An Evaluation of Dynamic Software Updating Techniques for Embedded Systems in C

LUND INSTITUTE OF TECHNOLOGY
Lund University

Gustav Ek (gustav.ek@gmail.com)
Björn Boyd Isacsson (dat12bis@student.lu.se)

July 26, 2017

Keywords: Dynamic Software Updating, C, Embedded Systems, Memory, Sockets, Downtime, Serialisation

Abstract

Software updating is an incredibly common and important part of modern computers. However, Updating software is often a costly process as the software in most conventional programs needs to be restarted to implement the update. This causes downtime and sometimes even a loss of information. If the program in question is in the midst of a task such as a complex calculation, this can be costly to do. And yet, most software today still handles updates this way, causing the program to be shut down under the whole update period.

This downtime can further lead to yet more serious issues. The more costly the downtime is for the user, the less likely the user is to actually update the software, in order to avoid this costly and annoying downtime. This means that any bugs which are fixed in updated versions still remain in the live version, which is especially important from a security perspective. A bug which leaves an open security flaw in the system means that a system which is not up-to-date is more likely to be subjected to hacks, which can have very serious consequences.

An option to completely avoid these losses would be very beneficial, especially to critical systems such as security or medical related systems where uptime is especially vital and data needs to be kept private. To bypass the issue with conventional updates, we then need to avoid the restart process needed to apply the update. This can be done by using dynamic software updating (DSU), which is a field of research that deploys different kinds of techniques to make the update process seamless. However, DSU systems are often resource intensive, whereas many security critical systems are resource limited embedded systems such as a security camera. In this paper we compare three different updating methods that implement the principles of DSU, evaluating their usability for this sort of embedded system.

Populärvetenskaplig Sammanfattning

Software updating has always been an important topic ever since the origin of computer science. There exists no so such thing as a perfect complex program, so there are always ways to improve it. A majority of updates are related to bugs, which can cause unintended behaviour in the software and also open up security flaws in the program.

Updating software is often a costly process, as the software in most conventional programs needs to be restarted to implement the update, causing downtime until the program has started back up and returned to its previously running state, and can also sometimes cause a loss of information. If the program in question is in the midst of a task such as complex calculation, this can be costly to do. Most software today still handles updates this way, causing the program to be closed under the whole update period, depending on the operating system. An option to completely avoid these losses would be very beneficial, especially to critical systems such as security or medical related systems where uptime is especially vital. This is the case for Axis security cameras, where certain users of these cameras will want to avoid downtime of the camera system. To bypass the issue with conventional updates, avoiding restarts is thus crucial. This can be done by using dynamic software updating (DSU), which is a field of research that deploys different kinds of techniques to make the update process seamless.

The main goal of dynamic software updating can be said to be able to transfer the state of a program, i.e. data used at the current point of execution, from the older application to an updated version of it. Enabling the transfer of state means that the updated program doesn't need to start from scratch but instead starts with an already initialised state where the program was executing before the update. In this paper, we looked at one previously developed way of dynamic updating, one partial and known way to update which we developed the missing parts to, and developed a newly possible third way using systemd. These three options are a library called Kitsune, the system command exec, and via the common Linux init system systemd.

Our results show that they each have certain advantages over each other. Kitsune makes all variable passing simple, as variables, pointers and file descriptors are all passed the same way. However, it also uses significantly more memory and processing time and is difficult to use for cross-compiling a program. The exec functions also makes file descriptor storing simple and is integrated as a minor part of the program unlike Kitsune. However, it requires a more complex system to pass variables, and pointers cannot be directly passed at all, instead requiring the data they point toward to be transferred. Finally, the systemd technique has the most complex variable passing, but allows the original program to shut down completely during an update. Unfortunately, it also increases memory usage significantly just before shutting down the original program, so it does not end up saving any memory by doing this.

In a general use-case without hardware restrictions, Kitsune is likely the best option, as its simplicity of use makes for only minimal code changes. If however, there are restrictions such as in an Axis camera, exec is the best option if you require fast update times and/or low memory usage. The systemd method turned out to be worse compared to the other options in all these aspects.

Contents

Chapter 1

Introduction

Software updates have been a very important topic ever since the origin of modern computers. Writing a complex program without any flaws is very hard. Given enough time and large enough collection of code, bugs will occur. Most bugs are related to unintended behaviour of the program, sometimes it's not apparent at all that a bug exists and can be a hidden security flaw. There are also performance issues that may need to be resolved, usability improvements for lackluster functionality, or content additions to the software. Regardless, no matter what the reason, some type of software update will generally be needed. This software will typically be very similar to its previous version, with some parts of the code being exchanged to mend any of the issues mentioned above.

A question one then might pose, is how do we then exchange parts of the code with as little trouble as possible. The easiest solution is to simply consider the update and the old version as two different programs entirely, remove the old and add the new one. There are some issues with this. First of all, removing the program causes a loss of live data. Some data can be saved depending on what program you update, but especially with programs that rely heavily on hardware while running, data loss issues can arise. By relying heavily on hardware we mean for example programs that conduct large floating point operations. This is an example of a program that utilises a lot of the memory and cannot save the result to hard drive until the calculation is done. Secondly, to update a program more than often requires a restart of said program, which causes non productive sections of time. The third and last issue is a by-product of the other two, namely that restarts and loss of data can cause issues in security critical systems as they lose the ability to perform their tasks. For example a security camera guarding a gate or medical machines that monitor patient health.

The three above mentioned core issues are important to the company whom we developed our thesis with, Axis Communirations. They are the current market leaders in network cameras. Out of the above issues, especially number two and three are important for Axis due to the nature of their business. In Axis cameras today, updates have to be manually applied by the user on-site. This can be problematic in systems which have such high security requirements, and needs to have the ability to update fast. Since the cameras are dependent on the Linux kernel, once a security loop hole has been identified in Linux, all cameras are suddenly vulnerable to that attack. Because users have to update manually, updates takes longer time to apply. The longer the vulnerability is exposed, the greater is the risk that the camera will be attacked by a tool exploiting this loop-hole. But security is not the only issue linked to manual updates. It creates extra work for the customer who then might either choose to avoid updates or simply forget to apply them. The reason Axis can't use a more automatic update process used by many other applications today, such as Windows 10 Home which restarts your device when it chooses, is because of the security related issues of automatically taking down the device for maintenance at a time when the camera needs to be online.

The aim of this thesis is therefore to try to improve traditional update methods by investigating and creating update methods which can circumvent the issues with loss of data, restarts and the related security issues. We aim to do this by investigating the problem deeper and at what previous attempts have been made to tackle these issues. Along with this we will also try to develop our own methods, to better understand and hopefully come up with a solution to the problem.

We are not aware of any modern day solutions where non-traditional updates are in use. The most common solutions for updates today can be seen in major operating systems. In Windows, updates are left to each application to sort out. Most applications are split between either prompting the user to update it, or they simply skip this and perform the update automatically, but then with the above mentioned program restart similar to Windows 10's own update system. In many Linux distributions, the set-up is similar. For system updates, the user can be prompted to choose whether to install these partial updates. For the remaining software upgrades, the most common solution is via a package manager such as Advanced packaging tool, which can update all software it has been made aware, but only does so when prompted to update them by the user.

An important reoccurring theme among all the above is that all the above ways to update software requires this software restart, meaning that there is a period during the update in which the software is unusable. Companies running a large amount of servers have come up with a solution that avoids restarts from the user's point of view. They do this is by starting a second server that runs the updated version of the software in parallel to the old server running the old software that has already been running for a while. All new traffic can then be relayed to the updated server, while the old server handles only the requests that were already active before the updated server came online. Once all those old connections are completed, the server with the old version can shut itself down, so the only remaining server is then the updated one. Unfortunately this approach isn't directly applicable to all local software, as it requires a constant stream of new incoming requests from an external source. The security critical systems mentioned above will typically just have the one initial request come in, and this request is never completed but is expected to run continuously for very long periods, unlike incoming server connections. So, our approach to solving this will have to look different.

Before continuing any deeper into this thesis we would like to give a brief introduction of the chapters in this thesis and what the reader can expect from them.

- To start off, we'll give some background information on the company Axis, whom we've worked with to make our thesis, the field of updating, and the method which we used in working on this thesis.

- Then, we will analyse the problem domain and sort out the specific problems that we attempt to solve in this thesis.

- The section following shows our proposed solution to the problem and discusses its implementation.
- After that we will present experimental results from our testing benchmarks and discuss their implications.
- Thereafter we give a rundown of similar work for the interested reader, and what possible future work that could result in from this thesis.
- Finally, we will conclude the thesis and our results.

Chapter 2

Background

2.1 Introduction

We'll first give a brief introduction to Axis Communications, which gives a background of the company and motivations for this thesis. We'll then give some slightly more in-depth perspective of traditional software updates in order to give a more whole picture of the general field, before we go into some common terms of dynamic software updating; the common name for seamlessly updating software. This won't be a full introduction to dynamic software introduction but rather just enough to make it possible for the reader to fully appreciate the concepts discussed later in this thesis. Finally, we'll discuss and motivate our work method, the development of the thesis over time and the changes we've had to make as more information on the subject came to light.

2.2 Axis

The company Axis Communications was founded in 1984 and has since the beginning been directed towards various network solutions. Originally their focus was toward network printing where they became one of the top leaders in the market. They then shifted focus towards cameras with their invention of the very first network camera in 1996[13]. Although revolutionising for its time, it could only be used for surveillance with low frame rate requirements, as it was not capable of delivering more than three frames per second. The company continued establishing itself as one of the major players in the surveillance market, setting standards for connectivity in network cameras as well as developing the first video encoder. Today Axis Communications is the world leading company in the market of network cameras[13].

To keep its competitive edge in the market, Axis' focus lies in innovating existing technologies as well as developing in new up-and-coming areas of technology. As such, keeping available software up to date for customers is a major concern within the company. Since most of Axis work is directed towards its cameras and their surrounding products, a lot of effort is put into developing these updates and to keep the software up to and above industry standards.

Axis devotes most of its resources to develop software targeted towards UNIX based systems. They put considerable effort into improving both the kernel side and user space applications within these systems. The software development which is done towards the kernel side is also shared upstream, meaning that the code is uploaded in an open source project available for anyone.

Axis current update method requires the user to manually initiate the software update, and it results in a restart of the system. Since this easily leads to software becoming out of date for various reasons, a better solution is being searched for. This search for better updating methods, which can avoid restarts when updating, is what led to the development of this thesis.

2.3 Software updates

The ability to update or patch software can be said to be an essential part of programming and indeed seems to have evolved concurrently with computer software. Software updates were performed already back when programming was still done on punched cards[17]. The possibility to continuously change a program on the fly is a luxury which most other businesses don't enjoy. For example If its discovered that a house has been built with the wrong measurements it cannot simply be fixed with an update to the house, most likely it would need to be torn down and rebuilt.

Why then are software updates needed, can't the programmer take extra care just as other businesses have to do? Joel Spolsky, CEO of Stack Exchange argues that it's very difficult to spot bugs in your own code and that it almost needs a second pair of eyes to detect them[18]. Due to this obstacle software updates are certainly justified, however bugs are not the only reason why one would like to perform updates of software. The most common reasons for updating software are to either solve bugs or that the programmer wants to add new features not present in the old software[14]. Despite this, many users today have a strong aversion against updating their software due to previous negative experiences[15] such as software becoming unstable after an update or being unable to see the reason of installing the updates i.e. why they would benefit them. Typically, these updates can include interface changes or changing other functionality which the user in question didn't require[16]. The reason we outline these potential risks with updates is that an aversion towards updates can cause safety risks if the patch is directed towards fixing security issues. In an industry like Axis this could be potentially damaging for the customer.

Updates today are most commonly distributed via internet and then the program in question usually performs a check if any such update exists. The three major operating systems available today, Windows, Apple and Linux all have similar approaches where an update is distributed via an update application, either manually or automatically, and usually at a fixed time interval if the user opts for the automatic approach.

Hardware that is designed to run indefinitely requires very stable software in order to function properly since software failure could result in a halt in production or missing a person trespassing if the hardware in question is a camera. The issues facing software of this kind is then as previously mentioned software that can become unstable and the addition that a loss of state i.e. a restart of the system, can be very costly. Due to this, new update methods are required which can minimise these type of scenarios. One such method is to use a dynamic software updating technique, which we'll discuss next.

2.4 Dynamic Software Updating

The term Dynamic Software Updating was first coined in a paper by M. Hicks, J. T. Moore and S. Nettles [1]. Their paper developed a first initial framework for dynamically updating programs written in the `c` programming language, as a response to the problem of having certain programs that needed to be run continuously but also updated over time. This is the origin of dynamic software updating, namely the typically contradictory needs of constant uptime and regular updates.

The principles behind dynamic updating are easy to understand, but much more complex to implement. The core idea that dynamic software updating must achieve is to perform an arbitrary program update, without resetting the running state of the program. With state or *running state*, we mean data that is actively in use by the application, and in memory. The state can be said to be a snapshot of the currently used data at a specific moment in time.

We have briefly mentioned the necessity for low downtime when updating a program. This is especially true for dynamic updating and can be said to be one of the core issues it tries to solve. The downtime during an update should ideally be zero or at least minimal enough to not be noticeable in the functionality of the program. This boils down to an update being applied without the user of the program being able to notice it.

An important property behind dynamic software updates is that frameworks are language locked. This means that it is incredibly hard, perhaps even impossible, to create a framework for one language, say `c`, that would then also be applicable to another language, like Java. This is because the way in which state is saved varies between languages, so a generic solution would require an understanding of memory-saved state in order to restore it between versions. The key issue here is that internal structures can change between versions, like adding a variable, so a blind restoration of memory could corrupt the data, which is why a dynamic software updating program needs to be aware of a programs internal structures. A framework that understands the structure of `c` variables, would not trivially be able to understand similar structures in Java, because of the inherent differences in memory usage between the languages.

This language barrier has led development to instead pursue language-specific solutions, which are integrated into the code by giving access to dynamic updating libraries. The previous mentioned paper by Hicks, Moore and Nettles was developed for `c` back in 2002. Hicks also developed a dynamic updating system for Java together with L. Pina.[6] We then also have the Kitsune paper, which is also a library developed for `c`.[2] All these use different methods of saving and restoring the state of a program , which is one of the most important parts of dynamic updating, namely how the state is transferred.

So to summarise the above points, the following things are important for the success of dynamic updates.

- Which language is the software built in.
- Which state is being transferred between applications.
- How is the state transferred.
- How much time an update takes.

2.5 Method

2.5.1 Questions

Our main goal of this thesis is to improve traditional software updates. Due to the natural time limit of a thesis project, we need to set out with a clear boundary of what we aim to accomplish. We are starting this thesis with two main questions that we feel are directly relatable to our main goal and which addresses the main issues with traditional software updates. To reiterate what we mentioned in the introduction these are: loss of state, downtime or loss of time due to restarts and security issues related to the restart. As stated before the last issue is directly related to the two first issues. Therefore we have set out with the following questions.

- How do we avoid loss of state from runtime when updating?
- How do we avoid excessive downtime when updating?

2.5.2 Approach

To answer these questions we have chosen two main approaches. Firstly we will be implementing one or more proof of concepts. Secondly we will conduct a literature study.

2.5.3 Proof of concept

Our motivation for doing a proof of concept stems from the following two reasons. Firstly, when implementing this we will most likely come across issues which will help us understand why it is inherently hard to solve the questions we have posed. Otherwise these kinds of updates would be more widely used. Secondly we want to provide Axis Communications with possible frameworks, which they in turn could use for their systems.

The proof of concept will mean looking deeper into the structure of the operating system, in this case the Linux kernel. This is primarily done to take advantage of existing functionality. The kernel is very well documented and the documentation is accessible directly in the operating system through the manual pages. This documentation will be our main resource when writing our proof of concept.

2.5.4 Literature study

We have also to chosen to conduct a literature study. We have done this since we have found little or no studies in our initial preparation phase that are related to our thesis. We find this somewhat strange, as we feel that the questions we have posed must have been asked before. This leads us to believe that further research might help us answer our questions. It could also help us avoid previous pitfalls.

We are aware that we might have to change direction in our thesis and alter the questions we have posed should we come across similar research which have answered the questions we have posed. We do not see this as a problem and will continue our first line of approach along with a comparative study of the existing programs in such a case.

Chapter 3

Analysis

3.1 Introduction

Dynamic software updating (DSU) is a very broad and complex subject, with many attempts to tackle it but without any major breakthroughs capable of making it into a common practice for software development. To understand why, we need to take a better look into the problem domain of DSU. The advantages of having a working dynamic updating framework could potentially be very powerful for some applications, so there has to be something in DSU that makes it too difficult to implement into large scale programs. This analysis should allow us to identify the common problems in DSU that likely keep it from becoming the default choice for developers when designing the update process for their software. Hopefully, we'll be able to come up with some of the most important of these and can then work to overcome them in this thesis as best as possible, or at the very least discuss the ones we haven't researched further.

In the first of the three sections of this chapter, we'll be looking at what exactly the state of a program is and why it's important to dynamic software updating. There are a lot of separate pieces that make up the state of a running program, and identifying all of these is important in order to dynamically update, which we'll see why in this chapter. For the reader to understand the requirements needed for DSU to function, they need to have a proper understanding of each of these parts of the state. So, we'll explain both what these pieces are and also why they are important for the update process to function correctly. This chapter is then divided into two further sections following the state section. First we'll present the reader with a more in depth introduction to the subject of DSU with an analysis of what a working DSU solution would require to function properly. Next we continue on with a problem section which features a more in depth analysis of the major problems when trying to implement and utilise a DSU framework.

3.2 What is state?

When talking about state, we're referring to everything that has happened from the start of the program until the point in the program where an update is started. State is a very important concept in dynamic software updating, because it's the core of what dynamic updating is about. In order to perform said dynamic update, the goal is to continue execution after an update without breaking the chain of execution.

This means that externally, it shouldn't be noticeable that an update occurred, other than by noticing the actual changes the update brought. The program shouldn't need to start again from the beginning, redoing any initial set-up or perhaps re-establishing any connections a server had with some number of clients, because this would likely both take too long and the client would likely notice the connection being re-established. The code should therefore be able to continue executing from the exact point where the update was started. The exception being of course if the new version removes that point in the code, in which case the programmer would have to define the new proper return point. Returning back to this point is the goal of a dynamic updating technique. And the way it does this is by restoring its state. The majority of the state is the data such as variables which were active in the old version of the program, which would of course crash the updated program if it tried to use a variable that hadn't yet been initialised in the updated program. The state also includes the actual line in the code where the program was executing, so that it is able to return to that precise line again after the update.

