     pm_dev->scsi_request_fn_address;
565             }
566             pm_dev->scsi_request_fn_address = NULL;
567         }
568         else {
569             /* If device has not been unplugged, restore its address */
570             if (pm_dev->unplugged == 0) {
571                 struct net_device *netdev = to_net_dev(pm_dev->dev);
572                 netdev->netdev_ops = pm_dev->old_ops;
573             }
574             memset(&pm_dev->my_ops, 0, sizeof(struct net_device_ops));
575             pm_dev->old_ops = NULL;
576         }
577 
578         wake_up_tasks(&pm_dev->event_queue);
579 
580         device_destroy(pm_class, MKDEV(major, pm_dev->minor));
581         cdev_del(&(pm_dev->pm_cdev));
582 
583         return 1;
584     }
585 
586 
587     /*
588      * Cleans data before unloading the module
589      */
590     static void clean_module()
591     {
592         int i;
593 
594         if (pm_devices) {
595             for (i = 0; i < MAX_DEVICES; ++i) {
596                 if (pm_devices[i].minor != FREE_SLOT)
597                     remove_device(&pm_devices[i]);
598             }
599             kfree(pm_devices);
600         }
601 
602         if (pm_class)
603             class_destroy(pm_class);
604 
605         unregister_chrdev_region(MKDEV(major, 0), MAX_DEVICES);
606     }
607 
608 
```

```
609 /*
610  * Creates scsi devices
611  */
612 static int assign_scsi_devices(struct device *dev, void *d)
613 {
614     int *idx = ((int*) d);
615 
616     if (scsi_is_sdev_device(dev) && !strcmp(dev->driver->name, "sd")) {
617 
618         if (*idx < MAX_DEVICES) {
619             int err;
620             struct pm_device *pm_dev = &pm_devices[*idx];
621             err = create_device(pm_dev, dev, *idx);
622             if (err < 0) {
623                 pm_dev->minor = FREE_SLOT;
624                 return err;
625             }
626             ++(*idx);
627         }
628         else {
629             struct scsi_device *sdev = to_scsi_device(dev);
630             unsigned char buf[MAX_SERIAL_NUMBER_SIZE];
631             set_scsi_serial_number(sdev, buf);
632             printk(KERN_WARNING "Cryogenic: Device %s could not be
633                    created [No available slot].\n", buf);
634         }
635     }
636 
637     return 0;
638 }
639 
640 
641 /*
642  * Sets the scsi device serial number on buf
643  */
644 static int set_scsi_serial_number(struct scsi_device *sdev, unsigned
        char *buf)
645 {
646     struct scsi_sense_hdr sshdr;
647     unsigned char inq_result[255];
648     int result, resid;
649     unsigned char scsi_cmd[16];
650     int length = 255;
651     int i;
652     int j;
653     int k;
654 
655     scsi_cmd[0] = INQUIRY;
656     scsi_cmd[1] = 0x01;
657     scsi_cmd[2] = 0x80;
658     scsi_cmd[4] = length;
659 
```

```
660     for (k = 0; k < 3; ++k) {
661         result = scsi_execute_req(sdev, scsi_cmd, DMA_FROM_DEVICE,
662                                   inq_result,  length, &sshdr,
663                                   HZ / 2 + HZ * scsi_inq_timeout,
664                                   3, &resid);
665         if (!result) {
666             j = 0;
667             for (i = 0; i < MAX_SERIAL_NUMBER_SIZE-1; ++i) {
668                 if (inq_result[i+4] > 32) {
669                     buf[j] = inq_result[i+4];
670                     ++j;
671                 }
672             }
673             buf[j] = '\0';
674             return 1;
675         }
676     }
677     return 0;
678 }
679
680
681 /*
682  * Creates net devices
683  */
684 static int for_each_net_device(int *n)
685 {
686     struct net_device *netdev = first_net_device(&init_net);
687
688     while (netdev) {
689         const char *name = netdev->name;
690         if (!strncmp (name, "eth", 3) || !strncmp (name, "wlan", 4)) {
691             if (*n < MAX_DEVICES) {
692                 int err;
693                 struct pm_device *pm_dev = &pm_devices[*n];
694                 err = create_device(pm_dev, &netdev->dev, *n);
695                 if (err < 0) {
696                     pm_dev->minor = FREE_SLOT;
697                     return err;
698                 }
699                 ++(*n);
700             }
701             else
702                 printk(KERN_WARNING "Cryogenic: Device %s could not be
703                        created [No available slot].\n", name);
704         }
705         netdev = next_net_device(netdev);
706     }
707
708     return 0;
709 }
710
711
```

