Informatica — Universiteit van Amsterdam

Bachelor Informatica

UvA Universiteit van Amsterdam

Multi-modal Visualization of Resource Consumption in Computer Systems

David van Erkelens

July 2, 2014

Supervisor(s): Raphael Poss (UvA)

Signed:

Abstract

Resource consumption in computer systems can be a very obscure aspect of an operating system. However, the consumption of resources can have a huge influence on the performance of the system. Therefore, it is useful for an user to gain insight in the resource consumption of processes running on its system and become able to control the consumption of resources. In this thesis, the development of an extensible framework to visualize, monitor and actuate constraints on the resource consumption of groups of processes on recent Linux systems will be discussed. The framework will use `/proc` for the monitoring of the resource consumption and `cgroups` to place limits on the resource consumption of groups of processes. Using these tools, the user can be provided with a high level visualization and management of resource consumption.

Contents

1 Introduction **6**
- 1.1 Existing task managers . . . 6
  - 1.1.1 ps . . . 7
  - 1.1.2 top . . . 7
  - 1.1.3 htop . . . 8

2 Theoretical background **10**
- 2.1 Monitoring system resources . . . 10
  - 2.1.1 System wide monitoring . . . 10
  - 2.1.2 Per-process monitoring . . . 11
- 2.2 Grouping and limiting system resources . . . 12
  - 2.2.1 cgroups . . . 12
  - 2.2.2 Grouping tasks . . . 13
  - 2.2.3 Limiting resource consumption . . . 13
- 2.3 Summary . . . 13

3 Framework Design **15**
- 3.1 Back end . . . 15
  - 3.1.1 Monitor agent . . . 15
  - 3.1.2 Actuator agent . . . 15
- 3.2 Front end . . . 16
- 3.3 Communication . . . 16
  - 3.3.1 Protocol . . . 17
- 3.4 Summary . . . 18

4 Implementation **19**
- 4.1 Developing the back end . . . 19
  - 4.1.1 Monitor agent . . . 19
  - 4.1.2 Actuator agent . . . 20
- 4.2 Developing the front end . . . 21
  - 4.2.1 Desktop . . . 21
  - 4.2.2 Android . . . 22
- 4.3 Summary . . . 24

5 Experiments **25**
- 5.1 System wide monitoring . . . 25
- 5.2 Group monitoring . . . 25
- 5.3 Resource limiting . . . 25
- 5.4 Working with Android . . . 25
- 5.5 Summary . . . 26

6 Conclusions **27**
6.1 Limitations of the framework . . . 27
6.2 Future development . . . 27
6.2.1 Extending the front end . . . 27
6.2.2 Extending the back end . . . 28
6.3 Conclusion . . . 28

Bibliography **29**

Appendices **30**

A Usage **31**
A.1 Dependencies . . . 31
A.2 Compiling and running . . . 31

B Extending the framework **32**
B.1 Extending the back end . . . 32
B.1.1 Monitor agent . . . 32
B.1.2 Actuator agent . . . 32
B.2 Extending the front end . . . 33
B.2.1 Desktop version . . . 33
B.2.2 Android version . . . 34

C Screenshots **35**
C.1 Desktop front end . . . 35
C.2 Android front end . . . 38

CHAPTER 1

Introduction

Today's computer systems are highly capable of multi-tasking. However, due to the closed nature of a computer, it can be hard for an user to understand which process is consuming a lot of resources. However, it is important for an user to understand which process is consuming a lot of power of the system, since limiting this process can improve the performance or increase the battery duration of the system. It is therefore useful to develop a tool which shows the user how many resources a process or a group of processes is consuming. Since computer systems know a wide range of different operating systems nowadays, it has been decided to narrow the subject and develop this tool for the latest versions of Linux (versions with a kernel version past 2.6.24 [5]), since they support `cgroups`. `cgroups` is a relatively new function in the Linux kernel, and is not widely used yet. However, due to the fact that `cgroups` provides an option to limit resource consumption on a group basis instead of a process basis, it grands a new perspective on resource management.

The main question answered in this thesis is: *How can modern Linux features be used to provide an user with high level visualization and management of resource consumption?*

In order to be able to answer this question, several sub questions have to be answered first:

- Which tools are available in Linux to monitor and actuate resource consumption?
- Which resources can be controlled with these tools?
- Which resources are useful to control?
- How can this information be visualized for the user?

Answering these questions will provide a framework for a tool to control the resources of a system. While answering these questions, an extensible framework is discussed. The goal of this framework is to provide the user with a basic, extensible interface to visualize, monitor and limit resource consumption on a Linux computer using multiple front ends, providing support for groups of processes. This framework should be capable of automatically generating groups of processes, so the user can define its own groups to manage without building the groups every time the framework boots.

1.1 Existing task managers

In this section, several existing task managers for Linux will be discussed. The differences between these task managers and the framework developed, will be discussed in section 6.3.

1.1.1 ps

`ps` is a command in UNIX based system which gives a snapshot of current processes [6]. `ps` is short for `process status` and uses `/proc` to retrieve the information about running processes. When run with the `aux` options, to list processes for all users, including processes without a controlling terminal and adding a column for the user running the process, `ps` returns data formatted as following:

```
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0   4056  1844 ?        Ss   00:08   0:05 /sbin/init
root          2  0.0  0.0      0     0 ?        S    00:08   0:00 [kthreadd]
root          3  0.1  0.0      0     0 ?        S    00:08   0:34 [ksoftirqd/0]
root          4  0.1  0.0      0     0 ?        S    00:08   0:30 [kworker/0:0]
root          5  0.0  0.0      0     0 ?        S<   00:08   0:00 [kworker/0:0H]
root          7  0.0  0.0      0     0 ?        S    00:08   0:00 [migration/0]
root          8  0.0  0.0      0     0 ?        S    00:08   0:00 [rcu_bh]
root          9  0.1  0.0      0     0 ?        S    00:08   0:24 [rcu_sched]
root         10  0.0  0.0      0     0 ?        S    00:08   0:02 [watchdog/0]
[...]
david      7099  0.0  0.1   6488  2888 pts/7    Ss+  04:54   0:00 bash
root       8712  0.0  0.0      0     0 ?        S    05:12   0:00 [kworker/u2:2]
root      11437  0.0  0.0      0     0 ?        S<   05:22   0:00 [kworker/u3:1]
david     11476  0.0  0.0   5244  1152 pts/0    R+   05:26   0:00 ps aux
```

`ps` only provides a snapshot of the system, and no repetitive update of the resource consumption. The columns in `ps` note the following properties of the processes:

- `USER`: The user name of the owner of the process
- `PID`: The ID of the process
- `%CPU`: The percentual load of the process on the CPU
- `%MEM`: The percentual load of the process on the RSS
- `VSZ`: The amount of virtual memory used, in kilobytes
- `RSS`: The amount of physical memory used, in kilobytes
- `TTY`: The controlling terminal of the process
- `STAT`: The state of the process, consisting of one of the following values:
  - D: Uninterruptible sleep, usually waiting for I/O
  - R: Running
  - S: Interruptible sleep
  - T: Stopped
  - Z: Zombie process
- `START`: The starting time or date of the process
- `TIME`: The time the process spent on the CPU
- `COMMAND`: The first 8 bytes of the base name of the process's executable file

1.1.2 top

`top` provides an interactive look at the processes currently running on the system [8]. The processes are sorted on CPU usage by default, but can also be sorted on memory usage and runtime. Contrary to `ps`, `top` continously updates the data instead of providing a snapshot. Besides that, `top` also provides system wide monitoring data and basic options to actuate resource consumption, like being able to kill or nice processes. `top` gets its data from the `/proc` filesystem. `top` outputs data formatted as following:

```
top - 17:45:44 up  7:37,  4 users,  load average: 0,32, 0,50, 0,50
Tasks: 161 total,   1 running, 160 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3,5 us, 12,8 sy,  0,0 ni, 83,4 id,  0,0 wa,  0,0 hi,  0,3 si,  0,0 st
KiB Mem:   2064648 total,  1865144 used,   199504 free,   208744 buffers
KiB Swap:  3426300 total,    90404 used,  3335896 free,   775324 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
 1358 root      20   0  213m 117m  71m S   4,9  5,8  25:34.32 Xorg
 6521 david     20   0  325m 120m  28m S   3,0  6,0   4:44.40 chrome
 1588 david     20   0  293m 8804 5716 S   2,6  0,4   0:27.97 gnome-terminal
 6315 david     20   0  763m 111m  42m S   2,3  5,5  11:53.60 chrome
12280 david     20   0  5224 1292  936 R   1,3  0,1   0:00.26 top
```

The columns in `top` note the following properties of the processes:

- `PID`: The ID of the process
- `USER`: The user name of the owner of the process
- `PR`: The priority of the process
- `NI`: The nice value of the process. Negative nice values indicate a higher priority.
- `VIRT`: The amount of virtual memory used
- `RES`: The amount of physical memory used
- `SHR`: The amount of shared memory used
- `S`: The state of the process
- `%CPU`: The percentual load of the process on the CPU
- `%MEM`: The percentual load of the process on the RSS
- `TIME+`: The time the process spent on the CPU
- `COMMAND`: The command used to launch the process