There are three different types of data in the state that need to be transferred when updating dynamically: variables, pointers and file descriptors. Variables should be well known to any programmer, and these generally contain fairly basic data such as an integer or a character. Pointers on the other hand are somewhat more complex, as they refer to a location in memory that the programmer wants to keep track of, such as the start of an array of variables. This becomes more complex to transfer, and we'll see why later in this chapter. Finally, file descriptors are likely the most difficult to understand of the three. A file descriptor can among other things be a socket connection, so lets use that as an example. A file descriptor is considered in program code to simply be an integer, and can be used as such, but its not the actual value of the integer that is in any way interesting. It is instead when this integer is passed to certain functions that utilise file descriptors that we make use of its value, because the operating system uses this integer to identify the corresponding file descriptor (the socket) with this integer. These file descriptors are also connected to each specific program, so two different programs can use the same integer and yet use two different file descriptors when passed into a function that uses file descriptors. This is especially important, because it means that passing the integer alone to the updated program won't be enough to transfer the actual file descriptor over an update, so other solutions are required. In fact, even if we do transfer the file descriptor to another program, that program may already have a file descriptor with that index (the value of the integer), so the transferred file descriptor would then correspond with another integer value entirely, one chosen by the operating system.

3.3 Solution Requirements for DSU

Now that we have a clear idea of what is included among the state of a program, lets use this in our discussion of what is needed to implement DSU. There are a few things that go beyond state, although state transfer is still the most important requirement for DSU to function. One of these other important parts comes from our additional requirement to perform the updates on embedded systems. This leads to certain hardware limitations, so any DSU implementation has to be efficient in certain aspects that are appropriate for the hardware system in question. For Axis'

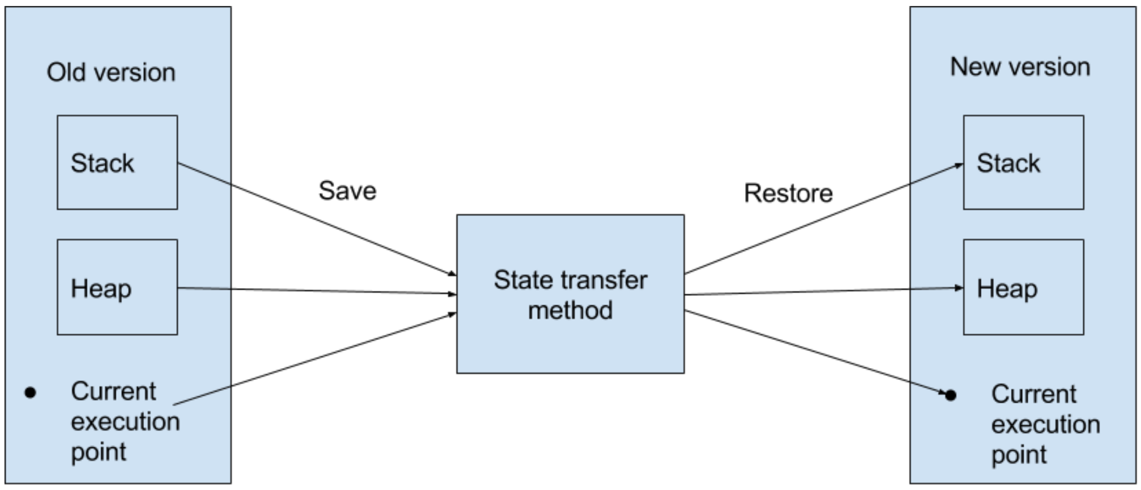


Figure 3.1: A DSU implementation where the state is transferred from an old version of the program to a new version via some unspecified state transfer method.

cameras, the experience of the people there whom we spoke to pointed out memory as one of the most limiting factors. Another requirement for DSU we would argue is that it must be user-friendly for the developer to use in their code. If it's too complex to integrate, nobody will bother, so it actually becomes a requirement to have a certain level of user-friendliness in order to see the DSU method be used. Finally there needs to be a good way to ensure the correctness of an implementation using the DSU method. Otherwise, the programmer runs a much greater risk of accidentally adding a bug that ends up crashing the software, which could have very severe consequences with certain systems. Now that we've quickly presented all of these key requirements, lets discuss them in some more detail in order to clarify the exact requirements and why they are actually so important for the implementation.

3.3.1 State Transfer

The key point of dynamic updating is that state must be retained over the update. This means that there must be a system in place that is capable of moving any state from one program to the other. The basic idea of how this works is shown in figure 3.1, which shows how an old version of a program transfers its relevant state to the newer version, using some unspecified state transfer technique. It shows most of the relevant state, but for image simplicity some of it is left out. For most of the variables, moving them means some sort of serialisation and deserialisation should be implemented to transfer the state that's available in the stack and heap, and for example saving them to and restoring them from the disk. Additionally, a system must be in place to save any file descriptors, without closing them along the way. Finally, the program must be able to return to the same executing spot from where the update was triggered. This could potentially be any single line throughout the code which can make this part very problematic to restore. It's however up to the programmer to choose how it's implemented, which can simplify the problem significantly.

3.3.2 Efficiency

Because of the additional requirement of the updates being performed on an embedded system, we have the previously mentioned requirements of it being an efficient program. Efficient in this case isn't just about running quickly and taking little CPU time, but also that it is memory efficient and that it adds very little to the file size of the object file. All of these three could be the main limiting factor that makes it unable to be applied to an embedded system. This means that any implementation should try to be as limiting on all these factors as possible. Especially important though is the memory, as the update simply doesn't work if it runs out, at the same time as memory is already quite restricted in the embedded system.

3.3.3 User-friendliness

An easily underrated aspect of a library is how user-friendly it is for the developer to implement into their program. A library which is complicated to use is significantly less likely to be introduced into a system because of the time it takes to understand how it functions. It also applies to the aspect of how easy it is to use after understanding how it works. An easy to use program would take very little time to apply to a pre-developed system, and it should be very clear what is happening in the code where it's added. Being user-friendly is especially important when the library is to be added into a large system. If it ends up being too difficult to implement, it's much more likely to be passed on as too expensive of an addition relative to its added value. So keeping it easy to both understand and use is an aspect that needs to be accounted for.

3.3.4 Correctness

Finally, its important that a developer is able to ensure the integrity of their program when adding dynamic updating. It should be easy to ensure that the program will function as intended over updates. This is especially true in any updates where there is a significant change in the way state is represented between program versions. There then needs to be a clear way for the programmer to include any necessary additions or changes to the internal state over these updates, so the updated program can still function if a lot of new variables are added in the update. So we will be accounting for how this is done, ensuring that state changes can be performed in an easily understandable way.

3.4 Problems with Implementing DSU

There are a lot of inherent problems that are encountered when trying to develop a DSU technique, and since we've been able to find many previous works relating to this type of implementation, we want to here learn something of what issues they encountered. So, we'll start at looking at what previous works have concluded as problematic with implementing DSU, as their research has valuable information for how we conduct our thesis. Then we'll add our own thoughts on what problems we consider important.

This chapter is spread into many smaller subsections, each relating to one specific problem in implementing a DSU technique. Below is a list of the core issues we've been able to identify in our thesis work. We've split each of these dots into a subsection below, where we further discuss their meaning and implication. Many of these are also related to the program developer's experience when they utilise a DSU method, and we'll touch more on which of these are related to the developer experience in the corresponding sections.

3.4.2 How to guarantee update correctness?

3.4.3 Which data do we serialise and what do we save directly by preserving the memory space?

   - Variables
   - Pointers
   - File Descriptors

3.4.4 How do we deal with program changes between versions that affect our live data?

3.4.5 Do we detect updates manually or automatically?

3.4.6 What resource limitations are we likely to see from adding dynamic updating?

3.4.1 Problems derived in previous research

The most recent contributions to the research field of DSU are the library Ekiden[3] and its continuation Kitsune[2]. Other previous notable contributions to the research field include Ginseng[4], POLUS[10] and UpStare[5]. In a paper from 2012 by E. K. Smith, M. Hicks and J. S. Foster [9] the authors present us with benchmarks between these programs, split into three parts: steady-state overhead, which is the overhead DSU introduces while running the program, compilation overhead, which is the overhead for compiling the program, and finally how often the update can be applied during execution. In their paper they report that Ekiden and Kitsune introduce the least amount of steady-state overhead. Compilation overhead on the other hand was harder to measure as most papers don't report it. Perhaps the most important result that the authors bring to light is that it's very hard to reason about the correctness of an update after it has been applied when updating by DSU methods. The authors say that some methods are possibly easier than others to reason about but that this is also purely anecdotal and based on subjective experiences. This is a very important topic for this type of updating, and so we later in this paper discuss what assistance there is for ensuring code correctness and how a programmer could go about making safer code. This paper is also why we choose to look at the steady-state overhead of the program, in order to ensure their viability for embedded systems.

3.4.2 Update correctness

A very important part of ensuring a proper dynamic update system implementation is the ability to serialise and deserialise data in a safe and reliable way. Any data corruption in this stage would cause the program to crash or execute incorrectly, which could potentially be even worse. Imagine for example a bank system suddenly giving you access to another person's account. Ekiden[3] and Kitsune[2] solve this by running a background library which keeps both versions of the program in memory as shared libraries. The serialised state can then effectively be transferred between the two versions as they are seen as a single program by the operating system. Other DSU methods such as Ginseng[4] use something called lazy state transfer which implies that the data used is always updated to the latest version when it's being accessed. Regardless of the way chosen to perform the transfer of state, serialisation and deserialisation of the data is one of the major problems the programmer is faced with when implementing a DSU technique, as a poor method here which can return corrupt data to the updated version can have serious consequences.

3.4.3 Preserving data

Before looking at the serialisation of the data in the program, lets take a step back and look at the bigger picture of this, namely the memory, to better understand why saving the data directly from memory is so complex. Looking at the traditional way of updating a program, everything in memory is lost when an update occurs. This is because as soon as a program shuts down, its memory space is recovered by the operating system and the data in it can be considered lost. However, when we want to update dynamically, without losing state, we would like to keep this data in memory, since reading from memory is much faster than having to save and read the data from the disk whenever we update.

There are two separate parts of interest in memory, the heap and the stack, and these have to be dealt with separately for a dynamic update. The stack is always linear in that there are no gaps in the memory between variables. If the stack pointer increased by 4 bytes, then we know that everything in those bytes is relevant to our program. With the heap however, data could be placed anywhere within it. The data isn't necessarily ordered in any way, so a pointer to a location in the heap could be pointing anywhere within that heap. This is problematic in that we can't then simply copy everything within the heap like we perhaps could do with the stack, as we wouldn't know what data was relevant, and we'd have to duplicate a lot of extra unnecessary data from all the unused locations if we did actually copy it all. Any pointers to the heap are also made incorrect upon release of the memory when the program shuts down. Using those after that point causes undefined behaviour in `c`, so when trying to save pointers, it's important that whatever data is saved is the actual relevant data that the pointers are used to access and not the actual pointers themselves. The one exception being if your DSU technique in fact shares a memory location between the two versions, as then all pointers will still be correct and no data would have to be copied from the heap.

Variables

We predict that variables will cause less hindrance when performing the serialisation due to the more complex nature of pointers and file descriptors. Variables often only contain primary data such as an integer or a character, which are easier to handle as one only needs to serialise the value of these variables and save them in an appropriate location. There is no native serialisation in `c`, so that part has to be added, but the entirety of the task still ends up being simpler than for pointers and file descriptors.

Pointers

When dealing with pointers we are instead dealing with an address to a type of variables. This can complicate things much more than when only dealing with said variables. If the memory region doesn't change, as mentioned at the end of section 3.4.3, then we can actually just save the value of the pointer without consideration of the data, since we know the location the pointers refer to will still be readable after the update. However, not all DSU techniques will be able to use this special case, so we also have to look at the case when the memory region changes between the old and the updated program. In `c`, pointers are often used to refer to arrays or even other pointers. This is problematic for data serialisation, as it's not possible to identify the size of an array simply from a pointer. This means that in order to save the data which a pointer is pointing to, you would also need to know the length of the data beforehand.

A similar problem arises when dealing with void pointers. Without knowing how many bytes of data is to be saved from where the pointers are referring to, the data cannot be safely saved. Saving too little would of course mean a loss of data, and saving too much could result in crashes from reading outside of one's allocated memory. Clearly, this is a complex case which is difficult to solve. If you don't internally know the correct number of bytes to save, it would require externally knowing the exact boundaries of the memory region. And even if we were to copy all the data in the heap to the updated program, the pointers would then be pointing to incorrect locations as the OS chooses what memory addresses the program is assigned. This could therefore be implemented into the operating system itself, but cannot be implemented into the actual program by a simple DSU technique. The DSU libraries are therefore restricted in that they can only save pointers either through the first case of using the same memory space and passing pointers as variables, or by saving the relevant data via receiving the exact number of bytes to serialise from a particular point in the heap.

File Descriptors

Finally, file descriptors. In `c`, a file descriptor is defined as a non-negative valued integer. The actual value of this integer does not pertain any information other than the value itself. Instead, the operating system has a list of indexes which, when this integer is passed into a function which uses these file descriptors, the function will use this index to identify the correct file descriptor to access. When creating a file or opening a network socket the process in question gets a file descriptor associated with it and the kernel keeps a reference to all processes associated with that file descriptor. When the reference count reaches zero it means that no processes are still associated with it any more and it can be safely closed. The reason this can become a problem when updating is that some file descriptors need to be transferred and kept open between processes. For example if the application is sending images over a network, it will have a file descriptor associated with it. This file descriptor needs to be transferred to the updated application if the connection is to be retained, else the whole reason behind updating dynamically would have failed and we'd have to redo the initialisation process of the socket. Recall here that a dynamic update has the goal of not needing to go over an initial setup again in order to return to the last executed point in the old version of the program. When dealing with serialisation, the problem of transferring these descriptors without closing them is generally the hardest, as they don't have any directly related data to them. Instead, certain functions are used to pass file descriptors between processes.

However, there is one workaround here to avoid having to use the functions capable of passing file descriptors in a system, as these are typically slow. Since file descriptors are connected to a specific process via its process ID, as long as the process ID of the running program doesn't change, the file descriptors will still be connected to the process and their corresponding indexes will be unchanged. The process ID has to also always be active, if it closes down and you then happen to get the same ID when starting the program back up again, it wouldn't work since during that period the socket would have closed. So, if the technique is able to utilise this specific case, it is again possible to simplify the issue to simply passing a variable (an integer in this case), since it will still refer to the same file descriptor in the index table.

3.4.4 Program changes between versions

Now that we've presented the problems of passing already existent data, lets take it one step further. A common part of an update is a change in the internal structure of data, such as a variable being added or a `struct` being changed. These are variables which haven't existed in previous versions and won't have any data to restore upon updating, and yet their values may be required to run the new version of the program. This means that the programmer needs to set up a way in which this data gets an initial correct value based on where it's being executed. There are a number of ways which this could be done, depending on how complex the calculation of the value is, but that's not the most important part of this problem. The main issue that is caused by this is instead that it becomes very difficult to update two or more versions at once. Since each update would only track the difference from the previous version of the program to the current one, updates would have to be applied one at a time. Having an update track differences from multiple previous programs would quickly make it bloated, which would likely reduce certain performance or size requirements. Following from this, even if running multiple updates sequentially, this means that a lot of potential dummy data could be getting added to the program if it was considered too difficult to properly initialise some of the variable additions and it was assumed they'd get their correct value at run time. This then comes back to the previous problem of how to ensure that the update functions correctly.

3.4.5 Update detection and initialisation

Now lets look at how to identify a new update. This can be done two ways, either manually or automatically. If it's to be done automatically there needs to exist a detection mechanism which is able to identify when there is an available update, either locally or available online. Once identified, the program should be able to initiate the update. In the other case, where an update is started manually, the user would have to tell the program when an update is available and perhaps even where it is located. This is likely more efficient in terms of CPU time, since it means the program doesn't need to keep track of when an update is available, but it comes at the cost of user convenience. In our opinion, the optimal mix would likely be an automatic detection of updates with a user prompt before initialising it, or to skip the prompt entirely as the update won't disrupt the user's usage of the program. This ensures that updates are directly going out to the user, and hopefully if they're given the choice they then choose to apply the updates right away.

3.4.6 Resource usage

Lastly there is the issue of scarce resources when performing dynamic updates. For most users this won't be a problem, but when dealing with some type of systems these types of updates might not be possible due to their hardware restrictions. Dynamic updating methods which load both versions of a program at once, that is both the old and the updated version of the program are both running simultaneously, requires double the ordinary memory usage to actually function. For an embedded system this could be an especially limiting factor when choosing the update method, as they are typically designed to have as little extra memory as possible to cut down costs. Of course, if the updated program uses a mere fraction of the total memory, perhaps because the operating system uses much more, then this could be a non-issue for that particular system. There are also other limiting factors when dealing with an embedded system, namely disk space and CPU time. Both of these resources are typically quite limited, but are less likely to be an issue as dynamic updating should generally add little overhead to both the file size and execution time. There is however the famous dilemma of execution time versus memory usage to take into account, where higher execution time requirements could lead to a lower memory requirement and vice-versa. Perhaps certain solutions could utilise this to lower the memory requirement on the hardware.

Chapter 4

Proof of Concepts

4.1 Introduction

This chapter presents implementations where the main goal is to show that the difficulties we've explained with dynamic software updating in the previous chapters can be solved and as such this chapter can be seen as a proof of concept for possible solutions. First and foremost this chapter is important to understand the later results & discussion chapter in-depth, as you do need the background information presented here to fully appreciate certain discussion parts. Should the reader wish to implement any of the methods included in this chapter, the information presented here is essential to understand in order to make a correct implementation and usage of it.

In order to solve the problems identified in the requirements from the previous chapter, we need to find a suitable solution which deals with all the key problems mentioned previously. This same solution should then also try to minimise the other smaller disadvantages that come with DSU, which we mentioned in the analysis chapter. Unfortunately, no single solution is able to fully achieve all the requirements that we have set out to solve. Improving upon one of the criteria often comes at a price to the others. We have therefore decided to compare multiple solutions to each other which can all achieve the core requirements of dynamic updating, while providing different trade-offs.

We will first introduce the reader to the program that we chose to implement our DSU methods on. After that we will continue on with the three different methods that we have explored where two of them we have developed. First we'll show a typical implementation of a dynamic software updating library called Kitsune and discuss its strengths and weaknesses, and then move to the two other options; systemd and exec which we ourselves have developed and do the same for those two methods.

4.2 The test program

In order to achieve a proof-of-concept, we first needed a system to try it out on. Since we chose to focus on making a proof-of-concept rather than immediately try to scale it into a live program, we wanted a program which includes as many of the real complexities as possible while still being small enough to quickly and easily work with. As a main part of Axis cameras is the ability to stream video from the

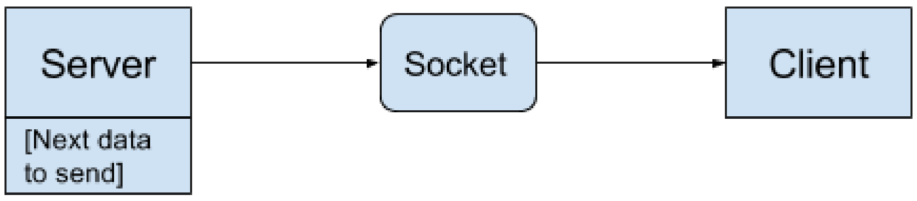


Figure 4.1: The important parts of the structure of the client/server program from a dynamic updating perspective. The server and client are connected via a socket, which the server sends some data over and which the client then reads. This data is dependent on the state of the server.

camera (server) to the user (client), we decided that we needed a system that ran a live connection. We therefore decided to go for a generic and simple client/server program to implement the updating system on. This not only has a live connection, but also an added state where the server sends a letter of the alphabet each tick (default one second). For a successful update, the next letter should follow the alphabet after an update, instead of resetting to the letter 'A'. This system is much easier to ensure update correctness with than it would be to send sequences of images from a video, as those images are nearly identical and missing one or even a few would be difficult to notice. Therefore, we chose the simple alphabet solution rather than the technically more correct video streaming solution, which is actually used in certain embedded systems such as security cameras.

