UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

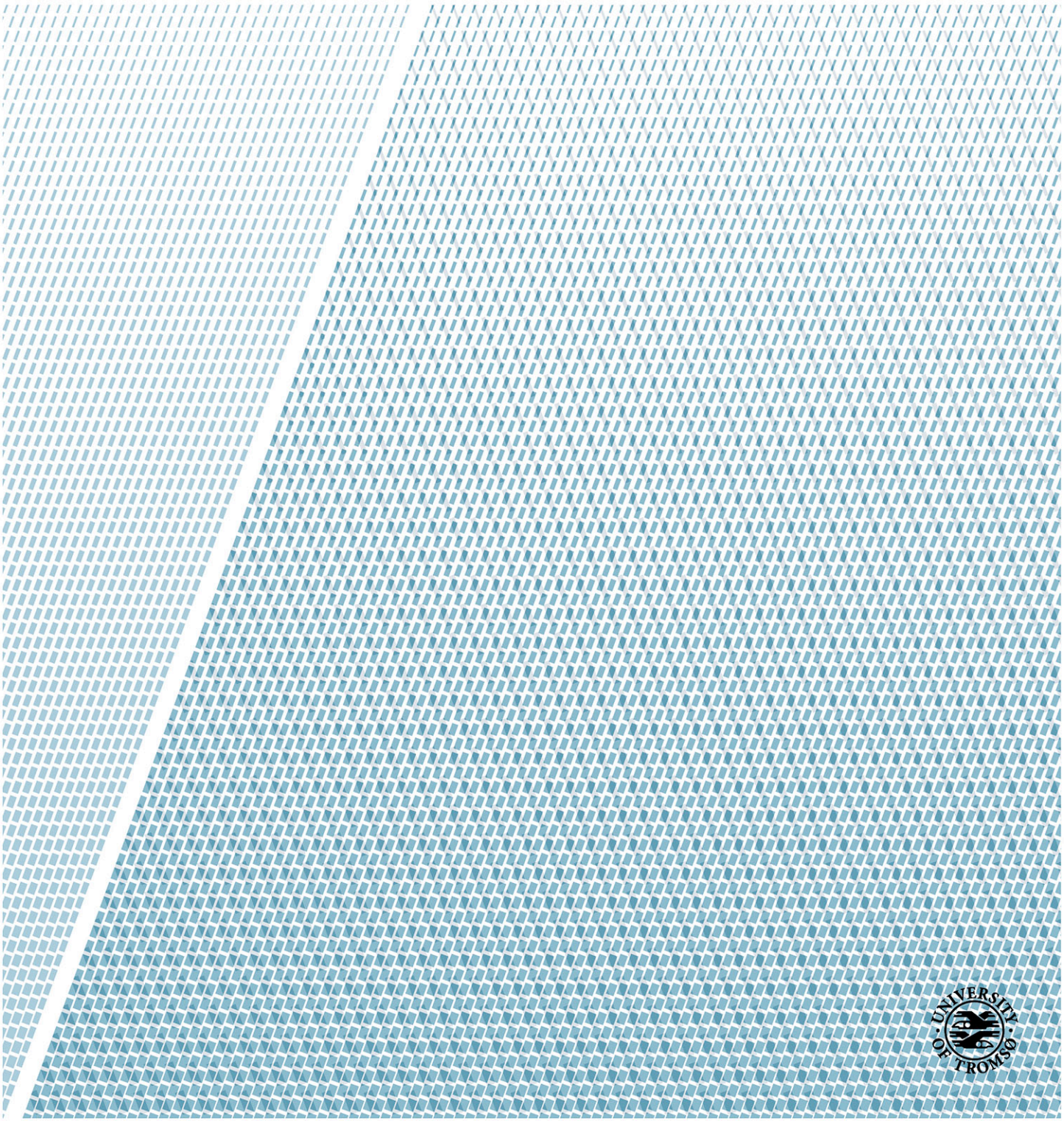
FACULTY OF SCIENCE AND TECHNOLOGY
Department of Computer Science

Simplified Device-Transparent Personal Storage

—

Andreas Kittilsland

INF-3990 Master's thesis in Computer Science
May 2016



Abstract

The number of personal computers we use every day has increased significantly the last couple of years, where the common model is a setup where each device has its own storage with separate files and applications. This forces the user to think in a certain way about files and applications, where they are to a degree bound to a device unless the user specifically moves the files, or installs/uninstalls the applications.
This thesis aims to explore the possibility of changing the way we interact with our files and applications by attempting to sever the connection between device and object (file and application) in the users mind.
In this thesis an option where all devices are aware of each other's files and applications, and are able to run remote files with remote applications is proposed.

Acknowledgment

I would like to thank the following people:

- Professor Otto Anshus, for the idea of a simplified device-transparent personal storage system, and guidance and input through the process of writing this thesis.

- Eivind Schneider, for helping with proof reading and structure.

- My wife, Elise Marie Kittilsland, for encouragement and support.

Table of Contents

List of Abbreviations

DS - Distributed System

FS - File System

GUI - Graphical User Interface

OS - Operating System

SDTPS - Simplified-Device Transparent Personal Storage

SSV - Single System View

List of Figures

List of Tables

1 Introduction

1.1 Problem Context

The number of devices an average person use per day has increased significantly the last decade. Cisco predicts that by 2019 2.5 devices will be connected to the Internet for every person on the earth, which translates to about 5 devices for every person with Internet access [1, 2]. Tablets and smart television sets seem to be the fastest growing category [3].
As the number of devices per person increases, the files belonging to each person becomes increasingly segmented onto different devices.
Usually each personal device has a different use case, and thus the applications and files present on any given device will reflect what that device is used for. For example a desktop or laptop computer is more likely to contain text documents than a cell phone. Also depending on the type of files, numerous copies of the same file may be distributed over the set of devices and over time have a set of different operations done to them. The user ends up with several versions of the same file [4]. For example a picture that is taken on your cell phone, copied to your laptop for editing, and later moved to a desktop for safekeeping and display. And it is not unlikely that the name is changed to fit into some sort of catalogue system in the final step.
Moreover, the example illustrates how the different devices have different use cases, and thus are likely to have a different set of applications. This requires the user to move the files between the machines to apply the operations he/she wants to.
As the research done in [6] shows, there is a need for simplification of the way unlearned users interact with files on multiple devices.

Further more, cloud services are increasing in popularity, though as mentioned in [35], the securities of such services are dubious at best. On-line solutions may not be in the interest of the individual, unless you want governments to have all your information served on a silver platter. And despite quotes by people like Steve Jobs and Vivek Kundra implying that cloud services are the future and local storage is "byzantine", cloud services come with factors of uncertainty for the user other than what is directly related to security; For example possibilities of discontinuation, and the fact that the entire movement is somewhat experimental, treating the customer as a guinea pig. [36]

1.2 Related Work

Several papers have been focused on this topic, attacking the problem from different angles, and applying similar ideas and solutions to similar problems.

Eyo: Device Transparent Personal-Storage is a relevant paper attempting to hinder the segmentation by providing "*device transparency*", aiming to let the user "*think in terms of "file X", rather than "file X on device Y"*". The authors propose a system where metadata of all files are present on every device, and where a version tree is kept to ensure no data is lost on conflicts. The metadata is thus split from the data file. The result is an interesting but rather complex system that works even with some loss of connection, but suffers under huge storage overheads when devices are disconnected over a long period of time. [4]

Amber: Decoupling User Data from Web Applications by Tej Chajed et al also has some interesting ideas, suggesting a system for storing user data and enabling web

applications to query them. Essentially (as the title says) decoupling user data from web applications, letting the users easily use different web services and applications with their data without having to upload a copy of each file to every service/application provider. In practice this means a user could upload his/her pictures to Amber, and then let f. ex. Facebook access the pictures for sharing with his/her network of friends while using an online image processing application from some other provider, and the pictures would not need to be re-uploaded to Facebook after editing. [5]

Perspective: Semantic Data Management for the Home suggests a change in how files are presented to the user. After studying patterns in how the average person interacted with their computer and the file system the authors concluded that a "view", a *semantic file system construct*, could simplify the management of personal distributed storage systems. Presenting the set of files in terms of views similar to how the term is used in relation to databases would let the user interact with their files in the same way as they use them on a regular basis through media players and catalogue interfaces. [6]

The Distributed Personal Computer, a master thesis by Karen Bjørndalen, looks at a design for a centralized distributed system that gives the user access to a set of operations that may be run on remote devices. The set can be extended by implementing and adding the needed functionality in external modules. The author aims to provide functionality similar to that of cloud service without sending the operations out of the network of the users personal computers. [35]

1.3 Thesis Statement

Previous work on this topic such as Eyo has been large and complex, attempting to solve everything at once [4]. The main objective during work on this thesis was to explore simpler possibilities compared to the work that has already been done, as well as introducing new or different elements, such as incorporating applications into the system.
The motivation for attempting to keep the design simple was reduction of complexity and the probability of failure, also simply to explore different avenues than what has already been worked on can be said to be of some importance.

The thesis is in large part based on Eyo by Strauss et al, while taking inspiration from systems such as Amber by using some of the ideas and applying them to the case of a small private network of personal devices. [4, 5]
Thus the objective of this project is to design a simple device-transparent personal storage system, where the user need not keep track of where the files are located. The goal was to reduce the importance of which device the user is currently operating when interacting with his/her files and applications.

Some assumptions are made to make simplifications, and to avoid tackling problems that have already been worked on extensively:

- We assume that there is always connectivity, meaning we wont attempt to handle disconnects and network segmentation.
- We assume that each device has a file system that is accessible for our application.

- We assume that it is possible to retrieve information on applications installed on each device.
- We assume that the OS allows for more than one of the same application to run at the same time.
- We assume that a given file is only touched by one application/user at a time

We also define a set of limitations (or goals if you will) to explore new possibilities:

- Applications run at the device they are installed.
- Objects are accessed at the device they are stored.

These limitations imply that we will not transfer a file or application to a remote device for access or execution, but rather find ways to give remote access while applying the requested operations locally. This means each object is only stored at a single device where the user put it, instead of an arbitrary device decided by a defined set of rules such as in Eyo [4]. Applications can however be installed on multiple devices.

1.4 Contribution

The contribution of this thesis is the exploration of new avenues of distributed personal storage systems with the inclusion of applications, and the possibility of opening and manipulating files in place with an approach that is tailed for each situation, thus reducing resource usage and network overhead. Some slightly different approaches to synchronization are also looked at.
Furthermore, the thesis put forwards thoughts on features for operating systems that could benefit these kinds of distributed storage systems, as well as likely benefit other types of solutions.

1.5 Outline

In chapter 2 a short introduction to technical terms and necessary background information to comprehend the topics discussed in this paper will be given, as well as a short introduction to the tools used for testing and experimentation.
Chapter 3 contains details of the design, architecture, and implementation of the prototype.
Experiments and test results will be shown and discussed in chapter 4.
Chapter 5 is a discussion about the design and implementation done for the prototype. The discussion will be about what could have been better, and what parts seemed to be good.
Chapter 6 looks into what areas have potential for further work, and possibly some overlap with chapter 5 in what areas that need further work.
The last chapter, 7, is a conclusion in which the thesis will be summarized and concluded.

2 Technical Background and Tools

In this chapter, some of the technology and tools used in this project will be explained in a few sentences. I will attempt to explain what they are, and how they are used for the thesis. The use of the technologies will be gone through in more detail when they are mentioned in the elaboration of the implementation.

2.1 Device-Transparent Distributed Systems

A distributed system refers to a collection of components distributed over a network of independent computers, enabling the computers to coordinate and work together to achieve a common goal. A user should perceive the system as a single coherent entity. [7, 8]

When a distributed system is device-transparent, it means the system attempts to hide its distributed nature from the user. There are however various ways in which a distributed system can be transparent. It is commonly categorized into: [8]

- Access transparency, hide how data is accessed.
- Location transparency, hide where resources are located.
- Migration transparency, hide that a resource could move to a new location.
- Relocation transparency, hide that a resource may move to a new location during use.
- Replication transparency, hide that a resource is replicated.
- Concurrent transparency, hide that multiple users may share a resource at the same time.
- Failure transparency, hide any failures and possibly recoveries of resources from the users.

2.2 Go

Go (sometimes referred to as Golang) is an open-source programming language developed by Google. It is statically and strongly typed as well as compiled. Its main selling point is its concurrency mechanisms that let the programmer easily take advantage of multiple cores and networked machines, and its extremely quick compiling. However, some of the drawbacks of Golang is poor debugging tools and the necessity of a bit bucket. [9, 16]

Development was prompted by frustration at Google at how "clumsy and slow" software development was with large software systems. The language has exploded in popularity over the last year (2015), particularly in Asia. [10, 11]

Most of the programming done for this thesis (building of the prototype) is done in Golang.

2.3 JavaScript

JavaScript is the world's most popular programming language, developed by Netscape for the Navigator 2 browser. It is commonly used to write client-side code for websites as all the popular web browsers support JavaScript out of the box. [12, 13]

It is used to write client-side code for the HTML-based GUI in the prototype.

2.4 QML

QML is a user interface specification and programming language, allowing the programmer to easily create cross platform GUIs for applications. It is often used to write graphical interfaces for applications with a backend written in more efficient

languages such as C++. But also Go has modules for use with QML, called Go-QML created by Gustavo Niemeyer. [14, 15]
It is used to create a GUI for a text editor application to go with the prototype of this thesis.

2.5 HTML

Hyper Text Markup Language, or HTML for short, is a language used to express web documents consisting of "elements" defined by "tags". The first HTML version came out in 1991, and there has since been many iterations improving on what has become the standard markup language on the Internet. [31]
HTML is used to describe most of the GUIs used for the prototype in this thesis.

2.6 TCP/IP

"Transmission Control Protocol/Internet Protocol" is a set of communication rules used for connecting computers over the Internet. It contains definitions for how information should be formatted and sent so that the intended receiver can get and comprehend the package of information. Bob Kahn and Vint Cerf developed it in 1978. [17]
The majority of communications in the prototype of this thesis is done over TCP/IP.

2.7 HTTP

Hypertext Transfer Protocol, or HTTP for short, is the protocol used the World Wide Web. Hypertext is text structured into nodes with logical links between them, where the hypertext commonly is HTML documents in the case of HTTP.
It is a stateless protocol, implying that two consecutive commands are independent and have no knowledge of each other.
The protocol is built as a layer on top of other protocols, usually TCP, but UDP can also be used. [32, 33]
In the thesis, HTTP is used for communication between the GUI described in HTML and scripted with JavaScript, and the prototype backend written in Golang.

2.8 Wireshark

Wireshark is a piece of free software designed to analyse network traffic. It is a cross-platform open-source software based on libpcap for capturing network packets. The software can be retrieved from Wiresharks webpages [26] for free. [27]
It is used for capturing packets sent between devices during testing of the prototype in this project.

2.9 Network Link Conditioner

The Network Link Conditioner is a preference pane application created by Apple included in the "Hardware IO Tools for Xcode" package. The application let's you set bandwidth, percentage of packets dropped, and delay for both up and downlink, as well as DNS delay. It is designed for developers to test their applications in poor network conditions. It affects the network throughput and delay by changing firewall settings and thus only work when going through a network interface other than loopback as of OS X 10.10 due to changes in firewall software. The tool can be retrieved at Apple Developer Downloads page [25]. [24]
It is used to emulate poor network conditions during testing of the prototype.

2.10 Routing Table

The routing table is a set of rules that determine where information during communication over IP is directed. When a "packet" is received, the destination is looked up in the routing table to figure out where to forward the packet. [23]
This table can be edited, letting the user determine where packets destined for specific IPs should be routed. By adding a rule, a packet that is sent from a process on one machine to a different process on the same machine can be forced to travel over the network instead of loopback. Doing so enable us to simulate the two processes being on opposite sides of a network, and forces the traffic through the local network device which lets us capture and manipulate it more easily. During testing of the prototype, the routing table was altered to let local traffic go over LAN via the router and back. My machine is at 192.168.1.138 in the LAN, and by default retrieving the rules for that IP returns (route -n get 192.168.1.138):

route to: 192.168.1.138
destination: 192.168.1.138
gateway: 127.0.0.1
interface: lo0

As the reader can see, the interface is lo0 as in loopback, but after adding a rule to the routing table for that IP to go via the router at 192.168.1.1 (route -n add 192.168.1.138 192.168.1.1), the interface is no longer lo0, but en0 as in Ethernet: [20]

route to: 192.168.1.138
destination: 192.168.1.138
gateway: 192.168.1.1
interface: en0

2.11 Table of Processes

The table of processes program, or top for short, retrieves and displays sorted information about processes on the system. It is often bundled with Unix-based operating systems. It is useful for retrieving statistics of CPU and memory usage of the system in its entirety or specific processes.
The program retrieves samples every second (by default, but can be changed). It then finds the difference between the samples for the properties where this is necessary. This means that for items such as CPU percentage, at least two samples must be taken. During testing of the prototype, CPU percentage and memory usage was retrieved by running "top -stats pid,ppid,cpu,mem -l 2" every third second. The command runs top retrieving only PIDs, PPIDs, CPU percentage, and memory usage for each process, and takes two samples before returning. [21]
The process status program, or ps for short, is also bundled with OS X and provides a lighter instant snapshot of most of the same properties, but is inaccurate for a property such as CPU percentage as it is not possible to calculate from a single sample (the CPU is either in use, or not in use). It instead returns a decaying average from over the last minute or less. This means that top is likely to be more useful for the purpose of retrieving statistics during testing as is done during this thesis. [22]

2.12 System Profiler

The system profiler is an application bundled with OS X that creates reports on system configurations of both hardware and software. The output can be in both plain text and XML format. It was used to retrieve a list of installed applications on devices in the prototype of this project, by running "system_profiler SPApplicationsDataType". Note that the application also comes with a GUI, System Profiler, which can be found under Utilities. [29]

2.13 Power Management Settings

Power Management Settings, or "pmset", is an application in OS X for managing power settings. Amongst other things, it lets you get the percentage of battery left, set automatic restart, time idle sleep etc. It is used to get the battery percentage in the prototype of this project. [30]
Power Management Settings was used to retrieve the battery percentage value in the prototype of this project.

2.14 Tailored Tools

A number of tools were written either for testing and evaluation, or debugging. A short list of the tools and their use is presented here. Their implementation will be explained more in detail in the next chapter.

2.14.1 SimpleTxt

SimpleTxt is a small and simple text editor, built to go with the prototype to show how the system can cooperate with application. The application has a GUI with some limited functionality; editing text, making text bold, making text underlined, and making text italic. The application also allows for opening and saving of files. All of the same functionality is implemented to work with the distributed system too.

2.14.2 Overview GUI

The overview GUI, or control panel, presents a view of the prototype system as a whole. It contains an excerpt of the GUI of all running daemons and their combined CPU and memory usage. A link to each daemons files and applications interface is given together with the excerpt of the corresponding daemon.
There is also functionality for changing global variables such as; how long to log values counted towards coordinator selection, how to weigh the different variables used in coordinator selection, and how often to evaluate the coordinator. The interface will only update when refreshed. The number of daemons will be recounted as well when the refresh request is received.

2.14.3 Logger

The logger is a testing tool. It finds all running daemons, and logs their individual CPU and memory usage. It takes samples every three seconds until it receives a <stop> command, or for the number of seconds passed as an argument when the tool is started. A dump of the values logged for each daemon is created under logs/<date and time>, together with a simple HTML document showing any noteworthy statistics in graph format. The HTML output only shows values above 0.0, while the dumps contain all data logged .txt files.

2.14.4 File Adder

The file adder is a tool for testing the prototype system. It can be used to continuously add files to the system. It takes three arguments; target daemon, refresh interval (in file additions), and number of files to add. Number of files is an optional argument; if it is empty it will run until a <stop> command is received. Type of synchronization or refresh can be set after start by typing <apps> or <objs>.
Examples:

- <./file_adder daemon_8591 10 200>
  Adds 200 files to daemon with port at 8591, and sends a synchronization request at the beginning, and after every 10 adds. Refresh returns when objects are synchronized.
- <./file_adder daemon_8593 1>
  Adds files to daemon with port at 8593 until it receives a stop command, and sends a synchronization request at the beginning, and after every single add. Refresh returns when objects are synchronized.
- <./file_adder daemon_8593 1
  apps>
  Adds files to daemon with port at 8593 until it receives a stop command, and sends a synchronization request at the beginning, and after every single add. Refresh returns when both objects and applications are synchronized.

2.14.5 The Application List Creator

The application list creator is a tool that emulates the system profiler, but instead of returning the actual list of installed applications, returns the list given to it. It makes it easier to test the prototype by feeding various amounts of applications to include in its single system view of applications. The list it returns is whatever is present in the list.txt file, where a new line separates each application.

2.15 Shell Scripts

A number of shell scripts were written to simplify certain processes. They are included in the source code.

2.15.1 Running the Prototype

The file run.sh starts a set number of daemons with specified coordinator mode, which communicates over LAN if the routing table has been set as specified under the "Routing Table" section (It will use the IPs ip_self and ip_to, set in the support_communication.go file of the daemon). Example of uses:

- <./run.sh 5 auto> Starts 5 daemons with automatic coordinator selection.
- <./run.sh 10 fixed> Starts 10 daemons where the coordinator is always at the first daemon started (daemon with port at 8590)

The script local_run.sh does the same as run.sh, just that it uses loopback instead. It takes the same arguments. To exit the prototype again, stop.sh kills all segments of the system. This works for both run.sh and local_run.sh.

2.15.2 Building and Cleaning

The build.sh script builds and compiles all source code in the project, even all independent packages. Similarly, clean.sh removes the binaries built by build.sh.

3 Approach and Implementation

3.1 The Idea of a Shared File View System

The chief objective of the thesis was to create a personal wide view of all files, and thus a "device-transparent personal storage system". To explore the concept, the idea was to create a system that for all devices works with the files in-place at their location, instead of transferring the files to the device being used. In other words a "shared view" of all data files. In addition to this, the idea includes the thought of creating a shared view of apps/application files. An app or application overlay if you will. In sum the idea is a shared file view system, a system where the location of files and applications are of little to no importance to the user. The goal is to let the user merely need to think of what device is in close proximity, or possibly reflect on what device fits the nature of the tasks at hand. The goal of a shared view of files and an overlay of apps and applications somewhat overlap. It would for example be difficult to create a shared view of data files where they are worked on in-place, and not moved, without a way of opening applications on the remote computer. Also, applications are after all files too.

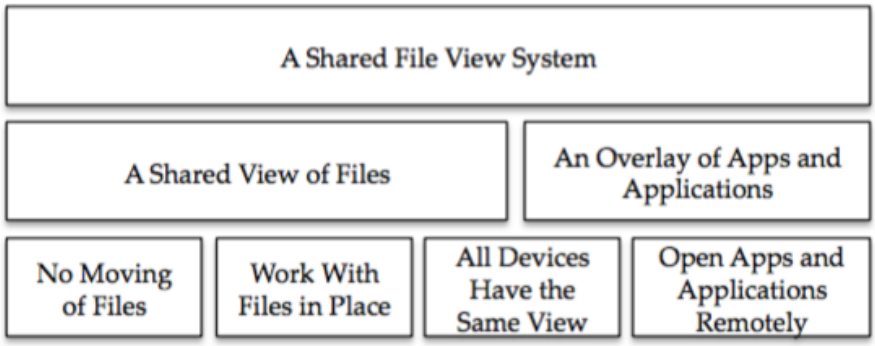


Figure 1 The concepts idea overview

3.2 Architecture of the Device-Transparent Personal Storage System

To achieve a shared file view system, we first of all need a "System-to-User" output module and vice versa so that the user can interact with our system. Next we need the output module to have access to a single system view of all the personal devices to create a personal wide view for both applications and objects. The module creating these views then obviously needs to know what applications and objects we have access to. Similarly we need a module allowing for operations entered from the interface, which also needs access to the objects and applications. Thus architecture similar to this is proposed:

Architecture

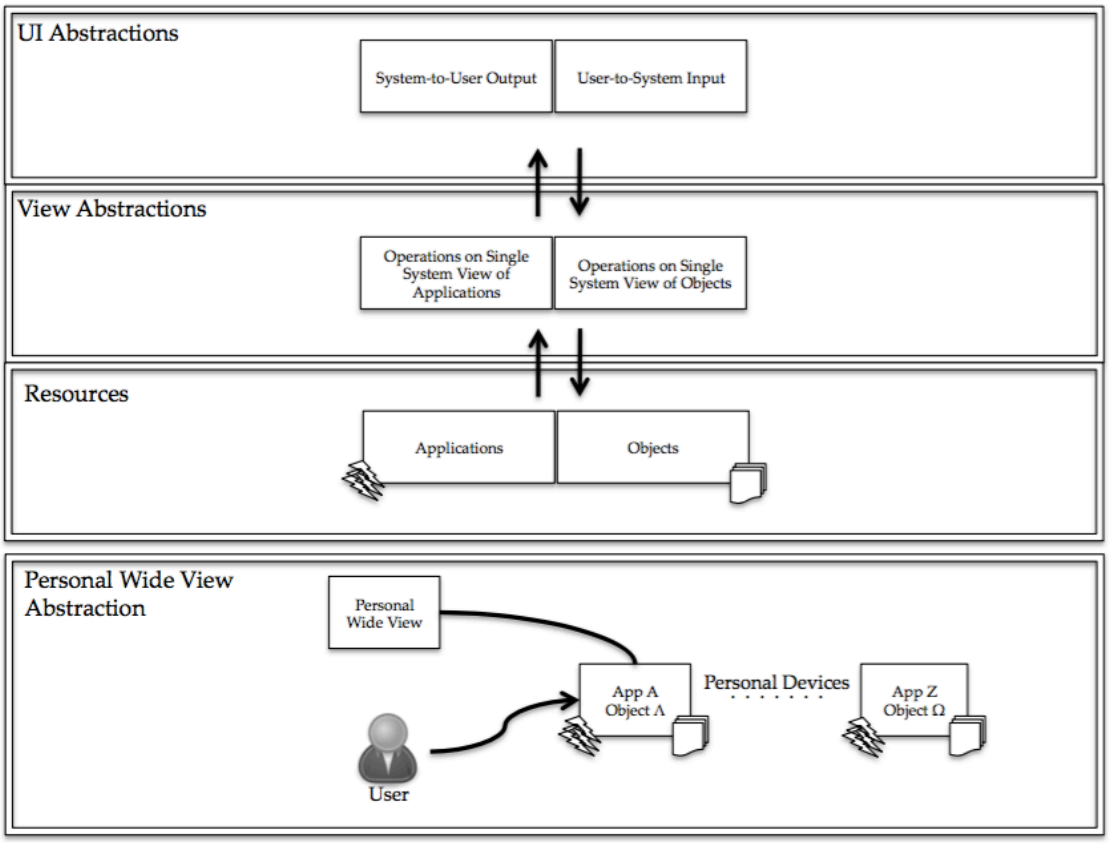


Figure 2 The concepts architectural overview

3.3 Design

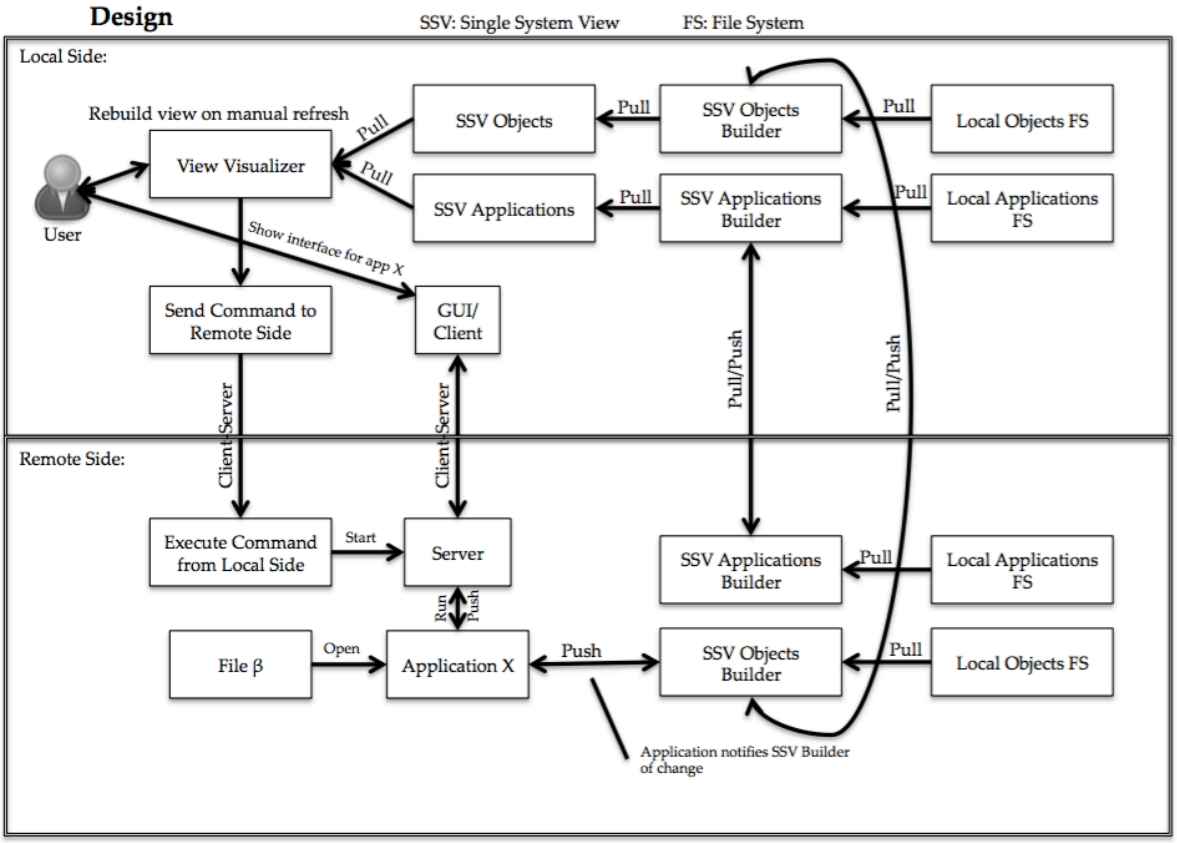


Figure 3 The concepts design overview

To achieve the described architecture, we need to create segments that can run on each device and build the "single system view" based on the local view of objects and applications, as well as information retrieved from other devices on what objects and applications those devices have. These segments of our system may then run on each device, feeding the GUI with information on the state of the objects and files in the network of personal devices. The user with access to this GUI should then through some local module be able to open applications with a specific file on remote devices, from which an interface of this application should be accessible on the device the user is interacting with. Any changes after operations on any files should also be detected and reflected in an updated SSV. A design such as shown above should then achieve the architecture defined in the previous section.

3.4 Implementation

The prototype essentially consists of four pieces; the overview GUI, the daemon, the change tracker, and the text editor application. All individual pieces are written in Golang, at least backend. Most of the GUI markup is in HTML, with the exception of the text editor where QML was used. HTML is used as it is easy to implement a HTTP-server in Golang with the standard library and thus the simplest way of creating a GUI, and QML as it is one of the most fully implemented GUI libraries for Golang at this point with good cross-platform compatibility (writing a text editor in HTML would not fit in a realistic scenario). Some JavaScript is used for client side coding to go along with the HTML GUI, which will be explained further later. JavaScript is of course used, as it is the

easiest and most commonly used client-side scripting language that more or less every browser supports out of the box.

The overview GUI is used just to get an overview of the prototype while running. It contains a view of all the running daemons it can find with links to each daemons GUI. It is not an integral part of the prototype and is only used to help get an overview during testing.

The daemon is the core of the prototype, where most all of the operations and communication is done. It is not implemented as a proper daemon in this prototype, but has a normal process.
The change tracker is a process that keeps track of updates and changes received for local files, either from the local GUI, or a remote device. It then feeds it back formatted to the daemon.