1.1.3 htop

`htop` is an interactive system monitor and process viewer [11]. It is designed as an alternative to `top`, and shares a lot of features with `top`. `htop` is graphically more enhanched than `top`: `htop` supports mouse operations, vertical and horizontal scroll which allows all processes to be viewed and displaying basic graphs of CPU, RSS and swap usage. `htop` also supports basic actuation of resource consumption, by being able to kill or nice processes. Like `ps` and `top`, `htop` also gets its data by reading from the `/proc` filesystem. `htop` outputs data formatted as following:

```
  CPU[|||||||||||||||                          31.8%]     Tasks: 112, 237 thr; 1 running
  Mem[|||||||||||||||||||||||||||||||||||868/2016MB]     Load average: 0.66 0.59 0.54
  Swp[||                                     87/3345MB]     Uptime: 07:42:54
```

```
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
  523 avahi      20   0  3492  1000   888 S  0.0  0.0  0:00.45 avahi-daemon: running
  524 avahi      20   0  3492   148   132 S  0.0  0.0  0:00.00 avahi-daemon: chroot helper
12770 david      20   0  5760  2052  1292 R  5.8  0.1  0:02.14 htop
 7094 david      20   0  290M 15376 11888 S  8.2  0.7  0:04.29 gnome-terminal
 6521 david      20   0  325M  121M 28984 S  2.2  6.0  4:51.75 /opt/google/chrome/chrome
 6351 david      20   0  763M  108M 43880 S  0.0  5.4  0:33.02 /usr/bin/google-chrome-stable
 6380 david      20   0  253M 45992 19972 S  0.5  2.2  0:59.31 /opt/google/chrome/chrome
 1627 david      20   0  325M 12064  7640 S  0.3  0.6  0:06.36 synapse --startup
 3661 david      20   0 31696  1196   888 S  0.3  0.1  0:25.67 adb -P 5037
 6345 david      20   0  763M  108M 43880 S  0.8  5.4  2:52.95 /usr/bin/google-chrome
 6525 david      20   0  325M  121M 28984 S  0.3  6.0  0:20.75 /opt/google/chrome
 1956 david      20   0  561M  170M 42568 S  0.5  8.5 34:40.25 /opt/sublime_text
 1404 david      20   0  5496  1184   916 S  0.0  0.1  0:01.72 init --user
```

The values indicated in the columns in `htop` are equal to the columns of `top`. The three task managers discussed in this section have in common that the visualization is based upon separate processes. If an application is implemented to utilize multiple processes, it becomes very hard to determine the resource consumption of the entire application. An example of an application which uses multiple processes is Google Chrome, which uses a separate process for each tab opened. To monitor this kind of applications, the user needs support to visualize groups of processes. Therefore, this will be implemented in the framework discussed in this thesis.

CHAPTER 2

Theoretical background

In this chapter, the theoretical background behind measuring and limiting resource consumption will be discussed.

All three major operating systems (Windows, OS X and Linux) have options to monitor and control resource consumption. Windows has a class named `Win32_Process` via which information about processes can be retrieved and limits can be set [10]. OS X has the `sysctl` function to get and set process and kernel information, and monitor and limit resource consumption [2]. In the Linux kernel, information about the kernel can be accessed and controlled using so-called pseudo- and virtual file systems. These file systems do not contain any actual files, but can be read like they do. When information is written to these files, it is applied to the kernel and serious harm can be done to the stability of the system. These file systems are very powerful, but should be used with caution.

Linux is the only operating system out of the three to provide `cgroups` [5], a feature which allows an user to limit resource consumption for groups instead of separate processes. Since this is a feature desired for the framework discussed in this thesis, it has been decided to develop the framework for Linux.

2.1 Monitoring system resources

To monitor the usage of system resources of processes, the pseudo-file system `procfs` is used. This filesystem is mounted at `/proc` and contains statistics about every process running on the system, as well as information about the total resources consumption of the system [4].

2.1.1 System wide monitoring

To monitor the system wide consumption of resources, the files mounted directly under `/proc` have to be used.

Monitoring CPU usage

In order to monitor the usage of the CPU, the file `/proc/stat` has to be opened. This file contains the following lines:

```
cpu  21407 3703 67249 217109 3581 0 1123 0 0 0
cpu0 21407 3703 67249 217109 3581 0 1123 0 0 0
intr 545139 221 10400 0 0 0 0 0 0 0 31706 14482 [..]
ctxt 1730199
btime 1401948542
processes 3076
```

```
procs_running 2
procs_blocked 0
softirq 393491 0 244927 655 18266 30202 0 9786 0 546 89109
```

The values noted behind `cpu` are the required values to calculate the CPU usage. The files starting with `cpuX` note the usage of the different cores of the CPU, the `cpu` line contains the accumulated values of all these lines. The first four values are the values required to calculate the consumption of CPU time. Since the `cpu` line is used in the calculations, the average load of all cores in the system is calculated. These values note the amount of time in jiffies[1] that the system spend in various states:

- **user** - Time spend in user mode
- **nice** - Time spend in user mode with low priority
- **system** - Time spent in system mode
- **idle** - Time spent in the idle mode

The total percentage of CPU usage can then be calculated using:

$$\frac{user + nice + system}{user + nice + system + idle} \times 100\% \tag{2.1}$$

However, these values are measured since the boot of the system. To obtain the usage at the interval between n and $n-1$, the following formula is used by the monitor agent:

$$\frac{(user_n - user_{n-1}) + (nice_n - nice_{n-1}) + (system_n - system_{n-1})}{(user_n - user_{n-1}) + (nice_n - nice_{n-1}) + (system_n - system_{n-1}) + (idle_n - idle_{n-1})} \times 100\% \tag{2.2}$$

Using this formula, the average CPU load between two points in time is calculated. The other values noted in `/proc/stat` are not used in the calculation of the CPU usage.

Monitoring RSS usage

In order to monitor the RSS usage of the system, the monitor agent has to open the `/proc/meminfo` file. Among the contents of this file are the following lines:

```
MemTotal:        1015028 kB
MemFree:           78472 kB
Buffers:           10060 kB
Cached:           141524 kB
SwapCached:        18700 kB
Active:           429272 kB
Inactive:         420796 kB
```

In order to calculate the RSS usage, the monitor agent has to apply the following formula:

$$\frac{(MemTotal - MemFree) + (Buffers - Cached)}{MemTotal} \times 100\% \tag{2.3}$$

Contrary to the way the CPU usage is measured, the number calculated using this formula is not an average measured over an interval, but a snapshot of the memory load at a given point in time. The other values noted in `/proc/meminfo` are not used in the calculation of the RSS usage.

2.1.2 Per-process monitoring

In order to provide statistics about individual processes and groups of processes, the resource consumption of individual processes has to be monitored by the monitor agent. This can also be done using the `/proc` filesystem. `/proc` contains a folder for each running process on the system with the ID of the process as folder name. The resource consumption of a group of processes can be calculated by determining the resource consumption of each individual process in the group.

[1] The number of CPU ticks, which has a rate of 100 per second on most architectures

Monitoring CPU usage

To monitor the CPU usage of an individual process, the monitor agent has to open the `/proc/[PID]/stat` file. This file contains data about the statistics of the process, and looks as following [7]:

```
2405 (pulseaudio) S 1 2404 2404 0 -1 4202560 1618 0 16 0 127 147 0 0 9 -11 3 0 7862
    101781504 1054 4294967295 134512640 134589500 3220419104 3220418272 3078337572 0 0
    3674112 19011 4294967295 0 0 17 2 0 0 97 0 0 134593380 134594612 143081472
    3220425462 3220425510 3220425510 3220426728 0
```

The numbers required to calculate the CPU usage are found in the 14^{th} and 15^{th} field of this file. These values contain the amount of time this process has been in user mode and kernel mode, in jiffies. However, like the system wide monitoring of CPU usage, is this value measured since the start of the process. Therefore, another average has to be calculated, as following:

$$\frac{(user_n - user_{n-1}) + (kernel_n - kernel_{n-1})}{total} \times 100\% \tag{2.4}$$

The total amount of jiffies should be equal to `CLK TCK`[2] times the time interval and can be calculated using the denominator of equation 2.2.

Monitoring RSS usage

In order to monitor the RSS usage of a process, the 24^{th} field of `/proc/[PID]/stat` has to be used. This field contains the number of pages the process has in real memory. Multiplying this with the size of one page gives the total amount of used memory. The percentage of used memory can be calculated by the monitor agent as following:

$$\frac{pages \times pagesize}{total} \times 100\% \tag{2.5}$$

The total amount of memory can be extracted from the `/proc/meminfo` file [4].

2.2 Grouping and limiting system resources

To group and limit the consumption of system resources, the `cgroups` feature of the Linux kernel is used. This feature is available since Linux kernel version 2.6.24. `cgroups` features a virual filesystem mounted at `/sys/fs/cgroup` [9].

2.2.1 cgroups

`cgroups` uses *subsystems* to control the different groups. These subsystems contain the link for a group to a hardware resource. Subsystems are also known as resource controllers. The different subsystems are folders in the `/sys/fs/cgroup` directory. The following subsystems are available [17]:

- `blkio` - This subsystem limits and blocks I/O from and to physical media like hard disk drives and USB sticks
- `cpu` - This subsystem uses the scheduler to prioritize CPU access for different groups
- `cpuacct` - This subsystem generates reports on CPU usage by different groups
- `cpuset` - This subsystem allows or denies access to different cores on a multi-core system for different groups
- `devices` - This subsystem allows or denies access to devices for tasks in a group
- `freezer` - This subsystem suspends and restarts tasks in a group

[2]The amount of clock ticks per second

- `memory` - This subsystem limits the usage of the memory for processes in a group
- `net_cls` - This subsystem tags network packets with an identifiers so the Linux traffic controller can identify packets from different groups
- `net_prio` - This subsystem prioritizes access to different network interfaces for different groups

Using these subsystems, `cgroups` is capable of resource limiting, prioritization of processes, accounting resource consumption and checkpointing groups of processes. `cgroups` allows one process to be part of multiple groups, given the limit that these groups can not be connected to the same subsystem.

2.2.2 Grouping tasks

To create a new group, the actuator agent creates a new folder in the directories of the subsystems the group has to be connected to. The name of the folder equals the name of the group. Inside these folders, the `cgroups` service creates serveral files, which contain the dynamics of the group. One of these files is `tasks`, which contains a list of IDs of processes in the group. In order to add a process to a group, its ID has to be written in this file.

2.2.3 Limiting resource consumption

A limitation or priorization of a group can be set by the actuator agent by writing in one of the files in `/sys/fs/cgroup/[SUBSYSTEM]/[GROUP]/`. Since different subsystems contain different files, a per-subsystem breakdown of the files to be edited will be given here.

