Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2016

Figure 1- Hosting in the cloud (Joctechnology, 2016)

Project Title:	**Framework for Device Allocation and Monitoring in a Distributed Cloud-like System**
Student:	**Andrea Fernandez Buitrago**
CID:	**00732133**
Course:	**EIE4**
Project Supervisor:	**Professor Erol Gelenbe**
Second Marker:	**Dr Jeremy Pitt**

Abstract

As our reliance on devices increases on a day-to-day basis, so does the need to manage them securely and efficiently. This project implements a kernel module which can be loaded on Linux machines, and controls a network of devices according to the sensitivity of the data flow. It prioritizes Quality of Service in order to satisfy diverse user requirements, which may vary from device to device. It focuses on using Random Neural Networks and applying load-balancing techniques in order to assign and monitor different devices, which may be connected to different nodes on the network, and which need to communicate with each other. It builds upon previous work, namely the Task Allocation Platform and the Cognitive Packet Network.

The system was assessed by evaluating performance across different hosts and device/topology configurations. It remains robust even at a high rate of requests, but it requires thorough testing at a higher scale in order to identify how it would perform under greater stress conditions.

Table of Contents

Chapter 1: Introduction

This chapter provides an overview on the objective of this project, setting the scene for the rest of the report. Time constraints have meant that the scope of the project had to be limited, and such limits will be described. Lastly, the structure of the report will be commented on.

1.1 Project Overview

Cloud computing has been gaining increasing popularity since its comeback in 2007 (Arutyunov, 2012). It allows a series of machines to be interconnected, either remotely or locally, appearing as a single entity to the accessing user. This allows flexibility when executing applications which have different characteristics and requirements, since they can be assigned to any of these machines depending on what resources are needed and are available.

This high customisability that exists in a cloud-like network requires a more complex system of control, which will need to be able to maintain the **Quality of Service** (QoS). Applications need to be assigned to the most appropriate host in order to ensure the best service, which should be both reliable and low on overheads. Communications also need to follow the best path, so that the system can adapt to node failure or temporary disconnection.

An existing system which strives to offer the best QoS in a network, where different tasks need to be allocated and executed, is the **Task Allocation Platform,** or TAP (Wang & Gelenbe, 2015). The TAP is a network of locally-connected computers that simulate a cloud environment, trying to produce the best QoS depending on task parameters.

This project expands upon this system, extending it to deal with devices rather than tasks. We shall define a task as a one-off event with a reduced lifetime; a device, on the other hand, is seen as a persistent object which may spawn many different tasks during its lifetime, making behavioural decisions based upon historic values.

A device may measure certain characteristics of the environment around it, be that software or hardware. It may act upon these measurements in order to implement a feedback loop. Devices are assumed to be extremely simple in their logic, so that it is up to the allocation system itself to manage and connect all devices effectively.

The underlying objective is that the resulting system will be applied in an **Internet of Things** (IoT) mind-set, so that all devices in a closed system may be managed in order to maximise delay, minimise loss or power, or any other objective customisable by the user. This could be applied as a way of interacting with physical objects, measuring and changing the world around them in a time-sensitive manner.

The amount of devices that are used by the average person increases every year, with the introduction of wearable devices such as watches and glasses. In the future, buildings themselves might be considered as systems, formed by many simpler devices that perform one or two tasks. It would be unrealistic to expect every system to have the same requirements, but it would be very costly to design a complete infrastructure from scratch for each of them.

The aim of this project is to provide a small-scale model of a system that could in theory be adopted by all IoT networks, regardless of their particular specifications, so that such systems can be easily implemented and managed, and that it may foster progress towards such a future.

1.2 Project Scope

This project is based around the design and implementation of a system which executes natively in Linux machines which run Ubuntu Lucid, version 10.04.4 LTS. There are two sides of execution, the device and the allocation system. Similarly, Linux has two modes of execution, which have different sets of instructions and permissions. This limits the actions that both sides of the system can take, and has greatly shaped the resulting infrastructure.

The allocation system is run as a kernel module, which runs on kernel space, having 'root' permissions and access to all low-level details of the machine. Kernel modules don't have access to many high-level abstractions. On the other hand, devices run in the user space, which greatly restricts their permissions, but have easier API (Application Performance Interface) and are easier to debug.

In order for the allocation system to instruct the devices on what actions they should take, a two-sided communication is required between the kernel and the user space, which restricts where the device may be generated. Communications between these two can only occur locally within the same machine. This limits a device to only being able to register locally, so that it must be physically connected to the machine which will manage it. Reasoning for this will be debated over the course of the report.

The system has been tested in Virtual Machines (VMs) and physical servers, both of which were connected via an intranet, so that network conditions in this environment will be artificially good. Network speed will only be limited by the speed of the machines in processing the messages. This has been used in order to measure system overheads, since any delays caused by network failures and network overheads can be dismissed.

The number of machines used in testing does not exceed five, since a number greater than this would require a management platform that could easily load and unload the kernel module, as well as generate devices. Such a platform is out of the scope of this project. Measurements of system performance are generated by the system itself, which may cause skewness in their veracity, since the system is measuring itself, instead of a third party.

The basic problem areas which will be taken in consideration read as follows:

REQUIRED	Priority
	Communication with the device
	Decentralisation
DESIRABLE	Load balancing
	Node swapping
	Security

Table 1. Design objectives

A full description of what these entail and how they were achieved can be found in Chapter 3: System Description.

Due to time constraints, it was not be possible to construct a full simulation as close to reality as possible; some sections needed to be abstracted and simplified. Hardware issues have not been taken into consideration into this project, aside from those of the network itself.

The devices which are dealt with in this project would in reality be hardware devices which are connected via USB (Universal Serial Bus) or directly built into the machine. For the purposes of generating a proof-of-concept we will assume that the device itself, as well as the interface that the device will use to communicate with our network, will not be delved upon. It will be assumed that such an interface has been developed separately, so that each device is aware of how to properly communicate with the network's nodes.

The problem has been abstracted so that each device performs only one task: it will either send measurements to the system, or it will act upon request of the system. This choice has been made for simplicity purposes, as well as the assumption that a device that could both measure and act can always be represented as two devices locally connected to the same machine, one which measures, and one which acts.

This project makes use of the neural network system already implemented in CPN, and does not strive to alter or modify it. It will be assumed throughout the report that the random neural networks used are correctly implemented as described in Chapter 2: Background discussion.

1.3 Report Structure

The first section of the report will be dedicated to describing the key background concepts behind this project. This will be followed by a section detailing what key requirements the resulting deliverable should meet, both required and desirable.

The design of the system will be discussed and analysed, explaining possible alternatives as needed. There will be mention of relevant implementation details, followed by an evaluation of how well the requirements were implemented in the deliverable. This will be backed up with graphs when relevant.

The report ends with a conclusion on points learnt during the project, as well as a user guide for any who might continue the work on the topic.

Chapter 2: Background discussion

This section introduces the basic technical concepts upon the knowledge of which this report relies on. It contains an introduction to cloud computing, followed by the basic principles of neural networks and how they have been utilised in this project. The Cognitive Packet Network, which will be introduced just after, relies on both these concepts, building a cloud-like system on top of the Linux kernel. The basic utilities that are used in kernel programming will be explained, lastly finishing on an introduction to the Internet of Things and previous work done in this field.

2.1 Cloud computing

Cloud computing is not a new concept, dating back to the 1960s. It had a resurgence during 2007 and 2008. This was caused by the increase in infrastructure needs for businesses (Arutyunov, 2012). The current use of the term refers to "*a model that provides on-demand services over the Internet [...] paid for per usage and may expand or shrink based on demand*" (Durao, et al., 2014).

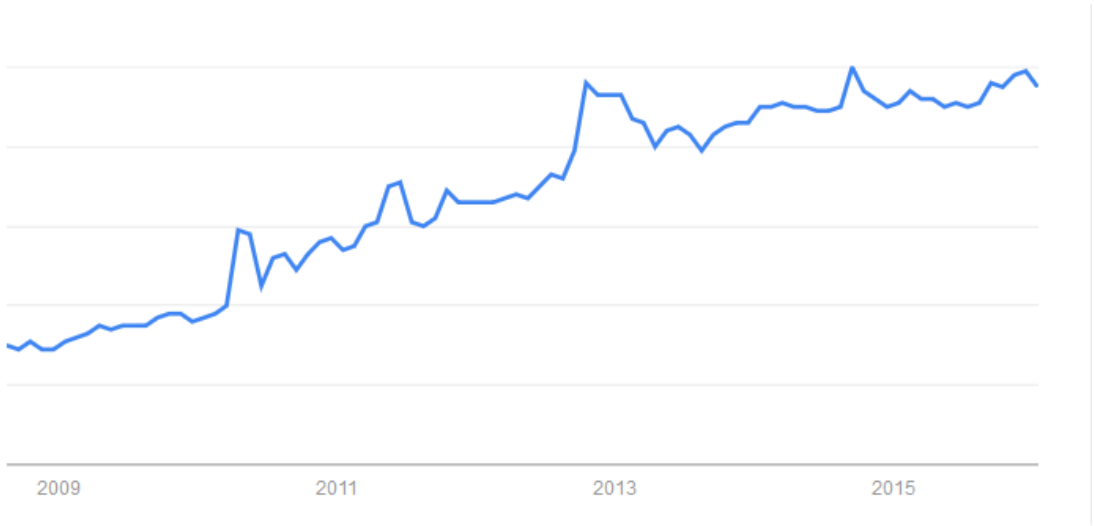


Figure 2- Google trends search for "cloud" (Google, n.d.)

Nowadays it is far too expensive to have a personal set of servers to store your applications— they require costly maintenance, such as a physical space, a team of people to maintain, air conditioning, and so on. This is often not viable, based on the fact that the machines may not be used continuously, but at random intervals of intensity during the day.

These periods of inactivity imply wasted money, so that it has now become popular to use services such as AWS (Amazon Web Services) (Amazon, n.d.). These platforms allow you to 'rent' completely configurable machines where applications can be stored and run. This is profitable to both parties; the buyers need only pay for the time used, instead of having to maintain their own system full-time, and the sellers can get rid of periods of inactivity by loaning the machines temporarily.

There is an abstraction present in the sense that the user does not know which machine they have been assigned to—the whole cloud should feel as a single entity, and machines will be allocated based on factors such as QoS or load.

There are different services that can be provided, depending on the layer of configuration that is needed. These can be classified as Sofware as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS) (Wang, et al., 2015).

2.2 Artificial and Random Neural Networks

Neural Networks (NNs) are used in Machine Learning in order to attempt to imitate the human brain; they are composed by a structure of so-called artificial neurons. They take multiple inputs in order to produce one or more outputs. In a weighted neural network, every input is weighted, so that each input has a different effect on what the output value will be.

A neural network can be composed of multiple layers, following a structure similar to the following:

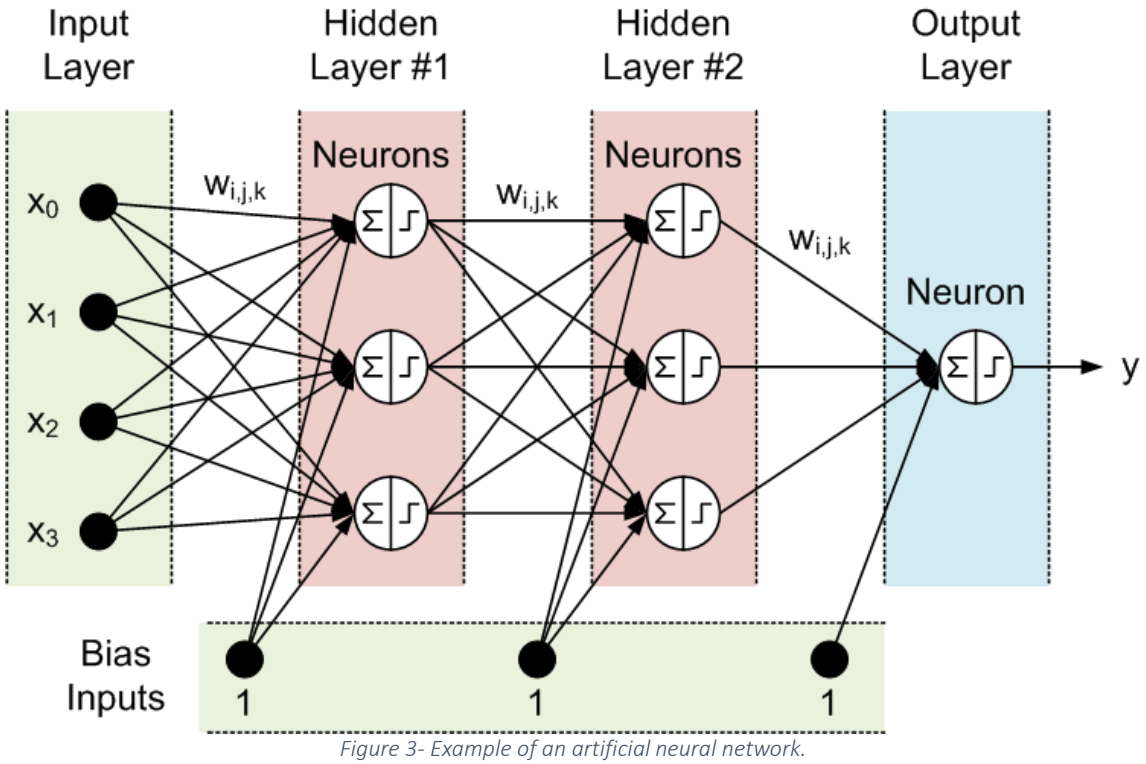


Figure 3- Example of an artificial neural network.

Neural networks are an instance of eager learning. In eager learning, all examples are provided at the beginning, and the weights and number of layers/neurons are determined at this stage of the learning process. The resulting NN, which contains the information from all examples, is used for all of the unknown problems which need to be solved. These examples are not retained, only the model resulting from them.

This model can be used to map problems for which the algorithm logic is not completely known, or is too complex to be easily represented. The system can predict the output depending on the input by learning from example. The better the examples model the problem, the better the performance of the neural network.

This process is called training the network. After K examples, the weights will be updated by a value proportional to the error of the neural network in evaluating these examples, in a reward-like manner. By repeatedly applying this process, using adequate learning rates, activation functions, training algorithms and network architectures, a reinforcement learning (RL) technique is applied. There are many different approaches to this, such as the Levenberg-Marquardt optimization-based technique (Benjapolakul & Rangsihiranrat, 2000).

Parameters such as the learning rate have to be set to an appropriate value, which depends on the target problem. If the learning rate is too gradual, learning might not occur fast enough to adapt to new conditions; if too steep, adaptation might be too fast, which makes the system subject to noise and temporary changes.

There are downsides to using neural networks, such as overfitting. Overfitting occurs when the model is too well adapted to the training examples, so that it can't correctly extrapolate it to unknown examples. This occurs when the error in the training examples is smaller than the error in unknown examples. Provisions must be made for this, for example by having three example sets: training, validation and test sets.

Neural networks have often been used in order to provide the best service possible, to model the conditions that result in the best Quality of Service (QoS) (Benjapolakul & Rangsihiranrat, 2000; Sarajedini & Chau, 1996; Dogman, et al., 2012; Girma & Lazaro, 2000; Alsamhi & Rajput, 2015).

The type of neural network that will be used in this project will be a Random Neural Network (RNN). Each neuron in the RNN starts with an initial value, which increases when it receives a positive stimuli, and decreases when it receives a negative stimuli. Each neuron has a probability of generating a positive or a negative stimuli with frequencies λ+ and λ-. Only neurons that have a positive current state are allowed to send excitatory or inhibitory signals (GELENBE, 1989). There is a probability of a stimuli entering or leaving the system, which represent the inputs and outputs of the system itself.

Using the overall firing rate r(i) and frequencies λ+ and λ-, we can represent a RNN by using a traditional artificial neural network:

$$r(i) = \sum_{j} |w_{ij}|$$

2.3 Cognitive Packet Network and the Task Allocation Platform

The Cognitive Packet Network or CPN is implemented for the Linux operating system as a kernel module. It uses server machines as routers in order to maximize the QoS of the network (GELENBE, et al., 2004). A goal is specified for the connection, towards which we train for by using RNNs.

There exist two types of packets that travel through the network, Smart Packets (SPs) and Dumb Packets (DPs), used to for discovery and to carry payload, respectively. SPs are generated when there is an incoming request, and the node receiving this request computes which outgoing link should be chosen, basing itself on the QoS information discovered up to this point, and the success or failure of previous SPs.

At each PC, there are as many neurons as connected nodes. Each node stores a different RNN for each source-destination pair, as well as for each QoS objective. The output link to forward the packet to is determined by choosing the output link that is most excited for that specific source-destination-QoS combination. We define excitement as

$$q_i = \frac{\lambda_+(i)}{\lambda_-(i) + r(i)}$$

Derived from the rates at which a neuron sends excitatory and inhibitory signals. These rates follow from the traditional input weights in NNs.

Once the optimal link is chosen for that node, the packet is forwarded to the next node, and the procedure repeated until the packet reaches its destination.

One of the issues with CPN is that it does not consider failures which occur a couple of links away from the current node, and that may influence selection of a specific link. Values are only updated when an ACK value is received, which means that if a packet is lost because of a broken connection, this change will not be updated. CPN deals with this by routing a portion of the SPs randomly, so that it may recover from this situation (Sakellari, 2011).

Based on the Cognitive Packet Network, there exists a system that applies its concept to task allocation in a cloud-like infrastructure. This system has been dubbed the Task Allocation Platform or TAP (Wang & Gelenbe, 2015). In TAP, there exists a central node, to which tasks will arrive at a certain rate. Tasks can be classified as different types, such as computation intensive or IO (input/output) intensive.

Depending on a task's category, its requirements may differ. An IO task may require good responsiveness, but have low memory requirements, while another task may require a more powerful machine, and no importance is attached to it replying in a timely fashion. Examples for these two types could be an interactive task and a data crunching task, respectively.

By using CPN's smart and dumb packets, the central node decides to which node to assign the task, as well as which path to follow. This has all already been implemented in a series of interconnected machines.

Several different task allocation algorithms were implemented: round robin, random, sensible algorithm and a RNN algorithm with reinforcement learning, between others. A comparison between the algorithms was made, so that the sensible and RNN algorithms were the ones with the best resulting performance depending on the conditions.

This so-called sensible algorithm calculates the probability of each host providing the best QoS, and then settles on the one that has the highest probability of fulfilling the QoS requirement.

2.4 Kernel Modules and Kernel-User Communication

The kernel is the central part of the Operating System (OS), so that it is the first piece of software to be loaded at system start-up. System crashes are caused by an unrecoverable fault in the kernel, such as a segmentation fault in C, which makes debugging in kernel-space a tiresome task.

It provides basic services for all other parts of the OS, such as *"memory management, process management, file management and I/O (input/output) management (i.e., accessing the peripheral devices)"* (The Linux Information Project, 2005). The interface between these services, and the rest of the OS, and user application programs is implemented via the use of 'system calls'.

The kernel runs in a protected area of memory, so that important functionalities may not be affected, for example, by poor programming from the user side of the system. Because of this, the OS is said to be separated into two spaces, the kernel space, and the user space, so that they may not interfere with each other.

There are several kernel types, such as *monolithic kernels*, *microkernels*, *hybrid kernels* and *exokernels* (The Linux Information Project, 2005). The Linux kernel, being a Unix-like OS, is a monolithic kernel. These type of kernels contain all core functions as well as any device drivers that are needed to interact with any external hardware.

Device drivers are an example of kernel modules. Kernel modules are pieces of software that may be attached onto the base kernel and extend its functionality as required. They can be loaded and unloaded without rebooting the system or affecting execution of other modules or services already running (Salzman, et al., 2009).

Since kernel modules run in kernel space, and, as has been mentioned, this space is cordoned off from the user space, ways have to be implemented in order to efficiently pass messages from one to the other.

There are several ways that this can be done, such as with system calls, which are equivalent to function calls. IOCTL (Input Output Control) calls are popular in this respect. Net devices, which appear as a '/dev' file, are also used as interfaces, and can be used to communicate between different machines or even locally from different OS spaces.

There is also an implementation via virtual file systems, such as 'procfs' and 'sysfs'. The kernel module can write data to these file systems, so that these files are generated on the fly as they are accessed. They can be written to from user space.

The next alternative is to use sockets in order to connect to the kernel. These sockets operate the same way as the usual TCP/IP sockets between clients and servers, but because of the way they have been implemented they are only usable as a one-way communication from the user to the kernel.

If bidirectional communication between the user and the kernel is needed, one of the best alternatives to use is netlink sockets. It is a "*datagram-oriented messaging system*" that allows all popular protocols, as well as user-defined protocols. They are architecturally portable, use event-based signalling, are easily extensible and allow for large data transfers (Neira-Ayuso, et al., 2010).

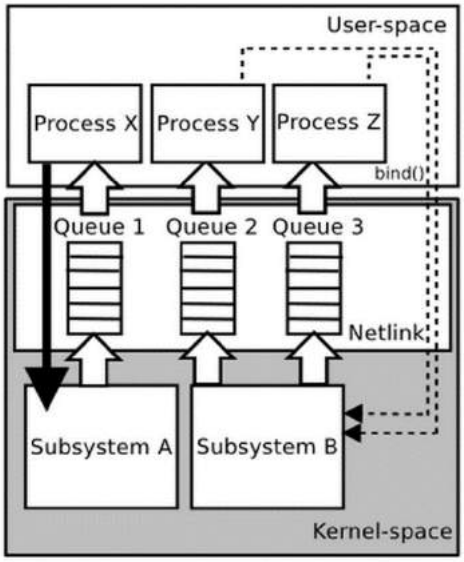


Figure 4- Overview of netlink socket usage (Gajdos, 2014)

Netlink sockets act just like sockets in that they need to bind to a port, to which the kernel module will be listening. Operations in the user-space are synchronous, so that they perform a blocking wait while the kernel replies. On the other hand, messages in the kernel space are sent asynchronously, so that there are no blocking waits. This would cause many issues for a kernel module, since each process executes and communicates asynchronously. Blocking operations in the kernel may cause kernel panic, and the machine to crash.

2.5 The Internet of Things

The Internet of Things (IoT) is a field that can be used to refer to "*pervasive presence of a network of things and objects that are able to interact, and cooperate with each other*" (Usman, et al., 2014). The objective is that the Internet doesn't only remain in the software realm, but that it can also affect physical objects, such as cameras, house-hold items, wearable technology, etc.

These devices may all have wildly different characteristics, so that it proves necessary to take into account a way to make these devices be compatible with each other, with dedicated interfaces. An example of this would be the range of each device, such as 10m for Radio Frequency Identification (ID) and 100m for Wireless Sensor Networks (WSN).

Part of the difficulty in developing an IoT system is the variety of standards that are currently available in the market. It is not viable to have to develop an interface per IoT service per device. The ideal situation would be to have one international standard, so that any device that used this would be able to join to any IoT application easily and effortlessly. The most emergent standards are currently EPCglobal, ETSI and IETF.

Chapter 3: System Description

The aim of this project is to implement a system to which devices can connect and use to *efficiently* communicate with each other. In this section, the assumptions that were made in order to simplify the development of such a system will be explained. This will be followed by brief descriptions of all the requirements that should be met by the system, and the section will finish with some representative use cases.

3.1 Assumptions

It will be assumed that CPN will be run on suitable machines, which contain all necessary software and correct versioning. A full description of OS and software requirements can be found in **Chapter 7: User Guide**. The machines used must also be connected to a network, and possess at least one working network device that can be used for such a purpose.

It is assumed that there will only be two possible types of devices, measuring and acting devices. A measuring device will generate a value every X seconds, and this value must be convertible to an integer. For the purpose of simplicity, all measuring devices generate floating point numbers, which are later converted to fixed point integers for sending.

Lastly, an 'action' that is executable by an acting device is assumed to be one or more commands that can be run by the Linux shell (`/bin/sh`) in one of the following ways:

```
/bin/sh -c example_filename.ext
/bin/sh -c 'command1 arg1 arg2; command2 arg1 arg2; ...'
```

3.2 Requirements

During the initial planning stage, a set of functionalities that the system should support was decided upon. They can be categorised into necessary characteristics that the system must have in order to provide a basic service, and desirable characteristics that would improve upon this service. They are as follows:

3.2.1 Necessary functionalities

3.2.1.1 Communication with the device

There needs to be a two-way communication between the CPN node and the device, so that a device might:

Register or deregister with a node.
Send measurements.
Receive or send message acknowledgements.
Receive action requests.

Table 2- Possible device-node communications

This communication should ideally be able to occur both locally and remotely, so that a device may not need to be physically connected to the node who manages it.

3.2.1.2 Decentralisation

A device should be able to register at any one node in the network, as well as spawn tasks from this node. This is so that all devices aren't registered to the same node, which could be the source of congestion issues. If this master node became temporarily isolated or unavailable, the whole system wouldn't be able to continue.

3.2.1.3 Measurement monitoring and storing

Measurements generated by a device should be stored at the node that manages it, so that they may be referred to at some point. This type of data might want to be stored indefinitely or otherwise by the user, in order to provide an overview of device behaviour over time, as well as establishing patterns.

Since devices are persistent, we need to deal with the issue of running out of storage space for the data. The system would therefore need a way to deal with this issue, such as compressing the data or saving to disk.

3.2.1.4 Measurement-dependent actions

The node managing a device should make the decision on whether to act upon these measurements or not, based on the characteristics of the task that the device performs. This takes complexity out of the measuring device, so that it can dedicate itself to reliably taking measurements instead of adding checks that may be complex and time-consuming. All algorithm complexity is shifted to the node network itself.