The last piece, the text editor, is an example application used to illustrate how a remote access system could be done while keeping the file in place during operation on it.
The figure below shows how all the pieces (except the overview GUI) fits together.

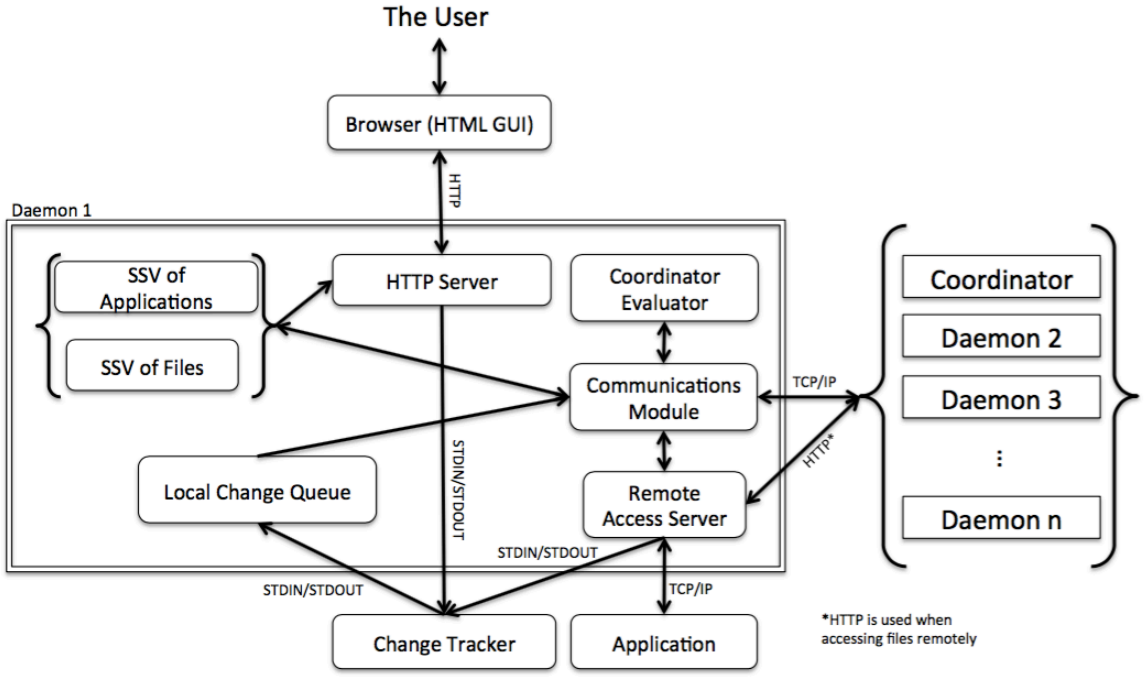


Figure 4 Implementation overview

Some controlling variables, such as max size, and a number of variables used for coordinator evaluation can be set in a file common to all the individual pieces of code, config.cfg. The most important of which is the maximum size of the network. This is the limit on how many ports the prototype should range over to look for daemons, thus keeping it as low as possible increases efficiency. The rest of the variables will be detailed later in this chapter, as they are also accessible through the overview GUI.

3.4.1 Implementation of the Daemons Core

To implement the system in a way that can operate seamlessly, having a daemon running on each device would be a good idea. This daemon can then operate as the core

of the system and exchange information with each other. One of the daemons is a coordinator, whose role is to enforce consistency across the network. All updates and changes pass through the coordinator when the user requests an update list of files.
It was decided to implement the daemons for the prototype in Go, as its concurrency mechanisms were thought to be useful for this project, and the language is designed in such a way that it is easily readable. The cross-platform compatibility of the language is also a very useful property.
The prototype implementation lets each daemon generally communicate in three ways: TCP/IP with other daemons, HTTP with the browser and thus the user, and standard streams with some child processes that will be explained further in the next section. Each daemon has a list of all the other daemons present in the network and has the potential to communicate with them, but direct communication has been kept to a minimum. Letting information pass by the coordinator to ensure consistency across the network was seen as a priority, though this consistency is in conflict with efficiency. This communications design is not retain in most scenarios requiring high efficiency with low latency however.
When started, each daemon is either given a port, or looks for an open port while also asking around for a coordinator. When found, the TCP/IP communications platform is initialized and a connection to the coordinator is established. The coordinator informs the rest of the network of the newly joined daemon. No information on specific files or applications are exchanged before the user gives a refresh/synchronization request, as shall be looked at further in the next section. But the coordinator sends a vector containing all files ID and their version number to the new daemon. When a refresh request is made, all daemons are told to check their vector and requests updates on any deviant tuples. The refresh operation can then run, in which all new daemons receive a full application list from the coordinator.
If connection is lost to a daemon, the metadata on all its files remain in the system, but the files cannot be opened until the daemon reconnects.

3.4.2 Implementation of a Single System View of Files

The core functionality for each device is incorporated within the daemon process, but functionality for keeping track of changes was separated out into its own process, from here on out referred to as the "change tracker". The change tracker communicates with the parent process, the daemon, through pipelines (standard streams). It is however possible that other methods of communication could be better, such as shared memory. Splitting the change tracker into a separate process might not have been necessary in the final implementation. Initially the thought was to let the change tracker communicate with the underlying OS, and possible other change trackers or daemons. However, the state of most operating systems more or less makes scanning the local files necessary if it is to keep in direct touch with each file. Optimally, functionality of subscribing to changes on each file would let the device tracker do its job. As this was not an option, the applications must inform the change tracker of any changes, so that it may inform the daemon. All other changes are applied through a GUI provided with the prototype, using HTML. This allows the prototype system to track changes and operations without having to scan the entire files system, but also limits the user to using this GUI, as any changes done directly in the OS would not be seen by the prototype. It also limits the number of applications that may be used to those that either has been designed to work with this system, or the libraries of the applications are altered to run via our code.

The design of this is not the optimal one for the current situation, but such a design could be a good way to do it in the future, given that OS designers improve on their solutions in a similar fashion to what is mentioned in this thesis. The topic will be discussed further in the discussion section.
The prototype has a flow similar to what is shown on the graph below.

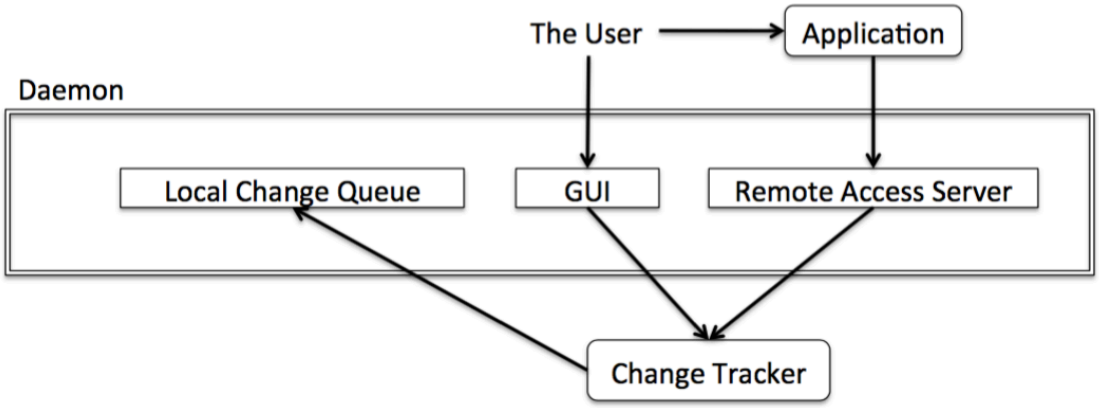


Figure 5 Change tracker flow in the prototype

When a local change has been detected, the change is put into the local change queue. The change queue contains all updates and changes done to local files since the last synchronization in chronological order. When the user requests a refresh/synchronization operation, the daemons pop the changes in the queue, and push them to a coordinator daemon one by one. The coordinator daemon then ensures that there are no synchronization conflicts, and apply the update locally. When the update is successfully applied in the coordinator, the change is forwarded to all the other daemons, including the initial daemon. All daemons then have a local queue of changes, containing all changes applied to the files local to their device. The design ensures that all changes are applied in chronological order on any given file, but it does not promise that changes on files from different devices are applied in chronological order.

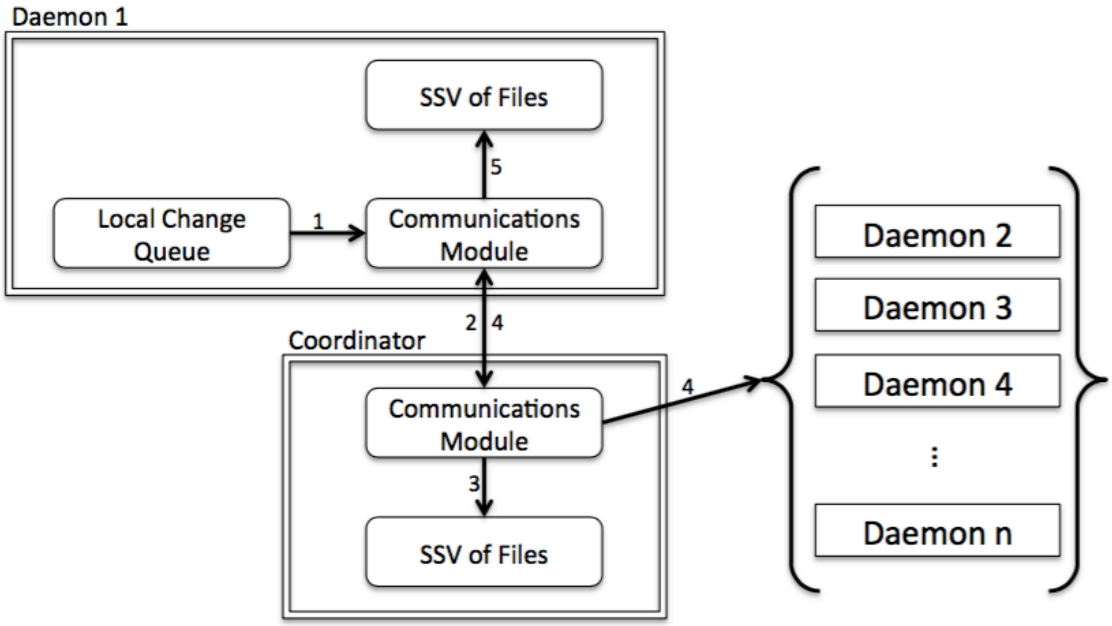


Figure 6 File synchronization flow

3.4.3 Implementation of a Single System View of Applications

The application single system view is implemented in a slightly different way than the SSV of files. This implementation reflects an alternative design, giving a different set of tradeoffs. For Applications, whenever a request for synchronization is received, the daemon queries the OS for installed applications. In the prototype, this is done using the system_profiler [29]. The system_profiler with argument "SPApplicationsDataType" compiles a list of applications and returns it to the daemon through the standard output pipe. The daemon then compiles a list of removed applications and new applications since the last synchronization. The list is then sent to the coordinator. The coordinator has a structure containing all the applications in the network and the number of devices that a particular application is present on. The coordinator will go through the received list; subtracting one from the removed list, and add one to the new additions list. If there are any applications that now are present on zero devices, the application is removed from the global applications list. If the global applications list has a new application, or has lost an application, an updated list is sent to all the daemons that update their own global list of applications. This means that each device has two lists of applications; one list of local applications, and one list of global applications. The global applications list is what is showed to the user in the GUI as the SSV of applications.

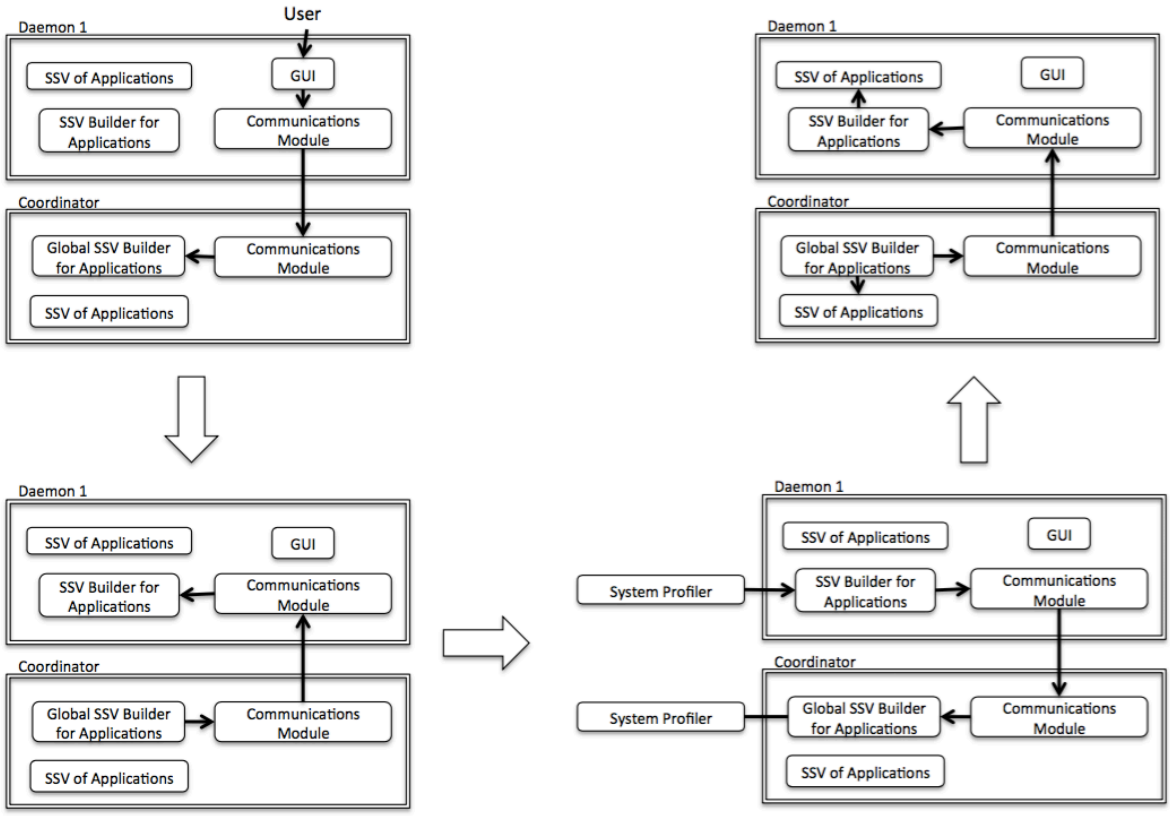


Figure 7 Application synchronization flow

The illustration above shows how a refresh/synchronization request is processed when synchronizing the SSV for applications. The request is first sent to the coordinators SSV builder, which has the task of putting the lists together. When the request is registered, it is forwarded to every daemon in the network, including the original sender. All the local SSV builders then retrieves the list of applications present on the local device, and sends the list of removed or added applications to the "global SSV builder". The lists are here put together, and checked for changes since previous synchronization. If a change has occurred, the new list is distributed to the daemons. All communication is done through TCP/IP, with the exception of the initial request from the GUI, which is by HTTP.

3.4.4 Implementation of Remote File Access

The HTML GUI of each daemon allows for opening the files present in the SSV. If opened, the browser is forwarded to a page with a simple HTML and JavaScript text editor. When the user presses the open button, the daemon checks if the file is local. If it is local, the file is opened. If not, a request is sent from the local daemon to the coordinator, which asks around for the file. When the daemon that has the file locally receives the request, a server and the appropriate application with the file open is started. A URL is then sent from the daemon with the file locally, to the original daemon that requested the file. The daemon then forwards the users browser to that URL, which leads to the HTML and JavaScript interface that the server on the device with the file now has set up. The server then asks the application for the visible excerpt of the file being worked on, which it forwards to the client. The client can then edit the excerpt, or change the view and get a new excerpt. If the client makes a change, a client side JavaScript is continuously running checking for changes. If it finds a change, the smallest possible subset of the excerpt that was changed (or just the requested operation) along with the position of the change, is sent back to the server. The server forwards the change to the

application, which applies the change in place. The file then never moved from its original location, and only a small excerpt of the file was ever transferred to the users device.

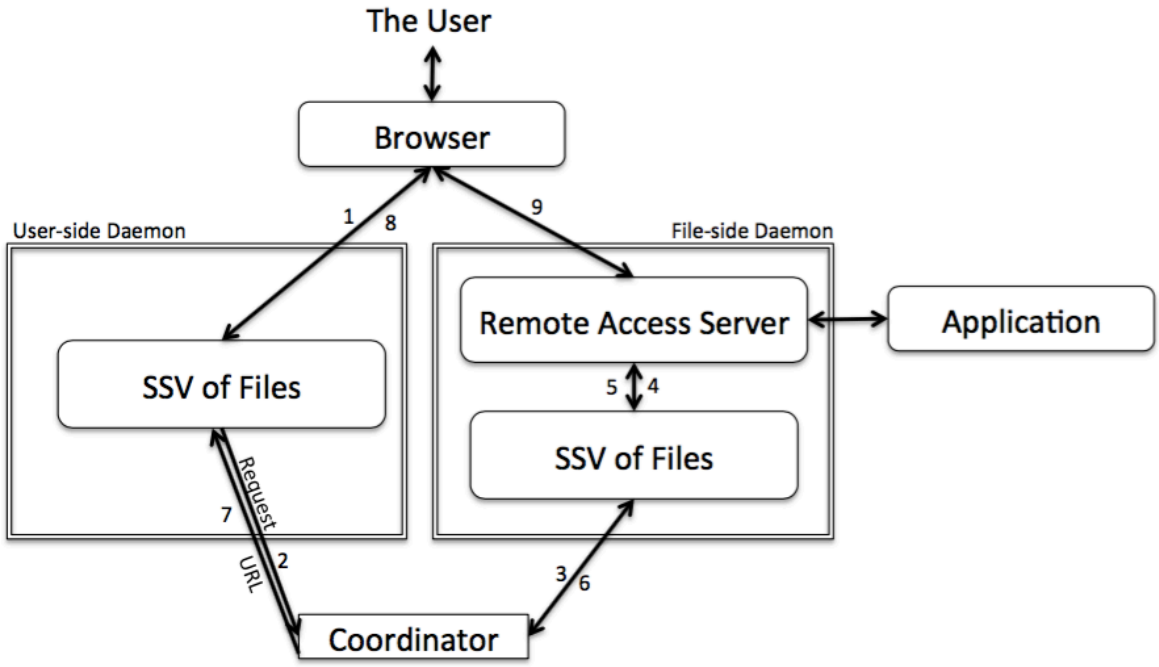


Figure 8 Remote access flow in the prototype

Optimally, the SSV of applications would be integrated with the remote access system. In the prototype this is not the case, as it has merely been implemented for a simple text editor that was made specifically to illustrate the functionality. The application is a simple text editor, using HTML as mark up. The backend is written in GoLang, with the GUI in QML using GoQML to connect them. The application has been called SimpleTxt, and works with what I've called ".stxt" files. This means that the interface will only open files with that extension.
Note that the HTML and JavaScript interface implemented in this prototype is not fully functional. It has several bugs caused by differences in the way the markup of the loaded file is formatted in the of the HTML GUI and the text box of the QML GUI. But it does illustrate how such a system could work for text files.
The update rate of the remote access system is set to 500 ms. This should possibly be user changeable. A higher update rate increases accuracy, but also increase CPU and network utilization. But with a better algorithm, the update rate would probably be less important. However, an update rate of 500 ms does not mean that data is transferred every 500 ms, but rather that the JavaScript checks for changes in the textbox every 500 ms, and sends the change if any is found. The topic will be looked at further in the experiments section.

The communication between the server and the application is done through TCP/IP in the prototype implementation, but this should most certainly have been changed to using standard streams. The application and the server is always on the same machine, thus using TCP/IP is completely unnecessary.

3.4.5 Implementation of the Coordinator and Coordinator Election

Which device is coordinator can have a huge impact on performance, as we will look more into in the experiments section. Thus a module for automatically selecting a coordinator can be of some use (though retaining the possibility of manually selection a device to be coordinator is of some importance). It was suspected that having most of the files on the device with the coordinator would be most efficient, as less communication overhead was likely. Other factors of course also was thought to count, such as what device is what device is being interacted with the most by the user.
Most personal devices today are also mobile, thus run on a battery. So to avoid having the coordinator die, or potentially eating what little battery remains, it was decided that battery percentage should count in too.
The election algorithm gives each device a score, based on the mentioned factors; battery percentage, number of local changes, number of application runs, and number of files. The battery percentage is retrieved using "pmset", with the argument "-g batt" [30]. If the device is found to be charging, the percentage is considered as 100 %. If the battery level is below a set threshold, the battery percentage is that devices score. If it is above the threshold, the score is the threshold + number of changes * a + number of runs * b + number of files * c, where the constants are weights given to each variable. The constants in the prototype are set to a = 5, b = 10, c = 1. We will however see in the experiments section that these numbers are likely to be suboptimal for this prototype. They do however provide an example.
The number of runs and local changes are based on a record from a set number of minutes back in time, set to 5 minutes back in time by default in the prototype. It is however changeable.
When the score is calculated, the variables of the other devices in the network is retrieved from the other daemons, and the score compared with their own score. The device that found its own score as the highest proclaims itself coordinator, and notifies everyone else. The daemons that did not find their own as the highest score simply wait for the new coordinator to tell them about the change. The previous coordinator keeps track of all the on-going coordinator jobs, and tells the new coordinator whenever it is done with all of the jobs that were already started. No new jobs are accepted during the transition period. When the transition is done, all daemons are notified, and work may continue as usual.
The process of evaluating the coordinator is then repeated in a set number of minutes or seconds, variable that should be changeable for the user as well. It is set to a mere 5 seconds by default in the prototype.

3.4.6 Implementation of the Overview GUI

The overview GUI, or control panel, was as mentioned in the introduction included to get a better impression of how the prototype is working, and to have easy access to each of the individual daemons GUI. It attempts to connect to all local daemons within a set range from port 8590, and retrieves their PID and the URL to their GUI. The interface has a table showing a small excerpt from each of the daemons GUIs through HTML iframes, with a link to the full GUIs.
It has also got some simple, but somewhat inaccurate stats on CPU and memory usage of the daemons combined; all retrieved with ps.
Some of the user changeable variables, like coordinator evaluation weights and evaluation intervals, can be set in the control panel. They are also settable in the config.cfg file.

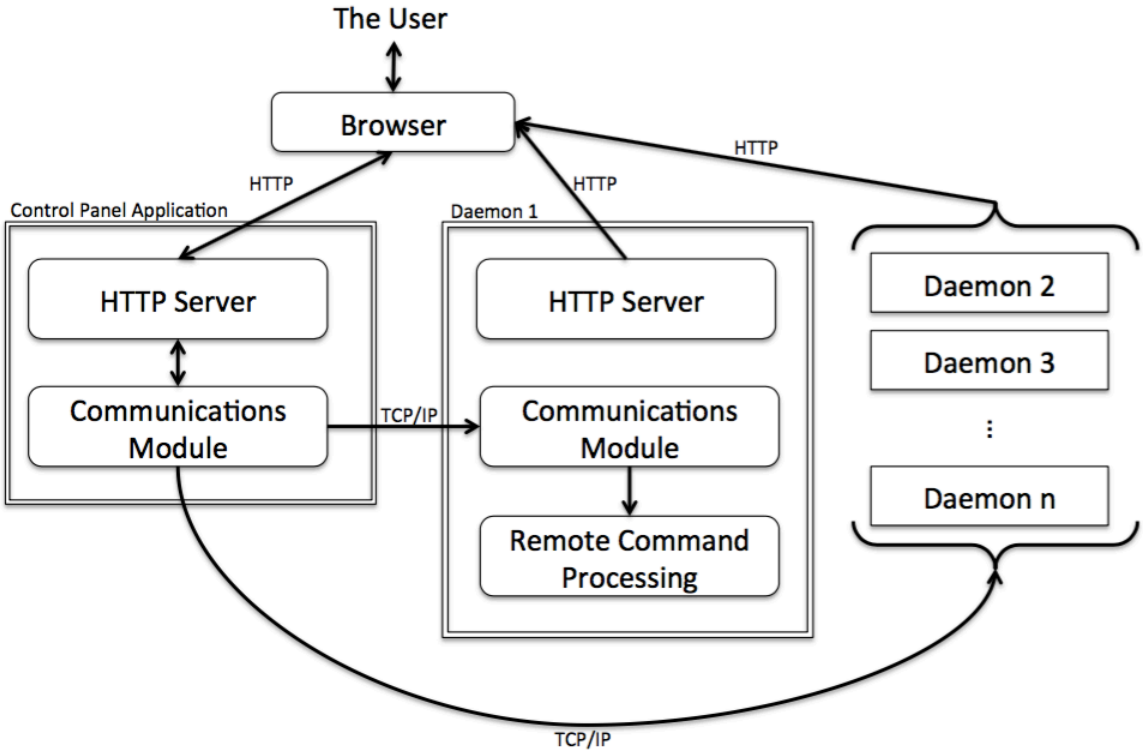


Figure 9 Flow of the control panel implementation

3.5 Implementation of Other Tools

3.5.1 Implementation of the Logger

The logger application can be run while the prototype is running locally, and it will record CPU and memory usage until it is stopped or the prototype stops.
It is implemented in much the same way as the overview GUI, in that it checks for running daemons by attempting to connect to all ports within the used range. It then asks for the PID of the daemons it found, and then logs the resource usage of processes with either a matching PID or PPID. The values are retrieved using ps to get an instant snapshot. This does mean that particularly the CPU usage is not entirely accurate, but give and indication. When the logging is stopped a HTML document is created based on the data recorded, and the data is dumped to text files where the data is split into a file each "device".

3.5.2 Implementation of the File Adder

The file adder is an application that takes an input file, and adds it a given number of times to the prototype network via a given device/daemon.
It opens the input file, which is whatever file is called "testfile" in the application folder, and recreates it in the folder specified for the given daemon with a new name. It then connects to the daemon via TCP/IP and sets an add request which is the processed the same way as an add request received from the GUI. The process is repeated until the limit on files per refresh/synchronization request is reached, and the request is sent. When the synchronization is done, the process is repeated until the set number of files has been added.

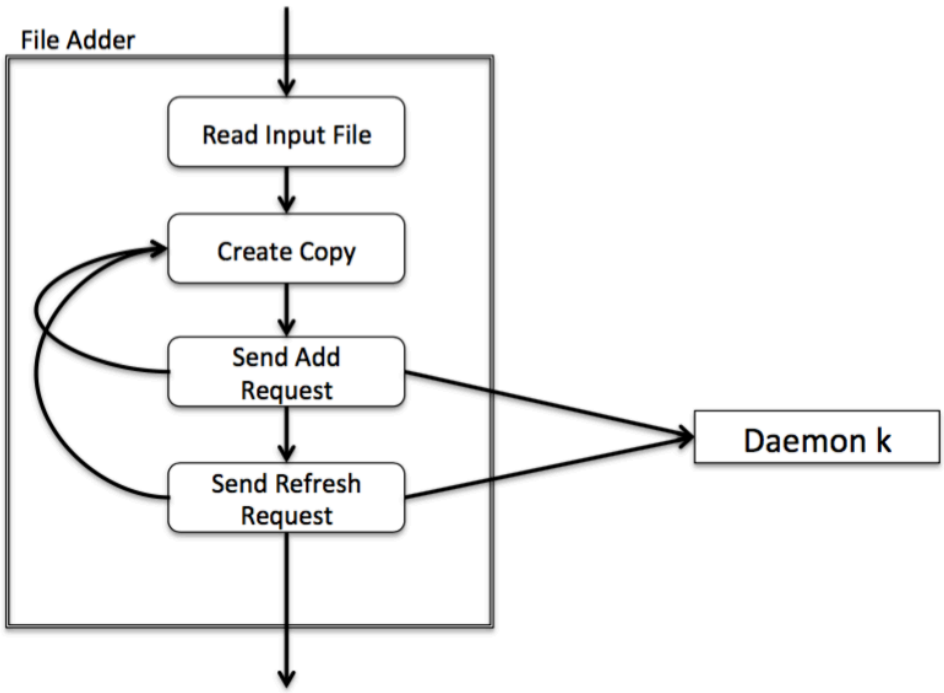


Figure 10 Flow of the file adder application

4 Experimentation and Evaluation

4.1 Methodology

4.2 Environment

To evaluate the prototype and pinpoint flaws in it, some experiments were made. They were run on a MacBook Air 13-inch, from mid-2012. The machine has an Intel Core i5, dual core with 1,8GHz clock frequency, 3MB shared L3 cache, and Hyper-Threading. The memory size is 4GB in 1600MHz DDR3L. It was running OS X El Capitan v. 10.11.4 at the time of the testing. Go version 1.2.1 darwin/amd64 served as compiler, the GUI was displayed using Safari version 9.1 and Chrome version 50.0.2661.94 (64-bit). [19]

The experiments were executed by running the devices on the same machine but in different processes. Each device consists of a central process, the daemon, which spawns other processes as needed. However this means that they use the same resources, and communication between them goes over loopback. As running all traffic between the segments would give significantly lower communication overhead than realistically possible, traffic between the "devices" was sent to a router on LAN and back again by editing the routing table. The router was an Asus RT-N56U operating at 192.168.1.1, and the traffic is being transmitted over Wi-Fi. The machine was at 192.168.1.138, thus adding a route for 192.168.1.138 to 192.168.1.1 would force the traffic out over the network instead of going over loopback when using 192.168.1.138. Thus running the following command in terminal "route -n add 192.168.1.138 192.168.1.1" adds the aforementioned route to the routing table. [20]
This means the delay and traffic is doubled as each packet crosses the network twice. Thus my results in these experiments are likely to be worse than what could be realistically expected. However it also means that we can use Apples Network Link Conditioner to limit traffic further, and Wireshark to capture traffic between the devices. Wireshark is set to snoop on packets over en0 using TCP on the port range of 8500 to 9500 only in these experiments.
To retrieve memory and CPU usage a small application that runs top every three seconds with two samples was written. It connects to the daemons on TCP/IP over loopback at the beginning of each loop to check that they are still up and retrieves their PIDs. It then runs top to retrieve a full list of processes with their PIDs, PPIDs, CPU percentage, and memory usage. When the list is retrieved, it finds all lines with PIDs or PPIDs that are one of the daemons PIDs and logs the retrieved stats for that line. The problem with this approach was that when system resources were close to being exhausted, this application was affected and slowed down. The long intervals also means we wont get more detailed information than at the three seconds level. It would be possible to get samples more often with ps, but the values seem less accurate as mentioned in chapter 2. [21, 22]

4.3 Metrics

4.4 Running Idle

In this experiment, CPU usage, memory usage, and traffic was measured while the prototype was running idle with no data or application files, nor any other interaction.

2, 5, 10, 15 and 20 devices networks where tested, where 20 devices proved to be the upper limit of what the test machine could handle with the automatic coordinator enabled.
However, when the coordinator was fixed to the first daemon, absolutely no CPU usage was measured, meaning it was less than 0.0 % (out of 400 % on the dual core CPU with Hyper-Threading) of CPU time.
The following graphs show the measured values during the experiments, where 20 devices has its own graph due to the high and some times missing values. A projected value of 400 % has been plotted for intervals where top was unable to run due to exhaustion of resources, though it is of course not completely maxed out in reality.
NB! The graphs in this section have different scales for readability.

