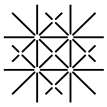


University of Basel

Managing and Distributing Software Updates Using Append-Only Logs

Bachelor's Thesis

Natural Science Faculty of the University of Basel
Departement of Mathematics and Computer Science
Computer Networks Research Group
cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: MSc. Fabrizio Parrillo

Maximilian Barth
m.barth@unibas.ch
19-051-770

20.06.2022

Acknowledgments

This endeavor would not have been possible without Prof. Dr. Christian Tschudin, who always pushed me in the right direction and gave insightful feedback on the current state of the project during our biweekly meetings. Special thanks also go to Dr. Erick Lavoie, who gave a presentation that inspired me to apply for a bachelor's thesis with the computer networks group and helped find the subject of this project. I am also very grateful to Simon Laube, who worked on a related thesis and always was available for developing and discussing new ideas. Further, I would like to extend my sincere thanks to Fabrizio Parrillo, who gave helpful feedback during the writing process of the project and helped me to properly structure this thesis. Finally I would like to thank my friends and family, who supported me during this project.

Abstract

Distributing and managing software updates across a solar-powered sensor network poses several challenges, such as devices losing power and potentially missing updates. Furthermore, sensors communicate via LoRa, which benefits from updates being encoded as compactly as possible.
In order to address these problems, a versioning system that allows developers to create file-specific updates is introduced. Further features of this system are update reversions and the ability to create different update-branches, similar to Git. Said versioning system is built on top of *tinyssb*, an append-only log protocol, which allows for the validation of every update's integrity and authenticity, preventing potentially malicious updates.
With the goal of reducing update sizes, changes in between two versions of a file are expressed as a set of insert and delete operations. These are computed using the longest common subsequence (LCS) of both files. Additionally, so-called *emergency updates* are introduced, which allow devices to ignore faulty updates even though they cannot be removed from the append-only log.
The implementation of the versioning system described above was tested both in UNIX and a small sensor network. Whilst it performed well in the former scenario, some device-specific compatibility issues occurred on the used microcontrollers, which have to be ironed-out in the future.

Table of Contents

1 Introduction

Updates are a vital part of any software system, allowing developers to fix bugs, improve performance and add new features. However, deploying a new version of a program can be tricky when dealing with a solar-powered sensor network: Devices may be difficult to retrieve and located far away from each other, therefore having to be painstakingly collected and updated by hand. The use of over-the-air-updates (OTA) does not solve these issues, since nodes are driven by solar power and cannot be guaranteed to be online when the update is broadcasted. This might cause sensors to miss patches and new features, which could lead to unexpected behavior or even security issues.

The goal of this project was to address these problems and contribute to creating a foundation for the future development of a solar-powered sensor network. For this, a versioning system was designed and implemented, which enables developers to create file-specific updates and manage the current version of files across a *tinyssb* network. The latter is an append-only log protocol, which allows nodes of the network to catch up with missed packets and provides message integrity and authenticity guarantees. The idea is that every file containing source code on the device is monitored by the versioning system and can be changed through OTA updates. Since the sensors' storage is limited and communication via *LoRa* has a low data rate, a great effort was put into minimizing the size of updates. For this, a method based on UNIX's `diff` utility was chosen and will be presented in chapter 3.
A further challenge of the project proved to be developing for the limited hardware capabilities of the chosen microcontrollers. These devices would frequently crash due to stack overflows, the fixing of which turned out to consume a considerable chunk of the implementation process.

The following chapter covers the terminology and background information of the project and includes a more detailed explanation of the *tinyssb* protocol. Thereafter, the choice of hardware and the software implementation process are presented in detail. Chapter 5 describes a test scenario and how the implemented system performed in it, both on the used microcontrollers and in UNIX. Finally, the project is reflected upon and possible expansions of the project are discussed.

2
Background

2.1 LoRa

LoRa, short for *long range*, is a network protocol, which is situated in the physical layer of the OSI reference model. Its properties are that it is very energy efficient and allows devices to communicate over large distances, which means 2-5 km in cities and up to 45 km in rural areas [2]. This makes it ideal for the use in this project and is also the reason why it is commonly seen in IoT (internet of things devices). However, long range and energy efficiency also come with a trade-off, specifically a low data rate. This is not an issue in the use case of this project, since the sensors of the network are not required to communicate in real-time.
In order to connect to the internet via a gateway, IoT devices use LoRa in combination with the LoRaWAN transport layer protocol. This is not used in this project, since the sensor network should function independently of the internet. Instead, the choice fell on an append-only-log-based protocol.

2.2 Append-only logs

The nodes of a solar-powered network may lose power at any time and do not have fixed operating hours. Therefore the employed transport layer protocol must have mechanisms that allow a node to catch up with missed messages. This is especially important in the case of updates, where missing a security patch may have fatal consequences. A further requirement for the protocol is that message authenticity and integrity must be verifiable, preventing potentially malicious updates.

In order to address the aforementioned specifications, Prof. Dr. Christian Tschudin conceived an append-only log protocol, named *tinyssb* [8]. As the name suggests, it was inspired by Secure Scuttlebutt [7]. Since nodes of the sensor network communicate via LoRa, tinyssb's goal is to reduce packet sizes to a minimum, whilst still guaranteeing message integrity and authenticity. This is achieved by sending each packet in a context that both the sender and recipient are aware of, allowing for parts of the original message to be dropped, which reduces the amount of bytes that have to be transmitted. The missing chunks of a

packet can then later be inferred by the recipient in order to reconstruct the contents of the original message.

Tinyssb operates with append-only logs, so-called *feeds*. Every feed is initiated by an anchor, which contains information such as its public key. Before a packet is appended to a feed, it is signed using the feed's private key, ensuring that only the owner of the feed can append valid new entries to it. This is done using `ed25519` and can be verified by other nodes in the network using the previously mentioned public key. Therefore only the anchor of a feed has to be trusted in order to sequentially verify the authenticity and integrity of each following entry. To ensure that already appended packets cannot be removed or altered, a hash of each packet's predecessor is added to its contents. The resulting hash chain can be followed from a feed's newest packet back to its (trust) anchor.
With the purpose of extending the aforementioned trust chain from one feed to another, the tinyssb protocol allows for the creation of child and continuation feeds. Since the former are vital for the versioning system that is presented in chapter 3, their concept is briefly explained: A child feed is created by appending a packet containing its public key to the parent feed and adding the hash of said packet together with the parent's public key as the first entry of the child feed. This way, the trust chain is expanded from the parent to the child, since the authenticity and integrity of the first child packet can be traced back to the trust anchor of its parent.
An exact specification of the protocol can be found on the tinyssb Github page:

github.com/tschudin/tinyssb [8].

Each node of a tinyssb network is initialized with at least one self-signed trust anchor. It then periodically requests the next (missing) feed entry of each feed that it is aware of. The sent request contains the sequence number and public key (feed ID) of the desired log entry. When a node receives a request, it checks whether the corresponding entry is available. If so, it is broadcasted to the network. Since the node that sent the initial request is expecting this packet, it is able to infer its sequence number, public key and predecessor's hash value. This context allows packet sizes to be reduced to 128 bytes, which includes a 48 byte payload.

2.3 UNIX's diff utility

An important part of the versioning system, which is the main subject of this thesis, is the file difference algorithm. Simply sending the entire file after each update wastes a lot of memory, since many lines of code are likely to remain unchanged. This is where the file difference algorithm comes in: Its job is to compute the most efficient insert and delete operations that transform the original file into its updated version.
Comparing two versions of a file can be seen as an instance of the famous longest common subsequence (LCS) problem, which can be solved using dynamic programming, resulting in the "overlapping" parts of both files. This LCS string can then be used in order to compute the insert and delete operations necessary to transform one file into another.
The implementation proposed in chapter 3 is based on work by J. W. Hunt and M. D. McIlroy, who worked on UNIX's `diff` utility and presented their findings in a technical

report at Bell Labs [3]. This subject will be explored in more detail during subsection 4.3.2.

2.4 MicroPython

MicroPython is a programming language that is mostly compatible with Python 3 and is optimized for running on microcontrollers. It supports most of Python 3's core libraries and allows programmers to easily interact with the device's hardware. For these reasons it was chosen as the main language for the software implementation.