3.2.1.5 Dealing with packet loss

Loss can occur when packets are sent across the network. This can occur in the following two cases: when a device communicates with a node, and when nodes communicate with each other. This needs to be dealt with so that a device's functionality isn't impaired in a non-ideal network. In critical situations, the loss of a packet may result in an essential action not being taking; this may have serious repercussions.

3.2.2 Desirable functionalities

3.2.2.1 Processing by priority

Every action that must be performed by the system must be assigned a priority, such that higher priority jobs are scheduled to execute before lower priority jobs. Lower priority jobs must not suffer from starvation, so that their priority must increase over time.

The objective is to improve responsiveness, so that jobs that will be relevant for a smaller period of time are given precedence over jobs that can wait, such as diagnostic tasks.

3.2.2.2 Load balancing between nodes

Having multiple devices managed by the same resource may affect the service being provided. A busy resource may process data slower and have slow responsiveness. There is a need to take into

account the number of currently active devices, as well as the size of the job queue to process, so that traffic may be routed to the next best option as a node becomes busier.

3.2.2.3 Swapping devices between nodes

The network's characteristics may change after a device has been assigned to a node. If no action is taken, the QoS may degrade over time. There needs to be a way for the device to be reassigned to a different node after its initial assignation.

3.2.2.4 Security

As is mentioned in section 3.2.1.3 Measurement monitoring and storing, the system's nodes will be storing user data. There is the issue of securely maintaining this data so that it cannot be used against the user.

Another concern that may arise is the *spoofing* of a device. Spoofing occurs when a malicious entity pretends to be part of the system, in this case, a device. There should be a way to determine whether the device can be trusted.

3.3 Example use case

In order to visualise the concept behind this project, a simple example follows. Let us consider a hospital that uses a monitoring system for tracking a patient's heart rate. There is a heart monitor attached to each patient, which takes measurements continually. If the rate deviates from a certain threshold, any available doctor should be notified as soon as possible.

We would characterise all devices in the system into the archetypal classes we have defined:

Type	Measuring	Acting
Device name	Heart monitor	Doctor's pager
QoS objective	Minimise delay	Minimise delay
Device priority	High	High

Table 3- Characterising the devices

Each monitor would be continuously sending measurements to its assigned node in the network. This node would be in charge of determining when an action must be taken. Every action inherits the QoS objective of the device that generated the measurements.

An example of what an action grid would look like is:

Action ID	Action priority	Range start	Range end	Action description
0	High	0	40	Heart rate is critically low or high. Emergency notify a doctor.
		120	300	
1	Medium	40	60	Heart rate is slightly above or below the average. Log as potentially problematic.
		100	120	
2	Low	60	100	Patient's pulse is in a normal resting heart rate. Log as in no danger.

Table 4- Possible actions

The node would continually log and store measurements, and send an emergency request to the nearest doctor whenever the heart rate would enter a critical range. In this way, we have an interconnected system of physical devices that communicate via a network, and that leave all of the logistics to the system itself.

Since a heart monitor and a pager would only have very simple logic implemented in them, the system itself has the task of making smart decisions, depending on the context's requirements.

The hospital then keeps growing and decides to adopt microemulsions, a microchip device that performs blood tests, to their system. Blood tests do not require a fast response from the doctors, but do not generate values often in order to not disturb the patients. Knowing these constrains, the hospital decides to prioritise packet loss over delay for these devices.

If a CPN-like system had not been used, and they had developed their own infrastructure, they would now need to refactor their whole system in order to support two different QoS objectives. This adds complexity, and might imply downtime in the service. This could be very costly and difficult to manage, even more so when the system provided is essential, as in this case.

Using CPN, the hospital would only need to design an interface for their new device, setting their QoS objective to be packet loss. The system would then automatically minimise both delay and packet loss, depending on the device used. The system could also support customisable QoS objective, which can be easily added by implementing a series of functions that would calculate the 'reward' used to train the neural network. This will be further described in 4.2 Kernel module design.

Chapter 4: Analysis and design

This section describes the infrastructure of the system at a high and medium level. It focuses on the design choices that were made, not in the order that they were implemented, but attempting to provide a coherent view of the system. Since the kernel module and the device are two independent components, they will explained in separate sections.

4.1 Device design

4.1.1 General design characteristics

The first stage in the development process was to identify which characteristics can be found in common between all inter-connected systems of devices. The objective in mind was that the resulting infrastructure was general enough that it could ideally be adapted to suit a high percentage of possible IoT systems.

Different examples of systems were analysed for any features which all of them might share. It was found that all analysed devices could be expressed in terms of two different behaviours: devices that generate measurements, and devices that act upon these measurements. In the possible case that a device may combine both, it can always be expressed as a combination of two different devices, so that this abstraction still holds.

	Device name		Device type
Smart house	Thermostat	Motion detector	Measuring
	Heaters	Doors	Acting
Hospital	Heart rate monitor	Blood tests	Measuring
	Pagers		Acting
Wearable devices	Stress level sensor		Measuring
	Call emergency contacts		Acting
Security system	Security camera		Measuring
	Security doors		Acting

Table 5- Examples of analysed IoT systems

As was mentioned in 1.2 Project Scope, the hardware side of the device was out of this project's scope. This was decided upon the basis that hardware properties would greatly differ depending on the particular IoT system. Since the objective is not to focus on one particular system, but to be relevant to as many as possible, adding hardware support is counter-productive to designing an abstract and extensible system.

This lets the project focus on the interface between the device and the kernel module. Both measuring and acting have many points in common, and they only differ on whether they are **transmitters** (sending to kernel) or **receivers** (receiving from kernel). Doing separate implementations for both was discarded early in the development process, since most of their behaviour is common for both. They are currently implemented in the same file, so that choosing between both is effected via command line arguments. Further details on this are found in Chapter 7: User Guide.

Diagrams which show the full behaviour of a device can be found in Appendix C – Diagrams.

4.1.2 Messaging the kernel

Measuring devices need to generate measurements, and communicate these to the kernel module that implements CPN. An acting device receives requests from the kernel module, and acts upon them. From this, we can generalise that a device needs to be able to both send and receive messages from the kernel.

This section was one of the last to be implemented, because of the difficulty of finding documentation on implementing Netlink sockets, the Linux kernel interface that provides the easiest way of using bidirectional communication between kernel and user. All the other options described in Chapter 2: Background discussion are either for unidirectional use, or require use of intermediary files. The original Task Allocation Platform combined applications in many different platforms: using C files, AWK files, and bash executable files. This makes managing the system quite a task, and makes the system less extensible.

The way of communication that had already been implemented consisted in the user sending data to the kernel modules via system calls or sockets, and the kernel replying by writing information to files that could be read by the user. This adds the burden of serialising the data so that it can be written to file, and parsing it back when being read. Most of the development process was spent researching how to remove this intermediary file from the process, so that the data could be directly sent from kernel to user.

Many types of messages need to be sent, which all need to be distinguished from each other. Kernel and user work asynchronously, so that it's not possible to predict what state the other will be in. Without knowing the other's state, it can't be known what type of message will be sent. In order to solve this issue, a message's payload was divided into two fields: a header, which is always the same size, and the actual payload, the size of which can be determined from the header contents.

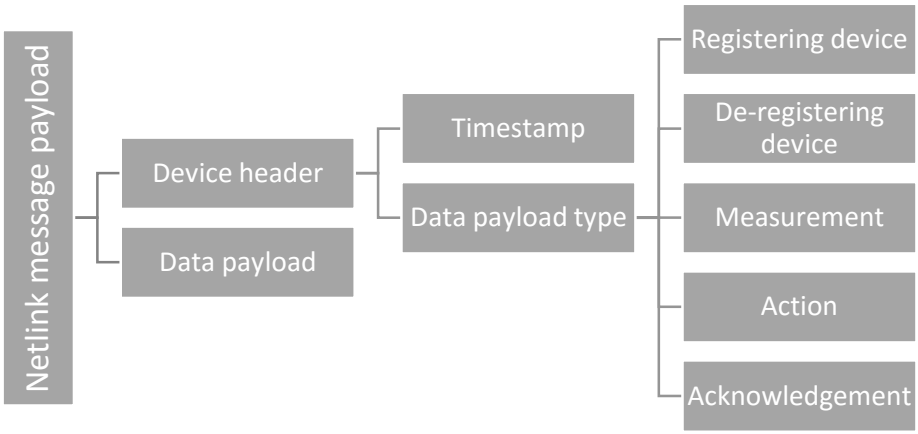


Figure 5- Netlink message structure

The main disadvantage in using netlink sockets is that all communication must occur locally, unlike with usual sockets. By locally, it is meant that a device must be physically connected to the node in the network with which it registers. A device is not able to be shuffled around between nodes as the load

and network characteristics change, since it can only communicate with the node it has been spawned in.

This causes a deviation in what the optimal cloud-like system should behave like. Ideally, devices should be mobile and independent from the cloud, swapping between the nodes of the network that manage them. Using netlink allows bidirectional communication, but disallows the possibility of adding node swapping functionality into the system.

Some options were considered as a work-around in order to simulate load swapping. The first of which was to assume that every node in the network could perform the work of all possible devices, so that they all had the same device source code available. Nodes would switch their devices on or off, depending on system load. This would accomplish swapping without actually having to communicate from user to kernel between hosts.

This was discarded since it is unrealistic to expect that all nodes in the network will be able to perform all tasks. In the case of expensive devices, replicating this device all along the network wouldn't be viable.

4.1.3 Registering a device

The action of registering a device with the network should in principle only occur once, unless communications between the module and the device are broken. The bulk of the communications between device and kernel should be measurements and action requests. If we estimate that a device can be online for days generating measurements, it will generate/receive thousands of messages during its lifetime, and only one or two of those will be a register request.

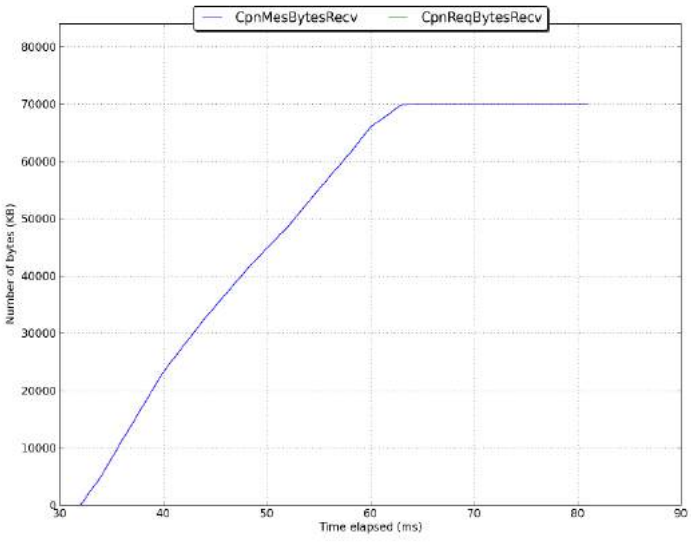


Figure 6- Proportion between measuring and registering requests. Only one registering request occurred, spanning 4KB.

This results in an estimated ~0.1% of the load to be caused by registering, and ~99.9% by measuring/acting. In network systems, the amount of time taken to send a message from one node to another increases proportionally with the size of the message. Since the bulk of communications is taken up by measuring/acting, we need to minimise the size of these messages in order to

minimise the time taken to transmit them. This becomes particularly true in networks with high latency.

After taking this into account, the message which registers a device should contain the maximum amount of device information, so that the measurement and acting requests only contain the minimum of information required to be able to act.

Let us analyse what information needs to be known by the node in order to be able to make decisions on behalf of the device, without needing to contact it further. The kernel should be able to distinguish between two devices; a unique id is needed. Taking into account that netlink sockets use a process' ID in order for the kernel to distinguish between listening processes, and that there can be no two processes with the same process id, the system uses PIDs (Process IDs) as a device's unique id. This has its advantages and disadvantages. The advantage is uniqueness, while the drawback is that the ID of a device cannot be predicted before it has registered with the system.

This results in that an action cannot be sent to a particular device, since we cannot predict what this device's ID will be. The current implementation of the system generates actions which are executable by any acting device connected to the system. As a future improvement, a way of polling the network could be implemented, so that it would return all devices which meet certain characteristics.

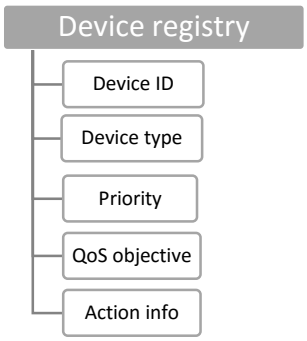


Figure 7- Device requirement structure

The kernel should also be able to separate between the different types of devices. This ensures that it only sends acting requests to the relevant acting devices. From 3.2 Requirements, other information that is needed is a device's priority and QoS objective. While a device's QoS will apply to all of its actions, actions are customisable and have a priority assigned to each of them. More details follow in the next section.

A design decision was made in that the device reads the structures for a device from the same file as the kernel. The rationale behind this was that future modifications would ripple through the design, affecting all code without having to modify multiple files. This entails, on the other hand, releasing sensitive code to the user. Extra security measures should be taken so that knowledge of the source files cannot be used to exploit the system.

Device registering now contains all relevant information for a device, so that any further messages only need to contain one piece of data (plus headers); only the measurement to be sent, or the action to be executed. Any other device information is stored in the network's and the devices' memory.

The only downside to this approach is that a device's characteristics cannot change once it has registered with a node. A device would need to de-register and register once again with its updated characteristics, which interrupts execution. This is unlikely to happen with simple devices we are dealing with in this project, since their functionalities are extremely limited.

Provisions are made for the case in which registering fails. A failure in registering can be caused by the following two cases: the kernel did not reply before a timeout; or the kernel replied with a failure in registering. More details on the latter in 4.2 Kernel module design. In this case, the device will continuously loop, attempting to register with the kernel, until it either succeeds or it device life expires.

4.1.4 Generating action grids

The action grid is part of a device's requirements, which are sent over when a device registers. The minimum an action can be characterised by is by a unique id, the action to perform, and when to perform the action.

A measuring device does not, in fact, need to know what the action itself is. The same can be said for the kernel module. Only the action ID, and the range of values for which to act are required. Extra fields that were added in order to increase responsiveness are priority and QoS objective. Because of this, the acting device is the only one that has knowledge of what the action will accomplish, since it is the only one that executes it.

The acting information field is empty for acting devices, since they do not originate requests, but only receive them.

The system supports multiple possible actions per device. All actions are included in the data sent when registering a device, so that the data size increases proportionately with the number of possible actions. This should not greatly affect the system.

Currently, all action information has been hardcoded. Actions greatly differ between systems, so that a device would need to be customised for each network. The system that was used as a working example was loosely based on a home's heating system. In order to show multiple action functionality, two actions which would not occur at the same time were hardcoded (their value ranges did not intersect). This was chosen for debugging ease, but could have equally have been set so that both actions were performed at certain values which intersect. There is support for an action occurring at more than one set of values.

	ID	PRIORITY	STARTING RANGE	ENDING RANGE	EXECUTES
ACTION 1	0	0	20	30	'echo 'Hello world!''
ACTION 2	1	0	30	40	'ls' (lists files in current dir)

Table 6- Action information used for experimentation

Actions were chosen so that successful execution was simple to detect, and so that they could be distinguished between both actions. This was done for the purposes of perfecting load balancing, so

that the overhead of the system could be studied judging by how it dealt with increasing the number of requests per second.

In a real environment, this would be implemented using configuration files, where existing device information could be stored and read at runtime.

4.1.5 Generating and sending measurements

This section is only applicable to measuring devices.

An initial measurement is formed by generating a random value in an established range. Additional measurements are obtained by modifying the previous value with a random deviation. This is to simulate the behaviour of a house's thermostat, for which temperatures will slowly vary.

MINIMUM VALUE	0.0
MAXIMUM VALUE	40.0
MAXIMUM DEVIATION	10%

Table 7- Possible measurement ranges

Currently, measurements consist of one value, but more complex structures could be added in the future. This would reduce performance on the kernel side, since more complicated checks would need to be effected. A simple device would most likely have no need for more than one generated value at a time.

This value can be of any type, but must be convertible to an integer. When creating an action grid and registering the device, the ranges of values must also be converted to integers. The kernel does not require to know what the real type of the value is, as long as the following properties are held:

- Each measurement must map uniquely to an integer.
- If a measurement X is bigger than another measurement Y, the integer that X maps to must be bigger than the integer that Y maps to.

This ensures that the kernel will be able to successfully check whether a measurement lies in a range of values, just by having their integer representation.

Temperatures are measured as floating point numbers, and they have been converted to integers and back to floats using the following formulas:

$$int(x) = x * \left((1 \ll FXPOINT) + 0.5\right)$$

$$x = \frac{int(x)}{1 \ll FXPOINT}$$

where FXPOINT is defined as 14.

Measurements are generated and sent every X seconds, an amount that is customisable for each execution.

TIME	0	X	2X	3X	4X
VALUE	26.340820	26.517456	28.303894	30.587585	29.369324

Table 8- Example of a list of generated values

The device waits for an acknowledgement from the kernel. The device will keep sending measurements even if no reply is received, keeping count of how many times the measurements have received no response. If this number grows above a certain threshold, which is customisable, the device considers that it has lost connection with the kernel, and it attempts to register again.

This approach was followed, taking into account that the device will only wait for a reply for a limited amount of time, before considering communication a failure. Waiting until a timeout saves the device from remaining in a blocking call permanently, but it may cause the device to miss a reply if the kernel is overloaded, and is simply being slow to respond.

In order to account for this case, we allow a certain number of measurements to go un-replied to, before we can safely assume that the kernel module has either crashed, or otherwise re-started. The device has no way of telling between both of them. If it has crashed, then nothing can be done from the device's side, but if it has re-started, the device will need to re-register. So, if a number of measurements go ignored, the device will reset to its registering phase, and keep attempting to register until it either succeeds or its device life runs out. If it succeeds, it will continue generate measurements as before.

4.1.6 Executing actions

This section is only applicable to acting devices.

In the same way a measuring device continuously sends messages to the network, an acting device continuously waits for requests. Requests are waited for until they are either received, or a timeout occurs. If the timeout is reached, the device will send a message to the node it´s assigned to, in order to confirm to the kernel that the device remains active. The device will continuously wait for requests until its life expires.

If a request has been received, the device will first confirm that the request contains the correct headers, and that it is a valid action. This is done by checking fields such as the device ID the request is meant for, the type of message, and the action ID. Additional information that is contained in the request, but not used for sanity checking, is the ID of the device that originated the request.

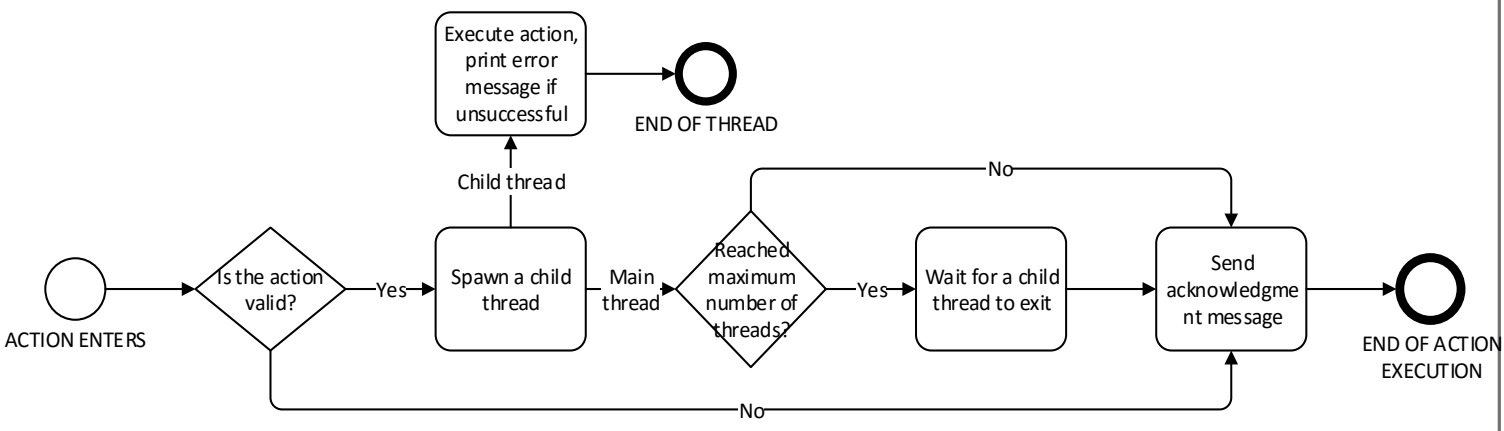


Figure 8- System diagram for action execution

Once the action has been confirmed as valid, it will start to be executed in a separate thread. This ensures that an action does not block the normal execution flow of the device. The child thread will execute while the main thread sends an acknowledgment message to the kernel, and awaits further requests.

If the number of threads grows unchecked, the machine may run out of memory. In order to deal with this, a maximum number of threads is set. If this amount is reached, the main thread of execution will wait for any one of the child threads to finish before continuing.

Actions are executed in the Linux shell. Because of its wide-spread use, most applications can be compiled to be run on the shell, regardless of the programming language it was developed in.

4.1.7 Device clean-up

After a device's life has expired, it needs to exit gracefully. The device may be at any point in execution, so this was implemented using an asynchronous interrupt. An alarm is set at the start, which will trigger clean-up after a certain amount of time. By using operating system calls, this will be called regardless of the device being in a waiting state (waiting for a message from the kernel).

While the device would automatically be removed from the network database after not communicating with it for a period of time, we want to minimise the amount of requests which are incorrectly assigned. To avoid this, the device sends a de-registering message, so that it is immediately removed from the active devices list at the kernel.

If there are any child threads currently running, a signal is sent to all of them to immediately finish execution.

4.2 Kernel module design

4.2.1 General design characteristics

The design approach taken was to first identify which sections of the kernel module would need to be modified in order to implement a device allocation system, using the Task Allocation Platform as a base. The structures themselves would obviously be the first to change - a device has different characteristics from a task. Ideally, the way of messaging would not need to be modified, in order to make it compatible with older versions of CPN. Since bidirectional communication was needed, and the communication tools already in the kernel did not support it, this could not be achieved.

What was essential to leave untouched was the protocols used to communicate between the CPN nodes in a network. Modifying this would affect backwards compatibility too much. In the end, half of the communications side remained unchanged; while the communication with the user had to be completely overhauled, any other communications were left mostly unchanged, such as messaging protocols between CPN nodes.

The routing of the messages and the conditions in which the messages are kept also needed modifying. The assumption in previous versions of CPN is that a node knows what the destination of all packets must be, and that its only task is to find the best way to them. When dealing with allocating

tasks, the destination is not known beforehand, and must be found. This will be explained in section 4.2.7 Allocating requests.

Any infrastructure needed to deal with measurements from a device would also need to be added, as well as a way of keeping track of currently active devices. The way of executing received jobs was completely redone, so that this was also managed inside the kernel module itself. This way, a 'controller' was born, which manages devices and the sending of jobs to them. This is also where load balancing was implemented.

The way that incoming jobs were dealt with was left almost unchanged, since the only changes that had to be made was the addition of priority and the ability to fetch jobs from the queue. As for all other functionalities of the system, they did not need to be modified. An example of such features is the maximising and minimising the QoS objective using Random Neural Networks. This had already been researched and successfully implemented, and the purpose of this project was not to improve upon these methods but to make use of them in a new context; it being persistent devices.

For this reason, the implementation of the RNN and the reward system will not be mentioned in detail; the full explanation of their workings can be found in 2.2 Artificial and Random Neural Networks.

4.2.2 Messaging in the kernel

For messaging a device, the same API is used as in 4.1 Device design, but mirrored to work at the kernel level. Headers are used to keep track between different types of messages. A message is ignored if the device ID contained in the header and the PID of the process sending the message do not match.

Whenever a message is received from a certain device ID, the first check is to determine whether the device is currently registered with the system. If it has not been registered, and the message isn't a register request, the message will be ignored. If the message is from a valid device, it will then be processed according to its type, and the device's life will be extended. As long as messages are received from a device, its status will be set to active.

After processing the request, an acknowledgement message will be sent to the device, with the exception of receiving an acknowledgement of an action request. This occurs when the module has sent a request to a device (the only case in which the kernel initiates communications), so that it is the device that sends an ACK; no further ACKs are sent in this case, so that we do not have an endless procession of ACKs being sent.

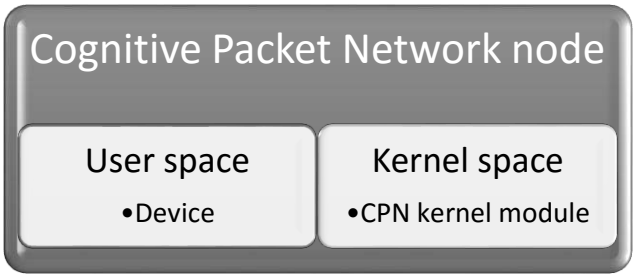


Figure 9- Overview of the Cognitive Packet Network

When messaging between CPN nodes, headers are also used in order to distinguish between different packet types. In fact, this is what inspired the use of headers for the communications between the module and its devices. Headers, aside from telling type, store the path between two nodes of the network. This can be built as the network is discovered (as is the case with smart packets), or it can be followed to deliver a payload (as with dumb packets). By also storing network characteristics as the packet is forwarded, we gain an idea of the characteristics of the network.

Messaging between user-kernel and between nodes use different protocols, so that a translation from one to the other must be performed as will be detailed in 4.2.6 Creating requests.

4.2.3 Registering and deregistering a device

When a device registering requests arrives into the system, the whole register request for the device is stored in memory. From 4.1.3 Registering a device, each request takes around 4KB of space; this will, of course, depend on the number of actions a device can take. Any machine nowadays should have enough memory to be able to deal with a large amount of devices registered at a node.