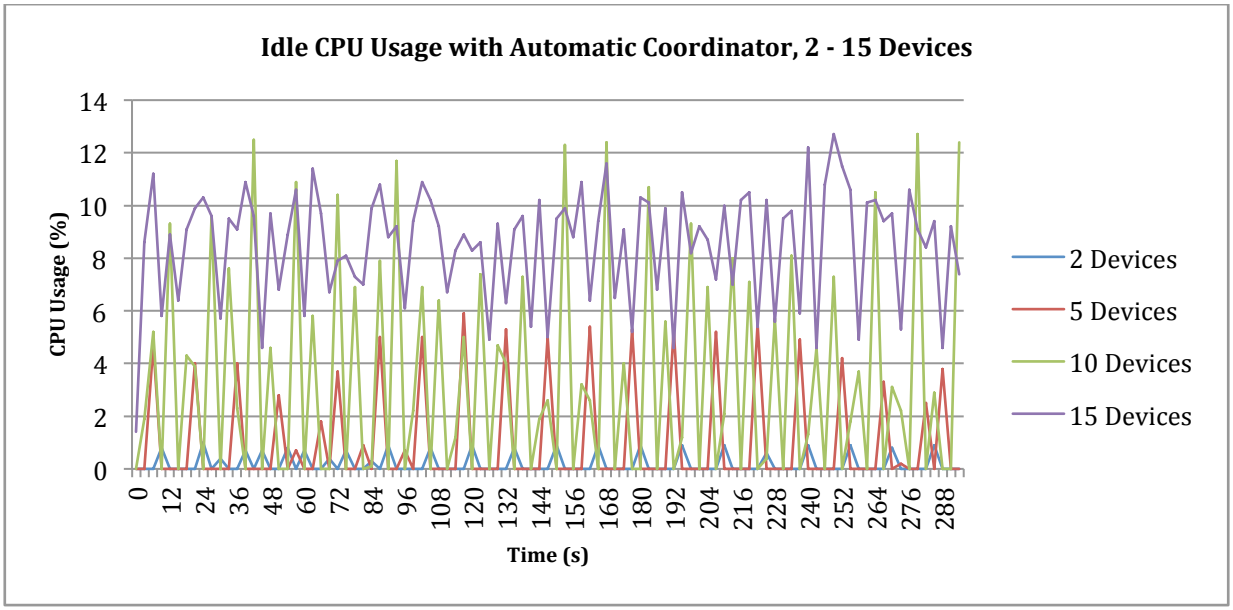


Figure 11 Idle CPU usage for 2 - 15 devices.

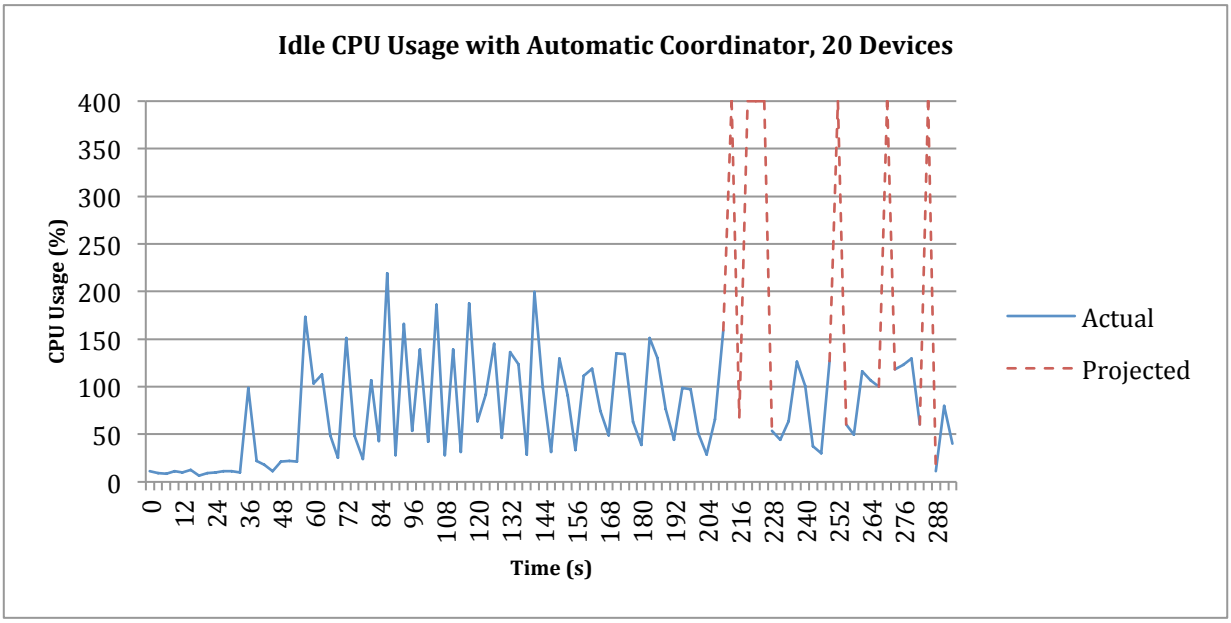


Figure 12 Idle CPU usage for 20 devices.

TODO...

4.5 Add and Refresh

In this experiment some devices were set up with no files in their system. Files were then added one by one, five by five, and hundred by hundred, meaning a refresh was done between each one added, each fifth add, and each hundred add. The three experiments was done several times with a few variations in environment and settings, changing one variable a time from what was set as the "default" settings; Fixed coordinator to one daemon, refresh returned when files/objects are synchronized (before applications are synchronized), with each file being a small sized text file of 365 bytes, and the network conditions being a normal LAN Wi-Fi connection. All variations were done twice, once with two devices running, and once with ten devices running. The statistic for the experiment with ten devices are not presented in its entirety here as they were not very successful; quickly reaching the limit on concurrent open files (256), and even with the limit increased (4096), resources were exhausted quickly.
The default file was decided to be so small due to space limitations on the hard drive as the size should not affect the performance of the prototype.
The files were added by a small application written in Golang, opening any file in its folder with the name "testfile" and creating a copy in the folder designated to the daemon it was to add the files to. The daemons change tracker was then notified by this application, and the daemon received a refresh request when each round of changes were finished. Both communications operations were done using TCP/IP locally (loopback).

4.5.1 Two Devices

Two devices where set up and the files were added to the device that was not the coordinator at the start (the device with the daemon running at port 8591) until it broke down under the following environments:

4.5.1.1 Default Setup

The coordinator is set to be the device with the daemon at port 8590, with the other non-coordinator, or profane daemon if you will, running at 8591.
Communications between the daemons go via the LAN and the router with no further limitations. The file added is small and of size 365 bytes, and the refreshes return as soon as the files has been refreshed.
During the testing with one file per synchronization, the prototype synchronized 1297 files before crashing when synchronizing the 1298th. It crashed with error “fork/exec /usr/sbin/system_profiler: resource temporarily unavailable”, meaning it was unable to retrieve a list of applications installed on the local machine. The error means that the application has reach OS Xs limit on how many processes it can have running at the same time, probably because we move on to add and synchronize another file before the previous synchronization process is entirely complete, meaning we keep stacking up processes over time.
When adding five files before each synchronization, as many as 3483 files were successfully synchronized before a crash stopped it. This time a segmentation fault occurred when attempting to access a map structure. This could imply that a mistake or some bad Go-programming practice was committed leading to some unexpected behaviour in rare cases (it was not obvious to me how the segmentation fault had occurred).
During the test adding 100 files and then synchronizing, the limit on concurrent open files was increased from 256 to 4096. It ran for about 20 minutes to synchronize 20000 files at which point it was stopped.

The following graphs show CPU usage in percentage, memory usage in kilobytes, and bytes sent and received per second over the course of the test. Note that the timestamps on the 100 files test for CPU and memory are a bit off as top took a bit longer then expected during some samples, and the logging started after the daemons had set up.

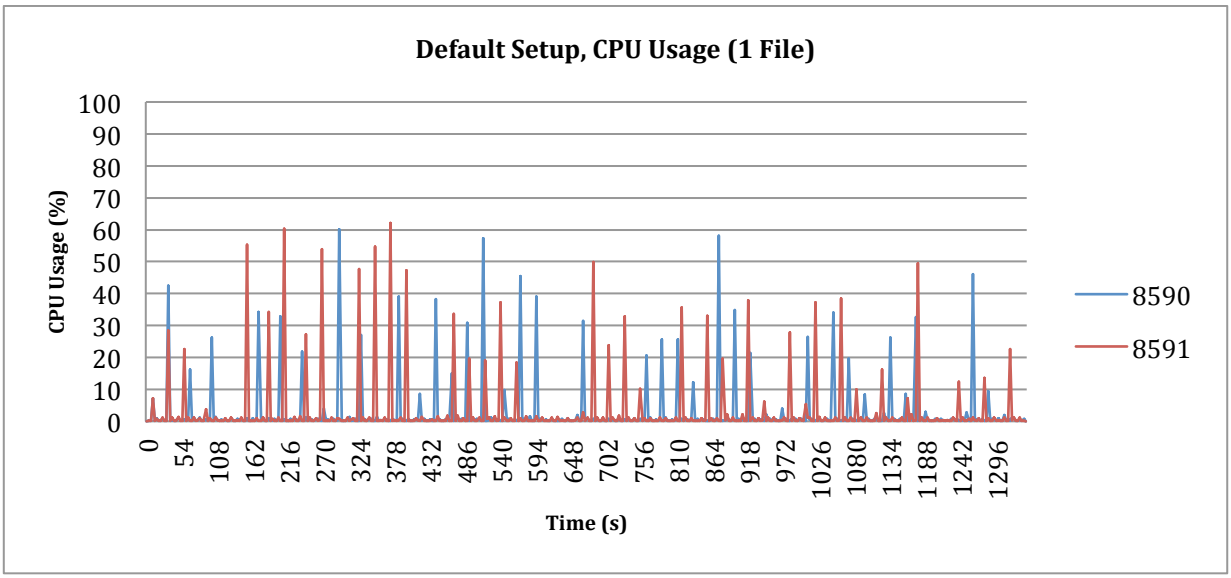


Figure 13 Default setup, two devices - CPU usage when one file added per synchronization

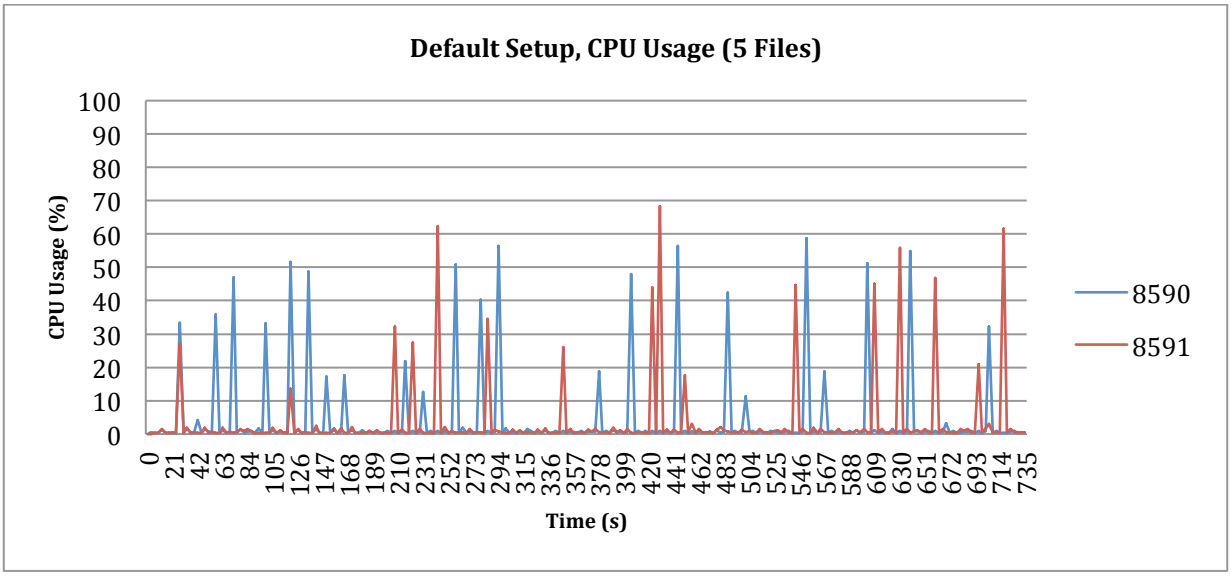


Figure 14 Default setup, two devices - CPU usage when five files added per synchronization

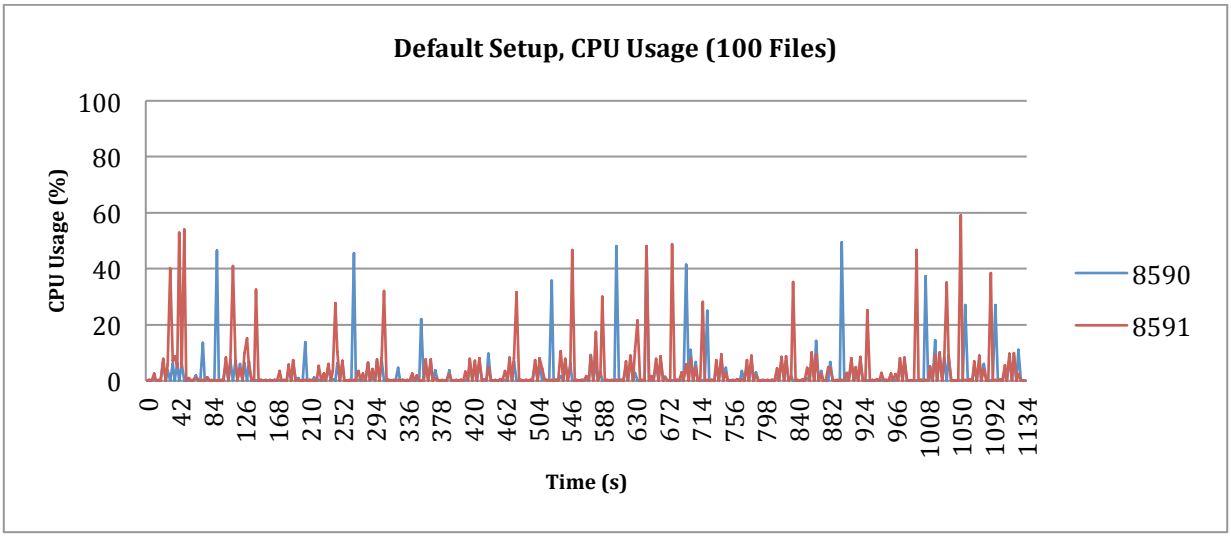


Figure 15 Default setup, two devices - CPU usage when 100 files added per synchronization.

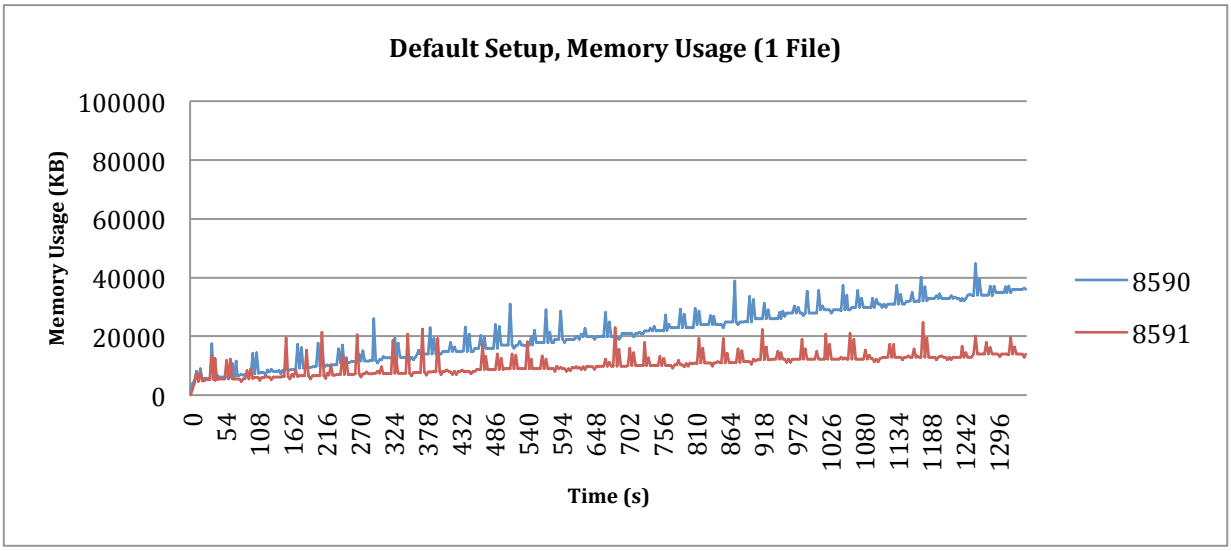


Figure 16 Default setup, two devices - Memory usage when one file added per synchronization

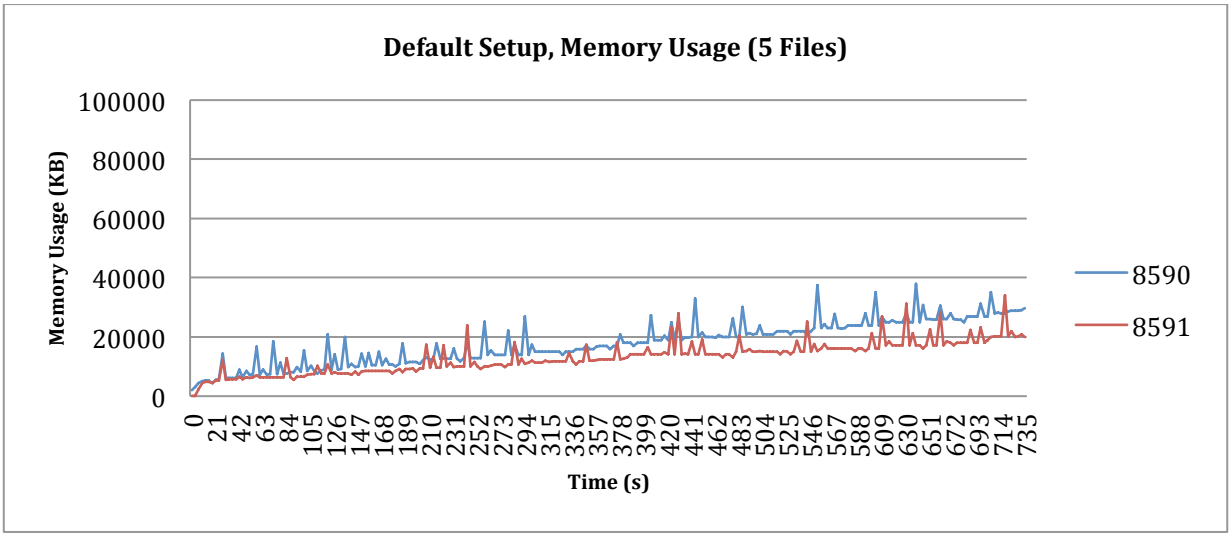


Figure 17 Default setup, two devices - Memory usage when five files added per synchronization

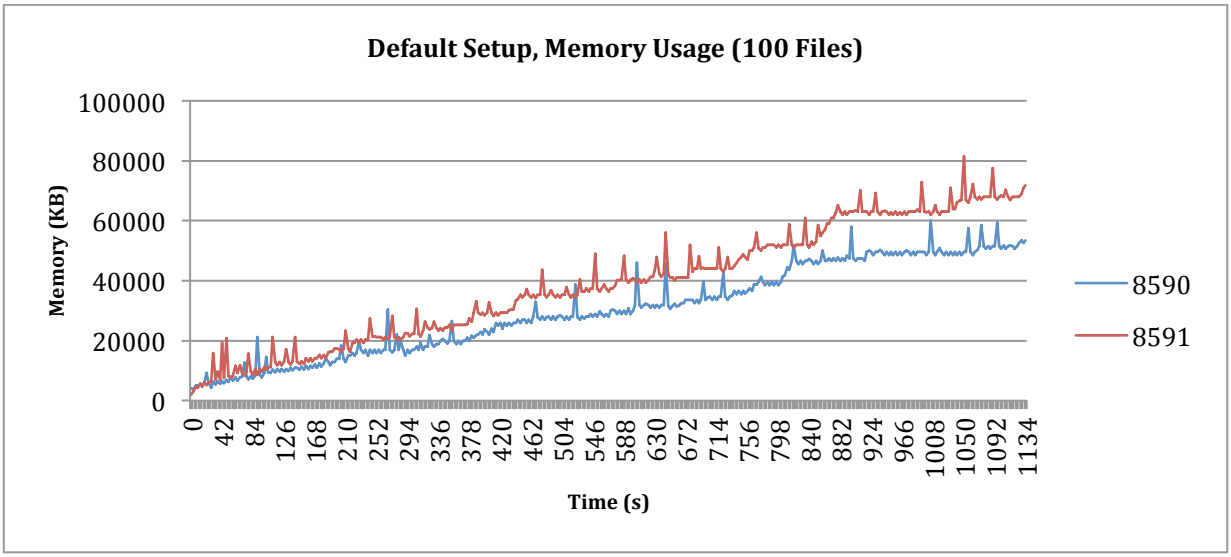


Figure 18 Default setup, two devices - Memory usage when 100 files added per synchronization

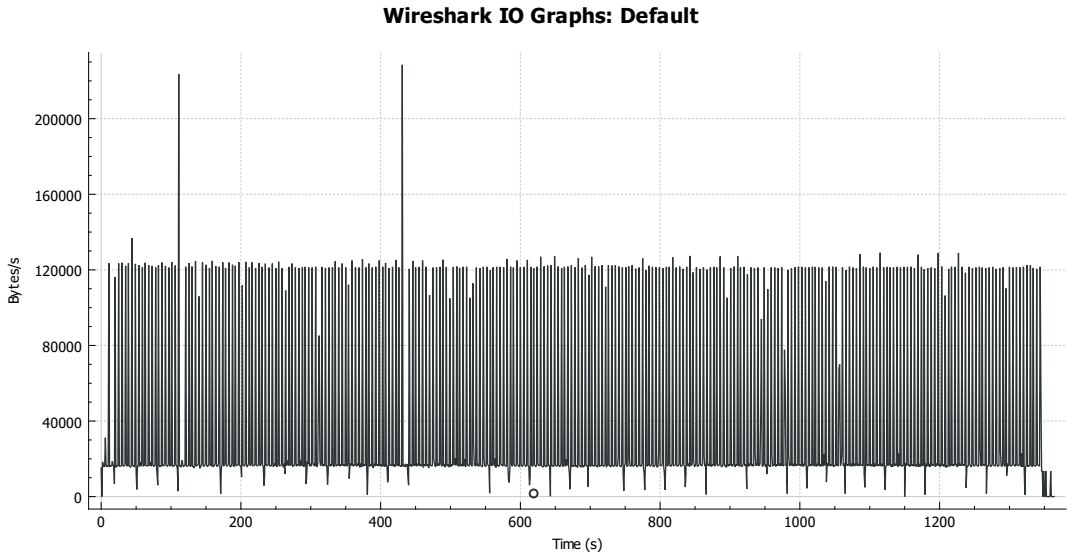


Figure 19 Default setup, two devices - Network traffic when one file added per synchronization

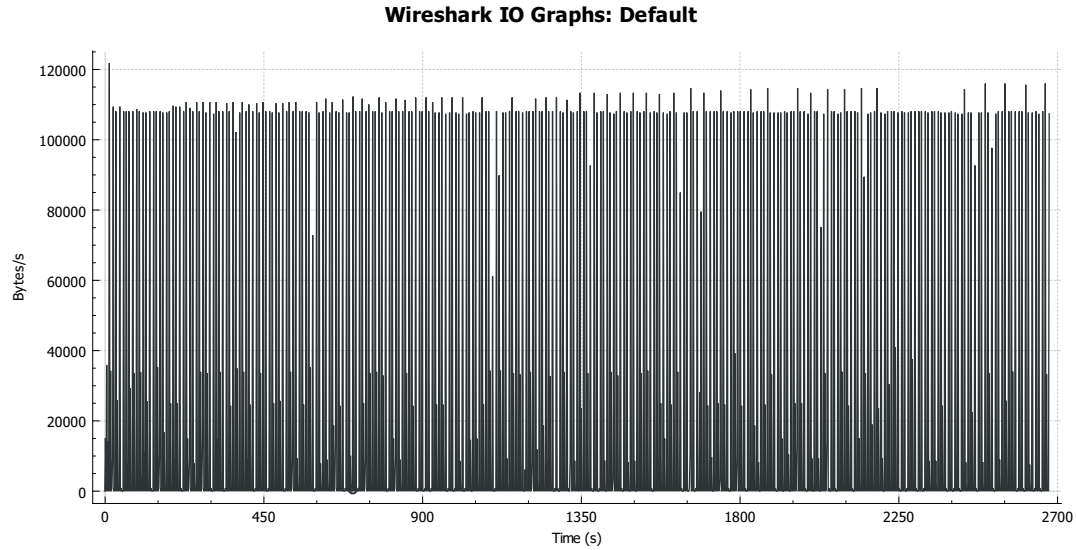


Figure 20 Default setup, two devices - Network traffic when five files added per synchronization

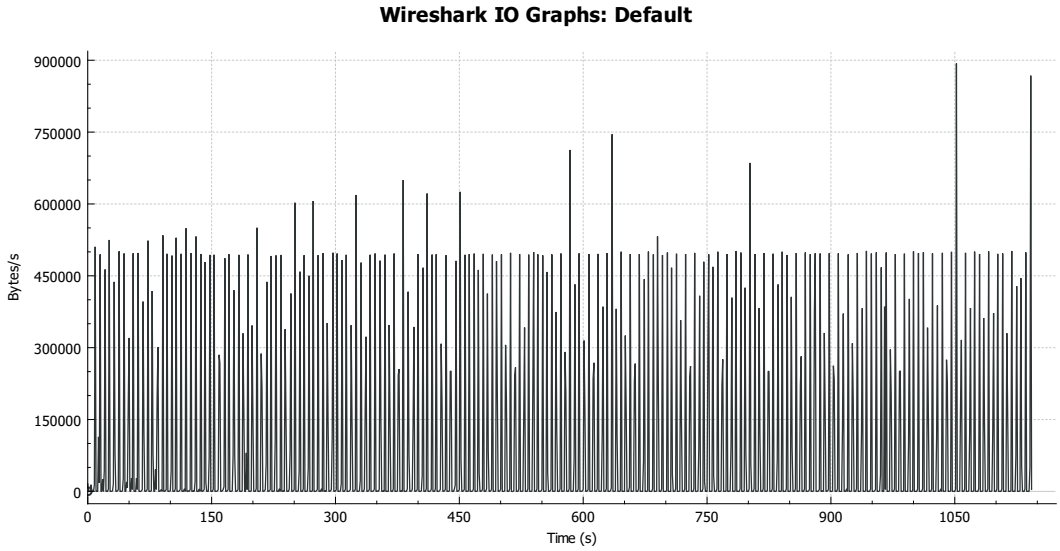


Figure 21 Default setup, two devices - Network traffic when 100 files added per synchronization

A run with 2 devices and 100 files per synchronization was mistakenly run with the tool emulating the system profiler instead of the actual system profiler. The difference in CPU usage is huge. the CPU usage when the actual system profiler is not running is much lower, but the memory usage is also a bit lower, and more stable:

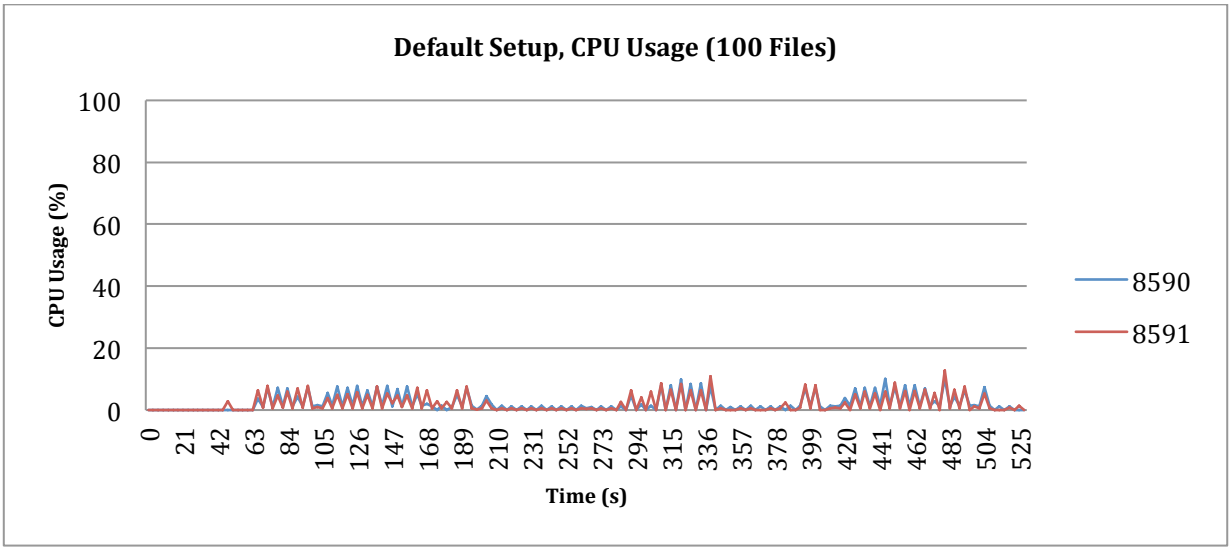


Figure 22 Default setup, two devices, but with the system profiler emulator - CPU usage when 100 files added per synchronization.

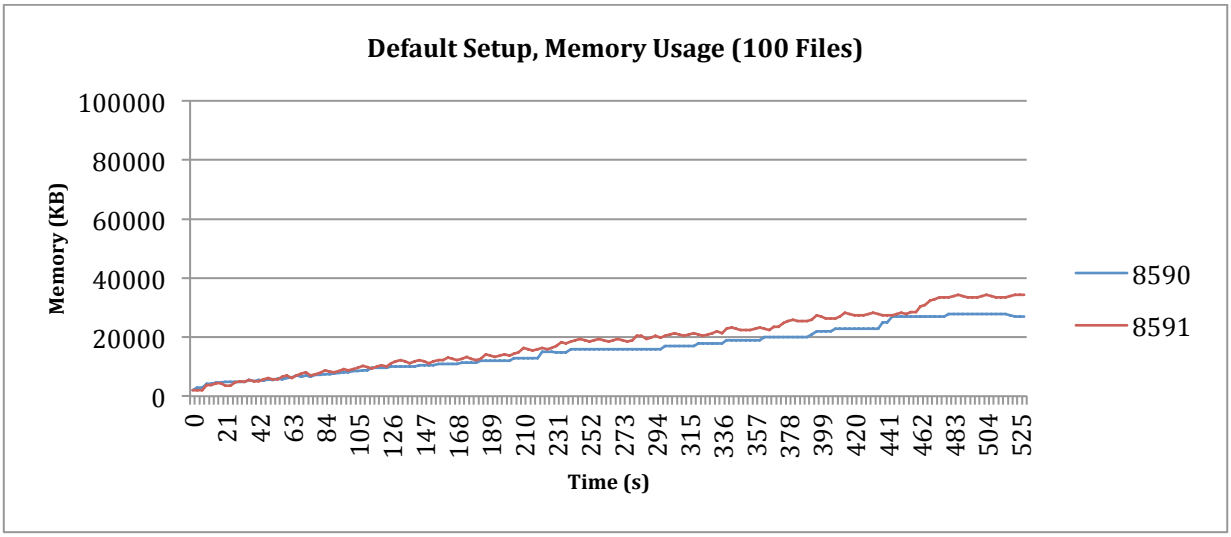


Figure 23 Default setup, two devices, but with the system profiler emulator - Memory usage when 100 files added per synchronization

4.5.1.2 Automatic Coordinator

The prototype is set to attempt to elect the best coordinator. Otherwise the environment is the same as in the previous test.
The coordinator was set to the device at 8590 at start, as it was the first device running, but switched to 8591 after the a few files had been added to the system.
Only 295 files were added before problems occurred during the first test. The daemon seemed to keep accepting requests and replying, but it did not synchronize properly. The new files being added did not show up in the system. It is possible the function evaluating the coordinator got stuck and held the synchronization up, causing a complete halt in progress other than communications. The same bug occurred when adding 5 and 100 files too, but 1445 and 5300 files were correctly synchronized before it locked down. The limit on concurrently open files was set to 4096 in this test too, as the limit would otherwise be reached very quickly.