```
712 /*
713  * Checks if queue is active and wakes up processes
714  */
715 static void wake_up_tasks(wait_queue_head_t *queue)
716 {
717     if (waitqueue_active(queue))
718         wake_up(queue);
719 }
720
721
722 /*
723  * Function called when the timer expires
724  * Wakes up devices that were waiting on the event_queue
725  */
726 static void timeout_wake_up(unsigned long private_data)
727 {
728     struct pm_private *priv = (struct pm_private *) private_data;
729     wake_up_tasks(&priv->pm_dev->event_queue);
730 }
731
732
733 /*
734  * Set the interceptors addresses to the scsi bus and
735  * class net uvent functions
736  */
737 static void enable_hotplugging(void)
738 {
739     struct net_device *netdev;
740     struct class *netclass;
741
742     scsi_uevent_address = scsi_bus_type.uevent;
743     scsi_bus_type.uevent = &scsi_uevent_interceptor;
744
745     netdev = first_net_device(&init_net);
746     netclass = netdev->dev.class;
747     net_uevent_address = netclass->dev_uevent;
748     netclass->dev_uevent = &net_uevent_interceptor;
749 }
750
751
752 /*
753  * Set the original addresses to the scsi bus and net
754  * class uevent functions
755  */
756 static void disable_hotplugging(void)
757 {
758     struct net_device *netdev = first_net_device(&init_net);
759     struct class *netclass;
760
761     scsi_bus_type.uevent = scsi_uevent_address;
762     netdev = first_net_device(&init_net);
763     netclass = netdev->dev.class;
```

```
764     netclass->dev_uevent = net_uevent_address;
765 }
766 
767 
768 /*
769  * Adds a device that has been hotplugged (but not reconnected)
770  */
771 static void plug_device(struct device *dev)
772 {
773     int i;
774     int err;
775     int free_slot = 0;
776 
777     if (!scsi_is_sdev_device(dev) || scsi_device_reconnected(dev) == 0)
        {
778         for (i = 0; i < MAX_DEVICES && !free_slot; ++i) {
779             if (pm_devices[i].minor == FREE_SLOT) {
780                 struct pm_device *pm_dev = &pm_devices[i];
781                 err = create_device(pm_dev, dev, i);
782                 if (err < 0) {
783                     printk(KERN_WARNING "Cryogenic: Device could not be
784                            added [create_device() failed].\n");
785                     pm_dev->minor = FREE_SLOT;
786                 }
787                 else {
788                     const char *name;
789                     if (scsi_is_sdev_device(dev))
790                         name = pm_dev->serial_number;
791                     else
792                         name = pm_dev->name;
793                     printk(KERN_INFO "Cryogenic: New device %s was
794                            added.\n", name);
795                 }
796                 free_slot = 1;
797             }
798         }
799         if (!free_slot)
800             printk(KERN_INFO "Cryogenic: Device could not be added
801                    [No available slot].\n");
802     }
803 }
804 
805 
806 /*
807  * Removes a device that has been unplugged
808  */
809 static void unplug_device(struct device *dev)
810 {
811     int i;
812     int device_found = 0;
813 
814     for (i = 0; i < MAX_DEVICES && !device_found; ++i) {
```