If the maximum number of devices has been reached, the request will be rejected, and a message confirming this decision will be sent to the device. The device will then repeatedly try to register with the module, until it accepts the requests or the device life expires. If the device has already been registered, there are two options to be considered. The device could not have received a confirmation on its registering earlier on, so that it is now trying again, or the module is currently under heavy load and it has received multiple register requests after a delay.

At this point in device execution, the device should not be under heavy load, so that the ACK message should definitely have been received. It is much more probable that the second option is the real case, and so the module is programmed to ignore multiple register requests from the same device, and only reply to the first of them it receives.

If the device is not already registered, and the maximum amount of devices has not been reached, we now store all relevant device information. If it is a measuring device, we create a measurement box for it, which will keep all measurements that are generated by the device, compressing them when necessary. Further information in 4.2.4 Storing device measurements. This measurement box is designed for fast access, given the fact that measurements will be received much more often than any other kind of data. We want to quickly be able to access measurement storage so that the next measurement can be processed as soon as possible.

Action ACKs from the device are received in batches as the requests are processed, and no action must be taken in response to them, while measurements are received continually at a set rate and all of them must be acted upon. This is shown in Figure 10.

A device can also be removed from the system. This can occur on two occasions: the device issued a de-register requests, or the node has not received any messages from the device in the past X seconds. The memory allocated to the device is freed, and if the device does not register again, any further messages from it will be ignored. Responsiveness measurements for the device are kept just in case the device manages to register again, and this does not pose a huge cost due to the simplicity of the data being stored. More information in 4.2.5 Storing device responsiveness.

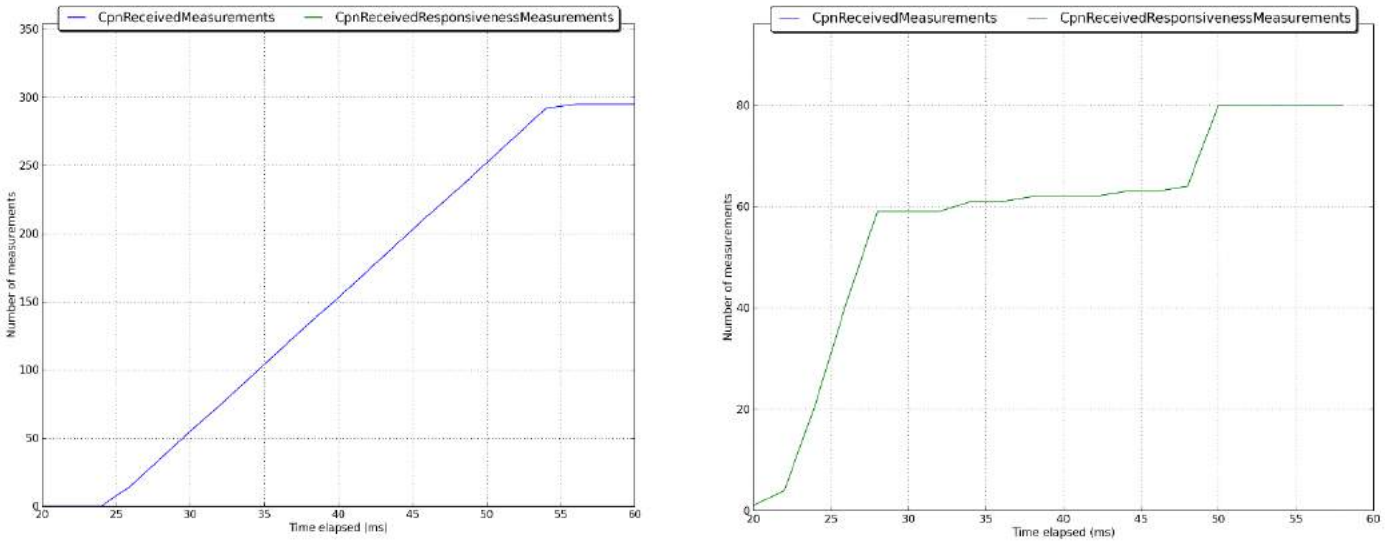


Figure 10- Comparing measurement rate to action ACK rate

4.2.4 Storing device measurements

Measurements are stored as they are received. In the current implementation, the module is not informed of a device's life or its measuring rate. As far as it is concerned, the device may generate measurements indefinitely, sending them without pause. The module needs to limit the amount of storage space it assigns per device. Knowing device life and measurement rate and allocate space according to this was considered, but this is vulnerable to rogue devices sending incorrect information.

The amount of measurement space is statically allocated when a device is registered. It will be populated with measurements as they are received, and when a defined threshold is reached, measurements will be compressed to half the original size.

Compression is carried out with a simple moving average algorithm. The mean between two consecutive measurements k and k+1 is calculated, such that by the end of the process the measurement box now contains half the measurements.

If we set the measurement box size to be 10 measurements, and follow the measurement box as it is filled up, we can see the effects of compression. The state after 9 measurements have been received is shown here, where the first number represents the timestamp for when the measurement was received, the second is the measurement expressed as a float, and the last is the measurement expressed as a fixed point value.

id=11129

226775047	40.00000	(655360)
226825049	38.33813	(628132)
226875050	40.00000	(655360)
226925052	40.00000	(655360)
226975053	40.00000	(655360)
227025055	40.00000	(655360)
227075056	40.00000	(655360)
227125058	39.88366	(653454)
227175059	39.70355	(650503)

When one more measurement is received, the size of the measurement box has now decreased by half, and all fields are now the result of averaging two of the values of the previous state. This means that as the number of measurements increases, the time resolution of these measurements decreases, so that instead of having 1 measurement every 1-2 seconds (for example), we now start having one stored measurement for every 3 seconds, then for every 5 seconds, etc.

The more information we receive, the less details we can provide about a specific point in time.

id=11129

226800048	39.16906	(641746)
226900051	40.00000	(655360)
227000054	40.00000	(655360)
227100057	39.94183	(654407)
227200060	39.85174	(652931)

Measurements expire when the CPN node stops execution. Stored measurements are not currently used for any particular purpose, aside from being able to debug the device and plot graphs of its behaviour. This topic wasn't developed further due to the number of possibilities that this could take. Some devices may want to use not just the latest value, but the average of the last X values. An infrastructure may want to be kept in a system-wide manner of all device behaviour, so that meaningful data can be extracted.

After taking into account that the data generated is currently meaningless, no study of it was implemented, and it is only being stored in order to show some of the compression tactics that can be used when scaling the system.

4.2.5 Storing device responsiveness

We define device responsiveness as the time it takes a device to reply to the kernel module; the longer the delay in a response, the less responsive the device is. To measure this, message headers are used. These include a timestamp from when the message was created. This timestamp will be in the same time reference for both kernel and device since they execute in the same physical context.

By calculating the difference between the time the message was received at the kernel and the time the message was sent, we can get the current delay between both. In reality, we only need to store responsiveness measurements for acting devices. The purpose of keeping track of responsiveness is to be able to make an informed decision on which device a task should be allocated to, when multiple devices are registered in the system. The way responsiveness progresses over time can be used to measure whether the device is currently overloaded.

The same infrastructure is used for this as with network status measurements. Measurements have a time to live, after which they become invalid. The latest responsiveness value is stored, as well as the average responsiveness and the jitter (deviation) of the values. If the latest measurement had a delay bigger than the average delay plus the jitter, we consider the responsiveness of the device to be decreasing. If responsiveness is decreasing, this generally means that the device is becoming, or is, overloaded.

$$Overloaded(delay) = \begin{cases} 1 \; if \; delay \geq \overline{delay} + jitter \\ 0 \; otherwise \end{cases}$$

With all the functionality from the past three sections, we now have enough information to be able to create requests and allocate them into the system. The sections that follow will focus on this.

4.2.6 Creating requests

Once a measurement has been received, a decision has to be made on if it should be acted upon. In order to make this decision, we first fetch all information we have on the device which spawned it. We check all of its possible actions, and the value ranges for which the action must be triggered. For each case in which the value is in the value range, we spawn a request.

The task in creating a request lies in creating a packet which can be released into the CPN network, will be received and understood by one of your neighbours and will be able to find its way to the best matching node. The only information that we have available is the device or devices that this action can be performed by. It may be a device in particular, or it may be any acting device which receives it. Since there can never be a device with an ID for 0, we use this value to signify that any acting device may claim this job.

The minimum information that can be sent in a request is: the device it is meant for, the priority of the request and the action it represents. We have also included additional information such as the device which originated the request, but this is merely additional information for statistics purposes and not at all necessary.

Source device ID
Destination device ID (0 if any)
Priority
Action ID

Table 9- Fields in an action request

We do not know the address and status of any nodes on the network aside from our neighbours. For our neighbours, we have information on the number and types of devices they have registered, but this information can't be the sole thing used in order to make a decision; a node may not personally have any devices, but may be connected to other nodes that do.

A dummy destination address is chosen, such that no machine in the system will have it. In our case it was chosen to be 0.0.0.0, but this can be set to any other value which would never occur. This way, the packet will travel through the network indefinitely attempting to find this 'fake' address that does not exist. In the next section the method used to allocate these packets to a node will be explained.

4.2.7 Allocating requests

The first decision to make is whether the request should be kept at the current node. If the current node has viable acting devices, the request will be kept and executed locally. While it may have been possible that a device exists in the network which is more efficient than our local devices, it is assumed that the network will always have a negative impact on performance. In these conditions, a local device

can be expected to have a better performance than a remote one. If communication with the device fails, then the request will be sent into the network.

Once it has been decided that the request should exit the current node, we first check whether there is a known path to the destination address. If this is the first time a request has occurred, no path will be known, so that we cannot send the request just yet. The request is added to a processing queue, where it waits for the time being. If a path was already known, then the request will be sent along this path, and an ACK packet will be expected at some point in the future.

Once we have either sent the request or not, we update our path towards this destination. We achieve this with the use of smart packets. Smart packets, along with dumb packets, both have dummy destination addresses, so that if nothing is done by the nodes in the network, the packet will wander until it expires. What was implemented in this case is a check at each node in the path. Each node that receives the packet looks at the packet's contents, checking whether they have an acting device which fits the requirements.

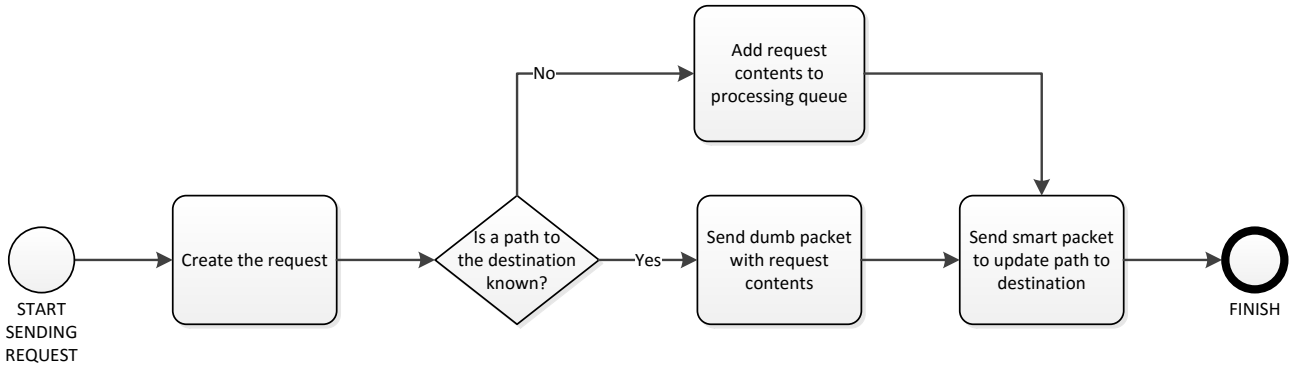


Figure 11- Sending a request in CPN

If they do have such a device, the packet will be claimed by the node, and an acknowledgment sent to the request's originator. This is used in smart packets to find the shortest path to a matching acting device. When an ACK is received by the node which created the request, it will update its list of stored paths, one per neighbour involved. If there were any requests to be processed, they will be sent at this point.

Figure 12- Updating paths in CPN

The next hop for a smart packet is chosen dynamically at each host, using the RNN to determine which of its neighbours results in the most reward for the packet's QoS objective. If there is not enough information, a random decision will be made on the matter.

This way, we have an infrastructure which continuously updates the best path to an acting device, and which can send it requests. An additional feature that was implemented is that, after a path has been found and the request sent, if during the request's path it encounters a node which can take the request, then this node will keep the request rather than keep forwarding it.

This was implemented with the objective responding in an ever quicker way to changes in the device configuration in the network, so that if a better matching node is found on the way, this node will be pre-emptively chosen, and delays and/or loss and/or power used minimised. Once a node has claimed a request, it will be added to its job queue.

4.2.8 Job queues

Jobs are inserted into a queue according to their priority. Higher priority tasks are inserted at the beginning of the list, and will be processed first. Priorities were chosen to be similar to those used by Linux when scheduling processes. Linux uses a scale from -19 to 20. This was adapted for the system to go from 0 to 39. This is equivalent, but removes the need for using signed integers when an unsigned integer would suffice. This eliminates the need for keeping track of the sign of the priority.

Whenever a job is read from the queue but not actually executed, its priority is increased by 1, so that the longer a request remains waiting for its turn, the higher the probability that it will be executed first. This removes the risk of a request suffering from starvation because of high priority requests continually arriving.

In the TAP, the job list is written to a file, which is then read, parsed and executed. No functionality was originally implemented to read from the job queues, which were originally two, one for each type of job. The first change to this section was to combine both job queues. The controller (described in 4.2.9 Device controller) will periodically fetch jobs from the queue, and attempt to execute them. Since the queue is ordered by priority, there's no need to implement any kind of sorting when fetching from the queue, it is only needed to fetch from the head of the queue.

In order to speed up processing, the amount of jobs read from the queue at a time is limited by a threshold. This serves two purposes: so that not too much time is spent reading the queue, and so blocking the ability of other threads of execution to add jobs to the queue; and to avoid swamping the devices with more requests than they can handle. Once this limit is reached, or there are no more jobs to read, the fetching of jobs finishes. Some load balancing already occurs at this stage. We may decide to not send the job over for execution, but to forward it to another node.

The job queue provides an intermediary data structure in which to store requests while they are waiting for execution. Requests may return to this stage if they fail at being assigned, or if the system is currently too busy to handle the requests.

4.2.9 Device controller

The controller, as its name indicates, controls the assignment and execution of tasks. It is also in charge of managing the list of devices, and finding suitable devices for a task. When a device is registered or de-registered, the controller is the one that modifies the list of devices.

We also keep track of the number and type of devices registered at neighbours. This is not currently in use, but is a prototype of a different type of load balancing that could be added to the system. This information could have been used in order to implement device swapping between nodes. Knowing the number of devices at the current node, and at its neighbours, we can get an image of what the

load in this region of the network is. If too many devices were to register at a particular node, and none at the neighbouring devices, we could use this information to swap one node to another, along with the measurements associated with a particular node. This was the original idea for implementing node swapping into the system, but this would have required extensive changes either in communication infrastructure, or the infrastructure of CPN itself.

The controller runs a periodic task that checks if there are any jobs to execute. If there are any, then it will first try to find a viable device for them. In order to find if a device is viable, we check which devices we do not have any responsiveness measurements for, or those that have measurements but they have expired. This is used in order to cycle execution between all registered acting devices, so that not all of the requests go to the same device all the time. If we haven't used a device in a while, probably because it had become slow, after the responsiveness measurements have expired the device is probably back to being responsive. This is the logic that drives this algorithm: if we don't have up-to-date measurements on a device, it should be used once again in order to confirm that it is still unresponsive.

If we have responsiveness measurements from all devices, we choose the most responsive device when allocating the request. If no viable device was found after searching all devices, the request is sent for allocating once again. In this re-allocation stage, we disable the functionality which allows us to keep the job. It is assumed that if the node has made the decision of re-allocating the job, then it cannot currently deal with it and it will not be tried again.

The same occurs when we send a request to the device but the device does not receive the message. In the original design, this would be taken as a sign that the device did not exist anymore, and so it would be removed from the system and ignored as a suitable device. After experimentation, it was found that at high rates of sending, the API cannot deal with the speed of sending from the user-side due to its synchronous nature. The device is still working in this case, and should not be removed.

This is why in the current design the device isn't deleted from the node, and the request is forwarded. It was considered that the request could be re-inserted into the job queue to be re-attempted. There are arguments to be made for both keeping and forwarding the request, but in the end it was decided that the request should be forwarded in order to reduce allocation time in the system. The full analysis on this matter is in Chapter 6: Evaluation.

4.2.10 Load balancing

When assigning jobs to devices, we do not only check whether a viable device exists. Another variable to take into account is the load of the system. In order to reduce load, we first need to analyse what variable wants to be maximised when assigning a job. Since we have no control over loss of packets, and the medium is mostly reliable as long as the rate of messages sent does not exceed a certain value, and the power used to communicate will remain constant, the only variable that we can control when communicating with a device is the delay, or responsiveness, in communicating.

From 4.2.5 Storing device responsiveness, we have the infrastructure needed to check the responsiveness of each of the acting devices registered at a particular node. Before any jobs have been assigned to any of them, we do not have any information, but a job will be allocated to a random device which has not been used before. This is in reality pseudo-random, where the first device to be

registered at the node will be the first device to be contacted, and the first device to be re-contacted once responsiveness measurements have expired for all devices.

Along with the concrete value for device responsiveness, we are also storing whether the value is increasing or decreasing over time, taking jitter into account. We can therefore consider that a device is overloaded if its responsiveness is decreasing. Now that it is known whether one particular device is overloaded, we needed a way of determining whether the whole system is overloaded. The idea behind this was, if the system is considered to be overloaded, requests will not be assigned to its devices, but rather forwarded to other nodes in the system.

Some test cases were ideated in order to find a satisfactory pattern for when the system is overloaded:

TOTAL NUMBER OF NODES	NUMBER OF UNRESPONSIVE NODES REQUIRED
1	1
2	2
3	2
4	3
5	4
6	4
7	5

Table 10- Analysing the conditions for system overload

The pattern can be expressed as:

$$n = \lfloor 0.7 * N \rfloor$$

Where N is the total number of nodes, and n is the number of unresponsive nodes required for the system to be considered overloaded.

A system is checked for the property of being overloaded at two points in time: first when fetching a job from the queue, and later when assigning the request to a particular device. Normally there should not be a change between the two points in execution, since they are executed consecutively, but the amount of requests read from a device may be a substantial amount. Each time a request is assigned, the state of the system will be checked, in case it has changed in the middle of assigning jobs. This is useful for the cases in which the system has become overloaded because of the rate of assigning jobs; whenever a device fails at receiving a request, it is automatically classified as overloaded.

In order to limit the amount of times that a request can be passed around without being executed, a maximum number of bounces was set. If this amount is reached, the node which currently holds the request must either execute it or drop it.

4.2.11 Keeping statistics

In order to measure the performance of the system, statistics are kept about relevant facets of the system. The number of bytes sent and received is kept track of, as well as the number of requests created, executed, and forwarded.

Other relevant measurements which are tracked are average responsiveness and average delay, as well as the current and grand total of devices which have been registered. Whenever Netlink drops a message, this is also recorded.

These statistics are the backbone of all graphs which have been generated for Chapter 6: Evaluation. They help us get an image of the status of the system over time.

Chapter 5: Implementation details

This section is focused on any relevant implementation details that were not mentioned in Chapter 4: Analysis and design. Fragments of code will be included when they would help illustrate the point. By the end of this chapter, the reader should have an approximate idea of the structure of the code implemented, as well as the development process followed.

5.1 Programming language

The programming language used is C. This was chosen in order to be compatible with the original work, which is developed in this same language. The Linux kernel is written in C, and while C++ is compatible with C, mixing C and C++ files is not ideal.

Refactoring the code into Python was considered as a possibility at one time, because of its ease of use in cloud-like systems such as OpenNebula (OpenNebula Systems, n.d.). This was rejected as the amount of functionality already implemented is substantial, and most of the development time would have been spent in rewriting, rather than implementing.

5.2 Development methodology

This project was developed using Git as a version control system. A private repository was created using the Department of Computing's GitLab service. As new functionalities were implemented, these were added as commits into the system, so that it was possible to roll back to previous working versions. The repository contains all files necessary to run both the kernel module and the device, including graphing utilities.

There were physical machines available for testing at a laboratory in level 10 of the Department of Electrical and Electronic Engineering. These machines were already set up for CPN and the Task Allocation Platform (TAP). The first stages of this project were centred on analysing the already existing code. Meetings were had with Lan Wang, the creator of the TAP, where the general flow of the system was discussed. This stage of the process was slow. The Cognitive Packet Network consists of many files which run concurrently, so that it can be complicated to trace the flow of execution.

The project relies heavily on accurate knowledge on Linux kernel modules, hence extensive research on the subject matter was required due to lack of previous experience. Developing a kernel module poses difficulties in debugging. While logging is possible, faulty code causes kernel panics and crashes. Thus, developing kernel code on a physical machine is not viable, since it would have to be manually restarted at every crash, which is not an efficient use of the time available.

It was decided to use virtual machines (VMs) to develop the system. Virtual machines are simple to restart and configure. By replicating the same VM multiple times, a real network environment could be simulated. This stage of development consisted of isolating the software requirements needed for CPN to successfully run on a machine. More details can be found in Chapter 7: User Guide.

Once a stable configuration was found, the design could begin to be implemented. Separate branches were used in Git for the main working code and new developments, so that there was always a working version available.

Load balancing was attempted to be implemented first, mistakenly thinking that it would be simpler to implement than the refactoring of the whole job execution system. Once it was clear that this refactoring would have to occur first, this was the next point of focus. The time spent on the first attempt at load balancing aided in gaining a deep understanding of the system, so that when it was attempted again at the end of the development process, it took much less time to accomplish.

The final stage of the process was to test how the system behaves depending on network configuration and number of requests per second. A graphing and statistics keeping infrastructure were implemented for this purpose. Since a set of data is generated per host per test, in order to keep the amount of data manageable the size of the network was limited to three nodes. Details on precise testing conditions can be read in Chapter 6: Evaluation.

5.3 Internal structure of the system

The kernel module is separated into multiple C source files. Each pair of source files (.c contents and .h header) represents a separate process. Processes execute concurrently and asynchronously. They have knowledge of all the other existing processes spawned by the module, so that function calls can be used to communicate between processes.

At the beginning of execution there is one process, implemented as `cpn.c`, which starts all others. Its task is to initialise and exit the module properly, as well as to set up all network devices and sockets. It is the backbone of the system, processing input and output from all devices, and re-directing it to relevant processes.

It is paired with a header file, implemented as `cpn.h`, which contains the common data structures and definitions. Definitions are used to set up constants which alter the parameter of execution, so that they can be modified in one centralised place and ripple to the rest of the processes. Some examples follow:

```
#define CPN_RAND_ADDRESS 0 // 'Don't care' destination address
#define NETLINK_USER 31 // Sets the port to which netlink listens to
#define CPN_VERSION 3   // Sets the module version
```

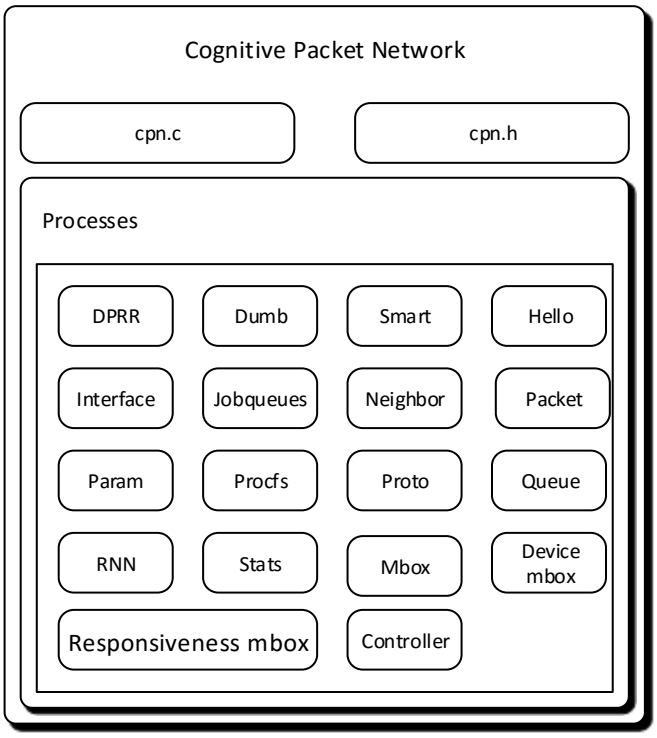


Figure 13- Overview of the processes in CPN

Some of the functionalities were already implemented in the system, and have not been modified during the development process. The reason for this is that they are still applicable to the current system, and could be used with minor changes (such as simple re-structuring). This was the most practical way of operating on a system of this scale.

UNCHANGED FUNCTIONALITIES	MODIFIED / ADDED FUNCTIONALITIES
Dumb Packet Route Repository (DPRR)	Dumb packets
Interface management	Smart packets
Parameter management	Hello packets (minor addition)
Procfs (Process management)	Neighbour management (minor addition)
Proto (QoS management)	Job queue management
Queue management	Packet management
Random Neural Network (RNN)	Statistics management
Measurement boxes	Device measurement boxes
	Responsiveness measurement boxes
	Controller

Table 11- Unchanged vs added functionalities

A short description of how the whole system is implemented follows. The bolded terms each represent a process in the system.

The **Dumb Packet Route Repository** keeps track of all known routes. Whenever a smart packet is received, the DPRR is updated. There are any previously unsent requests to the network waiting in the **queue** process, they will be sent at this point. Requests are added to the queue whenever there is no known path to the destination.