Prioritizing CPU usage

In order to limit the usage of the CPU for a group, priorities can be given to different groups. These priorities are dynamic. The priority of a group is set in the `cpu.shares` file inside the group folder. This file is part of the `cpu` subsystem. By default, the priority is set to 1024. Lowering this number grands the group a lower priority on the CPU, while increasing the number grands the group a higher priority. However, lowering the priority of a group does not necessarily imply that the processes in the group spend less time on the CPU, since this group could be the only group requiring CPU time. When two groups have their priorities set to respectively 1024 and 2048 and both groups require the same amount of CPU time, the second group will spend twice as much time on the CPU than the first group due to its higher priority.

Limiting RSS usage

Contrary to the limitation of the CPU, a static limitation can be set to the usage of the RSS. To accomplish this, the `memory.limit_in_bytes` file has to be edited. This file is part of the `memory` subsystem. The value of this file sets the maximum amount of user memory (including file cache) for the group. When no suffix is added to the value in this file, the amount is interpreted as bytes. However, larger units can be represented by adding a suffix like `K`, `M` or `G`.

2.3 Summary

In order to monitor resource consumption, the `/proc` file system has to be read. This file system contains information about every running process on Linux, as well as system wide information. To determine the system wide CPU usage, `/proc/stat` has to be opened by the monitor agent. This file contains information about the time in jiffies the system spend in user, kernel, system and idle mode. Using equation 2.2, the average CPU load in an interval can be calculated.

To monitor RSS usage, `/proc/meminfo` has to be read by the monitor agent. This file contains information about the usage and total amount of memory. Using equation 2.3, the total

usage of RSS of the entire system can be calculated. Contrary to the calculated CPU usage, this number is a snapshot and not an average over an interval.

`/proc` contains a subdirectory for every running process on the system. Inside this directory, a file named `stat` can be opened by the monitor agent. This file contains serveral numbers, among those numbers are two numbers indicating the time in jiffies spent in user and kernel mode. Using these numbers and equation 2.4, the CPU usage of a single process over the last interval can be calculated.

`/proc/[PID]/stat` also contains the number of pages in the RSS used by the process. Along with the size of one page and the total memory size obtained from `/proc/meminfo`, the RSS consumption of one process can be calculated using equation 2.5

To limit resource consumption, `cgroups` is used. `cgroups` features a virtual filesystem mounted at `/sys/fs/cgroup`. `cgroups` works with subsystems, which are resource controllers and are mounted directly under the `cgroup` folder. Inside the subsystem folder, a new folder can be created in order to create a new group. To add processes to a group, the process ID has to be added to the `tasks` file in the group directory. To limit consumption of a resource, a file inside the group in the corresponding subsystem has to be edited. In case of CPU limitation, a priority can be set to a group by editing `/sys/fs/cgroup/cpu/[GROUP]/cpu.shares`.

CHAPTER 3

Framework Design

In this chapter, the high-level design of the framework will be discussed. The framework is split into two parts: the front end and the back end. All calculations and operations are done by the back end, while the visualization of resource consumption and management of groups is handled by the front end.

3.1 Back end

The back end handles all read/write operations on the pseudo- and virtual filesystems, as well as all computations with the data from these read/write operations, as discussed in chapter 2. The back end is split into two separate agents. One of these agents is the monitor agent, which task is to read from the filesystems. The other agent is the actuator agent, whose task is to write to the filesystems.

3.1.1 Monitor agent

The monitor agent reads all required data from the pseudo- and virtual filesystems. Therefore, the monitor agent is in charge of the following tasks:

- Generating a list of active processes (from `/proc`)
- Generating a list of active groups (from `/sys/fs/cgroup`)
- Generating a list of processes in a group (both `/sys/fs/cgroup` and `/proc`)
- Determining system wide resource consumption (from `/proc`)
- Determining resource consumption of a group (both `/sys/fs/cgroup` and `/proc`)

The monitor agent is a stateless agent.

3.1.2 Actuator agent

The actuator agent is in charge of writing data to the pseudo- and virtual filesystems. Therefore, the actuator agent is in charge of the following tasks:

- Creating new groups (using `/sys/fs/cgroup`)
- Editing existing groups (using `/sys/fs/cgroup`)
- Setting or removing limits for groups (using `/sys/fs/cgroup`)

The actuator agent is a stateless agent.

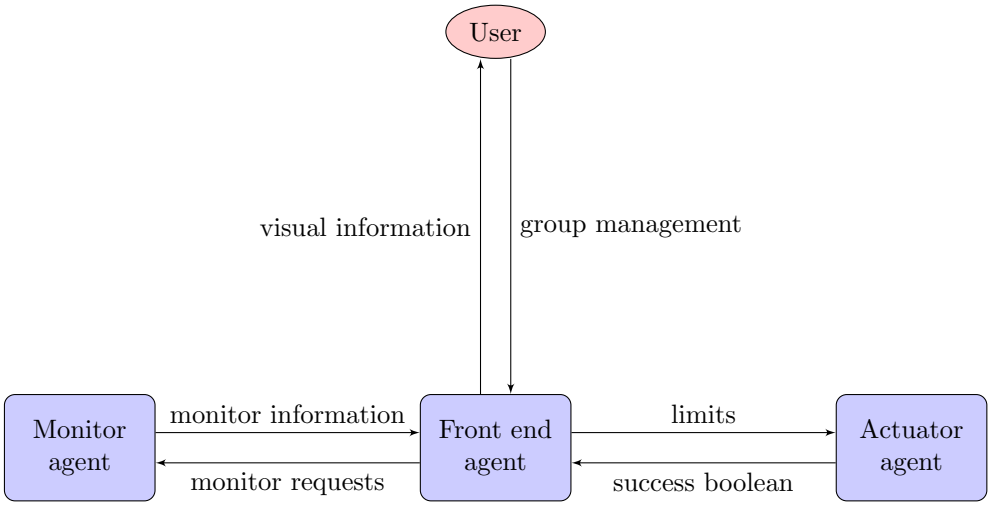

Figure 3.1: Communication between the agents

3.2 Front end

The front end provides the layer between the user and the back end. Using the front end, the user can view the visualization of resource consumption of the entire system, compared to the resource consumption of one group. The user is also capable of setting limitations on resource consumption for a group, as well as creating new groups and editing existing ones. Therefore, the front end agent is in charge of the following tasks:

- Visualising resource consumption
- Providing an interface to create new groups
- Providing an interface to edit existing groups
- Providing an interface to add limits to groups
- Providing an overview of processes running on the system
- Providing an overview of groups active on the system

To keep data up to date, the front end agent has a polling loop, requesting information from the monitoring agent every 0.5 seconds.

Contrary to the back end agents, the front end agent knows multiple states. This is caused by the front end agent requiring information about the active groups, but this information needs to be received from the monitor agent before the front end can send requests to edit or monitor these groups. The first call of the front end agent should be MR2 as defined in section 6.2.2. After this call has been answered, the front end agent is aware of the groups active on the system and requests to edit these groups can be made.

3.3 Communication

This section describes the network protocol used between the three agents. The different agents communicate over TCP, allowing the front end and back end to be run on separate systems. All communication is done in JSON format. This set-up allows the front end and back end to be run on separate systems, or a possible extension to communicatie with multiple back ends from one front end. The communication between the agents is designed as following:

3.3.1 Protocol

Front end to monitor procotol

The front end agent can send requests to the monitor agent formatted as following[1]:

```
monitor call: {act: 'monitor'}                                  (MR1)
            | {act: 'groups'}                                   (MR2)
            | {act: 'processes'}                                (MR3)
            | {act: 'group_monitor', args: {name: String}}      (MR4)
            | {act: 'group_procs', args: {name: String}}        (MR5)
            | {act: 'group_limits', args: {name: String}}       (MR6)
```

MR1 is a request to get the global resource consumption. MR2 is a request to get the groups currently active on the system. MR3 is a request to get the processes currently running on the system. MR4 is a request to get the resource consumption of a group. MR5 is a request to get the processes of a group, and MR6 requests the limits set for a group. These requests are answered by the monitor agent as following:

```
MA1:        {memory_load: Float, cpu_load: Float}
MA2:        {Group [, Group]*}
MA3:        {Process [, Process]*}
MA4:        {memory_load: Float, cpu_load: Float}
MA5:        {Process [, Process]*}
MA6:        {Limit [, Limit]*}

Group:      {subsystems: {[Subsystem [,Subsystem]*]},
            processes: {[Process [, Process]*]}}
Subsystem:  cpu | memory
Process:    String: Integer
Limit:      String: String
```

For example, a MR3 request to the monitor could be answered with:

```
{
    "upstart-udev-br": 395,
    "md": 21,
    "rcu_bh": 8,
    "scsi_eh_1": 45,
    "kthreadd": 2,
    "iprt": 461,
    "khubd": 20,
    "rsyslogd": 488,
    "kworker/u2:0": 6,
    "rcu_sched": 9,
    "kswapd0": 25,
    "gnome-terminal": 2847,
    "mdm": 1332,
    "cups-browsed": 571,
    "mintUpdate": 1689,
    "netns": 13,
    "avahi-daemon": 521,
    "NetworkManager": 729,
    "ksoftirqd/0": 3,
    "getty": 1384,
    [...]
    "python": 8971
}
```

[1] MR stands for Monitor Request, MA notes Monitor Answer

In this example, every entry in the JSON array notes a process, with its name as key and its ID as value.

Front end to actuator protocol

The front end agent can send requests to the actuator agent formatted as following[2]:

```
actuator call: {act: 'create', args:
                {name: string, procs: {[ Integer [, Integer]*]}}    (AR1)
              | {act: 'edit', args:
                {name: string, procs: {[ Integer [, Integer]*]}}    (AR2)
              | {act: 'limit', args:
                {name: string, limits: {Limit [, Limit]*}}}         (AR3)

Limit:          Subsystem: String
Subsystem:      cpu | memory
```

AR1 is a request to create a new group, providing the name of the group and the processes to be included. AR2 is a request to edit a group, also providing the name of the group and the processes to be included. AR3 is a request to set limits to a group, providing the group name and the limits to be set. All requests get the same answer from the actuator agent, indicating whether the request was executed successfully:

`AA1, AA2, AA3:` **boolean**

For example, an AR1 request could be formatted as following:

```
{
    act: 'create',
    args: {
        name: 'my_group',
        procs: [
            2986,
            8742,
            1665,
            8881,
            8971
        ]
    }
}
```

This requests the actuator agent to create a new group called `my_group` containing the processes 2986, 8742, 1665, 8881 and 8971. No communication exists between the monitor agent and the acuator agent.