Because some parts of the project required further optimization steps before properly running on the chosen hardware, MicroPython's `uctypes` module was used. It allows for the declaration of C-like structs and the use of pointers and references. This makes it possible to interpret (Micro)Python `bytearrays` as structs, whereby member variables can easily be accessed and assigned using the (`.`)-operator. A further advantage of this is that the aforementioned `bytearrays` can be read directly from files or network sockets, making the process of creating an instance of a struct very memory efficient. However, this also comes with a trade-off, namely that the `uctypes` module is incompatible with regular Python. Since the target devices run MicroPython and it can also be installed on desktop devices, this consequence was accepted.

3
Methodology

The goal of the project was to implement a working prototype of a versioning system that is compatible with the tinyssb protocol. This system should be reliable and have simple yet effective tools for distributing and managing updates across a tinyssb network. Further, it should consider the energy constraint of a solar-powered network, meaning that some nodes may be offline for longer periods of time. The system's task is to monitor every local file that contains source code and change the contents of said file when it is updated by the administrator. These changes then take effect as soon as the sensor is rebooted, which regularly happens with solar-powered devices. A further feature of the system is that any version of a file can be applied at any time. This allows for reversions of bad updates and was inspired by version control systems like Git.
In order to speed-up development and testing, the code was written for UNIX systems, where UDP multicast is used (instead of LoRa) as a communication channel in between nodes. The implementation was later modified, so it could also be run on low-powered Pycom devices that are LoRa and WiFi capable, whereby the latter is used for hosting a web GUI, which provides access to the versioning system.
The full implementation of this system can be found on the Github page of the project:

github.com/Laellekoenig/ussb

The idea behind this project resulted from multiple meetings with Prof. Dr. Christian Tschudin and Dr. Erick Lavoie, the latter having supervised a related project at the EPFL. Since Raspberry Pi Zero Ws were used in that project, they were also considered for this one at first. However, running a headless distribution of Linux led to relatively slow start-up times during initial testing and seemed to be overkill for a simple solar-powered sensor network. Prof. Dr. Christian Tschudin suggested the use of simpler and more energy-efficient development boards by a company named Pycom, since he himself had already worked with such devices and had made positive experiences. Two different Pycom devices were considered for the use in this project, namely the LoPy4 and the FiPy, both of which are LoRa, WiFi, Bluetooth and Sigfox capable. Since LoPy4s were sold-out at the time, the choice fell on the more expensive FiPy variant (more expensive because it is also LTE capable, unlike the LoPy4).

In order to connect to these devices via USB serial access, they were combined with the Pysense 2.0X board, which is also sold by Pycom. Further features of this board are the availability of a SD card slot, 5 different sensors, such as light and a connector for a possible solar panel, all of which seemed ideal for future expansions of the project. The development board natively runs MicroPython code and comes with I/O libraries that are maintained by Pycom. This made programming LoRa sockets fairly straight-forward. A further advantage of using MicroPython is that it does not have to be compiled, which facilitates the updating process.
A discussion of the implementation process and a detailed explanation of the versioning system are covered in the next chapter.

4

Implementation

The software system can be separated into two parts, the first being the implementation of the tinyssb protocol, which manages feeds, packets and the communication to other nodes. This aspect is not covered in this thesis, since it adheres to the tinyssb protocol. The second part is the versioning system, which makes use of the tinyssb protocol in order to manage updates. This system allows for continuous code deployments but also provides the administrator with version-control-system-like functionalities, such as reverting a file to any previous version and enabling the user to create and work on different branches of a file.
The final software is mainly written in MicroPython and makes use of some MicroPython-specific libraries, making it incompatible with regular Python (more on this in subsection 4.1.1). Some parts of the versioning system, namely the file difference algorithm, are written in JavaScript, since they are executed in the web GUI of the project. Another feature of the code base is that it works both in UNIX and on Pycom devices. This is done using (Micro)Python's `sys.platform` and makes it possible to precompute the initial configuration of a sensor on a desktop device. The resulting files can then be directly loaded onto the FiPy, saving setup-time in the process, since the used `pure25519` encryption is very slow on Pycom devices. At the heart of the implementation is the tinyssb protocol, which will be discussed in the next subsection.

4.1 tinyssb

The tinyssb protocol is the engine of the versioning system. Versions are added and applied through callback functions that are registered to the demultiplexing table and are executed as soon as new log entries are appended. The implementation adheres to the tinyssb protocol version of the third proof-of-concept by Prof. Dr. Christian Tschudin [8]. This includes the creation and management of child and continuation feeds. However, the first adaption ran into multiple problems on Pycom devices:

The first issue was that overwriting bytes of a file behaved differently on Pycom devices than in UNIX. This problem occurred when updating the anchor of a feed, which happens every time a new log entry is added, and would result in the loss of every appended packet. This

was due to the fact that starting from the index of alteration, all following bytes were deleted. Obviously this was unacceptable and had to be fixed. The first solution to this problem was to separate the anchor of a feed into a separate file. This change over the original protocol is still used in the current version, but does not have an affect on the communication with other nodes and therefore is still compatible with other tinyssb implementations. Later on, it was recognized that switching the file system formatting of the FiPy to `FAT FS` also took care of the file editing problem.

The second problem was that Pycom devices would frequently crash due to stack overflows. One of the primary causes of these crashes was the used implementation of `ed25519`. This is because the `pure25519` library is not optimized for devices with limited memory and, besides taking up to 10 Seconds to verify a message, frequently results in stack overflows, due to large recursion depths. Even after some optimizations by Simon Laube [4], this problem was not solved, and it was decided to disable the verification process on Pycom devices until an improved version of the `ed25519` algorithm is available.

Unfortunately, stack overflows still occurred when appending new updates to feeds, which likely resulted from a suboptimal implementation of the tinyssb protocol and versioning system in regards to memory usage.
In order to fix the persisting problems, it was decided to rewrite the project from scratch, using MicroPython specific memory optimizations, the process of which is described in the following subsection.

4.1.1 Optimization steps

With the goal of fixing the aforementioned stack overflows, MicroPython's `uctypes` was used to define structs for feeds and packets. This eliminated the need for equivalent classes and meant that `bytearrays` could be directly read from files or network sockets and used to create the corresponding struct instances.
Further memory optimizations were taken through improvements of the versioning system. This was done by avoiding nested function calls and continuing to reduce the amount of classes. Another optimization step was the introduction of a queue for pending applications of updates, which is meant to combat stack overflows that are caused by nested function calls: Incoming update-application-requests are executed after the new log entry has been appended and not during.
These optimization steps proved to be effective and fixed the problem of stack overflows on FiPys. However, as was later discovered, there are still some issues on LoPy4 devices, which require further attention.