Requests are sent via the **dumb** process to **neighbours**, the list of which is kept updated via a series of **hello** messages that occur every x milliseconds.

Smart packets are created every time a request arrives. The next hop a smart packet will take is decided either randomly (if there's not enough information), or using the **RNN**. The amount of randomness is determined by **param** (short for parameters). RNN weights are updated according to the reward system implemented in **proto**. Smart packets find the best route for that QoS objective and destination. The values associated with a particular route are kept in an **mbox** (measurement box).

Packets are sent and received by the **packet** process, using a networking **interface**. Whenever a request is received at the node, it will be added to the **job queue**. The **controller** regularly checks the job queue, and will re-send the packets into the network if the node is currently overloaded.

Measurements by measuring devices are kept in the **device mbox**, and the responsiveness of acting devices is kept in a **responsiveness mbox**. Responsiveness values are used to determine whether the node is overloaded or not.

Finally, **statistics** are kept track of, and every process can print the data it holds to file using the **procfs** interface.

5.4 Data structures

Each process has the same basic data structure, which is inherited from the common header file `cpn.h`. The data structure that is most frequently used is a type of list called `list_head`. This structure is Linux's implementation of a linked list.

Using the operating system's own structures guarantees that these will be optimised for fast(er) access. Operations on these lists are sufficient for our needs, since we only require traversal, insertion and deletion. They do not support sorting, but a simple **insertion sort** algorithm was implemented. This algorithm has a complexity of O(n^2), but it is more efficient in practice compared to other quadratic sorting algorithms.

It was chosen because of its simple implementation, and the assumption that the list it applies to would always contain a small number of elements. This holds true since the job queue list this is applied to is constantly being read from, and its elements removed, resulting in a permanently short list.

Every process in the system contains the same type of data structure, following the same naming **convention. It** contains both a list of elements, and a lock which must be used to access this list. This structure is defined in `cpn.h`, so that it can be modified for all processes concurrently. In order to create an instance of this template structure, it can be called using

```
STRUCT_DEF(name_of_process);
```

And it is implemented as:

```
#define STRUCT_DEF(name_of_process)
typedef struct cpn_##name_of_process##_struct {
    struct list_head list;
    rwlock_t lock;
} cpn_##name_of_process##_struct;
```

The elements of the list can be of any type, and it can even contain elements of different types (though this is not utilised in our implementation). Thus, this structure can be defined this way regardless of the elements the list contains.

In the particular case of netlink messages, this does not allow the use of a structure such as this linked list, since `list_head` only transmits the pointers to the data, not the data itself. It is for this reason that the structures which hold a device's requirements do not use `list_head`, but standard arrays. In order to manage these arrays, extra fields are added to the structure, which keep track of array size. Since an array's size needs to be known at declaration time, we define a constant which constrains the maximum size of the array.

```
#define MAX_ENTRIES 5
// Device info
typedef struct device_req {
    u_int32_t device_id;

    /*  ...
    other data
        ...
    */

    int num_entries; // Keeps current number of entries
    struct act_req act_info[MAX_ENTRIES]; // List of possible actions, maximum
number of entries is MAX_ENTRIES
} device_req;
```

5.5 Concurrency

Concurrency is achieved via the use of processes, as was mentioned in previous sections. Since one or more of these processes may be accessing the same data, there has to be a way to ensure that they do not conflict with each other. This was achieved using the read/write locking system provided by Linux:

```
write_lock_bh(&cpn_jobqueue_list.lock); // Locks for write
read_lock_bh(&cpn_jobqueue_list.lock);  // Locks for read

write_unlock_bh(&cpn_jobqueue_list.lock);   // Unlocks for write
read_unlock_bh(&cpn_jobqueue_list.lock);    // Unlocks for read
```

The main downside to this method is that this structure's exception management does not handle re-taking the lock from the same resource. If a process is holding a lock, and re-attempts this operation without releasing it, the system will not detect this and a deadlock will occur. In this situation, the whole system will crash without logging, so that it is hard to track the reason for failure.

This has occurred several times during the development process, and exclusively happens when a process takes a lock, and then calls a function which does so again. In order to bypass this, measures had to be added to notify the relevant functions that the lock is already held by the caller, and that they do not need to do so again.

5.6 Interrupts and alarms

Alarms and signalling are utilised by devices in order to finish execution gracefully. Using this, the system does not need to deal with constant time-keeping, which would not be able to be done accurately because of blocking waits. A signal handler is set at the beginning of execution. It is set to occur when an alarm happens, or when the process is told to finish execution by the kernel. This is to ensure that our signal handler will always be called before the end of execution.

```
signal(SIGALRM, signal_handler);
signal(SIGINT, signal_handler);
```

An alarm is set to be triggered after the desired time period has passed:

```
alarm(device_life);
```

The signal handler's task is to clean up after the device:

```
void signal_handler(int sig){
    fprintf(stderr, "device life exhausted. shutting down.\n");
    signal(SIGINT, SIG_IGN); // Ignore any other termination signals while
we're cleaning up
    cleanup();
    exit(0);
}
```

5.7 Timers

Timers are used for recurring actions, such as sending a ping to neighbours or checking the job queue for new jobs. Once again, the implementation provided by Linux is used.

```
struct timer_list cpn_controller_timer;
```

A timer consists of an expiration time and a call-back function.

```
setup_timer(&cpn_controller_timer,
    cpn_controller_callback,
    0); // Creates timer

mod_timer( &cpn_controller_timer, // timer to modify
    jiffies + msecs_to_jiffies(cpn_controller_interval) ); //new expiration
time

del_timer_sync(&cpn_controller_timer); // Deletes timer
```

When the call-back function is reached, it resets the timer.

There is currently an issue in deleting the timer when exiting the module. Attempts to stop it fail, and after some research the issue seems to lie in that the timer could possibly be set after it has been deleted. Even though there is a check in the call-back function to detect whether the module is exiting, in which case the timer is not reset, at runtime the API is not aware of this, and thus prefers to leave the timer untouched. This has not caused any crashes so far, so it has been logged as a minor bug.

5.8 Allocating memory

Allocating memory in the kernel was the origin of an interesting problem which would cause the kernel to crash. If memory was not allocated atomically, but with the usual allocation method, this could cause the process to sleep while memory was being allocated. This would conflict with the timer, since they could not access the call-back function while this was happening.

The solution was simple in practice (change the memory allocation options), but it means that the memory used must be RAM and not disk. RAM is usually smaller than disk so this could be a problem in the future, if the amount of data to store grows.

5.9 TCP/IP packets in Linux

When a request is created in the system, a packet needs to be created from scratch to be sent through the network. The network protocol that is utilised is TCP/IP, because of its wide-spread use. Any machine would have access to a TCP port. Linux uses `skb` as the packets that are sent in all communications.

This, along with the user-kernel communication, was one of the hardest functionalities to implement. Slightly incorrect headers mean that the message is not received at all, and the reason is hard to make out; you only have access to the sender's data. Documentation was easier to find than for netlink sockets, but differed too much between Linux versions to be useful.

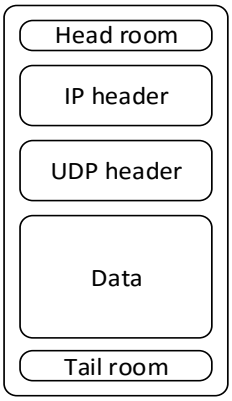


Figure 14- Structure of a skb packet

Space is allocated such that the packet won't run out of space even if the route is the longest possible (route lengths have a length limit, to prevent packets from circling endlessly). Since we do not know the destination address beforehand, and requests can be allocated to any viable node, a dummy routing table is generated, which will estimate a path using knowledge of neighbours.

```
struct flowi fl = { .nl_u = { .ip4_u = {
    .daddr = CPN_RAND_ADDRESS,  // Dummy destination address, does not exist
    .saddr = local,                 // Source address
    .tos = IPTOS_RELIABILITY } },   // Type of service
    .proto = IPPROTO_IP     // IP protocol to use
};

  ip_route_output_key(&init_net, &rt, &fl); // Generates dummy routing table
```

Chapter 6: Evaluation

This section will examine the behaviour of the system through experimentation, in order to determine whether the goals set in Chapter 3: System Description. The testing environment used will be described, as well as the hypothesis that were had before the trials began. The last topic to be examined will attempt to evaluate the characteristics of the system versus the desired characteristics. Some experimentation methods for further investigation will be proposed.

6.1 Testing environment

The bulk of the testing was carried out using Virtual Machines. The reasons for this are the same as the ones detailed in 5.2 Development methodology; a machine crashing has much less repercussions if they are cordoned off from the actual hardware. It is also simpler to change the network configuration in virtual machines in order to generate multiple topologies.

The software was also installed in some of the CPN machines provided by the Department, and sanity checks were carried out to ensure that it still behaves as expected in a physical machine.

The network used was local, and was configured for different topologies. Since it is a local network, any delays in assigning a job are completely due to system overheads. The amount of nodes available was limited, causing only ring and star-like configurations to be possible.

The number of machines used was 3, since this is the minimum amount that allows for the full functionality of the system to be tested. Having three machines was already difficult to manage for gathering results, since each of them produce a full set of results per test. In order to maximise the number of tests carried out, the number of nodes was reduced to the minimum.

By generating different types of devices at each node, or not generating a device at all, different test cases can be generated. With three nodes, all possible cases that occur in the algorithm flow can be simulated in one execution run:

- A node creates a request and keeps it.
- A node creates a request and forwards it.
- A node receives a request and forwards it.
- A node receives a request and keeps it.

A VM was originally created, and the other two generated by cloning this machine. Thus, all three machines have the exact same specifications. There shouldn't be any CPU-intensive applications running in the background that affect execution time, but this cannot be assured due to the OS itself being able to spawn processes whenever necessary, such as diagnostic jobs. This may have an effect on results.

Disk memory size	2GB
RAM size	400MB
Number of CPUs	2
Maximum execution limit	100%

Table 12- Virtual machine settings

All requests execute the same action, and this action has the least amount of impact possible in the execution time, so that it doesn't affect the behaviour of the system. Listing the directory was the action chosen, since it is one of the most used system calls and has been optimised for fast execution. Timing this command on the machines gives an average execution time of **4ms**, so that any effect this may have on the system can be discarded.

Measurements were set to always trigger a request, in order to better test how the system reacts under stress. The QoS objective that was chosen for every test was delay, so that we attempt to minimise delay over all tests. The objective was to prove that the variable chosen was being minimised/maximised over time; since the interface is the same for all QoS objectives, results for delay should be representative of the other objectives.

By controlling all these variables and keeping them constant throughout all testing, we can reach conclusions which are the closest to reality as possible, and that only depend on the variables being modified and no other factors.

Topology was also further customised by limiting the amount of neighbours per node. This forces the network to not be fully connected, and is useful for finding corner cases which the system might be vulnerable to. All connections are bidirectional.

6.2 Hypothesis

There are two hypotheses to be explored here, pertaining to the necessary and desired requirements of the system, respectively. The main objective that the system must fulfil is that the deliverable must be able to assign jobs between nodes in the network, so that a job is created in node X, and it is executed in node Y.

This is simple to test for, but non-trivial to quantify. A simple way of testing this is via the use of actions which trigger a message to print on the screen. As a result, the hypothesis can be described as the following:

HYPOTHESIS 1
A working system will print output on node Y whenever a measurement is generated in node X, provided both nodes are interconnected, that the system isn't overloaded, and that there is no other node to forward the request to.

This must be true regardless of the condition of the node and the network.

Once a basic working system has been established, the desired functionalities come into play. Load balancing was the main feature introduced into the system during development, so that its effect must be measured. The hypothesis is as follows:

HYPOTHESIS 2
A network is balancing the load between all its nodes if the amount of requests forwarded increases proportionally to the number of requests received per second, and decreases proportionally with the number of acting devices at each node.

These hypotheses will be proved or disproved in the next section via the use of repeated experiments on the system.

6.3 Experimentation

The first of the hypotheses is quite trivial to prove for simple cases, but impossible to prove with a certainty of 100% reliability. For higher rates of requests per second, the number of prints is not feasible, so that we simply take into account the statistics at each of the nodes.

Each node keeps track of its number of sent and executed requests. If the total number across the network of each is equal, it can be said that every time a request is generated it is executed, thus proving that the system is functional. The downside of this method is that it is the nodes of the system itself which keeps track of these statistics- there is no guarantee that they are indeed correct, or that the device did execute the command, only that the command was sent to it. A device could always crash in the middle of execution, regardless of how sophisticated the network is, so that this approximation should be close enough to reality.

The approach taken in this respect is that the tests should cover as many corner cases as possible. If the number of generated and executed requests is always equal regardless of the network conditions, then the system can be said to be reliable.

The first attempts were focused on trying to make the system fail by sending it increasing number of requests to process per second. In the earlier stages of the system, numbers higher than 100 requests per second would cause the system to crash. The main bottleneck is always with the Netlink API and sending the requests to the user side of the node. By adding better safeguards to the kernel module whenever the user-side couldn't handle the number of requests, it is now possible to increase the number of requests to be generated as fast as the CPU can handle it.

This was simulated by removing any sleeps from the device code, so that measurements were sent as fast as they are generated, and removing the infrastructure that waits for a reply from the kernel before sending another measurement. Even with these temporary modifications, the system could handle the workload without crashing.

The system performs as expected in all tested network configurations. Some problems that were encountered and fixed during testing about how the system would react when it got returned a request it had originally forwarded to another node. In this case, we mean the exact same request- if an acting device existed in that branch of the network, it could have created a different request with the same requirements; if no devices exist on that branch, then the same request could return to the original node.

A check was implemented to see whether the node had originally created the request that it had now received again. In that case, the node assumes there's no acting device on that branch of the system, and will either keep the request (most probable outcome) or forward it to a different branch of the network.

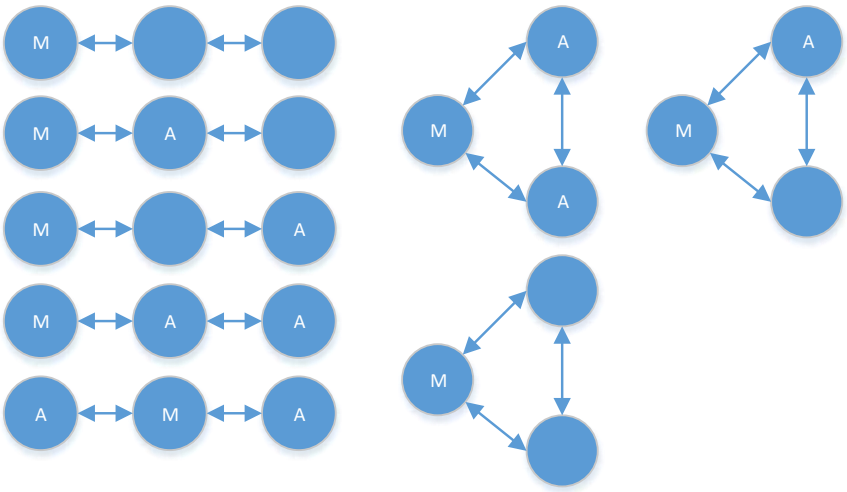


*Figure 15- Network configurations tested. **M** stands for 'measuring device', **A** stands for 'acting device'.*

For each test and each host, different variables were measured. From these, graphs were plotted. The bulk of the graphs can be found in Appendix B – Graphs. Only a portion of them have been included; this decision was made based on the fact that the data is mainly repetitive, as well as too extensive to show in a report of this type. A full list of all the statistics measured, as well as an example file, can be found in Appendix A – Statistics files.

The most relevant out of these graphs are those that deal with measuring the load of a node, and with minimising the delay in the network. Delay can be seen to decrease over time on all tests.

When having multiple nodes, the effects of load balancing can be seen in the generated graphs. When node X, which originally received the requests, becomes overloaded, it starts forwarding its requests to node Y. At that point, node Y becomes overloaded and returns the requests back to node X, which becomes overloaded again, etc. This follows from Figure 17.

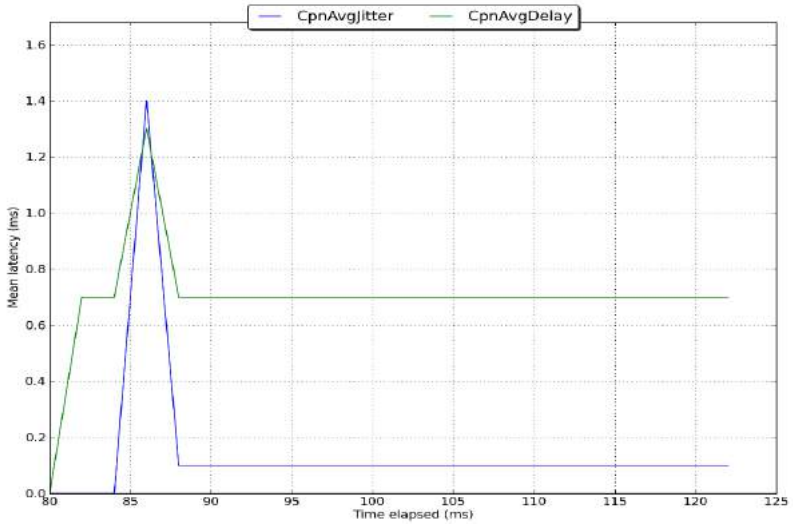
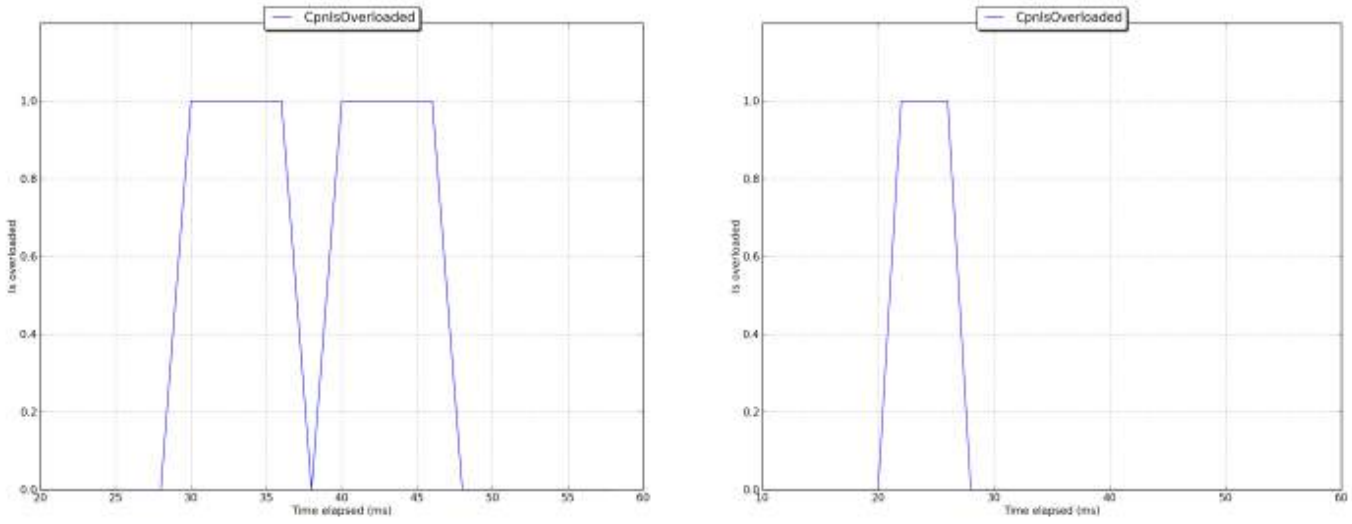


Figure 16- Graph showing how network delay varies over time.



Node X Node Y

Figure 17- Graphs comparing the system load between two nodes. Timing shifts must be allowed between both; the systems have different clocks.

Once again, further details can be found in Appendix B – Graphs.

Much more interesting is analysing the load balancing in the system. This was assessed in multiple ways, involving one or multiple nodes.

In order to measure how the percentage of forwarded jobs compared to the number of requests, only one node was used. The reason for this is that having multiple nodes means the requests would return to the node, and skew the results. Having one node, requests will be dropped rather than forwarded, so that they do not 'return'. This way we can isolate and study the behaviour of a node in the system as the number of requests it receives per second increases.

From hypothesis 2, the percentage of requests forwarded should increase as the number of requests per second increases, and decrease as the number of devices increases. Results are shown in Table 13.

The results seem to follow the expected trend. Requests forwarded increase with the number of requests in a non-linear way. Increasing the number of devices does decrease the number of forwarded requests, but perhaps does not have as big an effect as might be expected. By tripling the number of devices at 1000 and 10000 rps (requests per second), percentage of forwarded requests only decreases from 66.4% to 55.1% and from 78.5% to 61.2%, which is a meagre 11.3% and 17.3% less, respectively. It does not decrease by three times the value or is even remotely close to it.

# of requests per second	*Device life(s)*	*# of acting devices*	*# of kept requests*	*# of forwarded requests*	*Percentage forwarded*	*Time taken to execute (s)*
1		**1**	**30**	**0**	**0%**	**30.071**
		2	30	0	0%	30.026
		3	30	0	0%	30.011
10		**1**	**221**	**78**	**26.1%**	**31.362**
		2	263	35	11.7%	30.876
		3	286	13	4.3%	30.140
100		**1**	**2577**	**266**	**9.4%**	**34.090**
	30	2	2528	139	5.2%	30.109
		3	2536	50	1.9%	31.006
1000		**1**	**6789**	**13403**	**66.4%**	**32.327**
		2	9511	12808	57.4%	30.591
		3	9515	11676	55.1%	33.423
10000		**1**	**7257**	**26529**	**78.5%**	**49.870**
		2	9771	27628	73.9%	30.754
		3	12272	19357	61.2%	40.928

Table 13- Analysis on the amount of requests forwarded vs kept

The next point to focus on was figuring out why this occurs, and three devices do not always reduce the percentage forwarded by at least 1/3. The problem was identified as being caused by the maximum number of requests that are sent to acting devices at a time. Once the number of requests sent at a time increases higher than 100rps, the acting devices are not able to process the requests fast enough, and this error repeatedly occurs:

```
Error while sending back to user
```

Whenever a request can't be processed, it will get re-routed to a neighbouring node. So, as the number of requests increases, the number of requests which the device cannot process in time increases, and so the number of forwarded requests increases. This explains how the number of forwarded requests does not decrease linearly as devices are added, at higher rps rates.

This could also explain why 100rps has a much better performance than 10rps. While the percentage of forwarded requests is very different, from 26.1% (at 10rps) to 9.1% (at 100rps), the actual number of forwarded requests is in the same ballpark for both of these cases at around ~100-200 for one device, and similarly for the rest. This leads us to believe that there is a constant number of requests that are being forwarded not because the devices are overloaded, but because the devices cannot receive the requests as fast as they are sent.

At these lower rates, the system performs better than expected, by decreasing the percentage of requests forwarded from **26.1%** to **4.3%** at 10 rps, which is a reduction by a factor of 6. At a 100 rps, there's a reduction by a factor of 5 from **9.4%** to **1.9%.**

The trend line that seems to fit this data the best is a logarithmic function, but it is still not a perfect fit, with a R^2 value of only ~0.8. This indicates that number of requests forwarded changes in a $O(log\ n)$ fashion depending on the rps. This performs better than expected, and indicates that the system is quite scalable.

Figure 18- Graph showing requests per second vs percentage of forwarded requests. Trendline formula included.

There is a trade-off to be found between the time taken to execute the requests, and the number of requests forwarded. This can be controlled by modifying the number of requests that are read from the job queue at a time (variable MAX_PROCESSQUEUE_LEN). If a smaller number of jobs are read at a time, there is a higher chance that a device will be able to handle the job, but it slows down the processing of the job queue.

As a comparison, choosing 10 requests to be read from the queue at a time with three devices, at a 1000rps the number of requests kept will be around ~17000 kept, with only around ~2000 forwarded. This results in a percentage of ~10% forwarded jobs, compared to ~55% forwarded jobs when reading 100 requests at a time. But when comparing time taken to process these jobs, 10 requests at a time results in an execution time of ~100 seconds, taking approximately 3 times longer to process the same amount of jobs.

A design choice was made in this respect, in that the responsiveness of the system was considered as a higher priority than trying to minimise the amount of jobs forwarded. Still, this remains a customisable setting in the system, such that a developer may choose which one they would prefer to prioritise. It is currently set to 100rps read, a number chosen through experimentation, trying to keep a balance between execution time and percentage of requests forwarded.