3.4 Summary

The framework is split into three separate agents. The back end consists of two agents. One of them is a monitoring agent, in charge of reading the pseudo- and virtual filesystems. The other is an actuator agent which is in charge of writing to these filesystems. Both back end agents communicate with the front end agent, which is in charge of visualizing the data from the back end and providing an interface to work with different groups.

The agents communicate over TCP, allowing the back end and front end to be ran on separate systems. Communication between agents is formatted using JSON. The front end agent runs a polling loop, requesting data from the monitor agent on a regular interval. The front end agent knows multiple states: the first request of this agent is MR2 as defined in section 6.2.2 to ensure knowledge of active groups on the system.

[2] AR stands for Actuator Request, AA notes Acuator Answer

CHAPTER 4

Implementation

In this chapter, the implementation of the high-level design of the framework is discussed. To provide the multi-modal part of the framework, two types of front end have been developed. One version of the front end is a local version, meant to be run on the same system as the back end agents and is built to be controlled with a keyboard and mouse. The other version of the front end is meant to be run on an external Android device, and thus be controlled using a touch screen and touch gestures. This provides the user with two ways to look at the data provided by the back end.

4.1 Developing the back end

Both back end agents are developed in C++. C++ has been chosen because its a relatively low-level programming language, but is very powerful when combined with the Boost library [13]. Because C++ is a low-level language, the resource consumption of the back end is limited to a minimum. Since the design of the back end of the framework is specific for the Linux operating system, portability of the back end agents is not an issue.

4.1.1 Monitor agent

The monitor agent will listen to incoming calls on port `1209`. When a call is received, the JSON will be parsed and depending on the on the value of the `act` parameter, one of the following actions will be executed. Each action has its own function to generate a JSON string to return. To generate the JSON, `libjansson` [14] is used, which is an easy to use but powerful library for C(++).

monitor

When the `monitor` action is received, the monitor agent has to create a JSON string containing information about the global usage of the resources of the system. To calculate this, the `/proc/stat` file has to be parsed as discussed in section 2.1.1. Boost contains a handler for opening directories and files. When a file is opened using Boost, each line of the file can be read into a string. Using a stringstream, the first word of this string can be read and compared to `cpu` and `memory`, to check if the line read is a line required for calculations. Also, for each iteration of the `monitor` call, the last amount of jiffies noted on the `cpu` line is saved to calculcate the CPU consumption on the next iteration.

groups

When the `groups` action is received, the monitor agent has to create a JSON string containing all active groups on the system, including attached subsystems and a list of processes in the group. This is achieved by using Boost to open every `/sys/fs/cgroup/[SUBSYTEM]` folder. A map is

used to save all active groups. Due to the hierarchical structure of `cgroups`, it has to be checked wheter a group is attached to one or multiple subsystems. For each group in each subsystem, an entry in the map is created. However, if the group has already an entry in the map and thus has been added when scanning an earlier subsystem, only the subsystem is added to the already existing entry. When a group has no previous entry in the map, an entry is created containing the current subsystem and the processes in the group detected in the `tasks` file.

processes

When the `processes` action is received, the monitor agent has to create a JSON string containing the process ID and name of every active processes on the system. This is achieved by opening the `/proc` directory using Boost, and creating a folder iterator the examine every entry in the folder. If the folder entry is a directory, its name is checked against a regular expression to check if the folder name is a number, meaning the folder stands for an active process. If the folder passes the check against the regular expression, the `/proc/[PID]/status` file is read to extract the name of the process and a new process entry is added to the JSON string.

group_monitor

When the `group_monitor` action is received, the monitor agent has to create a JSON string containing the resource consumption of a group. This is achieved by first checking wheter the group exists, which is done by opening every `/sys/fs/cgroup/[SUBSYSTEM]` directory and checking for a subdirectory with the name of the group. As soon as this folder is found, the `tasks` file is opened. For each line in this file, indicating a process in the group, the amount of jiffies spend in user and kernel mode is added to a total. The total amount of jiffies for every iteration of the `group_monitor` is saved in a map for every group, and using the last amount of jiffies and the current amount of jiffies, the CPU usage can be calculated using equation 2.4. The RSS usage is calculated as noted in equation 2.5 for every process, and added to the total RSS usage of the group.

group_procs

When the `group_procs` action is received, the monitor agent has to create a JSON string containing the process ID and name of every process in the group provided in the request. This is achieved by first checking wheter the group exists, which is done by opening every `/sys/fs/cgroup/[SUBSYSTEM]` directory and checking for a subdirectory with the name of the group. As soon as this folder is found, the `tasks` file is opened. For each line in this file, indicating a process in the group, the corresponding `/proc/[PID]/status` file is opened to extract the name of the process. The pair of the process name and ID is then added to the JSON.

4.1.2 Actuator agent

The actuator agent will listen to incoming calls on port `1210`. Like the monitor agent, the actuator agent parses incoming JSON and depending on the value of the `act` parameter, one of the following actions is executed. The actuator agent also uses `libjansson` to generate JSON and `libcgroup` [16] to work with groups. Since the actuator agent requires root privileges to write to the virtual filesystem `/sys/fs/cgroup`, the agent has to be run using `sudo`.

create

When the `create` action is received, the actuator agent has to create a new group and add processes to it. First, the actuator agent check wheter the group does not already exist, which is done by checking every `/sys/fs/cgroup/[SUBSYSTEM]` directory for a subdirectory with the name of this group. If the group already exists, the request fails and `false` is returned. Otherwise, the actuator agent creates folders in every `/sys/fs/cgroup/[SUBSYSTEM]` directory with the name of the group. For each created directory, the list of processes is added to the `tasks` file.

edit

When the `edit` action is received, the actuator agent has to edit the list of member processes of the group. This is achieved by first checking every `/sys/fs/cgroup/[SUBSYSTEM]` directory wheter the provided group exists. If the group does not exist, `false` is returned. Whilst scanning every subsystem directory, a list of subsystems the group is connected to is built. For every subsystem, the `tasks` file is truncated and filled with the list of new processes provided.

limit

When the `limit` action is received, the actuator agent has to set limits to an existing group. To ensure validity of the provided limits, every limit provided is parsed and compared to the group name provided. This is done by extracting the subsystem from the limit name, which is trivial since a limit name is build in the form of `[SUBSYSTEM].[LIMIT]`. If the `/sys/fs/cgroup/[SUBSYSTEM]/[GROUP NAME]/[LIMIT NAME]` file exists, the provided limit is valid, since the existance of this folder implies that the group exists and the subsystem concerned is attached to it. If every limit is valid, the limits are applied to the group. Otherwise, `false` is returned.

4.2 Developing the front end

Two versions of the front end will are developed: one desktop version, written in Python and an Android version, written in Appcelerator Titanium [1].

4.2.1 Desktop

The desktop version of the front end is developed in Python. The choice for Python has been made due to the scalabilty and wide range of available libraries. The first choice for a library to develop a front end in Python was `Kivy` [15]. However, after working with `Kivy` for a few days, several troublesome bugs started to appear. For example, when drawing an interactive graph with `Kivy`, old plots do not get removed [12]. This causes 'ghosting' to appear, which clutters the graph and makes it incomprehensible. A workaround for this bug is to completely remove the graph and draw a new graph, but this causes the graph to flicker every 0.5 seconds. Mainly due to this bug, it has been decided to step away from `Kivy` and use `wxPython`. `wxPython` is a port of the C++ `wxWidgets` class to Python [18]. `wxPython` has native support for graphical interface items like buttons, menu bars and sliders. However, `wxPython` has no extensive support for drawing graphs, so `matplotlib` is used for drawing graphs.

The front end agent places a call to the monitor agent every 0.5 seconds. When the answer to this request is received, the result of every resource is added to an array with results of the previous requests. The last 100 items of this array are plotted by `matplotlib`.

The front end has a menu bar, in which group management can be accessed. Group management is accessible via seperate windows, which are implemented as different `wxPython` classes. The different windows, their design and behaviour will be discussed here.

Main window

The main window of the front end agents shows two `matplotlib` graphs, noting the current global resource consumption of the CPU and RSS. The main window has a menu bar, from which the different other windows can be accessed. When a group is marked as active, the resource consumption of the group is drawn in the same graphs as the system wide resource consumption, which is also requested from the monitor agent every 0.5 seconds.

The main window uses the `GridBagSizer` of `wxPython`, so the elements in the window are properly scaled to fit the size of the window. The `GridBagSizer` is based on a grid, but the columns and rows that will stretch are provided by the user. In the main window, this translates to the graphs being stretched, but labels remaining the same size.

New group window

The window to create a new group, as defined in the `NewGroupFrame` class, consists of two lists containing process IDs and names. One list indicates the processes to be included in the new group, and the other list indicates the processes not to be included in the group. In between those lists are two buttons, one to switch an item from the list of processes to be included to the list of processes not to be included, and one button to do the exact opposite. Below the lists is a textbox, in which the name of the new group can be written. The window is concluded with a button to submit the new group to the actuator agent. As soon as this button is pressed, the list of processes to be included in the group and the name of the group are converted into JSON, and send to the actuator agent.

The new group window uses a horizonal `BoxSizer` to scale the widgets: the lists get resized, but the buttons in between remain the same size.

Edit group window

The edit group window, as defined in the `EditGroupFrame` class, is largely the same as the new group window. Since the edit group window edits an already existing group, the name of the group does not have to be defined again and the list of processes to be included is already filled. The action send with the JSON to the actuator agent also differs (`edit` instead of `new`).