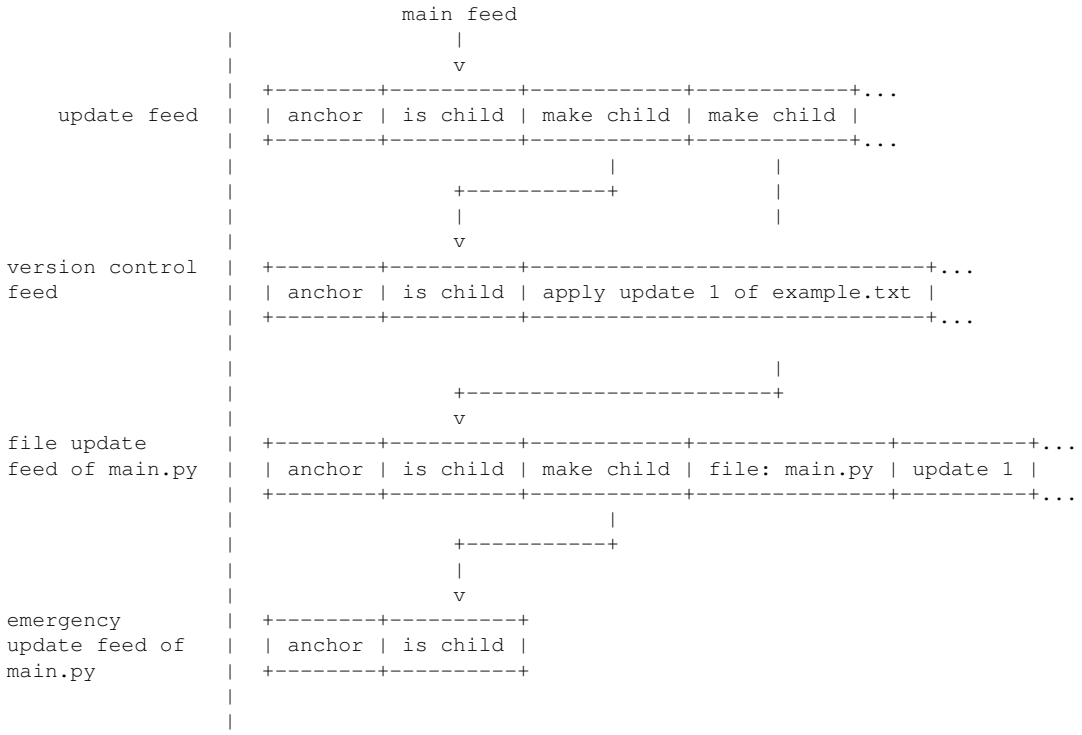
4.2 Feeds and blobs

This section discusses how feeds and blobs, which are at the core of the tinyssb protocol, are used to build the versioning system. First, the update feed and its purpose are explained, followed by a presentation of the version control feed, which is responsible for applying updates. Thereafter, file update and emergency feeds are discussed. Finally, the section covering update blobs talks about how updates are encoded.

4.2.1 The update feed

At the center of the versioning system is the so-called "update feed". Its sole purpose is to serve as an anchor feed for all other update-related feeds. For this reason it only consists of `mkchild` log entries, each referring to another (child) feed. By convention, the first child of the update feed is the *version control feed*, which is used for managing the roll-out of updates and will be discussed in the next subsection. All other children of the update feed are responsible for tracking every version of a file and are called *file update feeds*. These will be presented in subsection 4.2.3. The final feed type of the version control system are emergency update feeds. As the name suggests, they are responsible for recovering from bad updates and are the first child feed of their corresponding file update feed. Emergency feeds are also presented in their own subsection (4.2.4). A visualization of the feed constellation, described above can be seen in the following figure.

Figure 4.1: Feeds of the versioning system



4.2.2 Version control feed

The version control feed is the first child of the update feed. Besides the `ischild` packet at the start of the feed, it only contains log entries of type `applyup`. These are not part of the third tinyssb proof-of-concept and were introduced for the versioning system. Each `applyup` packet points to an update within a file update feed that should be applied. Packets of type `applyup` have the following structure:

1. The first 32 bytes of the payload contain the feed ID of the affected feed.
2. The next 4 bytes contain the version number of the update that should be applied, encoded as a 32-bit integer in big endian.

Version numbers start counting at zero (which is the original version of the file), increase by one for every appended update and persist across child feeds. This way every update can be identified by a filename and version number. The advantage of this system is that version numbers are unique, whereas sequence numbers are identical across child feeds.

In order to determine the current version number of a file, one simply has to look for the newest `applyup` packet pointing to the corresponding file update feed. This can easily be done by iterating over the version control feed in reverse. If no suiting log entry can be found, the current version number is 0, which is the original file.
Once a node receives an `applyup` packet, it checks whether the requested version is available. If so, the update is applied immediately. In the case that the update in question is not yet present, it is added to a dictionary, which contains the filenames and version numbers of all updates that could not be applied yet. Every time a new update is appended to a file update feed, the aforementioned dictionary is consulted and the newly added update is applied (if needed). In the case that a new `applyup` packet arrives before the anticipated update, the version number is overwritten in the dictionary and the application of the intermediary update is skipped. This is possible due to how version-jumping is handled by the versioning system, which is presented in subsection 4.3.1.
The strength of this system is that updates can be applied and reverted on a file-to-file basis. However, it is impossible to know the current version of a file from the state of the version control feed alone, since a node could lose power and fail to apply a pending update. This is the main weakness of the proposed system and is countered by saving the current version number of each monitored file to a log file. The system also cannot handle cross-file dependencies, as the application of updates cannot be synchronized. This could be added in a future version and is discussed in chapter 6.

4.2.3 File update feeds

Each file that is monitored by the versioning system gets its own file update feed. By default, the versioning system monitors every Python file that is located in the current execution's path, including subdirectories. The file update feed is where all updates of a file are appended to. In order to assign filenames to feeds, the packet type `updfile` was introduced. It's payload is structured in the following way:

1. The first byte contains the length of the filename, encoded as a `VarInt`. Since the filename is limited to 43 bytes (due to the maximum size of a log entry's payload), the `VarInt` will always take up exactly one byte.

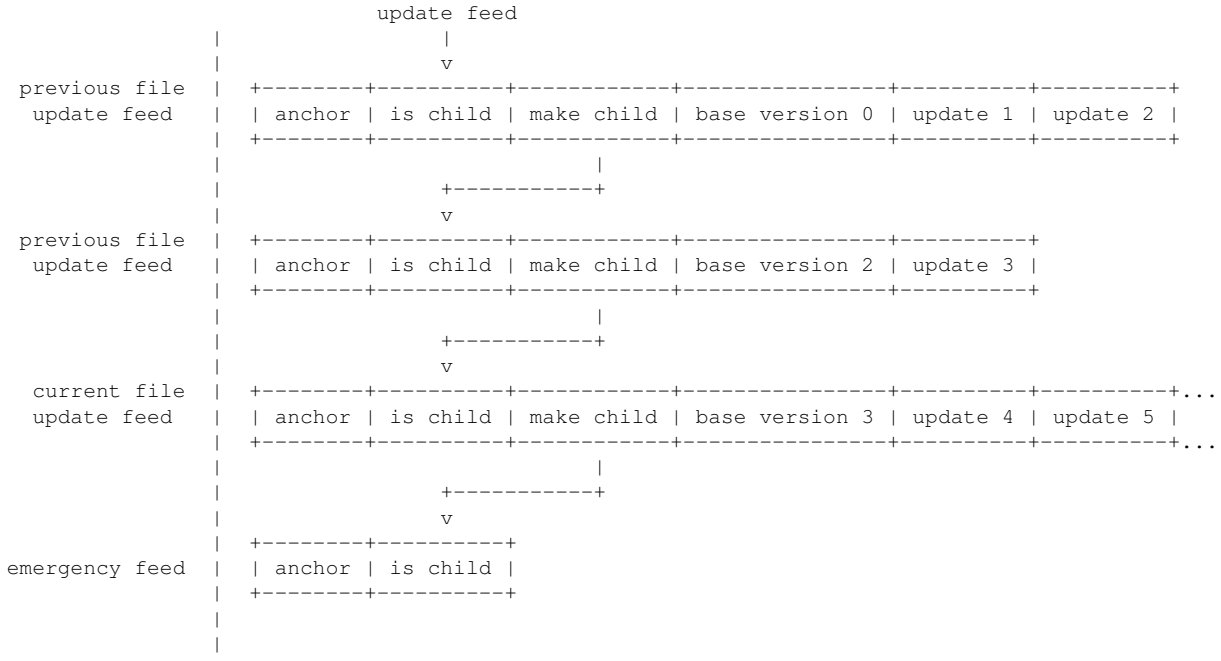
2. The following 4 bytes contain the base version number of a file update feed, encoded as a 32-bit integer in big endian. The first update feed of a file is assigned the base version version number 0. After that, each replacement of the update feed is assigned the base version number equal to the version number of the most recently appended update of its parent feed. The process of replacing the current update feed and its purpose are explained in subsection 4.2.4.

   The idea behind adding the base version number to a file update feed is to eliminate the need of recursing over all of its parent feeds in order to determine the correct version number of an update. This is needed when applying an update or building the dependency tree of a file.