The following graphs show CPU usage in percentage, memory usage in kilobytes, and bytes sent and received per second over the course of the test.

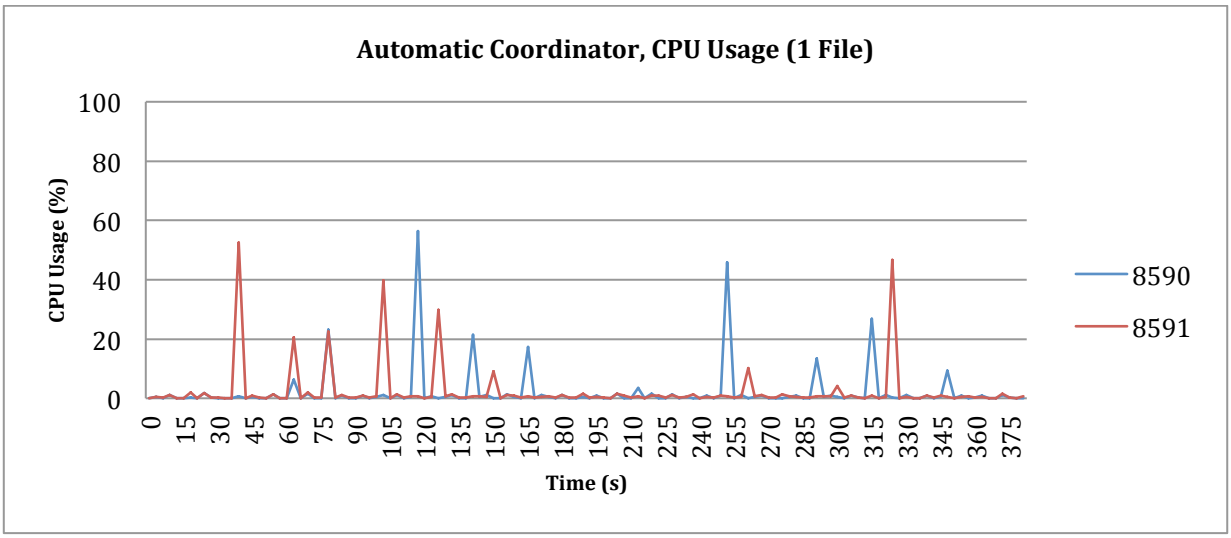


Figure 24 Automatic coordinator, two devices - CPU usage when one file added per synchronization

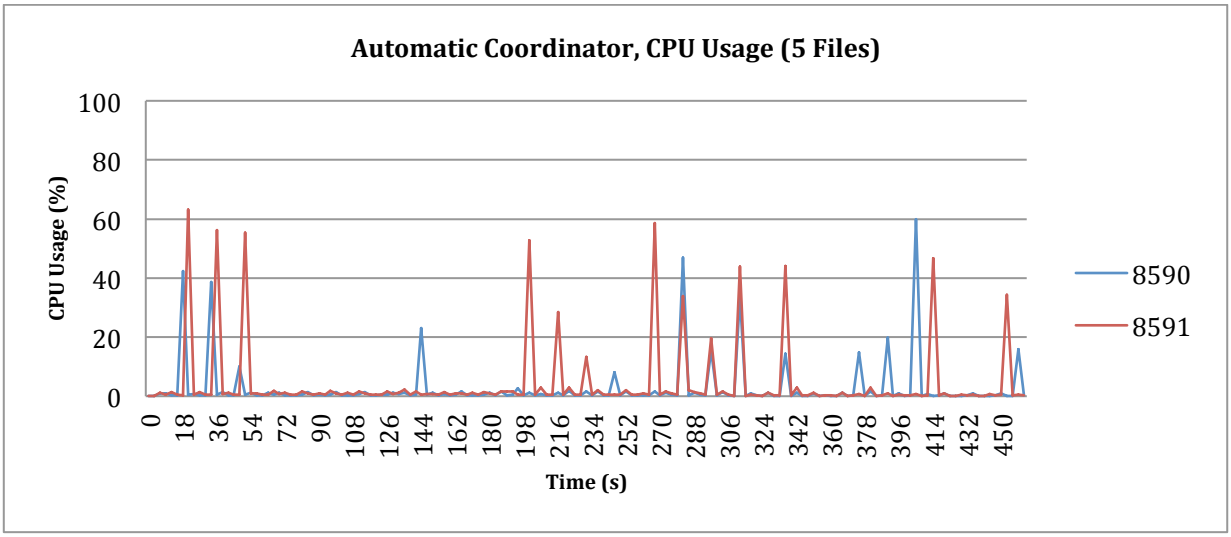


Figure 25 Automatic coordinator, two devices - CPU usage when five files added per synchronization

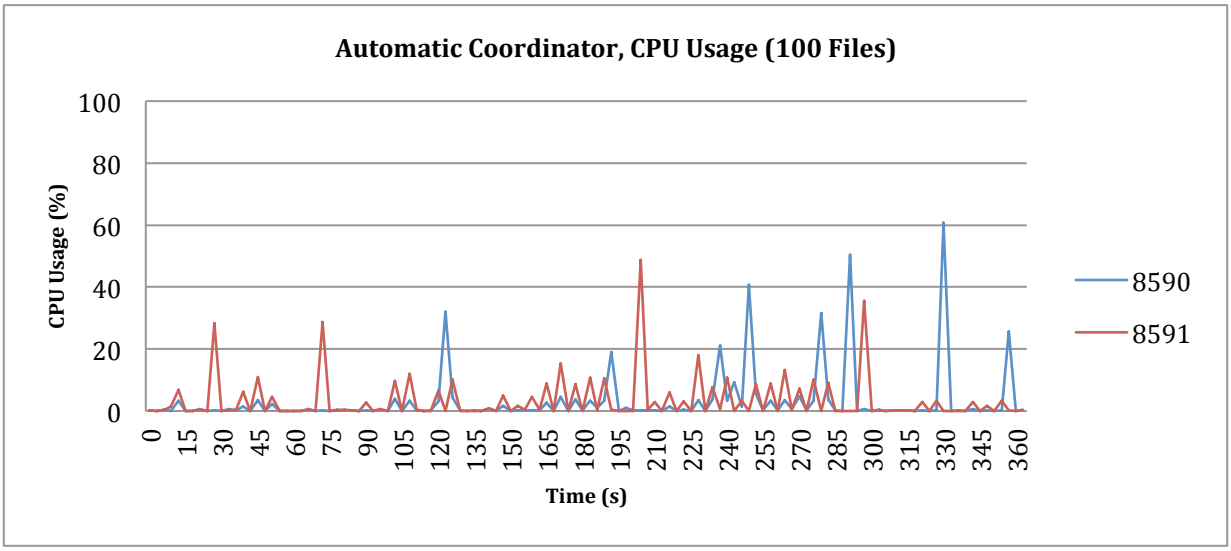


Figure 26 Automatic coordinator, two devices - CPU usage when 100 files added per synchronization

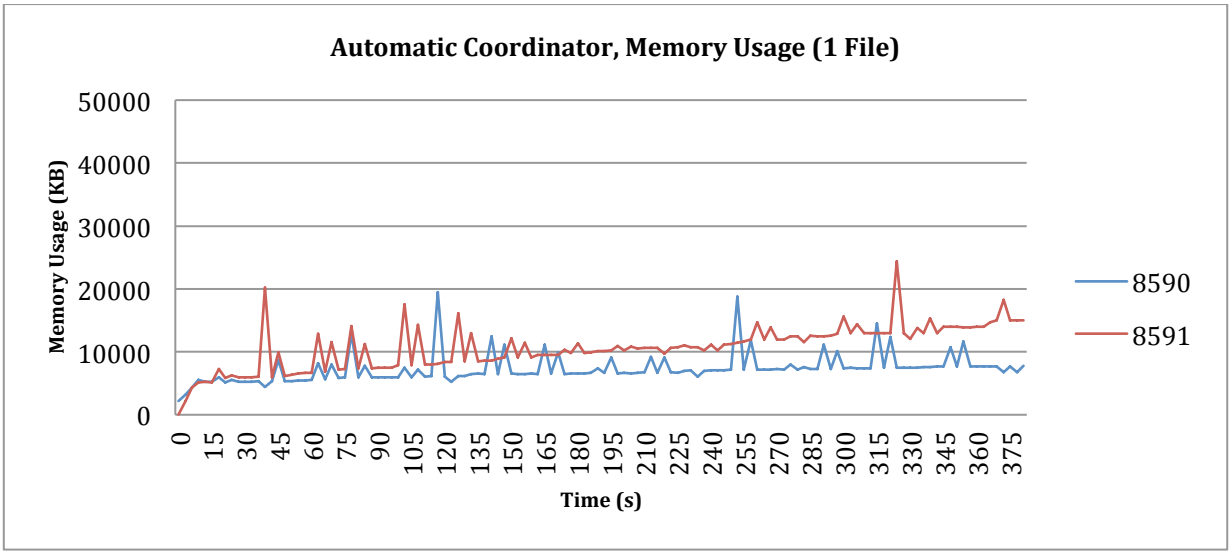


Figure 27 Automatic coordinator, two devices - Memory usage when one file added per synchronization

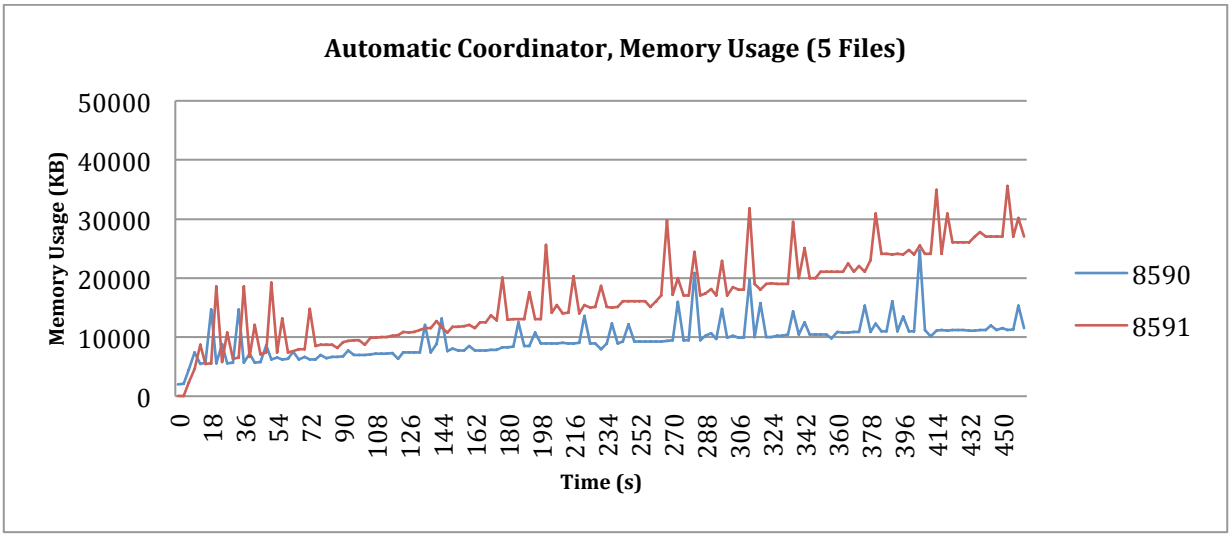


Figure 28 Automatic coordinator, two devices - Memory usage when five files added per synchronization

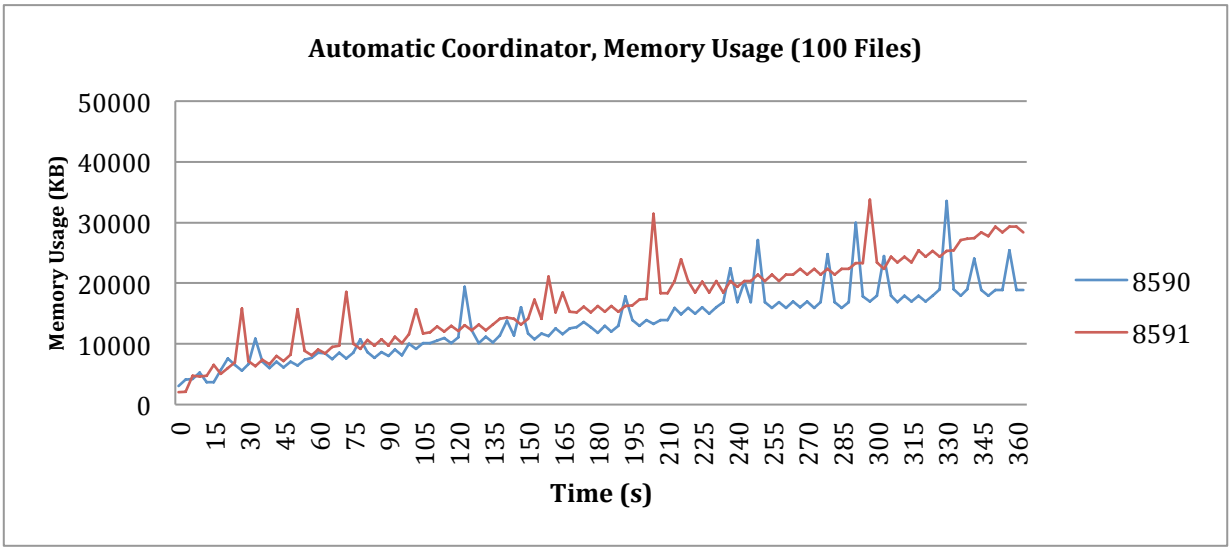


Figure 29 Automatic coordinator, two devices - Memory usage when 100 files added per synchronization

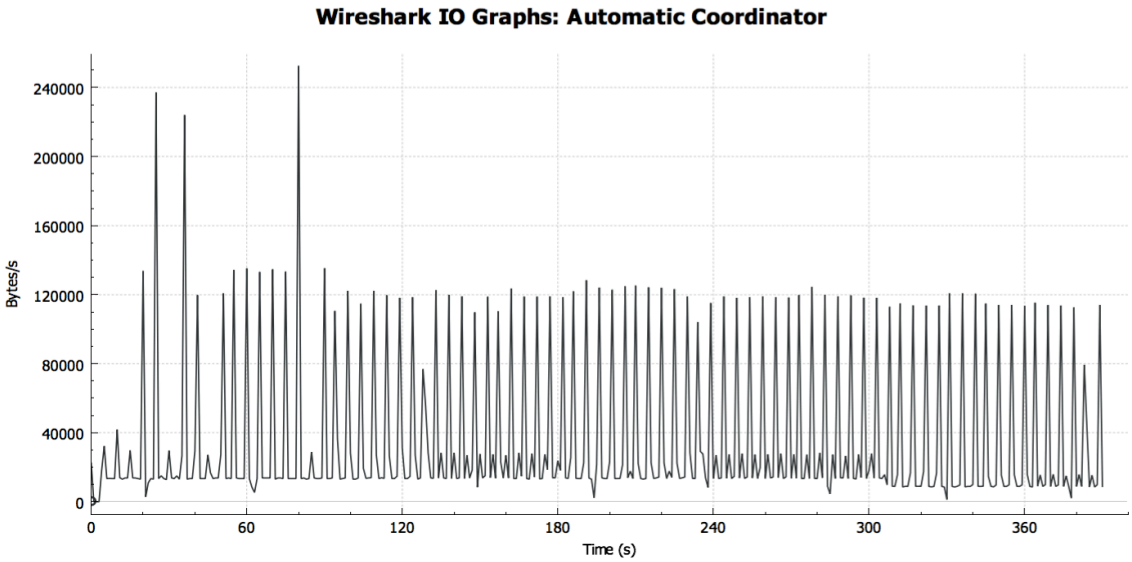


Figure 30 Automatic coordinator, two devices - Network traffic when one file added per synchronization

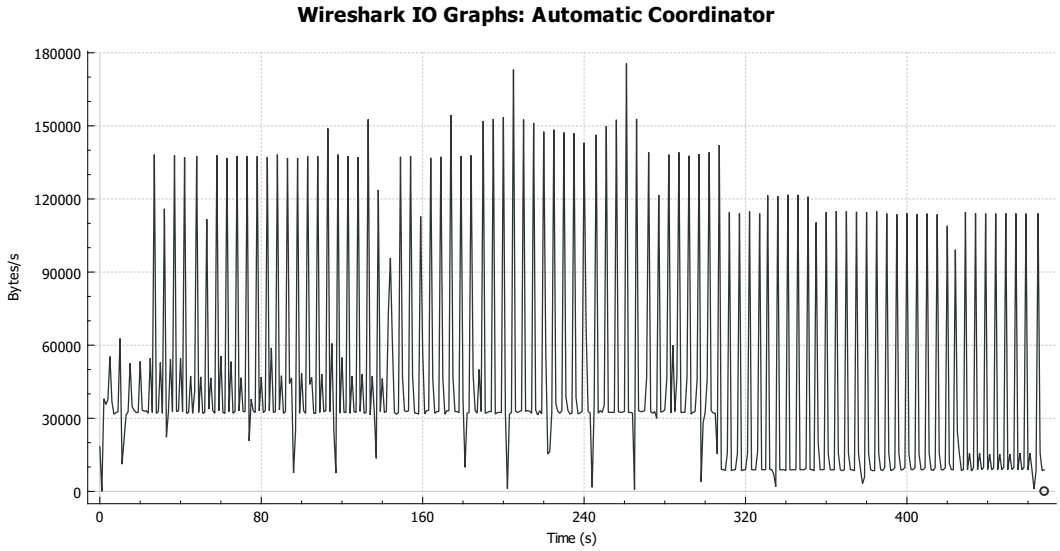


Figure 31 Automatic coordinator, two devices - Network traffic when five files added per synchronization

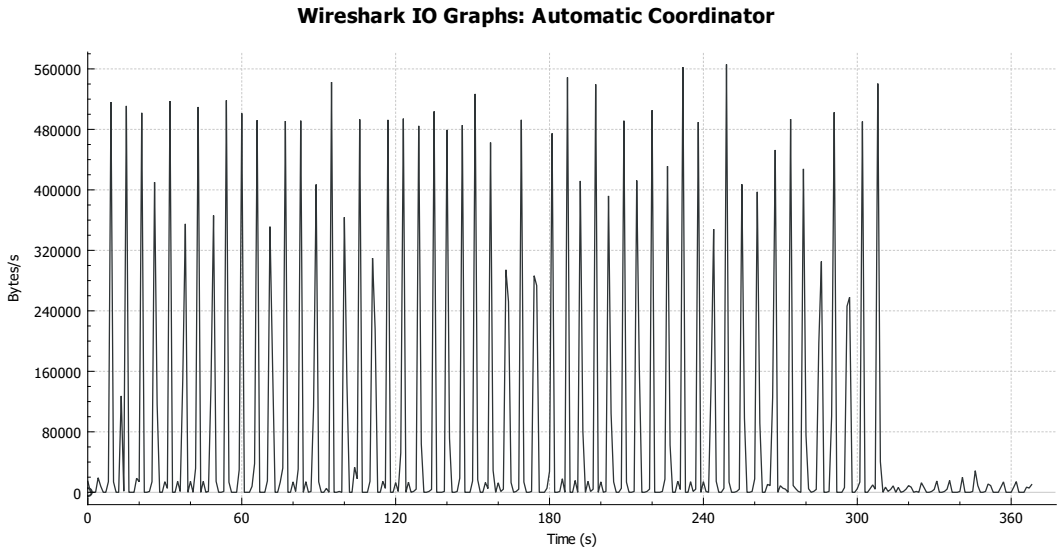


Figure 32 Automatic coordinator, two devices - Network traffic when 100 files added per synchronization

4.5.1.3 3G

The network is severely limited in this test. The Network Link Conditioner is set to 3G mode, which means the following rules apply;

Downlink:
Bandwidth set to 780 kbps
Packages delayed for 100 ms

Uplink:
Bandwidth set to 330 kbps
Packages delayed for 100 ms

Otherwise the environment is the same as in the "default" test, where the coordinator is at the device with the daemon at port 8590.
A mere 312 files were synchronized properly before receiving the same error as in the "default" scenario when adding one file at a time, and 1435 files when adding five a time;

“fork/exec /usr/sbin/system_profiler: resource temporarily unavailable”. It is likely the prototype communicates too much, and does so too often in this test.
When doing cycles of 100 files, over 20000 files were added over the course of almost 3 hours at which point the test was stopped. Open file limit was set to 4096 during the last test.

The following graphs show CPU usage in percentage, memory usage in kilobytes, and bytes sent and received per second over the course of the test. Note that there seems to be a huge spike in traffic right before the prototype crashes in the first test.

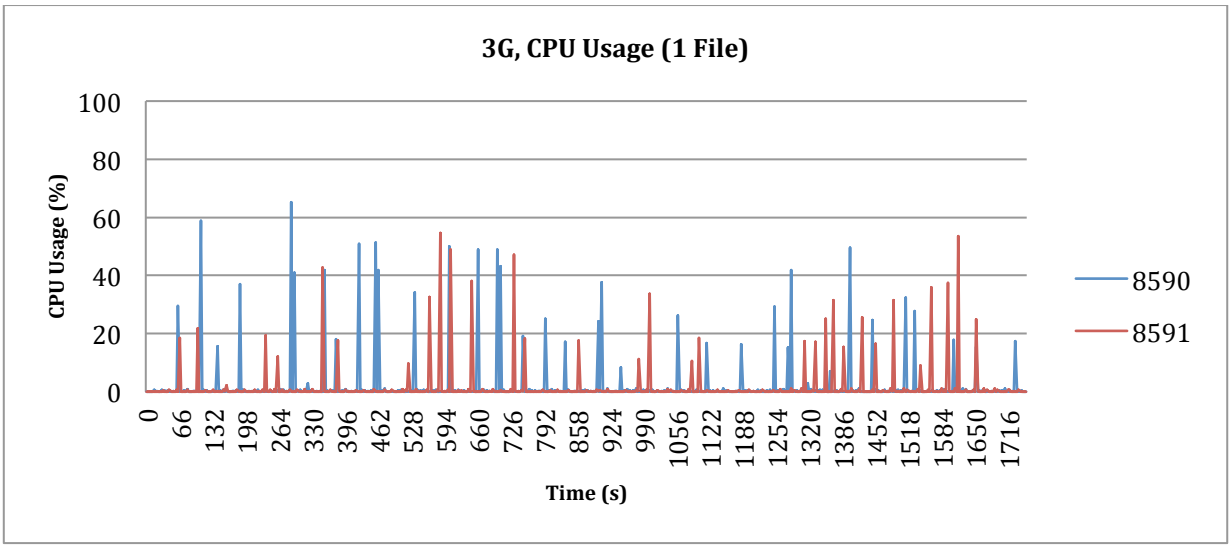


Figure 33 3G, two devices - CPU usage when one file added per synchronization

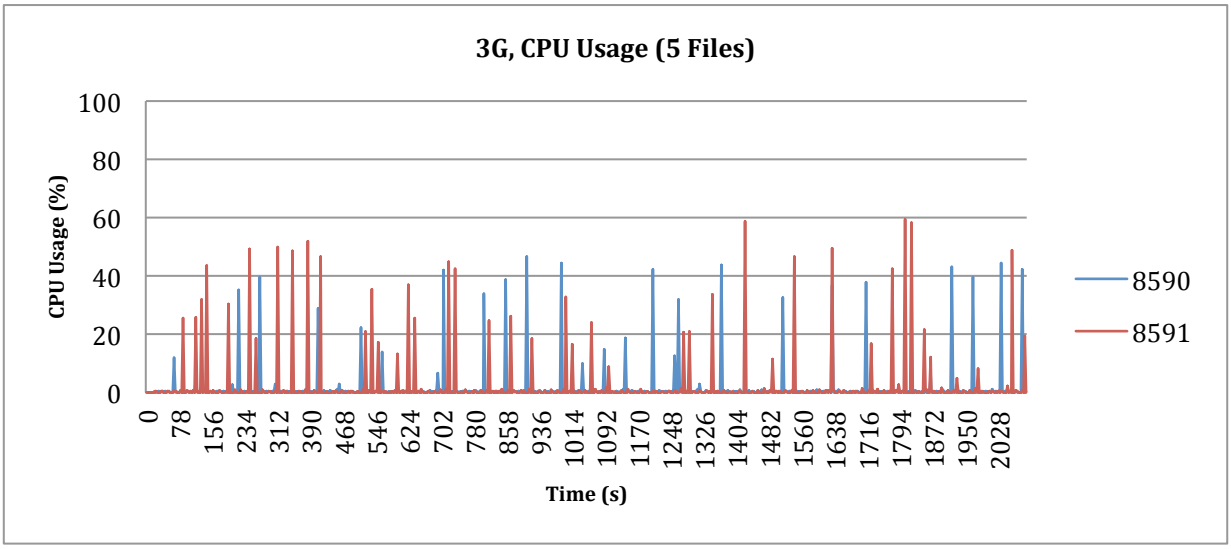


Figure 34 3G, two devices - CPU usage when five files added per synchronization

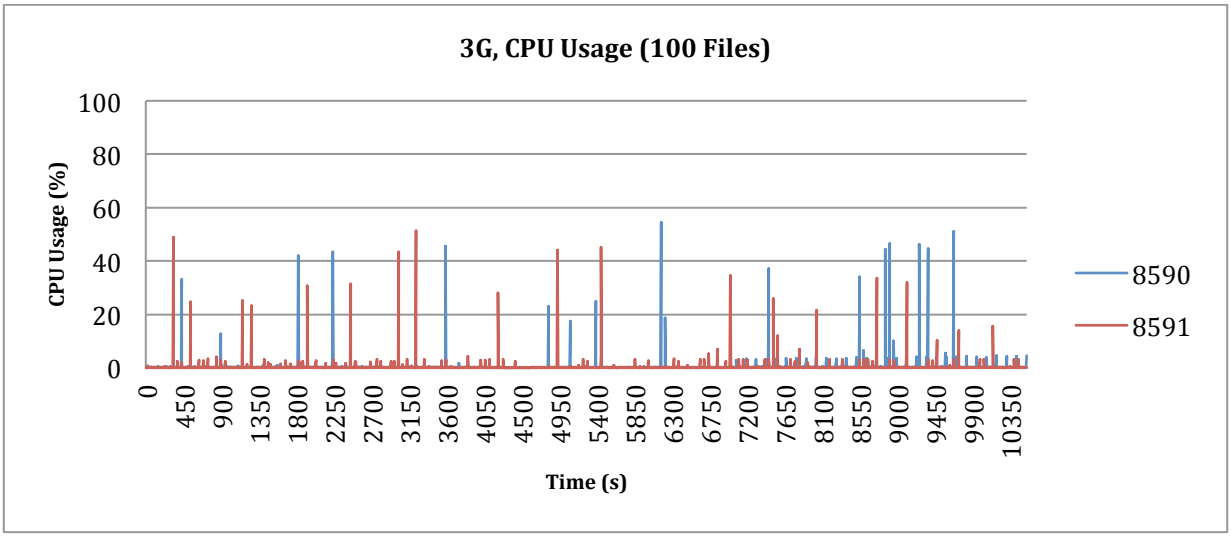


Figure 35 3G, two devices - CPU usage when 100 files added per synchronization

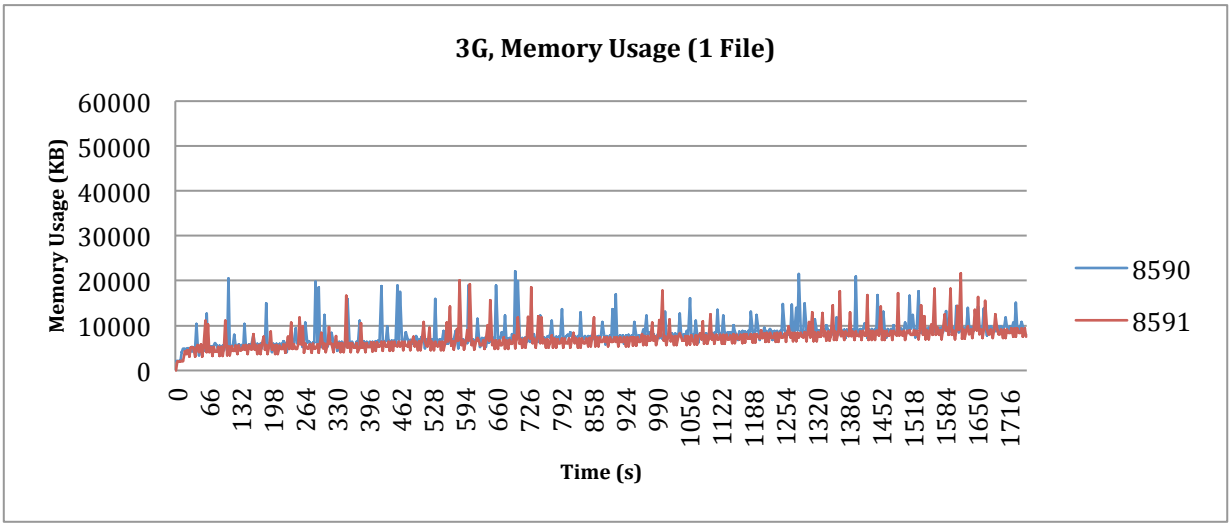


Figure 36 3G, two devices - Memory usage when one file added per synchronization

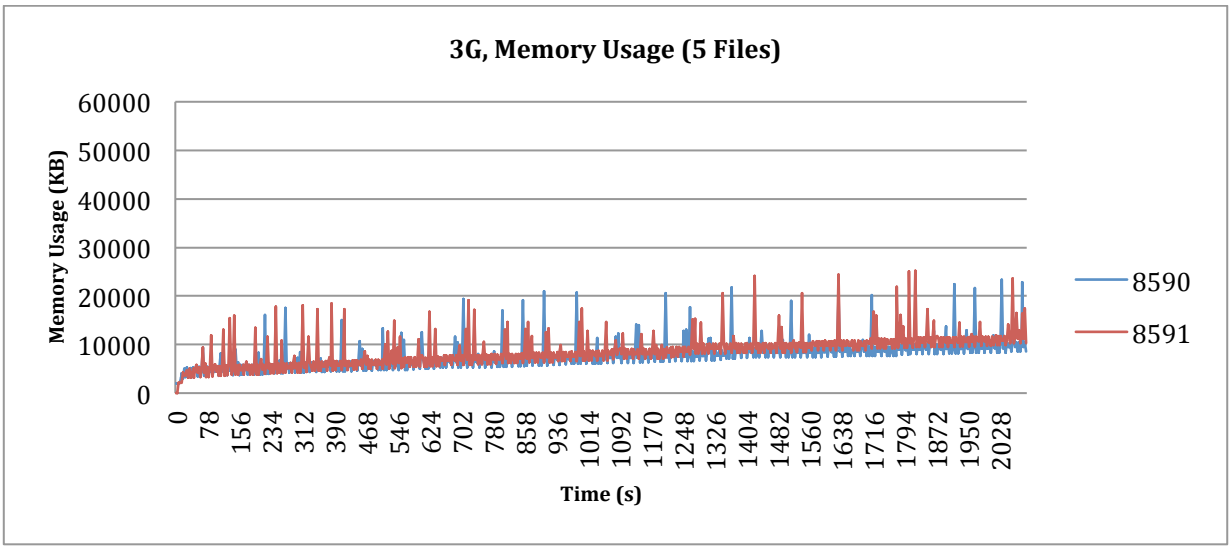


Figure 37 3G, two devices - Memory usage when five files added per synchronization