Set active group window

The set active group window, as defined in the `SetGroupFrame` class, consists of a dropdown box and a button to save the active group. The active group is saved in a global variable in the program and therefore accessible in every window class. Due to the basic layout of the set active group window, this window is not scalable.

Current groups window

The current groups window, as defined in the `ActiveGroupsFrame` class, consists of a tree list containing every group and its processes and attached subsystems. The current groups are fetched using a MR2 request, after which the received JSON is parsed and added to the tree. To make sure that the information in the list is up-to-date, a new MR2 request is send every time the current groups window is opened.

Set limit window

The set limit window, as defined in the `SetLimitsFrame` class, consists of a label showing the name of the active group and two widgets to set limits on the resource consumption: a slider to set the CPU priority and a textbox to set the RSS limit. These widgets get their value via a MR6 call and are placed in a panel, which allows a future extension of this window. Below the panel, a button is placed to send the limits to the actuator agent, using an AR3 request.

4.2.2 Android

The Android version of the front end is developed using Appcelerator Titanium. Titamium is a JavaScript based framework to develop applications for multiple mobile operating systems, like Android and iOS. Titanium has build-in TCP support and is therefore capable of communicating with the existing back end agents. The communication with the back end is identical to the desktop version of the back end, but the visualization is different.

When the application is booted, the local IP address of the system running the back end agents has to be provided. The back end agents provide the IP address when booted. As soon as the connection between the agents has been established, the user is provided with the main window of the application. Using swipe gestures, the user can navigate between the different windows as

indicated in figure 4.1. The Android front end agent consists of the following windows:

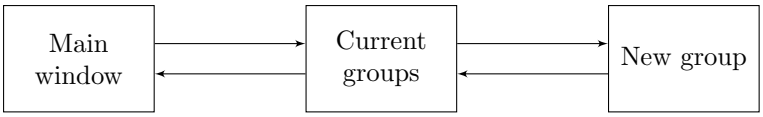

Figure 4.1: Swipe gesture navigation for the Android front end agent

Main window

The main window, as defined in `main.js`, serves the same function as the main window of the desktop front end agent. The Android front end agent uses `RaphaelJS` instead of `matplotlib` to draw graphs of the resource consumption [3]. The graphs can be tapped, after which an alert dialog containing the current consumption of the resource concerned is shown.

If an active group is set, the resource consumption of this group will also be drawn into the graphs. The front end agent sends a MR1 request to the monitor agent every 0.5 seconds to fetch the current resource consumption of the entire system, and a MR4 request every 0.5 seconds if an active group is set.

The graphs are drawn into a `ScrollView`, which allows scrolling of the content if the content is larger than the screen size. This provides an environment which can be easily extended.

Current groups window

The current groups window, as defined in `current.js`, shows a scrollable list of groups currently active on the system. Next to the name of the group, three buttons are shown: `Processes`, `Edit` and `Limit`. When the `Processes` button is clicked, a pop up containing a list of processes in the group is shown. The information for this list is fetched by sending a MR5 request to the monitor agent.

When the `Edit` button is clicked, a scrollable list of all processes on the system is shown, with checkboxes next to the name of every process. Each process included in the group is checked, and processes can be added to the group by checking the checkbox next to the name of the process. The information shown in this window is fetched by sending a MR2 and MR5 request to the monitor agent. As soon as the save button is clicked, an AR2 request is send to the actuator agent to save the edited group.

When the `Limit` button is clicked, a popup to limit the resources of the group is shown. This popup contains a slider to prioritize the CPU usage and a textbox to set a limit on the RSS usage. The value of these widgets is set using data fetched via a MR6 request. As soon as the save button is clicked, an AR3 request is send to the actuator agent to set the limits to the group.

New group window

The new group window, as defined in `new.js`, shows a scrollable list of active processes on the system. This information is fetched using a MR2 call. Each of these processes has a checkbox next to its name, indication wheter the process should be included in the new group. As soon as the save button is clicked, an AR1 request is formulated and send to the actuator agent to create the new group.

4.3 Summary

The back end agents of the framework are developed in C++, using the Boost library. The monitor agent listens to incoming requests on port `1209`, the acutator agent on port `1210`. These agents are platform specific, but this is no issue since the pseudo- and virtual filesystems required by these agents are also platform specific. The back end uses `libjansson` to generate the JSON used to transfer data between agents.

The front end agent of the framework is developed for two different platforms: a desktop computer and an Android device. The desktop version of the front end is designed to be executed on the same system as the back end agents, and is written in Python using the `wxPython` framework. Graphs are drawn using `matplotlib`. The Android version of the front end is developed using Appcelerator Titanium, and graphs are drawn using `RaphaelJS`. Both front end agents fetch data about resource consumption from the monitor agent every 0.5 seconds.

CHAPTER 5

Experiments

In this chapter, several experiments on the application developed will be discussed. All experiments are run on a Packard Bell Dot S netbook running Linux Mint 14.

5.1 System wide monitoring

In order to test the output of the application on system wide monitoring, a `make` process has been executed while monitoring the global resource consumption. The result of this action can be seen in screenshot C.1. The `make` process was started around timestamp 68 and finished around timestamp 88. Within this interval, a CPU usage of 100% and a significant increase in RSS usage can be seen.

5.2 Group monitoring

In order to test the output of the application on group monitoring, a group containing a Google Chrome process was created. With this Chrome process, severval YouTube videos have been viewed in order to increase CPU and RSS usage. The output of this experiment can be seen in screenshot C.3. In this screenshot, it can be seen that the Chrome process causes an increase in both the CPU and RSS usage, but does not consume all of the resources.

5.3 Resource limiting

In order to test the limiting of resources, the Google Chrome group from section 5.2 is used again. The CPU priority is set to a low level (40%), and the RSS usage is limited to 32MB. The same YouTube videos as in the previous experiment are opened. However, as seen in screenshot C.4, the consumption of the resources is much lower than before. Google Chrome even crashed, giving a notification about running out of memory.

5.4 Working with Android

In order to test the Android front end, the same experiments were executed as described above. When adding Chrome processes to a group and activating this group using the restraints described above, the results were similar to the results described above.

5.5 Summary

From these experiments, it can be concluded that the framework works. By setting a limit on both the CPU and RSS usage of Google Chrome, it became practically impossible to watch a few YouTube videos, because Chrome ran out of memory. The difference between the type of limit set on the CPU and RSS also became clear: the RSS had a hard limit which it could not exceed, but the CPU was limited using a priority, resulting in the group still using much of the CPU time since other groups did not require much CPU time. The framework also runs when the front end is executed from an external Android device.

CHAPTER 6

Conclusions

In this chapter, the conclusions drawn from the experiments and the development of the framework will be discussed.

6.1 Limitations of the framework

One of the features of `cgroups` is that one process can be part of multiple groups, as long as these groups are not connected to the same subsystem. In the framework discussed, when a group is created, it is automatically connected to every subsystem available. Due to this, a process can not be part of multiple groups in the framework. This is a design choice make to simpify working with `cgroups`, since making processes part of only one group is easier to understand for the user.

A serious issue with the external front end is the fact that the connection is not secured. If another user is aware of the IP address of the computer running the back end agents, it can connect using its own external front end to monitor and limit resource consumption on the system. The framework can be extended with a password, in order to make the connection secure. This can be done by extending all requests with an extra password key, and checking the password in the back end agents.

6.2 Future development

The framework discussed is extensible, and can be extended in the future. In this section, theoretical options for futher development of the framework will be discussed, while technical details about extending the framework can be found in appendix B.

6.2.1 Extending the front end

Currently, the framework consists of two resources being monitored: the CPU and RSS. However, a computer system contains way more resources to be monitored. Both front end agents display the graphs of resource consumption inside a container that will be scrollable as soon as the content size exceeds the screen size. This way, to add a new resource to be monitored, only a new graph has to be added to the container and the correct JSON has to be parsed. Adding a new resource to the front end also requires the resource to be added to the back end, which will be discussed in section 6.2.2

Another extension would be to port the front end to other operating systems. The desktop version of the front end is written in Python, which is a programming language which can be run on multiple operating systems. If the desktop version of the front end would be extended to connect to other IP addresses than `127.0.0.1`[1], the desktop front end could also be executed

[1]The IP address which a computer uses to indicate itself

on a Windows computer or Mac, while it would still manage the resources of a Linux system.

Titanium is a framework build to develop for multiple operating systems at once. Titanium code can be compiled to Android, iOS, Windows Phone, BlackBerry OS and Mobile Web. However, Titanium contains some small inconsistencies between the resulting applications on different operating systems, so porting the Android framework to other mobile operating systems could require some small rewritings of the code to provide a consistent front end application among all mobile operating systems.

6.2.2 Extending the back end

One of the possible extensions of the back end is to add new resources to be managed. When a new resource is added to the front end, as discussed in section 6.2.1, it also has to be added to the monitor agent and to the actuator agent if the resource should be limitable.

Another extensions of the back end is to completely rewrite the back end to work with another operation system. Even though `/proc` and `cgroups` are not available in operating systems like Windows and OS X, there are functions to monitor and limit resource consumption. As long as the protocol as defined in section is used, a modified back end works with the existing front end.

6.3 Conclusion

The framework discussed provides a basic set up for the visualization, monitoring and actuation of resource consumption in Linux computer systems. While the framework currently only supports CPU and RSS monitoring and actuation, it is easily extensible and usable for a wide variety of resources. Using a split front and back end communicating over TCP, it becomes possible to manage resources from an external device.

Using the pseudo filesystem `/proc`, information about the resources consumption of running processes and the entire system can be retrieved. This information is send to the front end agent, which plots the data into a graph which grands the user insight in the consumption of resources of the system and groups of processes on the system. Using the front end, the user can send requests to the actuator agent to limit the resource consumption of groups. The actuator agent implements the virtual filesystem `/sys/fs/cgroup`, from which groups can be managed using subsystems. `cgroups` is a very powerful function of the Linux kernel, and the framework can be extended to provide much more functionality of `cgroups`.