All the implementations which we will be introducing the reader to further down have been done on the client/server program. In essence the server initialises a socket with a pertaining port and then listens on this port, ready to send data when a client connects to the server. If we look at figure 4.1 we get a clearer picture of what the server does. It shows how the server and client are connected via a socket, that data flows from the server to the client, and that the data that is being sent over it depends on some state (a simple `char` in our program) within the server.

The base test program is quite generic, and anyone within the field of computer science should have come across a variation to it before. It was convenient to use since it becomes easy to prove if a DSU implementation was successful or not. First the socket will be accessed via a file descriptor. When an update is performed we can confirm its success by validating that the connection was kept open during the update, meaning that the client will not have to reopen a connection to the camera (represented by our server). Secondly we continuously pass on a state from the server which is updated after every send/read request. When an update is performed this state should remain the same and the data transfer should continue from the point of the last transferred data. The important part here is that the state could be anything, a set of images, a string of characters or a set of files which are serialised from the old version of the program and deserialised in the updated version. It's then very easy to verify if the state has been stored correctly as we are given an almost instantaneous verification of it when the client reads from the updated application.

4.3 Kitsune

Kitsune[2] is an updating library specifically built for C. C is a general purpose language closely associated with the UNIX operating system and is generally considered to be a 'low level' programming language[21]. This means that a C programmer deals with relatively primitive data compared to an object oriented language such as Java which features more complex data structures and class hierarchies. The typical challenges facing a C programmer as opposed to a more high level language is how to deal with garbage handling, heap memory transferring and pointers i.e. addresses. The implementation of Kitsune does its best to overcome these issues for the programmer when it comes to the software updating process, such that they don't need to free used memory, work closely with the heap or track pointer locations in the update process. We'll discuss this some more in the usage section below.

There are multiple parts to Kitsune, the core being the library that is included and used in your code. Other parts include a compiler command (which uses gcc with some special flags which we discuss in the structure chapter), a driver program, and a program to initialise manual updates. There is a lot of parts to explain how to use it, so we'll be splitting this sub-chapter into two sections. First we'll take a look at how it deals with the problems and requirements mentioned in the previous chapter. Then we'll discuss how to actually use and integrate it into a program.

4.3.1 Kitsune's Structure

The general setup of Kitsune has a lot of advantages due to the usability of the Kitsune library. The fact that a single line is able to restore all global variables is a big convenience feature. This is an advantage that comes from Kitsune's technique of having a controlling program (`driver`) that owns the memory space for both programs. This allows it to be directly aware of all global variables (as they are directly visible to the `driver`). There is then also a small addition done to each function to allow it to also transfer any relevant local variables. Again, this is a very convenient way to do it, as all variables are in the same memory space already due to the `driver` program owning it, and all the programmer has to do to restore them is essentially mention them, and `driver` sorts out the rest.

Because of how Kitsune is able to automatically migrate certain variables, there also needs to exist a step which can track the state in between this migration. Let's look at a quick example to demonstrate what we mean. Say that you have a global variable `number` in the first version of the program. In the second version of the program, this variable has been renamed to `count`. Now, if the migrate function is run, it will notice that count didn't exist in the first version, and thus cannot restore it. So here we need to add a step that links `number` to `count`. This step is done by the use of a Kitsune written tool called `xfgen`. It has two functions, the first being to deal with above transforms and the second being to initialise any newly introduced variables which didn't exist in the old version of the program.

These transformation files are then used to generate a c file using Kitsune's `xfgen` tool. Following that, it's compiled and combined into the shared library. The shared library, as we recall, is the way in which we compile our original program. So we've just combined the transformation code into our program. When the program later updates and tries to migrate the code, it will then be aware of any state changes and additions that it needs to initialise during the migration.

There are two ways to initiate an update in dynamic updating. Either the programmer starts it from within the code, i.e. "automatically", or the user runs it manually, as in some command that passes the update to the program. The DSU library Kitsune has solved this by implementing both ways of starting the update and essentially letting the programmer choose for himself which to use. For the automatic update, the program must know beforehand where the updated version is located, and it should be specified by the programmer when it should start this update. In a typical scenario, this would likely be once the updated file exists, alternatively is modified. A nuisance of this approach is that each version of the program will have to manually specify in the code where the next updated file is, and it cannot be the same file as was used to update to the current version of the program, else it would always see that the file existed and get stuck in an update loop updating to itself over and over.

For the manual version of updating, a shell file called `DoUpd` is used. Its function is very simple. It takes the process ID of the initial program and the location of the updated version of the program as parameters when ran. It then sends a signal to the initial program, which tells the program that an update is now available, and it also passes along the location of this updated program.

Kitsune makes use of other libraries to function. Most importantly, it uses Cilly in its compilation step. This is a rather large library that makes the base of Kitsune a fairly large program. It also complicates the compilation step for cross compilation. Changing the default compiler in Kitsune still won't change the compiler for Cilly. Instead, Cilly by default always compiles for the native system, and any cross compilation has to be done by manually setting all type byte sizes (such as perhaps 4 bytes for an int). We weren't able to successfully do a cross compilation of Kitsune because of this use of Cilly.

A second important mention used by Kitsune is OCaml, which is another programming language that was also used for parts of Kitsune. This becomes problematic because with a recent OCaml update, it broke Kitsune's functionality. This means that one must use the old version of OCaml that was used in the creation of Kitsune in order to compile it properly.

When using dynamic updates, you add an extra layer of complexity that can cause faults in the code on an update. For dynamic updates, a common problem would be the potential for data corruption. Kitsune doesn't add any specific tools to combat this, with any type of sanity checks or similar, so checking the validity of any transferred data is on the programmer to do. This has to be done for each specific variable, making it a boring and slightly time-consuming process, but also a fairly easy one if the checks are added at the same time as a variable is added or changed by a programmer. This ensures that the sanity check is valid for the variables most up-to-date correct values.

There is also no version checking, meaning that Kitsune cannot recognise version 1.1 from 1.4 which could cause issues. Big gaps between the versions could mean that there exists no compatibility between them which would crash the program after it has been updated. Most likely this would be due to data that the new version requires which the older application version has no knowledge about. However, a big advantage with Kitsune is how simple the update code actually is. While it may not add tools to help guarantee the correctness, its simplicity makes it much easier to get an overview of the update. This allows a programmer to easily check the code to convince themselves that the update is being done correctly, without introducing faults from the update process itself.

4.3.2 Usage

In order to use Kitsune in your own program, the first step is to integrate its functions into the code of your program. There are three parts of the code additions that need to be added: variable restoration which is a state transfer i.e. a serialisation and deserialisation of certain variables when doing an update. Update points which are points in the software code where the program itself checks for updates and finally return paths which are points where the software will return to after an update. This is needed to correctly do a proper state transfer otherwise if the updated software would start from the beginning again it could potentially reinitialise variables that already have a value from the state transfer and thus making the whole process futile.

For the variable restoration, Kitsune provides a function to restore all global variables in one go. Past that, any local variables that need to be restored are recovered one at a time by passing them to Kitsune with one of their functions. A convenient feature of these functions is that on initial start-up when there is no state to restore, these functions do nothing. This means that they don't need to be placed within an if-case, but can be called right after defining each variable that we need to pass between updates.

Secondly update points need to be added. An update point is a certain place in a program which is defined by the programmer as a good place to perform an update. If we recall from earlier we mentioned how we always needed to be able to return to the place of execution after an update. This is very problematic when an update can occur at any time, because we need to be able to jump back to any line in the entire program. With the use of update points, we can limit this to only being required to restore to certain convenient places in the code. This can also limit how much state restoration is needed for an update. To choose where these update points are to be placed, an initial thought might be that placing them in as many places as possible might be optimal. While this does mean that the program is more likely to update in a shorter time frame, we are talking about milliseconds for most programs. This adds little extra value over a basic but more thoughtful implementation.

Recall first here that the programs where these updates are to be used are going to be programs which are run continuously, not programs which are quickly run through and shut down. These programs do still have a lot of code that is only run once, and is generally run through very quickly, so it's unlikely that a new update will have become available while executing this specific part of the code. Instead, a better place to put update points are within repeated loops where the code is run frequently throughout execution of the program. This means that the update point will be frequently called and it's very likely that when an update becomes available, we'll be executing that certain section of code. An example of this in a server program would be to place update points between each client connection and each transmission with a client. These sections of code will always be repeatedly executed as long as the server is live.

The final code addition will be to add paths to return to the correct update point after we've performed an update. Kitsune provides functions to identify which update point the program was at when it updated. It also adds functions that check whether the program is currently recovering from an update, i.e. have we returned to the point where the update started from yet? The combinations of these two allows the programmer to use if-cases which call proper functions or use gotos to return to the correct point. In cases where the update was multiple steps of functions down in the program, it's important to retrace the same path back to the update point so that the variables can be restored in all relevant functions along the way. Once the same update point has been called as initially updated the program, the update is considered completed and normal execution should resume from that point.

Once all the code additions are completed, the next step is to compile the program. Kitsune provides the shell script ktcc which should be used to compile the program. Using this compilation tool, the program will be compiled into a shared library (.so) instead of a runnable program. This is because Kitsune also provides a program that is to actually be run, called `driver`. The shared library that should be run (i.e. your program) should be passed to the driver program as an argument, and any arguments that should be passed to the shared library are also passed to the driver program. It will then call the main function in this shared library with the corresponding correct arguments.

To update the running program, there are two ways to do it. We'll delve into these a bit more in-depth later in this chapter. The first option is to manually make the program aware of an available update. This update should have been compiled to a shared library just as the initial program. Kitsune then provides a program called `doupd` which is also run with the updated version of the program as an argument. The next time the initial shared library then passes an update point, the update will be initialised. The second option is to have the program be aware of name of the next updated version, and use a function call that mimics the same signalling behaviour as `doupd`, saying that an update is available. The program will then once again update the next time it reaches an update point. The communication between `driver` and `doupd` is shown in figure 4.2.

If there have been any variable changes between the versions, Kitsune deals with this by the use of their own language. You then add a `.xf` file that is included as part of the compilation step. In this file, all variable changes are to be included. This can be something as simple as a variable being renamed, but can also be an entirely new variable that needs to get an initial value, with this value then also being specified in the `.xf` file. This is a very convenient way to view all variable changes in one place, but also requires the programmer to edit a file externally from their actual program and learn a new syntax, albeit a fairly simple one. The advantage of this system is arguable but likely beneficial for more experienced developers in order to keep the actual program code somewhat cleaner.

4.4 systemd

Systemd is a suite that provides service and system process manager to a Linux system[11]. It was developed with the goal of unifying basic Linux services among all the various distributions. One of these newly developed services ends up being especially useful for dynamic updating. To understand what this functionality does,

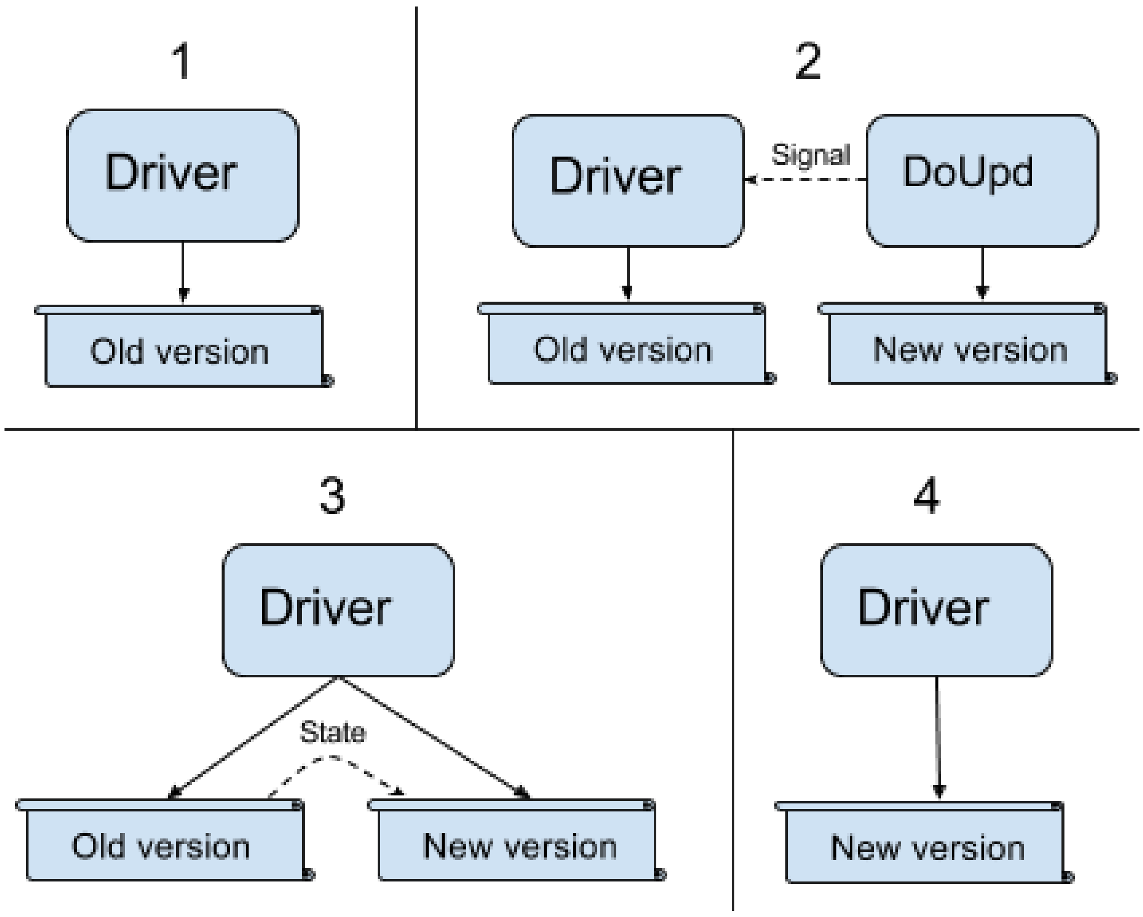


Figure 4.2: Shows the update process of Kitsune as four steps, and how the two programs 'Driver' and 'DoUpd' function relative to the software to update. In step 1, driver is only aware of the old version of the program. DoUpd is then in step 2 started with an updated program, and signals driver that an updated version is now available. In step 3, driver starts the new version and passes the state from the old version to the new version. Finally step 4 shows how the old version has been shut down and driver is now only running the new version of the program.

we first need some more background information into systemd. Systemd is directly started by the kernel when the operating system starts and then initiates the first process, called process ID 1 (PID 1 for short). PID 1 is also known as the init process. It acts as an initialisation for other processes which cannot be initiated since there doesn't exist a parent process for them to be forked from. Systemd has a lot of other responsibilities in the system but for the scope of this thesis only this part of systemd is relevant to dynamic updating. The reason for this will be explored in the structure sub-section of this chapter, along with an in-depth look at specifics of dynamic updating with the use of systemd. In the second sub-section we'll explain how to use the code we've written, and motivate why we chose to implement it as we did.

4.4.1 Structure

As the most complex part of state to save, coming up with an efficient way to pass file descriptors was our goal in using systemd. Recall that in c, a file descriptor is stored in code simply as an int, but it is used to access the actual process-specific file descriptor. There already existed a native way to pass file descriptors before systemd, namely by using something called a Unix domain socket. The Unix domain sockets are a set of sockets which are used for interprocess communication[19]. A problem that arises in this type of implementation is that you need two programs running at the same time, both the one with the receiving end of the socket and the one on the sender side, which currently has access to the actual file descriptor. This is the problem we attempt to solve by instead using systemd functions.

In a newly added function to the systemd family of functions was the ability to pass file descriptors to PID 1, a function that was added to allow for file descriptors to stay live even if the program temporarily shuts down. The reason the file descriptors stay live is that PID 1 is never shut down. The process is guaranteed to stay active, and therefore we can use this process to hold file descriptors during a program shut down. While the goal for this functionality was to preserve file descriptors during program shut down or crashes, it ends up being very useful for dynamically updating a program as well. What it allows us to do is to pass the file descriptors to PID 1 at the initial phases of a started update, then restore them from the updated program. It's worth to note here that file descriptors can only be restored to the same program on the file system, which is tracked by the file name and location of the program. To circumvent this for an update, we therefore replace the original binary file with the updated one in order for this update to function. To pass these file descriptors between the processes we make use of two systemd commands called `sd_pid_notify_with_fds` and `sd_listen_fds`.

The first command is used to pass PID 1 of a file descriptor while the latter is used to retrieve it after an update has been performed. The process is straightforward but does require some caution, especially if you want to pass multiple file descriptors. There is no system in place to distinguish between passed file descriptors other than the order in which they were passed, therefore the programmer has to take great care when retrieving them, otherwise the update can fail due to trying to use a file descriptor in an incorrect context. An example of this could be trying to send data over a file descriptor which is reading a local file instead of the intended socket. This is not as hard as it might seem, it just comes down to preserving the order in which

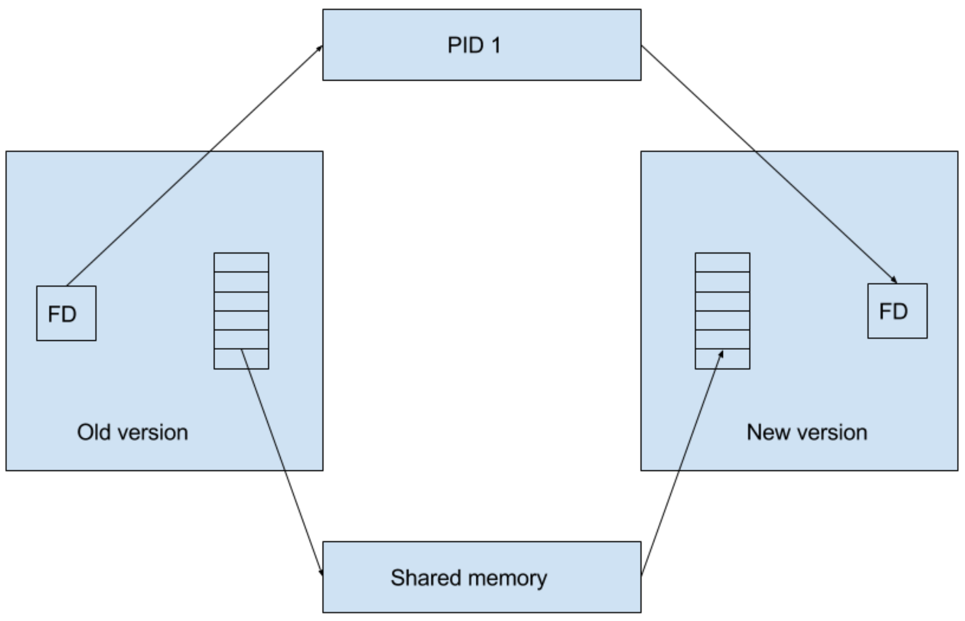


Figure 4.3: Figure showing a schematic over how a systemd update process is carried out. The file descriptor is transferred to PID 1 from the old version and retrieved by the new version of the application. The shared memory is mapped, serialised and transferred by the old version and then mapped to the new versions memory and retrieved for deserialisation.

the descriptors were saved when restoring them.