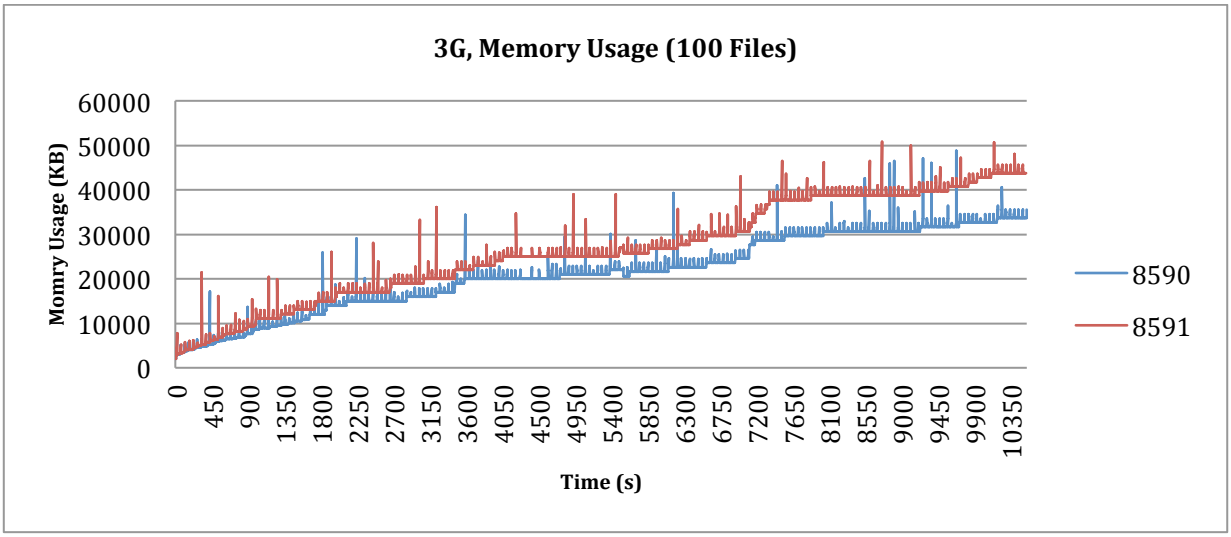


Figure 38 3G, two devices - Memory usage when 100 files added per synchronization

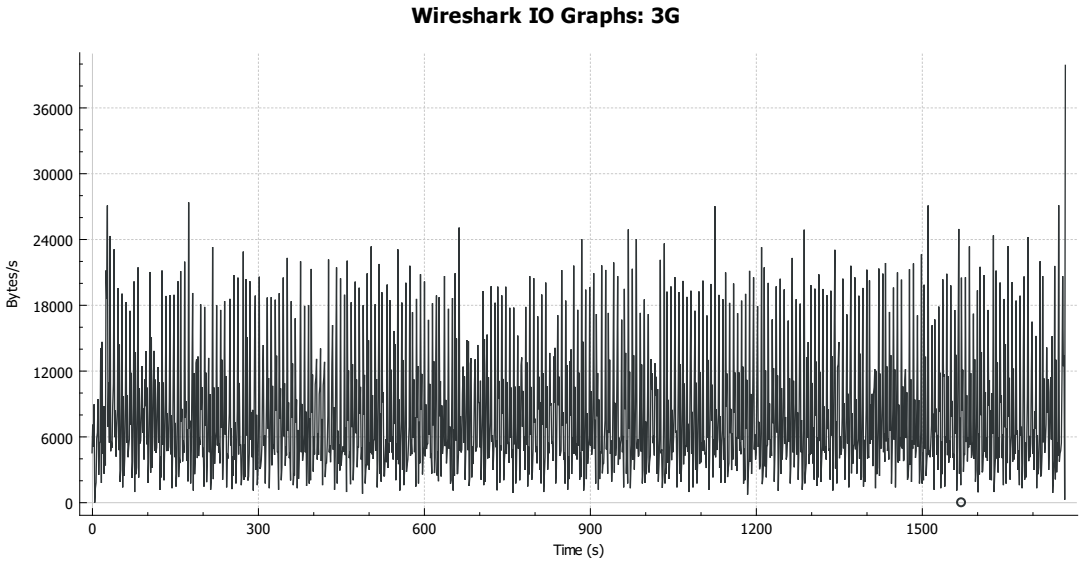


Figure 39 3G, two devices - Network traffic when one file added per synchronization

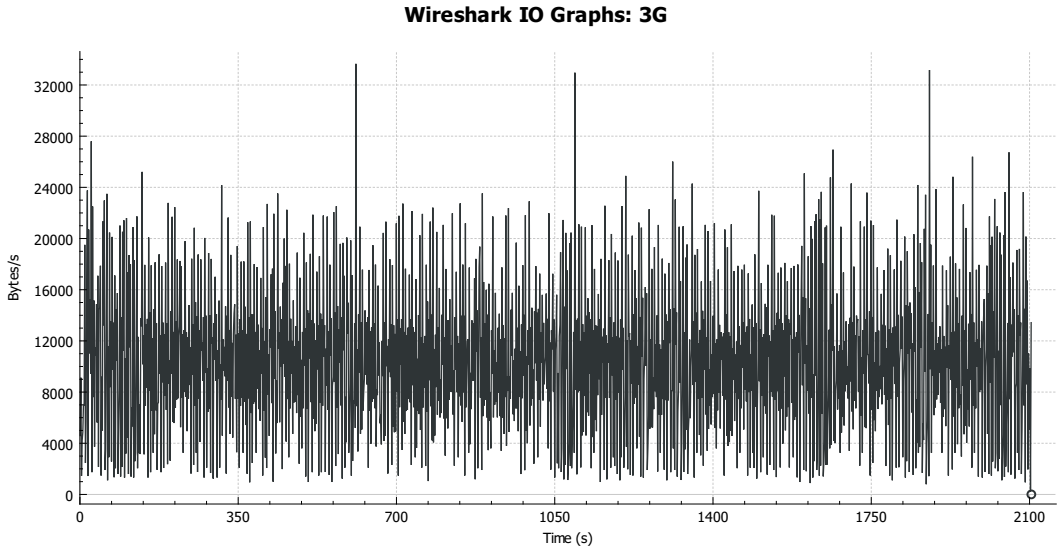


Figure 40 3G, two devices - Network traffic when five files added per synchronization

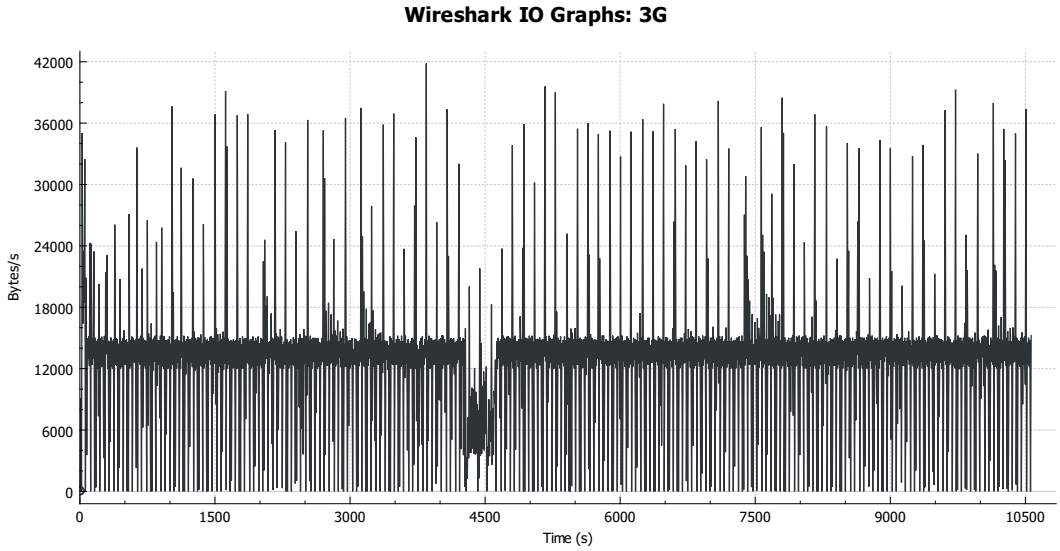


Figure 41 3G, two devices - Network traffic when 100 files added per synchronization

4.5.1.4 Larger File Size

To be certain that file size does not affect the performance of the prototype, an experiment was run with the exact same environment as in the default scenario, except that a file of size 3.6 megabytes was used in stead of a file at 365 bytes. 3.6 MB was chosen because it is about the same size as the average MP3 file according to [28].
The first result was similar to that of the default scenario, adding 1302 files successfully before again crashing with error "fork/exec /usr/sbin/system_profiler: resource temporarily unavailable".
When doing cycles of five and hundred files, warnings of running out of space started to appear, and thus the test was stopped after synchronizing 3605 and 2200 files. As in the previous runs of 100 files, the concurrent open file limit was increased from 256 to 4096 to get a proper test.

The following graphs show CPU usage in percentage, memory usage in kilobytes, and bytes sent and received per second over the course of the test.

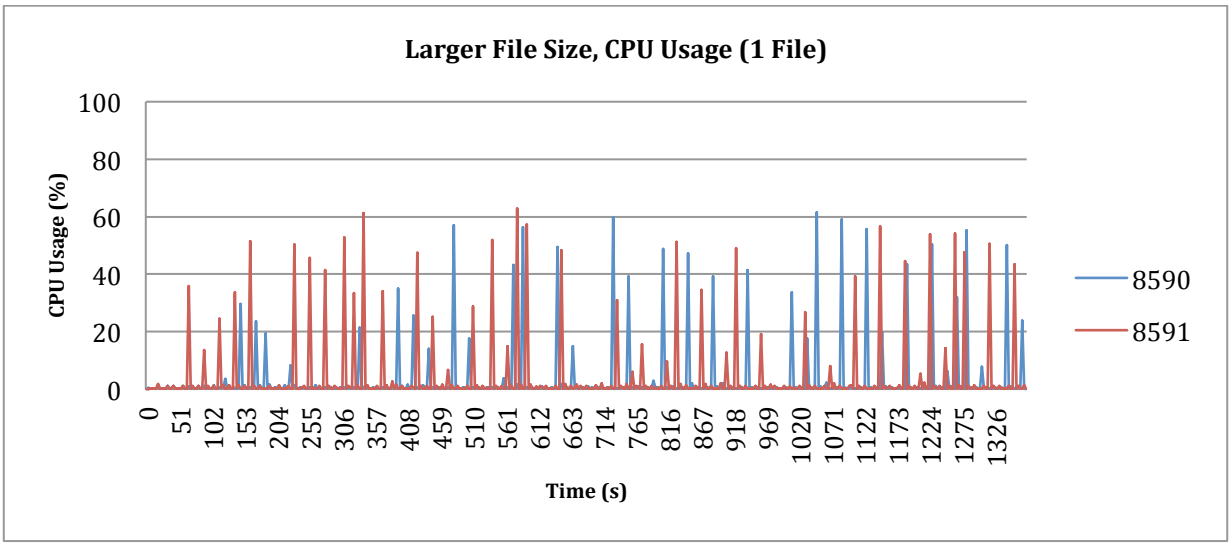


Figure 42 Large file, two devices - CPU usage when one file added per synchronization

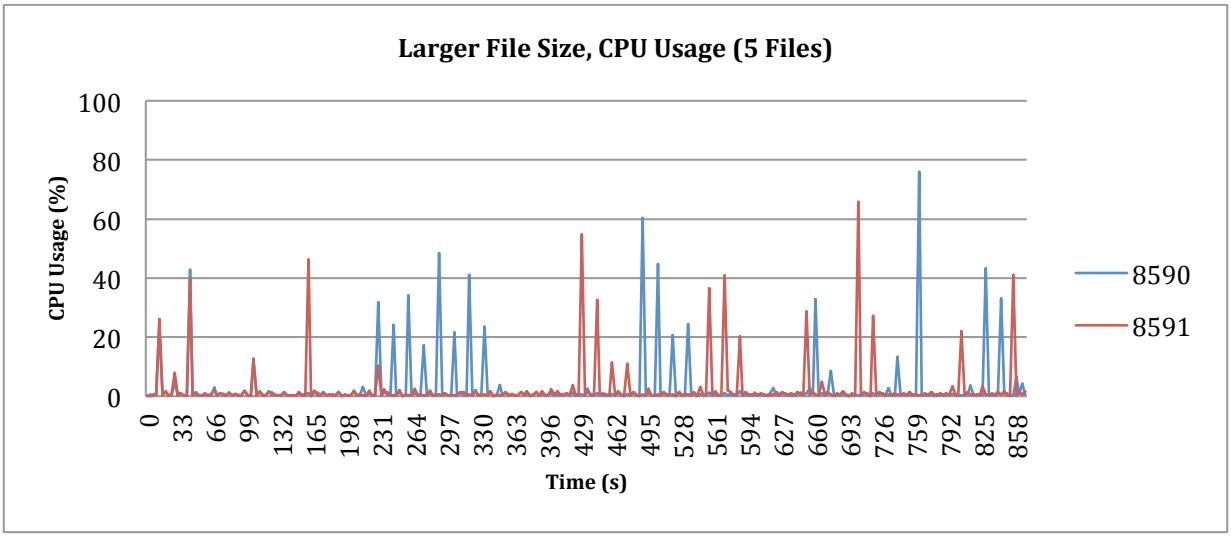


Figure 43 Large file, two devices - CPU usage when five files added per synchronization

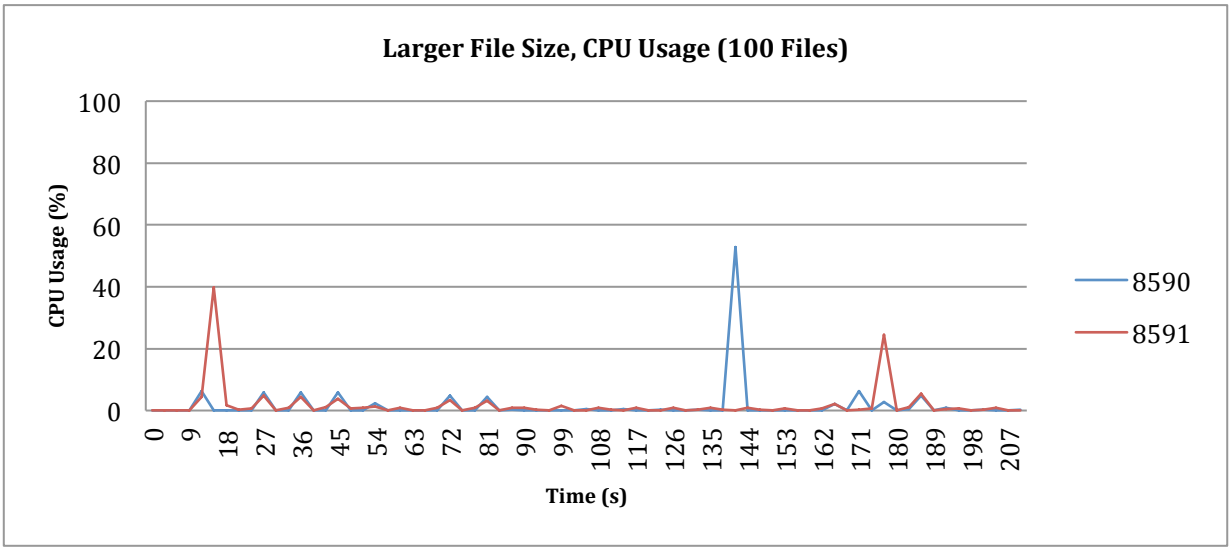


Figure 44 Large file, two devices - CPU usage when 100 files added per synchronization

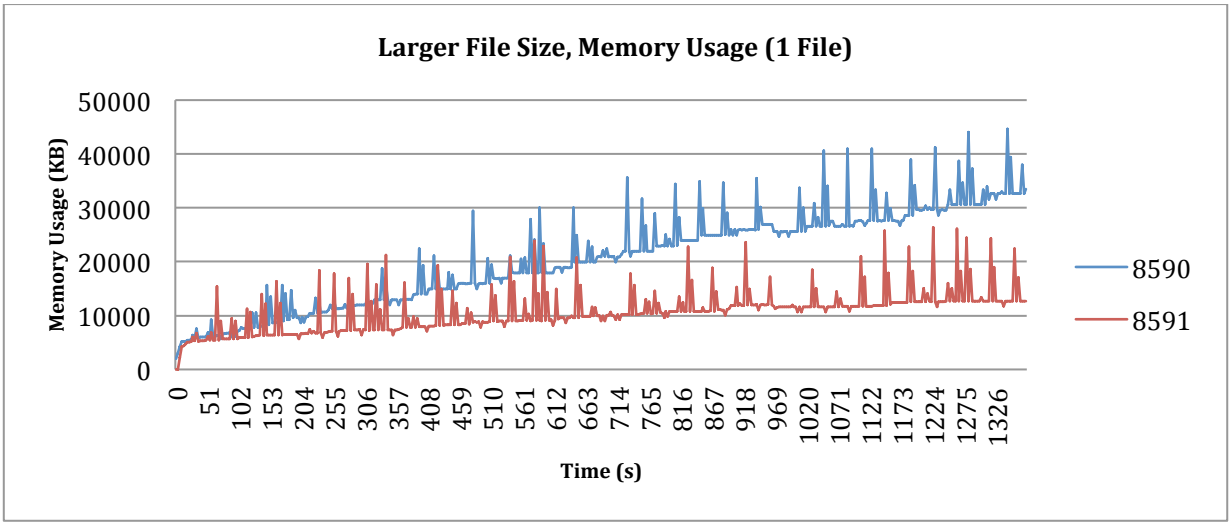


Figure 45 Large file, two devices - Memory usage when one file added per synchronization

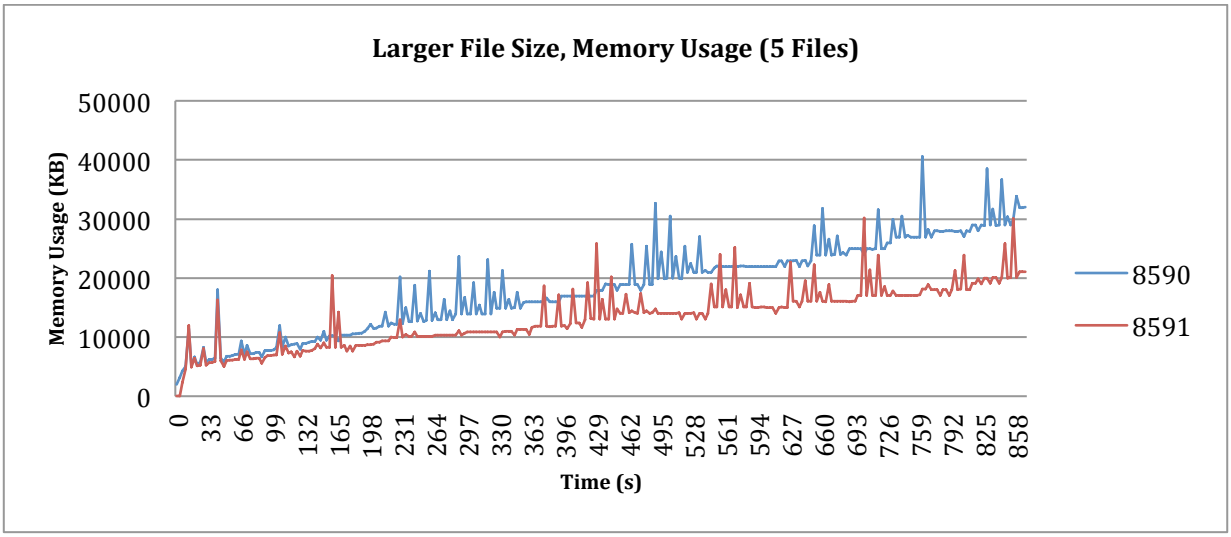


Figure 46 Large file, two devices - Memory usage when five files added per synchronization

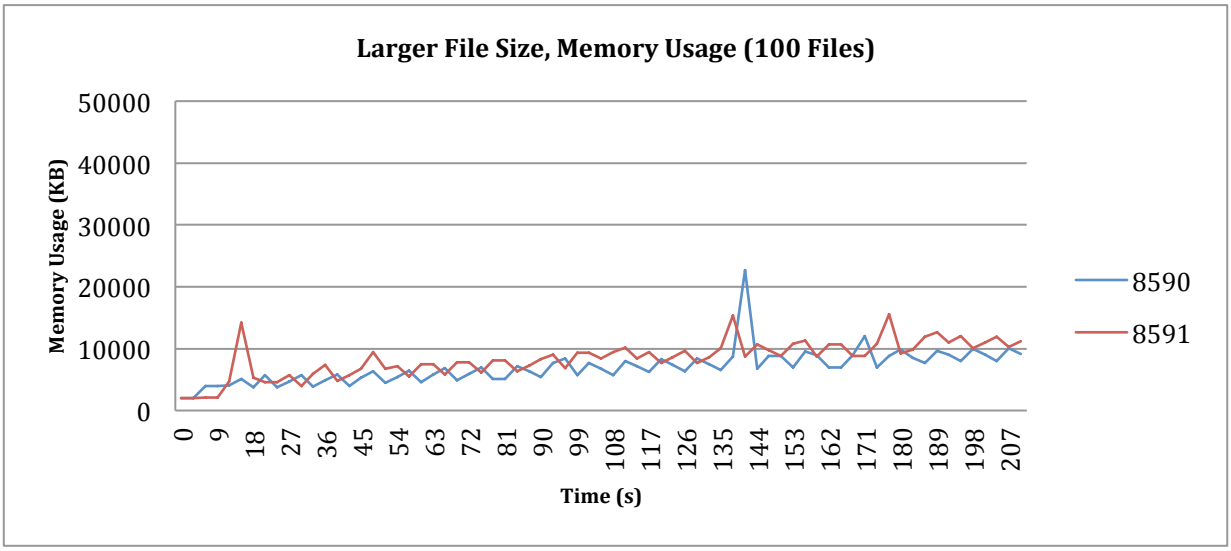


Figure 47 Large file, two devices - Memory usage when 100 files added per synchronization

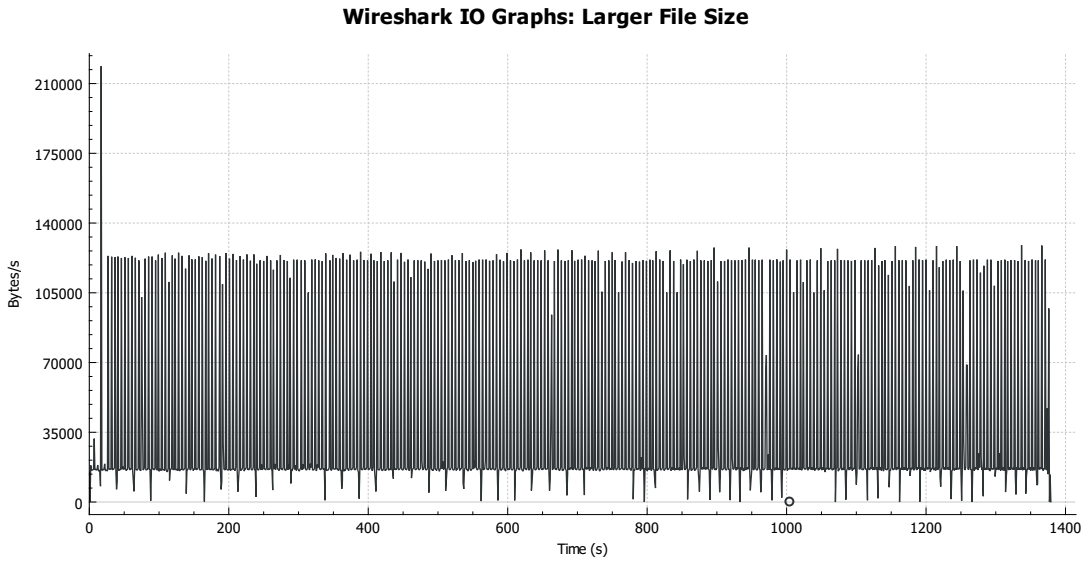


Figure 48 Large file, two devices - Network traffic when one file added per synchronization

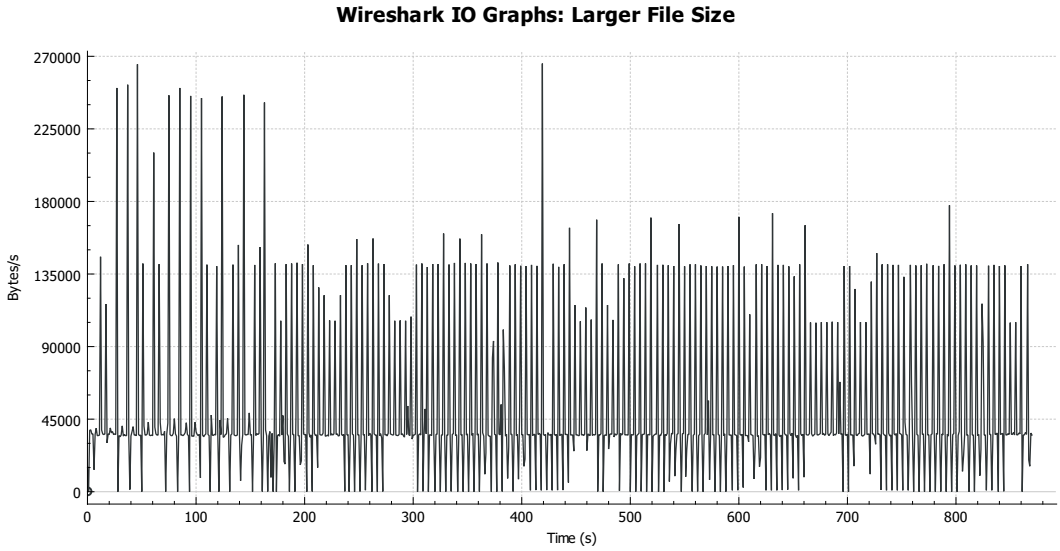


Figure 49 Large file, two devices - Network traffic when five files added per synchronization

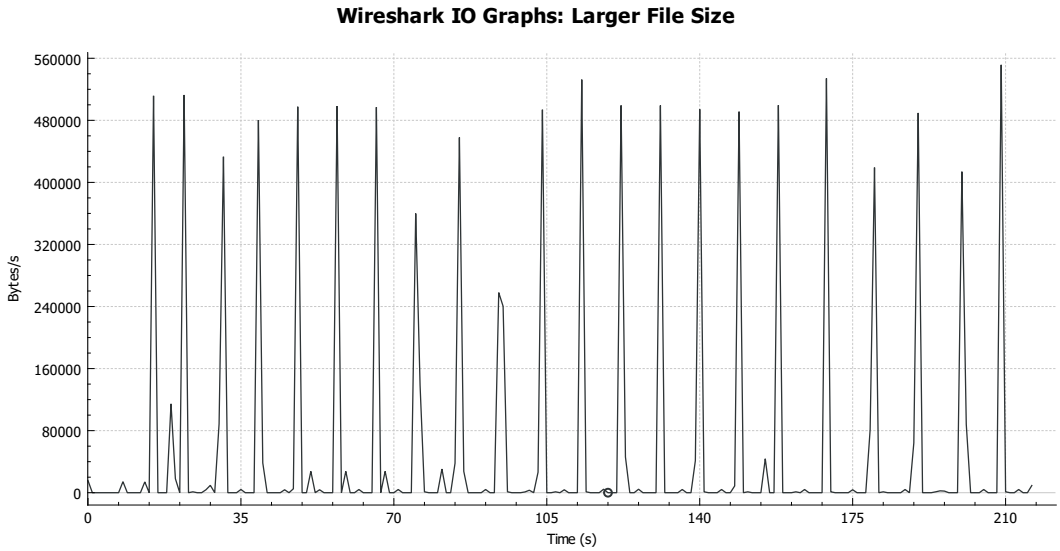


Figure 50 Large file, two devices - Network traffic when 100 files added per synchronization

4.5.1.5 Full Synchronization Before Returning

This test explores what happens when we wait for the synchronization to be fully complete before we allow for new operations. This implies that both files and applications are synchronized before the "refresh" operation returns.
The result of the first two tests was somewhat surprising, resulting in the same type of crash as before; “fork/exec /usr/sbin/system_profiler: resource temporarily unavailable”. This time after synchronizing no more than 282 files for adding one by one, and 1415 files when adding five by five. This could imply that not all functions or goroutines are returning properly, causing a leakage that grows over time.
When doing 100 files per synchronization, the prototype kept going for an hour, at which point 20000 files had been properly synchronized when "ulimit" had been increased to 4096. It was stopped at that point.