Using this information, we can study the behaviour of execution time as it changes with the number of rps. This would help us get an idea of the time complexity of the algorithm implemented. Results can be seen in the following graphs:

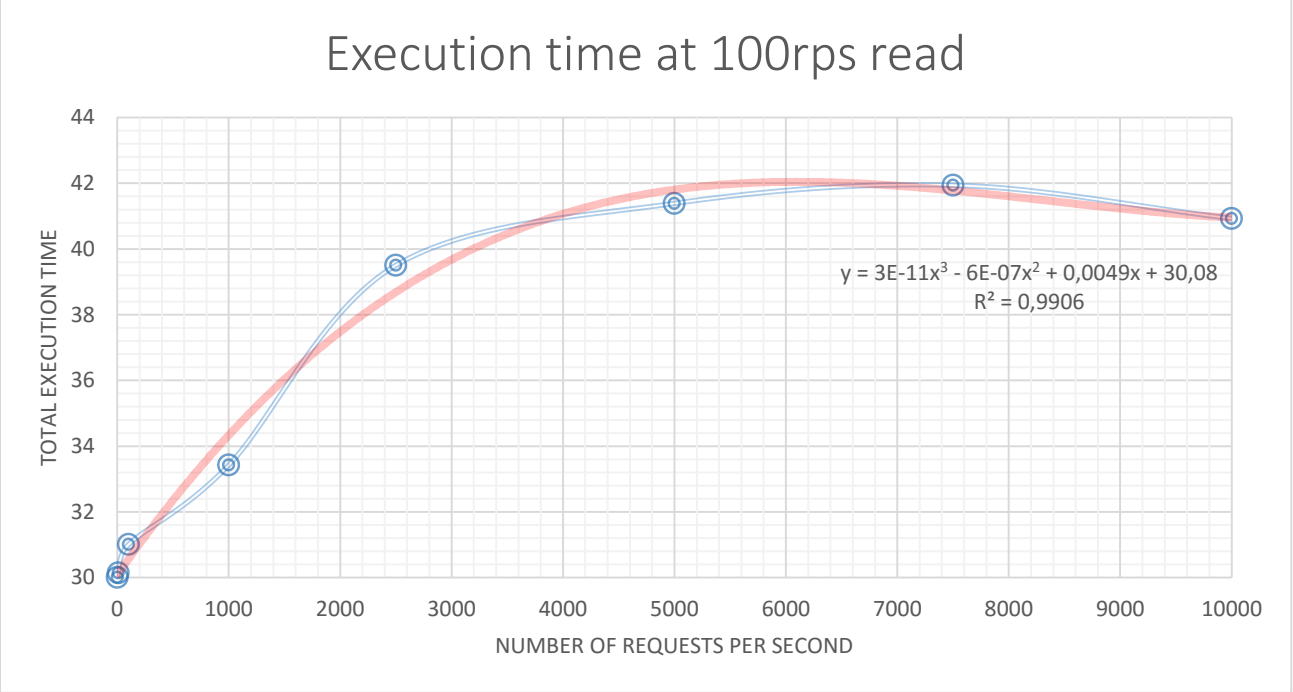


Figure 19- Total execution time varying with respect to number of requests per second. Trendline formula included.

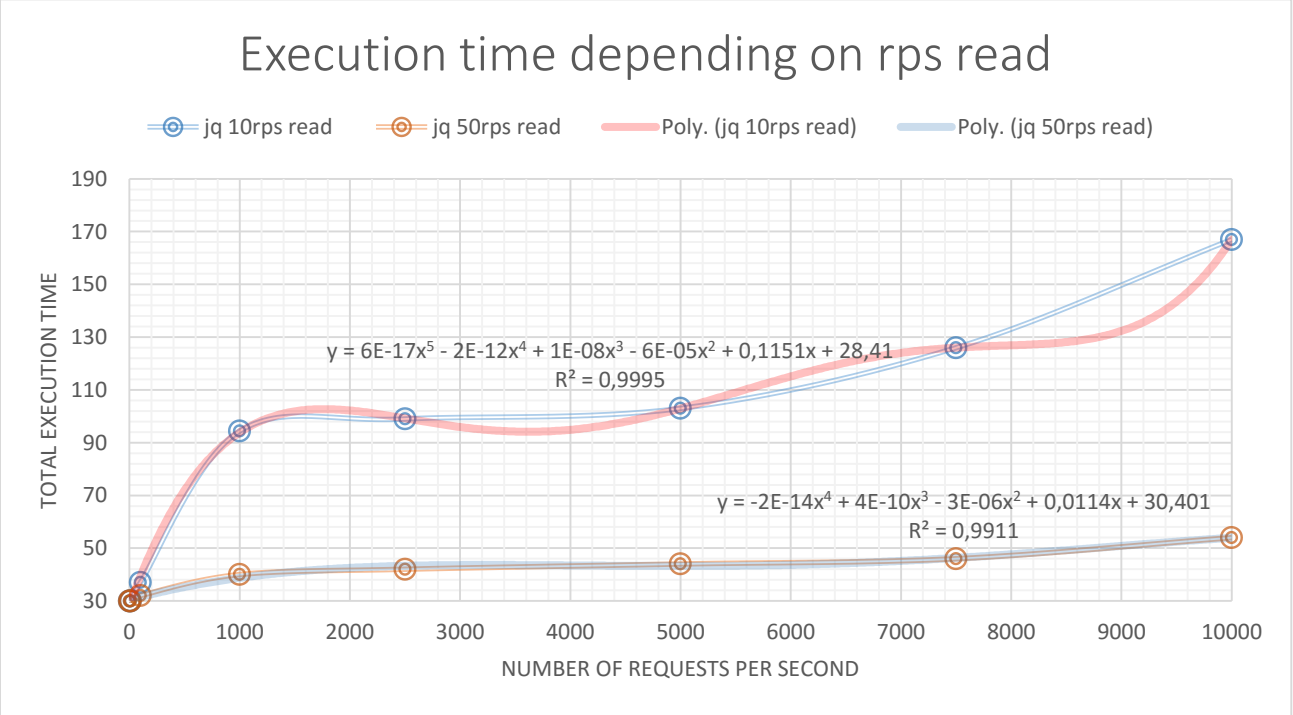


Figure 20- Graph showing how execution time varies depending on the number of jobs read at a time.

The three MAX_PROCESSQUEUE_LEN values tried, 100, 50, and 10, give time complexities of $O(n^3)$, $O(n^4)$ and $O(n^5)$, respectively. Neither of them are ideal due to their polynomial nature, but MAX_PROCESSQUEUE_LEN=100 offers the most scalable growth function, which is why it was chosen. Values higher than 100 would offer better time complexities, but the number of requests forwarded would increase too much, and performance would be affected.

One last thing to note is that, from Table 13- Analysis on the amount of requests forwarded vs kept, it can be noticed that the number of requests sent by the measurement device does not reach the expected amount. A simple example that aids us in noticing this is the one for 10rps, where 299 requests are sent in 30 seconds. The number of requests that should have been sent is 300; we can attribute this difference to overheads in the device code itself.

We can study how the number of measurements sent changes depending on the rps rate; this way we can model the overhead of a measuring device, which we can assume will be similar to that of an acting device, or perhaps smaller. This is based on the assumption that both acting and measuring devices are approximately equal in complexity.

# of requests per second	*Expected # of measurements sent*	*Actual # of measurements sent*	*# of devices*	*Communication overhead (s)*	*Average device overhead (s)*	*Percentage of comm. overhead*
1	30	30	1	0	0	0%
		30	2	0		
		30	3	0		
10	300	299	1	0.1	0.13	0.4%
		298	2	0.2		
		299	3	0.1		
100	3000	2843	1	1.43	1.46	4.9%
		2852	2	1.48		
		2854	3	1.46		
1000	30000	20192	1	9.81	8.77	29.2%
		22319	2	7.68		
		21187	3	8.81		
10000	300000	33786	1	26.62	26.57	88.6%
		37399	2	26.26		
		31629	3	26.84		

Table 14- Excerpt of the analysis of communication overhead on the measurement device's side.

As could be expected, the overhead of a measuring device does not depend on the number of acting devices. Overhead times grows very quickly, reaching a ~90% for 10k rps. This is quite severe; the number of measurements generated does not increase much from 1000rps to 10k rps, even though the total number of requests sent has increased by a factor of 10.

The fact that the same number of requests isn't generated every time makes our previous analyses less reliable; it makes us less able to compare values, since the total number of requests varies per run, and we cannot objectively measure what the real request rate is. When we plot these values, the best fit matches with the time complexity of the system, with a polynomial complexity of $O(n^3)$, as shown in Figure 21.

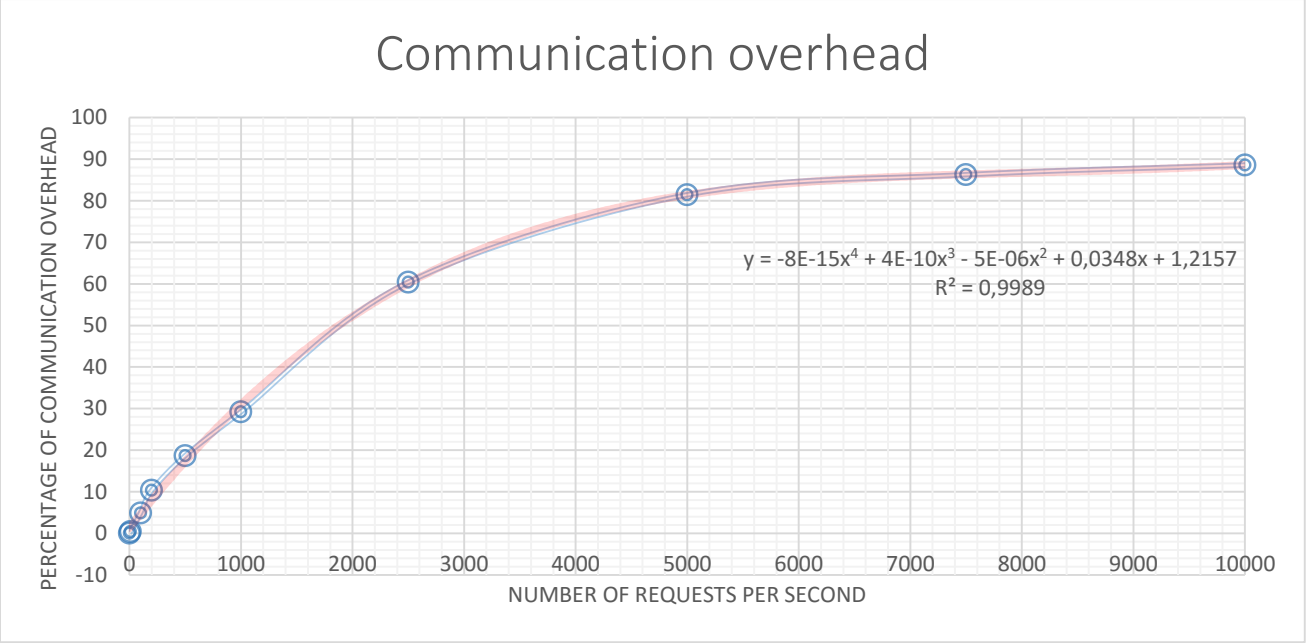


Figure 21- Graph showing how communication overhead increases with number of requests per second. Trendline formula included.

After gathering all the data presented in this section, we can now perform an informed evaluation on our hypotheses.

6.4 Conclusions and Further Work

Let us not forget what the objective of this project was, and the reason why it was designed in the first place. The aim was to implement a working system which would **efficiently** allocate devices, and the tasks they generate, across a network. It should be **reliable and flexible**, so that it can be applied to many different real-life situations. It must be **scalable**, since cloud-like systems can scale infinitely depending on demand. From the results acquired in the above section, we will now attempt to reach a conclusion on whether the system meets these three points.

These points also answer section 6.2 Hypothesis.

6.4.1 Efficiency

We would first need to revisit our definition of efficiency. We could say that the first step to attaining an efficient system is to have achieved a working system, following the definition in Hypothesis 1. It can be said without doubt that such a system was achieved.

We have a network of interconnected node, which are able to communicate between themselves and with the devices spawned in them. Any one node can generate and maintain a device, and keeps track of its measurements, both responsiveness and otherwise. Depending on certain conditions, actions are generated and released for assigning into the network, such that at the end they will be executed at one of the nodes in the system, as long as it contains at least one acting device.

This can be seen from the data gathered throughout testing, from both the statistics files and the data output by the processes which make up the module. Measurements are being stored and can be read from the procfs files, bytes are being sent and received, and outputs are being printed at separate nodes, whenever a request enters the system, regardless of the node it spawns in. Even though there can never be enough data in this respect, we can conclude that we do have a working system.

These were all the necessary characteristics that the deliverable needed to fulfil. Now that this has been established, we can focus on measuring our success in reaching our desired goals. We could define efficiency as the ability to minimise or maximise the QoS objective of the system. This can be said to be achieved for the QoS objectives attempted, since graphs show the delay decreasing over time and then reaching a stable state. More objectives could be tried, but this wouldn't be indicative of the effectiveness of the system- since it is designed to be extensible, it depends on the functions for the particular QoS being added. What has been proven is that the RNN will still find the best path to its objective according to its reward system. It is important to note that the additions made during the course of this project have not affected the already implemented features.

If we consider efficiency from the standpoint of total execution time, the system is not the most efficient, with a complexity of $O(n^3)$. This is not the most scalable, so that if the number of requests per second kept increasing indefinitely, the point in which the system would not be able to finish processing it in time would be reached quite quickly.

This could potentially be solved by further tuning the number of jobs read from the queue at a time, provided you were willing to sacrifice a good growth function for the number of forwarded requests.

From the point of view of the number of requests forwarded, the number of requests forwarded is affected greatly by the fact that "*the user-space listener is not fast enough to handle all the Netlink messages that the kernel subsystem sends at a given rate*" (Neira-Ayuso, et al., 2010).

The system can be considered to be quite efficient, with a complexity of $O(\log n)$ as the rate of requests per second increases. This is much more scalable than a polynomial, and can be considered to be quite efficient, even though it doesn't reach the ideal case of $O(1)$ (constant cost). The system does not, on the other hand, perform as expected as the number of devices increases; further work would need to be done on this respect.

If we consider efficiency from the point of view of overheads, we once again have a time complexity of $O(n^3)$. It was established during testing that this and the point made just before can both be attributed to the API used for sending messages. Customising the protocols used could possibly lead to an increase in performance in this respect.

In conclusion, we have established that our system performs well in respect to load balancing, but could be further improved as execution times and overheads are concerned. The system will perform better than was expected in the cases where the number of requests per second spawned at a particular node does not exceed 100 requests per second by a great amount. This will most likely be the case for our system, so that anything higher than this amount would probably only occur very seldom.

In a real life situation, there probably would be no more than 1-2 devices per node. Remembering that we are dealing with devices which measure the physical world around them, it would probably be accurate to assume that they would not spawn measurements at a rate higher than 10rps. The deliverable can be said to be a working system which fulfils both our hypotheses, and which is efficient for the context it will be utilised in.

6.4.2 Reliability and flexibility

The network was tested using many different network topologies, as well as increasing request rates. The network did not crash upon any of these occurrences, and only experienced minor slowing down at the affected nodes. The system remained reliable even when attempting to test for corner cases, so that we can assert that the system is reliable to some extent.

On the other hand, reliability is quite difficult to test accurately. The system was left online for long periods of time during test, receiving constant streams of input, and did not crash during this time. Further work could be done on this topic, by adding adaptive network conditions which could simulate a slow network or nodes shutting down. This would help get a fuller idea on the reliability of the system, but was not able to be done due to timing constraints, as well as the lack of the infrastructure needed to be able to faithfully monitor this.

In relation to the flexibility of the system, let's examine the number of settings that can be customised in the system. QoS objectives, priorities and the actions to perform have all been successfully implemented. There are points that could be developed, such as making the action creation system more interactive, rather than hard-coding it, since this limits the system's possibilities.

Measurements could also be further customised so that they encompass more than one value, and can be more complex structures. Currently they are only being stored and discarded when the system stops, but an interface could be added in order to store them to files for back-up. Decisions are made only by looking at the last value generated, but this could become more sophisticated in the future so that an average of the past X values is used, in order to reduce the possibility of erroneous values affecting the system.

Node swapping could not be implemented due to the limitations of netlink. As was mentioned in previous sections of the report, there could possibly be work-around to the issue, we could either make the CPN nodes spawn the device code when certain conditions are met, or we could use a different communication system. Taking into account that most of the overhead can accounted to the netlink API, the best choice would probably be to concentrate efforts on optimising communications further.

Security issues are mostly dealt in a node itself, and not between nodes- it is assumed that if a node is running correct CPN software that can interpret our messages, it is a trusted node. By checking PIDs against device IDs, we can prevent impersonation of devices, but in reality if a malicious entity has access to the hardware then it is not possibly to stop an attack on the system.

The conclusion we can take away from this section is while all required objectives were met, as well as the majority of the desired ones (except node swapping), this is such an extensive topic that there could be many different ways that it could keep building into a more complete system.

6.4.3 Scalability

Scalability was not really explored during this project. While this was set as a main goal in the Interim Report, when the experimentation phase came around it was realised that stream-lining the testing process was not going to be possible. Experimenting on 3 machines was already time-consuming due to the sheer amount of data, so a choice was made to limit the number of machines so that the data could be studied in depth. The alternative would have been to shallowly analyse a greater amount of nodes, and not be able to detect certain features which were faulty.

From the study that was done on a limited amount of nodes, as well as how the number of forwarded requests scales with $O(\log n)$, we can venture that the system should in theory be scalable. In order to be able to properly test scalability in the future, the concept that was planned was to use OpenNebula in order to horizontally scale the system by adding more and more machines. It would be set up to automatically download the CPN software, in order to make the process simpler.

Another addition that would be needed in order to carry this out would be to develop better performance measuring tools. A combination of logging and reading from multiple files was needed to gather data. If there were tools that could help visualise the system, this would be very helpful in studying the system as the number of nodes increases.

So while we have no concrete data that proves that the system is scalable, it seems promising according to the data gathered until this point.

Chapter 7: User Guide

This section details the set of steps that are necessary in order to install and use the CPN module, as well as suggesting ways of debugging the system for anyone that would later pick up this work and attempt to continue it. Finally, a way to generate graphs that track system performance will be provided.

7.1 System requirements

This project compiles and executes correctly on a machine with the following characteristics:

Operating system	Ubuntu
Operating system distribution	Lucid
Operating system version	10.04.4
Kernel version	2.6.32-38-generic
GCC version	4.4.3

Table 15- System requirements overview

Newer versions of the kernel may use updated versions of many of the system calls used by the kernel module, so it is recommended that the mentioned versions are used. Further work may be done at some later point in time to make this compatible with newer Ubuntu versions.

The user needs to have root permissions in order to start CPN, but devices may be generated with user permissions.

7.2 Loading the kernel module

The kernel module's source code is self-contained in a folder named `'moduledevice'`, and can be built and loaded with the following commands:

```
make && make install
/etc/init.d/cpn restart
```

There needs to exist a file `'/etc/cpn.conf'` which defines which CPN address the host shall take when it registers, as well as which network devices to register for use with CPN. It will normally contain something similar to:

```
CPN_ADDRESS="15.0.0.1"
CPN_INTERFACES="eth2"
```

7.3 Registering devices

In order to create a device, the source code is self-contained in a folder named 'util', and can be built the same way as with the kernel module.

Device usage is as follows:

```
Usage: device -dt DEVICE_TYPE(0/1) [-v] [-l LOOPNUM] [-p PROTO] [-ttl
DEVICE_LIFE] [-ty PRIORITY] [-mr MEASURING_RATE]
```

An example command on how to generate each type of device follows:

```
device -v -p 0 -dt 0 -l 20 -ttl 30 -ty 0 -mr 5  # Measuring device
device -v -p 0 -dt 1 -l 20 -ttl 300000 -ty 0 -mr 5    # Acting device
```

7.4 Debugging

In order to follow the execution flow or for debugging purposes, the module prints debugging messages which can be accessed via the following command:

```
tailf /var/log/syslog;
```

CPN's current status and parameters can be accessed from the '/proc/cpn' folder, where the available /proc files are detailed in Table 16.

/proc/cpn/controller	Shows number of every type of device per neighbour.
/proc/cpn/dprr	Shows all known routes.
/proc/cpn/jobqueue	Shows queue of jobs left to process.
/proc/cpn/neighbour	Shows list of neighbours.
/proc/cpn/responsive_mbox	Shows responsiveness data for all registered acting devices.
/proc/cpn/sp_percen	Parameter which introduces randomness to Smart Packets.
/proc/cpn/stats	Shows miscellaneous stats.
/proc/cpn/device_mbox	Shows all recorded measurements for every measuring devices.
/proc/cpn/interface	Shows which net devices are registered.
/proc/cpn/mbox	Shows QoS measurements for each known destination and neighbour combination.
/proc/cpn/queue	Shows the list of messages which have yet to process while a suitable path is found.
/proc/cpn/rnn	Shows neural network weights for each destination and neighbour combination.
/proc/cpn/sp_random_percen	Parameter which introduces randomness to Smart Packets.

Table 16- Available /proc files

7.5 Generating graphs

In order to generate graphs that show system performance, the following two scripts are necessary:

stats_save.sh plot_stats.py

They can be found under the 'util' folder.

The plotting is carried out in Python using the module **matplotlib**, and the following versions were used:

Python version	2.6.5
Matplotlib version	0.99.1.1

Table 17- Python requirements for graph plotting

The code should in principle be compatible with newer versions of both.

The `stats_save.sh` script records the contents of the `/proc/cpn/stats` file every 2 seconds, saving the results in a 'measurements' folder at the same level as 'util' and 'moduledevice'. Once we are done recording data, the script should be killed, as it will run forever otherwise.

In order to plot the results, `plot_stats.py` should be called with no arguments. The path to the measurements folder, as well as other session data, can be customised by modifying the global constants in the file. This will then generate, in the 'measurements' folder, PDF and PNG versions of the graphs that accompany the stats recorded.

Additional files required for management of the system can be found in 'cpn3/include' and 'cpn3/etc'.

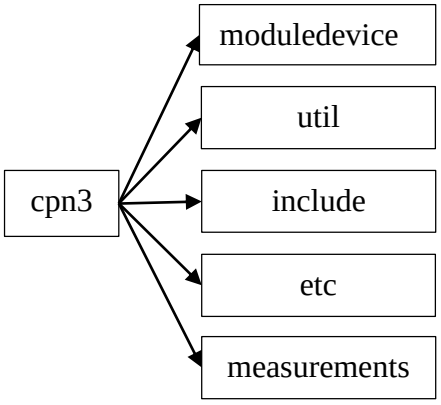


Figure 22- CPN folder hierarchy

Conclusions

This project successfully implements a system to allocate and monitor devices in a non-ideal network of nodes. Due to the amount of directions this project could be taken, it had to be greatly limited in scope since early in development, and further limitations have arisen during the life of the project.

It touches many areas which are currently relevant, such as neural networks and cloud systems. While the software it is built on limits its portability, porting it to other platforms would probably make the design approaches not viable. Many of the programming features used have been specially designed for best performance, and translating the project into another language such as Python would probably make it too slow to be able to make decisions in real time.

The requirements set upon at the beginning were mostly met, and whenever they could not be achieved reasons were given for this. The success of the implementation has been discussed, and points of further work have been proposed so far. These were many, which speaks for the open-endedness of this project.

The one thing to take away from this project is that it proves that a general system of this type is feasible. By building additional features on top of it, such as better diagnostic tools, or better device customisation, this could in the future become a product that could be used to make daily life easier and better for all of us. For this purpose, a user guide was included to help future researchers continue this subject.

Works cited

Alsamhi, S. H. & Rajput, N. S., 2015. An Intelligent Hand-off Algorithm to Enhance Quality of Service in High Altitude Platforms Using Neural Network. *Springer Science+Business Media New York.*

Amazon, n.d. *Amazon Web Services.* [Online]
Available at: http://aws.amazon.com/
[Accessed 23 01 2016].

Arutyunov, V. V., 2012. Cloud Computing: Its History of Development, Modern State, and Future Considerations. *Scientific and Technical Information Processing,* 39(3), pp. 173-178.

Benjapolakul, W. & Rangsihiranrat, T., 2000. Aggregate Bandwidth Allocation of Heterogeneous Sources in ATM Networks with Guaranteed Quality of Service Using a Well-Trained Neural Network. *IEEE,* pp. 348-351.

Dogman, A., Saatchi, R. & Al-Khayatt, S., 2012. Evaluation of Computer Network Quality of Service Using Neural Networks. *IEEE Symposium on Business, Engineering and Industrial Applications,* pp. 217-222.

Durao, F., Carvalho, J. F. S., Fonseka, A. & Garcia, V. C., 2014. A systematic review on cloud computing. *J Supercomput,* Volume 68, pp. 1321-1346.

Gajdos, M., 2014. *Tenus - Golang Powered Linux Networking,* s.l.: s.n.

GELENBE, E., 1989. Random neural networks with negative and positive signals and product form solution. *Neural Computation 1,* pp. 502-510.

GELENBE, E., LENT, R. & NUNEZ, A., 2004. Self-Aware Networks and QoS. *PROCEEDINGS OF THE IEEE,* 92(9), pp. 1478-1489.

Girma, D. & Lazaro, O., 2000. Quality of Service and Grade of Service Optimisation with Distributed DCA Schemes Based on Hopfield Neural Network Algorithm. *IEEE VTC,* pp. 1142-1147.

Google, n.d. *Google Trends.* [Online]
Available at: https://www.google.com/trends/explore#q=cloud
[Accessed 23 01 2016].

Neira-Ayuso, P., Gasca, R. M. & Lefevre, L., 2010. Communicating between the kernel and user-space in Linux using Netlink sockets. *Softw. Pract. Exper.,* Volume 40, pp. 797-810.

OpenNebula Systems, n.d. *OpenNebula.* [Online]
Available at: http://docs.opennebula.org/4.14/index.html
[Accessed 24 01 2016].

Sakellari, G., 2011. Performance evaluation of the Cognitive Packet Network in the presence of network worms. *Performance evaluation 68,* pp. 927-937.

Salzman, P. J., Burian, M. & Pomerantz, O., 2009. *The Linux Kernel Module Programming Guide.* s.l.:CreateSpace Independent Publishing Platform.

Sarajedini, A. & Chau, P. M., 1996. QUALITY OF SERVICE PREDICTION USING NEURAL NETWORKS. *IEEE,* pp. 567-570.

The Linux Information Project, 2005. *The Linux Information Project-- The Kernel.* [Online]
Available at: http://www.linfo.org/kernel.html
[Accessed 31 May 2016].

Usman, M. J. et al., 2014. A Framework for Realizing Universal Standardization for Internet of Things. *Journal of Industrial and Intelligent Information,* 2(2), pp. 147-153.