3. The remaining bytes are reserved for the name of the monitored file. Hence, the maximum filename length is 43 bytes.

The first log entry that is added to a new file update feed is a reference to its child feed, which is created as a precaution to prevent bad updates from blocking its successors, since feeds require every packet to be appended in sequence in order to maintain the trust chain. This child feed is called an emergency feed and is presented in the section 4.2.4.
The following figure shows a typical structure of a file update feed and its children. It also demonstrates how version numbers of updates are counted across multiple feeds including the use of `updfile` log entries.

Figure 4.2: Determining the version number of an update



4.2.4 Emergency feeds

Because the versioning system is built on top of the tinyssb protocol, already appended updates cannot be removed. This can be critical in the case of bad updates. If, for example, the wrong file is uploaded, every node must append the faulty update before being able to receive the correct version. This problem is partially solved by the version control feed, which orchestrates the application of updates: It is possible to revert back to a stable version of the file. However, a large and bad update could congest the network, since the full blob chain of the new version must have been received before the next update can be added and applied. This is due to the fact that log entries must be appended in sequence for the trust chain to continue.
To enable nodes to "skip" bad updates, so-called emergency feeds were introduced. Every file update feed has exactly one child, which is created before the `updfile` log entry or any updates are added. This is the file's emergency feed and allows the administrator to create *emergency updates*.
An emergency update goes through the following steps:

1. First, a child feed of the emergency feed is created. It will become the replacement of the current emergency feed during the next step.

2. Secondly, a log entry of type `updfile` is appended to the emergency feed, making it the new file update feed. The aforementioned log entry contains the name of the monitored file and the version number of the most recent update that is appended to the old file update feed.

3. Next, the emergency update (same encoding as a regular update) is appended to the new file update feed.

4. Finally, a log entry of type `applyup`, referring to the emergency update, is appended to the version control feed. This prevents nodes from switching to the bad version.

Finding the current update and emergency feeds of a file is simple: They can be determined by recursively iterating over the children of the original update feed, whereby the last child is the emergency feed and its parent the current file update feed.
The addition of emergency feeds allows nodes to skip bad updates and jump to a newer version of the file, even if the previous version's blob chain has not been fully appended yet.

4.2.5 Update blobs

Every update is appended in the form of a tinyssb blob to the corresponding file update feed and consists of a chain of insert and delete operations. Blobs are used in order to allow for updates of variable size, since regular feed entries only allow for a payload of up to 48 bytes. An exact specification of tinyssb blobs can be found on Github [8]. Each change that is contained within an update, is represented as a triple containing:

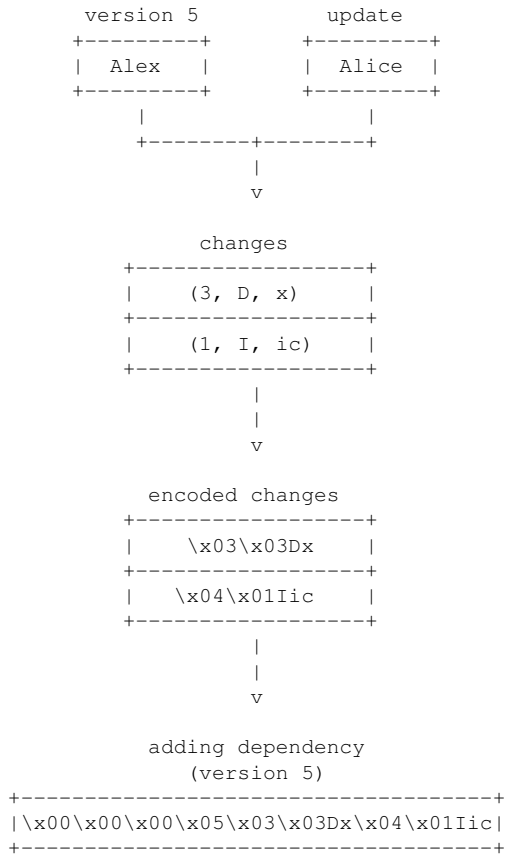
1. The index in the edited string.

2. The operation, which either is deletion ('D') or insertion ('I').

3. The content that is being inserted or deleted. This might seem unnecessary for delete operations, but allows for the reversion of updates, which is a key feature of the proposed versioning system.

The algorithm, responsible for computing changes in between two versions of a file is presented in subsection 4.3.2.
A change is encoded as bytes by converting its index to a `VarInt`, its operation to a single ASCII byte and its content to the equivalent UTF-8 encoded bytes. Finally, the total number of bytes of the encoded change is prepended in the form of a `VarInt`. A visualization of this process can be seen in figure 4.3.
With the intent of allowing for more advanced version management than continuous code deployments, every update also contains a dependency. This dependency is expressed through the version number of the file, on which the new update is based on. It is encoded as a 32-bit integer in big endian and represents the first four bytes of every update. Using this feature, a dependency tree can be built within the update feed of a file, which enables users to change a file to any of its other versions, even if said version is located in a different update branch.

Figure 4.3: Example encoding of an update



4.3 Versioning system

The following two subsections will discuss how the versioning system makes use of the information stored within the update-related feeds, which were presented in the previous subsections. First, the process of jumping between any two versions of a file is discussed, followed by an overview of how new updates are created and appended to file update feeds. This includes a presentation of the file difference algorithm, which is used to compute the necessary insert and delete operations that transform one version of a file into another.

4.3.1 Jumping between different versions of a file

Allowing for arbitrary jumps in between different versions of the same file is one of the main benefits of the proposed versioning system. It is used when a new version of a file is applied and when a user selects to view a specific version of a file in the GUI.
In order to jump from one update to another, a list of all changes between the two versions has to be computed. This is done by constructing the dependency tree from the corresponding file update feed. First, a vertex is created for each version that exists of a file. During the next step, the algorithm iterates over every update and inserts an edge in between its version number and its dependency. This is repeated recursively for every parent feed, an example of which can be seen in figure 4.5. Every new update introduces exactly one edge into the graph (because it has exactly one dependency), which means that it is impossible to create cycles within the version graph of a file.

In order to jump from one version to another, the shortest (and only) path must be found. This is done using breadth first search and might involve reverting an update of a file. This is done by flipping the operation of each change (e.g. deletions to insertions and vice versa) and swapping the order of the insertion and deletion blocks. A visualization of this process can be seen in figure 4.4.