Secondly we need to serialise the rest of the state, namely variables and pointers. Our solution doesn't directly save anything off the heap, because of the earlier mentioned problems in trying to save data from the heap. Instead, pointers are indirectly saved by saving the data which they are pointing to. We'll discuss how this is done by the programmer in sub-section 4.4.2. To serialise state which is not dependent on any type of file descriptor we utilise the memory and virtual mappings to achieve our goal. There are two essential system commands we use when serialising data when an update is performed. Firstly `shm_open` is used to create and or open a shared memory object. Secondly `mmap` is used to map a virtual address space to the shared memory object. If we look at figure 4.3, we get a clearer view of what happens.

The shared memory object function is essentially that of a file stored on disk, but since we can directly map it to memory the read and write performance is greatly improved. The actual serialisation is done by a using a write command to the shared memory object. Our serialisation technique is functional but does not feature any kind of automatic analysis as the Kitsune library does. This means that the programmer needs to do the work of ensuring correct state transfer. We'll discuss our suggestions for how to do this safely in the usage sub-section, 4.4.2. Essentially this means that the programmer using this type of method has to be aware of the size of the different objects they wish to save as failure in doing so would result in loss of state and possible segmentation faults. This means that they need to ensure enough memory is allocated when creating the shared memory object that is to hold the serialised state.

Lastly to ensure that the software is restored to the proper update point we've added functionality in the state saving process to check for an existing shared memory object. This is to be used in the serialisation as well as in the deserialisation of an update. The application enters the serialisation point by looking at the modification time of the binary it's currently running. An initial time is saved when the program is launched and if it detects that the binary has been modified it serialises the state and then the running application kills itself. We utilise systemd here, which restarts the binary with very little overhead. For the restart to actually start up the updated version, the binary file which it restarts has to have been updated during program execution. This replacement of the binary can either be done manually, or could be done by some updating system such as a package manager.

4.4.2 Usage

The overall structure of this method using systemd is less programmer friendly than the previously discussed library Kitsune. The reason for this is that the programmer implementing this method has to tread carefully when performing the required steps as there exists no automated tools which can determine the size of data that is to be serialised. This has to instead be taken into consideration when creating the shared memory objects and mallocing memory space for them, as well as when reading them for deserialisation. Since this could potentially be quite error prone if done value-by-value with the programmer manually setting and restoring the location of each variable, we suggest a somewhat more robust approach for the programmer to use. By creating a struct for all the saved state, it becomes much easier to reserve the correct amount of memory required. It also becomes easier to restore the state, with each variable being possible to refer to by name rather than memory location. Finally, it's also easier to add variables to the saved memory, since the programmer won't have to manually change all memory locations placed after the variable (all of which are moved by the size of the variable).

The same is true for when migrating the file descriptors, which we previously mentioned that the programmer has to retrieve in the right order to have correct program execution and potentially to avoid program crashes. Update correctness ties in with the state transfer here as well, there exists no way of knowing if an update is going to behave properly until the update is done. Of course the programmer can check the code extra carefully, but since the program additions are in the actual update, the only way to test them is to actually perform the update. Therefore, the best way to test the update beforehand is to try to run it locally before pushing it out onto a live environment. An advantage of this method of updating is also that it has no special dependencies other than an up-to-date version of systemd (r228), so if it works locally, there shouldn't be anything system dependent that can affect the update correctness.

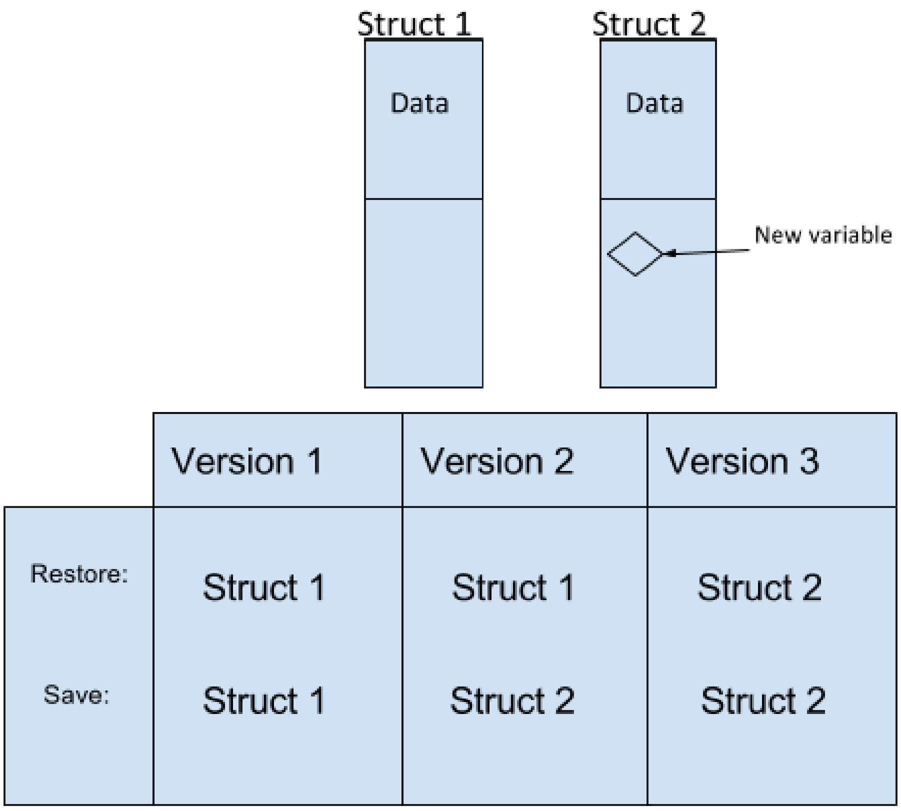
The update points of this method is not vastly different from Kitsune which we previously introduced. The same logic applies in that the update points should be placed in a loop in the main program to check continuously for update points. The update process in itself is automatised in all respects except that the person in question must replace the binary file they wish to update, the rest is handled by a combination of the application itself and the background service. The binary file replacement could however also be handled by a package manager, which would allow for dynamic updating by even some of the least technical home users.

When the program is launched after an update, there's also the important part of returning the code execution to the correct point (where the update occurred), as this too is part of our definition of state. As with Kitsune, the idea is to first go through a series of checkpoints to reach this point. This checkpoint starts with a check for an existing shared memory object. If such an object does exist, we jump to a specific area of the program which features our deserialisation point. Exactly how this point is implemented depends on what state the programmer has intended to save. The overall structure will look like this: the shared memory object is mapped to the virtual address space of the updated application. Then the serialised data is retrieved and deserialised i.e. data is put into structs to restore the state. As mentioned previously, this is the part that will differ depending on the program's implementation and what state was serialised. The program will then return to its intended running state. This can be done by either placing the deserialisation point at a point where this is possible to continue naturally or by using a label. We recommend the first as labels can make the code harder to read. If an update wasn't detected, the program continues its normal execution.

Now that we've presented how an update works, lets discuss how variable changes are handled in this program. A variable change is when a variable is e.g. renamed or added over an update. This becomes something the programmer must handle, but is not too complex. Picture the case, a programmer has decided to add a variable to version 2 of the program `server`. Let's call it `server2` to easily refer to it, and the initial version as `server1`. From `server1`, the programmer will have a struct of saved state. The newly introduced variable won't be available in this struct, so in the step where the data is retrieved from the struct, the new variable should also be initialised. When `server2` is then to be updated again, we'll need a new struct that includes the new variable, in order to save it. This means that `server2` would hold two different structs, both the old struct to restore the previous state, and an updated struct in order to save the new variable as well. For the next version thereafter, `server3`, it would then remove the old struct and both restore and save state using the newest struct. Figure 4.4 shows all three versions of `server` and which struct they use to restore and then save the state for an update.

4.5 exec

Here we introduce the third and last update method we have been working on built on the system command exec[20]. As you will notice, this last section is a lot shorter than the two methods we previously introduced. This is because both Kitsune and the systemd method are new to the unfamiliar reader and therefore require a proper introduction. The systemd method and the exec method we are introducing here share a lot of similarities in their implementation and the main difference lies in how the update points are handled and how file descriptors are saved. While they are



	Version 1	Version 2	Version 3
Restore:	Struct 1	Struct 1	Struct 2
Save:	Struct 1	Struct 2	Struct 2

Figure 4.4: Two structs used in the program `server` over the course of two updates, from `server1` to `server3`. For each version of the program, it shows which structs it uses to restore and save state during an update. Struct 1 is the original struct, and struct 2 is the updated one which includes the added variables.

done using a kill command and restarted with a background service in systemd, exec takes care of all this as we will see below. Again, the section starts with introducing how the technique works, then it goes into a sub-section explaining how the usage differs to that of systemd.

4.5.1 Structure

Exec is a family of system commands specifically designed to replace a currently running process with a new process from a different running image[20]. The command has some interesting design features which coincides with features that we are interested in when designing a dynamic update method. The exec command does not preserve any of the variables when executing the new binary, but we already presented a way to transfer data between programs in the systemd section 4.4.1, which we reuse here. There is however some information that is transferred between the two running methods when exec is called. The most interesting to this thesis is the ability to transfer the file descriptors between methods. This is shown in figure 4.5, where the file descriptors are now shared between the versions of the program. The command also transfers some other data that could ease the programmer's experience while updating but is not necessary for the method to work. These include transfer of signals, process id and parent id to mention a few of them.

As we do not rely on systemd any more for the restart, the whole update method is done differently. To detect an update we instead continuously look at the time at which the currently executing binary was last modified on the file system. If the binary is determined to have been modified the update process is initiated with creating a shared memory object and serialising the state that we wish to transfer. After this has been done the exec command is executed and the old version of the application is replaced by the running binary of the updated application. The updated application initiates with checking for any shared memory objects, if these are detected the state that has been saved in them is deserialised and put into the proper structures of the updated application. This is a full cycle of an update using the exec method. It does save the programmer some work and possibly trouble by not having to worry about the transfer of file descriptors and the order of them. There is also no need any more for using background services, as exec does not need them, which does save some work.

4.5.2 Usage

The usage is essentially the same as for systemd, as variable restoration is done the same way. The only difference is that file descriptors can be considered ordinary variables in this way of updating. There is thus no need to pass file descriptors between the programs, simply saving the ints which are used to access the file descriptors is enough. There is also a minor difference at the start of an update, where instead of killing the running program you simply start the updated one directly.

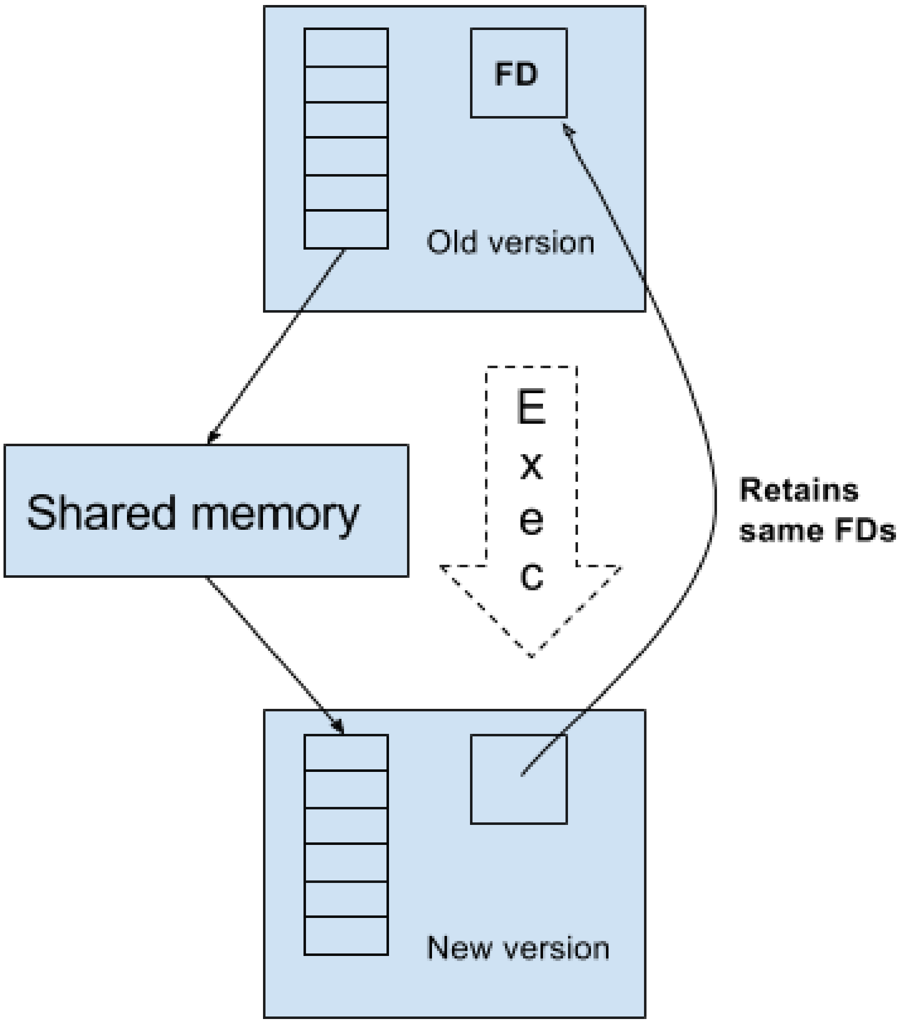


Figure 4.5: Figure showing the update process of the exec method. File descriptors are transferred automatically when the exec function is called while the shared memory has to be created and mapped to the old version where data is serialised and stored. The new version then maps the shared memory and deserialises the data.

Chapter 5

Results & Discussion

5.1 Introduction

We want to be able to compare the different methods we have presented you with in the proof-of-concept section to provide context to the reader what different trade-offs with each of the methods we have developed and tested. We chose a benchmark to outline the main differences in terms of performance. This chapter should provide the reader with an immersed view of the different strengths and weaknesses for the methods which we have explored in this thesis. The reader should not take the results presented here as a final conclusion of what methods are the best as there are other factors than just performance to take into consideration when using these methods.

There are two main reasons that we chose performance as a main benchmark for our results. Firstly it has to do with the difficulty to quantify other results such as code complexity and difficulty to implement among other things are that these are more subjective areas which are highly tied to personal opinions. The ability to reproduce results is very important when performing research as it strengthens the conclusion of the research project. Secondly this research project was conducted in collaboration with Axis communications where a lot of their software is run on embedded systems, therefore it was important to present benchmarking tests since resources as explained in earlier chapters are often a lot more scarce on embedded hardware.

Despite the complexity to implement a method being quite opinion based, we have done what we can to quantify it, since how easy the code is to actually implement is one of the most important aspects of implementing dynamic updating, so we wanted to include something comparable regarding that aspect. We've done this by adding a section regarding code additions which compares the different methods in terms of line of code to implement for the same program which can serve as some pointer of hardness to implement. We will also compare difference in binary sizes and difference in toolchain sizes. A toolchain can be said to be the amount of underlying dependencies and programs needed to compile and run a binary file. For example the toolchain of a simple C program which outputs "Hello World" would simply be a functioning C compiler such as the popular `gcc`. We also discuss our own opinions on which option is the easiest to use as a programmer.

This chapter will be divided into six different subchapters. We will begin with an introduction to the design of the setup which will introduce the reader to the system we have used, which tests we have done and the motivation behind choosing these. We will then continue with update time results which is a presentation how long the different methods takes to perform a full update which will be presented with figures and possible reasons for why these results were achieved. After that follows a presentation and explanation of the memory usage of the different methods when performing an update, much similar to the update time section the reader will also be presented with graphs and a discussion of what reasons the methods have for performing differently. We then continue with the sub-chapters code additions and file sizes which will compare the number of lines required to implement the different methods and the needed dependencies and sizes required for the sizes. Finally, the chapter is concluded with a discussion of the conclusions we can draw from the results we have presented in this chapter.

5.2 Design of test setup

We performed our test on UNIX based system with the Debian distribution. The computer in question has an Intel i7-2600 CPU with 16 gigabyte ram memory. We chose to perform our tests on a PC as opposed to benchmarking on an embedded system as the main goal was to acquire whether the methods had any performance differences and if they had which were they. These test could have been carried out on an embedded system as well, but it added extra complexities for the compilation step when using Kitsune. And since the goal was to gain an understanding of performance differences, the methods should not behave differently relative to each other when run on a different system, even though the actual running time may be faster. In terms of memory usage there should be no difference. Therefore, we chose to use a pc as it was faster to implement the proof-of-concept there, since building the programs for embedded systems requires more time due to them having more extensive tool chains.

The program which we updated to test all this is our basic server/client implementation presented in the proof-of-concept chapter. Each of the three methods have been separately implemented on the server such that it is capable of dynamically updating using these corresponding methods.

The update times were timed by simply inserting a function call which reported the time before the update and after the update had finished and then taking the differences of these two times to get the time it took for the program to restore itself to the same execution point. These times were clocked 20 times. We did this for two reasons; firstly we wanted to be able to get a mean value of the update time. Secondly we wanted to measure enough times to see if these times were representative or if the update times was irregular. If they were irregular the tests might not be good enough or there might be other factors at play. This first test which measures the update times can be seen as most representative in terms of performance test as factors such as the CPU clock and cache will be a big factor here.

The second test that we performed was a memory usage test. This test is also an essential performance test and as the thesis was done in collaboration with Axis we felt that this test was one if not the most essential for them as excessive memory usage can cause problems on an embedded system or even not possible to run due to too high memory usage. The memory usage was measured with a tool called Valgrind which is a collection of profiling and debugging tool. We made use of the

	Kitsune	systemd	exec
Mean	0.460	0.419	0.104
Std. Dev.	0.0557	0.0692	0.0127
CV	0.121	0.165	0.122

Table 5.1: The mean, standard deviation and coefficient of variation for the update time of each of the updated programs with their respective methods. The mean and std. dev. are both measured in milliseconds, whereas the cv has no unit.

Valgrind tool massif which is a heap profiler by default but has the capabilities to also measure stack memory usage if the user chooses to enable it. The test were carried out a multitude of times on all three methods, similar to the update time measurements, but showed exactly the same behaviour on each run.