Wang, L. & Gelenbe, E., 2015. *Adaptive Dispatching of Tasks in the Cloud.* [Online]
Available at: http://san.ee.ic.ac.uk/publications/TaskallocationIEEE2015.pdf
[Accessed 23 January 2016].

Wang, Z., Jiang, N. & Zhou, P., 2015. Quality Model of Maintenance Service for Cloud Computing. *IEEE 17th International Conference on High Performance Computing and Communications,* pp. 1460-1465.

Appendices

Appendix A – Statistics files

An example of what a statistic file looks like follows:

```
cpn_stats.CpnSmartPktIn = 25     // Number of smart packets received
cpn_stats.CpnSmartPktOut = 5     // Number of smart packets sent
cpn_stats.CpnSmartDrop = 0       // Number of smart packets dropped
cpn_stats.CpnSmartLocalDelivered = 25   // Number of smart packets kept
cpn_stats.CpnDumbPktIn = 24      // Number of dumb packets received
cpn_stats.CpnJobsPktIn = 0       // Legacy stat
cpn_stats.CpnJobsPktTest = 0     // Legacy stat
cpn_stats.CpnJobsIn = 0          // Legacy stat
cpn_stats.CpnDumbPktOut = 5      // Number of dumb packets sent
cpn_stats.CpnDumbDrop = 0        // Number of dumb packets dropped
cpn_stats.CpnDumbLocalDelivered = 24    // Number of dumb packets kept
cpn_stats.CpnSAckPktIn = 5       // Number of smart ACK packets received
cpn_stats.CpnSAckPktOut = 25     // Number of smart ACK packets sent
cpn_stats.CpnSAckDrop = 5        // Number of smart ACK packets dropped
cpn_stats.CpnSAckLocalDelivered = 5 // Number of smart ACK packets kept
cpn_stats.CpnDAckPktIn = 5       // Number of dumb ACK packets received
cpn_stats.CpnDAckPktOut = 24     // Number of dumb ACK packets sent
cpn_stats.CpnDAckDrop = 0        // Number of dumb ACK packets dropped
cpn_stats.CpnDAckLocalDelivered = 5 // Number of dumb ACK packets kept
cpn_stats.CpnDumbBytesSent = 402     // Number of dumb bytes sent
cpn_stats.CpnDumbBytesRecv = 1496    // Number of dumb bytes received
cpn_stats.CpnDAckBytesSent = 1832    // Number of dumb ACK bytes sent
cpn_stats.CpnDAckBytesRecv = 332     // Number of dumb ACK bytes received
cpn_stats.CpnSmartBytesSent = 770    // Number of smart bytes sent
cpn_stats.CpnSmartBytesRecv = 3500   // Number of smart bytes received
cpn_stats.CpnSAckBytesSent = 3850    // Number of smart ACK bytes sent
cpn_stats.CpnSAckBytesRecv = 700     // Number of smart ACK bytes received
cpn_stats.CpnReqBytesRecv = 2064     // Number of register request bytes received
cpn_stats.CpnRackBytesSent = 32      // Number of register ACK request bytes sent
cpn_stats.CpnMesBytesRecv = 0        // Number of measurement bytes received
cpn_stats.CpnMackBytesSent = 0       // Number of measurement ACK bytes sent
cpn_stats.CpnAreqBytesSent = 912     // Number of action request bytes sent
cpn_stats.CpnAackBytesRecv = 39216   // Number of action request ACK bytes
received
cpn_stats.CpnDreqBytesRecv = 0       // Number of de-register request bytes
received
cpn_stats.CpnDroppedNetlinkMessage = 0   // Number of dropped netlink messages
cpn_stats.CpnMboxCompression = 0         // Number of compressions on device
measurement boxes
cpn_stats.CpnIsOverloaded = 0        // Is the system currently overloaded
cpn_stats.CpnReceivedResponsivenessMeasurements = 19    // Number of
responsiveness measurements gathered
cpn_stats.CpnReceivedMeasurements = 0   // Number of measurements received from
measuring devices
cpn_stats_qos.CpnAvgDelay=2.00000ms      // Average delay
cpn_stats_qos.CpnAvgJitter=0.00000ms     // Average jitter
cpn_stats_qos.CpnMeanDelay=0.00000ms     // Legacy stat
cpn_stats_qos.CpnMeanJitter=0.00000ms    // Legacy stat
cpn_stats_qos.CpnMdevDelay=0.00000ms     // Legacy stat
cpn_stats_qos.CpnMdevJitter=0.00000ms    // Legacy stat
cpn_stats_qos.CpnSumDelay=0.00000ms      // Total delay
cpn_stats_qos.CpnSumJitter=7.89996ms     // Total jitter
cpn_stats_qos.CpnSumDelaySquare=0.00000ms // Total squared delay
```

```
cpn_stats_qos.CpnSumJitterSquare=0.00000ms  // Total squared jitter
cpn_stats_qos.CpnDumbRecv=0.00000ms     // Legacy stat
cpn_stats_qos.CpnDumbLoss=0.00000ms     // Legacy stat
cpn_stats_qos.CpnDumbPktOut_V=0.00000ms     // Legacy stat
cpn_stats_qos.CpnDAckPktIn_V=0.00000ms  // Legacy stat
cpn_stats_qos.CpnDumbLoss_V=0.00000ms   // Legacy stat
cpn_stats_qos.CpnMeanJitter_V=0.00000ms // Mean jitter in power QoS
cpn_stats_qos.CpnMeanDelay_V=0.00000ms  // Mean delay in power QoS
cpn_stats_qos.CpnSumDelay_V=0.00000ms   // Total delay in power QoS
cpn_stats_qos.CpnSumJitter_V=0.00000ms  // Total jitter in power QoS
cpn_stats_qos.CpnAvgDelay_V=0.00000ms   // Average delay in power QoS
cpn_stats_qos.CpnAvgJitter_V=0.00000ms  // Average jitter in power QoS
cpn_stats_qos.CpnDumbLocalPktOut=0.00000ms // Legacy stat
cpn_stats_job.CpnJobsRecv = 5   // Number of jobs received
cpn_stats_job.CpnCreatedRequests = 5    // Number of requests created
cpn_stats_job.CpnFetchedJobs = 24       // Number of jobs fetched from the job
queue
cpn_stats_job.CpnRedirectedJobs = 5     // Number of jobs redirected to other
nodes
cpn_stats_job.CpnRetriedJobs = 0        // Number of jobs retried
cpn_stats_job.CpnAssignedJobs = 19      // Number of jobs kept
cpn_stats_job.CpnDroppedJobs = 0        // Number of jobs dropped
cpn_stats.CpnSumExecutionTime=0.00000ms // Legacy stat
cpn_stats.CpnMeanExecutionTime=0.00000ms    // Legacy stat
cpn_stats.CpnSumResponseTimeatHost=5.89996ms    // Total response time at the
node
cpn_stats.CpnMeanResponseTimeatHost=0.00000ms   // Average response time at the
node
cpn_stats.CpnMeanResponseJitteratHost=0.00000ms // Average response jitter at the
node
cpn_stats.CpnSumResponseTimeatCtrl=0.00000ms    // Legacy stat
cpn_stats.CpnMeanResponseTimeatCtrl=0.00000ms   // Legacy stat
cpn_stats.CpnSumLatency=0.00000ms               // Legacy stat
cpn_stats.CpnMeanLatency=0.00000ms              // Legacy stat
cpn_stats_device.CpnTotalStaleDevices = 0   // Number of stale devices
cpn_stats_device.CpnTotalDeregisteredDevices = 0    // Total de-registered
devices
cpn_stats_device.CpnTotalRegisteredDevices = 1  // Total registered devices
cpn_stats_device.CpnActiveMeasuringDevices = 0  // Number of active measuring
devices
cpn_stats_device.CpnActiveActingDevices = 1     // Number of active acting
devices
cpn_stats_device.CpnActiveDevices = 1           // Total number of active devices
```

All graphs in **Appendix B – Graphs** were formed by plotting how these statistics changed with time.

Appendix B – Graphs

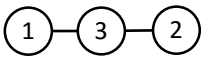
Graphs for some of the tests performed follow.

Test 1

- Connectivity: Three hosts, all connected to all.
- Host 1: One measuring device at 2 requests per second for 30 seconds
- Host 2: One acting device.
- Host 3: One acting device.

Test 2

- Connectivity:

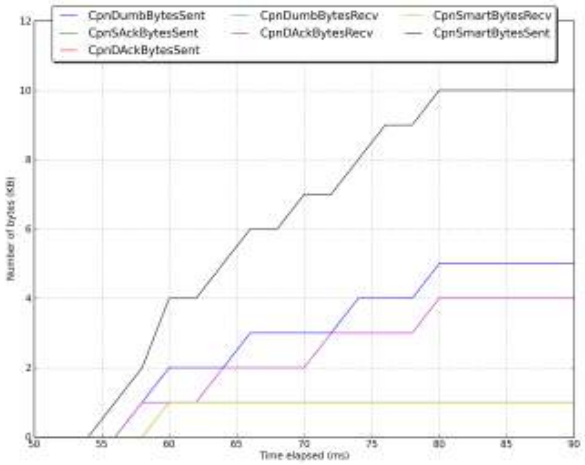


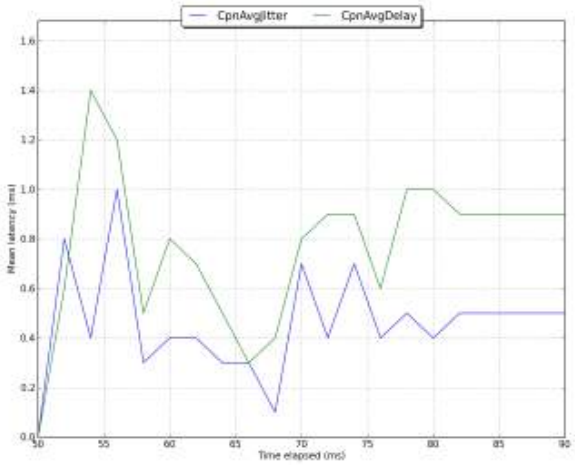
- Host 1: One measuring device at 1000 requests per second for 30 seconds.
- Host 2: One acting device.
- Host 3: One acting device.

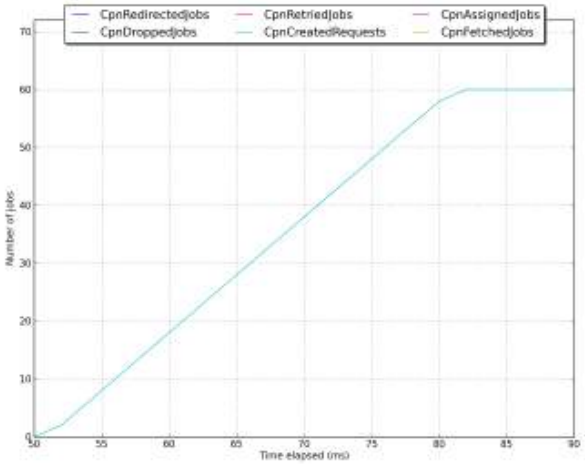
Test 3

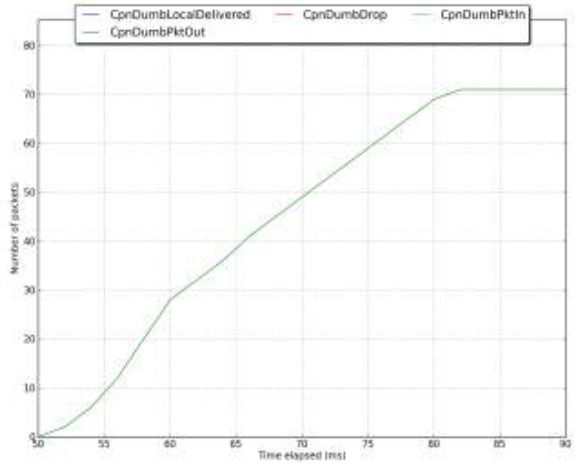
- Connectivity: Three hosts, all connected to all.
- Host 1: One measuring device at 10 requests per second for 30 seconds
- Host 2: One acting device.
- Host 3: One acting device.

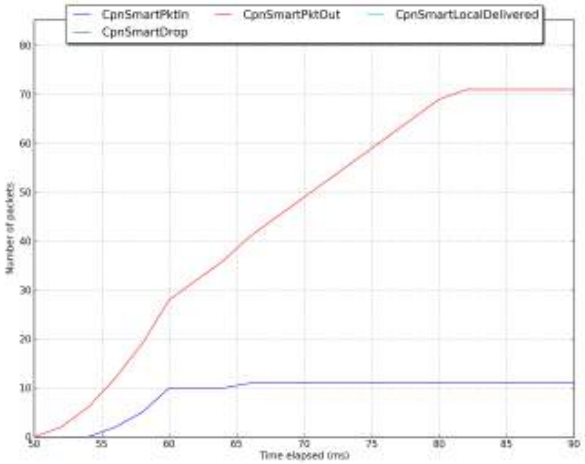
Figure 23- Graphs for test 1, host 1


CpnDumbBytesSent
CpnDumbBytesRecv
CpnSmartBytesRecv
CpnSAckBytesSent
CpnDAckBytesRecv
CpnSmartBytesSent
CpnDAckBytesSent
Number of bytes (KB)
Time elapsed (ms)


CpnAvgJitter
CpnAvgDelay
Mean latency (ms)
Time elapsed (ms)


CpnRedirectedJobs
CpnRetriedJobs
CpnAssignedJobs
CpnDroppedJobs
CpnCreatedRequests
CpnFetchedJobs
Number of jobs
Time elapsed (ms)


CpnDumbLocalDelivered
CpnDumbDrop
CpnDumbPktIn
CpnDumbPktOut
Number of packets
Time elapsed (ms)


CpnSmartPktIn
CpnSmartPktOut
CpnSmartLocalDelivered
CpnSmartDrop
Number of packets
Time elapsed (ms)

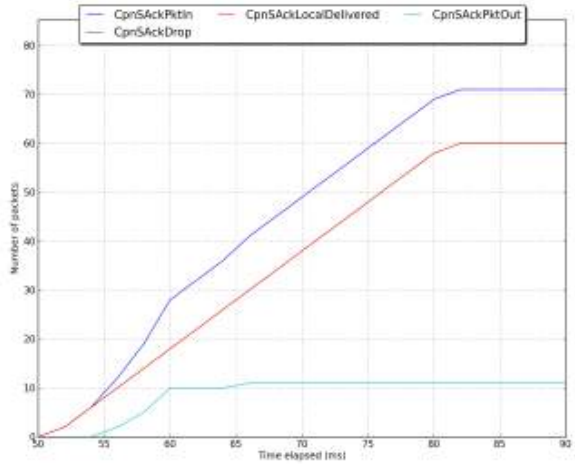
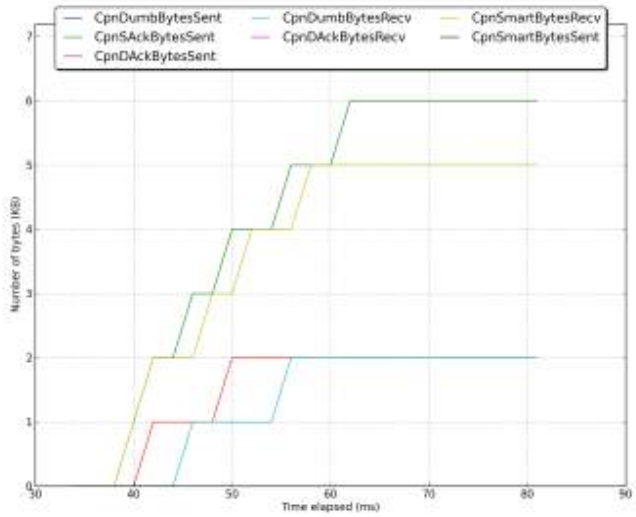
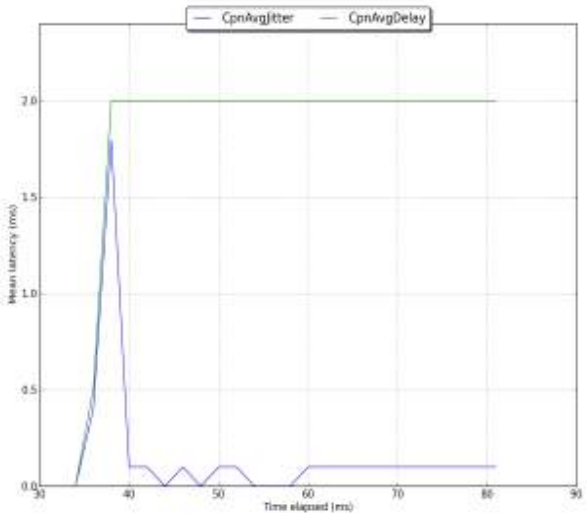
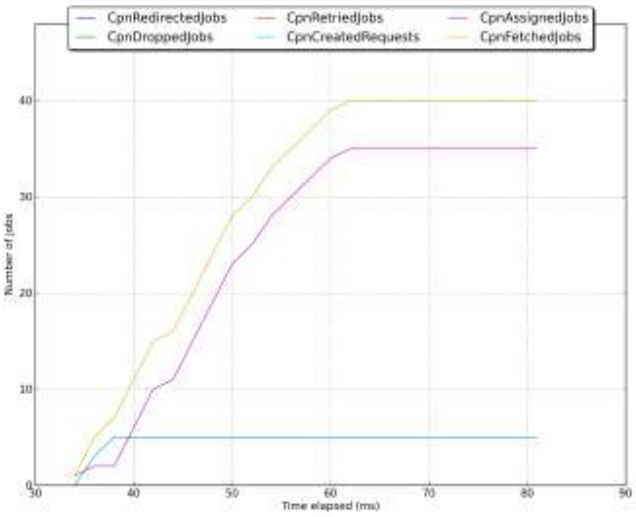
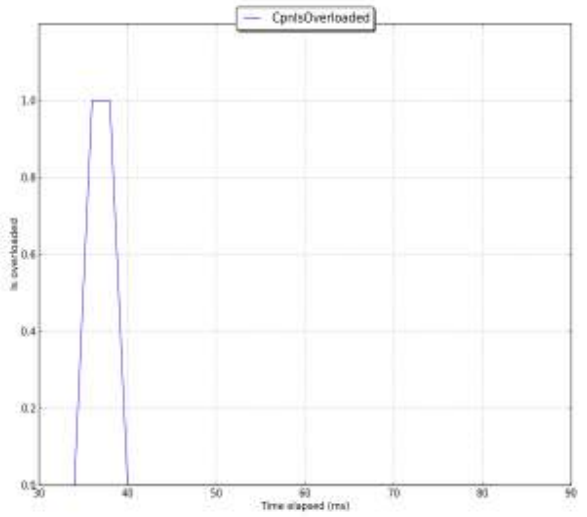

CpnSAckPktIn
CpnSAckLocalDelivered
CpnSAckPktOut
CpnSAckDrop
Number of packets
Time elapsed (ms)

Figure 24- Graphs for test 1, host 2









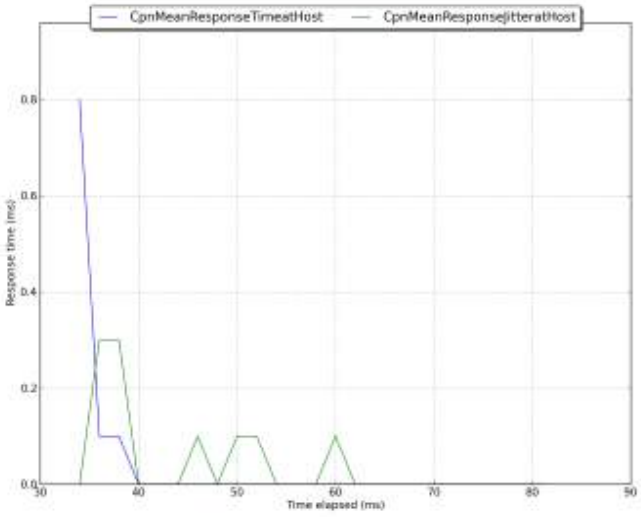
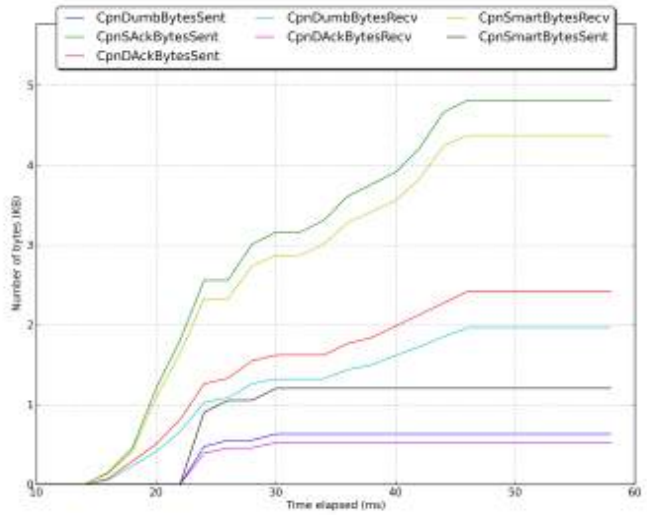
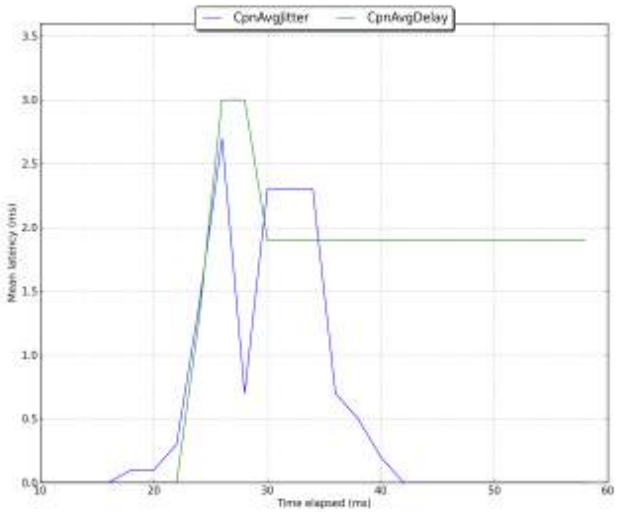
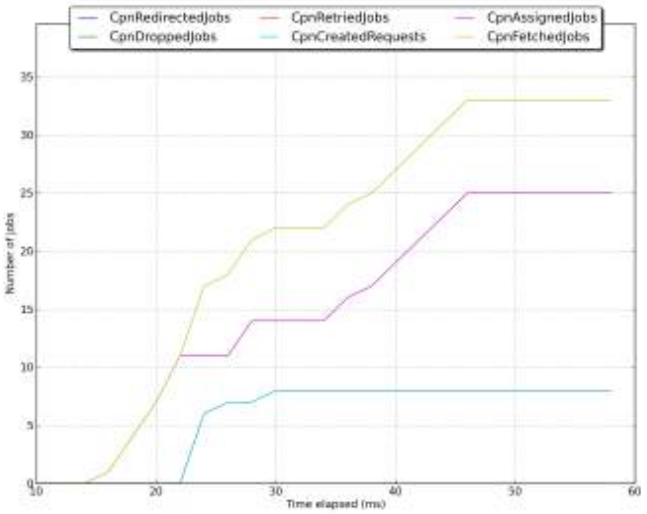
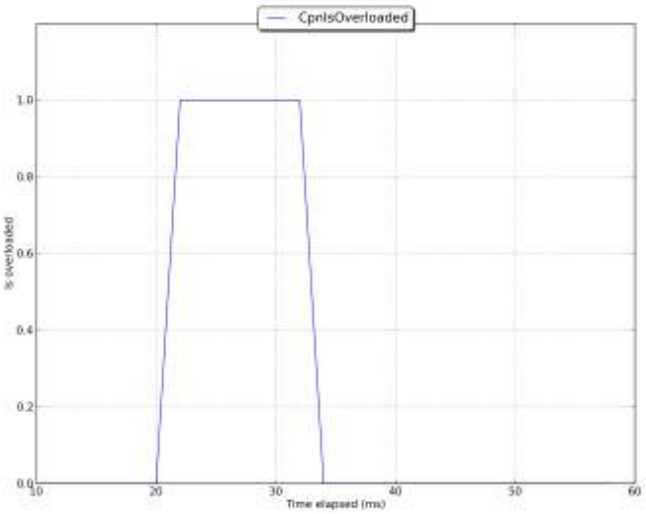