Figure 4.4: Reverting updates

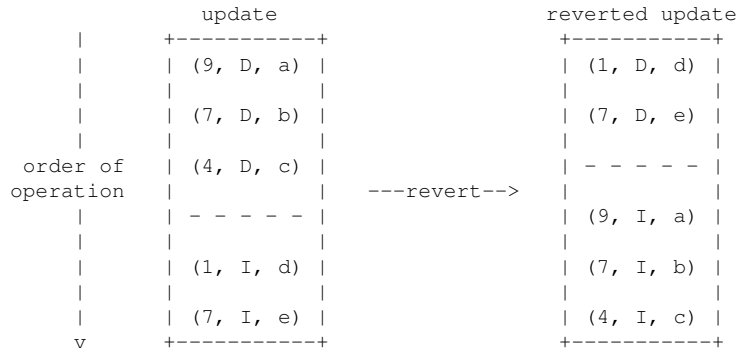
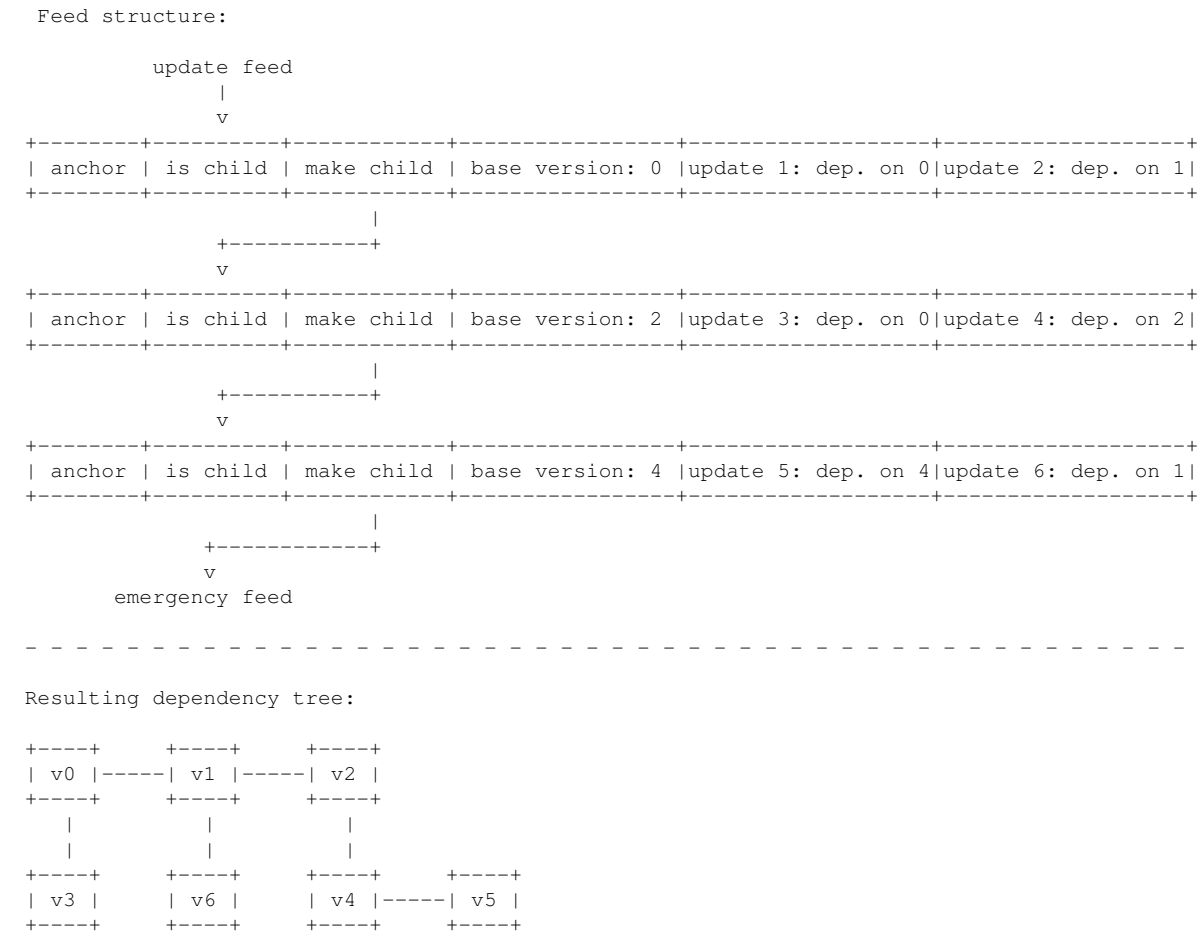


Figure 4.5: Constructing the dependency tree



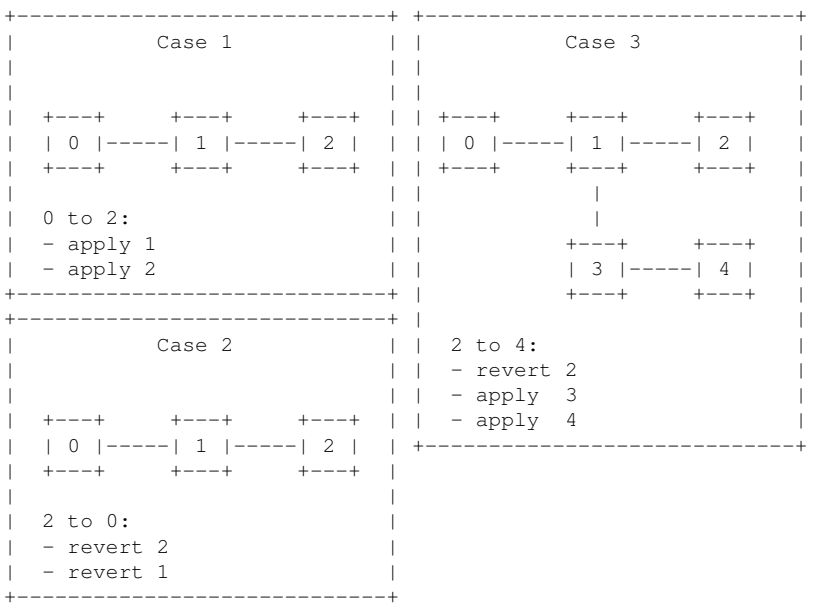
Found paths can be grouped into three different categories:

1. The version numbers increase for each step of the path. This means that every update can be read and applied directly from the feed, resulting in the desired version.

2. The version numbers decrease for each step of the path. This is exactly the opposite scenario of case (1), meaning that every update must be reverted. This is done by swapping the order of insert and delete operations and inverting their actions (a delete becomes an insert and vice versa).

3. In the only remaining scenario, the version numbers first decrease to a minimum version number and then increase to the desired version. This is the case, if two versions are located within two different branches of the dependency tree. Since it does not contain cycles, there only exists one possible path from the starting to the target node, which is via the their last common ancestor. In order to jump between two such versions, the first half of the updates is reverted, and the second is applied, resulting in the target version of the file. An example of this can be found in figure 4.6.

Figure 4.6: The three possible cases of version jumping



4.3.2 File difference algorithm

Seeing that LoRa has a relatively low bandwidth, reducing the size of updates is a vital part of the versioning system. Simply appending the whole updated file to a feed is very inefficient in regards to memory, as many lines of the code are likely to stay the same. This is where the file difference algorithm comes in: Its job is to compute the best chain of insert and delete operations that transform the original file into its updated version.

During this project, two different approaches to this problem were taken. The first was an own attempt at implementing a file difference algorithm, by comparing the old and new versions of a file line by line. This strategy is also used in UNIX's `diff` program, which was introduced in a technical report by James W. Hunt and Douglas McIlroy at Bell Labs [3]. However, the first home-made implementation of the algorithm was very naïve, as it did not consider solving the longest common subsequence problem.
First, both files are split by newline characters and are inserted into two stacks, one for each version. During the next step, the first line is popped from each stack and compared. This results in one of two possible scenarios:

1. **The lines are identical:**
   This means that nothing has to be changed and the algorithm continues to the next iteration.

2. **The lines are different:**
   In this case, it is checked whether the old line exists in the stack of the new lines. If not, this means that it was deleted during the update. The current line number and the content of the old line are marked accordingly.

   However, if the old line is present in the new stack, this means that the new line is

an insertion. The current line number and the content of the new line are marked accordingly and the old line is pushed back to the stack.

As soon as one of the stacks is empty, every remaining line of the other stack is considered an insertion or deletion, depending on which stack has run out. For example if the stack containing the new lines is empty, all remaining old lines are considered deletions and vice versa. An implementation of this algorithm can be found in figure A.1 in the appendix.
This implementation generates correct results, however they turn out to be very inefficient in some cases, as small changes in a file would lead to unnecessarily many insert and delete operations. This can be illustrated with the following example:

Let us assume that we have a file that contains exactly two empty lines, one at the start and one at the end. If we now choose to delete the leading empty line, the algorithm will check whether it exists later in the new file. As the last line of the file is also empty, it is pushed back to the stack and every following line is considered an insertion of the update, even though they were not changed. This continues until the first line of the old file and the last line of the new line match up. After this, the stack containing the new lines is empty and all remaining old lines (which is every line, but the first) are considered deletions. This results in a correct chain of insert and delete operations, but is obviously very inefficient.

In order to solve this issue, the problem of the longest common subsequence (LCS) was taken into account, as used in `diff` [3]. This was combined with further advances in the field, published in the paper "An O(ND) Difference Algorithm and Its Variations" by Eugene W. Myers, who proposed the use of so-called edit graphs for finding the shortest chain of operations needed to transform one file into another [5]. This approach is also commonly used in version control systems, such as Git [6].
A further difference of the second approach is that the files are no longer split by lines but considered as one single string. This allows for more precise edits, meaning that changes of a single character do not result in the replacement of the whole line. The current implementation of the file difference algorithm goes through the following three steps:

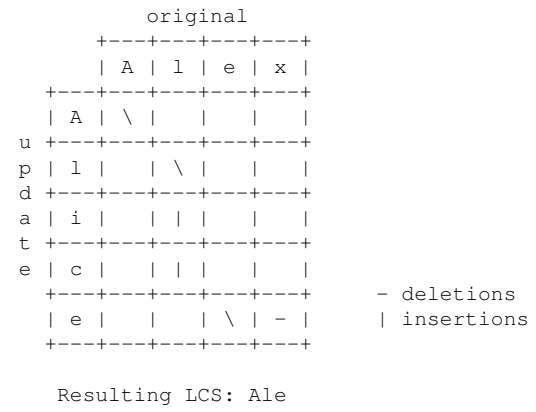
1. An edit graph [5] is built from the two different files, read as single strings. It can be used for determining the longest common subsequence and the necessary insert/delete operations in order to form the original file into its updated version. An example of such an edit graph can be seen in figure 4.7.