When compared to existing task managers discussed in section 1.1, the framework discussed provides two key features which other task managers lack: the option to actuate groups of processes and place limits using `cgroups`, and the option to manage resources from an external device. The goal of the framework as discussed in the introduction, to provide the user with a basic, extensible framework to visualize, monitor and limit groups of processes, is reached within this thesis.

The framework discussed also has potential to be scaled to more resources and to function on more devices, or even visualize multiple back end inputs into one front end. The development of the framework should therefore be continued beyond the scope of this thesis.

Bibliography

[1] Appcelerator. *Titanium Mobile Application Development*. http://www.appcelerator.com/titanium/.

[2] Apple. *sysctl(3) Mac OS X Developer Tools Manual Page*. https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/sysctl.3.html.

[3] Dmitry Baranovskiy. *Raphael JS*, 2012. http://www.raphaeljs.com/.

[4] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold. *The /proc filesystem*, 2009. https://www.kernel.org/doc/Documentation/filesystems/proc.txt.

[5] Jonathan Corbet. *Notes from a container*, October 29, 2007. http://lwn.net/Articles/256389/.

[6] The Open Group. Base specifications. *IEEE Std 1003.1*, (7), 2013.

[7] Michael Kerrisk. *proc(5) - Linux manual page*, 2014. http://man7.org/linux/man-pages/man5/proc.5.html.

[8] William LeFebvre. *top - display and update information about the top cpu processes*, 2007. http://www.unixtop.org/man.shtml.

[9] Paul Menage. *cgroups*. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[10] Microsoft. *Win32_Process class*. http://msdn.microsoft.com/en-us/library/aa394372%28v=vs.85%29.aspx.

[11] Hisham Muhammad. *htop - an interactive process-viewer for Linux*. http://hisham.hm/htop.

[12] sirpercival. *Moving a MeshLinePlot leaves artifacts*, May 6, 2014. https://github.com/kivy-garden/garden.graph/issues/8.

[13] Boost team. *Boost C++ libraries*. http://www.boost.org/.

[14] Jansson team. *Jansson - C library for working with JSON data*. http://www.digip.org/jansson/.

[15] Kivy team. *Kivy: Cross-platform Python Framework for NUI Development*. http://www.kivy.org/.

[16] Libcgroup team. *libcgroup*. http://libcg.sourceforge.net/html/index.html.

[17] RedHat team. *Introduction to Control Groups*. https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html.

[18] wxPython team. *wxPython*, 2013. http://www.wxpython.org/.

Appendices

APPENDIX A

Usage

In this appendix, all information about running the framework will be discussed.

A.1 Dependencies

In order to compile the back end and run the front end, serveral dependencies have to be installed on the system. These programs have to be installed on the system:

- `libboost-all-dev`
- `libjansson-dev`
- `libcgroup-dev`
- `cgroup-bin`
- `python`
- `python-wxgtk2.8`
- `python-matplotlib`

A.2 Compiling and running

Both the monitor agent and the actuator agent can be build using the `make` command. When finished, the monitor can be started using the `./monitor` command from the `monitor` folder. The actuator agent needs root privileges, so it has to be started using the `sudo ./actuator` command from the `actuator` folder. As soon as both back end agents are booted, the desktop front end agent can be started using the `python frontend.py` command from the `gui` folder, or the Android front end agent can be booted. In order for the Android front end agent to be able to connect to the back end agents, the Android device has to be connected to the same network as the system running the back end agents.

APPENDIX B

Extending the framework

The framework discussed is built to support extensions. In order to extend the framework, some files have to be edited. This section described which files have to be edited in order to add new functionality to the framework.

B.1 Extending the back end

In this subsection, the way to edit the back end agents to manage more resources will be discussed.

B.1.1 Monitor agent

In order to edit the monitor agent to support more resources, the `main.cpp` file in the `monitor` folder has to be edited. To support a new resource in the monitoring of the system wide resource consumption, the `std::string monitor()` function has to be extended. In this function, the CPU and RSS usage are computated and added to a JSON object called `toSend`, which is converted into a string by `libjansson`. A new item can be added to this JSON object using the function `json_object_set_new`. This function expects three parameters: the JSON object to add an item to, a string with the key of the object and the item to add to the JSON object, which should also be a JSON object. `libjansson` contains functions to generate JSON objects from basic data types like integers, floats and strings. For example, the CPU usage is added to `toSend` using `json_object_set_new(toSend, "cpu_load", json_real(cpu_load));`. When adding a new item to this JSON object, a similar syntax should be used.

In order to add a new resource to group monitoring, the function `std::string cgroup_status` has to be edited. This function contains a loop computating resource consumption for every process, and adding this to a total. To extend this function to support more resources, this loop can be utilized. After this loop, another JSON object, `group_status`, is generated. The generated totals from the process loop are added to this object, which is then returned as a string. To add a new item to this JSON object, another `json_object_set_new` call has to be added.

B.1.2 Actuator agent

To support actuation of extra resources, the `main.cpp` file in the `monitor` folder has to be extended. Due to the dynamic handling of limits by the actuator as described in section 4.1.2, only the link to the subsystem required to actuate the resource has to be added. This has to be done in the `int new_cgroup` function. The creation of groups is handled by `libcgroup`, by adding items to a cgroup object. Adding subsystems to a group is done by calling the function `cgroup_add_controller`, which requires two parameters: the cgroup object to attach the subsystem to, and the name of the subsystem to add. For example, the `cpu` subsystem is added to the new group by calling `cgroup_add_controller(group, "cpu");`.

B.2 Extending the front end

In this section, the way to extend the front end to support more resources is discussed.

B.2.1 Desktop version

The desktop version of the front end can be extended to support more resources by editing the `frontend.py` file in the `gui` folder. To monitor more resources, the `TaskGroupManager` class has to be edited. In this class, multiple functions have to be edited. In the `InitUI` function, a new graph has to be added. This is done using the following syntax:

```
self.cpu_plot = MatplotPanel(panel)
```

As discussed in section 4.2.1, this class utilizes a `GridBagSizer` to display all widgets. The new graph has to be added to this sizer using the following syntax:

```
sizer.Add(self.cpu_plot, pos=(1,0), span=(1,4), flag=wx.EXPAND|wx.LEFT|wx.RIGHT,
        border=5)
```

The graphs indicating the CPU and RSS usage are located at rows 1 and 2, so a next graph should be inserted at row 3. However, the `GridBagSizer` needs to know which rows and columns should be stretched when resizing the window, to one last line has to be added to this function:

```
sizer.AddGrowableRow(X)
```

In which `X` indicates the row at which the new graph is inserted.

To keep track of the consumption of the new resource over time, a global array needs to be declared. Every time an update about the system wide resource consumption is received, the value for the new resource has to be added to this array. To draw the newly received data in the graph generated before, the JSON has to be parsed, added to the array and the data in the graph has to be updated. This can be done using the following syntax in the `newInfo` function:

```
cpu_load.append(float(received_json['cpu_load']))
self.cpu_plot.updateData(cpu_load, 'cpu')
```

In which `cpu_load` is the global array, `received_json` contains the JSON received from the monitor agent and `self.cpu_plot` the graph created before. The new resource also has to be monitored on a group level. The `newInfo` function also contains a loop iterating over every active group, and requesting the resource consumption of that group. The new resource can be added to the cgroup history using the following syntax:

```
cgroups[i][1].append(float(groupjson['cpu']))
```

The CPU and RSS usage are placed at index 1 and 2 of the group object, so new resources should start at index 3. However, the group object needs to be extended to support more arrays, which is done by editing the `setGroup` function of the `SetGroupFrame` class. When adding a new resource to the framework, the line `cgroups.append([(self.cb.GetValue(), [0], [0])])` should become `cgroups.append([(self.cb.GetValue(), [0], [0], [0])])` to support an extra resource.

Some adjustments have to be made to the `updateData` function of the `MatplotPanel` class. To give the plot the correct label, a check for the `mode` variable has to be extended. This is done using the following syntax:

```
if mode == 'cpu':
    self.axes.set_title('CPU usage', size=12)
```

In which `mode` is the second parameter used in the `updateData` call used before.

Finally, to support actuation of a new resource, a new widget should be added to the `__init__` function of the `SetLimitFrame` class. Then, in the `SendLimits` function of this class, the value of the widget has to be processed and added to the JSON object to send to the actuator agent, using the limit to set as key.

B.2.2 Android version

In order to extend the Android version of the framework, the `main.js` file has to be edited. Like the desktop version of the front end agent, the history of the resource consumption is kept in global arrays. To support a new resource to be monitored, another global array has to be created. Then, in the `getData` function, the value of the new resource has to be extracted from the JSON and added to the global variable, using the following syntax:

```
cpu_usage.push(received_json['cpu_usage']);
```

The global variable keeping track of the resource consumption of the active group also has to be extended. This is done by extending the `initGroupData` function, using the following syntax:

```
group_data['cpu_load'] = new Array();
```

In which `cpu_load` should be replaced by a key to represent the resource to be added. Then, in the `initUI` function, a graph has to be created for the new resource. This is done by calling the `insertNewGraph` function, using the global variable declared before, the key used in the global array for group resource consumption and the label to display above the graph as parameters. The framework takes care of adding the graph to the view and refreshing the graph when new data arrives.

In order to allow the new resource to be actuated by the Android front end agent, the `current.js` file has to be edited. A new widget should be added to the `addLimitWidgets` function. Then the `setLimitValues` function has to be extended to extract the value of this widget and add it to the JSON object to be send to the actuator, using the limit to set as key.