Figure 25- Graphs for test 1, host 3









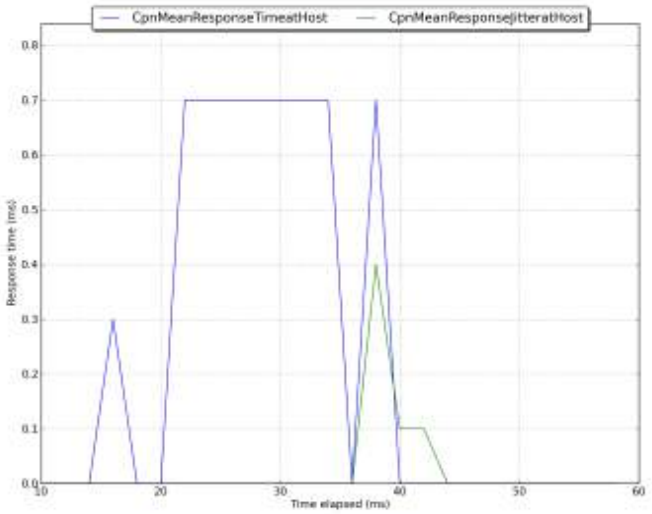
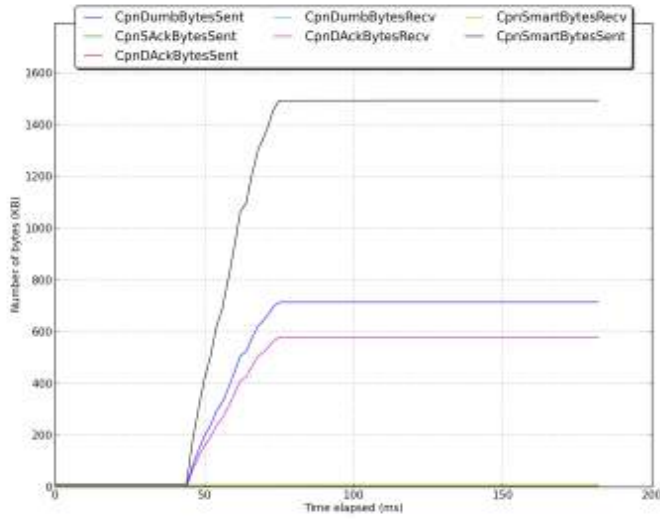
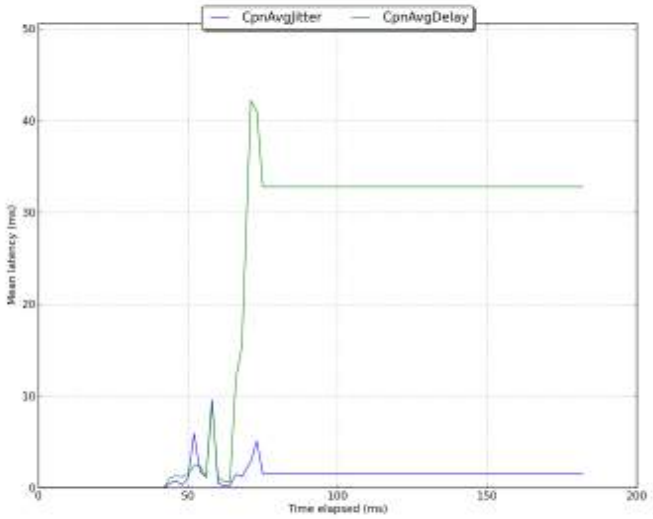
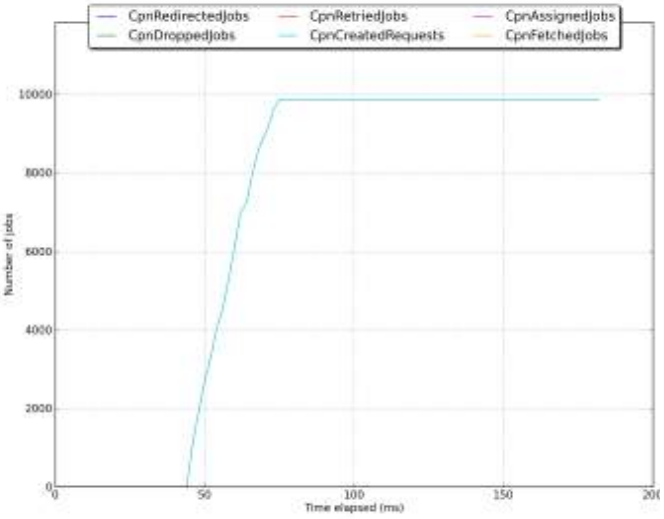
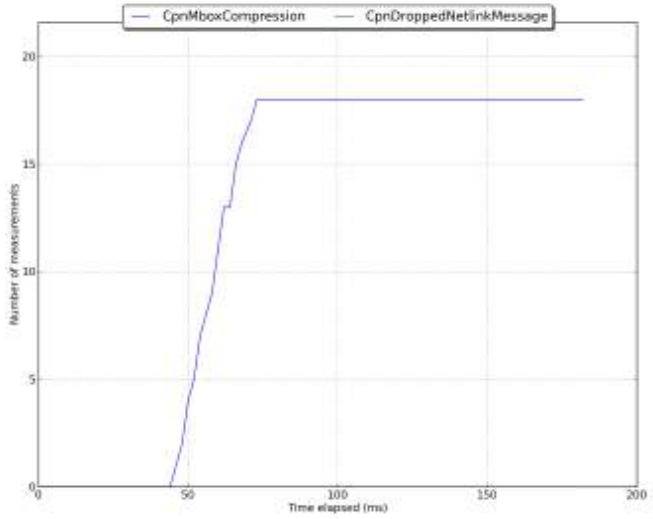


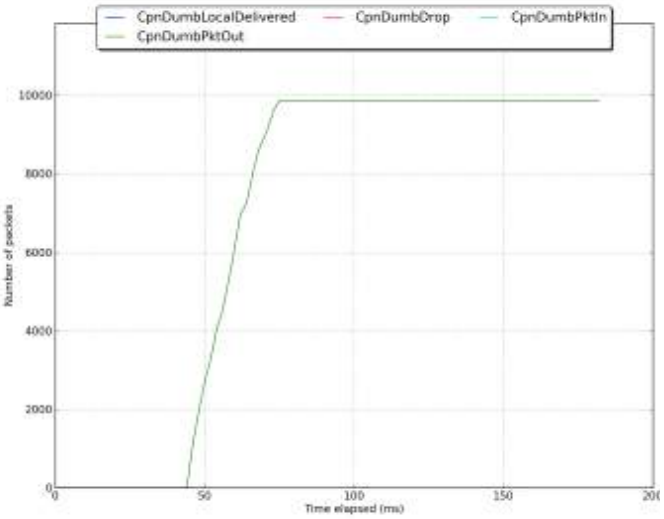
Figure 26- Graphs for test 2, host 1











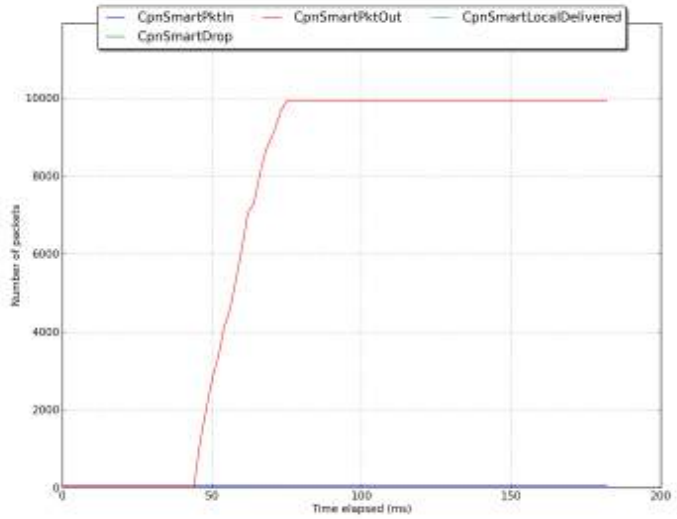
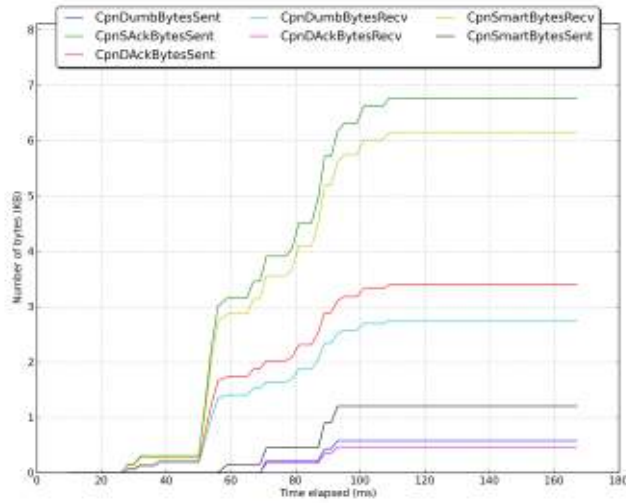
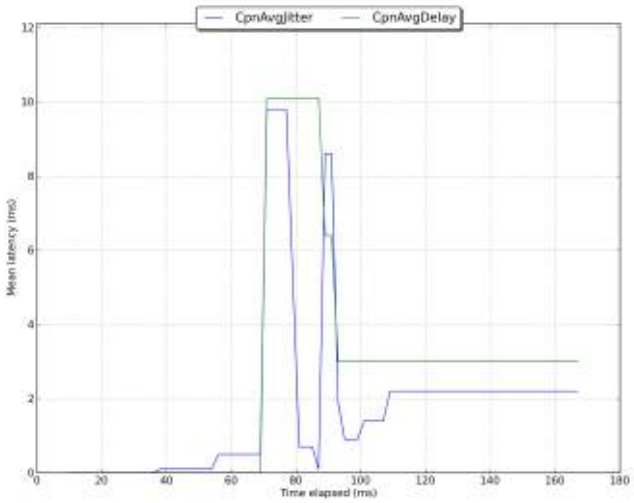
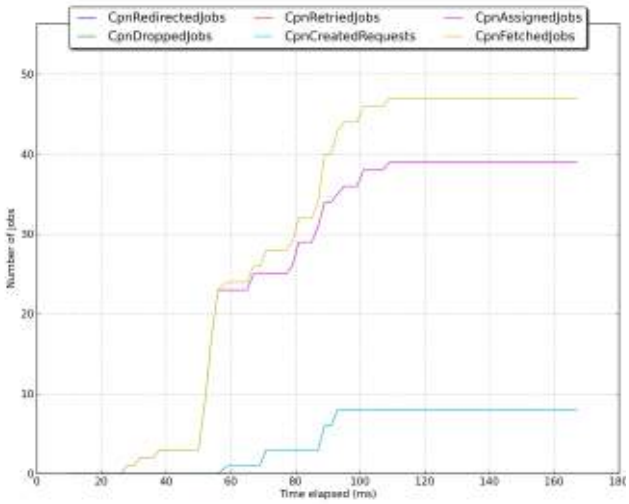
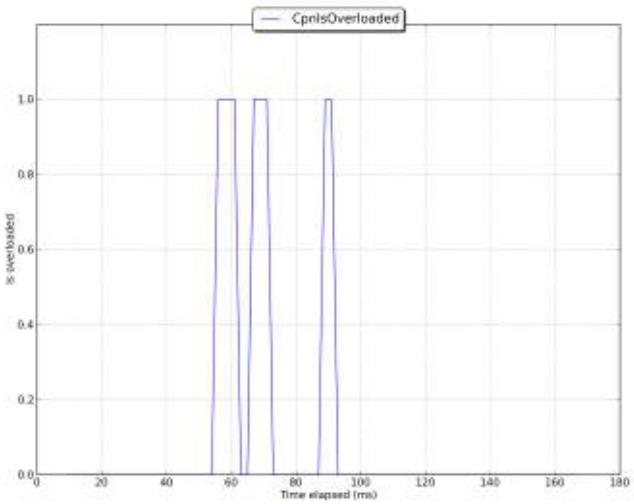


Figure 27- Graphs for test 2, host 2









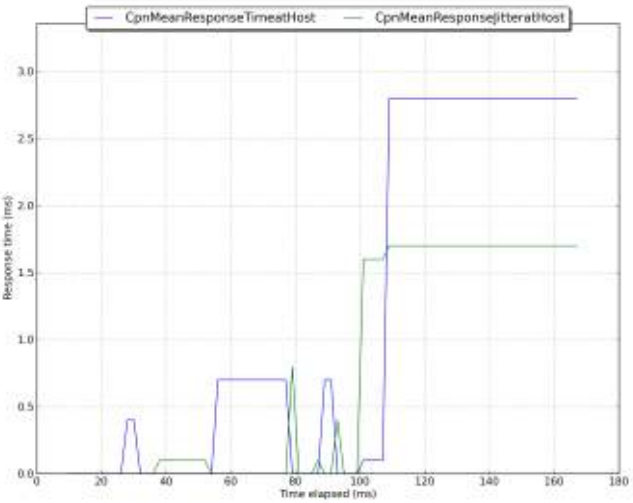
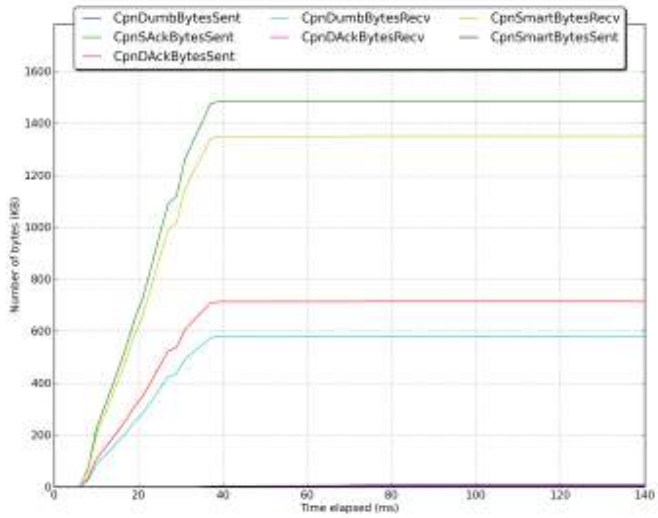
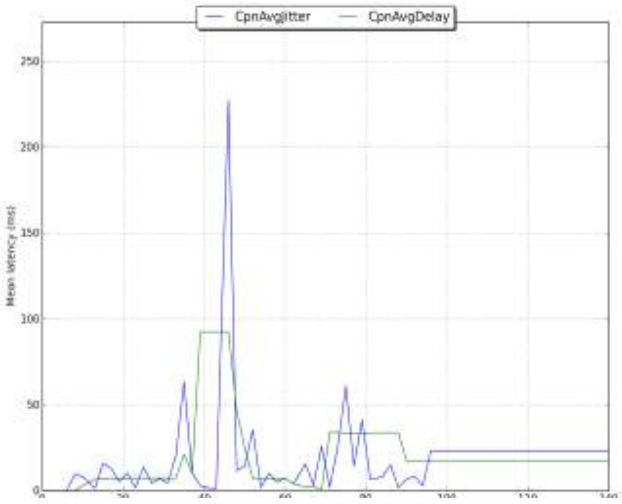
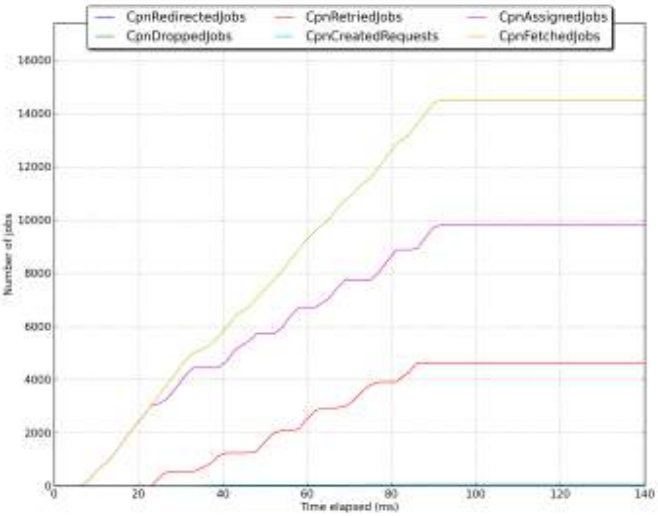
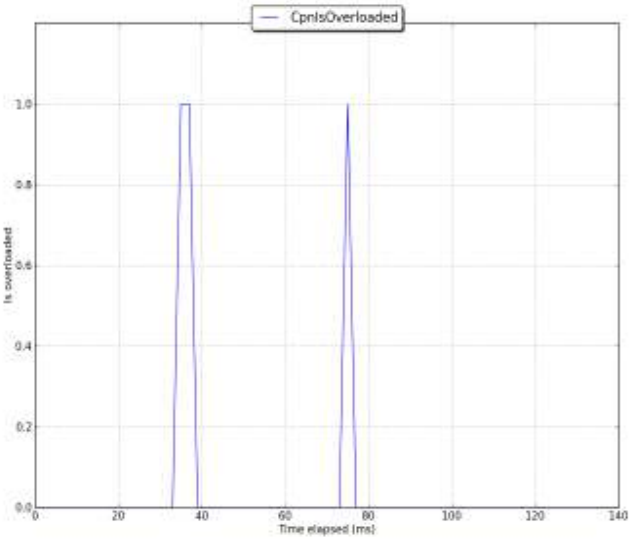


Figure 28- Graphs for test 2, host 3









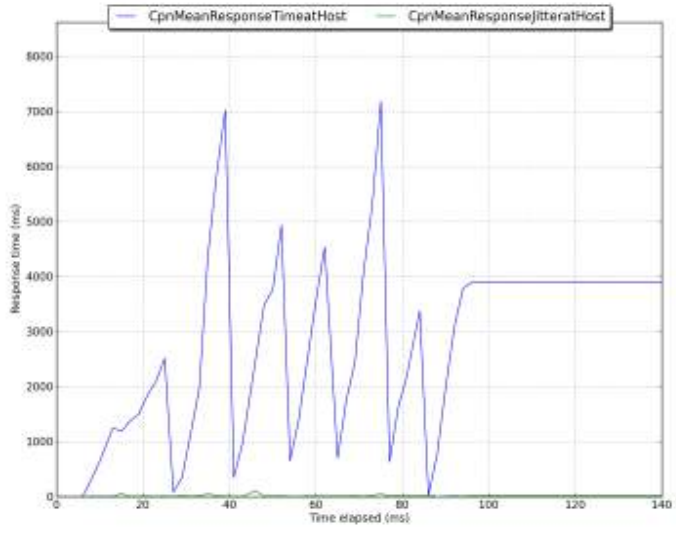
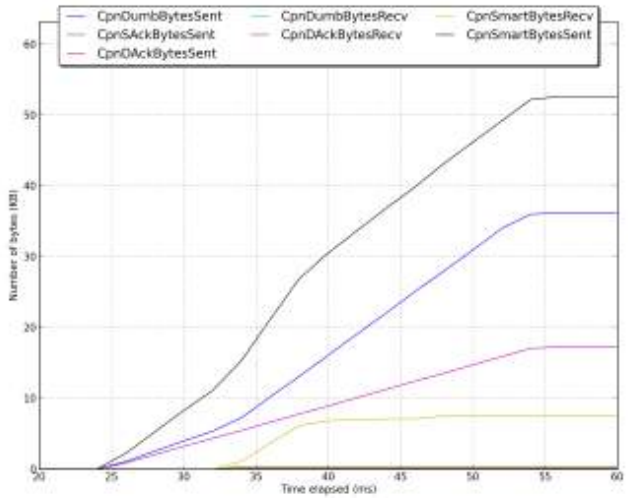
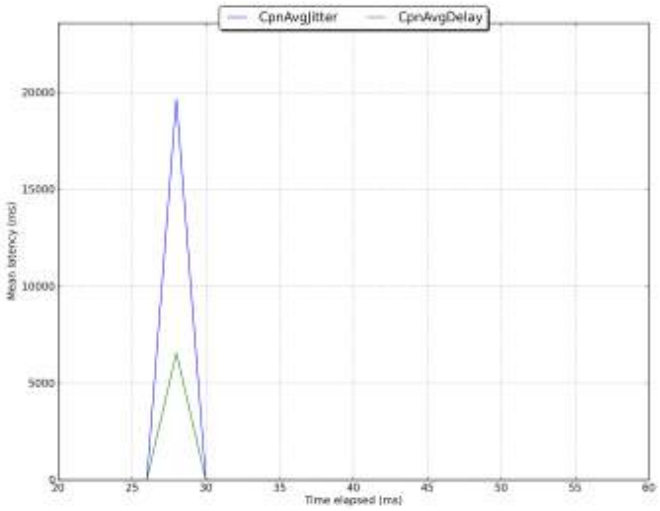
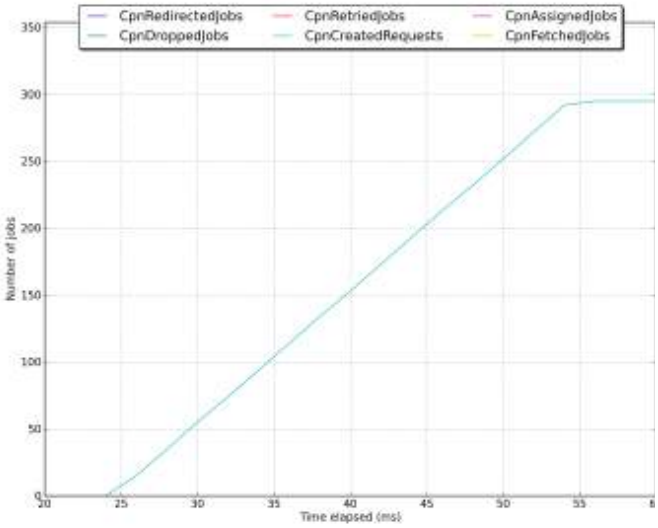
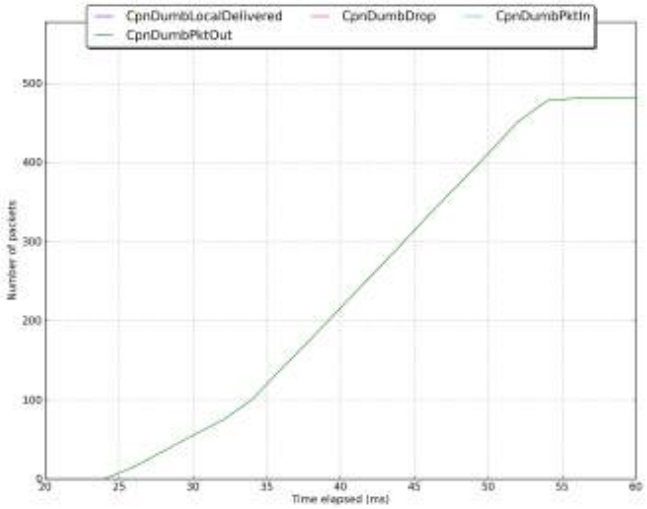


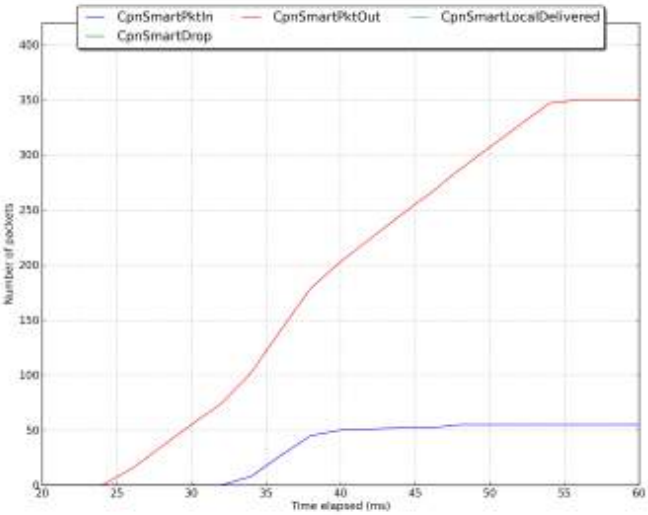
Figure 29- Graphs for test 3, host 1











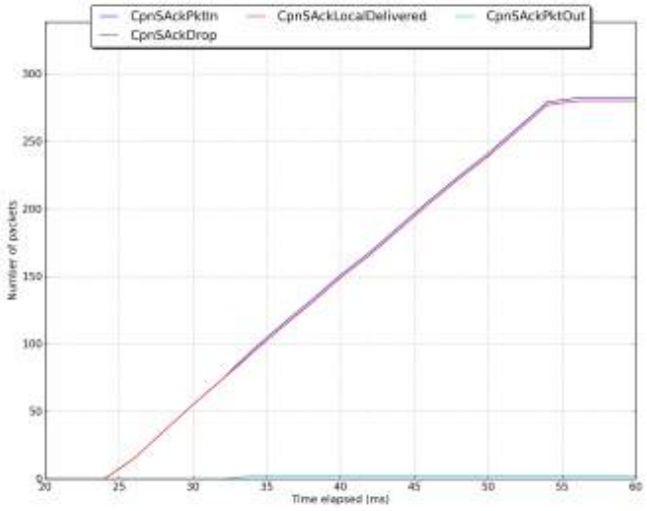
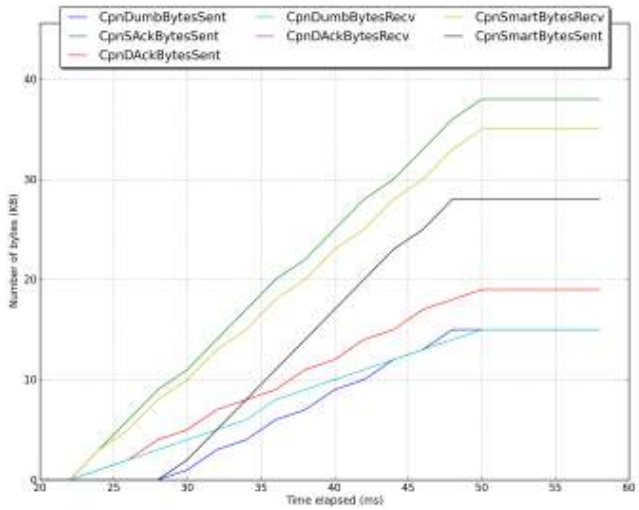
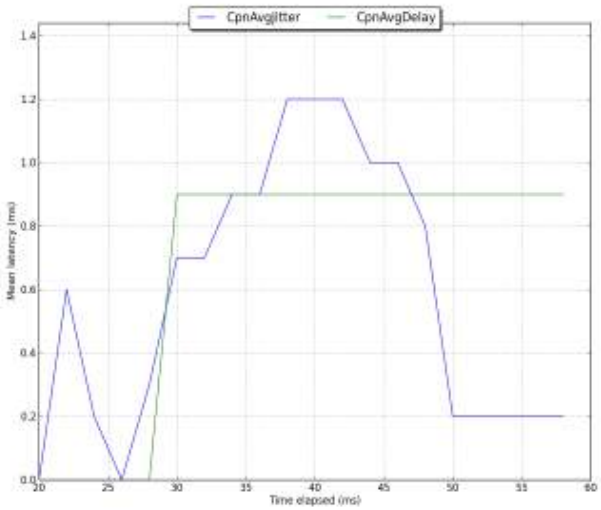
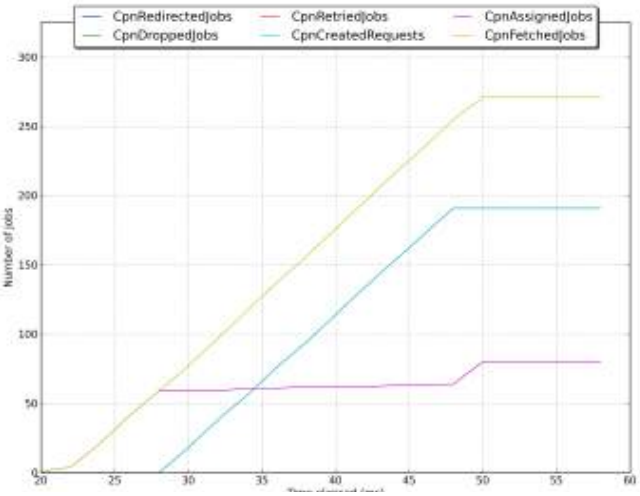
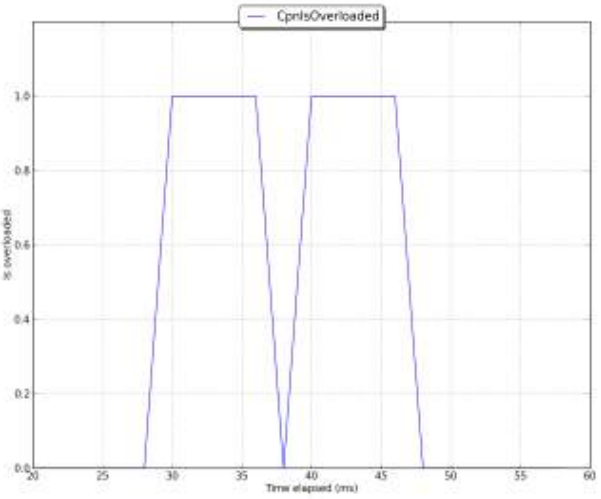