The following graphs show CPU usage in percentage, memory usage in kilobytes, and bytes sent and received per second over the course of the test. Note for how long the test

ran compared to the default scenario; almost 45 minutes versus close to 23 minutes in the first test

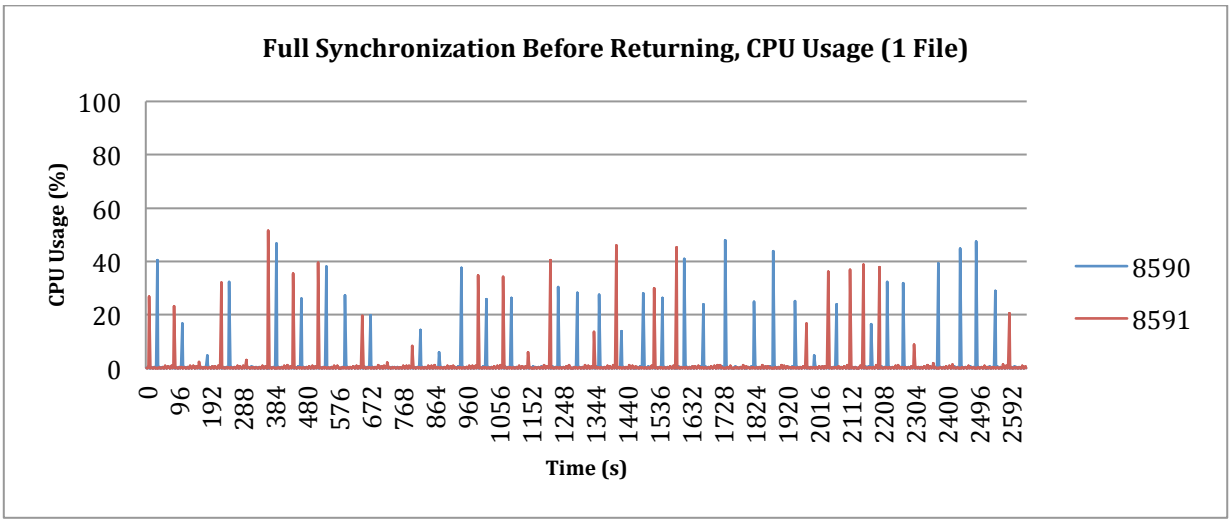


Figure 51 Full sync, two devices - CPU usage when one file added per synchronization

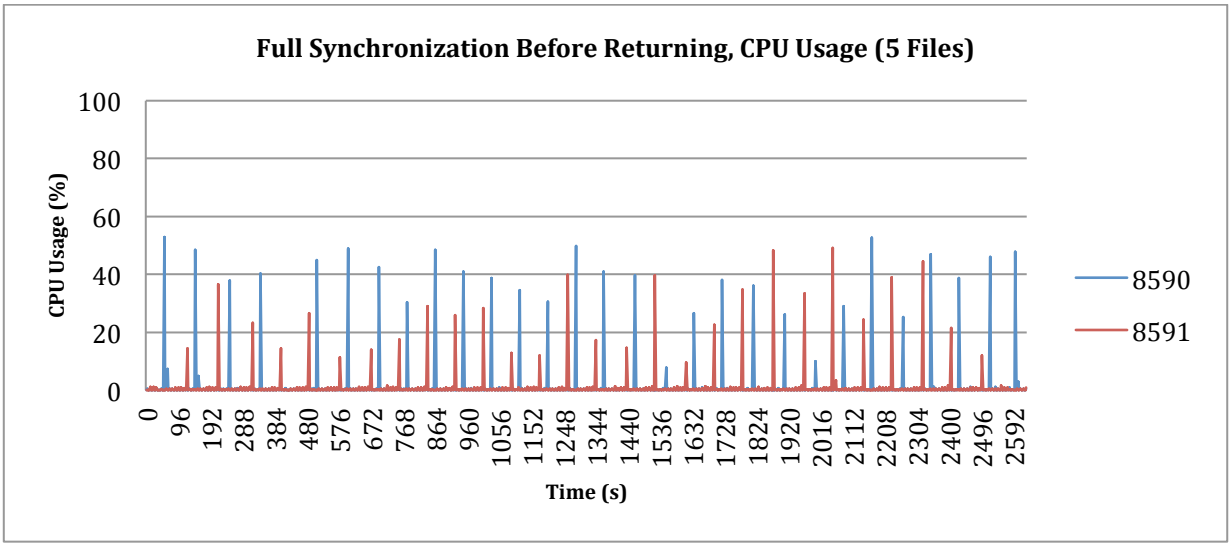


Figure 52 Full sync, two devices - CPU usage when five files added per synchronization

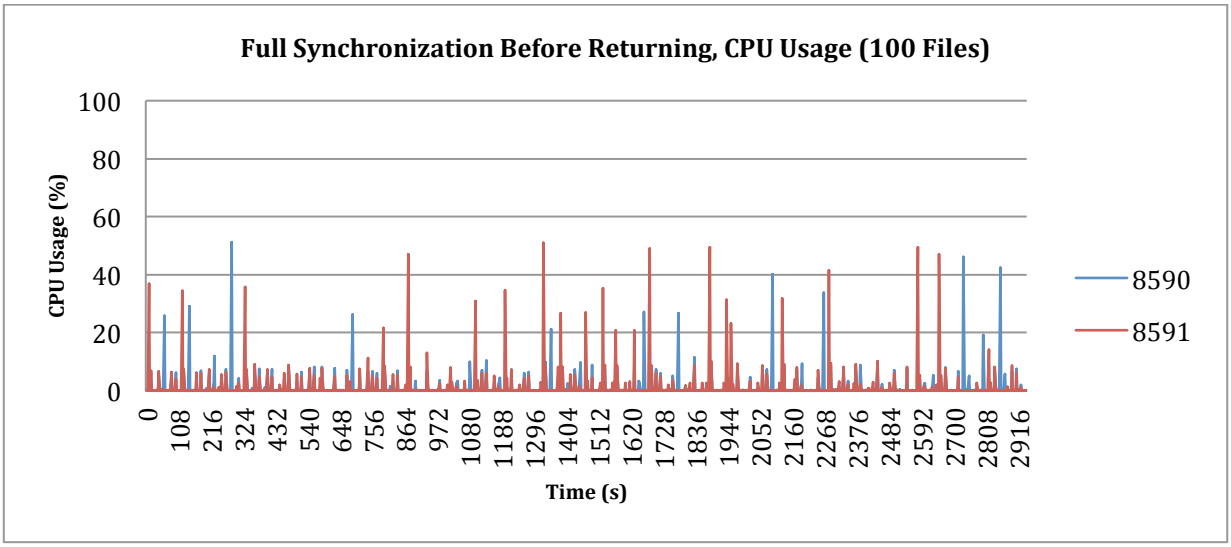


Figure 53 Full sync, two devices - CPU usage when 100 files added per synchronization

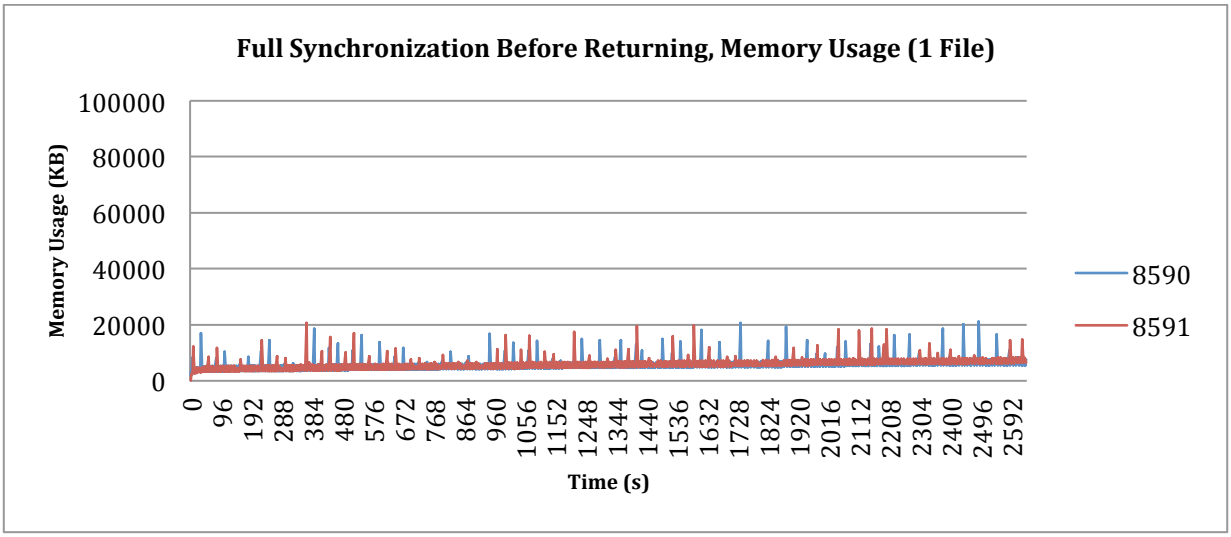


Figure 54 Full sync, two devices - Memory usage when one file added per synchronization

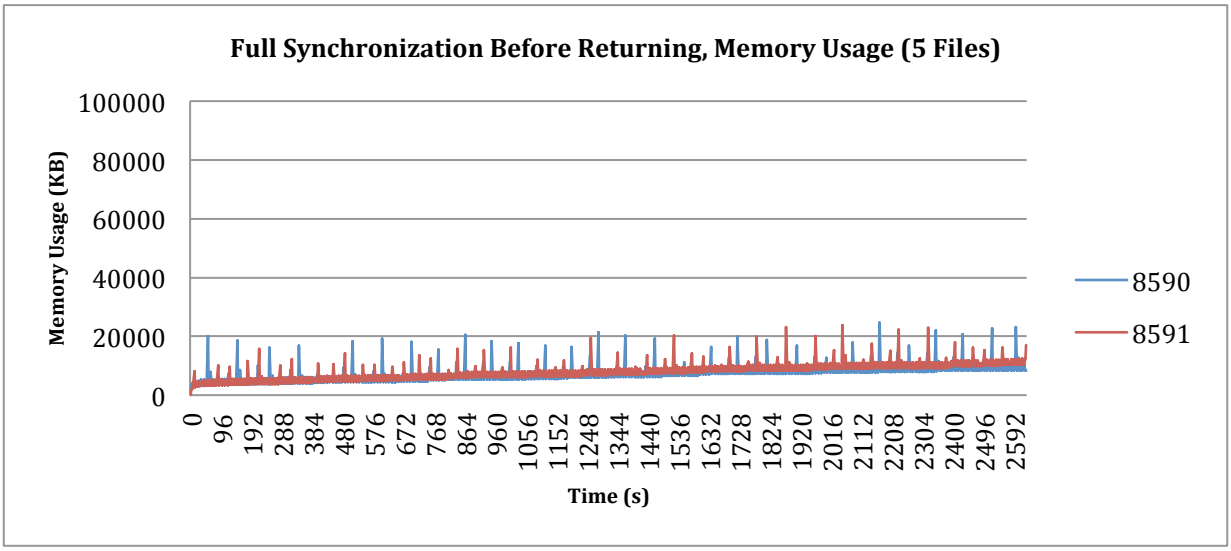


Figure 55 Full sync, two devices - Memory usage when five files added per synchronization

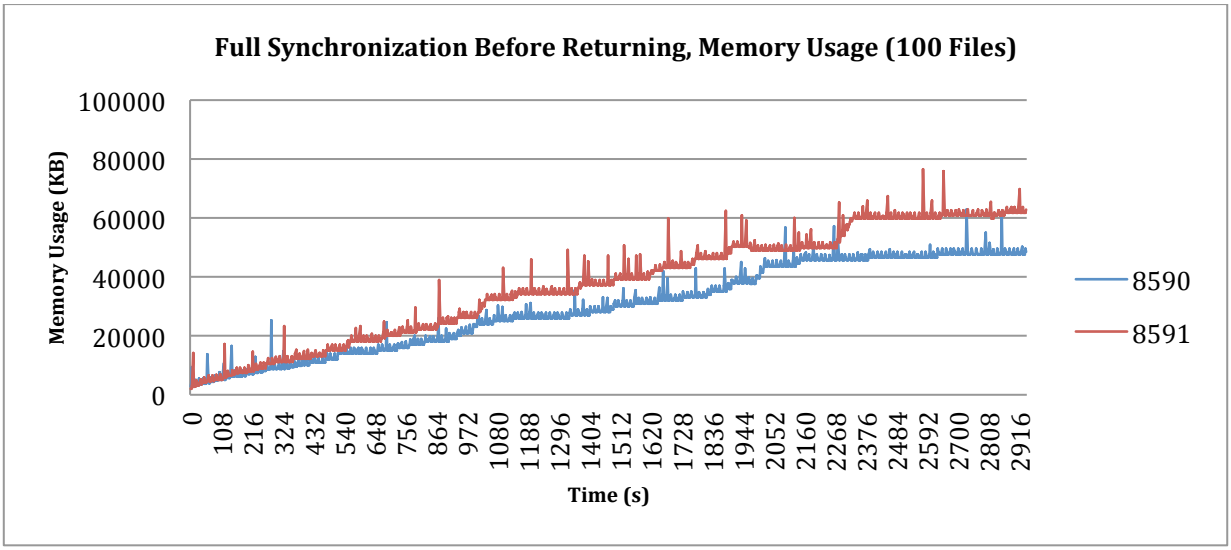


Figure 56 Full sync, two devices - Memory usage when 100 files added per synchronization

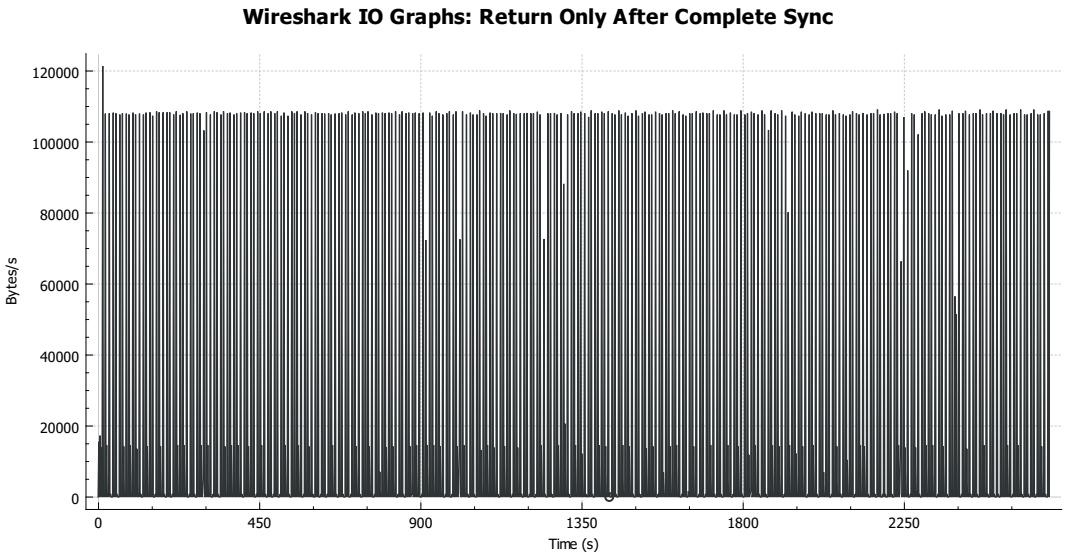


Figure 57 Full sync, two devices - Network traffic when one file added per synchronization

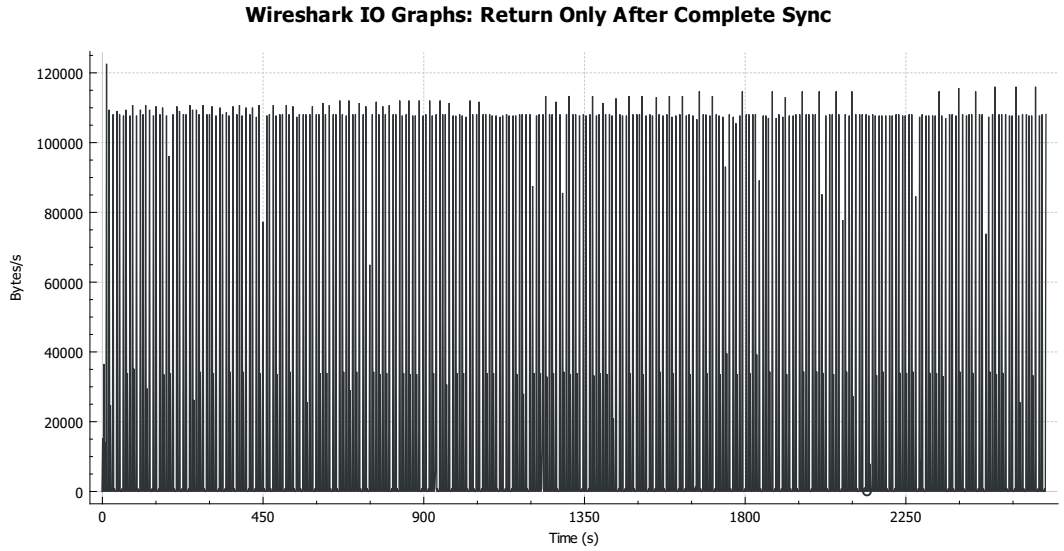


Figure 58 Full sync, two devices - Network traffic when five files added per synchronization.

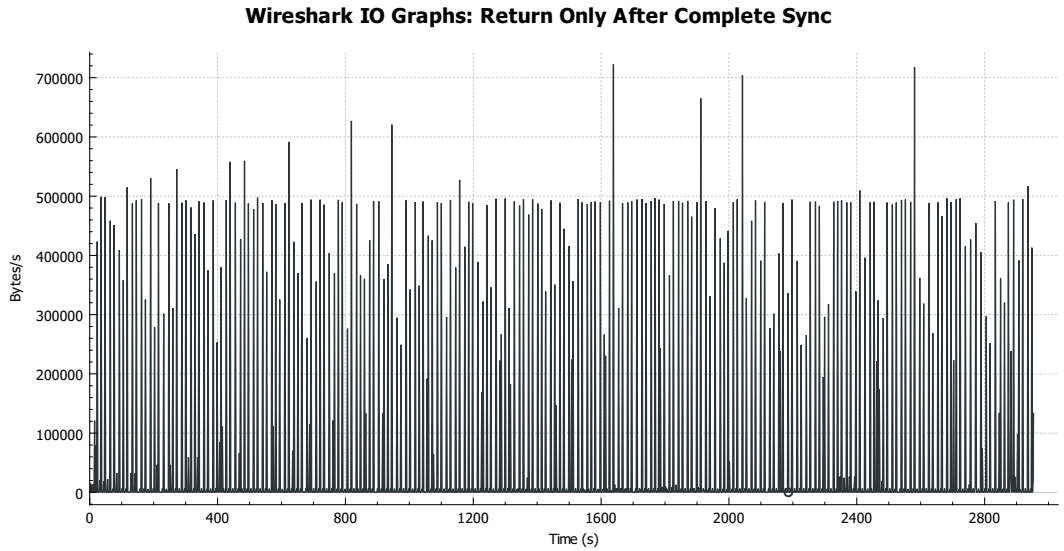


Figure 59 Full sync, two devices - Network traffic when 100 files added per synchronization

4.5.2 Ten Devices

The machine used during the experiments struggled with having ten devices running locally at the same time, frequently eating up almost all CPU time. Also the limit on concurrently opened files was initially often reached very quickly. Merely 200 - 400 files were successfully synchronized in most cases, even after increasing the file limit to 4096. Some noteworthy exceptions are included here.

4.5.2.1 Automatic Coordinator

When adding one file per synchronization, the prototype timed out on connections during coordinator evaluation after adding 197 files successfully.
Neither the memory or CPU graph shows anything out of the ordinary, but the CPU data contains some interesting spikes every 15 - 20 seconds, possibly when the evaluation of the coordinator is running. The time between evaluations is set to five seconds, but it is likely that locks and communication times etc. add a few seconds to the intervals in combination with the samples used having intervals of three seconds. The data recorded by Wireshark does however show that there is basically no communication going on for the last few seconds, which does fit with a time out, but could possibly be blamed on the capturing not being stopped in time (a similar coda appears in the data for the "five files" test with no timeout errors).
The exact cause of the timeout is hard to pinpoint, as I was unable to reproduce the error.
Do note that the timings on the CPU and memory data do not fit the data from Wireshark. This is likely because the application logging the CPU and memory data seems to have been affected by the high pressure on the CPU causing the timings to seemingly get skewed.

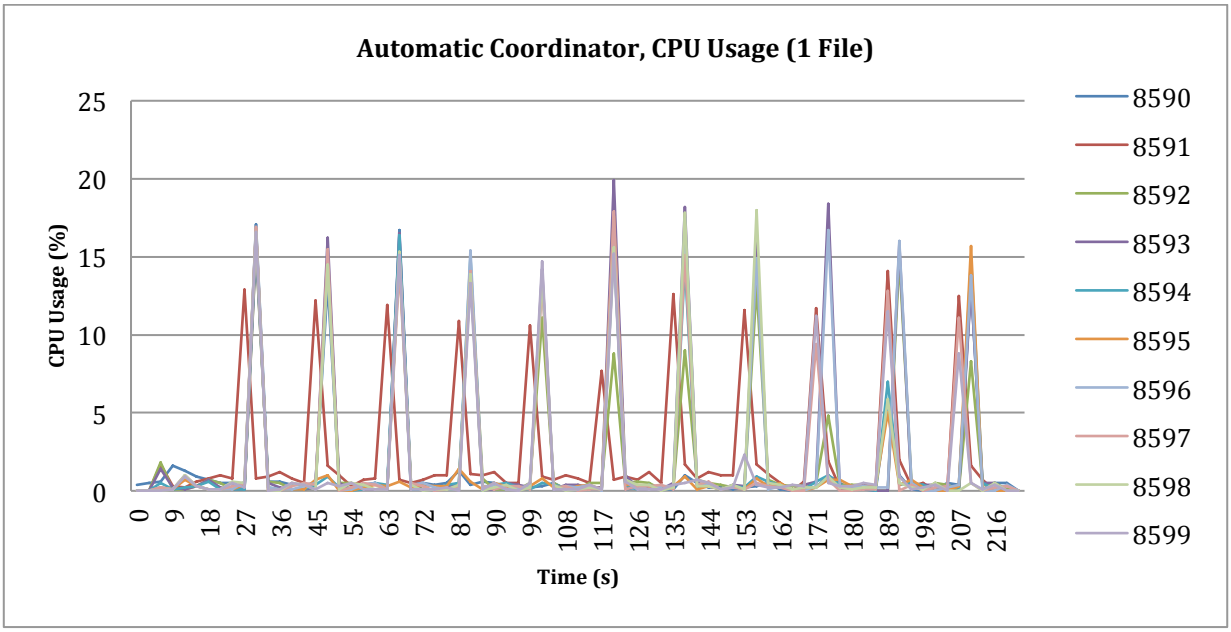


Figure 60 Automatic coordinator, 10 devices - CPU usage when one file added per synchronization

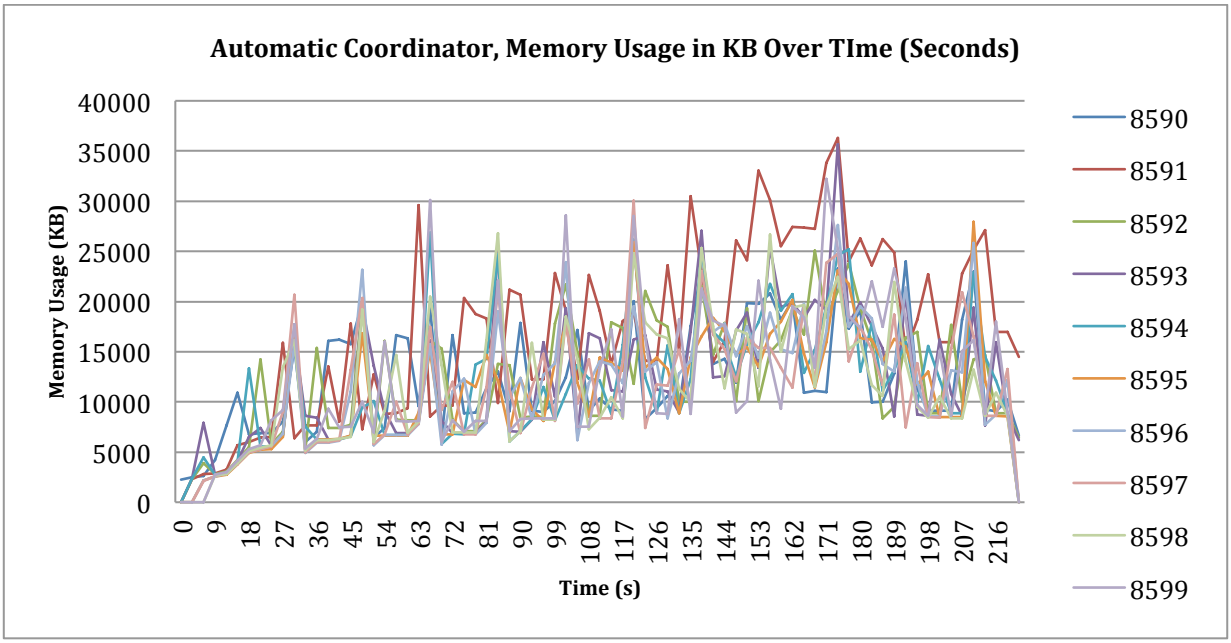


Figure 61 Automatic coordinator, 10 devices - Memory usage when one file added per synchronization

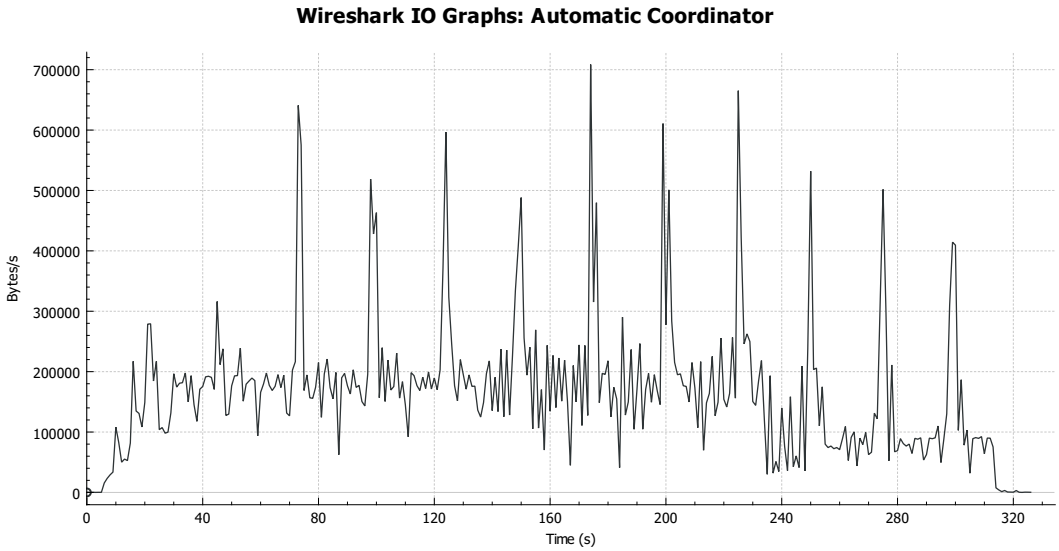


Figure 62 Automatic coordinator, 10 devices - Network traffic when one file added per synchronization

4.5.2.2 3G

The data for the 3G test with 100 files adder per synchronization shows interestingly that it managed to add 2898 files before reaching the 256 concurrently-open-files-limit. This is much more than both the average of the tests run with ten devices, and the result from the tests with five and one files per synchronization with the same limit. Likely the added delays caused by the slow network gave the prototype more time to return goroutines properly, but the two other cases should have returned similar results if that was the entire story. It does look like there is a leak of goroutines or functions that do not return properly that add up in correlation with either time or number of synchronizations. 2898/100 files means 28 successful synchronizations, while for five and one we get 308/5=61 and 76/1=76 respectively. As for time, the 1 and 5 test ran for about 9 - 10 minutes, while the 100 test ran for almost 27 minutes. The result is in fact very close to the result from the same test with just two devices where the number of successful synchronizations is about the same, but with a slightly shorter runtime of

about 24 minutes. Note that the results from the 2-device tests does look similar for the 1 and 5 files test as well, but the difference from 1 and 5 files to 100 files was less profound (these values are from when running two devices with a file limit of 256, not 4096 as shown in the data earlier).
When the file limit was increased, the 100 files test managed to synchronize 5600 files before exhausting resources and crashing. It seems like the delay helps when it comes to keeping number of concurrent files down, but when that aspect is removed, the delay does not seem to have any other effect than to slow things down (453 files synchronized at a file limit of 256 and 5785 files at a file limit of 4096 for same set up with a normal LAN connection). Clearly the problem is with reading the files.

4.5.2.3 Full Synchronization Before Returning

The data for the full synchronization test was similar to that of the 3G test in that the amount of files successfully synced was 2976 before reaching the open file limit when adding 100 files per synchronization, while it was 275 for five files and 55 for one file, both stopping when receiving the " fork/exec /usr/sbin/system_profiler: resource temporarily unavailable" error. Again, the slower progression of the prototype under this setting is likely helping the number of open files to stay low. Quite a few of the ten-devices-tests with normal file limits crashed due to too many open files.
When the file limit was increased, 5700 files were synchronized before resources were maxed out: a value that is very much comparable to that of the setup with 3G and the default setup. Running a combination of full synchronization and 3G, which reached 5600 files before crashing in the same way, more or less confirms the suspicion that these two factors matters little on resource usage, at least to a point (when running two devices, these two tests had to be stopped after running for hours, while the default setup timed out at 8500 files).

4.6 Coordinator Selection Time

In this experiment, a specified number of "devices" were set up, some of them with a small amount of files just to test the automatic coordinator function in the prototype. The first devices/daemon at port 8590 had eight files, the second on port 8591 had 11 files, and the fourth on port 8593 had one file (this one file is of course not included when running less than four devices). The automatic coordinator selection was enabled, and the network situation the same as in the previous experiments. The file size and type varied, but as proven in the previous experiments, this should not affect anything. The first daemon at 8590 was set to be the initial coordinator, and as soon as the initial set up phase where the daemons connect to each other, a refresh/synchronization request was sent via the HTML GUI on the first device. The time was then taken from the instant the coordinator evaluation was initiated to the daemons were told a new coordinator was elected (should be the daemon at 8591 as it has the most files). The times were measured by getting the time at the start of the function choosing the coordinator, and then subtract that time from the current time (using the Golang "time" library) when the request had been fully processed. The time of the slowest daemon was then taken to be the coordinator selection time. The times represented in the graph are based on 5 sample averages. The open file limit was increased from 256 to 4096 when running 50 or more devices, as the limit would otherwise be reached. When running with 80 devices, time outs would frequently happen, thus attempts at testing 90 or more on LAN proved fruitless.

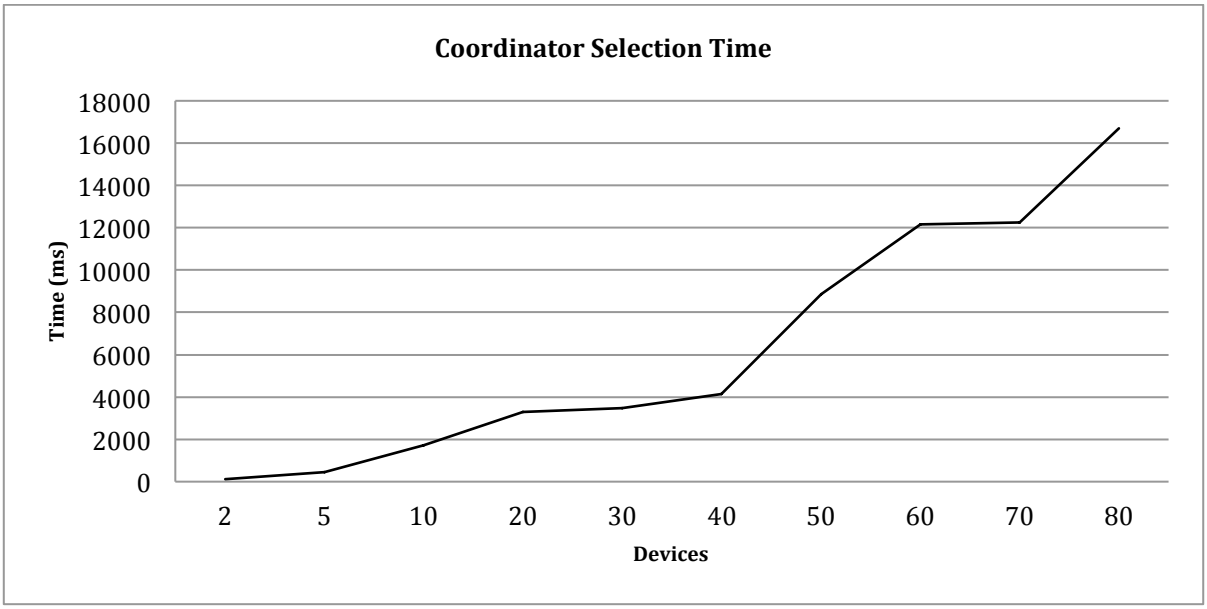


Figure 63 Time spent selecting a new coordinator

4.7 Setup Time

This experiment was done to measure the time it takes from starting a daemon until it has initialized and is ready to receive requests and synchronize with the other devices. The times were fetched using the same method and library as in the previous experiment (Golangs time library).

The set number of devices was started with one second apart. The experiment was executed with each daemon started 1 second after the previous one. The results shows an average of 1.58 seconds when setting up 200 devices, with a clear point at around 145 devices on which breakdown starts to occur. The lowest time is the first daemon at 51.9 ms, and the highest is the last with 15.3335 s. Also notice that there is almost a 100 % increase from the first to the second device (97.7 ms) from which the time is almost constant from device number two to device number 130 (112.8 ms).

You can start to see a slow ascent beginning at 115, increasing rapidly at around 145. That is a decent value for our purposes, as a break down at above 100 devices is negligible for a network of personal devices.

