Faculty of Science and Technology
Department of Computer Science

DPC

The Distributed Personal Computer

—

Karen Bjørndalen
INF-3990 Master's thesis in Computer Science - January 2014

Abstract

Nowadays people have many different personal devices, like laptops and tablets, that they use to access and process data. Very often it is desirable to access and process the same data on different devices without having to copy it from one device to another. Commercial cloud services provide good services for achieving this, but recent events, such as the Snowdon disclosures, have illustrated some of the trust issues of using external services.

This project introduces the Distributed Personal Computer (DPC), which aims to give a single system view of the user's personal devices without the use of external services. The DPC is meant to be for a single user with multiple devices. A prototype has been designed and implemented, and experiments have been conducted to evaluate the prototype.

The implemented prototype, and the experiments conducted on it, show that the concept of the DPC is worth pursuing further. The experiments show that the operation overhead is small enough to allow several hundred operations to run per second, and that the architecture and prototype for the DPC appears to be good enough for personal use.

Acknowledgements

I would like to thank the following people:

- My advisor, Professor Otto Anshus, for the original idea of the DPC, useful guidance, encouragement when I needed it, and patience.
- My husband, John Markus Bjørndalen, for many useful discussions and all his support and encouragement.
- Jan Fuglesteg, for encouraging conversations and all his assistance with the practicalities around being a master student.
- Ole Martin Bjørndalen, for introducing me to Bottle and letting me use his JSON-RPC implementation for Bottle.
- My fellow students for many fun discussions and distractions.
- My daughters and stepson, for putting up with their preoccupied (step)mother.
- and many others!

Contents

Chapter 1

Introduction

Nowadays people have many different personal devices, e.g. laptops, tablets, smartphones etc., that they use to access and process data. Very often it is desirable to access and process the same data on different devices without having to copy it from one device to another. For example one could start writing a document on a stationary computer, then for some reason need to move elsewhere and continue working on it on a tablet or laptop. Having the data stored in a common place where it can be accessed and worked on from any device is therefore very useful.

Commercial companies, like Google, Apple and more, provide good services for achieving this, so is there a need for something else? The answer to this question is a matter of trust, and raises another question: Can people trust commercial companies with their data?

Even before Snowdon's[1] recent disclosures about the NSA[2], people have been unwilling and/or hesitant to entrust their data to commercial companies for various reasons, among them trust. After these disclosures it has become obvious that people can't be ensured of the security of their data when using services provided by commercial companies, even if they trust the company providing the services.

People still choose to use these services, but there is an obvious need for a system like these where they have full control of their data and the services they use to process this data.

[1]http://en.wikipedia.org/wiki/Edward_snowdon
[2]http://en.wikipedia.org/wiki/2013_Global_surveillance_disclosure

1.1 The Distributed Personal Computer

This project introduces the Distributed Personal Computer (DPC) as a means of providing this kind of a system where a person can have the required control. It builds and expands on the work done for the author's individual special curriculum [1], where a personal system for storing and synchronizing data from different devices and applications was designed and a simple prototype implemented.

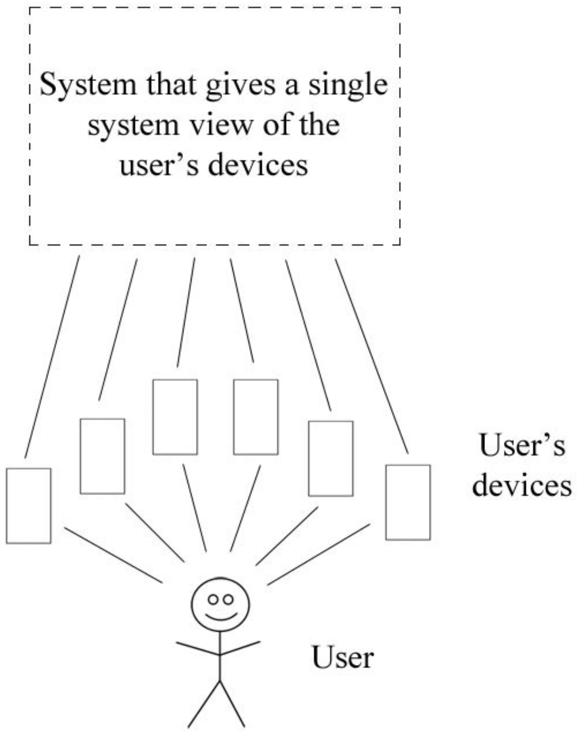

System that gives a single system view of the user's devices
User's devices
User

Figure 1.1: *The idea of the DPC*

The idea of the DPC is to give a single system view (SSV) of the user's personal devices with a system that the user has full control over. An SSV gives the appearance of the user's devices all being part of a single system, the user's data can be accessed with different devices as if the data and the devices were all part of a single system. Figure 1.1 illustrates the idea of the DPC.

The DPC is intended to be a single user system, i.e. the system is used and run by one single person. Because of this, scalability and security is a lot easier than it would be in a multi user system.

The user has full control over the system and doesn't need to rely on services provided by external providers. Therefore the DPC can be a self-contained private cell and in the extreme case it could all be in-house without any connections to the outside world. By connecting the different parts with cables instead of wi-fi the user can avoid monitoring of data (sniffing) in transit.

1.2 Assumptions

To limit the scope of the project the following assumptions were made:

- There is more storage than needed and it is always available.
- All parts of the system can be reached through a LAN
- Application data on the devices is available when needed.

Chapter 2

DPC Architecture and design

2.1 Architecture

The DPC consists of a client application that is to be run on the user's devices, a server that the client applications communicate with, and extra services that are connected to the server. The client application is mainly the interface to the user, it provides the connection between the devices and the server. The server is the biggest part of the DPC, without it nothing much will happen, it gives the client access to the functionalities needed to give the SSV. The extra services provide most of the functionalities mentioned above.

The client doesn't communicate with the extra services, it only communicates with the server. Because of this it makes no difference to the client where the functionalities it uses are. They could be run directly on the server, or on one or more extra services, and the extra services could be run anywhere, on the same machine as the server or elsewhere. Thus the architecture of DPC facilitates the possibility of the extra services being where one wants. They can be mapped to the computers one wants.

Figure 2.1 gives an overview of the architecture of the DPC.

In this system the server is a single point of failure, the system will not work without it. This means that one is dependent on having a computer with the server up and running available at all times.

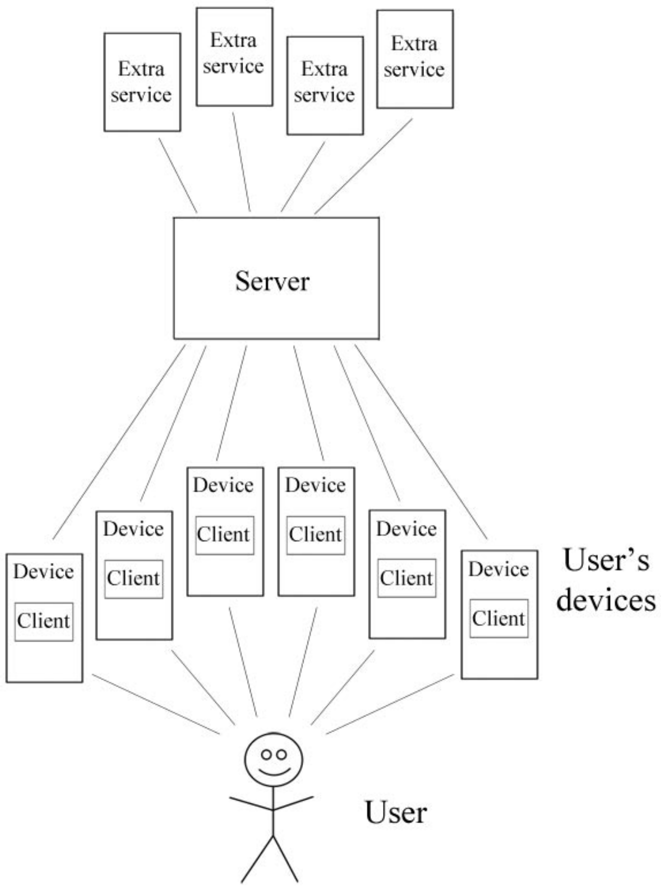


Figure 2.1: *DPC architecture*

The dependency on a specific computer can be solved by, for example:

- Using some kind of a name service that the client uses to find the server.
- Using some kind of a discovery mechanism, e.g. the client could broadcast a message asking for information about the server, whereupon the server responds with the url it can be reached at.

Both these alternatives give a more adaptable system in that the server url doesn't need to be preconfigured. The dependency on having a single computer that is always available can then be reduced by for example having one or more extra servers ready to take over if the active server fails (redundancy). Having a server up and running is still a vital criteria, but by adding redundancy this is less likely to be a problem.

Although the first alternative removes the dependency on a single server, one is still dependent on the name service being up and running, so the second alternative is possibly more robust as it relies on fewer running components.

A peer-to-peer architecture could solve the issue of being dependent on a server that must be available at all times, but this is potentially more complex to implement and maintain, and possibly harder for the user to understand. The client-server architecture was therefore a natural choice for this system.

2.2 Design

The DPC server consists of a frontend that handles communication with the client application and a backend that handles communication with the extra services. These are run independently in separate threads.

The extra services register themselves and the functionalities they provide with the server at runtime. This means that new functionality can be added at runtime by starting a new extra service that registers itself with the server.

Figure 2.2 gives an overview of the design of the DPC.

The server is split into a frontend and a backend to achieve modularity and flexibility. Since they have no shared memory and communicate with each other with JSON-RPC they are independent of each other even though they are running in the same process. In principle this means that they could be run as individual processes, and could even be run on separate computers.

Initially the frontend and backend were intended to use shared memory. The chosen solution gives the desired independence and modularity. It is also easier to implement and maintain as one avoids the need for the logic and lock mechanisms needed for shared memory.

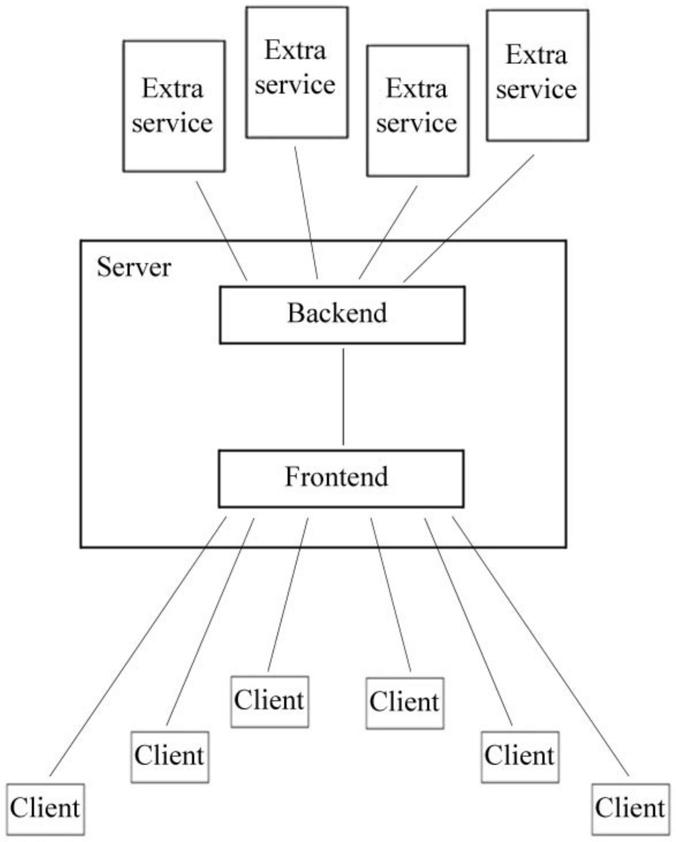


Figure 2.2: *DPC design*

Chapter 3

DPC Implementation

Within the timeframe of this thesis it has not been possible to implement a complete DPC. Therefore only a prototype of the basic parts of the proposed system, with some simple functionality, has been implemented.

The prototype consists of two main parts, *K-serv* and *K-app*, and two extra services, *Tst-serv* and *Tst-serv2*. K-serv is the server that provides the required framework for DPC. K-serv has a small set of built in functionality, the rest is provided by the extra services that make their functionality available through K-serv. K-app is the client application that runs on the user's devices.

Figure 3.1 gives an overview of the prototype, more information about the different parts is in sections 3.1 to 3.3.

Development and testing of this prototype has been done on Linux, and everything is written in Python.

Communication between the different parts of the system is done with JSON-RPC over HTTP. Bottle[1], a Python web framework, is used together with a simple JSON-RPC implementation for Bottle, bottle-jsonrpc [2], and a Python implementation of the JSON-RPC specification, jsonrpclib[3], to provide the means for this communication.

[1]http://bottlepy.org

[2]https://github.com/olemb/bottle_jsonrpc

[3]https://github.com/joshmarshall/jsonrpclib

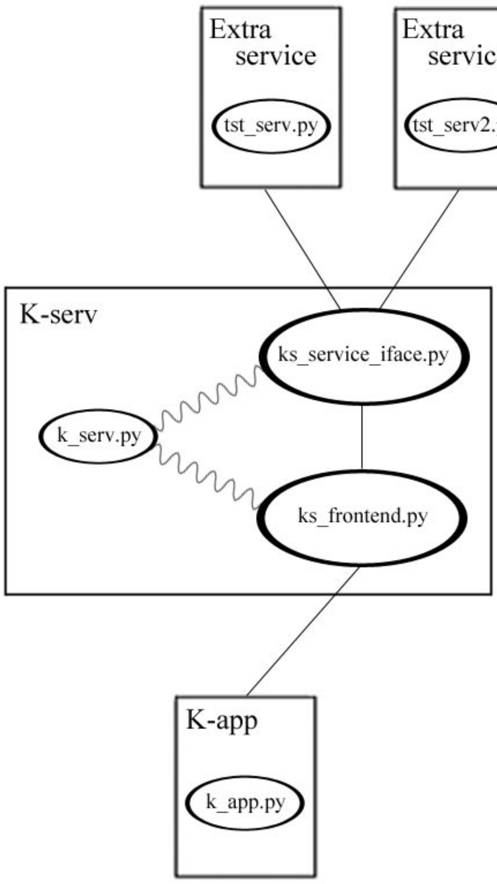


Figure 3.1: *Overview of the prototype*

JSON-RPC over HTTP was chosen because it is supported by most platforms and programming languages, thereby making it possible for the different parts to be written in different languages and support different platforms.

Information that should be configurable or is needed by more than one part of the system, like paths, port numbers and urls, is in a separate file, *common.py*. This avoids hardcoding information that may need to be changed, and avoids having the same information in several places in the code.

3.1 K-app

K-app is the interface to the user. It makes the functionality provided by K-serv available to the user, handles requests and input from the user, and displays the results. Since this is a prototype the focus has been on testing functionality, not using or creating real applications.

As mentioned above, most of the functionality in the DPC is provided by the extra services but some is built into K-serv. K-app handles the built in functionality differently to the extra functionality. Each built in functionality is handled individually, whereas the extra functionality is handled generically. This because the handling of the built in functionality has been kept the same way as it was for the author's individual special curriculum[1], changing it was not a priority for this project. As things are now, changes made to the built in functionality in K-serv could require changes to K-app's handling of them.

It should be possible to change things in K-serv without this affecting K-app, so it would be an advantage to change the code so that K-app handles all the functionality generically. This would make the divide between K-app and K-serv cleaner, K-app would be less bound to the present implementation of K-serv, and the code in K-app should be easier to maintain.

3.2 K-serv

K-serv consists of two parts that are run in separate threads: *Ks-frontend*, the frontend that K-app communicates with, and *Ks-service-iface*, the backend that provides an interface to and from the extra services. The two parts have their own Bottle servers with unique port numbers.

As mentioned in 2.2, the frontend and backend could be run as separate processes. The choice of having them run as two threads in K-serv was made as it is tidier and starting up the system is easier.

3.2.1 Ks-frontend

Ks-frontend receives requests from K-app, handles them and gives response. K-serv's built in functionality (save, merge and get) is in Ks-frontend, so requests for these are handled directly. Requests for other functionality are passed on to the backend to be handled there. Response from the backend is then passed on to K-app.

3.2.2 Ks-service-iface

Ks-service-iface has an interface for connecting the extra services and an interface for making the functionalities these services provide available for the frontend. It receives requests from both the extra services and from the frontend.

The requests from the extra services are for these to register themselves and their functionalities with K-serv, and thus make their functionalities available via K-serv, and also for them to deregister themselves when they are terminated.

The requests from the frontend are for getting information about the functionalities provided by the registered extra services and for calling these functionalities.

Ks-service-iface stores urls to the registered services in a file called *registered_services.info* so that it doesn't loose information about them when it is restarted. When it starts up it first checks if there are any urls to registered services stored in the file. If there are it sends a reregister request to all the urls in the file then deletes the file. When the services receive a reregister request they register themselves again.

With this way of doing things only the functionalities offered by services that are still available when the backend starts up again will be registered in the backend's information about available functionality, and the backend can handle restarts without loosing information about available functionality or keeping stale information.

Alternatively Ks-service-iface could store information about the registered functionalities between restarts, but this would add complexity in maintaining updated and correct information about available functionality.

registered_services.info is updated when the services register and deregister themselves.

3.3 Extra services

Extra services provide one or more functionalities to the system by registering at K-serv. In principle the extra services could be written in any programming language so long as they have an interface that uses jsonrpc over http, but this has not been tested. The Extra services can be run locally on the same computer as K-serv is running on, or externally on another computer.

Upon startup the extra services must register themselves at K-serv, and when terminated they should deregister from K-serv. In addition to the functionality they provide they must also receive and handle requests from K-serv to reregister.

Information about functionalities the extra services provide must be defined somewhere, and it was not desirable to have it hardcoded, so it was logical to have this in separate files. This information is stored in YAML[4] files (.yml), one file for each service.

YAML was chosen because it's a human friendly data serialization standard for all programming languages that supports nested data. Being able to use nested data makes it simple to organize the information about the functionalities.

PyYAML[5], a YAML parser and emitter for Python, is used for parsing the YAML files. PyYAML can be installed with or without bindings to LibYAML[6], a YAML parser and emitter written in C that is faster than the Python version. For simplicity, and because speed was not expected to be an issue, the choice was made to manage without bindings to LibYAML for this prototype.

Because the information about an extra service's functionality is stored in a separate file, registration of new functionality can easily be done when the service starts up. The service registers itself at K-serv by sending its yml file and its url as parameters in the registration request to Ks-service-iface. In this way all the functionality provided by the new service is registered with one request.

[4]http://www.yaml.org/
[5]http://pyyaml.org/wiki/PyYAML
[6]http://pyyaml.org/wiki/LibYAML

The yml files contain the following information for each functionality the service provides:

1. *funct_name*: Name of functionality.
2. *args_list*: List with the following information for each required argument:
   (a) *arg_name*.
   (b) *arg_type*.
   (c) *arg_description*.

   Empty list means no arguments are needed.
3. *return_type*: Return type. Empty return type means nothing is returned.
4. *app_list*: List of which applications can use the functionality. Empty list means all applications can use the functionality.
5. *description*: Description of the functionality

Functionality names are expected to be unique, but the system should be able to handle situations where functionality names from different extra services are identical.

The application list is included so that functionalities that are made specifically for given applications kan be provided. These might not work for other applications, and can therefore not be made available for all applications.

The built in functionality in the system is described in a Yaml file in the same way as the extra services' functionalities are described.

3.4 The system from multiple viewpoints

This section describes how the system works as observed from the different parts of the system.

3.4.1 K-app's viewpoint

Appendix A contains screenshots of K-app.

- K-app starts by displaying a menu where one can choose to quit or choose an app one wishes to do something related to.
- When the user has chosen an app, K-app sends a request to K-serv for a list of available functionality for the chosen app. K-serv's frontend receives the request and forwards it to the backend, it then receives the response from the backend and forwards it to K-app.
- K-app displays this list (including the built in functionality) as a new menu. The menu also includes "quit", "change app" and "refresh". "Refresh" gets K-app to request the list of available functionalities from K-serv again og display the new list. This can be used when one has stopped or started an external service and needs the functionality list to be updated.
- When the user has chosen a functionality K-app checks if any arguments should be included with the request for this functionality. If arguments are needed, K-app checks if they are standard arguments or if input is needed from the user. If input from the user is needed, K-app will ask for this (e.g.: *add* takes two numbers as input and the user will be asked to input these.)
- K-app then sends a request for the chosen functionality with the required arguments to K-serv. The built in functionality in K-serv is handled directly by the frontend and a response is returned. Requests for other functionality is forwarded to the backend, the backend then sends the request to the service that provides this functionality, receives the response and returns it to the frontend that forwards it to K-app.
- K-app checks the result code in the response. If something went wrong this will be reported and a "refresh" of the functionality list will be

done. Otherwise the result will be printed and the menu of available functionalities for the selected app will be displayed again.

3.4.2 An extra service's viewpoint

- When the extra service has started it registers with K-serv, i.e. it sends a register request to Ks-service-iface (K-serv's backend). This request includes the Yaml file that describes functionalities the service provides and the url to the service. Ks-service-iface parses the Yaml file and stores information about the functionalities provided by the service and what url to use for requesting them to an internal structure. As mentioned earlier, the services url is also saved to a file that is used by K-serv when it is restarted.
- After registration the extra service waits for requests and handles them. If a reregister request is received the service registers with K-serv again, in the same way as described above.
- If Ctrl-C is pressed (controlled stop) the service deregisters itself at K-serv and stops.

3.4.3 Ks-service-iface's viewpoint

- When Ks-service-iface is started it reads *registered_services.info*, sends reregister request to all of the urls in the file, then deletes it. It then has no services registered and only the services that react to the reregister request by registering themselves again will be registered with K-serv. This avoids situations where services that are no longer running remain in the system.
- After this the backend waits for requests and handles them.

3.5 Handling failure

The fact that this system is for a single user and the user's personal devices, reduces the failure scenarios a lot. The scale of the system is minute compared to systems that cater to millions of users and devices. Therefore scalability isn't much of an issue and the likelyhood of failure is small.

The most likely scenario is for one or more of the parts of the system to fail in some way, they could crash, loose data connection, become slow etc.

So what happens if one of the parts fail?

3.5.1 Failure of K-app

The rest of the system won't be affected by K-app failing, so the user will just need to restart K-app on the device. Bottle handles situations with clients that fail, so K-serv will continue working even if K-app fails between sending a request and receiving a response.

3.5.2 Failure of extra services

K-serv doesn't do any check for aliveness of the extra services, so if an extra service fails it will not be noticed until one of it's functionalities is requested by K-app.

K-serv will then try to connect to the service, and when this fails it will assume the service is down. K-serv will deregister the service so that it no longer has any information about the failed service or the functionalities it provides, and return a result code to K-app indicating that the requested functionality is unavailable. In case the failure to connect was due to the service being unavailable for just a short moment, K-serv also attempts to send a reregister request to the missing service.

Upon receiving a result code indicating an error from K-serv, K-app will inform the user that the functionality is unavailable and refresh its list of available functionalities.

When the failed extra service is restarted it will register itself as normal. K-app will not list the restarted service's functionalities until the user asks for a refresh.

Other possibilities for detecting failed extra services:

- K-serv could check if the extra services are up every time K-app asks for something from K-serv. This involves more work for K-serv, and would add latency, especially if there are many extra services registered.

It would also increase the amount of traffic from K-serv to the extra services.

- The extra services could send some kind of heartbeat to K-serv at regular intervals. This would increase the amount of traffic from the extra services to K-serv, and potentially slow down K-serv, but is probably the best of the two alternatives.

3.5.3 Failure of K-serv

K-serv is the central part of the system, so not much will work without it. As soon as the user requests something from K-app that requires connecting to K-serv, K-app will report the connection error and exit. K-app must then be restarted when K-serv is running again.

The extra services will not notice that K-serv is down, they will continue running. When K-serv starts back up it will send reregister requests to the extra services it knows about, as described earlier.

If K-serv starts up, sends reregister requests, deletes the *registered_services.info* file and then fails before the extra services have reregistered, K-serv will no longer have information about the extra services. However, this can easily be solved by restarting the extra services, and since this is a personal system that the user has full control over this is not a big problem.

Chapter 4

Experiments

Testing and experiments for this project have been done on the HPDS[1] group's display wall cluster, *Rocksvv*, at the Department of Computer Science at the University of Tromsø.

The Rocksvv nodes have the following configuration:

- HP Z400 Workstation
- Intel Xeon processor W3550
- 12 GB RAM
- Gigabit ethernet
- Rocks Linux distribution 5.4 with CentOS 5.5

The prototype uses Python 2.7.2, PyYAML 3.0 and version 0.10.11-1 of Bottle.

During development of the prototype, testing was done locally on the laptop used for development, and also on Rocksvv. Finally, to test that the system works as required K-serv was run on *tile-0-0*, K-app on *tile-0-3* and *tile-1-0*, Tst-serv on *tile-0-1* and Tst-serv2 on *tile-0-2*. The conclusion of these tests was that the prototype works as designed.

All experiments were done with K-serv running on *tile-0-0*, K-app running on *tile-0-3* and Tst-serv running on *tile-0-1* on Rocksvv.

[1]http://hpds.cs.uit.no/

4.1 CPU and Memory Use

CPU usage on the tiles was initially measured with *ps*, but it was discovered that *ps* gives cpu usage as an average over the time the process has been running. Since this was not what was wanted, the measurements were then done with *top* instead.

top reported K-serv's CPU use to be constantly at 99-100% when K-serv was doing nothing, and above 100%, up to 109%, when K-app's nop experiments (described in section 4.2) were running. This led to the suspicion that K-serv had a busy loop somewhere. It was discovered that after starting the frontend and backend in separate threads, K-serv went in to a busy loop while waiting for the program to be terminated. This was fixed by letting K-serv sleep between iterations.

The same was discovered for Tst-serv, it had a CPU use of 99-100% while doing nothing, and 110-140% when K-app's nop experiments were running. Again a busy loop was found and fixed.

Measurements of CPU usage were done after fixing K-serv, but before fixing Tst-serv, and after fixing both K-serv and Tst-serv. The fix to K-serv drastically reduced its CPU use, down to 25-35% when running the nop experiments. After fixing Tst-serv as well Tst-serv's CPU use dropped to 10-15% when running the nop experiments. However, K-serv's CPU use increased to 38-60%, but this is not surprising since it is no longer waiting for Tst-serv as much as before Tst-serv was fixed. K-app's CPU use (after these fixes) was at 25-40%.

top reported K-app and Tst-serv's memory use when K-app's nop experiments were running to be consistently at 15 MB.

For K-serv, *top* reports that its memory use starts at 16 MB, but increases with approximately 2 MB for each benchmark run. This may be due to a bug that leaks memory, but this bug has not yet been found. Another possibility is that the garbage collector hasn't removed all of the data yet. Running the tests for a longer period could identify this as just a garbage collector side effect.

Table 4.1 gives a summary of CPU and memory use on the different parts while running the nop experiments, after the busy loop bugs in K-serv and Tst-serv were fixed.

	CPU use (%)	Memory use (MB)
K-serv	38-60	16 and growing
K-app	25-40	15
Tst-serv	10-15	15

Table 4.1: *CPU and memory use while running the nop experiments, after fixing the busy loop bugs in K-serv and Tst-serv*

4.2 Roundtrip Latency

To measure roundtrip latency, two nop requests were added to the code, one as a built in functionality, *nopKserv*, and one as an external functionality provided by Tst-serv, *nopExtra*. These functionalities do nothing, they just return *None*.

When running the experiments, K-app sends *numRepeats nopKserv* requests and reports the time taken to complete all the requests, then does the same for *nopExtra*. The nop experiments were done with *numRepeats* set to 1000.

For *nopKserv* the numbers were stable, both when preparing the experiments and while running them. For *nopExtra* the numbers were mostly stable, but for a short period while preparing the experiments, measurements that were approximately doubled were observed. This was only for a short period, and didn't occur again, and was probably because something was going on on the node that the extra service was running on (*tile-0-1*). One explanation could be that others where running something on the node at the same time. Before this could be checked the performance numbers were back to normal.

Table 4.2 shows the time measurements from the initial experiment, the average time taken, and how many operations were run per second (*numRepeats*/average time taken).

	Time (seconds)	
	nopKserv	*nopExtra*
Run 1	2.29	11.35
Run 2	2.31	11.19
Run 3	2.40	11.29
Run 4	2.35	11.37
Run 5	2.31	11.33
Average	2.33	11.30
Ops/second	429.18	88.50

Table 4.2: *Roundtrip latency from the initial experiment.*

As mentioned in section 4.1, busy loops were discovered in K-serv and Tst-serv. The initial experiment for roundtrip latency was done before this was discovered, so new experiments were done after fixing this in only K-serv, and after fixing it in both K-serv and Tst-serv.

The second experiment was done after the busy loop had been removed from K-serv, but before Tst-serv had been fixed. Table 4.3 shows the time measurements from the second experiment, the average time taken, and how many operations were run per second (*numRepeats*/average time taken).

	Time (seconds)	
	nopKserv	*nopExtra*
Run 1	2.48	7.00
Run 2	2.45	7.08
Run 3	2.46	6.87
Run 4	2.46	6.71
Run 5	2.46	6.60
Average	2.46	6.85
Ops/second	406.50	145.99

Table 4.3: *Roundtrip latency from the second experiment. K-serv's busy loop removed, Tst-serv still with busy loop.*

The third experiment was done after the busy loops had been removed from both K-serv and Tst-serv. Table 4.4 shows the time measurements from the third experiment, the average time taken, and how many operations were run per second (*numRepeats*/average time taken). The roundtrip latency for *nopExtra* was significantly reduced when the busy loop bugs were removed.

	Time (seconds)	
	nopKserv	*nopExtra*
Run 1	2.49	2.69
Run 2	2.46	2.74
Run 3	2.47	2.74
Run 4	2.47	2.73
Run 5	2.51	2.83
Average	2.48	2.75
Ops/second	403.23	363.64

Table 4.4: *Roundtrip latency from the third experiment. Both K-serv and Tst-serv's busy loops removed.*

Chapter 5

Discussion

From the numbers in table 4.4 in chapter 4 it appears that the architecture and prototype for the DPC is good enough for personal use. The operation overhead is small enough to allow several hundred operations to run per second.

Depending on what you're trying to do in a real application, the operation execution time can increase significantly and dwarf the overhead of the DPC system. For example the cost of storage access can easily dwarf this overhead.

The performance numbers from the experiments have not indicated that splitting K-serv into a frontend and a backend introduce a significant overhead. Further experiments with larger data and operations should be examined before concluding that this split doesn't add too much overhead.

Because of the architecture, extra services can make use of platform specific libraries and applications to implement functionalities. A full system could implement things that are Windows, Apple and Linux specific concurrently, as the system allows users to have extra services running on multiple computers.

Another thing the architecture could permit, because requests from K-app go via K-serv, is the possibility to implement caching in K-serv. This would require that K-serv knows whether a particular functionality is cachable, this could be included in the yml files. This is one of the advantages of describing functionalities using configurable files.

5.1 Lessons Learned

5.1.1 Problem related to how Bottle does things

Initially there was a problem in Tst-serv with the things in *main* running twice when Bottle was running. When Tst-serv ran without starting Bottle, things ran just once, as they were supposed to.

It was discovered that this problem could be solved by removing the in-parameter “reloader = True” in the call to *bottle.run*. When this parameter is set to True, the Bottle server reloads and restarts whenever any of the source files used for the Bottle server change.

It appears that Bottle forks a separate process when reloader is set to True. The reloader probably runs in the parent process keeping track of when the source files change, while the server itself runs in the forked process. If Bottle is then run in a thread, which is what is done in Tst-serv, this causes problems.

5.1.2 Problems encountered with the reregistration things

Two problems were encountered with the reregistration things for extra services, a deadlock problem and a timing problem.

The **deadlock problem**: Reregister was first implemented such that K-serv sent a reregister request to Tst-serv upon startup/restart, Tst-serv would then send a register request to K-serv. As both processes are single-threaded, K-serv would be waiting for the reregister request to Tst-serv to complete and return, and Tst-serv would be waiting for the register request to K-serv to complete. Thus creating a deadlock.

This was solved by having Tst-serv handle reregister requests by setting a reregister flag instead of reregistering immediately. The reregister flag is checked and handled elsewhere.

The **timing problem**: K-serv sends reregister requests to all the services it has in *registered_services.info* when it restarts. The Bottle server takes over control when it is running, so the reregister requests are sent before K-serv's Bottle server is started. If Tst-serv tries to register immediately after receiving the reregister request it will get a connection refused exception. To handle this Tst-serv catches the exception, waits a little, then tries to register again, repeating this until the registration is successful.

5.1.3 The experimental environment

The Rocksvv cluster is a shared environment that many users can use concurrently and is therefore difficult to do controlled experiments on. It is difficult to control what is going on on the nodes, and activity from others can easily influence the results.

5.2 Future work

As of now the user interface is text based and obviously not suitable for mobile devices, a graphical user interface is therefore needed. More complex functionality should be implemented to test and evaluate the architecture and design of the system.

Some data types may not be natural to stream through K-serv as this could be a bottleneck. For example DPC could be extended to coordinate streaming of media files between devices.

Some kind of mechanism should be added to ensure that only authorized devices can access the system. As things are now the only requirement to send requests to K-serv and receive responses is that K-serv's url is known, there is no check as to whether the device is permitted to connect. A possible way of handling this is to use ssl og certificates. Device white lists can also be considered, but the first suggestion is probably more secure than simple white lists.

Error checking for the yml files should be added. As things are now there is no checking for errors in the yml files when parsing them.

Some kind of logging mechanism for K-serv and the extra services should be implemented. As things are now alle messages are displayed in the terminal windows. This can be difficult to keep track of when there are many processes running on different computers.

As mentioned in section 3.1 it would be an advantage to change K-app so that it handles both the built in functionality and the extra functionality in the same generic way. As things are now K-app creates the list of built in functionality itself and merges it with the list of functionality it receives from K-serv. The yml file describing the built in functionality, *default.yml*, should be parsed by K-serv, not K-app, and the list of built in functionality should be included in the list K-app gets from K-serv upon request.

Chapter 6

Related Literature

6.1 Eyo

The Eyo paper[4] describes a personal storage system that provides device transparency and continuous peer-to-peer synchronization across devices, and enables applications to automatically handle update conflicts.

Device transparency means that a user sees the same content on all devices (as they say in the paper: ”a user can think in terms of “file X”, rather than “file X on device Y””). A user can ”view and manage the entire collection of objects from any of their devices, even from disconnected devices and devices with too little storage to hold all the object content.”

To achieve device transparency and continuous synchronization, metadata is separated from the content. The metadata, which is small enough to store everywhere, is replicated across all devices as quickly as connectivity permits. The content is only stored on some of the devices according to placement rules described by the applications. Basically, the idea is to only store content where it is needed. Device to device connectivity is provided by an overlay network.

Eyo includes a new storage API that provides explicit version history to applications so that they can handle conflicts automatically by using application specific knowledge about the conflicted data. Eyo has separate stores for metadata and content, manages network transfers, and notifies applications when updates happen to files they are interested in.

6.2 Anzere

The Anzere paper[3] introduces ”a technique for policy-based replication in a network of personal devices, both physical and virtual.” The authors show that it is possible to support much more expressive policies than in previous systems without loosing scalability. This is facilitated by using equivalence classes to reduce the size of the problem, even when the number of data items is large.

Anzere is a storage system for personal clouds that was developed to validate this technique. The motivation for developing this technique and system was automatic personal data management.

In this paper they define personal clouds as a small collection of personal devices, such as phones, tablets, laptops etc., and storage rented from cloud providers.

In Anzere they use device- and content-neutral replication policies, which makes it flexible with regards to devices entering and leaving the personal cloud. An example from the paper of such a replication policy: ”a policy requiring objects of type=jpeg and location=phone to be replicated to at least one device of type=fixed and with tag=owned.” This policy guarantees that photos taken with the phone will be replicated to a fixed, owned, device.

Anzere uses a self-managing overlay network with an elected coordinator node. It fully replicates metadata and policies, and policies are evaluated by the elected coordinator.

6.3 PodBase

PodBase[2] is a system for transparently managing data storage across personal devices, in households with one or more users, for durability and availability. PodBase is designed to require minimal user interaction, the data management is done autonomously. It seeks to exploit available resources (i.e. unused storage space and incidental connectivity among devices) to replicate files and propagate system state, requires no central server, and is device, vendor and OS independent.

Metadata about the state of the system and the data stored in the system is stored on each device, and devices use pair-wise connections to reconcile their metadata and exchange data. Through these pair-wise interactions the metadata and data eventually propagates among the devices in the system.

Replication is done for data availability, data should be available on all devices where it is useful, and data durability, data should be durable despite loss or failure of devices. By using a linear programming approach to compute adaptive replication plans, the system is able to adapt to changing conditions.

6.4 Eyo, Anzere and PodBase vs DPC

All of the above systems provide personal storage systems, but this is also the only thing they provide. In the DPC project the idea is to provide the possibility of doing processing on the stored data, even across data from different applications.

The definition of what a personal storage system entails with regards to what devices and resources should be included differs between these papers and also with this project. Anzere includes storage rented from cloud providers, which is something this project seeks to avoid. PodBase includes one or more users in households, whereas DPC is for single users. Eyo specifies that it is for a single user's personal data and device collections, but they also mention the possibility of using cloud services in their system.

Some of the ideas in these papers could possibly be used in a project like the DPC. Although a sentralized service has been chosen as the hub of this system, it is possible that peer-to-peer connections could be used for localized synchronization when the service is unreachable for some reason or other. This could be an addition to consider in future work.

Chapter 7

Conclusion

This project introduced the Distributed Personal Computer (DPC), which aims to give a single system view of the user's personal devices without the use of external services. The DPC is meant to be for a single user with multiple devices. A prototype has been designed and implemented, and experiments have been conducted to evaluate the prototype.

The architecture of the DPC is based on a traditional client-server architecture, but moves most of the server's functionality into separate services that can run on many different hosts and platforms concurrently. Extra functionality can be added or removed at runtime.

The implemented prototype, and the experiments conducted on it, show that the concept of the DPC is worth pursuing further. The experiments show that the operation overhead is small enough to allow several hundred operations to run per second, and the architecture and prototype for the DPC appears to be good enough for personal use.

As discussed in the future work section (section 5.2), the system is currently not complete. Among other things a graphical user interface needs to be developed, as the current user interface is text only and will not work well on mobile devices. However, the implemented parts indicate that the chosen architecture and design provides modularity and flexibility, allowing the user to compose a set of desired functionalities.

References

[1] Karen Bjørndalen. *Personal Multi Device and Multi Purpose Data Store*. INF-3993 Individual Special Curriculum, University of Tromsø, Dept. of Computer Science, March 2013.

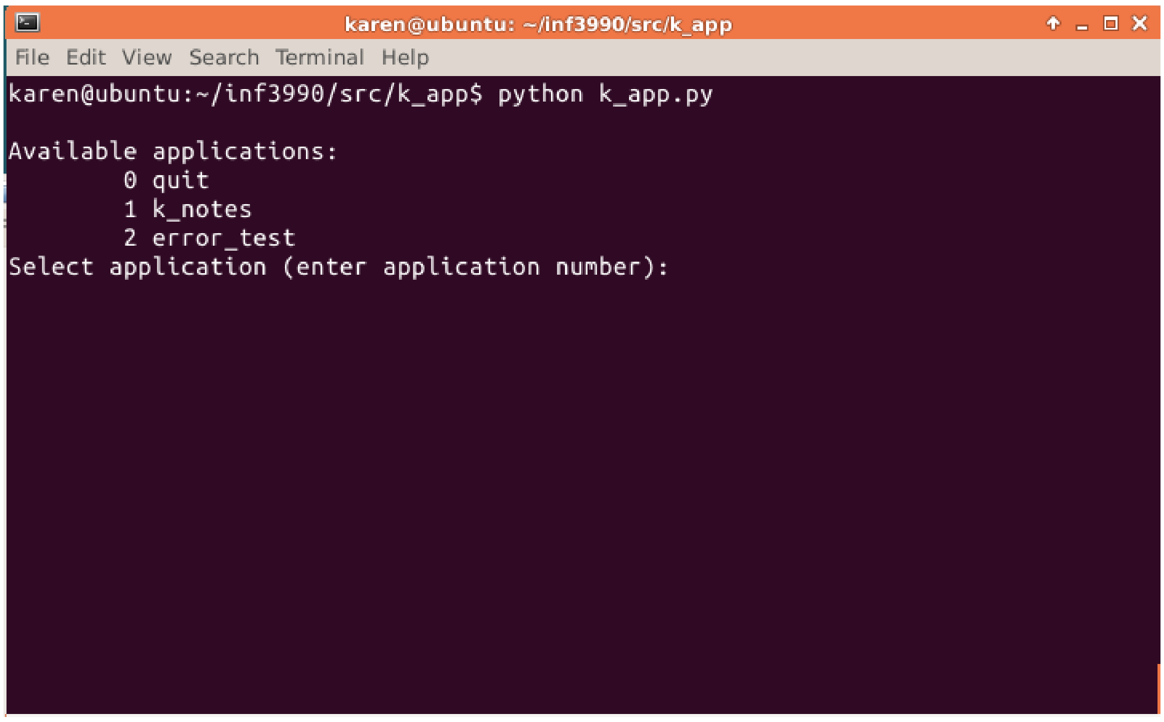
[2] A. Post, J. Navarro, P. Kuznetsov, and P. Druschel. *Autonomous Storage Management for Personal Devices with PodBase*. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIXATC '11, 2011.

[3] Oriana Riva, Qin Yin, Dejan Juric, Ercan Ucan, and Timothy Roscoe. *Policy Expressivity in the Anzere Personal Cloud*. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11. ACM, 2011.

[4] Jacob Strauss, Justin Mazzola Paluska, Chris Lesniewski-laas, Bryan Ford, Robert Morris, and Frans Kaashoek. *Eyo: Device-Transparent Personal Storage*. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIXATC '11, 2011.

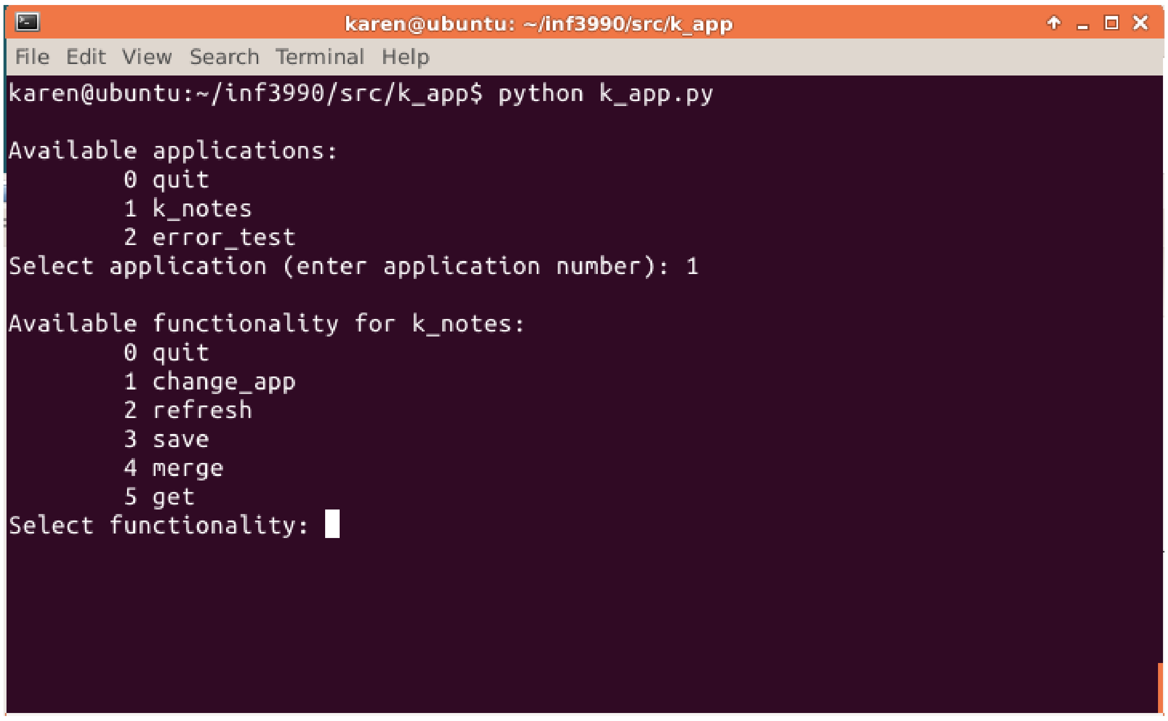
Appendix A

Screenshots of K-app


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

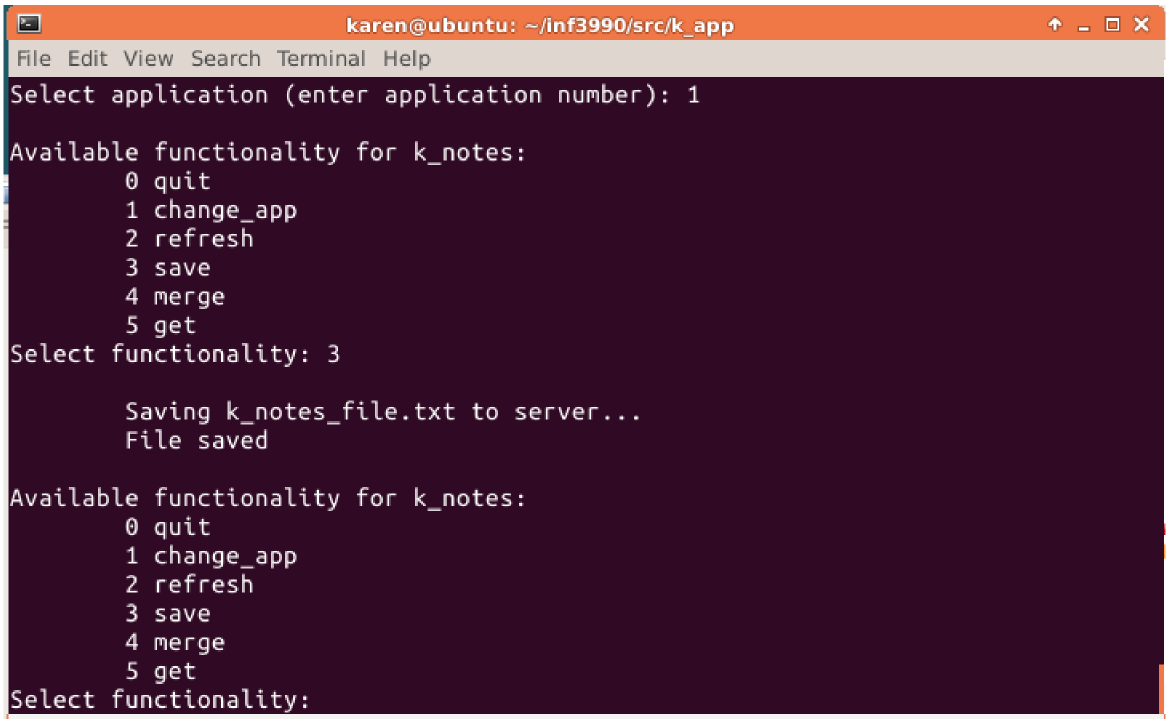
```
karen@ubuntu:~/inf3990/src/k_app$ python k_app.py

Available applications:
        0 quit
        1 k_notes
        2 error_test
Select application (enter application number):
```

Figure A.1: *K-app has started up and is ready for selection of application.*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

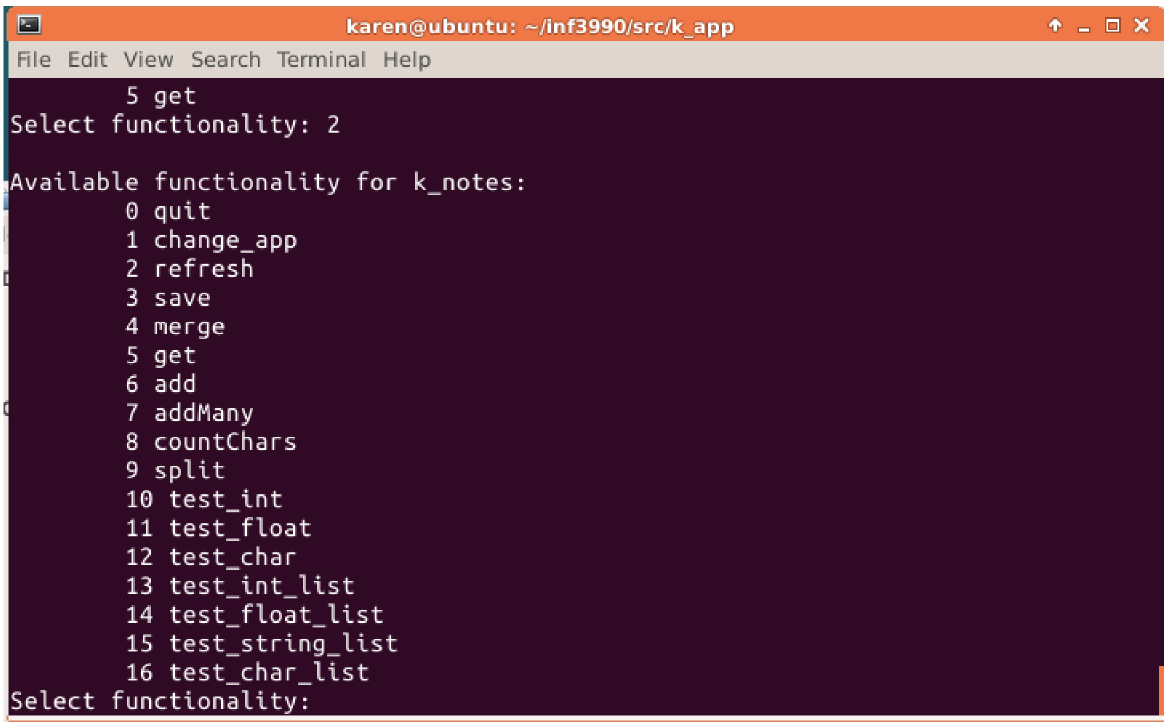
```
karen@ubuntu:~/inf3990/src/k_app$ python k_app.py

Available applications:
        0 quit
        1 k_notes
        2 error_test
Select application (enter application number): 1

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
Select functionality: 
```

Figure A.2: *Application (K-notes) has been selected. Functionality available for K-notes is displayed. K-app is ready for selection of functionality.*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

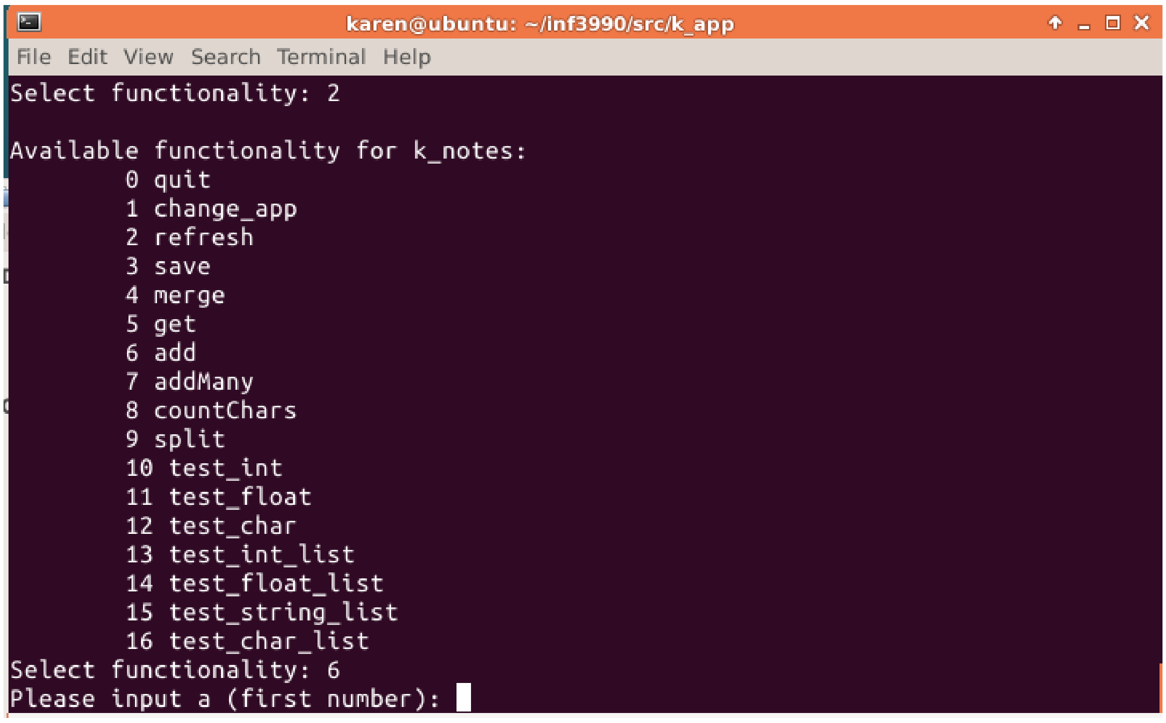
```
Select application (enter application number): 1

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
Select functionality: 3

        Saving k_notes_file.txt to server...
        File saved

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
Select functionality:
```

Figure A.3: *Save (functionality number 3) has been requested and done. K-app is ready for next selection.*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

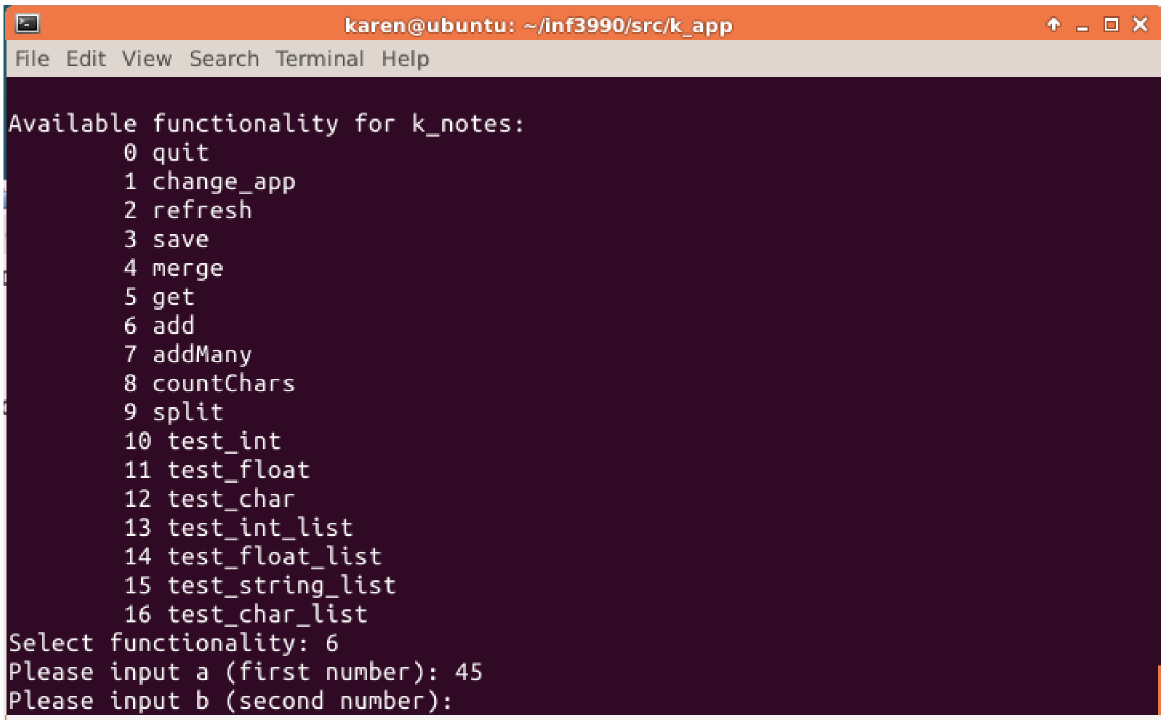
```
        5 get
Select functionality: 2

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
        6 add
        7 addMany
        8 countChars
        9 split
        10 test_int
        11 test_float
        12 test_char
        13 test_int_list
        14 test_float_list
        15 test_string_list
        16 test_char_list
Select functionality:
```

Figure A.4: *An extra service (Tst-serv) has been connected. Refresh (functionality number 2) has been requested and done. Functionality list now includes the functionality provided by Tst-serv.*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

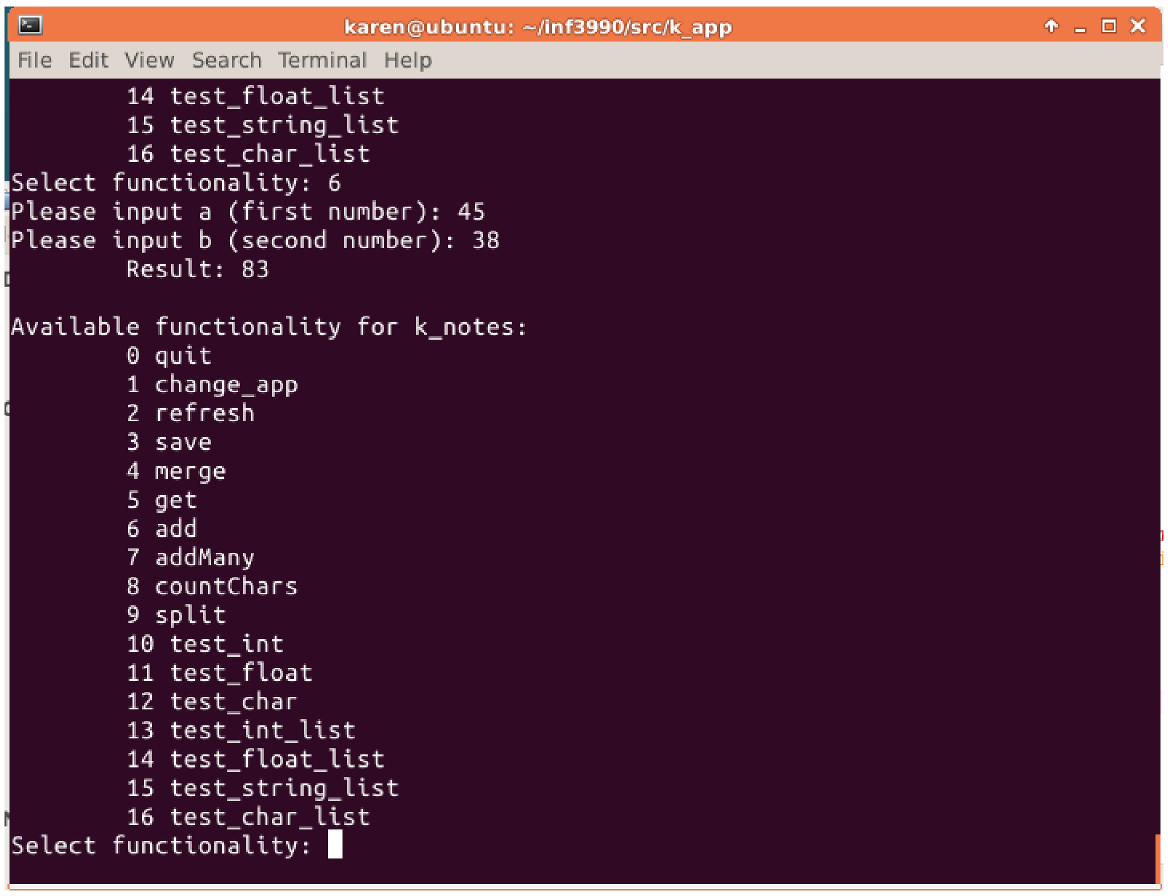
```
Select functionality: 2

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
        6 add
        7 addMany
        8 countChars
        9 split
        10 test_int
        11 test_float
        12 test_char
        13 test_int_list
        14 test_float_list
        15 test_string_list
        16 test_char_list
Select functionality: 6
Please input a (first number):
```

Figure A.5: *Add (selection number 6) has been requested. K-app is waiting for the user to input a (first number).*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

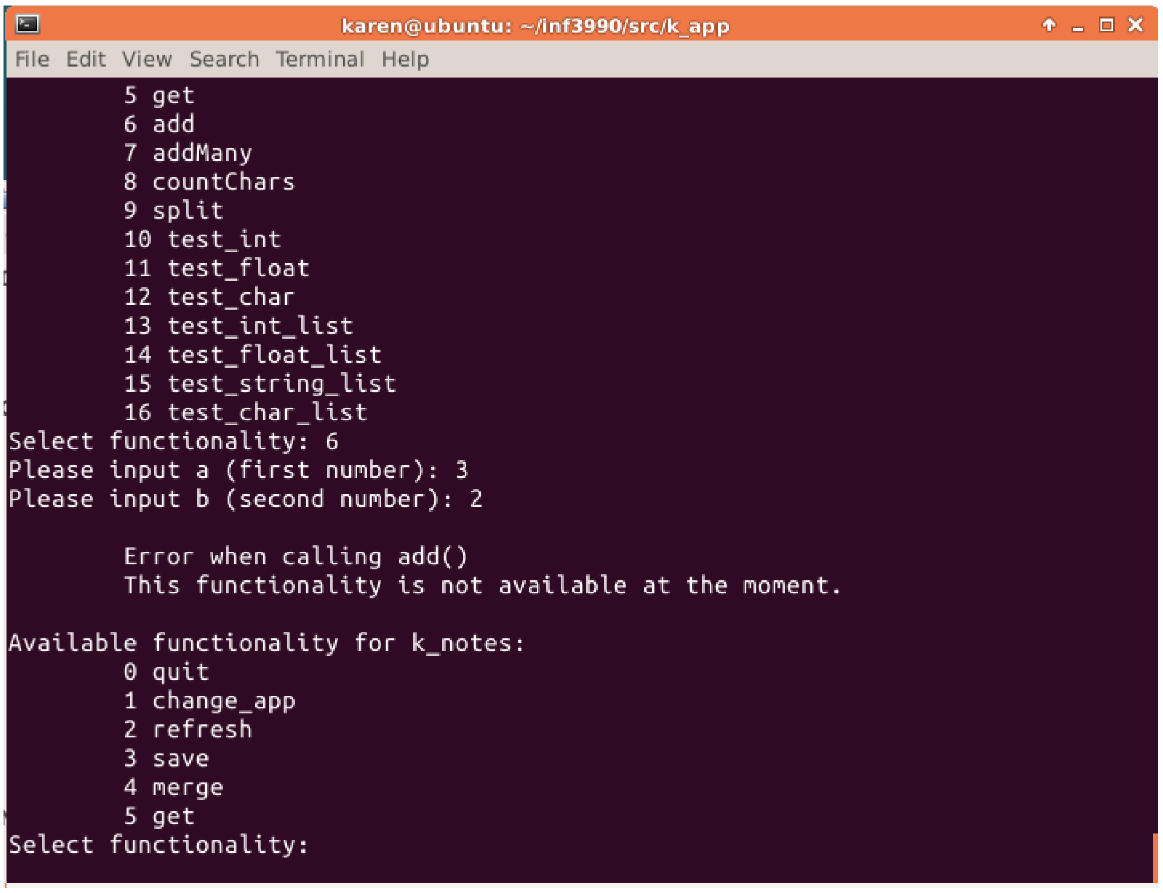
```
Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
        6 add
        7 addMany
        8 countChars
        9 split
        10 test_int
        11 test_float
        12 test_char
        13 test_int_list
        14 test_float_list
        15 test_string_list
        16 test_char_list
Select functionality: 6
Please input a (first number): 45
Please input b (second number):
```

Figure A.6: *Add continued. Input of first number has been done. K-app is waiting for the user to input b (second number).*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

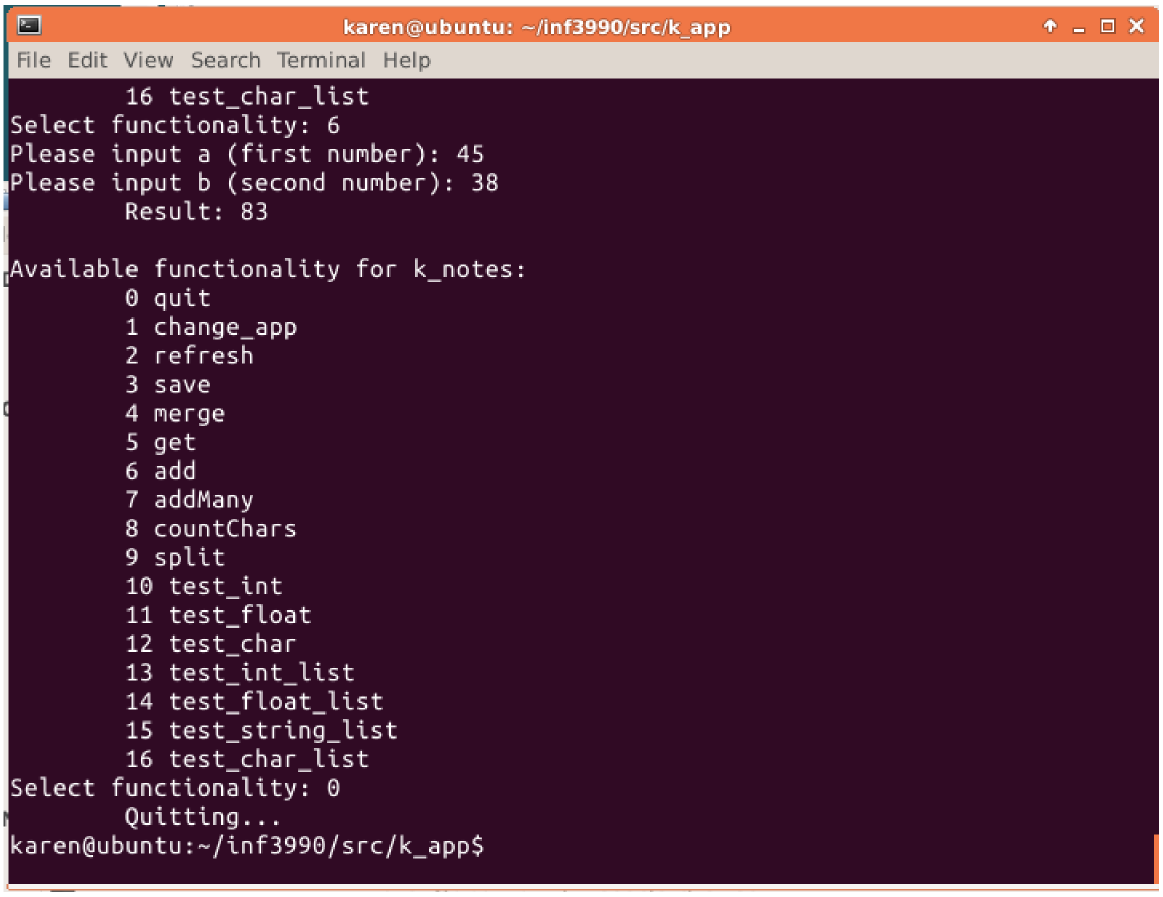
```
        14 test_float_list
        15 test_string_list
        16 test_char_list
Select functionality: 6
Please input a (first number): 45
Please input b (second number): 38
        Result: 83

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
        6 add
        7 addMany
        8 countChars
        9 split
        10 test_int
        11 test_float
        12 test_char
        13 test_int_list
        14 test_float_list
        15 test_string_list
        16 test_char_list
Select functionality: 
```

Figure A.7: *Add continued. Second number has been input. The request has been handled (sent to Tst-serv via K-serv) and the result is displayed. K-app is ready for next selection.*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

```
        5 get
        6 add
        7 addMany
        8 countChars
        9 split
        10 test_int
        11 test_float
        12 test_char
        13 test_int_list
        14 test_float_list
        15 test_string_list
        16 test_char_list
Select functionality: 6
Please input a (first number): 3
Please input b (second number): 2

        Error when calling add()
        This functionality is not available at the moment.

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
Select functionality:
```

Figure A.8: *Tst-serv is no longer available. Add (selection number 6) has been requested and numbers input, but K-serv reported that Tst-serv was unavailable. A message about this has been printed, and a refreshed functionality list is displayed. K-app is ready for next selection.*


karen@ubuntu: ~/inf3990/src/k_app
File Edit View Search Terminal Help

```
        16 test_char_list
Select functionality: 6
Please input a (first number): 45
Please input b (second number): 38
        Result: 83

Available functionality for k_notes:
        0 quit
        1 change_app
        2 refresh
        3 save
        4 merge
        5 get
        6 add
        7 addMany
        8 countChars
        9 split
        10 test_int
        11 test_float
        12 test_char
        13 test_int_list
        14 test_float_list
        15 test_string_list
        16 test_char_list
Select functionality: 0
        Quitting...
karen@ubuntu:~/inf3990/src/k_app$
```

Figure A.9: *Quit (selection number 0) has been requested and K-app has terminated.*