Code additions are measured by taking the difference in lines of code from the original implementation of the program to the different methods. File sizes are also self-explanatory, we simply measure the number of bytes that the different files and libraries consist of.

5.3 Update time results

Below we present the graphs for the update times of the three different methods starting with Kitsune, then systemd and finally the exec method. We also present the reader with a combined graph of the methods to give a weighted impression of how they look compared to each other. We have calculated the mean of the values, the standard deviation of them, and finally the coefficient of variation (CV). As our measured values are simply a sample, not an entire population (there isn't any way to measure the entire population in our case), we have used Bessel's correction in calculating the standard deviation. The CV is calculated as $\frac{\text{std. dev.}}{\text{mean}}$, and can be used to compare the variation between different data sets since the value of the CV isn't affected by the actual values themselves, only their variation relative to each other. For all of these calculated values, lower is better.

As can be seen from table 5.1, systemd displays the largest relative variance of the methods. Meanwhile, exec and Kitsune have a significantly smaller relative variance, with Kitsune being just slightly more stable than exec. The variance we see is most likely due to a combination of the operating system and memory accesses. With operating system noise we mean that depending on amount of processes running applications can show a variety in execution time due to the time slices they are given and possible cache misses caused by context switching. This problem can be minimised by shutting down as many other programs as possible, but the operating system itself and any programs it starts and controls will still leave some of this noise. Another explanation was that the test data was collected in a continuous series, there were no pauses between the data collection of the tests and caching in the memory might have influenced some tests.

Even so figure 5.4 gives us a very clear picture of what is going on, the exec method is not only faster in all aspects compared to the other two it also displays an essentially equal stability to Kitsune, without any greatly diverging measurements as Kitsune has. In Kitsune's case when the update process occurs the two versions of

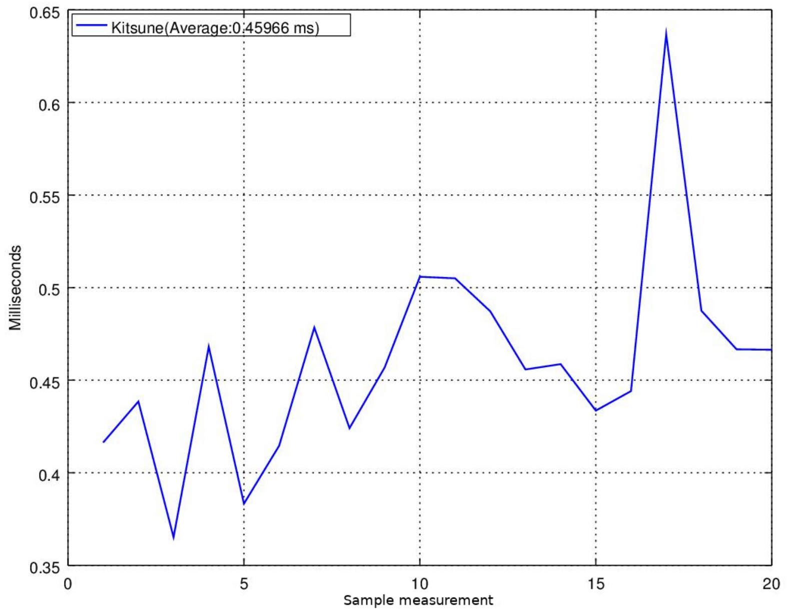


Figure 5.1: Shows Kitsune update times on the vertical axis and the number of updates on the horizontal axis

the program are run simultaneously with positionally independent memory meaning that they essentially share the same physical memory. We suspect that Kitsune's way of mapping these from the old version of the program to the new brings some overhead to the update process which is what we are seeing. In systemd's case we are confident that the transfer of file descriptors is causing the wide ranges of update times that we are seeing. We draw this conclusion from the fact that exec makes use of the same type of state transfer when performing an update but passes file descriptors through inheritance while systemd sends and retrieves them through an external process.

Systemd and Kitsune both also display a single highly divergent measurement. Systemd 0.15ms which is way below the average update time and Kitsunes at 0.65ms which is way above the average update time, it's hard to determine exactly what caused these spikes as they do not seem to be a natural occurrence. That is there is no other spikes that show this extreme behaviour out of the 20 tests that were measured. Since they both are one time occurrences we do not credit them to say something about the actual process and is most likely the product of system noise present.

Overall the conclusions we can draw from these tests is that the exec method is on average 0.3-3.5ms faster than both the other update methods with the bonus of it also having the smallest amount of span of oscillations of all the methods, ranging only at it's worst at 0.04ms compared to the 0.2ms ranges we see in Kitsune and the 0.3ms ranges in Systemd. This suggests to us that the exec method is not only the fastest of the three, but along with Kitsune it's also the most stable and reliable as we see from the measurement of the coefficient of variation.

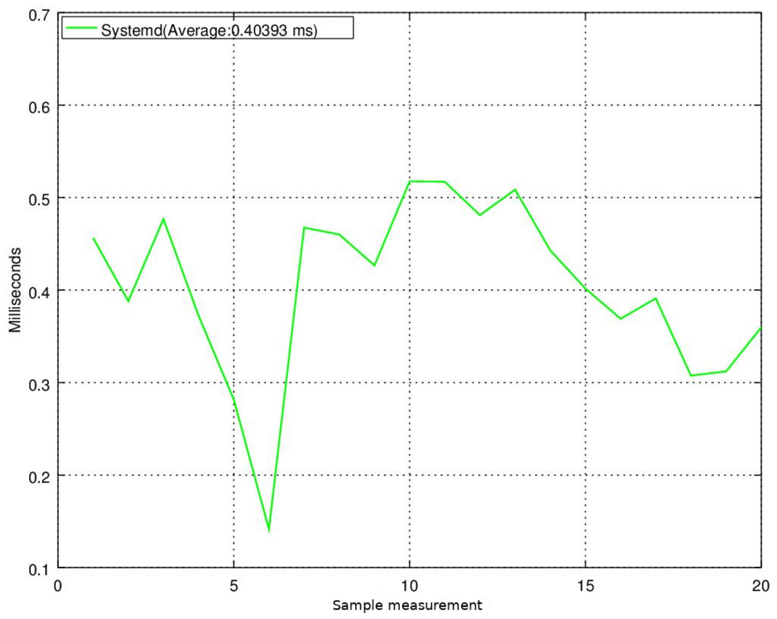


Figure 5.2: Shows Systemd update times on the vertical axis and the number of updates on the horizontal axis

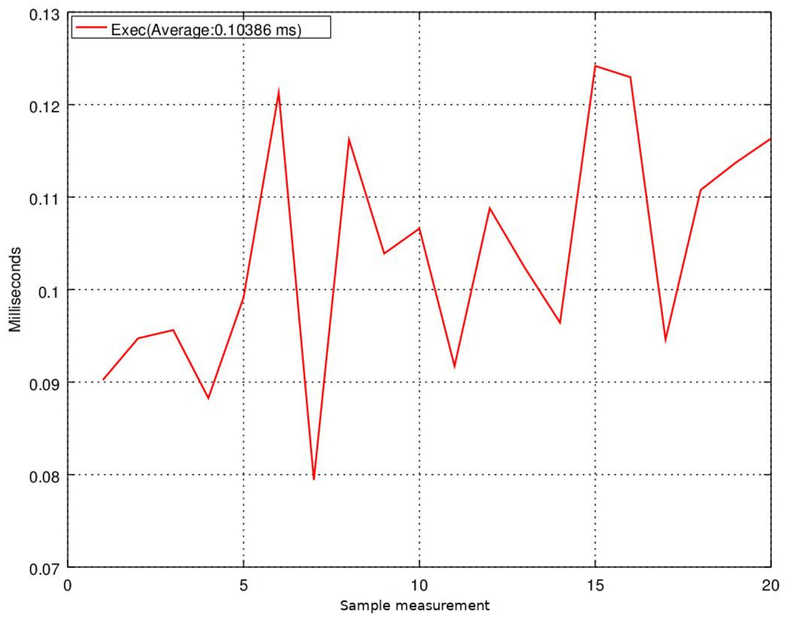


Figure 5.3: Shows Exec update times on the vertical axis and the number of updates on the horizontal axis

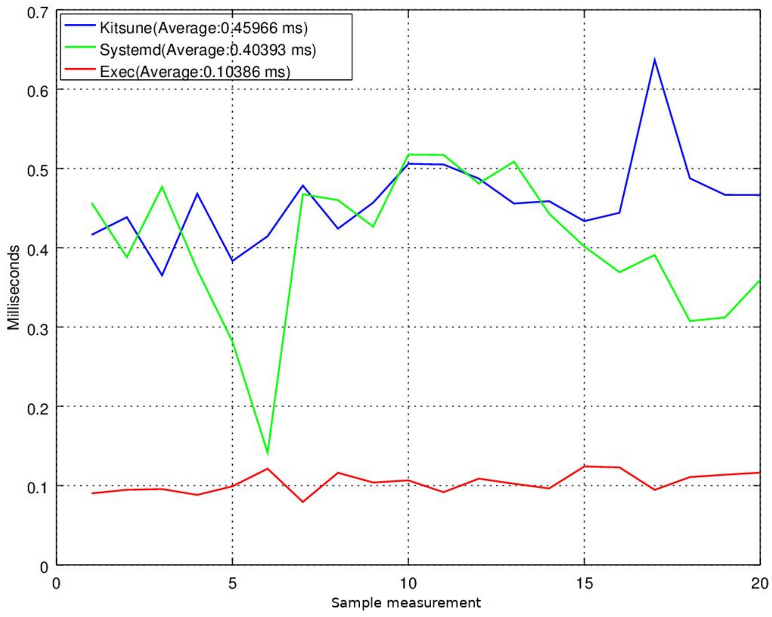


Figure 5.4: Shows all combined update times on the vertical axis and the number of updates on the horizontal axis

5.4 Memory results

For measuring the memory usage, we measured both heap and stack memory usage and combined them into one graph. We don't actually care about the specific places where memory is used, but only if our total memory usage is higher than we have available. Therefore, combining them into one graph only simplifies the readings. For the memory usage we present three different graphs. These show the memory usage of a program from the start of its execution, during the update taking place, and then until exiting the program. There is however an exception for the systemd updating system. Since the updated program and original program have different process IDs, we had to run the memory collecting tool on each of them separately. The interesting data is however only on the initial version, as that is the one to actually save its state in the shared memory, the secondary program merely restores that data and doesn't allocate any extra memory above what the initial program did. As such, we'll only be showing the memory usage of the non-updated version of the server for systemd. The other two options, Kitsune and exec, both include the actual update and the updated program shut down in their graphs.

The graphs are plotted with memory usage on the vertical axis and instructions executed over time on the horizontal axis. This means that the time it takes for us to start the update after starting the program won't have as significant of an effect on the graph, but rather only the initial code execution will. The memory usage is measured in kilobytes, with the peaks varying between 1 KB to 60 KB for the different programs. The graph visualisation program cannot differentiate between measuring only the heap versus measuring both stack and heap memory, so the label on the y-axis is partially incorrect. The memory usage of the server using Kitsune to update is plotted in figure 5.5, the systemd version in figure 5.6 and the exec version

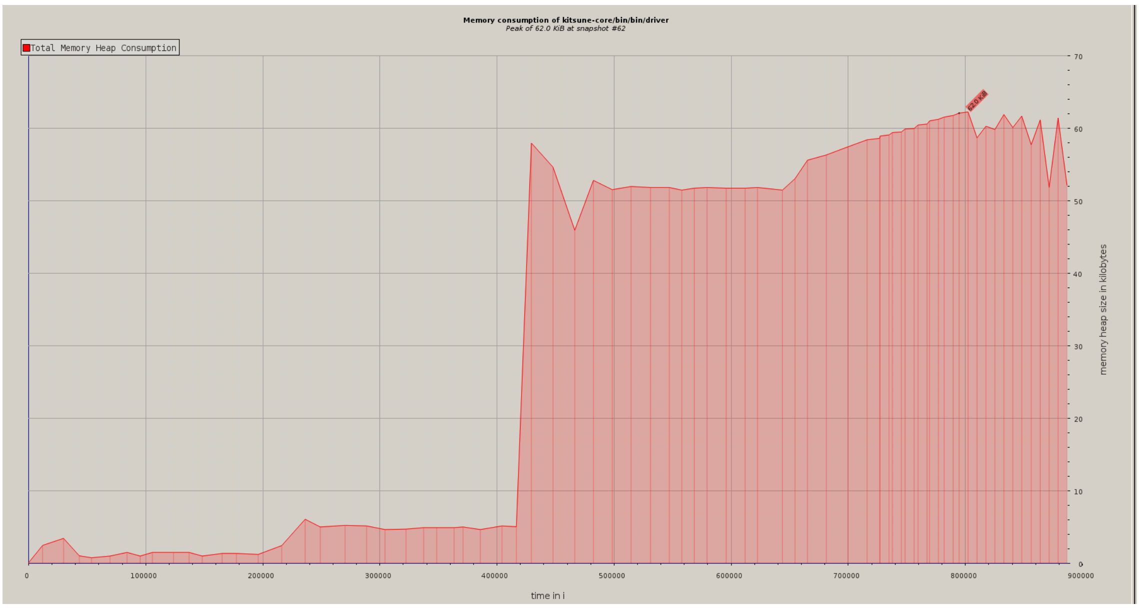


Figure 5.5: The memory usage of our two server programs which dynamically updates with the use of Kitsune. The update occurs roughly halfway through the graph, where we see the memory usage increase from 5 KB to 62 kB.

in figure 5.7. For Kitsune, it's clear that the update occurs just roughly half-way through the graph. For systemd, the update starts at the end of the graph, due to the reasons mentioned just above. We only see the first half of that update in the graph, namely the part in which state is saved before program shut down. Finally, for exec the update again occurs towards the end, but here we see the entire update process including after the restoration of state in the updated version. The peak for exec is significantly smaller than for the other implementations but it is still the same server update that is occurring.

For Kitsune, the memory usage is especially interesting because it doesn't actually decrease again after the update has occurred. This could hint at there being potentially memory leaks that occur in the update process. It could also be that the `driver` program doesn't release the memory of the old version of the program until the updated version itself gets updated (meaning that two updates have to take place before the memory is released). This could just be a solution to improve the execution time when performing multiple updates, such that when a second update arrives, there is already memory available to use to transfer the state to the newer version. Regardless of the reason to it though, the memory usage is not that large for a modern computer system but could pose potential problems on systems with less memory. Since this is a very small scale program it's troubling that it adds so much extra overhead to such a small program, especially if it was to be run on an embedded system. It's fully possible that the overhead is constant, and won't grow in size as the scale of the program grows. But we equally can't rule out that it won't keep growing along with the size of the program and/or size of the state that's being saved. As such, Kitsune is potentially quite worrying in terms of its memory usage, at least when looking at embedded systems.

When looking at the systemd graph, there's a few points worth discussing. But

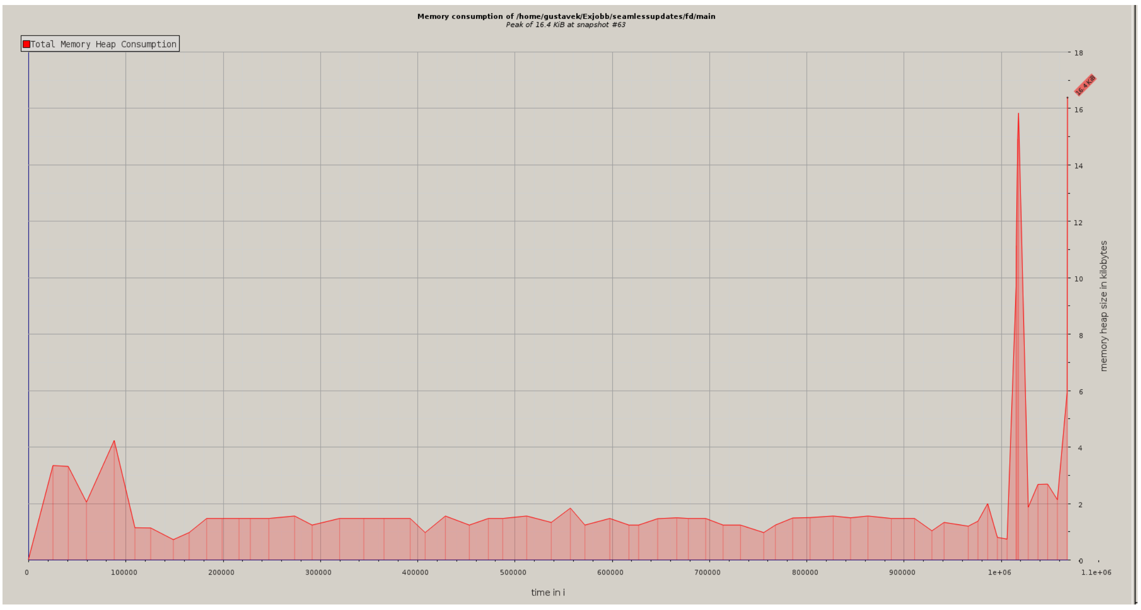


Figure 5.6: The memory usage of the non-updated version of the server, which dynamically updates with the use of systemd. The update occurs at the end of the graph at the secondary peak, where we see the memory usage increase from 2 KB to 16.4 kB.

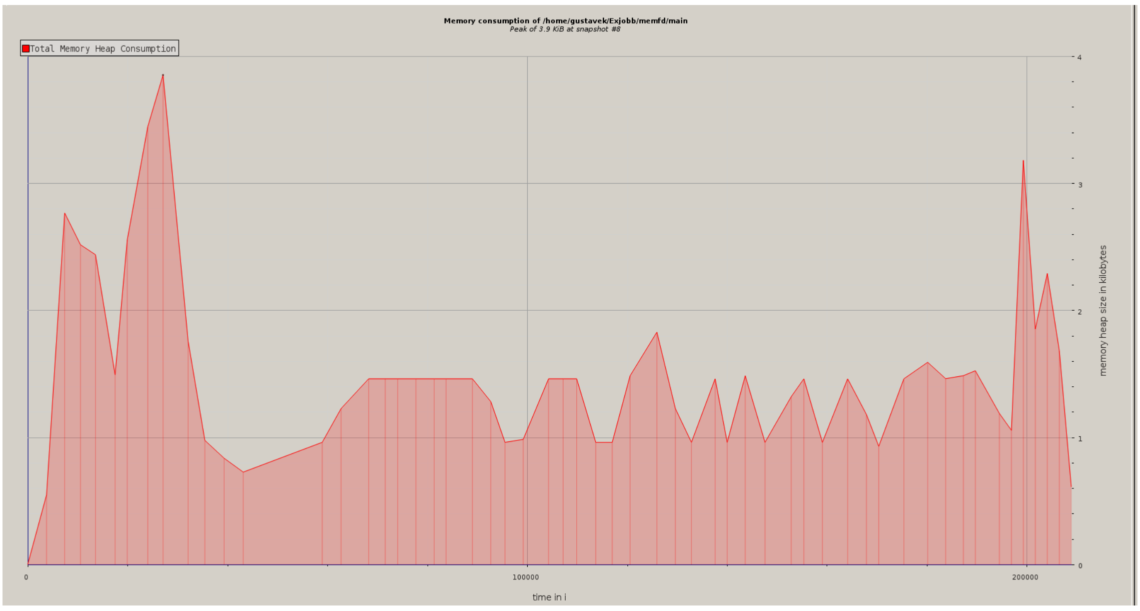


Figure 5.7: The memory usage of our two server programs which dynamically updates with the use of exec. The update occurs at the end of the graph, in which the memory usage peaks to 3 KB memory usage from its previous usage of 1.5 kB.

especially interesting is the first of the two peaks in terms of memory usage. Clearly there is something causing a roughly equally large memory increase as actually saving the state into shared memory (the second peak). Since the program is nearly identical to the exec version, besides then of course the usage of systemd to save file descriptors, this would mean that the memory peak could be coming from the functions in which the file descriptors are passed to PID 1. This is somewhat worrying, as we've already seen that this process is relatively slow. If it on top of that also requires a significant amount of memory in order to pass these file descriptors, it could make the entire process much slower in a memory restricted environment. We also see a large increase in memory usage at the end of the program's lifetime. Similarly to Kitsune, we don't know how this scales into a bigger scale, but it could be troublesome. However, the memory usage does decrease back down to the lower memory level when the updated program starts (not shown as the relevant section of this graph is simply the restore section and only varies between 1.5-4 kB, much lower than the original program).

Exec has a very low memory usage overall. This graph also shows the entire update process, including the second running program. However, because of the properties of exec in which the memory sections of the programs overlap (remember it's similar to a `fork()` command), so less memory is used in the transfer of the state. This also means that it's not nearly as obvious that an update even occurs, because the update peak is still smaller than the initial start-up of the program. This start-up peak can be seen in all the programs, but is relatively larger for exec because of it using so much less memory to update. The exec version of the program never goes above 4 kB.

To quickly compare the three, it's obvious that exec is the best option in terms of memory use. It uses the least amount of memory for the update and it also has the smallest comparative peak versus its normal execution memory usage. The update only doubles that memory usage, whereas for Kitsune and exec it goes up by about ten times their normal memory usage. This is likely a lot due to general overhead of the specific implementation, combined with certain functions we use requiring a lot of memory usage to run. This suggests that exec might be the option that is most likely to scale well into larger programs, however since we haven't properly investigated if Kitsune's or systemd's memory overhead is constant independent of the program size, so we can't definitively say which will be the best suited for larger applications rather.

5.5 Code additions

Our test setup consists of 78 lines of code. This is much less than any of the actually used programs in Axis cameras, but it makes this current case the worst case for code additions. There is going to be some static overhead for each program, and then additions depending on the amount of variables to save. In a small program like this, that static overhead will have a much larger impact than it would in a large-scale program, and so the total code additions can be perceived to be much worse than they would be in a more realistic program. This will serve as a base comparison towards our three other methods, where an increased amount of lines of code, that is the larger the difference is between our test program in amount of lines of code the more complex the method will be considered. There are of course other things that can be considered to add complexity to a program as well, and we will try to point these out. If we start with the method that needs the lowest amount of lines of code needed, Kitsune comes in at first place having 96 lines of code to get a functional test program. A total addition of 18 lines of code. Exec needs 118 lines of code for a functioning implementation and systemd 126 lines of code. Percentage wise this is a 23% increase for Kitsune, 51% for Exec and 61% for systemd. The paper which presents Kitsune states that the amount of code needed to implement Kitsune is generally related to the amount of heap resident state and initialisation rather than the overall code size[2]. As we have only implemented our methods on one test program we can only give a qualified guess whether this holds true for systemd and exec as we would need further research to confirm this. If we look to see what the main difference between exec and systemd this gives us some points on what would produce more lines of code. The main difference in systemd and exec is the transfer of file descriptors which needs to be taken care of manually in systemds case while exec transfers them automatically. Therefore, a program which holds many file descriptors open such as writing or reading to several files or servers would further complicate an implementation of the systemd method and create more lines of code, while exec would not be affected by such a thing. Secondly the thing which does produce more code in both exec and systemds is the actual update points which complicate things the most. While Kitsune can handle this kind of transfer in simple one line function calls both our own developed methods currently have no possibility of doing so and require more extensive checking. Kitsune handles an update point in the following manner `Kitsune_update(‘‘name’’)` both our methods require more extensive checking at update points and as such produces more code. Much of this error checking is directly implemented into Kitsune's own library, and we could do the same to help the programmer implementing our methods to avoid much of this tedious work. This turns into a relevant section for future work, which would likely improve the current number of needed code additions.

It's very hard to conclude that more lines of code automatically produces more complex code as it is somewhat subjective. What could be argued is that more lines of code would require more time to write and therefore the methods we have developed are then more complex as they require more work to make fully functional. From our own experience, it is somewhat easier to write and read Kitsune's code as it can be more easily integrated into the existing code without significant changes. The additions of if-cases for exec and systemd, which can switch the entire layout of the code, are harder to follow and therefore also somewhat harder to write. There are also more things to keep in mind, with variable initialisation and placement in memory, that their code complexity simply is just a bit more than Kitsune. After having worked with them for about two months though, we don't believe that any option is especially hard to work with, and even though Kitsune is somewhat easier it doesn't make systemd and exec unviable choices.

5.6 File sizes

Next we'll be looking at various file sizes. We'll be comparing two different types of files. First we'll look at the file sizes that are needed to compile the program, i.e. the file size of the source code. This problem can be avoided completely by cross-compiling. In order to add some extra flexibility to the program, being able

	Base program	Kitsune	systemd	exec
Source code	N/A	31 600	6	7
Binary	16	142	27	28

Table 5.2: The file sizes, in KB, of the source code required to compile the program with each library, as well as the binary file that's produced when the libraries are compiled into the binary by being statically linked. The source code for the base program is N/A because it will be the same for all four implementations and is therefore considered to be 0, and we're interested in the additional usage that's added on top of that.

to compile it locally on a hardware restricted system could be beneficial. Secondly, and somewhat more importantly, we'll be looking at the file sizes of the binaries. All the dynamic updating techniques we've tested are statically linked into the binary, so the file size should reflect which updating method is being used. While this is unlikely to be the biggest problem in an embedded system, with disk space being relatively cheap today, it is still relevant and could still be a limiting factor.

The comparisons of the file sizes for both the necessary source code and the produced binaries are presented in table 5.2. Since all methods produce two binary files, an initial version of the server and an updated version, we'll simply present the initial version's binary. The code change between the files is simply the removal of a single duplicate line, so the file size between versions is negligible.

It's quickly obvious that Kitsune is a lot larger of a library than our systemd and exec implementations, especially in terms of the source code. However, it's worth noting once again that the file size of the source code is easy to circumvent by instead cross-compiling the software from a system without the same hardware restrictions, should disk space end up being an issue. Also, for the binary Kitsune adds a lot more to the file size of the program than the others. Most likely though, it won't add more overhead into larger programs, as the majority of the code is now already available in this basic binary. In other words, when saving a larger state, the appropriate function calls to do so are already included in the file size of this binary. Thus, the addition in terms of binary file size will only be the addition of calling these functions an extra time, thus meaning that adding Kitsune to a larger program is unlikely to produce significantly more overhead than systemd or exec. Instead, it's most likely that the file size difference will stay at about 120 KB, which is a very small difference for the disk space of modern systems, even for embedded systems.

5.7 Combined Evaluation

The most important aspect for dynamically updating software in an embedded system is the memory aspect. Since most solutions focus on duplicating the state for at least some period of time, this makes the total memory use become a very large factor, as we saw from our results. Our small scale tests are worst-case scenarios as mentioned previously. Additions to a codebase this small will produce significantly larger overhead than with a larger codebase, but it clearly shows the effects they can have on memory usage. The secondary most important aspect is the ease of use for the programmer, which is closely related to the code additions. If dynamic updating is too difficult or too time-consuming to add, then it's not likely to see any common use, even where it could be beneficial. All the update times are relatively low, and in most scenarios selecting a good update point, which provides minimal end user interruption during the update, will likely end up being the most important aspect for the execution times. Finally, for the file sizes, cross-compiling for embedded systems is already a common practice, so source code file sizes won't be a large hindrance. And similarly to the execution time, the binary file sizes end up being so small that they don't provide any especially large advantage to any method.

So, with exec method being the best choice when it comes to memory usage. Also, it's the best in the more minor aspects such as execution time, it does seem to be the overall preferred choice. The other option would be to select Kitsune, due to its strength in usability, with low code additions required and well-developed state change system in place. However, there are other usability aspects to Kitsune outside of the code that make it less usable. The library uses other external libraries, one of which it requires running an outdated version to function correctly. Another of these external libraries is difficult to cross-compile together with Kitsune. It's in no means impossible to fix these issues, but they require a lot of extra maintenance work which exec simply does not have. Exec does lack in being simple to use to transfer variables between versions, however its other advantages relative to Kitsune's disadvantages does still place exec as the choice we consider to be the best for most scenarios within Axis, and embedded systems as a whole. Generally we feel that even though state transfer is somewhat more complicated to do with exec than Kitsune, it is worth it due to Kitsune's large amount of dependencies which are much harder to get properly working.

Chapter 6

Related work

6.1 Introduction

This chapter serves as an evaluation and review of previous work done in the field of DSU or closely relatable to this thesis. The goal is to present the reader with a short but comprehensive background on what previous work has been done in the field, as well as make the reader aware of any significant differences with these different projects to what we've presented in this paper. It should give a brief insight into the type of work currently being developed in the field and a look into what other options there are and could be for dynamically updating programs.

6.2 Ginseng

Ginseng can be said to be one of the predecessors to many of the DSU research projects which exist today. Ginseng was developed in 2005 by I.Neamtiu, M.Hicks, G Stoyle and M.Oriol. The research project set out to develop a DSU method which could accomplish the following three things: Firstly the method should be easy to implement and easy to verify that it's been correctly implemented. Secondly it should be easy to change data structures in a program between updates i.e. the DSU method should not hinder the programmer from updating their software as they best see fit. Lastly it should not require a massive overhaul of the application to make it functional with a dynamic software update. Looking over the structure that is required to fully implement ginseng into a program it's very easy to see that a lot of things are still in an experimental stage compared to later projects. The program code is much more rudimentary and does in fact require a significant overhaul from the original code in order to make it work with ginseng. From the examples they have supplied in the paper more or less all code needs to be changed or re-factorised for the analysis tool to function properly. The Kitsune paper does several comparisons to ginseng[2]. Not surprisingly Kitsune excels in nearly all instances, but without ginseng the well needed lessons for developing a better DSU method would probably not have been available. As such ginseng can be seen as an early effort to try to make the whole process of dynamic updating easier for all parties dealing with it, a stepping stone in the field. Despite its more elementary implementation than its successors it brings a lot to the table. The library includes an established way of rewriting the code and provides us with automatic update points. An analysis tool which provides static analysis which is supposed to help the programmer avoid bad update points, and it does so with very little overhead. The most impressive part about ginseng is its analysis tool which today still, guarantees correct updates due to its ability to perform static analysis. Gingeng is one of the few DSU tools which can guarantee this[7]. Ginseng compared to our methods exec and systemd can be seen as superior in many ways. However just like Kitsune it suffers from the added cost of having a larger toolchain and more dependencies.

6.3 Kitsune

What differs Kitsune from previous tools such as Ginseng and UpStare, both which are mentioned in the paper, is that updating with Kitsune is not only easier but the process is highly customisable and more user-friendly than previous tools. It also adds extremely little overhead to the update process compared to any other tool that is currently available[2]. To achieve this they have developed a framework which is to be integrated with the existing application that the programmer wants to update. This allows the programmer to migrate local and global variables and to set update points where the updated application takes over. The researchers have also developed a new tool which they call xfgen which allows for new variables and data structures to be initiated beforehand, or changed in case the previous version now has outdated and incorrect values.

There exists similar research and developed tools before Kitsune which in many ways, achieve much the same thing as the authors of this research paper have accomplished. The authors are however very much aware of both the issues and opportunities that this presents. As such they take valuable lessons on what previous research has done well and what could be improved in order to develop Kitsune. Kitsune thus brings some interesting improvements to the field of DSU. For example compared to the previous mentioned program Ginseng along with some older programs, Kitsune solves one of the security issues that these suffer from. By transferring the complete state of the application instead of updating individual functions it strictly prohibits code injection and can also deal with compiler inlining, thus making it both more secure and faster than most predecessors. There exists tools which handles things similarly but adds more overhead, often by using more memory than Kitsune does. The authors are however very clear about Kitsune not being perfect and that it suffers from some issues, mainly when dealing with transfer of objects and pointers using xfgen. Other notable contributions lie in their test suite where they update and evaluate performance and overhead of several programs many of which have never been tested with a DSU tool before. The most notable being the application TOR which is a project for communicating anonymously over the internet, which has a code base of over 76 thousand lines of code.

6.4 Rubah

Rubah[8] is a Java continuation of Kitsune. The authors present Rubah as the first efficient dynamic software system which also is portable. Portable in this case means that Rubah can be used with any underlying JVM (Java virtual machine) compared to previous DSU systems for Java. As Rubah is a continuation of Kitsune, much of the framework and methodology used in the Kitsune project is implemented in a similar manner for Rubah. Rubah does however employ newer algorithms and a somewhat different strategy for implementation than Kitsune employs. Much of this is due to the change from C to Java, as Java is not only a much more type-safe language than C, but it is also object oriented and therefore different strategies can be applied when performing an analysis for a state transfer.

One of the key changes in using Rubah over Kitsune is how variables are restored. In Kitsune, each variable is restored individually, and any variable changes are specified in an external file for each. For Rubah on the other hand, it is possible to restore entire objects. This means there are often significantly fewer lines of code needed to restore the variables, since each object can hold multiple variables. To deal with any changes to the variables within an object between updates, the programmer can write their own object transfer functions, that specify how an object is to be transformed when it is changed during an update. An advantage of using Rubah though is that a variable which is added into an object will by default be created with its default value, without requiring code to manually initialise it. This means that the programmer needs to write yet less code, and decreases the likelihood of an error in the update process. Rubah also adds the ability to wait with updating objects once the program is updated, and first transfer their state from the old program once the object is used in the new program. However, they concluded that this was generally a slower process, because the compiler wasn't able to optimise the code as well when using this technique.

The most important addition for Rubah though is their ability to run tests during an update. A large problem with Kitsune is its inability to ensure that the transferred state is correct. Any faults in this state could cause the program to run in any possible unpredictable way, so it's especially important that this doesn't occur. For Rubah, the programmer can add test cases which are run during the update process, thus checking that the transferred state is still correct for the new program. This makes the updating process much safer, as it's significantly less likely to include bugs when the programmer utilises this technique.

6.5 Multi-version Execution

This next paper was written with a quite significantly different goal in mind than was the aim of our thesis. Their goal is namely to avoid program crashes. Their proposed solution to this problem does however happen to also become very relevant to dynamic updating. We'll motivate what applications their solution has after we first present their paper.

In modern day software, new updates often result in the introduction of new bugs into the code, bugs which weren't identified during development. This is problematic, since it makes users running functional software to become less prone to updating it. Much fewer people are willing to take the risk of updating when it means they might be getting dysfunctional software. This also means that critical security or functional updates to the program go missed, since many users won't ever update to the versions that solves these bugs.

The paper "Safe software updates via multi-version execution"[12] investigates the possibility to run two different versions of a program simultaneously, in order to avoid crashes. The way this is done is by monitoring both programs, and when one crashes it will be restored to a previous state and then instead call the same function on the secondary program, and passing the output from that function back to the program that originally crashed. There are three different steps that have been implemented in this paper. First at compilation, an analysis is done of the programs in order to properly link their execution. Secondly, a monitor is put in place that monitors all external behaviour (input to and output from the program). When this monitor identifies any differences between the two running programs, the third part takes over. This has to choose how to deal with the differences, or recover from a potential crash. The general solution is to have the old or crashed version instead run the newer version's function, then ensure that the two programs are again in an equal state.

In the feasibility study of the paper, the authors looked into the amount of changes to LoC per program revision, as well as the corresponding number of changes to external functionality. However, in the comparison of changes to the external functionality, they also perform a post-processing step in which they ignore many parameters and return values from the programs functions, because they are pointers to memory and thus can't be directly compared. With pointers being such a crucial part of programming languages, it's fair to assume that properly comparing the values that these pointers refer to would change the result of the feasibility study. It is also mentioned that the program can in fact handle this situation, so a reasonable solution would be to utilise the functionality of the program to complete this section of the study.

There is still a lot of functionality missing in the current implementation, that is mentioned as something to be added later. One of the most significant of these is the ability to restore a program after the two programs diverge. A diverge is any point during the program execution when the two programs give different output. This causes a shut down of the version with the worst output, which is by default the older one unless the new version has crashed. Presumably, a diverge between programs of different versions is a common enough occurrence that including it in the implementation for the writing of the paper should have been done. Without the ability to restore the program, the dual setup of running two programs at once will only be able to stop a single crash, and only when the programs never diverge before the point of the crash, thus greatly reducing its value as a safe updating tool.

So, lets look at the relevance of this paper to dynamic updating. In multi-version execution two different versions of the same program are run in parallel, choosing the "best" output (typically, the newest version's output) of each function call as the actual output from the function. This becomes very applicable to be used as a dynamic updating tool. Whenever a new version of a program is released, the older of the two programs could be replaced and the newest could then be restored in accordance with the above (not yet implemented) functionality at an appropriate point. Not only would this allow for seamless updates, but it also adds the added benefit which the paper itself focuses on, namely avoiding program crashes.

Chapter 7

Future Work

7.1 Introduction

In the making of this thesis, we came across a lot of divergent ideas that would have been interesting to pursue and which would have added value to our work, but that we simply didn't have the time to pursue further. Our implementations do also still have weaknesses which we have covered and are not suitable to be used in production. In this chapter we'll therefore present some of the most important and interesting further work and research that we see as possible continuations to our thesis.

7.2 A full-scale program implementation

The first step to further developing our program would be to actually try these methods in a full scale program, that is significantly larger than our proof-of-concept implementation. Running similar tests to the ones we've used here would be our suggestion, to ensure that the methods do scale the way we've predicted in this paper. We'd also like to see our proof-of-concept made into an external library to be universally available, but this should be a trivial task as our code was developed with such a goal in mind. This could then be further used to develop this idea into a functional product to be used for adding dynamic updates to programs.

7.3 Improved state saving

One of the things we felt Kitsune did very well in their library, was to create a generalised way of saving and initialising states. Their trade-off in this case is that they require heavy analysis using C intermediate language when compiling and running positionally independent code, which requires more memory. Our two methods which we have developed are more elementary in a sense, they require more work to properly function but in turn are faster and require less resources. As it stands now, to be able to serialise data properly, the programmer implementing our methods must make use of their own created data structures to save the desired state.