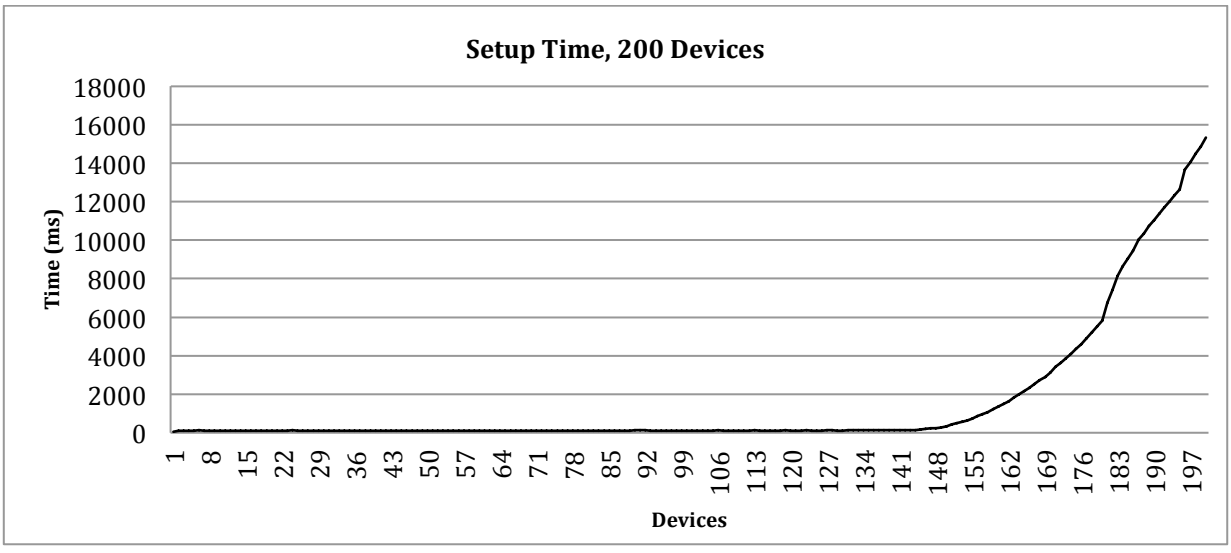


Figure 64 Setup time of 200 devices, an overview

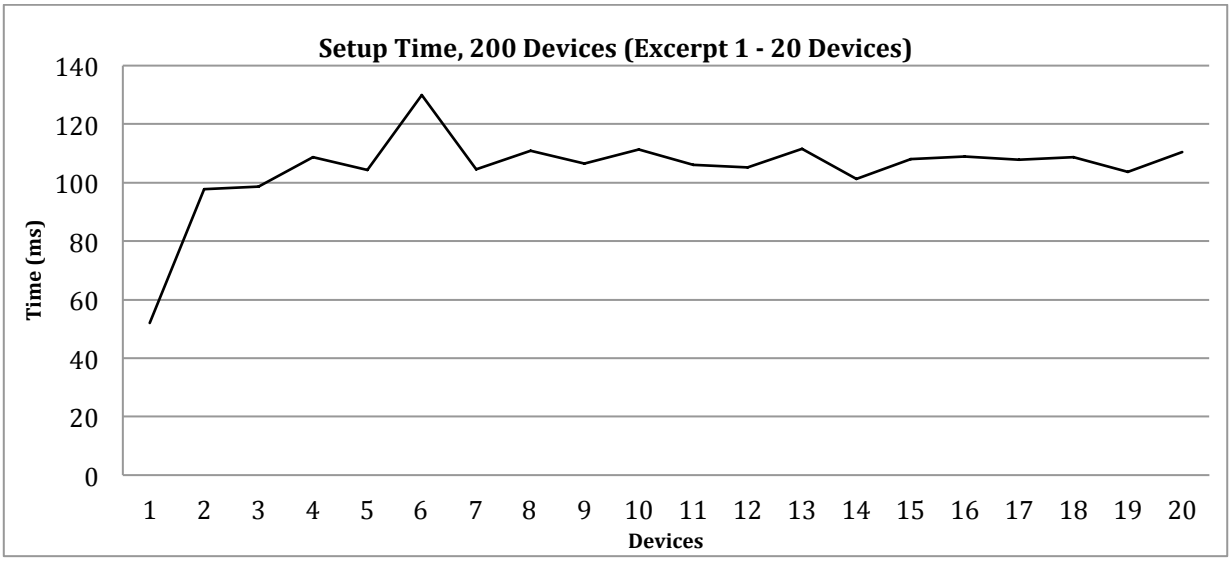


Figure 65 Setup time of 200 devices, excerpt of 1 to 20 devices

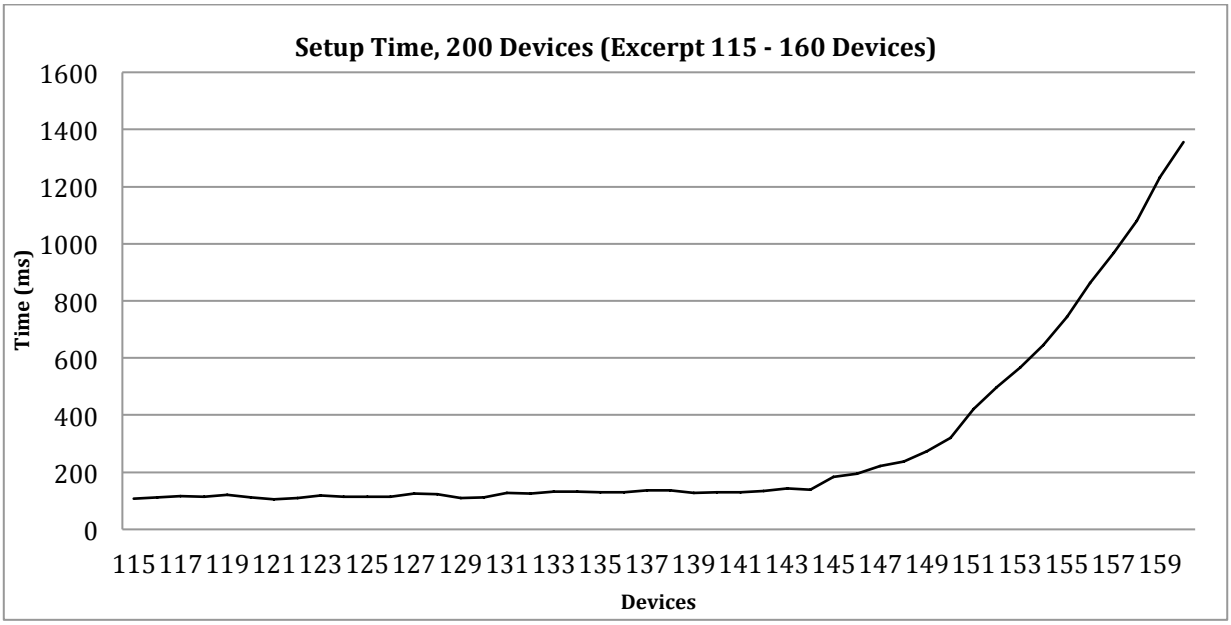


Figure 66 Setup time of 200 devices, excerpt of 115 to 160 devices

It is likely that it is a limit on the prototype in terms of communications as there seemed to be hardly any hit on the memory, and the CPU did not seem to be maxed out based on the values presented in the OS Xs Activity Monitor (CPU percentage per core, including Hyper-Threading):

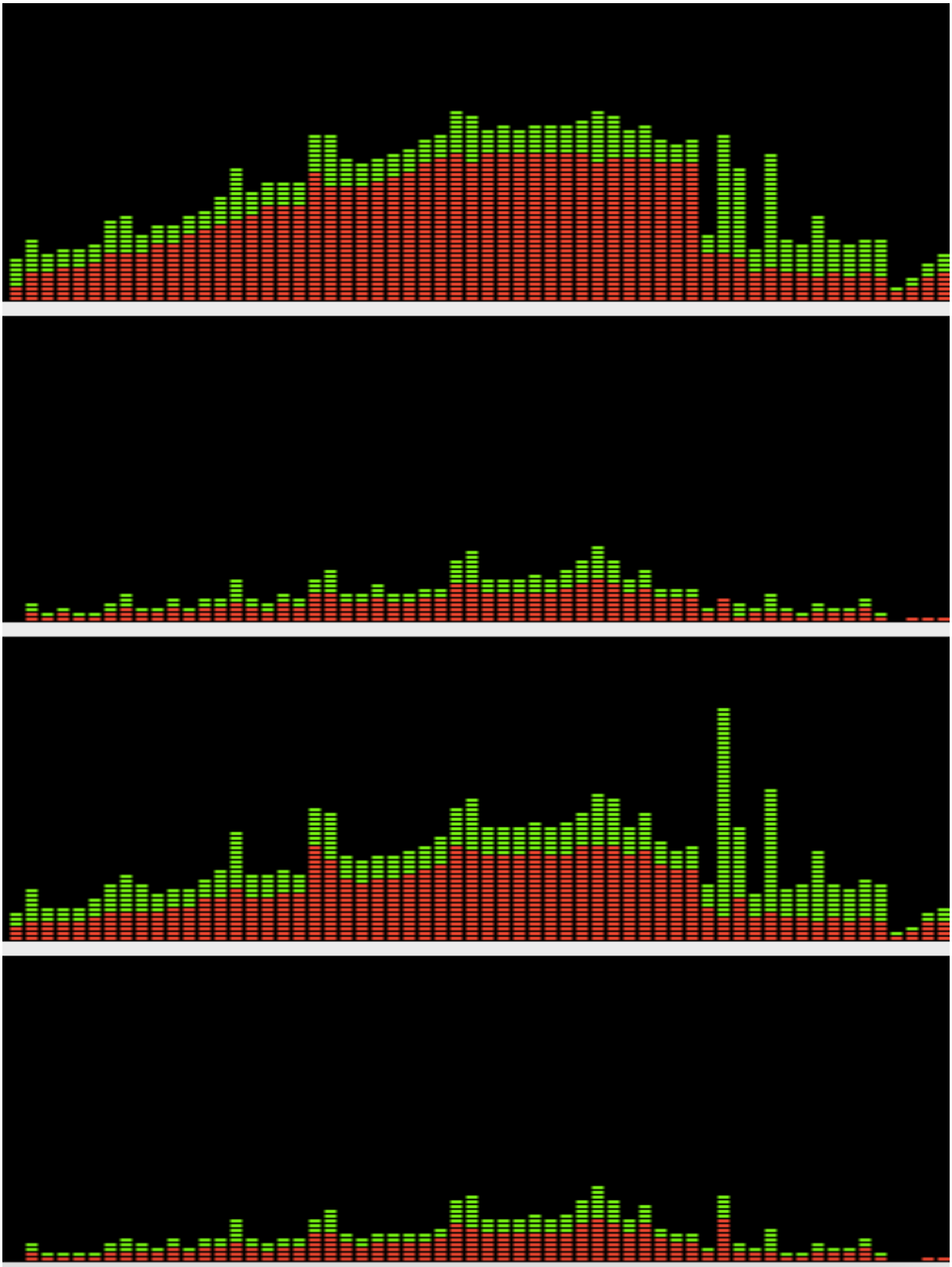


Figure 67 Total CPU usage according to Activity Monitor

4.8 Opening a File Remotely

The first experiment was designed to see how the size of a document affects the time it takes to open it in the remote access interface. A file of three different word counts was opened using the remote access interface; 200 word, 1000 words, 5000 words. No real variance in time spent opening the files nor amount of traffic was found. 200 words gave

a time of 3.5s, 1000 words 3.51s, and 5000 words 3.6s. The time was found by subtracting the time at the point when the file was opened from the time when the request was made.
Wireshark was used to capture the traffic, with the result amount of captured data virtually identical. This indicates, as expected, that the time is dominated by the work done to find the file and open the application.
In a second experiment, the time between the same points was measured, but a file was opened from the GUI at the non-coordinator with the file at the coordinator, and then the other way around. The times were again found to be around 3.5s, regardless of what device was the coordinator. The results indicate that the number of application runs is probably unnecessary to include when evaluating the devices for coordinator election.

4.9 Remote Access System

The update rate is set to 500 ms by default in the prototype, as this was found to be a decent balance between accuracy and resource utilization. In this experiment, the aim is to test the performance of the remote access system, as well as look at how the update rate affects accuracy and performance.
In this first part of the experiment, a document with a few thousand words was opened, and the GUI was moved through the document with a time of about 10 seconds per view/page to see the impact on the network. The traffic per view change/page flip can clearly be seen as spikes in the graph below. The average time spent processing a page flip request server side was 9.3 ms. The CPU and memory impact was not noticeable.

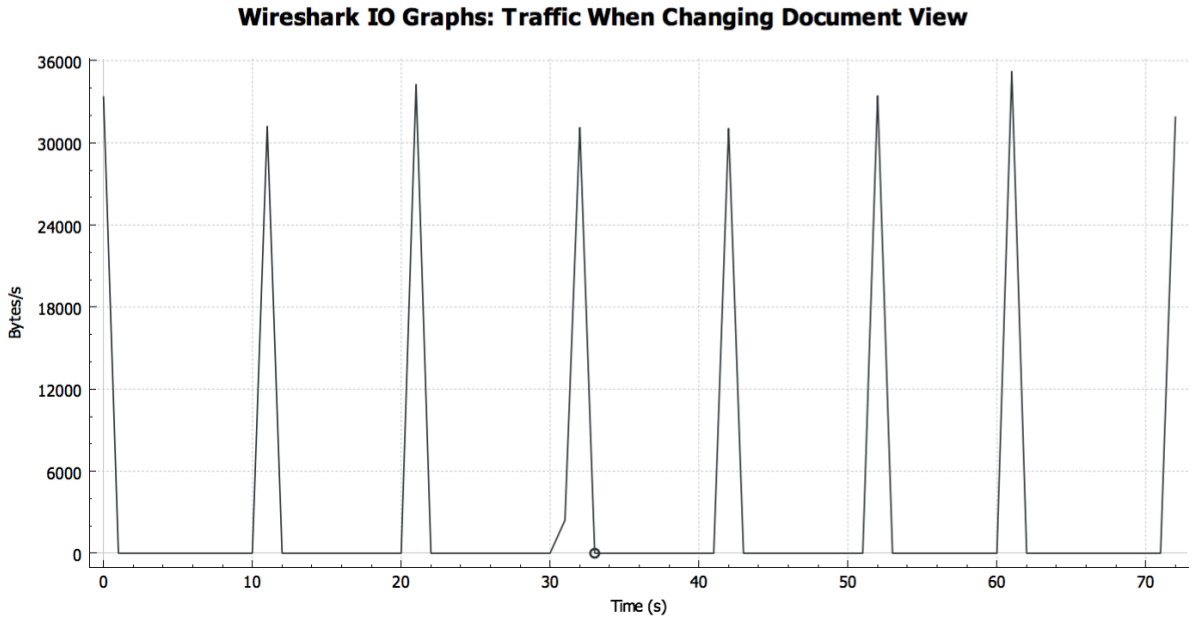


Figure 68 Traffic in B/s when changing document view

Next an attempt at measuring the time from editing the document to the change being applied was measured. Unfortunately, the times printed by the client-side javaScript was not in sync with the times printed by the server-side times printed by the application written in Golang (the changes are sent asynchronously). The values can however be used as a reference between the difference between the different lengths of changes, though the precise values are incorrect.

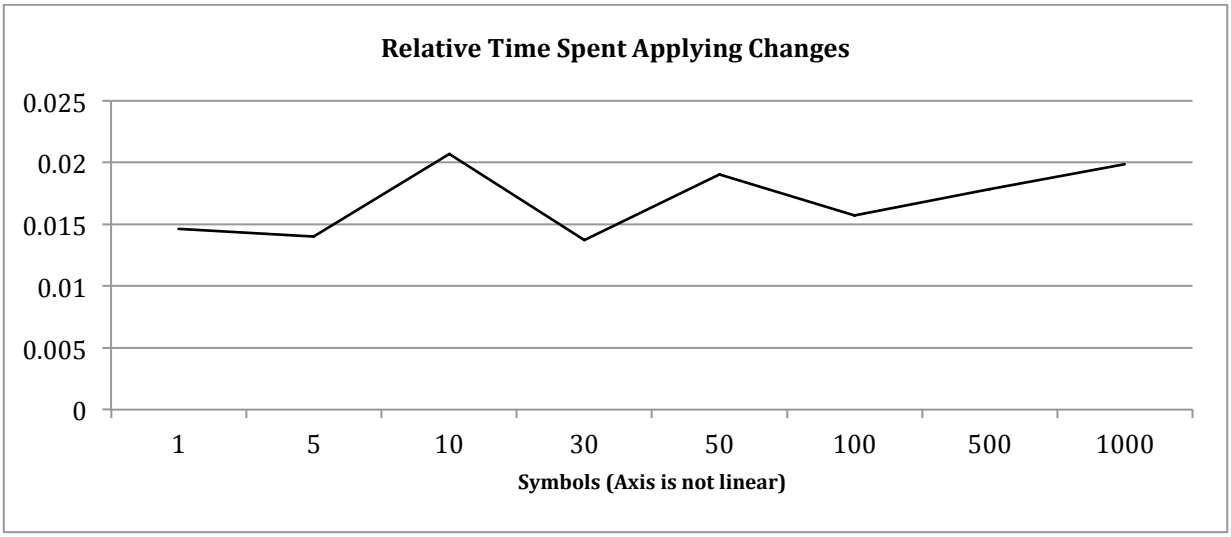


Figure 69 Relative time spent applying updates of different lengths. Note that the time times given at the Y-axis are not correct times, but merely relative values.

There does not seem to be any significant difference in time spent in relation to length of the change within range of realistic values. And the CPU usage was again insignificant, with the CPU peaking at 3 % at one point, but otherwise never exceeding 1 %. The network traffic and memory usage does however show some increase with length, but it is not a very big impact. We should expect some increase in time too when the length of the changes grow larger, but it is unlikely that a change done within 500ms is large enough to make a noticeable difference, unless very large portions of text is copied and pasted in.

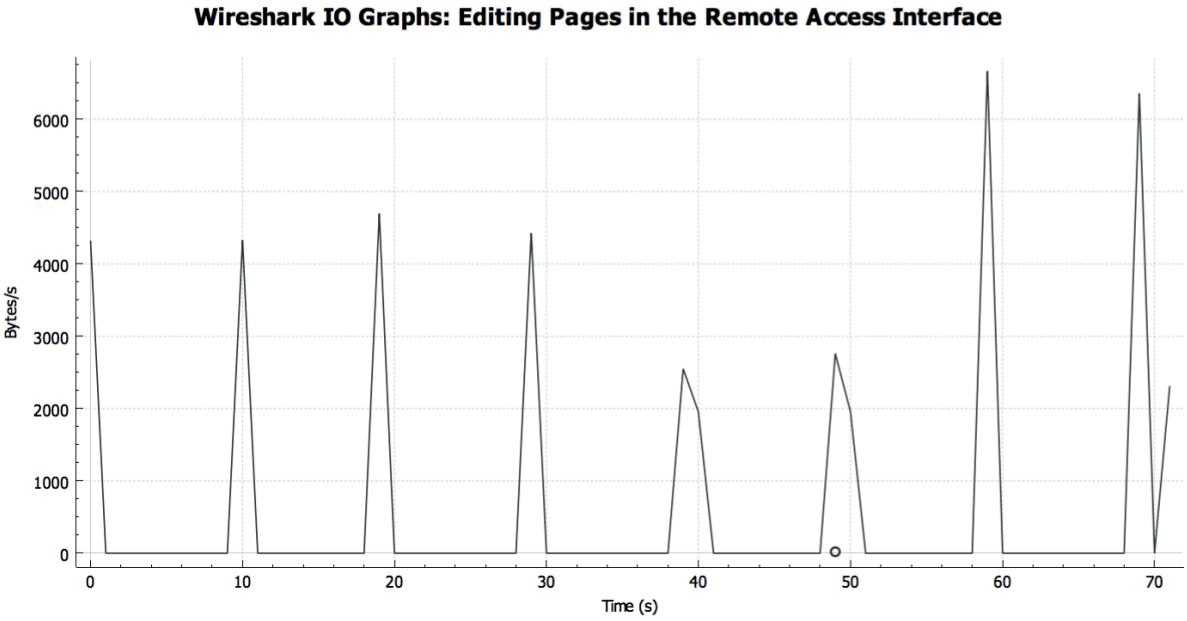


Figure 70 Traffic captured during editing of files through the remote access interface

The application was running under the daemon with port at 8590, while the GUI was at 8591. The symbols were added, without the set of symbols from the previous set of test being removed. Thus a build up is to be expected. The memory samples are not from the same run of the experiment as the Wireshark IO Graph, thus the variation in time stamps.

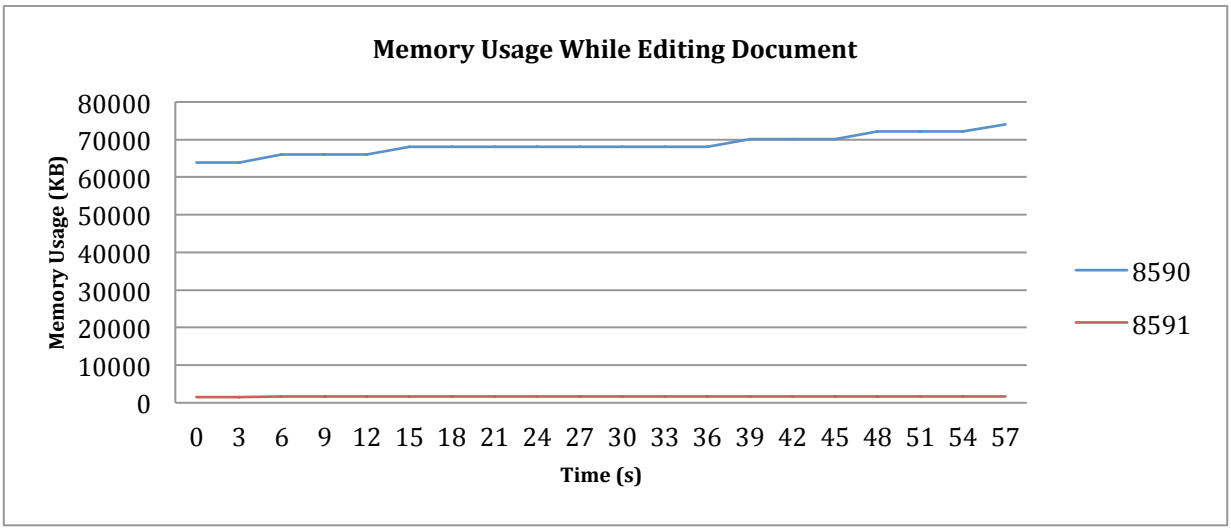


Figure 71 Memory usage during changing of a file through the remote access interface

All the changes applied in these tests seemed to be correctly applied to the actual file, likely because all the changes were additions inserted in the middle of sentences. The algorithm was noticed to behave abnormally when changes were done at the end of lines, at the end of the documents, or in combinations with removing text.

4.10 Synchronization Time

An experiment to check how the amount of files already present in the network affects the time of adding new files was run. The existing files were present on the first device with the daemon at port 8590, which was also set as the coordinator. The file was then added to the second device. None, 50, 250, 500, 1000, 2500, 5000, 10.000, 20.000, and 30.000 already existing files were tested.
The experiment was then re-run with the files present on the second device (same as the one the file is being added to).
The times shown are from when the request to synchronize the specified file is sent, to the change is committed and applied. As can be inferred from the data, the number of files already present in the system does not seem to affect the performance much (at least not when adding a single file), but which device the files is placed on does appear to have a significant impact. The first experiment has a much higher average then the second.
Finally, the second set of data has no points for higher number of files. This is because the prototype appeared to be too unstable when 10.000 files was reached with those settings, timing out when synchronizing a high number of files. The experiment then implies that the coordinator is more stable than the other devices, thus having the number of files affect which daemon is elected the coordinator is a good decision.

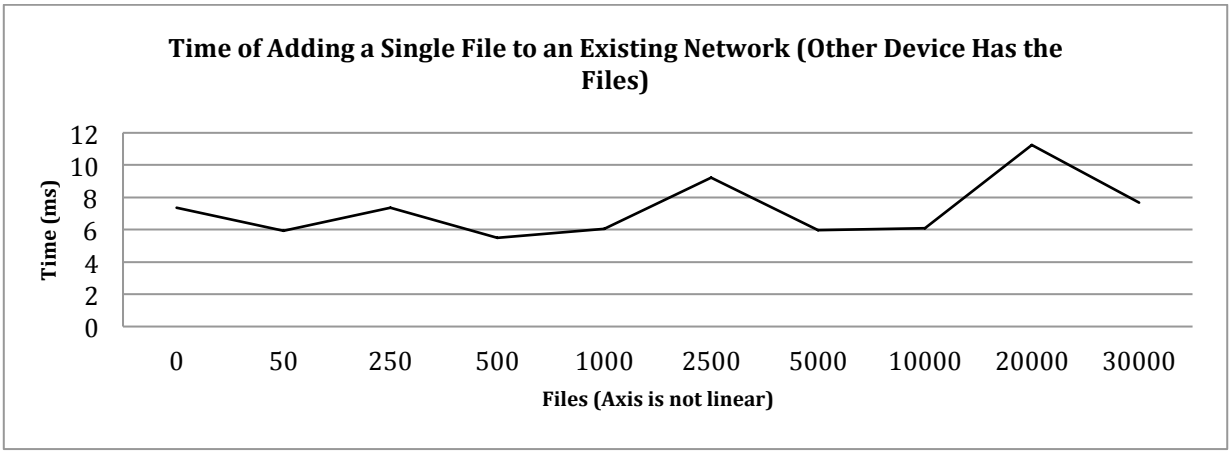


Figure 72 Time spent adding a single file to a network with a varying amount of files on the coordinator (the other device)

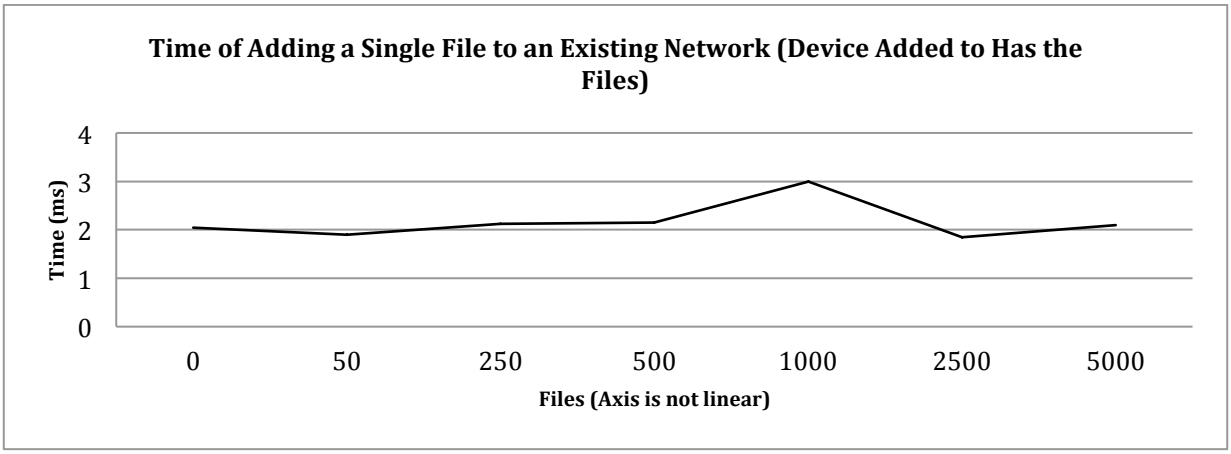


Figure 73 Time spent adding a single file to a network with a varying amount of files on the device the file is being added to

To look deeper into the difference between the times from the two experiments, and to confirm that the number of files already present in the network does not affect performance, an experiment with a slightly different twist was devised.
Here two daemons where the first is fixed to be coordinator are added 1, 50, 100, 500, 1000, and 1500 files. Both adding to the coordinator and adding to the second device was tested. The experiment was run twice, once with an empty network, and once where both devices had 500 files each. The time was then measure (in the coordinator) from the time a synchronization/refresh request was received, until all the changes were applied and pushed. The change in points of measure was to increase accuracy as this meant timing variables could be kept within a single function (in contrast to global variables), reducing the chances of unexpected factors affecting the result.
The numbers shown in the graphs are averages of five samples.

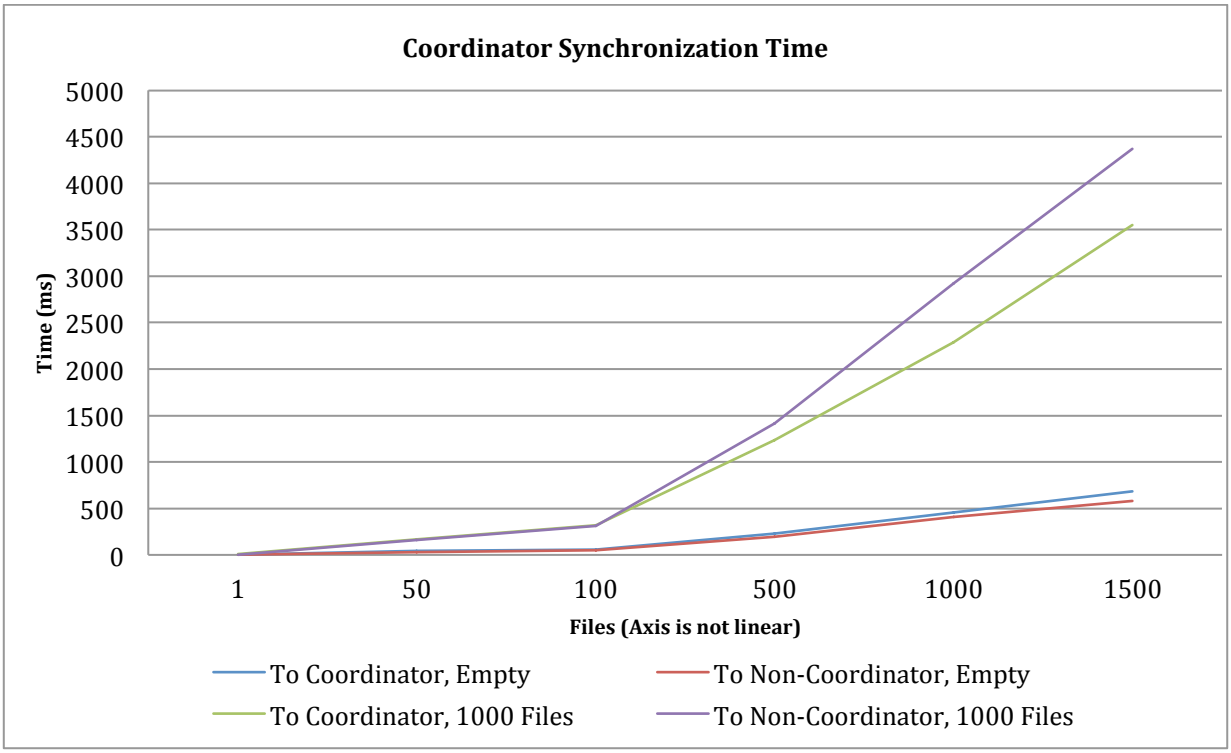


Figure 74 Time taken for the coordinator to enforce synchronization under various conditions

The new experiment shows that the number of files in the system clearly affects the performance when you start to synchronize more than a single file. However, when taking an excerpt of the lower end of the graph, we can see that this experiment is still in agreement with the previous one. When synchronizing only a few files, the performance hit is minimal. Furthermore, which device the files are added to (coordinator or not) seem to have a significant effect on the performance when there are files already in the network (same as previous experiment showed), but has no performance impact on an empty network. The average time for adding a file to the coordinator when there are 1000 files in the network already is about 9ms, while in every other case it is about 2ms. Thus it seems the coordinator is much more heavily affected by a high number of files than the other devices, when adding few files per synchronization. However this does not seem to be the case when a high number of files are added at a time. When more than a 100 files are added per synchronization, the coordinator is in fact more effective, but the trend of the coordinator being slower persists when both devices are empty! This means that deciding it is no solution in terms of where to put the coordinator that fits every case, it should rather be determined by the way the system is used. Either informing the user, and let him/her decide, or implementing a system that determines the most effective solution based on historical use.