   The implementation of the algorithm that computes the edit graph and LCS of two strings can be found in figure A.2 in the appendix.

2. The resulting edit graph is used to compute the insert and delete operations that transform the original file into its updated version. This is done by first comparing the LCS with the original string, whereby differences are considered deletions. The same is done with the new string, where changes are considered insertions. Every operation, consisting of the index within the edited string, its type (deletion or insertion) and modified character, is added to a list. Since each operation only considers a single

character, consecutive changes are chained together in order to reduce the size of the update.

Figure 4.7: Example of an edit graph



This implementation of the file difference algorithm results in compact updates and is a clear improvement over its first iteration. There also exists further performance improvements of the `diff` algorithm, which were proposed in the paper "Speeding-up Hirschberg and Hunt-Szymanski LCS Algorithms" [1] and could be implemented in a future project.

4.4 Web GUI

The only thing missing from the versioning system is a way for users to interact with it. Since Pycom devices do not have any displays, it was decided to create a web-based GUI. This comes with many benefits that will be presented in this subsection.
Each node runs a HTTP server, which hosts a web GUI that can be used for visualizing the current state of a node, displaying debugging information and interacting with the versioning system. On the FiPy this is done by creating a WiFi network through which the aforementioned webpage can be accessed. Since some requests can be computationally intensive for Pycom devices, only one connection is accepted at a time.
The idea behind opting for a web-based GUI was that it is easily expandable, works across multiple platforms and can be accessed wirelessly on Pycom devices. A further advantage is that it allows the offloading of some computationally heavy tasks from the FiPy to the client's browser. The file differential algorithm, for example, is implemented in JavaScript and runs on the client device instead of the FiPy.
Note that the current implementations of the HTTP server and the web GUI were only tested in Mozilla Firefox and may behave differently in other browsers.

The web GUI provides the following pages and functionalities:

1. The first page is the "visualizer". Every time a packet or request is received or sent, the corresponding feed and time stamp (in the form of a constantly increasing tick) are logged to a file. The web page requests the contents of this file once every second

and creates a live waterfall-like visualization of all feed activities. Each feed is colored using the hex code, formed by the first six bytes of its public key (feed ID). In order to differentiate incoming from outgoing messages, packets that are being sent are plotted with an alpha value of 50%, making them appear darker on the black background of the page. The visualizer is deactivated on Pycom devices, since it requires keeping an additional log file, which wastes memory and is computationally intensive because the feed belonging to each outgoing packet or request has to be determined (for incoming messages this is done anyways). A screenshot of the visualizer can be seen in figure 4.8.

2. The second page of the GUI contains a debugging visualization of the current state of the node. Every locally stored feed is plotted, including the type of each appended packet together with its sequence number. Child feeds are indicated by offsetting their anchor packet to the same vertical position as the respective `mkchild` packet within the parent feed. The visualization is text-based and can also be displayed in the terminal. An example of this can be seen in figure 4.8.

3. The third page displays a list of all files that are monitored by the versioning system together with their corresponding dependency trees and public keys. The currently applied version of the file is highlighted in the graph.

4. The final page is the file browser. It lists every file that is monitored by the version control system but does not display the dependency graphs. This reduces the computation needed for generating the page on the server-side and improves loading times on Pycom devices, where generating all dependency trees may take tens of seconds.

   Using this page, new (blank) files can be created and added to the versioning system. It is also possible to place these into directories, by adding a path to the filename. In the case that a directory does not exist yet, it is created. The creation of a new file and/or directories is automatically propagated through the tinyssb network.

Pages (3) and (4) also allow the user to open the file viewer, which displays the currently applied version of the selected file. It is also possible to visualize any other available version by selecting the corresponding version number in the page's menu. Doing so will send a request to the HTTP server, which then fetches the selected version using the dependency tree of the file. By clicking on the `apply_update` link, the versioning system will append a corresponding `applyup` log entry to the version control feed and update the local version of the file. Note that this is only possible on the node that has the private key of the version control feed, allowing it to append valid new log entries.
Finally, the user has the opportunity to open an editor of the currently selected version of a file, making it the dependency of a new update. As previously mentioned, the file difference algorithm is run on the client side, computing the changes that occurred through the update of a file. These are then encoded as a json object and sent to the server, where they are encoded as bytes and appended to the respective file update feed.

The editor also allows for the update to be sent as an emergency update. This goes through the same steps as above but posts the changes to a different path on the HTTP server, which then activates the emergency update procedure, as previously described.