APPENDIX C

Screenshots

C.1 Desktop front end

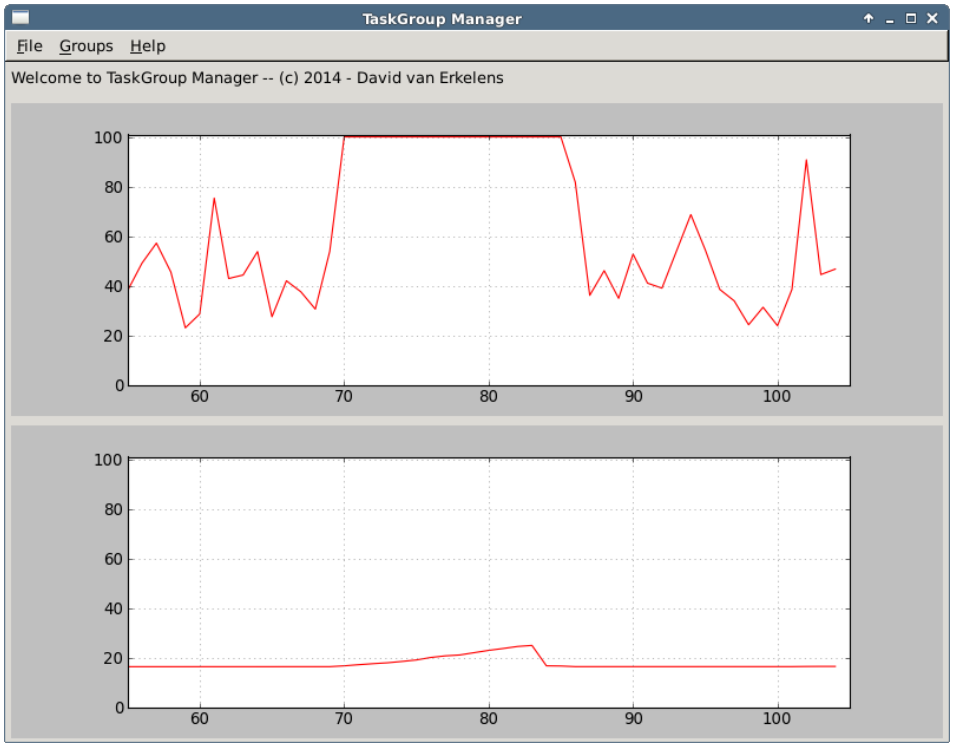

Figure C.1: Main interface. This screenshot shows the main window during a make operation, in which the CPU is fully used and the usage of RSS increases significantly.

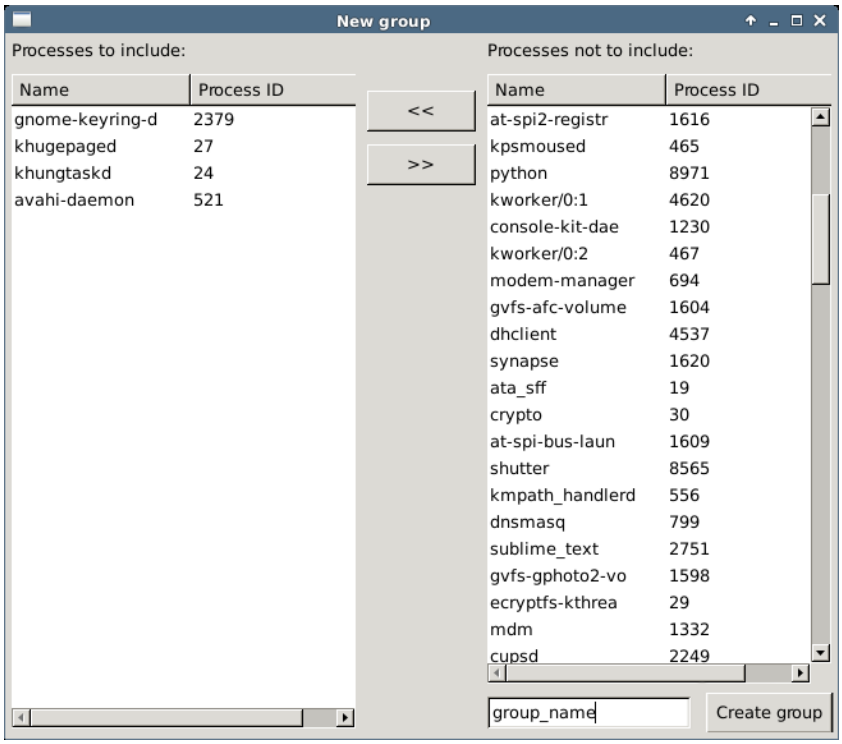

Figure C.2: Interface to create a new group. This screenshot shows the creation of a new group called group_name, containing processes 2379, 27, 24 and 512.

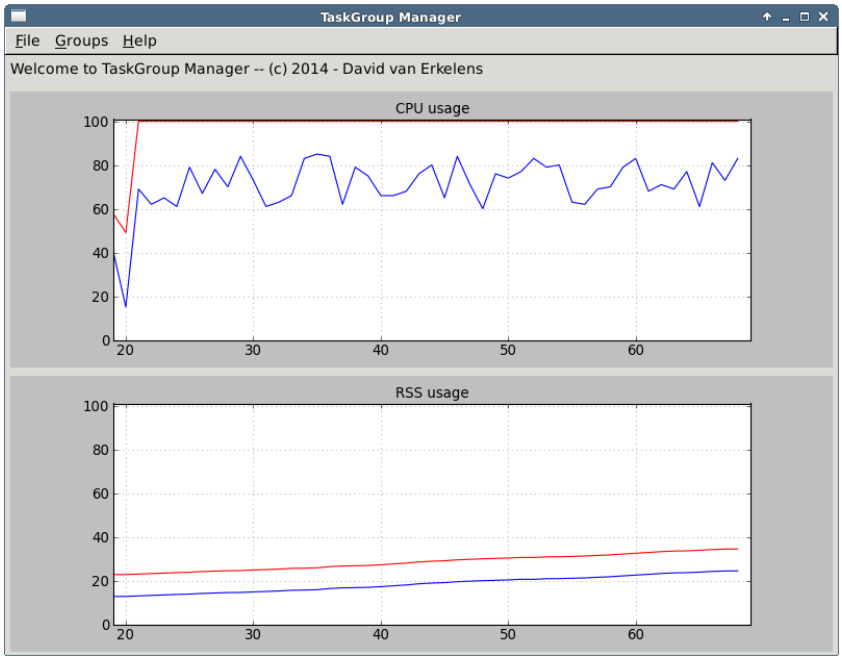

Figure C.3: Behaviour of the program when working with Google Chrome. The blue lines indicate the group containing the Chrome processes, which consumes a lot of the resouces.

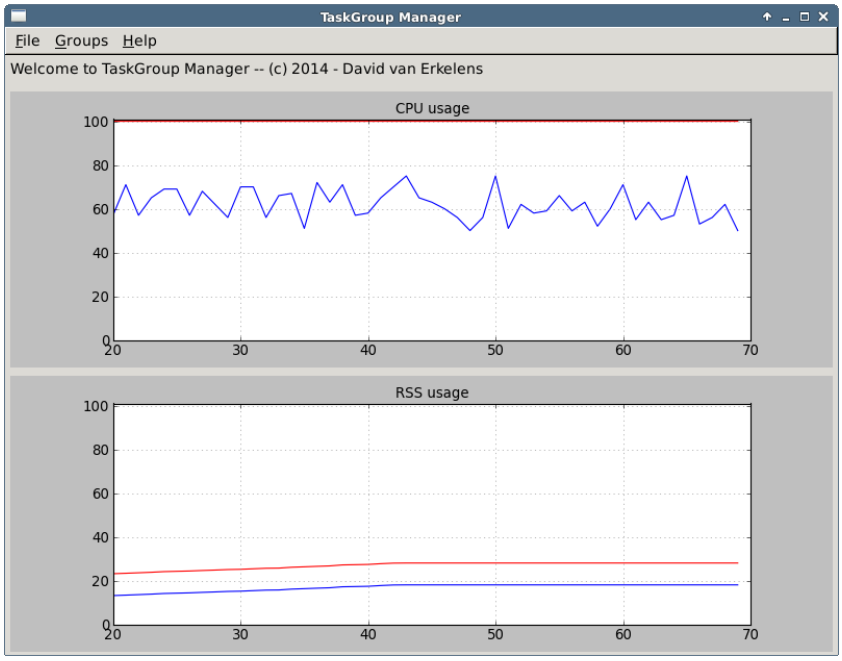

Figure C.4: Behaviour of the program when working with a limited group. When compared to figure C.3, it can be seen that the resource consumption of the group is lower.