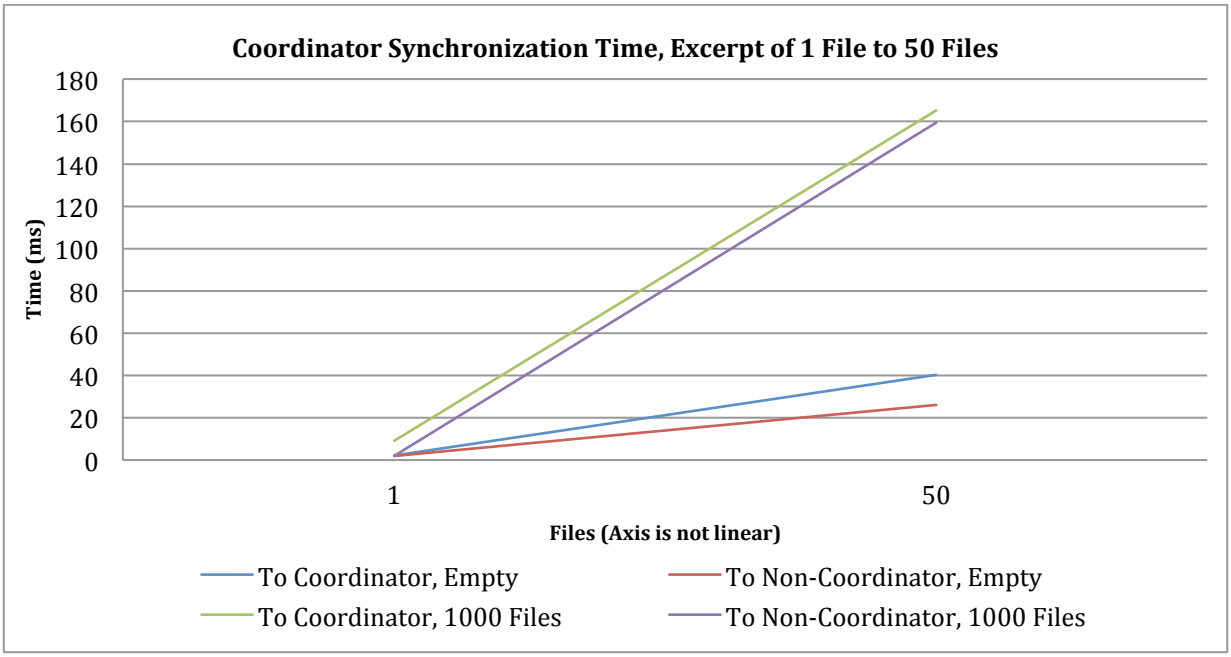


Figure 75 Excerpt of synchronization times

The last experiment heavily implies that which device contains the already existing files is not what mattered to the result in the first experiment, but rather which one was the coordinator. To test this, a third experiment with three devices was executed. In this, the second device had 1000 files, while the others had none. New files were then added first to device number two, then device number three (having the deleted the previously added 100 files first). If the times of adding to the two different devices were somewhat equal, that should imply that only the coordinator impacts the performance, while the location of the files does not. If they are significantly different, that means both location of files and coordinator has a significant impact.
The data is based on a single sample, as the difference was big enough that I did not see any need for multiple samples, except for the first data point from adding a single file. That value is based on an average of few samples to make sure there was no mistake due to its unexpected value. The data shows that when adding a few number of files in the synchronization operation, the empty device is a bit faster (2.2ms versus 4.6ms at a single file), but when a significant number of files is reached, the non-empty device operates quite a lot faster (442ms versus 41ms at 50 files)! We must then conclude that both file location and coordinator position has a significant impact on performance during synchronization, and that electing the coordinator to be the device where files are added most frequently is a good idea, if we can assume that synchronizations only occur when 50+ operations have been queued.

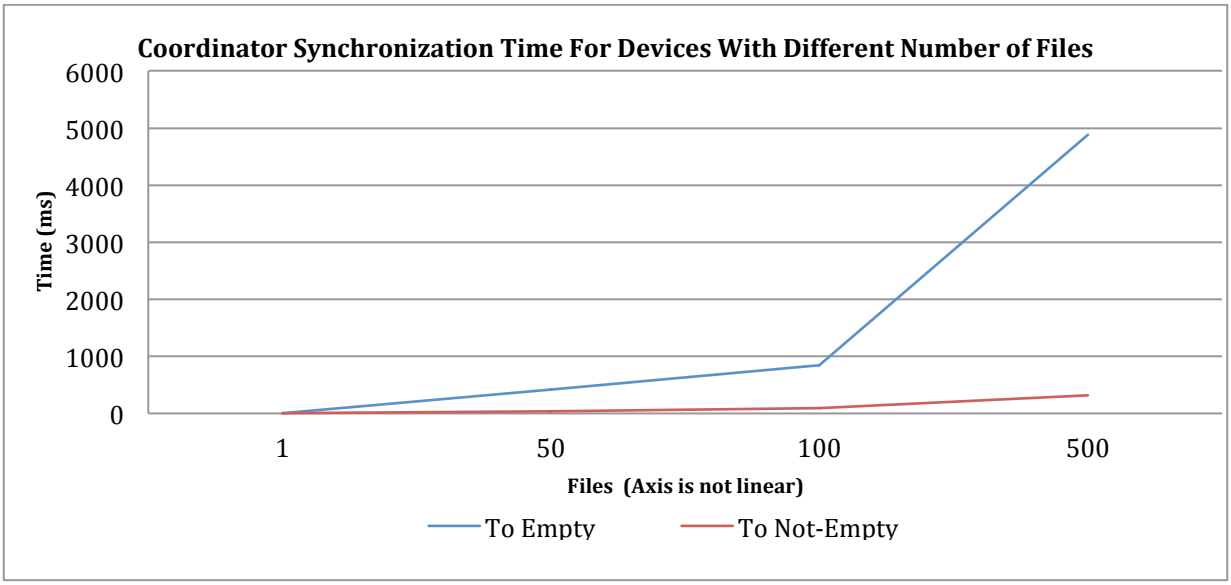


Figure 76 Time spent synchronizing for the coordinator when adding to either an empty or not-empty device.

Note that "ulimit -n" was increased from 256 to 4096 for these experiments in order to operate on higher number of files, and the experiments could not use higher numbers of files than what is shown because communications would break down without any meaningful errors. It is likely that too many concurrent TCP/IP connections caused something to go wrong with the TCP/IP server.
(Note that these experiments focused on data files. Applications are retrieved using OS Xs system profiler, which used a lot of time on synchronizing applications.)

4.11 Application Synchronization

In this experiment how the number of applications affect the time spent synchronizing was tested. To get a varying amount of applications, an application returning a set number of names for placeholder application files was created and used instead of the system_profiler. This application, the app list creator, was set to use approximately the same time as the system profiler before returning a list (6 seconds), as to simulate a realistic environment as close as possible. The time is taken from the point at which the coordinator starts to work on application synchronization, till it is done processing and sending local applications. This should include most if not all of the processing of application synchronization on all devices. Do note that all devices have the same applications in this experiment.
Three different sets of test were executed with the same number of applications: one with two devices, and one with ten devices, and one with 20. Communication was done over loopback to begin with.

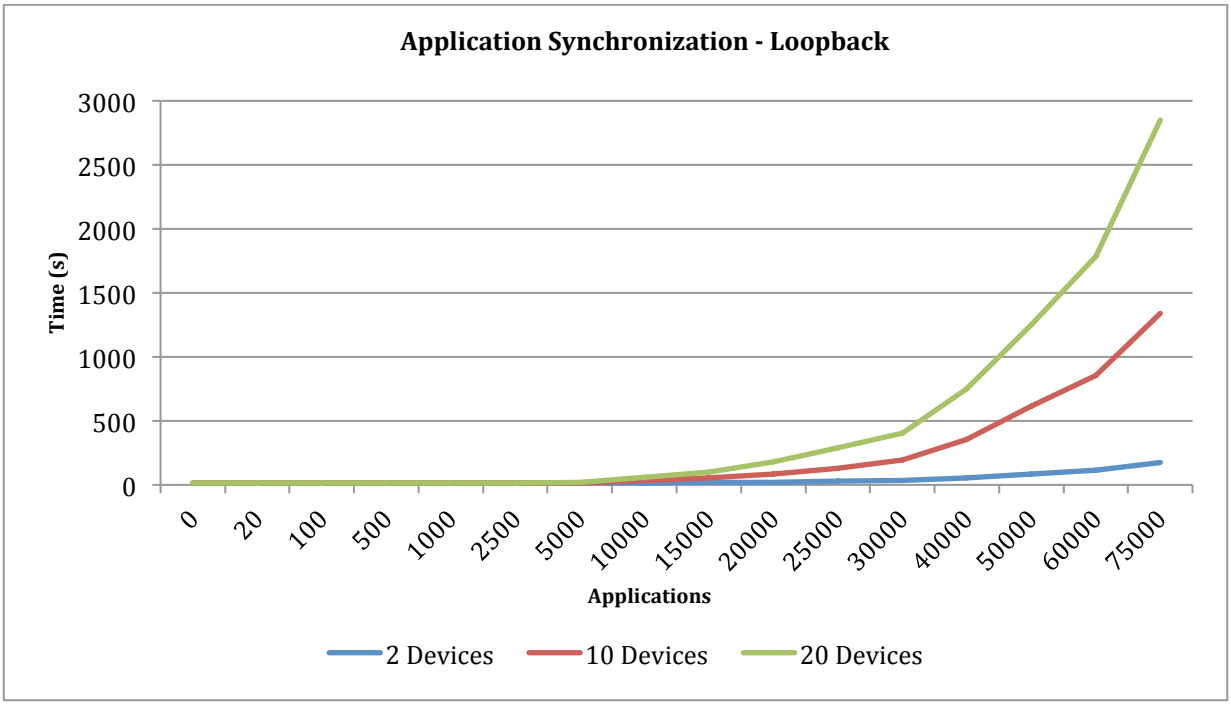


Figure 77 Time spent synchronizing applications over loopback

The experiment was then re-executed over Wi-Fi, with the packets sent via the router and back. Unfortunately, timeouts started to occur when the number of applications was high in combination with many devices, thus the lack of data for two of the lines.

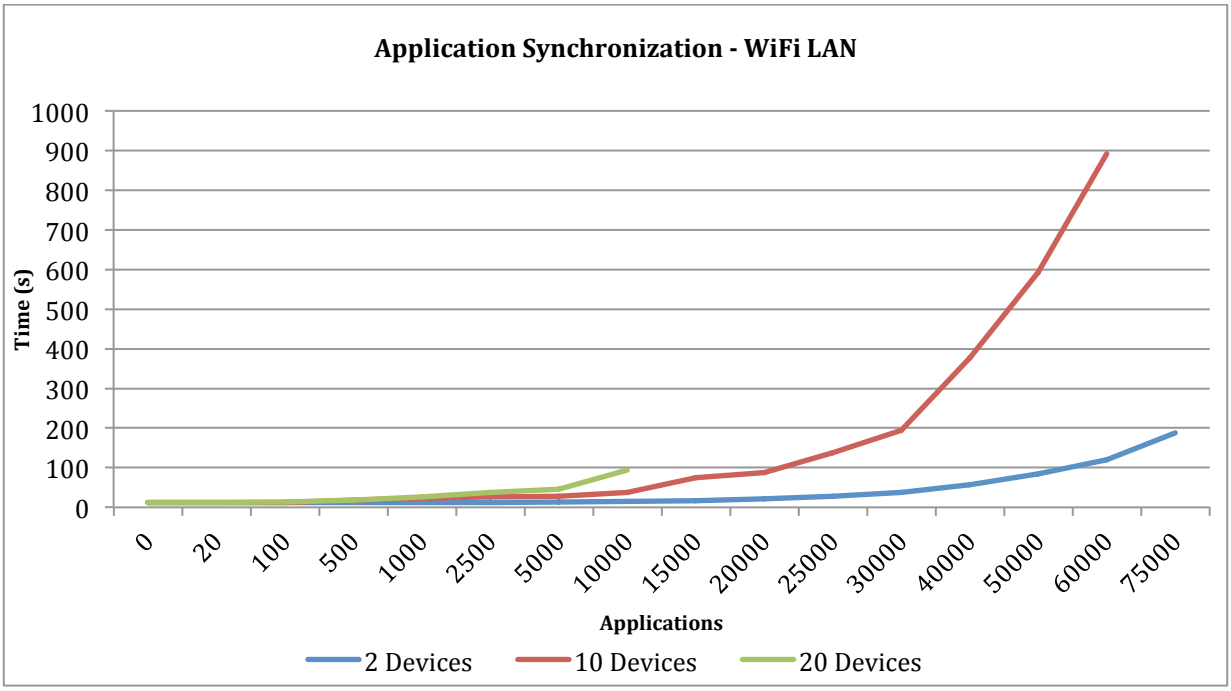


Figure 78 Time spent synchronizing applications with communications passing by the router on LAN

The graph for two devices appear to be quite similar between the two data sets, with both ending in the 180 seconds area with 75.000 applications, and with a similar incline. Much the same can be said for 10 devices, though the prototype timed out when attempting 75.000 applications. There is as expected a slight increase in time when

running over LAN, though quite small; An average increase of 4 % for two devices, and 5 % for 10 devices based on the values for the runs that finished properly.
On the other hand, 20 devices had a huge increase of 64 % based on the tests up to 10.000 applications. It is worth noting that the increase is most noticeable in the area around the beginning of the incline of the curve as the incline hits slightly earlier due to the increased time spent in communication. As the prototype times out just after the point of breakdown, the performance hit seems greater than it really would have been if it was capable of surviving. For comparison, when based on the same data points 10 devices had an increase of 40 %, while two devices had 1 %. It is thus reasonable to believe the difference would have evened out had the prototype been capable of continuing on with higher values.
Note that some significant fluctuation (though within reasonable intervals) in the times for 500 - 5000 applications when running 20 devices did occur when routing the packets over LAN. But they were inconsistent and the amount and direction varied from day to day, and between reboots. A variation in the networks environment is likely to blame.

5 Discussion

5.1 Coordinator Versus Complete Decentralization

A design with a coordinator enforcing consistency was chosen for the prototype. It is not clear that it is the best design, as it comes with a couple of negative effects; In the real world, failure and connection break down is a likely event. Having a solution that tends towards a centralized design creates a single point of failure unless good solutions for handling these scenarios are created. And even if you have a good way of surviving these problems, network splitting would create further difficult challenges to tackle.
However, the design used in the prototype does give simplicity and ensure synchronization, more so than what would have been easily implemented with a completely decentralized system. The system only every being used by a single user with his/her personal devices also significantly reduces risks of failures and conflicts. It also implies that scalability is not important, as the number of devices involved is likely be in the tens at most.
The final implemented prototype does not handle these scenarios very well, but it should not be very difficult to ensure that it does. The prototype is currently set to exit when a communication error occurs, which could be changed to just cancelling the operation and re-evaluating the connections to the other daemons. A new coordinator can be chosen if the previous one was lost, and the state of the entire system re-evaluate as if it was setting up for the first time. When a daemon comes back online, any changes to files that were saved to disk but not committed before the crash would be applied. The challenge would be changes to metadata. As the prototype stands, metadata of devices that were lost would also be lost during re-evaluation. A possible solution is to let the new coordinator keep the view it had prior to the crash, and if none of the daemons lay claim to files that it does not have locally either, the files are assumed to be out of reach until a the device reconnects or a device lays claim to the files.
Most of these points were originally done and the functions are to some extent there, but where deactivated and set to its current state during debugging, testing and experimentation to make it more transparent.

5.2 OS and Applications Interface

A problem for a system like this is how to efficiently notice changes in files. In this thesis, we let the user interact with our own interface in order to have full control over which applications and files the user can see, as well as keep track of what files and applications are opened. It is however necessary that the applications run have an interface for interacting with the system, otherwise the remote access functionality would not work, and the system would be unable to detect changes done by local applications. Thus the prototype only allows for opening files that may be opened in the text editor created for this project. In other words taking a solution that is explicitly not transparent to the applications.
It is clear that any application that is to be used remotely must have an interface that lets the networked system communicate with it; however, it should be possible to enable other applications that are only to be run locally. Currently the problem is that there is no easy way for the SDTPS-system to track changes applied to files by the only local applications that do not communicate with our modules. We could of course scan the entire FS for changes, but it is a very inefficient solution and would eat up resources. A better solution would be a subscribe feature implemented in the OS. The processes could let the OS know that it is paying attention to specific files, thus whenever a change

occurs in a file, the OS triggers an event letting the processes know about the change. This way only the applications the user needs to run remotely needs to be changed to have an interface for communicating with the SDTPS-system, while whenever all other applications apply changes, the OS automatically lets the system know. Such a feature would be generally useful in several use cases, not just for a device transparent system. For example an image catalogue application could subscribe to the images the user has added and automatically update its catalogue whenever an image is changed, while just keeping all the other images as they were. There would be no need to use resources on going through all the folders and catalogues in the system except for the first time when the user adds them.
Another implicitly transparent solution with regards to the application would be to alter the libraries the application makes use of for saving, having all saving operations go by our own code.
A graph showing the flow of changes in the current prototype was given under the implementation section. The proposed changes would enable an implementation with a flow similar to one of the flows in the following graph:

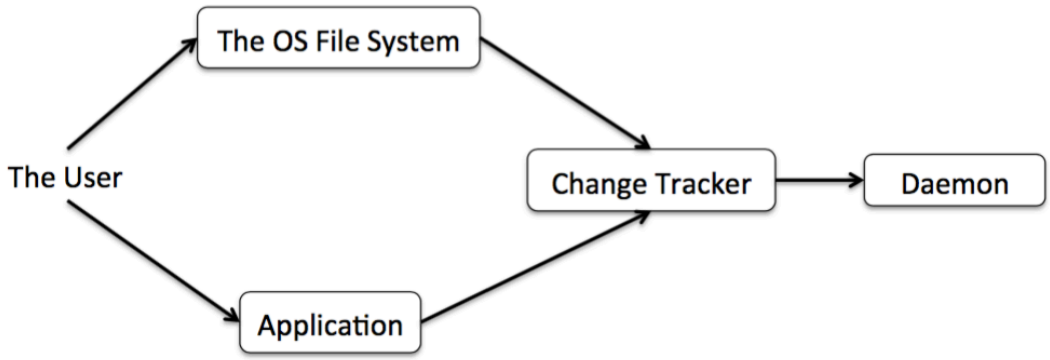


Figure 79 Possible change tracker flows given a file subscription system or editing of saving libraries

5.3 Automatic Versus Manual Refresh/Synchronization

The prototype makes use of a manual refresh function, which there is many ways of implementing. The prototype is implemented in such a way that a "profane" daemon may not commit updates without the coordinator/consistency enforcers' permission. Thus the system could push the updates to the consistency enforcer as soon as they are done and let the enforcer put them in a queue, then when the user refreshes the changes are pushed to the entire network. Or the changes could be queued locally on each daemon, and then pushed to the consistency enforcer whenever a refresh operation is made.
A third option is to let the profane daemons commit changes locally and thus create a local view that is merged with the other devices upon refresh.
The first option ensures high consistency. All changes across the network are applied in chronological order on all devices. The amount of traffic and extra memory used in the consistency enforcer would be linearly related to number of changes, if the changes are few between each refresh, so is the amount of traffic and size of extra memory used. However, if a huge amount of changes are done between each refresh, so is the amount of traffic and size of memory used.
The second option does not guarantee that all changes from different devices are applied in chronological order, but all changes from a single device will be applied

chronologically. Thus consistency should still be enforced, as a file may only be stored at a single device at a time, meaning all changes for a single file will be applied in order. This option has much the same characteristics as the first option; only difference is the memory usage is not concentrated at the consistency enforcer.
The third options offers a more predictable and likely larger amount of traffic, but probably less extra memory overhead as it is only necessary to keep track of what files are present locally, not the changes in their entirety. It would give an amount of traffic proportionate to the number of files across the networked devices, thus as long as the number of changes between refreshes are fewer than the number of files across the network, either one of the two first options will have less traffic overhead. But it is possible to reduce this traffic overhead somewhat by not refreshing the SSV of every device when a refresh request is received, only the devices the request came from. This is more difficult for the first two options as, as it is the change that is sent when synchronizing, not the new view. The coordinator would have to keep track of which devices have received each update, and keep it until every single devices is synchronized. The prototype makes use of the second method during normal synchronizations, but the third method when a new daemon joins.

Automatic refresh, or simply continuous synchronization is also a possibility. Meaning whenever the change detector sees a change in a file or in the application list, the local daemon pushes the update right away. The problem with this option is the likelihood of frustration for the user if resources and network traffic slows him/her down with the user having little control over it. Letting the user decide when the system synchronizes ensures that the user has increased control over his/her own resources and that it does not affect the performance of the devices and the network at an inconvenient time.

5.4 Remote Application Access

The remote application access is not a new topic, but the aim during work on this project was to look at it from a fresh angle. The thought behind the solution used in the prototype was to avoid having to transfer a copy of the file from the remote host to the local device where the user is working. Instead the goal was to work with the file on the host and apply the changes locally where the file habituates. One way would be to send a picture of the application interface; similar to the way X Window does it [18]. That would however be a very inefficient solution in many cases, thus a more tailored approach is likely to reduce the traffic load significantly. A local GUI communicating with the remote device, running the actual application is a much more efficient solution with for example text processing. The remote host can be limited to only sending the part of the file that is actually visible on the GUI the user is interacting with, thus avoid sending the file in its entirety. Operations can then be sent from the GUI the user is interacting with back to the application where they are applied on the original file. The prototype tries to find the position of each change, and sends the position and the operation requested to the application where the application applies the changes to its local file. However, there are cases where different approaches must be taken, such as image post processing. Sending the image over to the GUI might be necessary in some cases, but the core of applying the operation on the file at the device where the file resides can still be kept. In fact in the case of editing an image, just sending a picture of the entire GUI running the application might be the most efficient solution.
But for this to be feasible in the real world, any application that is to be run remotely by merely sending operations and small intervals of a file must have an API for

communicating with the remote access system. In other words, corporations such as Microsoft and Apple would need to agree to implement such APIs, or a reliance on equivalent open source applications would be necessary.

5.5 Security

Any system that connects several devices together will have security issues. With this system, the threat of someone using it to hack your personal files is of course always there, and theft is a real possibility.

A hacker could easily figure out the ports the prototype is using, and get a way in. The daemon can then be dissected in memory using a debugger to find valid commands, as they are hard coded strings constants. The attacker at this point has full access to the victims' files, and potentially applications. All communications are also in plain text. Obviously any real implementation must handle communications more carefully; though a cabled setup would increase the security of the network, but hinder the use cases drastically.

A thief would be able to access all your files on all your devices easily, and launch applications on devices that are still in your possession.

It is clear that anyone attempting to implement and deploy a similar system would need to take security measures.

A feature allowing the user to kick devices out of the network of devices in cases where a device is compromised and the owner has no physical access to it would be very useful. Though any unwelcome person could also abuse this feature with access to one device, so perhaps it would be necessary to either identify the owner through a password, or use a quorum system so that you need physical access to more than one device to commit administrative changes.

It is also possible that it would be best to limit what type of applications the system allows the user to run remotely. Perhaps it should only be possible to run applications that are designed for processing files, such as photo editing and text processing. Access to the OS' administrative application could cause serious harm to all a persons devices if only a single device is compromised.

6 Further Work

6.1 Relocation and Device Transparency

For a distributed storage to work in the real world, complete device transparency is unlikely to be feasible. Devices disconnect, get turned off, or become out of range in reality, and complete transparency may thus cause files that are essential to the user to become unavailable. Thus users will almost certainly want some control over where specific files are, or at least specify where to store groups of files. A management system similar to that of *Perspective* [6] would allow for more control in line with this thesis objective.

An interface where the user can view specific files or group of files in a tree of folders under the devices would allow for more specific and intuitive control, however as stated in [4], the user is rarely very good at predicting which files he/she is likely to use on what device, thus an automated AI system for example based on a neural network is feasible. The system could then be taught were to put certain files while there is full connectivity, later giving the user access to the files he/she wants when there is no connectivity while still being fully device-transparent to the user.

6.2 Completely Decoupling Application from Device

An interesting topic to do more research on would be the possibility of handling applications in the same way as you handle files normally, as in moving applications between devices similarly to how you can move files between devices. The design discussed in this thesis only decouples access from device, but not functionality, in that the data file and the application files must be on the same device. To achieve the highest level of transparency, data file and application files should both be decoupled from the union of device and each other.

There are of course some issues that would arise, such as both software and hardware compatibility, but some of the problems could possibly be solved through virtual machines and emulation etc.

There are some software developers who have tried to explore this concept to some extent, but their systems seem limited and appear lack fluidity. [34]

Other solutions could also be looked into, such as designs similar to that of the one discussed in this thesis with the users device connected to the device with the application, which in turn is connected to the device with the data file. This could also prove difficult to implement in an elegant way, as the file in its entirety would likely have to be given to the device with the application, breaking with part of the philosophy behind this thesis.

6.3 Pick and Choose Application

A necessary feature for a distributed personal system aiming to incorporate applications would be to allow for opening of files in different applications. In this project only implementation of a single application was done, thus the consequences of having more than a single application capable of opening a file was not thoroughly explored. A scheme for deciding which application to use in different scenarios must be made, and the possibility of letting the user choose what application to open a given file in could be very useful.

6.4 Caching

The efficiency of the system could likely be increase by the use of caching. For example, daemon through which a file has been opened recently could keep the position of the file so querying the coordinator when the file is reopened is unnecessary.
Caching of pages/views in the remote access interface also has some potential, and in fact even pre-fetching is possible here. When the interface is started, the local daemon could keep track of previous pages and store them locally while the file is open. At the same time, the next page could always be fetched and kept in memory in case the user wants to advance through the file. Caching of pages would likely reduce the perceived loading time of each page radically, but would also require redesigning of the application.

7 Conclusion

TODO

References

[1] Cisco. 2015. Visual Networking Index. Available at: http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html#~completeforecast. [Accessed 24. February 16].

[2] Google. 2016. World Development Indicators. Available at: https://www.google.no/publicdata/explore?ds=d5bncppjof8f9_&hl=en&dl=en. [Accessed 24. February 16].

[3] Julie Bort. 2013. We'll each have 5 Internet devices (and more predictions from Cisco). Available at: http://www.businessinsider.com/cisco-predicts-mobile-2013-5?op=1&IR=T. [Accessed 24. February 16].

[4] Jacob Strauss et al, (2011). Eyo: Device-Transparent Personal Storage. In Usenix Annual Technical Conference. Portland, OR, USA, 17th June. Berkley, CA, USA: USENIX Association. 1-14.

[5] Tej Chajed et al, (2015). Amber: Decoupling User Data from Web Applications. In HotOS 15. Kartause Ittingen, Switzerland, 19. May. Berkley, CA, USA: USENIX Association. 1-6.

[6] Brandon Salmon et al, (2008). Perspective: Semantic Data Management for the Home. In CMU-PDL-08-105. Pittsburgh, PA, USA: Parallel Data Laboratory, Carnegie Mellon University. 1-22.

[7] Computer Hope. Distributed systems definition. Available at: http://www.computerhope.com/jargon/d/distribs.htm. [Accessed 2. March 16].

[8] Otto J. Anshus, (2015). Lecture on Transparancy & the Definition of Distributed Systems. In COMP3200, Distributed System Fundamentals. Tromsø, Norway.

[9] The Go Programming Language. Documentation. Available at: https://golang.org/doc/. [Accessed 3. March 16].

[10] Texlution. Why Golang is doomed to succeed. Available at: https://texlution.com/post/why-go-is-doomed-to-succeed/. [Accessed 3. March 16].

[11] Google Trends. Golang. Available at: https://www.google.com/trends/explore#q=golang. [Accessed 3. March 16].

[12] w3schools.com. Javascript Introduction. Available at: http://www.w3schools.com/js/js_intro.asp. [Accessed 6. March 16].

[13] Douglas Crockford. A Survey of the JavaScript Programming Language. Available at: http://javascript.crockford.com/survey.html. [Accessed 6. March 16].

[14] Qt. Qt Documentation: QML Applications. Available at: http://doc.qt.io/qt-5/qmlapplications.html. [Accessed 6. March 16].

[15] GitHub. QML support for the Go language. Available at: https://github.com/go-qml/qml. [Accessed 6. March 16].

[16] DevNation, YouTube. DevNation 2015 - Vincent Batts - Golang: The good, the bad, & the ugly. Available at: https://www.youtube.com/watch?v=cMYhGNofHA4. [Accessed 10. March 16].

[17] Computer Hope. TCP/IP. Available at: http://www.computerhope.com/jargon/t/tcpip.htm. [Accessed 11. March 16].

[18] X.Org Foundation. X.Org. Available at: http://www.x.org/wiki/. [Accessed 13. March 16]

[19] Apple. MacBook Air (13-inch, mid 2012) - Technical Specifications. Available at: https://support.apple.com/kb/SP670?locale=en_US. [Accessed 10. April 16]
[20] Apple. OS X Man Pages: route. Available at: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man8/route.8.html. [Accessed 10. April 16]
[21] Apple. OS X Man Pages: top. Available at: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/top.1.html. [Accessed 10. April 16]
[22] Apple. OS X Man Pages: ps. Available at: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/ps.1.html. [Accessed 10. April 16]
[23] TechTarget - SearchNetworking. Definition: routing table. Available at: http://searchnetworking.techtarget.com/definition/routing-table. [Accessed 10. April 16]
[24] GitHub, Speedlimit. Issue: Not work on Yosemite OSX 10.10. Available at: https://github.com/mschrag/speedlimit/issues/13. [Accessed 10. April 16]
[25] Apple Developer Downloads. Hardware IO Tools Search. Available at: https://developer.apple.com/downloads/?q=Hardware%20IO%20Tools. [Accessed 10. April 16]
[26] Wireshark. Download Wireshark. Available at: https://www.wireshark.org/#download. [Accessed 10. April 16]
[27] Wireshark. Wireshark Frequently Asked Questions. Available at: https://www.wireshark.org/faq.html. [Accessed 10. April 16]
[28] FileCatalyst. Today's Media File Sizes - What's Average?. Available at: http://filecatalyst.com/todays-media-file-sizes-whats-average/. [Accessed 11. April 16]
[29] Apple. OS X Man Pages: system_profiler. Available at: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man8/system_profiler.8.html. [Accessed 15. April 16]
[30] Apple. OS X Man Pages: pmset. Available at: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/pmset.1.html. [Accessed 15. April 16]
[31] w3school.com: HTML Introduction. Available at: http://www.w3schools.com/html/html_intro.asp. [Accessed 17. April 16]
[32] Webopedia: HTTP. Available at: http://www.webopedia.com/TERM/H/HTTP.html [Accessed 19. April 16]
[33] Wikipedia: Hypertext Transfer Protocol. Available at: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol. [Accessed 19. April 16]
[34] The Guardian: How can I move my files and programs to a new PC? Available at: https://www.theguardian.com/technology/askjack/2015/sep/17/move-files-programs-to-new-pc-windows-10. [Accessed 19. April]
[35] Karen Bjoerndalen, (2014). The Distributed Personal Computer. UiT The Arctic University of Norway. Tromsoe, Norway, 15th January. Tromsoe, Norway: Masters Thesis, UiT The Arctic University of Norway. 1-58.
[36] Joe McKendrick. 2013. 10 Quotes on Cloud Computing That Really Say it All. Available at: http://www.forbes.com/sites/joemckendrick/2013/03/24/10-

quotes-on-cloud-computing-that-really-say-it-all/#6441bfff2102. [Accessed 3. May 16].