Figure 30- Graphs for test 3, host 2









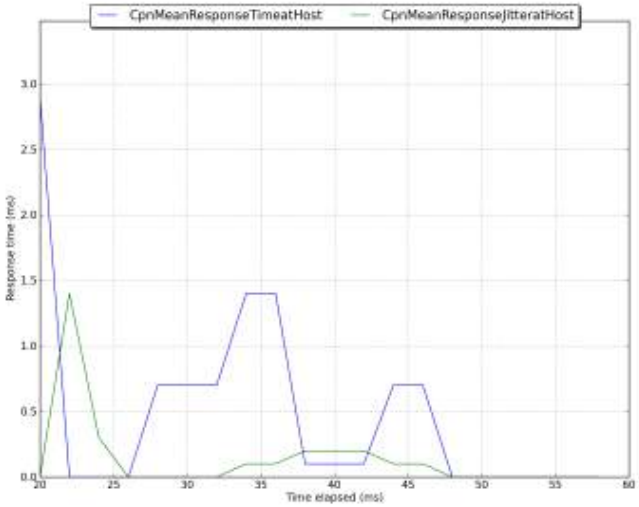
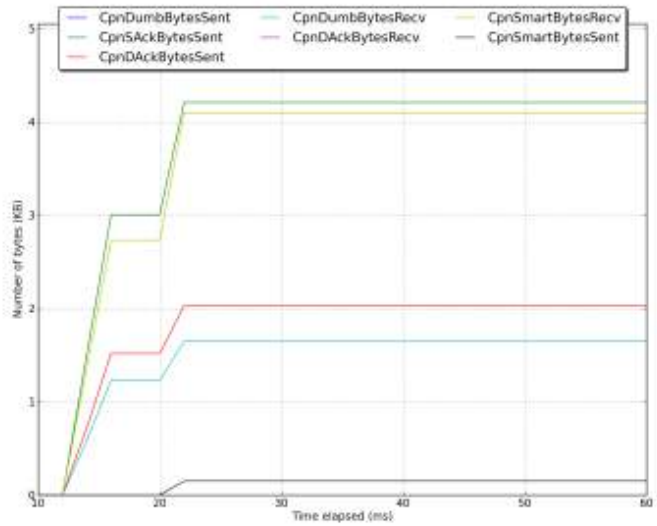
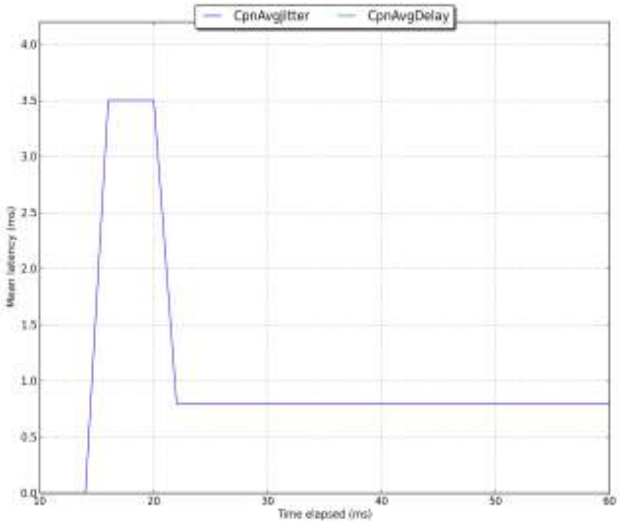
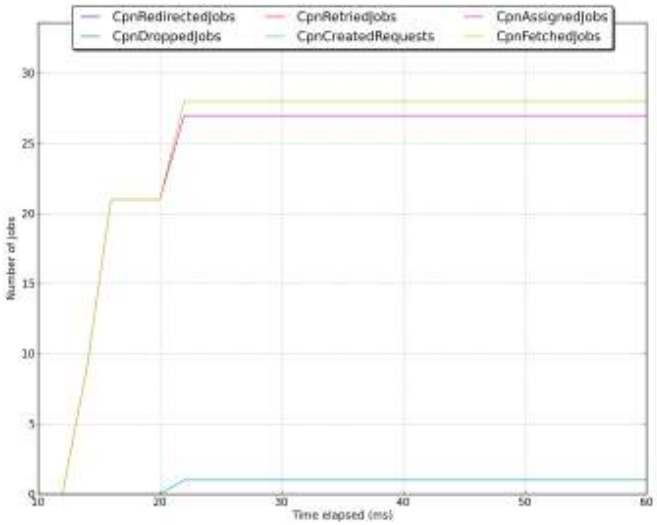
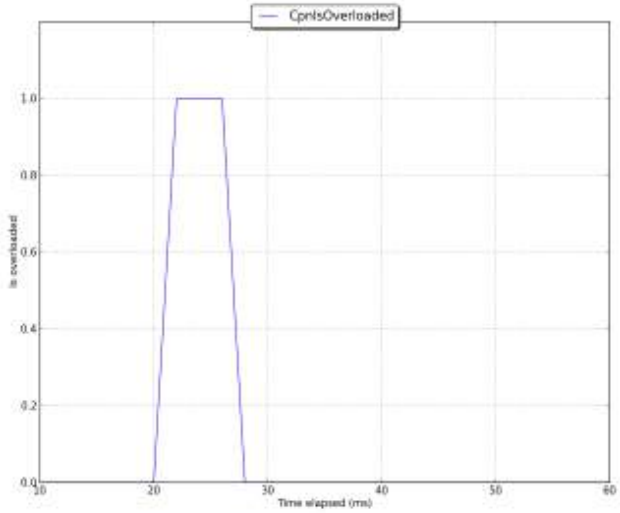


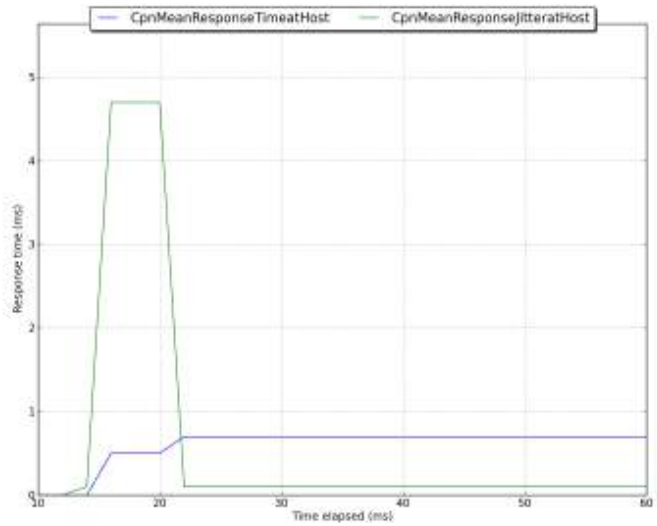
Figure 31- Graphs for test 3, host 3











Appendix C – Diagrams

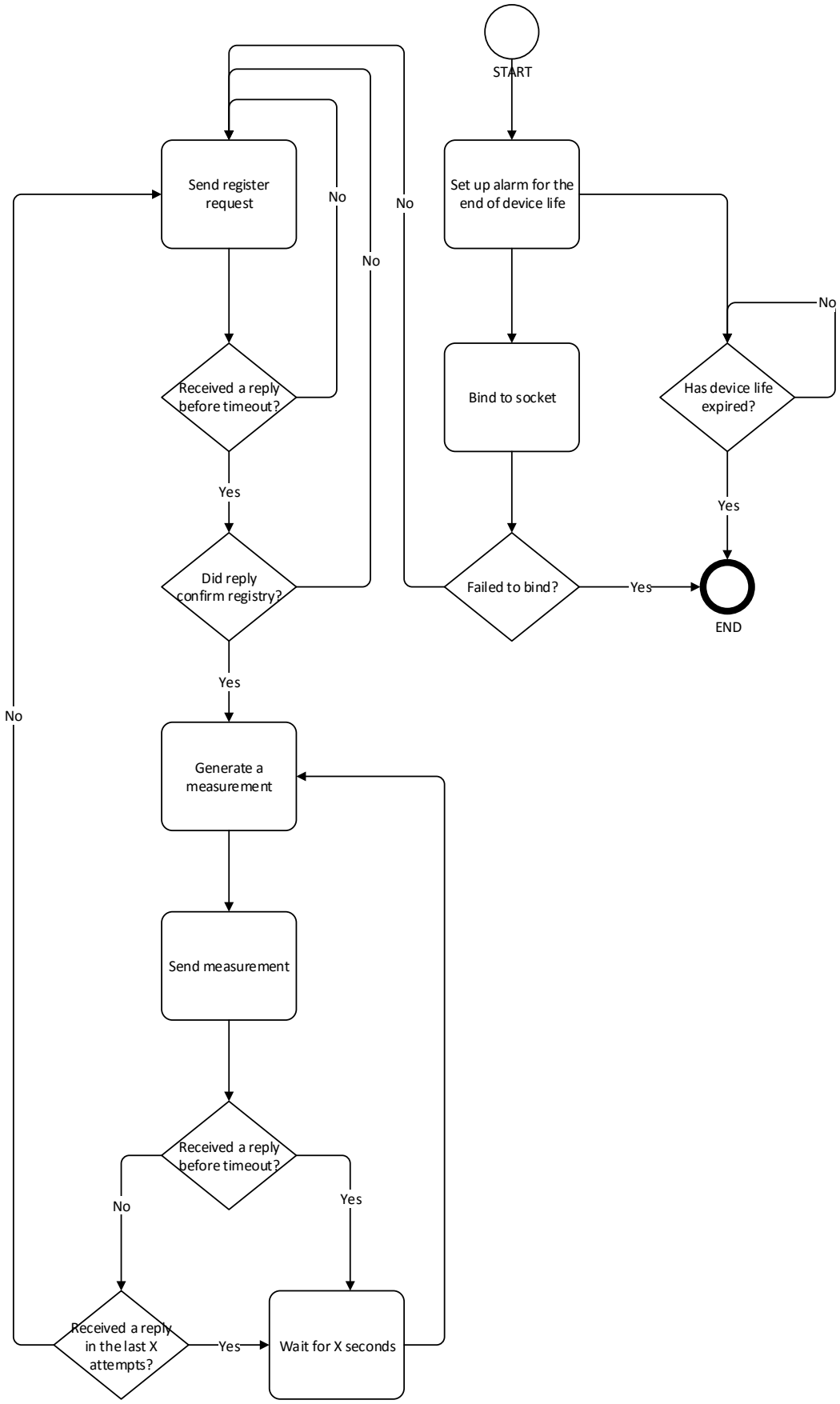


Figure 32- System diagram for a measuring device

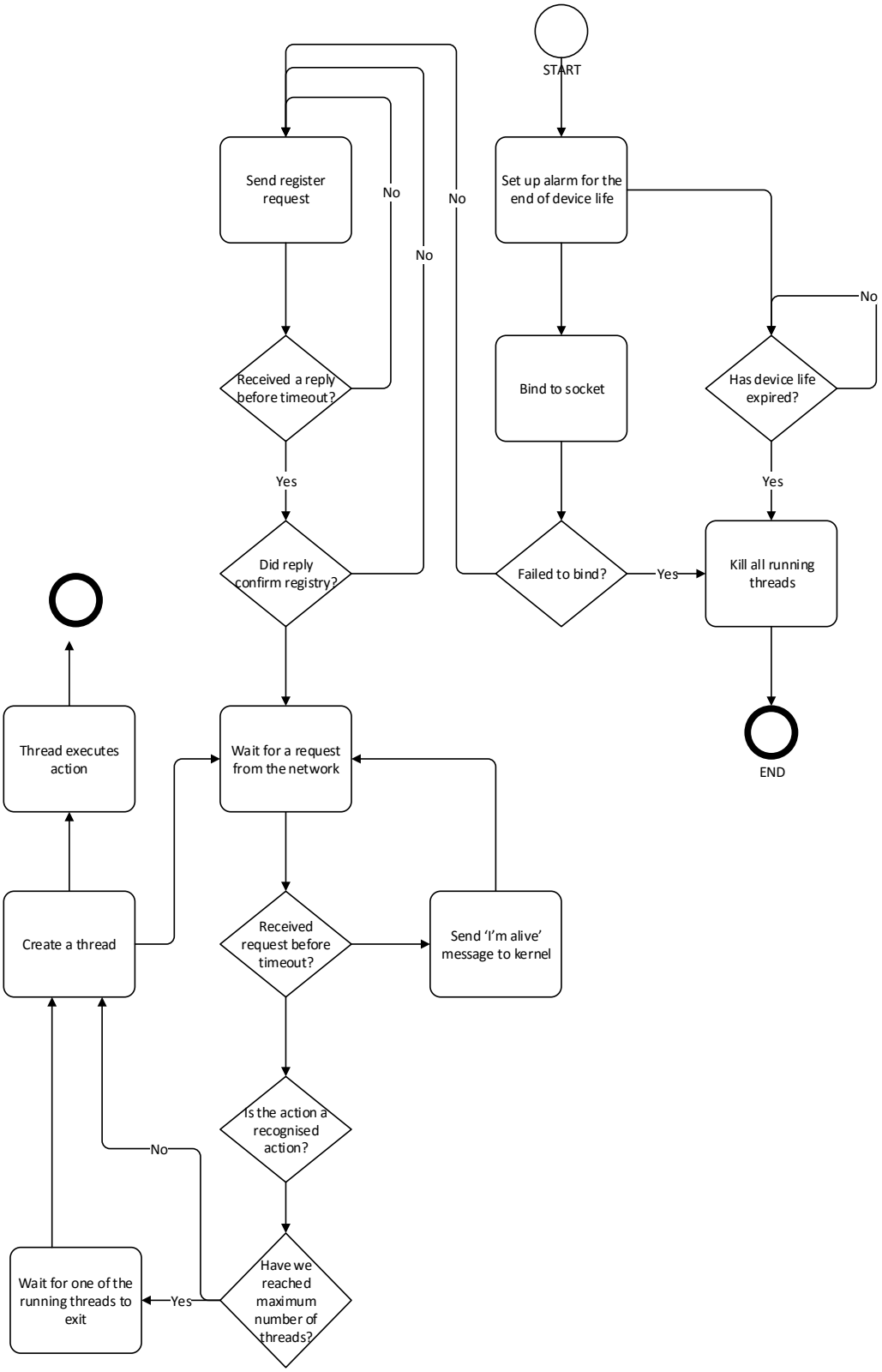
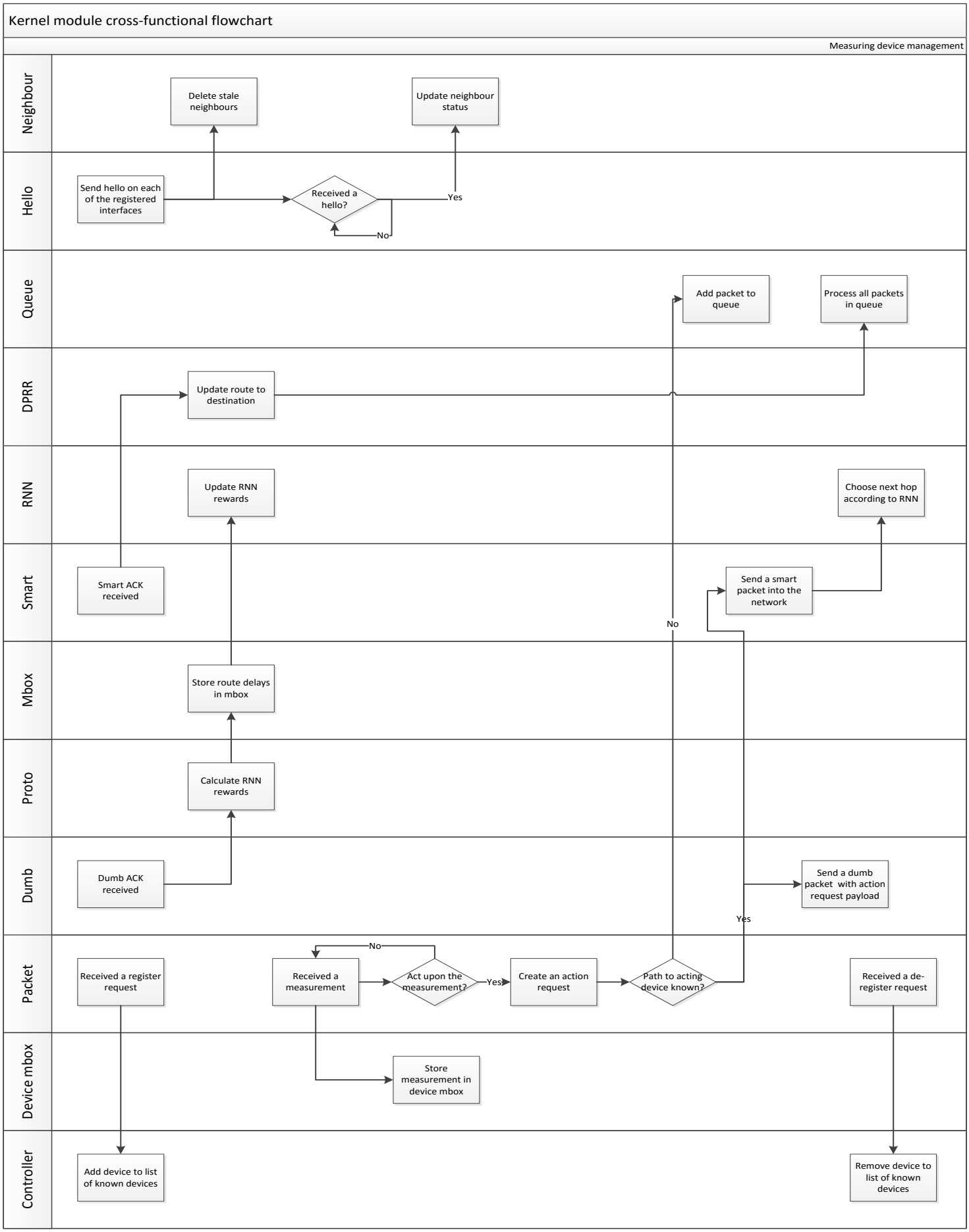
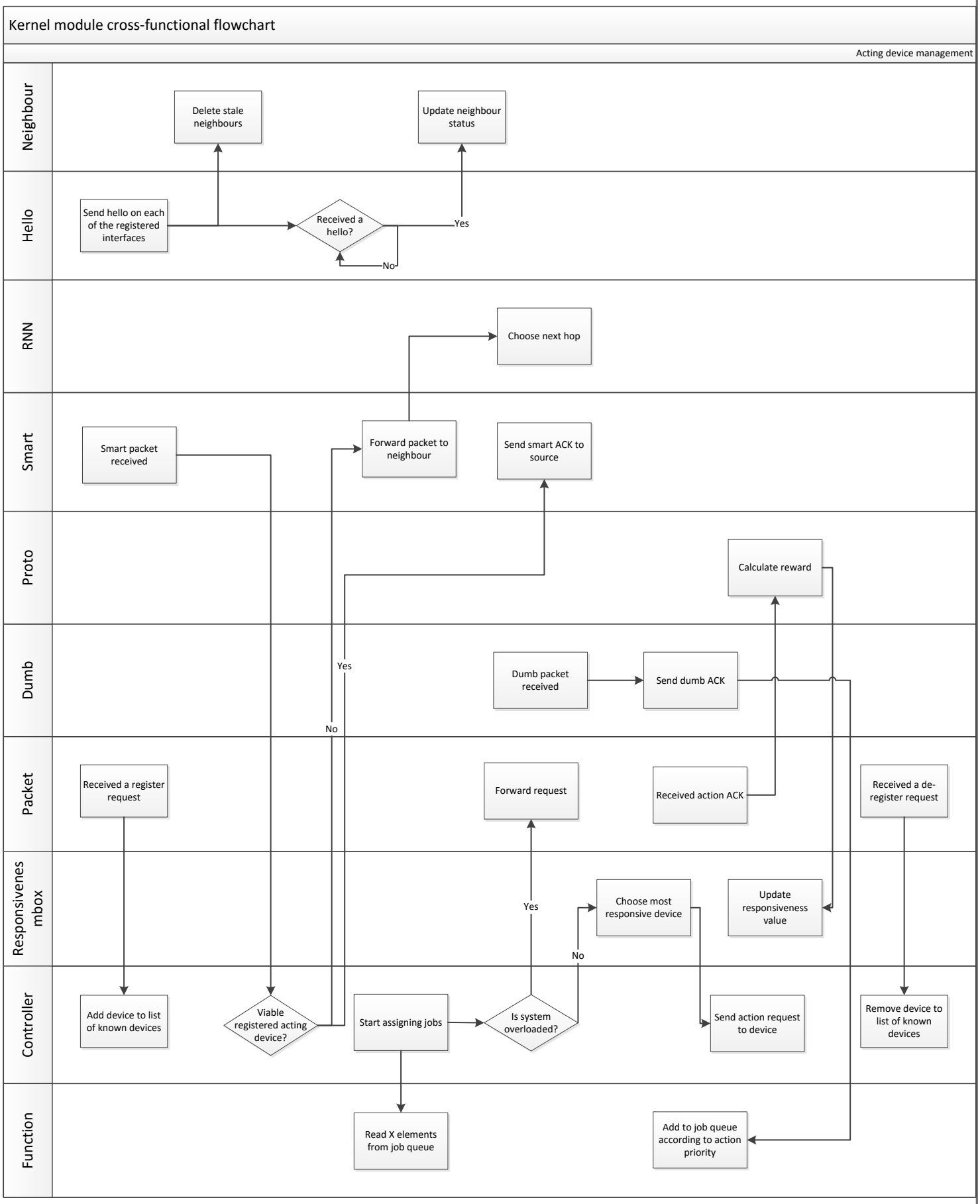


Figure 33- System diagram for an acting device


Kernel module cross-functional flowchart
Measuring device management
Neighbour
Hello
Queue
DPRR
RNN
Smart
Mbox
Proto
Dumb
Packet
Device mbox
Controller
Delete stale neighbours
Update neighbour status
Send hello on each of the registered interfaces
Received a hello?
Yes
No
Add packet to queue
Process all packets in queue
Update route to destination
Update RNN rewards
Choose next hop according to RNN
Smart ACK received
Send a smart packet into the network
No
Store route delays in mbox
Calculate RNN rewards
Dumb ACK received
Send a dumb packet with action request payload
Yes
No
Received a register request
Received a measurement
Act upon the measurement?
Yes
Create an action request
Path to acting device known?
Received a de-register request
Store measurement in device mbox
Add device to list of known devices
Remove device to list of known devices


Kernel module cross-functional flowchart
Acting device management
Neighbour
Hello
RNN
Smart
Proto
Dumb
Packet
Responsivenes mbox
Controller
Function
Delete stale neighbours
Update neighbour status
Send hello on each of the registered interfaces
Received a hello?
Yes
No
Choose next hop
Smart packet received
Forward packet to neighbour
Send smart ACK to source
Calculate reward
Yes
No
Dumb packet received
Send dumb ACK
Received a register request
Forward request
Received action ACK
Received a de-register request
Yes
No
Choose most responsive device
Update responsiveness value
Add device to list of known devices
Viable registered acting device?
Start assigning jobs
Is system overloaded?
Send action request to device
Remove device to list of known devices
Read X elements from job queue
Add to job queue according to action priority

Acknowledgements

I would like to thank both my supervisor Professor Erol Gelenbe for his help and support throughout this project, as well as PhD student Lan Wang, who has repeatedly taken time out of her schedule to show interest in my project.