C.2 Android front end

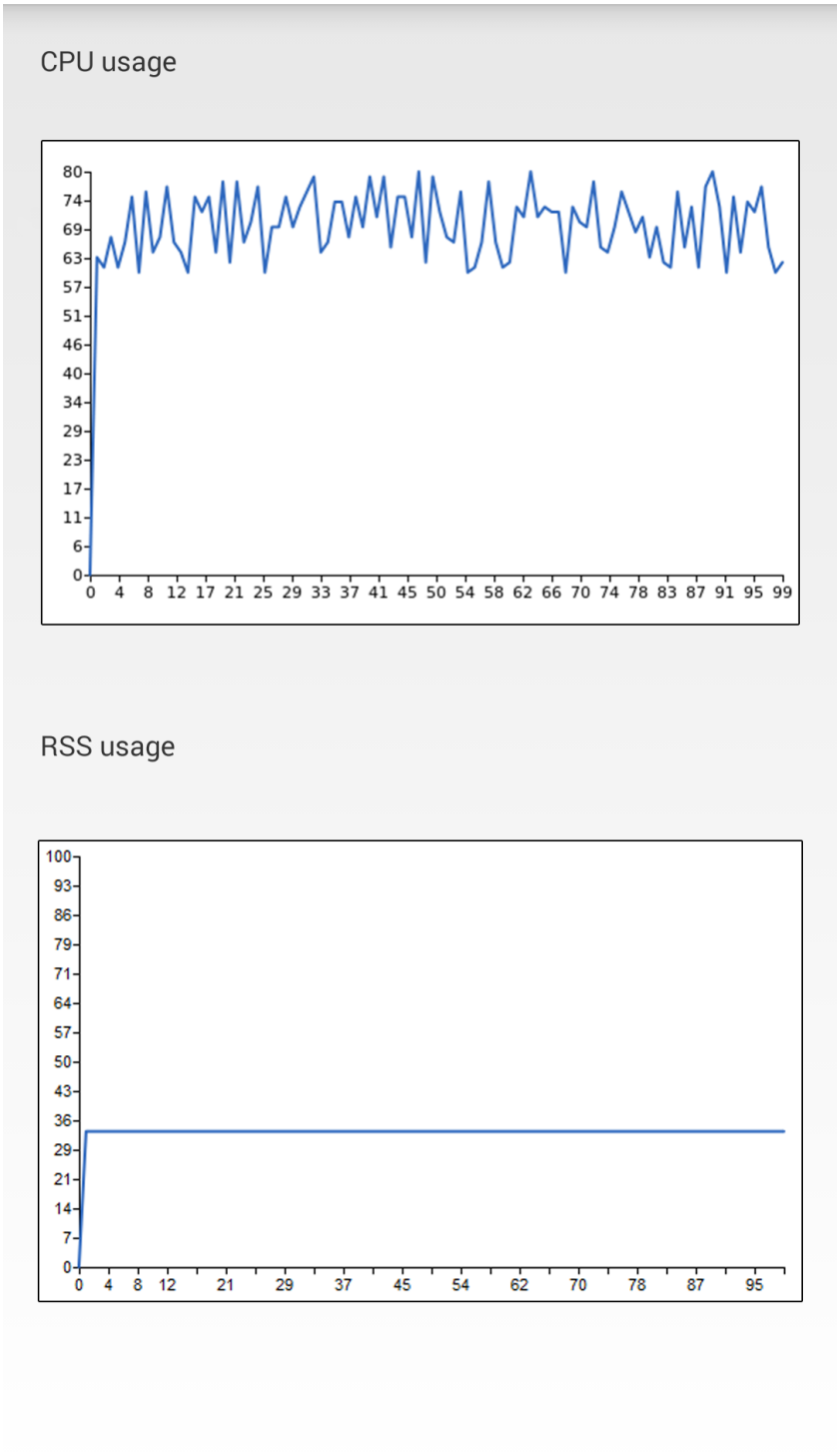

Figure C.5: Main window of the Android front end agent, showing the resource consumption of the entire system.

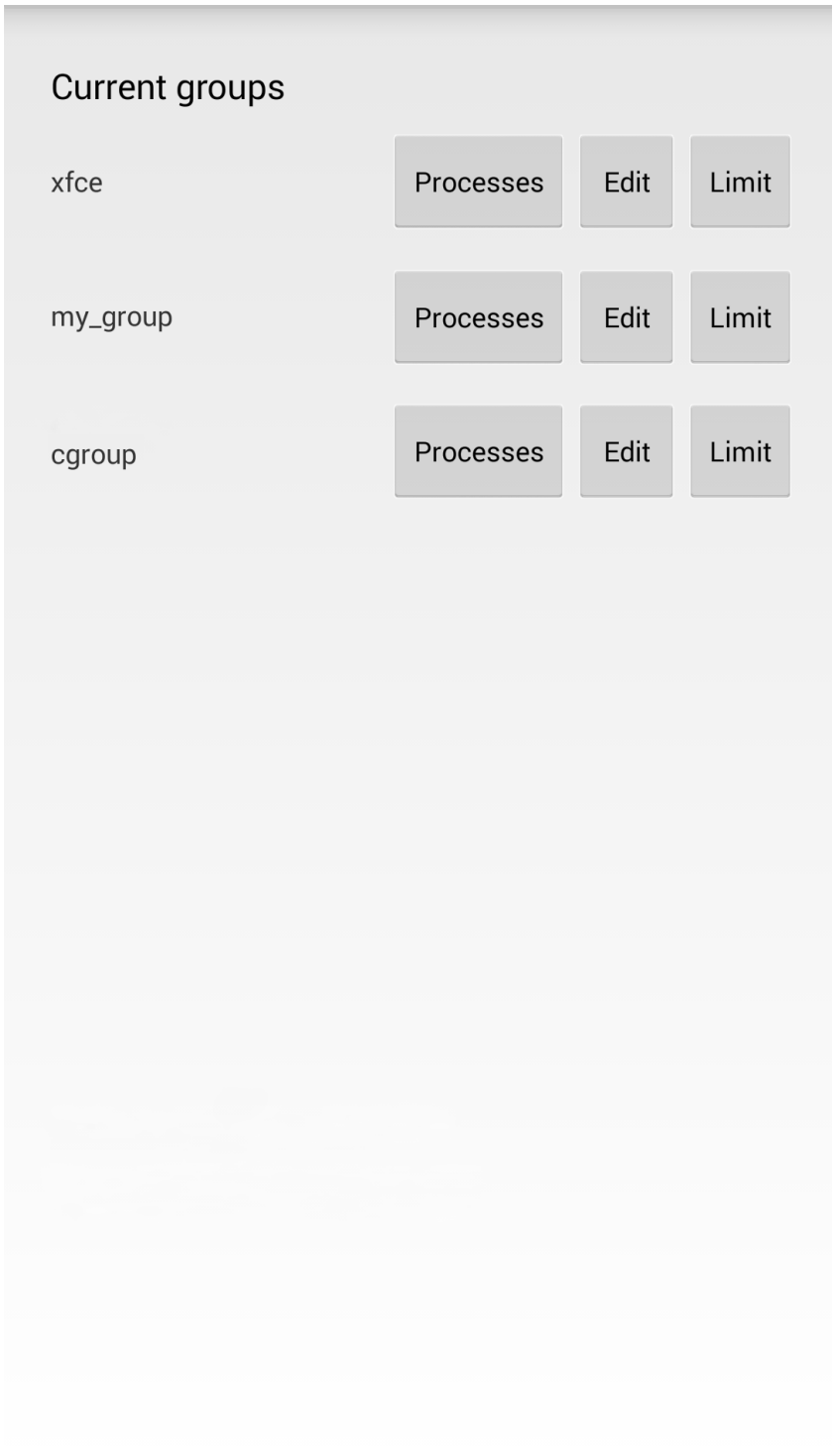

Figure C.6: Overview of current groups in the Android front end agent

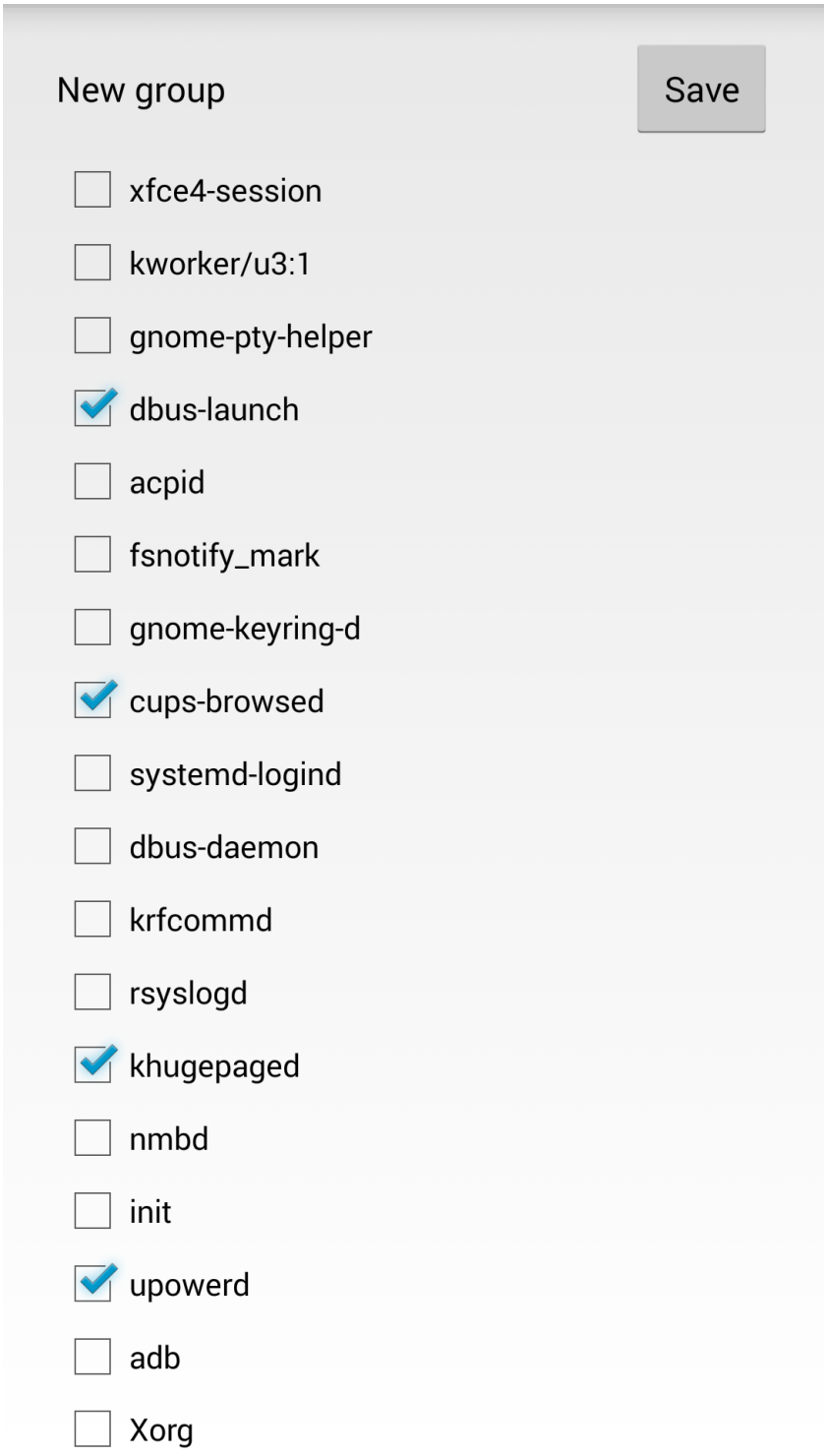

Figure C.7: Creating a new group in the Android front end agent