```
815             if ((pm_devices[i].minor != FREE_SLOT) &&
816                  !strcmp(pm_devices[i].name, dev_name(dev))) {
817                 pm_devices[i].unplugged = 1;
818                 if (remove_device(&pm_devices[i])) {
819                     const char *name;
820                     if (scsi_is_sdev_device(dev))
821                         name = pm_devices[i].serial_number;
822                     else
823                         name = pm_devices[i].name;
824                     printk(KERN_INFO "Cryogenic: Device %s was
825                            removed.\n", name);
826                     pm_devices[i].minor = FREE_SLOT;
827                     pm_devices[i].unplugged = FREE_SLOT;
828                 }
829                 else
830                     printk(KERN_INFO "Cryogenic: Device %s was unplugged
831                            but not removed because it is being used.\n",
832                            pm_devices[i].serial_number);
833                 device_found = 1;
834             }
835         }
836     }
837
838
839     /*
840      * Checks if a scsi device has been reconnected. If so, resets
841      * the necessary fields and returns 1. Otherwise, returns 0.
842      */
843     static int scsi_device_reconnected(struct device *dev)
844     {
845         int i;
846         char tmp_serial[MAX_SERIAL_NUMBER_SIZE];
847
848         struct scsi_device *sdev = to_scsi_device(dev);
849         set_scsi_serial_number(sdev, tmp_serial);
850
851         for (i = 0; i < MAX_DEVICES; ++i) {
852             if (pm_devices[i].unplugged == 1 && !strcmp(tmp_serial,
                pm_devices[i].serial_number)) {
853                 struct scsi_device *sdev = to_scsi_device(dev);
854                 pm_devices[i].unplugged = 0;
855                 /* The following fields may change after the reconnection.
856                  * Thus, we must reset them */
857                 pm_devices[i].name = dev_name(dev);
858                 pm_devices[i].dev = dev;
859                 pm_devices[i].scsi_request_fn_address =
860                     sdev->request_queue->request_fn;
861                 sdev->request_queue->request_fn = &request_fn_interceptor;
862                 printk(KERN_INFO "Cryogenic: Device %s was reconnected.\n",
863                        pm_devices[i].serial_number);
864                 return 1;
865             }
```

```
866         }
867         return 0;
868     }
869
870
871     module_init(pm_init);
872     module_exit(pm_exit);
```

B. Test programs

In this chapter we include the full version of the C programs that were modified in Section 3.3 in order to use Cryogenic. We also present other C programs and shellscripts used to perform the experimentation of this work.

B.1. UDP client

B.1.1. client-rand.c

```c
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include <time.h>

#define BUFLEN 512
#define PORT 666
#define SRV_IP "131.159.74.67"

int main(int argc, char *argv[])
{
    struct sockaddr_in sock;
    char buf[BUFLEN];
    int sock_fd;
    int i;

    sock_fd = socket(PF_INET, SOCK_DGRAM, 0);
    if (sock_fd < 0) {
        perror("socket() failed\n");
        exit(1);
    }

    memset((char *) &sock, 0, sizeof(sock));
    sock.sin_family = AF_INET;
    sock.sin_port = htons(PORT);
    if (inet_aton(SRV_IP, &sock.sin_addr) == 0) {
        fprintf(stderr, "inet_aton() failed\n");
        exit(1);
```

```
35      }
36
37      unsigned long period;
38      struct timeval exec_t;
39      struct tm *t;
40
41
42      period = atoi(argv[1]);
43
44      i = 1;
45      while(1) {
46
47          sprintf(buf, "%02d\0\n", i);
48
49          struct sockaddr *saddr = (struct sockaddr *) &sock;
50          if (sendto(sock_fd, buf, BUFLEN, 0, saddr, sizeof(sock)) < 0) {
51              perror("sendto() failed\n");
52              exit(1);
53          }
54          gettimeofday(&exec_t, NULL);
55          t = localtime(&exec_t.tv_sec);
56          printf("Sent %02d [%02d:%02d:%02d.%03d]\n", i, t->tm_hour,
57                 t->tm_min, t->tm_sec, (int) exec_t.tv_usec/1000);
58          fflush(stdout);
59          ++i;
60
61          double r = drand48();
62          useconds_t d = (useconds_t) (r * period * 1000.0 * 2);
63          usleep(d);
64      }
65
66      close(sock_fd);
67
68      return 0;
69  }
```

B.1.2. client-cryo.c

```
1   #include <arpa/inet.h>
2   #include <netinet/in.h>
3   #include <stdio.h>
4   #include <sys/types.h>
5   #include <sys/socket.h>
6   #include <unistd.h>
7   #include <stdlib.h>
8   #include <string.h>
9   #include <time.h>
10
11  #define BUFLEN 512
```

```
12 #define PORT 666
13 #define SRV_IP "131.159.74.67"
14
15 #include <sys/stat.h>
16 #include <fcntl.h>
17
18 #include <sys/ioctl.h>
19 #include <sys/select.h>
20 #include <sys/time.h>
21
22 #define PM_MAGIC 'k'
23 #define PM_SET_DELAY_AND_TIMEOUT _IOW(PM_MAGIC, 1, struct pm_times)
24
25 struct pm_times {
26     unsigned long delay_msecs;
27     unsigned long timeout_msecs;
28 };
29
30 int main(int argc, char *argv[])
31 {
32     struct sockaddr_in sock;
33     char buf[BUFLEN];
34     int sock_fd;
35     int i;
36
37     sock_fd = socket(PF_INET, SOCK_DGRAM, 0);
38     if (sock_fd < 0) {
39         perror("socket() failed\n");
40         exit(1);
41     }
42
43     memset((char *) &sock, 0, sizeof(sock));
44     sock.sin_family = AF_INET;
45     sock.sin_port = htons(PORT);
46     if (inet_aton(SRV_IP, &sock.sin_addr) == 0) {
47         fprintf(stderr, "inet_aton() failed\n");
48         exit(1);
49     }
50
51     struct timeval exec_t;
52     struct tm *t;
53
54     int fd = open("/dev/cryogenic/eth0", O_RDWR);
55     if (fd < 0) {
56         perror("open() failed");
57         exit(1);
58     }
59
60     struct pm_times times;
61     times.delay_msecs = 4000;
62     times.timeout_msecs = 6000;
63
```

```
64
65      i = 1;
66      while(1) {
67
68          int r;
69          r = ioctl(fd, PM_SET_DELAY_AND_TIMEOUT, &times);
70          if (r < 0) {
71              perror("ioctl() failed");
72              exit(1);
73          }
74
75          fd_set wr;
76          FD_ZERO(&wr);
77          FD_SET(fd, &wr);
78
79          r = select(fd+1, NULL, &wr, NULL, NULL);
80          if (r < 0) {
81              perror("select() failed");
82              exit(1);
83          }
84
85          if (FD_ISSET(fd, &wr)) {
86              sprintf(buf, "%02d\0\n", i);
87
88              struct sockaddr *saddr = (struct sockaddr *) &sock;
89              if (sendto(sock_fd, buf, BUFLEN, 0, saddr, sizeof(sock)) <
                    0) {
90                  perror("sendto() failed\n");
91                  exit(1);
92              }
93              gettimeofday(&exec_t, NULL);
94              t = localtime(&exec_t.tv_sec);
95              printf("Sent %02d [%02d:%02d:%02d.%03d]\n", i, t->tm_hour, t
                    ->tm_min, t->tm_sec, (int) exec_t.tv_usec/1000);
96              ++i;
97          }
98      }
99
100     close(fd);
101     close(sock_fd);
102
103     return 0;
104 }
```

B.2. Filesystem synchronization

B.2.1. sync-cryo.c

```
1  #include <arpa/inet.h>
```

```
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <unistd.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <time.h>
10
11 #define BUFLEN 512
12
13 #include <sys/stat.h>
14 #include <fcntl.h>
15
16 #include <sys/ioctl.h>
17 #include <sys/select.h>
18 #include <sys/time.h>
19
20 #define PM_MAGIC 'k'
21 #define PM_SET_DELAY_AND_TIMEOUT _IOW(PM_MAGIC, 1, struct pm_times)
22
23 struct pm_times {
24     unsigned long delay_msecs;
25     unsigned long timeout_msecs;
26 };
27
28 int main(int argc, char *argv[])
29 {
30     char buf[BUFLEN];
31     int i;
32
33     int fd_file = open(argv[1], O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
34     if (fd_file < 0) {
35         perror("open() failed");
36         exit(1);
37     }
38
39     struct timeval exec_t;
40     struct tm *t;
41
42     int fd = open("/dev/cryogenic/WD-WCAU46069319", O_RDWR);
43     if (fd < 0) {
44         perror("open() failed");
45         exit(1);
46     }
47
48     unsigned long tolerance = 2000;
49     struct pm_times times;
50     times.delay_msecs = 5000 - tolerance/2.0;
51     times.timeout_msecs = 5000 + tolerance/2.0;
52
53     struct timeval start_t;
```

```
54      gettimeofday(&start_t, NULL);
55
56      i = 1;
57      while(1) {
58
59          int r;
60          r = ioctl(fd, PM_SET_DELAY_AND_TIMEOUT, &times);
61          if (r < 0) {
62              perror("ioctl() failed");
63              exit(1);
64          }
65
66          fd_set wr;
67          FD_ZERO(&wr);
68          FD_SET(fd, &wr);
69
70          r = select(fd+1, NULL, &wr, NULL, NULL);
71          if (r < 0) {
72              perror("select() failed");
73              exit(1);
74          }
75
76          if (FD_ISSET(fd, &wr)) {
77
78              int b = sprintf(buf, "%d\n", i);
79
80              if (write(fd_file, buf, b) < 0) {
81                  perror("write() failed\n");
82                  exit(1);
83              }
84              sync();
85              gettimeofday(&exec_t, NULL);
86              t = localtime(&exec_t.tv_sec);
87              printf("Written %02d [%02d:%02d:%02d.%03d]\n", i, t->tm_hour
                    ,
88                      t->tm_min, t->tm_sec, (int) exec_t.tv_usec/1000);
89              ++i;
90
91              unsigned long start_t_msecs = (start_t.tv_sec*1000) + (
                    start_t.tv_usec/1000);
92              unsigned long exec_t_msecs = (exec_t.tv_sec*1000) + (exec_t.
                    tv_usec/1000);
93              times.delay_msecs = start_t_msecs + 5000*i - exec_t_msecs -
                    tolerance/2.0;
94              times.timeout_msecs = times.delay_msecs + tolerance;
95              printf(" - New delay: %lu\n", times.delay_msecs);
96              printf(" - New timeout: %lu\n", times.timeout_msecs);
97          }
98
99      }
100
101     close(fd);
```

```
102     close(fd_file);
103
104     return 0;
105 }
```

B.3. Shellscripts

The following sections present the shellscripts used during our experimentation. Note that the periods that appear in Sections B.3.2, B.3.3, B.3.4 and B.3.5 correspond to Experiments 1 and 2. The periods used in Experiments 3 and 4 are displayed in Table 4.6. The periods for Experiment 5 are illustrated in Table 4.9.

B.3.1. baseline.sh

```
#!/bin/sh
echo "Starting baseline..."
../pins/set-pin # setting R-Pi pin
sleep 60
../pins/unset-pin # unsetting R-Pi pin
echo "Test completed"
```

B.3.2. test1.sh

```
#!/bin/sh
echo "Starting test WITH cryogenic using interface $1..."
../pins/set-pin # setting R-Pi pin
./client 5003 > client-1.log1 & # every ~5s,
pid1=$!
./client 3533 > client-2.log1 & # every ~3.5s
pid2=$!
# Note use of prime numbers to make collisions unlikely...
./client-cryo $1 3001 1511 > client-cryo-3.log1 & # every 3s, 50% tolerance
pid3=$!
./client-cryo $1 2503 1249 > client-cryo-4.log1 & # every 2.5s, 50% tolerance
pid4=$!
sleep 60
kill -9 $pid1
kill -9 $pid2
kill -9 $pid3
kill -9 $pid4
../pins/unset-pin # unsetting R-Pi pin
echo "Test completed"
```

B.3.3. test1r.sh

```
#!/bin/sh
echo "Starting test WITH cryogenic and randomization using interface $1..."
../pins/set-pin # setting R-Pi pin
./client-rand 5003 > client-1.log1 & # every ~5s,
```

```
pid1=$!
./client 3533 > client-2.log1 & # every ~3.5s
pid2=$!
# Note use of prime numbers to make collisions unlikely...
./client-cryo $1 3001 1511 > client-cryo-3.log1 & # every 3s, 50% tolerance
pid3=$!
./client-cryo $1 2503 1249 > client-cryo-4.log1 & # every 2.5s, 50% tolerance
pid4=$!
sleep 60
kill -9 $pid1
kill -9 $pid2
kill -9 $pid3
kill -9 $pid4
../pins/unset-pin # unsetting R-Pi pin
echo "Test completed"
```

B.3.4. test2.sh

```
#!/bin/sh
echo "Starting test WITHOUT cryogenic..."
../pins/set-pin # setting R-Pi pin
./client 5003 > client-1.log2 & # every ~5s,
pid1=$!
./client 3533 > client-2.log2 & # every ~3.5s
pid2=$!
# Note use of prime numbers to make collisions unlikely...
./client 3001 > client-3.log2 & # every 3s, NO tolerance
pid3=$!
./client 2503 > client-4.log2 & # every 2.5s, NO tolerance
pid4=$!
sleep 60
kill -9 $pid1
kill -9 $pid2
kill -9 $pid3
kill -9 $pid4
../pins/unset-pin # unsetting R-Pi pin
echo "Test completed"
```

B.3.5. test2r.sh

```
#!/bin/sh
echo "Starting test WITHOUT cryogenic and randomization..."
../pins/set-pin # setting R-Pi pin
./client-rand 5003 > client-1.log2 & # every ~5s,
pid1=$!
./client 3533 > client-2.log2 & # every ~3.5s
pid2=$!
# Note use of prime numbers to make collisions unlikely...
./client 3001 > client-3.log2 & # every 3s, NO tolerance
pid3=$!
./client 2503 > client-4.log2 & # every 2.5s, NO tolerance
pid4=$!
sleep 60
kill -9 $pid1
kill -9 $pid2
kill -9 $pid3
kill -9 $pid4
```

```
18 ../pins/unset-pin # unsetting R-Pi pin
19 echo "Test completed"
```

C. Raspberry Pi GPIO pins

The C programs included in this chapter are used, respectively, to set and unset the high value of the GPIO 11 of the Raspberry Pi.

C.1. set-pin.c

```
#include <stdio.h>
#include <bcm2835.h>

#define PIN RPI_GPIO_P1_11

int main(int argc, char **argv)
{
    if (!bcm2835_init()) {
        printf("bcm2835_init() failed\n");
        return 1;
    }

    bcm2835_gpio_fsel(PIN, BCM2835_GPIO_FSEL_OUTP);

    bcm2835_gpio_write(PIN, HIGH);

    return 0;
}
```

C.2. unset-pin.c

```
#include <stdio.h>
#include <bcm2835.h>

#define PIN RPI_GPIO_P1_11

int main(int argc, char **argv)
{
    if (!bcm2835_init()) {
        printf("bcm2835_init() failed\n");
        return 1;
    }

```

```
13      bcm2835_gpio_fsel(PIN, BCM2835_GPIO_FSEL_OUTP);
14
15      bcm2835_gpio_write(PIN, LOW);
16
17      return 0;
18  }
```

Bibliography

[1] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[2] Microsoft Corporation. Windows Timer Coalescing, 2009.

[3] Microsoft Corporation. Timers, Timer Resolution and Development of Efficient Code, 2010.

[4] D. Domingo, R. Landmann, and J. Reed. Red Hat Enterprise Linux 6 Power Management Guide, 2010.

[5] P. Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7, 2011.

[6] D. Hughes, E. Cañete, W. Daniels, G. S. Ramachandran, J. Meneghello, N. Matthys, J. Maerien, S. Michiels, C. Huygens, W. Joosen, M. Wijnantsx, W. Lamottex, E. Hulsmansy, B. Lannooz, and I. Moermanz. Energy Aware Software Evolution for Wireless Sensor Networks. In *14th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2013.

[7] A. Hylick, R. Sohan, A. Rice, and B. Jones. An Analysis of Hard Drive Energy Consumption. In *IEEE International Symposium on Modelling, Analysis and Simulation of Computers and Telecommunication Systems*, 2008.

[8] Apple Inc. OS X Mavericks Core Technologies Overview, 2013.

[9] B. Jeff. Advances in big.LIITLE Technology for Power and Energy Savings, 2012.

[10] R. Kravets and P. Krishnan. Power Management Techniques for Mobile Communication. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, 1998.

[11] A. Mahesri and V. Vardhan. Power Consumption Breakdown on a Modern Laptop. In *Proceedings of the 4th International Conference on Power-Aware Computer Systems*, 2005.

[12] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, and S. Blionas. Instruction Level Energy Modeling for Pipelined Processors. In *International Workshop on Power And Timing Modeling, Optimization and Simulation*, 2003.

[13] R. Randhawa. Software Techniques for ARM big.LITTLE Systems, 2013.

[14] P. Reviriego, J. A. Hernández, D. Larrabeiti, and J. A. Maestro. Performance Evaluation of Energy Efficient Ethernet. In *IEEE Communications Letters*, 2009.

[15] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy Management in Mobile Devices with the Cinder Operating System. In *Sixth Conference on Computer Systems*, 2011.

[16] S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Apprehending Joule Thieves with Cinder. In *1st ACM Workshop on Networking, Systems and Applications for Mobile Handhelds*, 2009.

[17] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.