Figure 4.8: Screenshots of the web GUI

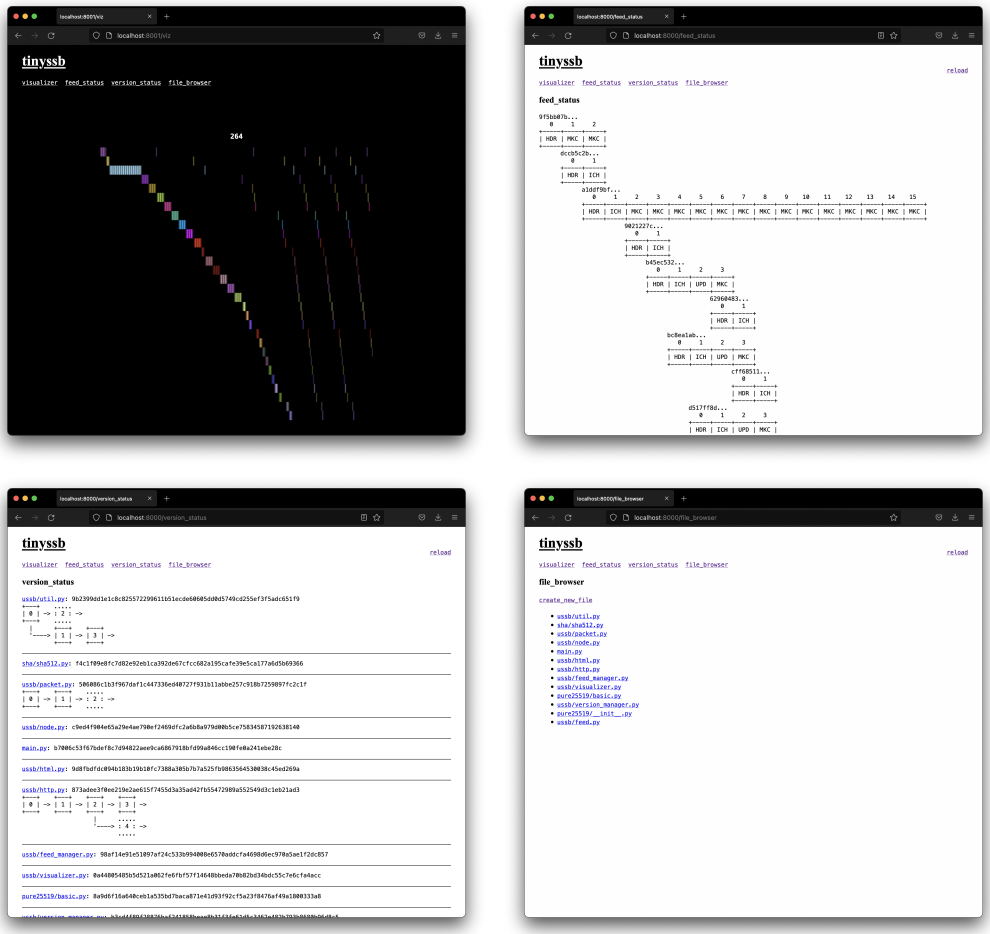
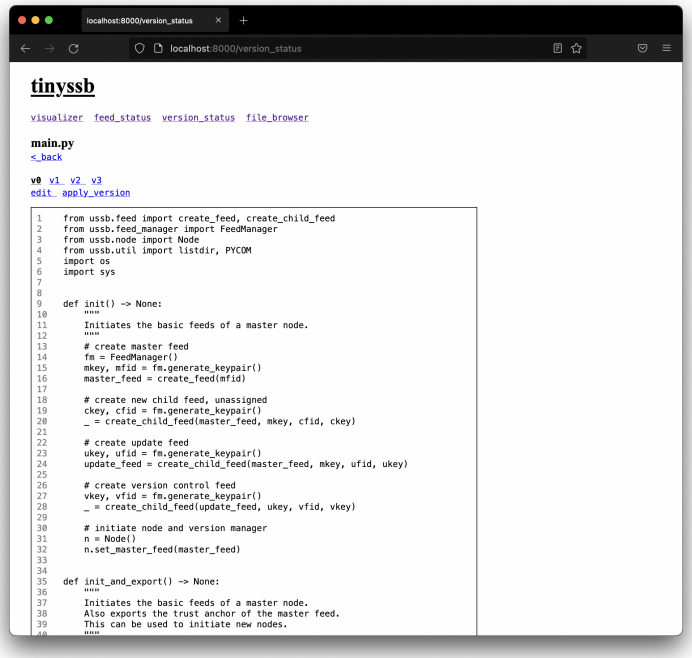
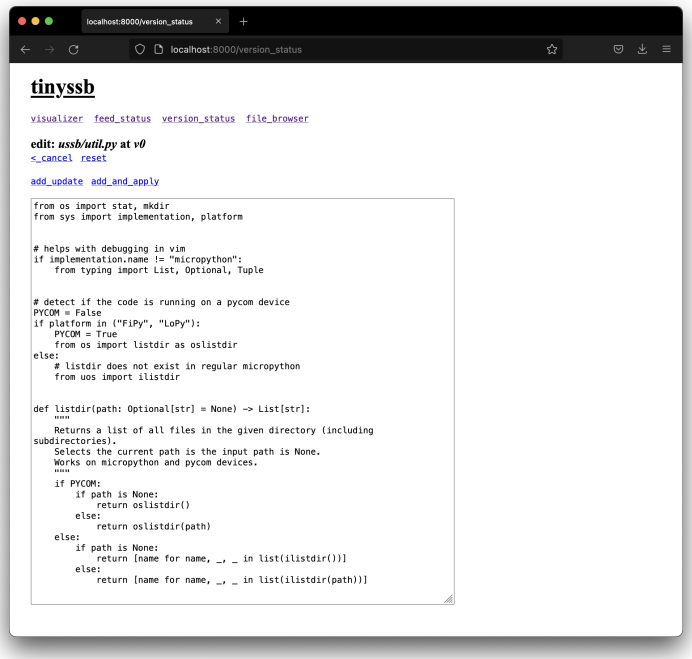


Figure 4.9: Screenshots of the file viewer (top) and editor (bottom)





5 Results

With the goal of testing every feature of the versioning system, a scenario involving three nodes was created and played through. This was done both in UNIX and on three Pycom devices, two of which were FiPys and one was a LoPy4. The idea was to test different aspects of the versioning system, such as executing emergency updates, reverting files and jumping between different branches of a file's dependency tree. This was combined with manually turning off nodes, which is meant to simulate sensors not having enough sunlight and losing power. The following table contains a description of the used test scenario, including the expected behavior of the nodes:

Table 5.1: Test scenario using three nodes

	A	B	C	Expectations
1.	Initialize as update admin.	-	-	-
2.	Create and apply an update of `main.py`, which changes LED color to white.	On	Off	Node A's LED turns white after rebooting the device. Node B requests and applies the update. After rebooting, node B's LED also turns white.
3.	Off	On	On	Node C receives the update of `main.py` from node B. After rebooting, node C's LED also turns white.
4.	Create an update of `main.py` with version 1 as its dependency. The update changes the LED's color to red and represents a bad update.	Off	On	Node A creates the update and displays it in the GUI. The dependency tree is also displayed correctly. Node C requests and receives the update. Both nodes do not apply the update yet.
5.	Version 2 of `main.py` is applied.	On	Off	Node A applies the bad update. After rebooting, its LED turns red. Node B requests and receives the bad update and applies it. After rebooting, its LED also turns red.

	A	B	C	Expectations
6.	An emergency update based on version 1 of `main.py` is created. It changes the LED's color to blue.	On	On	Node A applies the emergency update. After rebooting its LED turns blue. Node B requests the emergency update and applies it. Its LED also turns blue after rebooting. Node C receives both the bad and the emergency updates. It applies the emergency update and its LED turns blue after rebooting.
7.	`main.py` is reverted to version 1.	On	On	Nodes B and C request and receive the apply command. All three nodes apply version 1 of `main.py`. After rebooting, all LEDs turn white again.
8.	A new file and directory are created: `hello/world.txt`	On	Off	Both node A and B create an empty file named `world.txt` in the new directory `hello`.
9.	The file `world.txt` is updated to contain the string "Hello world!".	On	On	The update is displayed in the GUI of node A but is not applied yet. Node C creates the new file and directory and also receives the update. It is also correctly displayed in node B and C's GUIs.
10.	Version 1 of `hello.py` is applied.	On	On	Nodes B and C request and receive the apply-update-command. All three nodes switch to version 1 of `hello.py`.

To create the update admin node, `main.py` was run with the argument `i` (initialize). This creates all the necessary feeds and configuration files. Excluding `fm_config.json` (feed manager configuration), which contains all of the private keys, all of the files were copied to the nodes B and C. Therefore all three nodes start with the preconfigured versioning system and do not have to catch up with the state of each file update feed. This is also how the versioning system is intended to be used in an actual sensor network.
In the following section the results of the UNIX scenario are presented, where instead of changing LED colors, corresponding print outs were added to the file.

5.1 Results of the test scenario in UNIX

In UNIX the test scenario could be played through without running into problems. The update of step (2) was applied correctly on both nodes, whilst taking just a few seconds to take effect. During step (3), node C requested the aforementioned update from node B without needing help from the admin node. This shows that the node in charge of updates does not have to be running at all times for new versions of files to be propagated through the network.

Steps (4) and (5) also ran as expected, displaying the correct dependency graphs and file versions in each of the node's GUIs. This is also the case with the emergency update of step (6). However, node C received and applied the bad version of the file before switching to the emergency update. This behavior is not ideal, since the node could end up running the bad version if it loses power before receiving the patch.
A possible fix for this issue could be to send a request for the next entry of the version control feed before applying an update. If no new version control packet arrives within a given time frame, the update is applied. Otherwise the apply-command is overwritten. This way a node would check if a more up-to-date apply command exists before switching to a different version of a file.
Finally, the creation of the new file and directory during steps (7 - 10) was successful on all three nodes and was correctly displayed in their GUIs.
Summing up, it can be said that the version control system passed the test scenario and appears to be robust when run in UNIX. By changing the packet requesting policy of nodes, the issue of applying updates without checking if a newer version already exists could be addressed.

5.2 Results of the test scenario on Pycom devices

The test scenario was played through using two FiPys and one LoPy4, whereby the update admin node was run on a FiPy. The first notable difference was the response time of the GUI, which took tens of seconds to fetch the debugging page. Nevertheless, it functioned properly when given enough time to load.
Steps (2 - 5) worked just like in UNIX but took longer to execute. This is due to the decreased computational power of the devices and longer intervals in between messages, caused by the use of LoRa. Also noticeable was the delay after creating a new update on the admin node: It took more than ten seconds, probably due to `ed25519`, which is used for signing the packet that contains the update and is not optimized for Pycom devices.
During step (6), node C applied the bad update before switching to the emergency update. This is the same issue as in UNIX and could be solved the previously proposed fix. However, node B crashed due to stack overflows every time it received the emergency update. This is a fatal error and was not caught during development, since the software was only tested on FiPys.
Reverting back to version 1 of `main.py` during step (7) worked on all three nodes, even on node B, which continued crashing periodically. This was possible, since version 1 had already been appended to the file update feed and node B still received and sent messages to and from other devices.
The last three steps of the test were successfully executed on node A. However, nodes B and C did not create a local instance of the new file and directory even after receiving the new file update and emergency feeds. Through further investigation, an issue with the configuration file of the version control system was found: All of its contents were deleted, resulting in unexpected behavior. Interestingly, this issue only occurred on Pycom devices and could not be reproduced in UNIX. Further, it also did not occur when retesting the creation of

files using two FiPys. In order to determine the cause of this bug, more testing on Pycom devices has to be done.
In conclusion, most features of the version control system work on FiPy devices, whereas LoPy4s seem to run into problems when receiving emergency updates. There also appear to be some inconsistencies between the behavior of the code in UNIX and on Pycom devices, which have to be explored through further testing.