An automated state saving function would thus save the programmer a lot of work and the ease of use would greatly improve. What we would like to have developed is a function which takes a data structure and determines the size of the data, serialises it and stores it for future updates. Likewise, the deserialisation would function in a similar way, restoring the serialised data into the proper structure for the updated application. The future work, would here lie in researching the possibilities for making such a functionality work and what kind of alternatives that would be available, mostly could it be done in a better way than Kitsune is doing it at the moment, i.e. with a smaller library and less memory usage.

Another section of restoring state is the ability to return to the point in the code from which the update occurred. The current way of doing so involved a lot of if-cases and jumping between methods to restore all active variables, for both Kitsune and our implementations. It would therefore likely be very beneficial for the code clarity to have a more well-developed method of returning to an execution point.

7.4 Improved error checking

An important part of dynamic updating is the ability to ensure that any transferred data is correct. Either because the variable could have been misplaced in the data, causing it to be read from the wrong location, or because of a change in the program which makes the old value of the variable incorrect in the current execution point. In the current setup, the programmer would have to manually do a sanity check on each individual variable after restoring them. This is quite a gruesome process which means that many are likely to avoid it. Adding an overall system to deal with these sanity checks, that is simpler to write, use and update than the current manual version could lead to more safely written programs.

7.5 Cross compiling Kitsune

We spent quite some time trying to cross-compile a program which we had implemented dynamic updating on with Kitsune. However, we needed to cross-compile it to a different processor architecture in order to run it on the embedded system we had available to us. This turned out to be problematic, because Kitsune uses multiple other libraries to function. The main issue was in its use of CIL [22], which was complex (but should be possible) to cross-compile together with Kitsune. To make Kitsune viable for embedded systems, developing a way to cross-compile it is essentially a necessity.

Chapter 8

Conclusion

Typical software updates today are slow and cause a loss of state due to the restart required when updating the software. In this thesis we have outlined and developed a proof of concept for two updating methods using systemd and exec and compared them to an existing research project in dynamic software updating called Kitsune. In doing so we have demonstrated that there exists multiple options available to the programmer when developing a dynamically updatable program which have different dependencies and are suitable for projects depending on hardware needs. The update methods we have developed are less dependent on existing libraries but do in return require more work than existing methods to get a working update. Exec is the simplest method of them since it only requires a working Linux environment. The systemd implementation is similar in code size and implementation but requires more work for transferring file descriptors and newer updates of systemd to properly function.

The method which introduces the least downtime of the three has been shown to be exec, and similarly we have also shown that it uses the least memory to perform the update. Kitsune however is still the superior option in user-friendliness and requires the least amount of changes in an existing code base to function properly. However, since Kitsune is dependent on quite large libraries to be properly installed, some of them quite old, if it is to be used for implementations today it would need a significant amount of development time to function correctly. Since Kitsune has a large dependency library it is also not suitable for environments where memory, disk space or CPU time is an issue. Thus, it would be a lot more difficult to use Kitsune for embedded systems. This makes exec would the best choice, since it is also significantly better than systemd with both memory usage and update time. Since the thesis was done in collaboration with Axis, we would like to make the following recommendation: We recommend using the exec method for dynamically updating software, since it introduces minimum downtime and has the lowest memory usage of all the options we've studied.

We'd suggest that the next step be to preferably improve some of the problems we discussed in the future work chapter, rather than straight going into making it a fully functional library. From there it would be to actually use it in some of their software to ensure that the method scales appropriately for the hardware restrictions, before implementing it into all relevant software.

Chapter 9

References

9.1 Main References

[1] M. Hicks, J. T. Moore, and S. Nettles, "Dynamic software updating," *SIGPLAN Not.*, vol. 36, pp. 13–23, May 2001.

[2] E. K. S. M. H. J. S. F. Christopher M. Hayden, Karla Saur, "Kitsune: Efficient, general-purpose dynamic software updating for c," 2012.

[3] M. H. J. S. F. Christopher M. Hayden, Edward K. Smith, "State transfer for clear and efficient runtime upgrades," 2011.

[4] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," June 2006.

[5] K. M. R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," 2009.

[6] "Tedsuto: A general framework for testing dynamic software updates,"

[7] L. Pina, "Practical dynamic software updating," Feb. 2016.

[8] M. H. Luıs Pina, Luıs Veiga, "Rubah: Dsu for java on a stock jvm," 2013.

[9] E. K. Smith, M. Hicks, and J. S. Foster, "Towards standardized benchmarks for dynamic software updating systems," in *Hot Topics in Software Upgrades (HotSWUp), 2012 Fourth Workshop on*, pp. 11–15, IEEE, 2012.

[10] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *Proceedings of the 29th international conference on Software Engineering*, pp. 271–281, IEEE Computer Society, 2007.

[11] *systemd*. https://www.freedesktop.org/wiki/Software/systemd//.

[12] "Safe software updates via multi-version execution.," *2013 35th International Conference on Software Engineering (ICSE), Software Engineering (ICSE), 2013 35th International Conference on*, p. 612, 2013.

9.2 Sub-References

[13] "Axis communcations."

[14] R. Wash, E. Rader, K. Vaniea, and M. Rizor, "Out of the loop: How automated software updates cause unintended security consequences," in *Symposium on Usable Privacy and Security (SOUPS)*, pp. 89–104, 2014.

[15] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 612–621, IEEE Press, 2013.

[16] K. E. Vaniea, E. Rader, and R. Wash, "Betrayed by updates: how negative experiences affect future security," in *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pp. 2671–2674, ACM, 2014.

[17] D. Fisk, "Programming with punched cards," 2005.

[18] J. Spolsky, "Top five (wrong) reasons you don't have testers,"

[19] *Unix(7)*. http://man7.org/linux/man-pages/man7/unix.7.html.

[20] *Exec*. http://pubs.opengroup.org/onlinepubs/009695399/functions/exec.html.

[21] B. W. Kernighan and D. M. Ritchie, "The c programming language," 1988.

[22] G. Kerneis, *C Intermediate Language*. https://github.com/cil-project/cil.

Appendix A

Appendix

A.1 systemd implementation

```
#ifndef STATE_H
#define STATE_H

 /**
 * Checks the currently executing file's last modified time, and sets it
 * default time to compare to when running is_file_modified();
 */
void init();

/**
 * Checks if the currently executing file has been modified since init
 * was called.
 *
 * return: 1 if the file has been modified, 0 otherwise
 */
int is_file_modified();

/**
 * Saves n number of bytes from state into a shared memory. The returned
 * is then passed along to PID 1.
 *
 * state: A pointer to the state to be saved.
 * n: The number of bytes to be saved.
 *
 * return: Returns 0 on success, else returns a non-zero value.
 */
int save_state(const void * state, const int n, const char* memarea);

/**
 * Restores the state which was previously saved into shared memory. Only
 * if called by the same binary that initially saved the state. The state
 * saved into s.
 *
 * s: The pointer to a location in memory to save the state to.
 * len: The number of bytes to restore into s.
 *
 * return: Returns 1 on success, 0 if there is no state to be restored, a
 */
int restore_state(void *s, int len, const char* memarea);

/**
 * Saves a socket by passing it to PID 1. Can take at most 10 sockets at
 * defined by the settings in the .service file that runs the program.
 *
 * fd: The fd to pass to PID 1.
 *
 * return: Returns 1 on success, -1 on failure.
 */



/*
*  Executes the update function, takes path and then a maximum of 32 argu
*
*  Last argument should be NULL to let the function know that no more arg
*
*  path: the updated file which should be executed
*
*  return: -1 on failure otherwise a nonnegative integer.
*/
int exec_update(const char* path, char* arg1, ...);

/**
 * Saves a socket by passing it to PID 1. Can take at most 10 sockets at
 * defined by the settings in the .service file that runs the program.
 *
 * fd: The fd to pass to PID 1.
 *
 * return: Returns 1 on success, -1 on failure.
 */

int save_socket(int fd);

/**
 * Retrieves the socket from PID 1 that was passed via save_socket. The p
 * sockets will be ordered from 0-9 starting from id 0.
 *
 * id: The id of the socket to be returned.
 *
 * return: Returns the fd corresponding to the id passed in. If there was
 * corresponding to the id, -1 will be returned instead.
 */
 int retrieve_socket(int id);
#endif
```

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>
#include <sys/stat.h>
#include <systemd/sd-daemon.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "state.h"

#define MAXARGS 32

static char *path = NULL;
static struct timespec orig_mod_time;

static char * get_path() {
        if (path) {
                //printf("This is path: %s\n", path);
                //syslog(5, "This is path: %s", path);
                return strdup(path);
        }
        size_t max_path_len = 100;
        char *ret = calloc(sizeof(char),max_path_len);
        readlink("/proc/self/exe", ret, max_path_len);
        path = strdup(ret);
        //printf("This is p_ret: %s\n", ret);
        //syslog(5, "This is p_ret: %s", ret);
        return ret;
}

void init() {
        char *fp = get_path();
        struct stat orig_stat;
        stat(fp, &orig_stat);
        orig_mod_time = orig_stat.st_mtim;
        free(fp);
}

static char * get_statepath() {
        size_t max_path_len = 100;
        char *ret = malloc(sizeof(char) * (strlen("/tmp/") + max_path_len
        ret[0] = '\0';
        ret = strcat(ret, "/tmp/");
        char *buf = ret + strlen(ret);
        char *tmp = get_path();
        strcpy(buf, tmp);
        free(tmp);
        for (int i = 0; i < strlen(buf); ++i) {
                if (buf[i] == '/') {
                        buf[i] = '.';
                }
        }
        //printf("This is ret: %s\n", ret);
        //syslog(5, "This is ret: %s", ret);
        return ret;
}

int is_file_modified() {
        struct stat curr_stat;
        char *fp = get_path();
        stat(fp, &curr_stat);
        struct timespec curr = curr_stat.st_mtim;
        free(fp);
        //print("osec: %ld - csec: %ld\nonsec: %ld - cnsec: %ld\n", orig.
        return curr.tv_sec != orig_mod_time.tv_sec
                || curr.tv_nsec != orig_mod_time.tv_nsec;
}

/*Saves state into memory, on sucess returns a non-negative integer other
int save_state(const void *state, const int len, const char* memarea)
{
        /*Open a shared memory object*/
        int fd = shm_open(memarea, O_RDWR | O_CREAT | O_TRUNC, 0666);
        if(fd < 0 ) {
                syslog(5, "shm_open call failed\n");
                return -1;
        }

        /*Truncate the shared memory object*/
        if(ftruncate(fd, len) == -1) {
                syslog(5, "ftruncate call failed\n");
                return -1;
        }

        /*Write to state into the shared memory object*/
        int wb = write(fd, state, len);
        if(wb != len) {
                syslog(5, "Failed to write %d bytes, wrote %d instead\n",
                return -1;
        }

        /*Map the shared memory object */
        void *ptr = mmap(0, len, PROT_WRITE | PROT_READ, MAP_SHARED, fd,
        if(!ptr) {
                syslog(5, "mmap call failed\n");
                return -1;
        }

        if( close(fd) < 0) {
                syslog(5, "close fd failed\n");
        }

        return 0;
}
/* Restore state from memory */
int restore_state(void *s, const int len, const char* memarea) {
        /* Open up the shared memory area */
        //shm_unlink(memarea);
        int fd = shm_open(memarea, O_RDONLY, 0666);
        if(fd < 0 ) {
                syslog(5, "shm_open call failed, given area does not exis
                return 0;
        }

        void *ptr = mmap(0, len, PROT_READ, MAP_SHARED, fd, 0);
        if(!ptr) {
                syslog(5, "mmap call failed\n");
                return -1;
        }
        /* Read the state from shared memory */
        int rb;
        if (rb = read(fd, s, len) != len) {
                syslog(5, "Failed to write %d bytes, wrote %d instead\n",
                return -1;
        }

        if(close(fd) < 0)
                syslog(5, "Close failed\n");

        if(shm_unlink(memarea) < 0 )
                syslog(5, "shm_unlink failed\n");

        return 1;
}

int exec_update(const char *path, char* arg1, ...) {
        int nargs = 0;
        va_list ap;
        char *argv[MAXARGS+1];

        va_start(ap, arg1);
        while(arg1 != 0 && nargs < MAXARGS) {
                argv[nargs++] = arg1;
                arg1 = va_arg(ap, char *);
        }
        argv[nargs] = (char*) 0;
        va_end(ap);

        return execv(path, argv);
}

int save_socket(int fd)
{
        static int count = 0;
        sd_pid_notify(0,0,"READY=1");
        syslog(5, "fd is %d", fd);
        int err = sd_pid_notify_with_fds(0, 0, "FDSTORE=1\n", &fd, 1);
        if (err < 1) return err; //TODO doesn't correspond to .h file
        return 1;
}

int retrieve_socket(int id)
{
        int n = sd_listen_fds(0);
        if (n < id)
                return -1;
        return SD_LISTEN_FDS_START + id;
}
```

```
#include <time.h>
#include <netinet/in.h>
#include <syslog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <signal.h>

#include "state.h"

#define on_error(...) { fprintf(stderr, __VA_ARGS__); fflush(stderr); sys
#define print(...) { printf(__VA_ARGS__); fflush(stdout); syslog(5, __VA_

struct state {
        char response;
        struct timespec last_sent_time;
};

static char response[] = { 'A' };

/*
 * Initiates a socket on the passed port.
 *
 * port: The port to initiate the socket
 *
 * return: Returns the fd of the socket
 */
int init_listen_socket(int port) {
        int r, n, opt_val = 1;
        struct sockaddr_in server;
        int server_fd, err;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd < 0) on_error("Could not create socket\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(

        err = bind(server_fd, (struct sockaddr *) &server, sizeof(server)
        if (err < 0) on_error("Could not bind socket\n");
        err = listen(server_fd, 128);
        return server_fd;
}
```

```
int main (int argc, char *argv[]) {
        struct state s;
        struct sockaddr_in client;
        int server_fd = 0, client_fd = 0, err;
        socklen_t client_len = sizeof(client);
        struct timespec last_sent_time;
        struct timespec state_start, state_end;

        init(); // Call state.c's init() function, in order to later ider
        // an update is available.

        switch (restore_state(&s, sizeof(s), "mem")) {
                default:
                        print("Failed to restore an existing state, runni
                case 0:
                        if (argc == 1) {
                                server_fd = init_listen_socket(5000);
                        } else if (argc == 2) {
                                server_fd = init_listen_socket(atoi(argv[
                        } else {
                                on_error("Usage: server [port]");
                        }
                        break;
                case 1:
                        //print("Restoring state\n");
                        response[0] = s.response;
                        last_sent_time = s.last_sent_time;
                        client_fd = retrieve_socket(0);
                        server_fd = server_fd + 1;
                        //print("After restore - server_fd: %d, client_fd
                        goto restore;
        }
        if (server_fd < 0) on_error("Failed to open server_fd with errno
        while (1) {
                client_fd = accept(server_fd, (struct sockaddr *) &client
                if (client_fd == -1) on_error("Failed to open client_fd w
                //print("Accepted a client connection");
                while (1) {
                        clock_gettime(CLOCK_MONOTONIC, &last_sent_time);
                        err = send(client_fd, response, 1, 0);
                        err = send(client_fd, response, 1, 0);
                        if (++response[0] > 'Z')
                                response[0] = 'A';
                        if (err < 0) {
                                print("Send failed, closing socket\n");
                                response[0] = 'A';
                                break;
                        }
                        struct timespec rem;

                        rem.tv_sec = 0;
                        rem.tv_nsec = 1000000000 - last_sent_time.tv_nsec
                        if (is_file_modified()){
                                clock_gettime(CLOCK_MONOTONIC, &state_sta
                                //print("File is modified, saving state a
                                s.response = response[0];
                                s.last_sent_time = last_sent_time;
                                if (save_state(&s, sizeof(struct state),
                                        on_error("Failed to save state\n"
                                //print("State saved");
                                if ((err = save_socket(server_fd)) < 1)
                                        on_error("Failed to save server_f
                                if ((err = save_socket(client_fd)) < 1)
                                        on_error("Failed to save client_f
                                //print("Sockets saved successfully\n");
                                clock_gettime(CLOCK_MONOTONIC, &state_end
                                print("Save state took: %ld s, %ld ns \n"
                                raise(SIGKILL); //exit kills the socket,
                        }
                        nanosleep(&rem, NULL);
                }
                close(client_fd);
                print("Closed socket connection with client");
        }
}

#include <time.h>
#include <netinet/in.h>
#include <syslog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <signal.h>

#include "state.h"

#define on_error(...) { fprintf(stderr, __VA_ARGS__); fflush(stderr); sys
#define print(...) { printf(__VA_ARGS__); fflush(stdout); syslog(5, __VA_

struct state {
        char response;
        struct timespec last_sent_time;
};
```

```
static char response[] = { 'A' };

/*
 * Initiates a socket on the passed port.
 *
 * port: The port to initiate the socket
 *
 * return: Returns the fd of the socket
 */
int init_listen_socket(int port) {
        int r, n, opt_val = 1;
        struct sockaddr_in server;
        int server_fd, err;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd < 0) on_error("Could not create socket\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(

        err = bind(server_fd, (struct sockaddr *) &server, sizeof(server)
        if (err < 0) on_error("Could not bind socket\n");
        err = listen(server_fd, 128);
        return server_fd;
}

int main (int argc, char *argv[]) {
        struct state s;
        struct sockaddr_in client;
        int server_fd = 0, client_fd = 0, err;
        socklen_t client_len = sizeof(client);
        struct timespec last_sent_time;
        struct timespec state_start, state_end;
        init(); // Call state.c's init() function, in order to later ider
        // an update is available.

        clock_gettime(CLOCK_MONOTONIC, &state_start);
        switch (restore_state(&s, sizeof(s),"mem")) {
                default:
                        print("Failed to restore an existing state, runni
                case 0:
                        if (argc == 1) {
                                server_fd = init_listen_socket(5000);
                        } else if (argc == 2) {
                                server_fd = init_listen_socket(atoi(argv[
                        } else {
                                on_error("Usage: server [port]");
                        }
                        break;
                case 1:
                        print("Restoring state\n");
                        response[0] = s.response;
                        last_sent_time = s.last_sent_time;
                        client_fd = retrieve_socket(0);
                        server_fd = server_fd + 1;
                        clock_gettime(CLOCK_MONOTONIC, &state_end);
                        print("Restore state took: %ld s & %ld ns \n", st
                        //              print("After restore - server_fd:
                        goto restore;
        }
        if (server_fd < 0) on_error("Failed to open server_fd with errno
        while (1) {
                client_fd = accept(server_fd, (struct sockaddr *) &client
                if (client_fd == -1) on_error("Failed to open client_fd w
                //print("Accepted a client connection");
                while (1) {
                        clock_gettime(CLOCK_MONOTONIC, &last_sent_time);
                        err = send(client_fd, response, 1, 0);
                        if (++response[0] > 'Z')
                                response[0] = 'A';
                        if (err < 0) {
                                print("Send failed, closing socket\n");
                                response[0] = 'A';
                                break;
                        }
                        struct timespec rem;
restore:
                        rem.tv_sec = 0;
                        rem.tv_nsec = 1000000000 - last_sent_time.tv_nsec
                        if (is_file_modified()){
                                //print("File is modified, saving state a
                                s.response = response[0];
                                s.last_sent_time = last_sent_time;
                                if (save_state(&s, sizeof(struct state),"
                                        on_error("Failed to save state\n'
                                print("State saved");
                                if ((err = save_socket(server_fd)) < 1)
                                        on_error("Failed to save server_f
                                if ((err = save_socket(client_fd)) < 1)
                                        on_error("Failed to save client_f
                                //print("Sockets saved successfully\n");
                                raise(SIGKILL); //exit kills the socket,
                        }
                nanosleep(&rem, NULL);
            }
            close(client_fd);
            print("Closed socket connection with client");
        }
        return 0;
}
```

A.2 Kitsune implementation

```
#include <time.h>
#include <netinet/in.h>
#include <syslog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <kitsune.h>

#define on_error(...) { fprintf(stderr, __VA_ARGS__); fflush(stderr); sys
#define print(...) { printf(__VA_ARGS__); fflush(stdout); syslog(5, __VA_

static char response[] = { 'A' };

/*
 * Initiates a socket on the passed port.
 *
 * port: The port to initiate the socket
 *
 * return: Returns the fd of the socket
 */
int init_listen_socket(int port) {
        int r, n, opt_val = 1;
        struct sockaddr_in server;
        int server_fd, err;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd < 0) on_error("Could not create socket\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(

        err = bind(server_fd, (struct sockaddr *) &server, sizeof(server)
        if (err < 0) on_error("Could not bind socket\n");
        err = listen(server_fd, 128);
        return server_fd;
}

int main (int argc, char *argv[]) E_NOTELOCALS {
        struct sockaddr_in client;
        int server_fd = 0, client_fd = 0, err;
        socklen_t client_len = sizeof(client);
        struct timespec last_sent_time;
        kitsune_do_automigrate();
        MIGRATE_LOCAL(server_fd);
        MIGRATE_LOCAL(client_fd);
        MIGRATE_LOCAL(last_sent_time);

        if(!kitsune_is_updating()) {
                if (argc == 1) {
                        server_fd = init_listen_socket(5000);
                } else if (argc == 2) {
                        server_fd = init_listen_socket(atoi(argv[1]));
                } else {
                        on_error("Usage: server [port]");
                }
        } else if(kitsune_is_updating_from("send")) {
                goto restore;
        }
        if (server_fd < 0) on_error("Failed to open server_fd with errno
        while (1) {
                client_fd = accept(server_fd, (struct sockaddr *) &client
                if (client_fd == -1) on_error("Failed to open client_fd w
                print("Accepted a client connection");
                while (1) {
                        clock_gettime(CLOCK_MONOTONIC, &last_sent_time);
                        err = send(client_fd, response, 1, 0);
                        err = send(client_fd, response, 1, 0);
                        if (++response[0] > 'Z')
                                response[0] = 'A';
                        if (err < 0) {
                                print("Send failed, closing socket\n");
                                response[0] = 'A';
                                break;
                        }
                        struct timespec rem;
                        struct timespec before_update;
restore:
                        clock_gettime(CLOCK_MONOTONIC, &before_update);
                        rem.tv_sec = 0;

                        rem.tv_nsec = 1000000000 - last_sent_time.tv_nsec
                        syslog(5, "The time before update is: %ld s & %ld
                                        before_update.tv_sec,
before_update.tv_nsec);
                        kitsune_update("send");
                        nanosleep(&rem, NULL);
                }
                close(client_fd);
                print("Closed socket connection with client");
        }
        return 0;
}
```

```
#include <time.h>
#include <netinet/in.h>
#include <syslog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <kitsune.h>

#define on_error(...) { fprintf(stderr, __VA_ARGS__); fflush(stderr); sys
#define print(...) { printf(__VA_ARGS__); fflush(stdout); syslog(5, __VA_

static char response[] = { 'A' };

/*
 * Initiates a socket on the passed port.
 *
 * port: The port to initiate the socket
 *
 * return: Returns the fd of the socket
 */
int init_listen_socket(int port) {
        int r, n, opt_val = 1;
        struct sockaddr_in server;
        int server_fd, err;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd < 0) on_error("Could not create socket\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(

        err = bind(server_fd, (struct sockaddr *) &server, sizeof(server)
        if (err < 0) on_error("Could not bind socket\n");
        err = listen(server_fd, 128);
        return server_fd;
}

int main (int argc, char *argv[]) E_NOTELOCALS {
        struct sockaddr_in client;
        int server_fd = 0, client_fd = 0, err;
        socklen_t client_len = sizeof(client);
        struct timespec last_sent_time;

        kitsune_do_automigrate();
        MIGRATE_LOCAL(server_fd);
        MIGRATE_LOCAL(client_fd);
        MIGRATE_LOCAL(last_sent_time);

        if(!kitsune_is_updating()) {
                if (argc == 1) {
                        server_fd = init_listen_socket(5000);
                } else if (argc == 2) {
                        server_fd = init_listen_socket(atoi(argv[1]));
                } else {
                        on_error("Usage: server [port]");
                }
        } else if(kitsune_is_updating_from("send")) {
                goto restore;
        }
        if (server_fd < 0) on_error("Failed to open server_fd with errno
        while (1) {
                client_fd = accept(server_fd, (struct sockaddr *) &client
                if (client_fd == -1) on_error("Failed to open client_fd
                print("Accepted a client connection");
                while (1) {
                        clock_gettime(CLOCK_MONOTONIC, &last_sent_time);
                        err = send(client_fd, response, 1, 0);
                        if (++response[0] > 'Z')
                                response[0] = 'A';
                        if (err < 0) {
                                print("Send failed, closing socket\n");
                                response[0] = 'A';
                                break;
                        }
                        struct timespec rem;
                        struct timespec before_update;
restore:
                        rem.tv_sec = 0;
                        kitsune_update("send");
                        rem.tv_nsec = 1000000000 - last_sent_time.tv_nsec
                        clock_gettime(CLOCK_MONOTONIC, &before_update);
                        syslog(5, "The time after update is: %ld s & %ld
                                        before_update.tv_sec,
before_update.tv_nsec);
                        nanosleep(&rem, NULL);
                }
                close(client_fd);
                print("Closed socket connection with client");
        }
        return 0;
}
```

A.3 Exec implementation

```
#ifndef STATE_H
#define STATE_H

 /**
 * Checks the currently executing file's last modified time, and sets it
 * default time to compare to when running is_file_modified();
 */
void init();

/**
 * Checks if the currently executing file has been modified since init
 * was called.
 *
 * return: 1 if the file has been modified, 0 otherwise
 */
int is_file_modified();

/**
 * Saves n number of bytes from state into a shared memory. The returned
 * is then passed along to PID 1.
 *
 * state: A pointer to the state to be saved.
 * n: The number of bytes to be saved.
 *
 * return: Returns 0 on success, else returns a non-zero value.
 */
int save_state(const void * state, const int n, const char* memarea);

/**
 * Restores the state which was previously saved into shared memory. Only
 * if called by the same binary that initially saved the state. The state
 * saved into s.
 *
 * s: The pointer to a location in memory to save the state to.
 * len: The number of bytes to restore into s.
 *
 * return: Returns 1 on success, 0 if there is no state to be restored, a
 */
int restore_state(void *s, int len, const char* memarea);

/**
 * Saves a socket by passing it to PID 1. Can take at most 10 sockets at
 * defined by the settings in the .service file that runs the program.
 *
 * fd: The fd to pass to PID 1.
 *
 * return: Returns 1 on success, -1 on failure.
 */


/*
*  Executes the update function, takes path and then a maximum of 32 argu
*
*  Last argument should be NULL to let the function know that no more arg
*
*  path: the updated file which should be executed
*
*  return: -1 on failure otherwise a nonnegative integer.
*/
int exec_update(const char* path, char* arg1, ...);

#endif
```

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <unistd.h>
#include <syslog.h>
#include <string.h>
#include <sys/stat.h>
#include <systemd/sd-daemon.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "state.h"

#define MAXARGS 32

static char *path = NULL;
static struct timespec orig_mod_time;

static char * get_path() {
        if (path) {
                //printf("This is path: %s\n", path);
                //syslog(5, "This is path: %s", path);
                return strdup(path);
        }
        size_t max_path_len = 100;
        char *ret = calloc(sizeof(char),max_path_len);
        readlink("/proc/self/exe", ret, max_path_len);
        path = strdup(ret);
        //printf("This is p_ret: %s\n", ret);
        //syslog(5, "This is p_ret: %s", ret);
        return ret;
}

void init() {
        char *fp = get_path();
        struct stat orig_stat;
        stat(fp, &orig_stat);
        orig_mod_time = orig_stat.st_mtim;
        free(fp);
}

static char * get_statepath() {
        size_t max_path_len = 100;
        char *ret = malloc(sizeof(char) * (strlen("/tmp/") + max_path_len
        ret[0] = '\0';
        ret = strcat(ret, "/tmp/");
        char *buf = ret + strlen(ret);
        char *tmp = get_path();
        strcpy(buf, tmp);
        free(tmp);
        for (int i = 0; i < strlen(buf); ++i) {
                if (buf[i] == '/') {
                        buf[i] = '.';
                }
        }
        //printf("This is ret: %s\n", ret);
        //syslog(5, "This is ret: %s", ret);
        return ret;
}

int is_file_modified() {
        struct stat curr_stat;
        char *fp = get_path();
        stat(fp, &curr_stat);
        struct timespec curr = curr_stat.st_mtim;
        free(fp);
        //print("osec: %ld - csec: %ld\nonsec: %ld - cnsec: %ld\n", orig.
        return curr.tv_sec != orig_mod_time.tv_sec
                || curr.tv_nsec != orig_mod_time.tv_nsec;
}

/*Saves state into memory, on sucess returns a non-negative integer other
int save_state(const void *state, const int len, const char* memarea)
{
        /*Open a shared memory object*/
        int fd = shm_open(memarea, O_RDWR | O_CREAT | O_TRUNC, 0666);
        if(fd < 0 ) {
                syslog(5, "shm_open call failed\n");
                return -1;
        }

        /*Truncate the shared memory object*/
        if(ftruncate(fd, len) == -1) {
                syslog(5, "ftruncate call failed\n");
                return -1;
        }

        /*Write to state into the shared memory object*/
        int wb = write(fd, state, len);
        if(wb != len) {
                syslog(5, "Failed to write %d bytes, wrote %d instead\n",
                return -1;
        }

        /*Map the shared memory object */
        void *ptr = mmap(0, len, PROT_WRITE | PROT_READ, MAP_SHARED, fd,
        if(!ptr) {
                syslog(5, "mmap call failed\n");
                return -1;
        }

        if( close(fd) < 0) {
                syslog(5, "close fd failed\n");
        }

        return 0;
}
/* Restore state from memory */
int restore_state(void *s, const int len, const char* memarea) {
        /* Open up the shared memory area */
        //shm_unlink(memarea);
        int fd = shm_open(memarea, O_RDONLY, 0666);
        if(fd < 0 ) {
                syslog(5, "shm_open call failed, given area does not exis
                return 0;
        }

        void *ptr = mmap(0, len, PROT_READ, MAP_SHARED, fd, 0);
        if(!ptr) {
                syslog(5, "mmap call failed\n");
                return -1;
        }
        /* Read the state from shared memory */
        int rb;
        if (rb = read(fd, s, len) != len) {
                syslog(5, "Failed to write %d bytes, wrote %d instead\n",
                return -1;
        }

        if(close(fd) < 0)
                syslog(5, "Close failed\n");

        if(shm_unlink(memarea) < 0 )
                syslog(5, "shm_unlink failed\n");

        return 1;
}

int exec_update(const char *path, char* arg1, ...) {
        int nargs = 0;
        va_list ap;
        char *argv[MAXARGS+1];

        va_start(ap, arg1);
        while(arg1 != 0 && nargs < MAXARGS) {
                argv[nargs++] = arg1;
                arg1 = va_arg(ap, char *);
        }
        argv[nargs] = (char*) 0;
        va_end(ap);

        return execv(path, argv);
}
```

```
#include <time.h>
#include <netinet/in.h>
#include <syslog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <signal.h>

#include "state.h"

#define on_error(...) { fprintf(stderr, __VA_ARGS__); fflush(stderr); sys
#define print(...) { printf(__VA_ARGS__); fflush(stdout); syslog(5, __VA_

struct state {
        char response;
        struct timespec last_sent_time;
};

static char response[] = { 'A' };

/*
 * Initiates a socket on the passed port.
 *
 * port: The port to initiate the socket
 *
 * return: Returns the fd of the socket
 */
int init_listen_socket(int port) {
        int r, n, opt_val = 1;
        struct sockaddr_in server;
        int server_fd, err;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd < 0) on_error("Could not create socket\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(

        err = bind(server_fd, (struct sockaddr *) &server, sizeof(server)
        if (err < 0) on_error("Could not bind socket\n");
        err = listen(server_fd, 128);
        return server_fd;
}
```

```
int main (int argc, char *argv[]) {
        struct state s;
        struct sockaddr_in client;
        int server_fd = 0, client_fd = 0, err;
        socklen_t client_len = sizeof(client);
        struct timespec last_sent_time;

        init(); // Call state.c's init() function, in order to later iden
        // an update is available.

        switch (restore_state(&s, sizeof(s), "mem")) {
                default:
                        print("Failed to restore an existing state, runni
                        break;
                case 0:
                        server_fd = init_listen_socket(5000);
                        break;
                case 1:
                        response[0] = s.response;
                        last_sent_time = s.last_sent_time;
                        server_fd = strtol(argv[0], NULL, 20);
                        client_fd = strtol(argv[1], NULL, 20);
                        goto restore;
        }
        if (server_fd < 0) on_error("Failed to open server_fd with errno
        while (1) {
                client_fd = accept(server_fd, (struct sockaddr *) &client
                if (client_fd == -1) on_error("Failed to open client_fd 
                while (1) {
                        clock_gettime(CLOCK_MONOTONIC, &last_sent_time);
                        err = send(client_fd, response, 1, 0);
                        err = send(client_fd, response, 1, 0);
                        if (++response[0] > 'Z')
                                response[0] = 'A';
                        if (err < 0) {
                                print("Send failed, closing socket\n");
                                response[0] = 'A';
                                break;
                        }
                        struct timespec rem;
restore:
                        rem.tv_sec = 0;
                        rem.tv_nsec = 1000000000 - last_sent_time.tv_nsec
                        struct timespec file_mod_start, file_mod_end;
                        if (is_file_modified()){
                                clock_gettime(CLOCK_MONOTONIC, &file_mod_
                                s.response = response[0];
                                s.last_sent_time = last_sent_time;
                                if (save_state(&s, sizeof(struct state),
                                        on_error("Failed to save state\n'
                                char arg1[20], arg2[20];
                                sprintf(arg1, "%d", server_fd);
                                sprintf(arg2, "%d", client_fd);

                                clock_gettime(CLOCK_MONOTONIC, &file_mod_
                                printf("Save state took: %ld s & %ld ns \
                                int exec = exec_update("/home/gustavek/E
                                if(exec == -1)
                                        exit(0);

                        }
                        nanosleep(&rem, NULL);
                }
                close(client_fd);
                print("Closed socket connection with client");
        }
        return 0;
}
```

```
#include <time.h>
#include <netinet/in.h>
#include <syslog.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <signal.h>

#include "state.h"

#define on_error(...) { fprintf(stderr, __VA_ARGS__); fflush(stderr); sys
#define print(...) { printf(__VA_ARGS__); fflush(stdout); syslog(5, __VA_

struct state {
        char response;
        struct timespec last_sent_time;
};

static char response[] = { 'A' };

/*
 * Initiates a socket on the passed port.
 *
 * port: The port to initiate the socket
 *
 * return: Returns the fd of the socket
 */
int init_listen_socket(int port) {
        int r, n, opt_val = 1;
        struct sockaddr_in server;
        int server_fd, err;

        server_fd = socket(AF_INET, SOCK_STREAM, 0);
        if (server_fd < 0) on_error("Could not create socket\n");

        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof(

        err = bind(server_fd, (struct sockaddr *) &server, sizeof(server)
        if (err < 0) on_error("Could not bind socket\n");
        err = listen(server_fd, 128);
        return server_fd;
}
```

```
int main (int argc, char *argv[]) {
        struct state s;
        struct sockaddr_in client;
        int server_fd = 0, client_fd = 0, err;
        socklen_t client_len = sizeof(client);
        struct timespec last_sent_time;
        struct timespec file_mod_start, file_mod_end;
        init(); // Call state.c's init() function, in order to later iden
        // an update is available.
        clock_gettime(CLOCK_MONOTONIC, &file_mod_start);
        switch (restore_state(&s, sizeof(s), "mem")) {
                default:
                        print("Failed to restore an existing state, runni
                        break;
                case 0:
                        server_fd = init_listen_socket(5000);
                        break;
                case 1:
                        clock_gettime(CLOCK_MONOTONIC, &file_mod_end);
                        printf("Restore state took: %ld s & %ld ns \n",
                                        file_mod_end.tv_sec-file_mod_star
                                        file_mod_end.tv_nsec-file_mod_sta
                        response[0] = s.response;
                        last_sent_time = s.last_sent_time;
                        server_fd = strtol(argv[0], NULL, 20);
                        client_fd = strtol(argv[1], NULL, 20);
                        goto restore;
        }
        if (server_fd < 0) on_error("Failed to open server_fd with errno
        while (1) {
                client_fd = accept(server_fd, (struct sockaddr *) &client
                if (client_fd == -1) on_error("Failed to open client_fd w
                while (1) {
                        clock_gettime(CLOCK_MONOTONIC, &last_sent_time);
                        err = send(client_fd, response, 1, 0);
                        if (++response[0] > 'Z')
                                response[0] = 'A';
                        if (err < 0) {
                                print("Send failed, closing socket\n");
                                response[0] = 'A';
                                break;
                        }
                        struct timespec rem;
restore:
                        rem.tv_sec = 0;
                        rem.tv_nsec = 1000000000 - last_sent_time.tv_nsec
                        if (is_file_modified()){
                                s.response = response[0];
                                s.last_sent_time = last_sent_time;
                                if (save_state(&s, sizeof(struct state),
                                        on_error("Failed to save state\n"
                                char arg1[20], arg2[20];
                                sprintf(arg1, "%d", server_fd);
                                sprintf(arg2, "%d", client_fd);
                                printf("%s, %s\n", arg1, arg2);
                                int exec = exec_update("/home/gustavek/E
                                if(exec = -1)
                                        exit(0);

                        }
                        nanosleep(&rem, NULL);
                }
                close(client_fd);
                print("Closed socket connection with client");
        }
        return 0;
}
```