6
Conclusion and Future Work

The versioning system that was implemented during this project proved to be a viable solution for the management and distribution of updates across a tinyssb network. It was thoroughly tested in UNIX and also performs reasonably well on FiPys. However, the remaining bugs on Pycom devices are proof that there is still more work to be done. This is especially true for the stack overflows that occurred on the LoPy4, which could probably be fixed through further device-specific optimizations in the code base. These compatibility issues were one of the main challenges of this project: Developing for Pycom devices proved to be more difficult than expected, since testing was substantially slower when compared to desktop devices and seeing that code sometimes behaved differently than in UNIX.

An important next step for the versioning system would be to field-test it using more than three nodes. Seeing how the system performs in a larger network and less controlled environment would deliver more insights on its effectiveness. This could be combined with an expansion of the code base, making use of the sensors of the Pysense 2.0X development board and displaying the recorded measurements in the web GUI.

Another interesting project idea would be to optimize the request-sending-policy of nodes. For example, a node could learn which feeds belong to which of its neighbors and use this information in order to improve its packet-requesting behavior. This could be used in order to reduce the amount of unanswered requests that are being broadcast to the network and possibly increase the speed at which updates are being distributed. Going one step further, important feeds (for example the version control feed) could be assigned a higher priority when determining the next request to be sent. This could be a step towards improving the effectiveness of emergency updates.

Finally, the versioning system itself could be expanded to allow for cross-file dependencies. This would be a great addition, since code that is contained in one file is often closely linked with (or even dependent on) another file. This concept could also be expanded to device-specific updates, which would allow the use of different devices within the same versioning system and tinyssb network.

Bibliography

[1] Maxime Crochemore, Costas S. Iliopoulous, and Yoan J. Pinzon. Speeding-up hirschberg and hunt-szymanski lcs algorithms. *Fundam. Inf.*, 56(1–2):89–103, jan 2003. ISSN 0169-2968.

[2] Jonathan de Carvalho Silva, Joel J. P. C. Rodrigues, Antonio M. Alberti, Petar Solic, and Andre L. L. Aquino. Lorawan — a low power wan protocol for internet of things: A review and opportunities. In *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*, pages 1–6, 2017.

[3] J. W. Hunt and M. D. Mcilroy. An algorithm for differential file comparison. computer science. Technical report, 1975.

[4] Simon Laube. Private communication, April 2022. URL https://github.com/simonlaube/bsc.

[5] Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1: 251–266, 1986.

[6] Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. How different are different diff algorithms in git? *Empirical Software Engineering*, 25(1):790–823, September 2019. doi: 10.1007/s10664-019-09772-z. URL https://doi.org/10.1007/s10664-019-09772-z.

[7] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. pages 1–11, 09 2019. ISBN 978-1-4503-6970-1. doi: 10.1145/3357150.3357396.

[8] Christian Tschudin. Private communication, March 2022. URL https://github.com/tschudin/tinyssb.

A

Appendix

A.1 File difference algorithms

Figure A.1: First implementation of a file difference algorithm

```
get_changes(old_version: str,
            new_version: str) -> List[Tuple[int, str, str]]:
changes = []
old_lines = old_version.split("\n")
new_lines = new_version.split("\n")

line_num = 1
while len(old_lines) > 0 and len(new_lines) > 0:
    old_l = old_lines.pop(0)
    new_l = new_lines.pop(0)

    # lines are the same -> no changes
    if old_l == new_l:
        line_num += 1
        continue

    # lines are different
    if old_l not in new_lines:
        # line was deleted
        changes.append((line_num, "D", old_l))
        new_lines.insert(0, new_l)  # retry new line in next iteration
        continue

    # old line occurs later in file -> insert new line
    old_lines.insert(0, old_l)  # retry new line in next iteration

    changes.append((line_num, "I", new_l))
    line_num += 1

# old line(s) left -> must be deleted
for line in old_lines:
    changes.append((line_num, "D", line))

# new line(s) left -> insert at end
for line in new_lines:
    changes.append((line_num, "I", line))
    line_num += 1

return changes
```

Figure A.2: Implementation of the LCS-based file difference algorithm

```
function getChanges(oldV, newV) {
    if (oldV === newV) return [];

    // get lcs string
    mid = extractLCS(oldV, newV);

    // compute insert/delete operations
    let changes = [];

    // start with delete operations -> difference to original
    let j = 0;
    for (let i = 0; i < oldV.length; i++) {
        if (oldV[i] !== mid[j]) {
            // check for consecutive deletions
            const sIdx = i;
            let x = oldV[i];
            while(++i < oldV.length && oldV[i] !== mid[j]) {
                x = x.concat(oldV[i]);
            }
            changes.push([sIdx, 'D', x]);
            i -= 1;
            continue;
        }
        j += 1;
    }


    // compute insert operations -> difference to update
    j = 0;
    for (let i = 0; i < newV.length; i++) {
        if (newV[i] !== mid[j]) {
            // check for consecutive insertions
            const sIdx = i;
            let x = newV[i];
            while (++i < newV.length && newV[i] !== mid[j]) {
                x = x.concat(newV[i]);
            }

            changes.push([sIdx, 'I', x]);
            i -= 1;
            continue;
        }
        j += 1;
    }

    return changes;
}
```

Figure A.3: Function that computes the LCS of two strings

```
function extractLCS(s1, s2) {
    // computes the lcs of two given strings

    // get grid
    mov = getLCSGrid(s1, s2);
    let lcs = '';

    // "walk" through grid
    // 0 -> left, 1 -> diagonal, 2 -> up
    let i = s1.length - 1;
    let j = s2.length - 1;

    while (i >= 0 && j >= 0) {
        if (mov[i][j] == 1) {
            lcs = s1[i] + lcs;
            i--;
            j--;
            continue;
        }
        if (mov[i][j] == 0) {
            j--;
            continue;
        }
        i--;
    }

    return lcs;
}
```

Figure A.4: Function that computes the LCS grid of two strings

```
function getLCSGrid(s1, s2) {
    // computes the lcs grid of two strings
    const m = s1.length;
    const n = s2.length;

    // left = 0, diagonal = 1, up = 2
    let mov = new Array(m).fill(-1).map(() => new Array(n).fill(-1));
    let count = new Array(m).fill(0).map(() => new Array(n).fill(0));

    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            if (s1[i] === s2[j]) {
                let val = 0;
                if (i > 0 && j > 0) {
                    val = count[i - 1][j - 1];
                }
                count[i][j] = val + 1;
                mov[i][j] = 1;
            } else {
                let top = 0;
                if (i > 0) {
                    top = count[i - 1][j];
                }

                let left = 0;
                if (j > 0) {
                    left = count[i][j - 1];
                }
                count[i][j] = top >= left ? top : left;
                mov[i][j] = top >= left ? 2 : 0;
            }
        }
    }
    return mov;
}
```

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

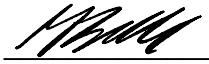
Title of Thesis: Managing and Distributing Software Updates Using Append–only Logs

Name Assesor: Prof. Dr. Christian Tschudin

Name Student: Maximilian Barth

Matriculation No.: 19–051–770

With my signature I declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Place, Date: Basel, 18.06.2022 Student: 

Will this work be published?

☐ No

☒ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: Basel, 18.06.2022 Student: 

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .