UNIVERSIDADE DE VIGO

ESCOLA DE ENXEÑERÍA DE TELECOMUNICACIÓN

PROXECTO FIN DE CARREIRA

Intelligent Update and Self-Test Agent for Collaborated Embedded Software

AUTORA: Fátima Castro Jul
TITORA: Ana Fernández Vilas, Rebeca P. Díaz Redondo
CURSO: 2013-2014

OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Developed under the Erasmus framework
Promotor:
Prof. Juha Röning
Daily supervisor:
Teemu Tokola

Fátima Castro Jul

Intelligent Update and Self-Test Agent for Collaborated Embedded Software

Master's Thesis
Degree Programme in Computer Science and Engineering
May 2014

Acknowledgements

I would like to express my gratitude to Juha Röning and the Oulu Security Programming Group (OUSPG) in the University of Oulu for this opportunity to write my Master Thesis in Finland. I want to thank specially Teemu Tokola, for his guidance during the last months.

Many thanks to my family and my friends, who have been supporting me during my studies. And of course to my Erasmus friends, with whom I have shared so many good moments.

Resumen

Debido a la cantidad de dispositivos electrónicos que se manejan actualmente, los usuarios deben hacer frente a una creciente cantidad de software y a su consiguiente mantenimiento. Las actualizaciones son una parte clave de este mantenimiento y, para facilitar la labor de los usuarios, los sistemas de actualización se están volviendo cada vez más automáticos. De hecho, las notificaciones y los envíos de actualizaciones se realizan de forma automática en muchos de los sistemas actuales. La actualización de código en ejecución, conocida como actualización dinámica, representa el siguiente paso de esta tendencia. La actualización dinámica es especialmente interesante en el caso del software embebido, en continuo desarrollo debido al incremento del número de dispositivos interconectados, que forman lo que se conoce popularmente como el Internet de las Cosas. Actualizar los dispositivos embebidos ha sido siempre difícil a causa de su autonomía y, precisamente por esta razón, los sistemas de actualización dinámica se adaptan mejor a ellos. Sin embargo y a pesar de este potencial, hasta la fecha no se ha incluido la integración de las actualizaciones dinámicas en sistemas de actualización completos. Por otras parte, el uso cada vez más frecuente de repositorios colaborativos los convierte en una importante fuente de software para actualizaciones y, por lo tanto, en otra valiosa herramienta que debe tenerse en cuenta en el diseño de mecanismos de actualización.

Este proyecto presenta un sistema completo de actualización destinado a software embebido que aprovecha el potencial de estos elementos. Se trata de un sistema basado en software colaborativo que incluye soporte para actualizaciones dinámicas. Es, además, flexible y capaz de actualizarse a sí mismo. El sistema consiste en un agente de actualización y una librería C. El agente de actualización se encarga de llevar a cabo cada paso del proceso de actualización, incluyendo los tests y la notificación de errores, mientras que la librería C proporciona la funcionalidad necesaria para convertir código en actualizable. En el diseño, se ha prestado especial atención a la seguridad ya que esta cobra importancia con la automatización. El sistema incluye un mecanismo de autentificación de dos factores y ha sido diseñado para resistir a ataques destinados a actualizaciones, como rollback y freeze attacks. El uso del sistema ha sido descrito y demostrado.

Abstract

As a result of the increasing quantity of electronic devices users handle nowadays, they have to deal with a growing amount of software and the consequent maintenance. Updates are a key part of this maintenance and, to ease users' role, update systems are moving towards automation. In fact, features such as update notifications and delivery are commonly performed automatically. The update of code in execution, known as dynamic software update (DSU), is considered the next step in this trend. It is specially interesting for embedded software, which is continuously spreading due to the expansion of interconnected devices, commonly referred to as the Internet of Things. Due to their autonomy characteristics, embedded devices have traditionally been a complicated target for updates and just because of that, DSU systems have a chance to succeed with them. In spite of this potential, existing proposals have not included DSU integration in complete update systems. As the use of collaborative repositories is spreading, they are likely to become a major source of update software and therefore they are another valuable tool that should be considered in updaters' design.

This Master Thesis presents a complete update system targeted to embedded software that takes advantage of the potential of these elements. It is a flexible and self-updatable mechanism based on collaborative software features that includes support for dynamic updates. It consists of an update agent and a C library. The update agent handles every step of the update process, including testing and crash report, while the C library provides the functionality required to make source code updatable. Security features, with increased importance owing to automation, have also been considered. The system includes a two-factor authentication mechanism and it has been designed resistant to attacks targeted at updates, such as rollback or freeze attacks. The usage of the system has been successfully described and demonstrated.

Keywords: dynamic update

Contents

Acknowledgements....i
Resumo....ii
Abstract....iii
Contents....iv
List of figures....vi
List of tables....vii
Notation / Abbreviations....viii
Chapter 1.Introduction....1
1.1 Motivation....1
1.2 Contribution....2
1.3 Structure....2
Part I. State of the art....3
Chapter 2.The update process. Concepts and approaches....3
2.1 Definition of an update....3
2.2 The update process....3
2.2.1 Advertise Update/Receive Information....3
2.2.2 Deliver/Receive Update....3
2.2.3 Configure Update....4
2.2.4 Install/Deploy Update....5
2.2.5 Apply Update....5
2.2.6 Activate/Deactivate Update....5
2.2.7 Remove Update....6
2.2.8 Rollback/Uninstall Update....6
2.2.9 Feedback/Crash report....6
2.2.10 Recovery of tainted state....6
2.3 Update mechanisms....6
2.4 Dynamic updates....7
2.5 Security threats....12
2.5.1 Update distribution security....12
2.5.2 Update acceptance security....15
2.6 Update of embedded systems....16
Chapter 3.Technical background....17
3.1 Raspberry Pi and Raspbian....17
3.2 Git and GitHub....17
3.3 GPG....18
3.4 Rpcgen and xdr....18
3.5 Crontab....18
Part II. Contribution....19
Chapter 4. A dynamic update system through collaborative repositories....19
4.1 Update system description....19
4.2 Download mechanism....20
4.3 Two levels of collaborators....20
4.4 Update procedure....21
4.4.1 Advertise Update/Receive Information....21
4.4.2 Deliver/Receive Update....21
4.4.3 Configure/Reconfigure update....21
4.4.4 Install/Deploy Update....21
4.4.5 Apply Update....24

4.4.6 Activate/Deactivate Update....24
4.4.7 Remove Update....24
4.4.8 Rollback/Uninstall Update....24
4.4.9 Feedback....24
4.4.10 Recovery of tainted state....24
4.5 Security features....25
4.6 Crash report....25
4.7 Configuration of the update system....26
4.8 Self-update....27
4.9 Connection checking....27
Chapter 5.Updatable development library....29
5.1 DSU library....29
5.2 Update points....30
5.3 State transfer....31
5.4 Crash report....32
5.5 Forced update....33
5.6 Version compatibility....33
Part III. Example & Conclusion....35
Chapter 6.Example of usage....35
6.1 Building an updatable program....35
6.1.1 Files to be included....35
6.1.2 Code modifications....36
6.2 Demonstration....37
6.2.1 Dynamic update....37
6.2.2 Dynamic update not compatible....38
6.2.3 Self-update....39
6.2.4 Signature attack....39
6.2.5 Unsuccessful dynamic update....40
6.2.6 Compilation failure....41
6.2.7 Test failed....42
Chapter 7.Conclusion and future work....43
7.1 Discussion....43
7.2 Future work....44
7.2.1 DSU spread....44
7.2.2 Embedded devices....45
7.2.3 Distributed updates....45
7.2.4 Software maintenance tool....45
7.2.5 Collaborative software....46
References....47

List of figures

Figure 1: Update process model....4
Figure 2: Data indirection....8
Figure 3: State transfer....9
Figure 4: Update over insecure channel....12
Figure 5: Freeze attack....13
Figure 6: Downgrade attack....13
Figure 7: Modification of the update files....14
Figure 8: Update over wireless network....14
Figure 9: Update from untrusted source....15
Figure 10: Key compromise....15
Figure 11: Embedded system....16
Figure 12: Raspberry Pi....17
Figure 13: Update system architecture....19
Figure 14: Structure of code in the repository....20
Figure 15: Flowchart of repository update....23
Figure 16: Unauthorized user may use stolen key to deceive the update system....25
Figure 17: Example of crash report in Github....26
Figure 18: Command line options....26
Figure 19: Flowchart focused on self-update....28
Figure 20: Dynamic update sequence....32
Figure 21: Optimal use of the update system....35
Figure 22: Updatable code....36
Figure 23: Output of the version option....37
Figure 24: Output of the first version of the program....37
Figure 25: Output of successful updater execution....37
Figure 26: Updated version output....38
Figure 27: Output of the version option of non-compatible release....38
Figure 28: Output of updater when dynamic update not supported....38
Figure 29: Output of updater when forced update....38
Figure 30: No state transfer after forced dynamic update....39
Figure 31: Self-update of the update agent....39
Figure 32: Untrusted update release....39
Figure 33: Disabled key check....40
Figure 34: Security issues due to unsafe update....40
Figure 35: Example of security crash report....40
Figure 36: Dynamic update failure....41
Figure 37: In case of update failure, the old version continues its execution....41
Figure 38: Compilation failure....41
Figure 39: Test failure....42
Figure 40: Example of test failure crash report....42

List of tables

Table 1: Comparison of automation in update systems.....7
Table 2: Main characteristics of current DSU systems.....8
Table 3: Comparison of existing DSU systems.....11
Table 4: Variables in the DSU library.....29
Table 5: Functions in the DSU library.....30

Notation / Abbreviations

DSU	Dynamic Software Update
GUI	Graphical User Interface
MITM	Man in the Middle
OS	Operating System
RFC	Request for Comments
RFID	Radio Frequency Identification
XDR	External Data Representation

Chapter 1. Introduction

1.1 Motivation

Nowadays users own more and more electronic devices, e.g. laptops, tablets, smartphones, e-books. These devices run different software: operating systems, programs and applications. They all have something in common: they need updating.

Updating means downloading software and installing it on our device. That implies an internet connection and therefore is a security risk. Furthermore once installed updates are accepted as part of the own software and, consequently, trusted. As a result, updates should be handled carefully.

On the one hand, administrators should be aware of update reliability and act accordingly. That may be an arduous task as a device may have dozens of different software, with a different update system for each one of them [1].

On the other hand, they should bear in mind the advantages of software updating: improvements, new features and security patching. As a consequence, avoiding updates as a preventive measure may be a security risk itself. Owing to that fact, silent updates [2] are being developed, as a tool for the software to update automatically when available. Dynamic updates [3]–[8] take a step forward, replacing software in current execution. Nevertheless, they are not yet completely developed so as to be commonly used in product software.

Dynamic updates are especially interesting when considering embedded devices, which are more autonomous than PCs or laptops, at whom traditional updates are targeted. Embedded devices may have no user interface at all, being remotely managed and only accessible through network connection. They are in continuous development as they are used in a wide variety of applications (home appliances, consumer electronics, vehicles, medical or any other type of technical equipment). Additionally, they are key in the expansion of the Internet of Things [9], which propose the idea of having every object around us connected to the Internet, able to fetch any information they require with no need of human interaction. Full implementation of a massive project like this demands a massive quantity of specifically-designed software. In order to perform such a task collaborative embedded systems offer a useful tool that eases cooperation between teams of developers. Project management, code revision and bug detection are examples of stages in a software project development that can be benefit by this approach.

Collaborative repositories provide a mechanism to maintain collaborated software online and are constantly available [10]. As a result, they can also be used as software distribution tool. Web-based hosting services for collaborative repositories such as Github, Stash or Bitbucket are nowadays broadly employed in well-known projects such as Ruby on Rails or Homebrew packet manager. Availability and increasing influence in software development of those repositories make them an attractive option for update

systems, as opposed to traditional rigid ones.

On the whole, embedded devices and collaborative software raise new issues regarding software updates and provide a wide variety of new possibilities still to be developed.

1.2 Contribution

The main goal to be achieved on the update development would be a secure and completely automatic mechanism, about which users can be completely confident even though they are unaware of its execution. That is a challenging objective and this Master Thesis aims to make a contribution in this demanding research.

The objective of this Master Thesis is the design and implementation of a system for updating embedded devices from a collaborated repository.

A review of previous work is accomplished to identify the features and functionalities the updater requires. The review pays special attention to dynamic updates, as they are considered particularly suitable for embedded devices.

The system is subsequently designed and implemented based on GitHub and Raspberry Pi. The approach comprises an script-based autonomous client updater and a C library to be used in dynamically-updatable programs development. It has been designed as a flexible and self-updatable dynamic update mechanism that handles every step of the update process, including testing and crash report, without forgetting about security.

1.3 Structure

This Master Thesis is organized in nine different chapters. In Chapter 2, update system features are discussed and a review of previous work is accomplished. Main steps of the update process are defined and explained, with a focus on the dynamic applying. Chapters 3 and 4 describe the update system that has been designed and developed. Chapter 3 deals with the client updater whereas Chapter 4 covers the C library for dynamic updates. Technologies used in the implementation of both of them are presented in Chapter 5. A demonstration of usage is included in Chapter 6.

Chapter 7 summarizes the Master Thesis content and discusses its features, comparing with previous approaches. The discussion continues on Chapter 8, where future lines of work in the update field are reviewed. Finally, Chapter 9 includes the final conclusions.

Part I. State of the art

Chapter 2. The update process. Concepts and approaches

As the presence of software increases in every aspect of our lives, its maintenance and security becomes more complex. Software update, as a part of the software deployment life cycle, has been approached following different methodologies and assumptions. In addition to that, dynamic updates and embedded systems represent new challenges for researchers and software vendors.

2.1 Definition of an update

A software update is the act of modifying an existing software configuration. The modification may be an addition, removal, replacement or reconfiguration of software functionality in order to fix or improve it [11].

A software update also refers to the collections of files used in the update process.

2.2 The update process

Here follows a step-by-step description of the update process based on [11] and presented in Figure 1. The term client may refer to an end user or to software existent on his device.

Main steps presented in every update system are: advertisement, delivery/reception, configuration, installation and application. Optional features such as removal of updates, rollback, reconfiguration, vendor feedback and recovery of tainted state are also analysed.

2.2.1 Advertise Update/Receive Information

Once the update is available users may be informed by different channels, such as web sites or e-mail. As the process is more and more automatic, resident update notifiers installed on their devices are the most common checking mechanism.

2.2.2 Deliver/Receive Update

As soon as users (or at least software in the users' side) are informed about the update, they can proceed to obtain a copy of the update files in their device. The delivery may consist in distributing CDs with the needed software or a download from the Internet, which is the most common choice nowadays [11]–[14]. As common applications updates need to be targeted at a significant number of users, its large scale distribution can be

accomplished through mirror servers or P2P networks [12].

In this step it is fundamental to check authenticity, integrity and resource constraints (user's disk space, bandwidth). For authenticity checking, MD5 checksum is widely used [15].

Update files can be delivered to customers in different formats:

- Software Packages – Groups of files including metadata.
- Files – Both Source or binaries.
- File deltas – Differences between the customer configuration and the new one. Sending only the difference is more efficient than sending the complete files. Rsync algorithm [16] is commonly used for this purpose.

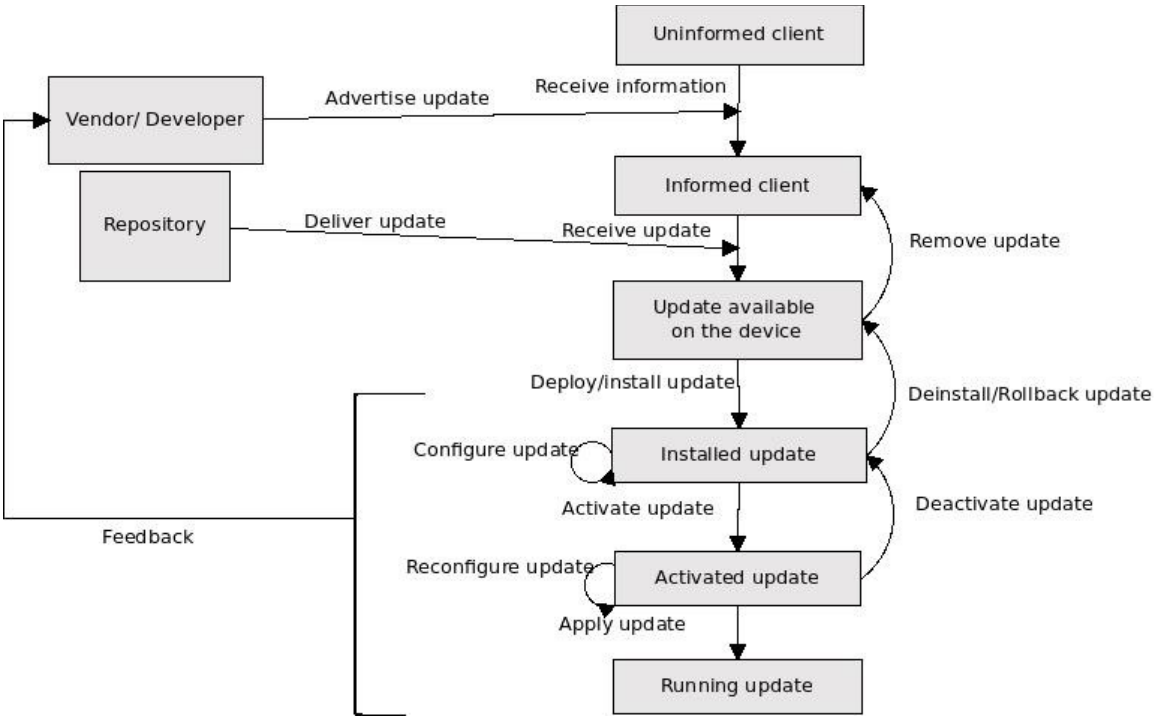

Figure 1: Update process model

2.2.3 Configure Update

Updates can be often configured before installing. Usually this configuration can be performed using configuration files from the previous version, maintaining users' local settings. In rpm packages [17] , configuration files are marked and changes between old package, users file and new package are checked using MD5 checksum.

An update can be reconfigured after activation. Reconfiguration often happens through a configuration interface at runtime, but can also be done by reconfiguring system settings

that affect software behaviour.

2.2.4 Install/Deploy Update

The update deployment is the core of the update process. It consists in the modification of the program files with the update changes. The modification normally implies overwriting previous files. However, it may be only the addition of a plug-in (incremental update).When a rollback feature is considered and no versioning system is used, old files are not overwritten but saved apart.

Before installation, checking to be performed include: verification (completeness, synchronization), dependencies, compatibility and presence of required components.

Dependencies checking is broadly accomplished using metadata, especially in packet management tools. Yum [15] downloads this metadata separately and maintains it cached on user's computers in order to quickly check dependencies before downloading.

Tests on performance of the software update compare the execution with the old software. Hosek and Cadar [18] have developed a system that runs concurrent and separately two different versions of the same software, checking for differences in their external calls and fixing crashes with software from the other version. The result is a more reliable multiversion application. However this approach was implemented in a multi-core processor and may not be suitable for simpler ones. There have been more proposals on testing methodologies [19]–[21] but in general it is not possible to conclude how acceptable is a software version without information on its intended performance, if no evident execution error is encountered.

Compilation or any other preparation for execution of update files is also necessary in this step, except for updaters that provide binary compatibility [6].

2.2.5 Apply Update.

The update applying is the execution of the updated software. The term applying is used instead of application to avoid confusion with the denomination of pieces of software.

Static updates are executed by users while dynamic updates are executed by the update mechanism substituting previous running version. They solve the problem of software disruption, although they are increasing complexity [22]. Although Dynamic Software Update (DSU) systems have been developed using a variety of methods and contexts, they still remain not usable enough to be widely spread [23]. Commercial automatic or silent updates are executed at reboot and are therefore static. DSU systems are reviewed in detail in Section 2.4.

2.2.6 Activate/Deactivate Update

Proprietary software often needs a license approval for the installation or execution. Software vendors may revoke a license after a period of time or users may decide not to renew it.

2.2.7 Remove Update

Users may remove from their devices updates they have decided not to apply or old updates they do not need any more. This feature is useful in systems that allow changing between different software versions and therefore, need to have all the needed ones available.

2.2.8 Rollback/Uninstall Update

The rollback/uninstall feature grants the possibility to go back to previous version in case the new one is troublesome or is not working as expected. It requires incremental updates [24] and/or the maintenance of previous versions on the computer. Instead of uninstall, the term deinstall is used in some literature [11].

2.2.9 Feedback/Crash report

Users may be able to inform software vendors about their experience with the update, which may be useful for future software development. Although some product updaters provide this tool [25], this feedback is currently received mostly through Internet comments, reports, forums [26] or blogs.

2.2.10 Recovery of tainted state

Frequently, correctness of running software is assumed. Using callback functions in several points of the update, software state can be checked and changed to a safe one [27]. Despite being desirable, this feature requires awareness of potential failure points to be checked and how they could be solved.

2.3 Update mechanisms

Abundant possibilities have been developed to accomplish the update process.

Generally, every operating system provides its own mechanism, as well as most popular programs or vendors and even some smaller projects. However, the latter usually lack the resources to create and maintain it, which results in weaker software managers [1]. An alternative for those projects and standalone applications are generic updaters and package deployment tools. Those types of updaters agree with the classification on the overview by Jansen, Ballintijn and Brinkkemper [11]. They also provide a classification of several updaters according to its performance of the different update steps. However, it lacks an analysis of the automation like the one in Table 1.

Systems compared include both vendor [28], [29] and open-source software [15], [30] as well as commercial products [25] and projects [31]. The most automatic systems is Omaha [32], an autoupdate tool for all Google software, although it may be used for any developer for their own software. Its update method without any kind of disturbance or notification to users is named silent update [2].

Most of the mechanisms have not been specifically designed for the update process, but

as complete deployment or distribution tools.

Table 1: Comparison of automation in update systems

Update systems	**Windows installer**	**Apple Software Updater**	**Exact Software's Product Updater**	**Software dock**	**Omaha**	**Apt**	**Yum**
Type	Vendor software updater	Vendor software updater	Vendor software updater	Generic software updater	Generic software updater	Package deployment tool	Package deployment tool
Target	OS	OS	Application	Application	Application	Application	Application
Update notification	Automatic	Automatic	Manual	Automatic	Automatic	_____	____
Update delivery	Automatic	Automatic	Automatic	Automatic	Automatic	Automatic	Automatic
Update installation	Automatic	Automatic	Automatic	Automatic	Automatic	Automatic	Automatic
Notification to end user	Yes	Yes	_____	__*	No	____	____
Update applying	Static	Static	Static	Static	Static	Static	Static

* not considered

2.4 Dynamic updates

The formal basis and essential concepts of dynamic updating were described in the nineties by Gupta, Jalote and Barua [33], although some approaches had already appeared before [34]. They studied crucial issues such as update validity and identified state transfer and determination of valid update points as the main difficulties in implementation. Later work has substantially relied on their conclusions. As an evidence, it has been the most referenced paper on the topic. Despite this formal basis, there has not been a common trend on the development of DSU systems. However, we can distinguish two main classes: those that rely on compilation tools and those oriented to developer modifications to source code. The former try to automate as much as possible the dynamic update process while the latter focus on flexibility and adaptation to the program and developer's needs. Both approaches are outlined in Table 2.

Table 2: Main characteristics of current DSU systems

Based on compilation tools	Based on developer modifications
Automatic Conservative checks to apply some changes Data indirection Restrictions in code	Developer work (Specific libraries) Flexible State transfer

Even when there is no programming work involved in principle, developing an updatable program demands extra effort since there are restriction to be taken into account. That happens because there is no method to perform totally automatic updates. In fact, one of the Gupta, Jalote and Barua's conclusions was that the update point is in general undecidable, which means that in most cases a safe execution point where to apply the update cannot be determined. As a result, update points should be carefully chosen using system support [35], which makes it sometimes less complicate and more reliable to simply ask the developer to indicate when to update in the source code. The mark of a potential update point consists usually on a specifically-designed function. It checks if there is an update available and stops the execution if true, being substituted by a new process that starts the actual execution at this point. Automatically or not, update points have to be placed on parts of the code where the state of the program is quiescent. Indeed, update points are also referred to as quiescent points or states [5], [36]–[38]. However, there is no general agreement about their definition. DSU systems whose target is the full program use points where there are no partially-completed operations, as they may imply the modification of relevant information [3], [4], [39]. Modular or function-based DSU systems consider a safe point where the affected module or function is not being accessed or executed [6], [7]. This alternative imply some limitations as functions in continuous execution, such as *main()*, can never be updated. It has been argued that updates hardly ever affect to those functions. However, a usable update system should support any kind of changes. Timeout mechanisms can be used to prevent too long waits for a module or function to stop its execution [34].

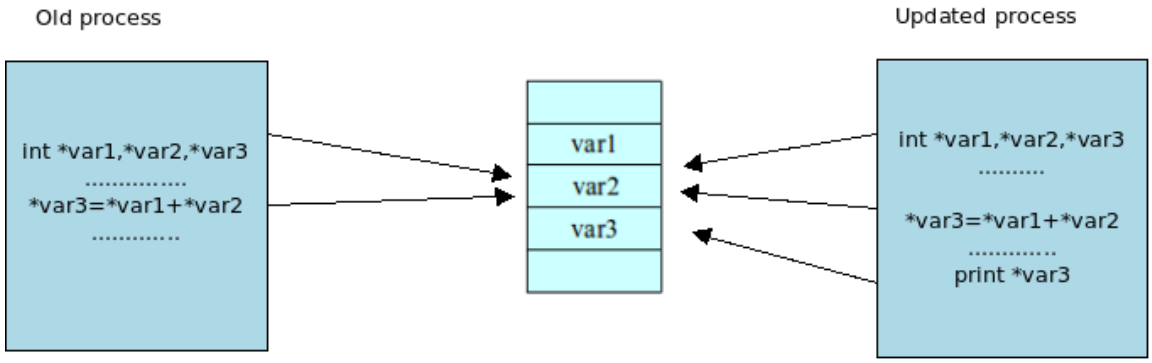

Figure 2: Data indirection

Together with the decision of update points (or state mapping problem), the other main issue regarding dynamic updates is the state transfer. It is essential to make sure that the old and the new version executions uses the same data and values so process substitution can be neatly applied, which guarantee representation consistency. State transfer uses interprocess communication to send data between the old and the new version, as opposed to in-place updating which uses indirection tools to make the old data accessible to the new process. The latter mechanism is used on the compiler-based DSU systems, which transform the source code. In the new code variables and functions are never used by value but by reference, i.e. they are replaced with pointer to their adresses. In case of update they are changed to point somewhere else, where the new function is or where the old data has been saved before (Figure 2). This indirection provokes an steady state overhead [5], [24] in the running processes, as the access time increases. Type-wrapping is also used [4] combined with type transformers included in the dynamic patches. However, in some cases, type transformations are only allowed when they meet given criteria. For example, Ginseng only supports changes if they do not expand beyond a preallocated space [4].

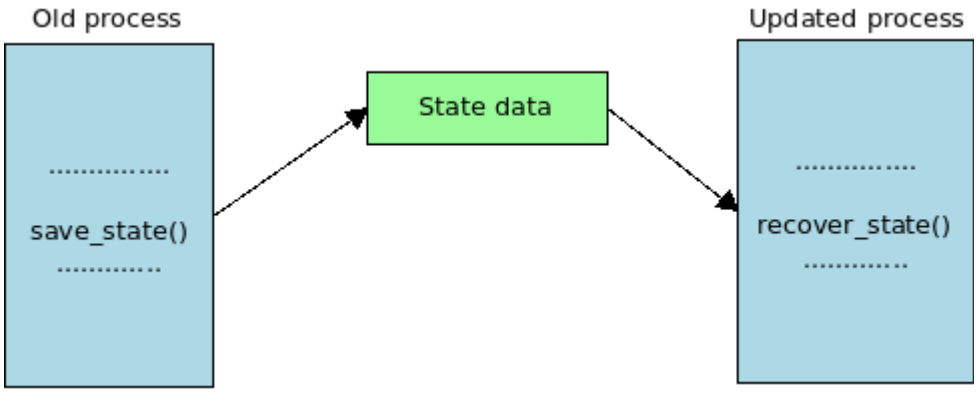

Figure 3: State transfer

In state transfer mechanisms (Figure 3), the current state is copied at the update point and sent to the new version to be restored at its start. Data is then serialized [5] , copied to the heap [3] or address-mapped [6] so it remains available. Developers may be able to decide which variables need to be saved and which are not necessary. Choices can be marked using annotations or tags on the source code [5] or using a specific migration generator [3]. The migration generator plays the same role as the type transformers previously mentioned, allowing any change in the variables in the update code.

Compilation-based approaches require a complex tool to be developed. Optimized compilation is not supported [7], [24], as it causes that executables are less predictable and more difficult to control. Moreover, some tricks are required, such as loop-extraction [4], [6] .

Because of the wide range of different implementations and proposals, it is difficult to

compare their performance and also to establish a common standard for DSU systems [23]. Multiple benchmarks and metrics have been used and Smith, Hoster and Micks believe the standardization of evaluation strategies would help the development and use of DSU systems.

A comparison of some existing DSU systems is shown in Table 3. They have been analysed based on their design key points. As it can be inferred from Table 3, there is no general agreement in any of them. Not even what a DSU should be is defined.

Most of the reviewed systems choose C as a target language due to its large use although there have been some proposals aimed at other common programming languages, such as Java [40] or Python [41].

Support for multithread and even multiprocess programs has also been taken into account. For these programs it is, in general, more difficult to find quiescent points where all the threads can be updated [6]. Therefore, a relaxed consistency model is proposed. Use of barriers [3], [8] is an alternative. It is a mechanism based on the synchronization of threads or processes in the update points. The update system waits until they have all reached a safe point to apply the update.

Table 3: Comparison of existing DSU systems

DSU system	**Ksplice**	**Ekiden**	**Kitsune**	**Upstare**	**Polus**	**Ginseng**	**DYMOS**	**Erlang**	**Approach presented in this Master Thesis**
Operation level	Object level	Code level	Code level	Compilation level	Function level	Compilation level	Module level	Module level	Code level
Programming language	C, Assembly	C,C++	C	C	C	C	StarMod	Erlang	C
Target	OS patching	Full Application	Full Application	Application patching	Application patching	Application patching	Module	Module	Full Application
State transfer	Information provided on patch.	Serialization of annotated data	Migration generator controls copy to the heap	State mapping from stack	Bidirectional state transfer provided by programmer.	State transfer included in patches	Data converter routine provided by programmer	Data indirection	Serialized data
Update points	Automatically inserted	Inserted by programmer	Inserted by programmer	Automatically inserted	___	Inserted by programmer	Inserted by programmer	Inserted by programmer	Inserted by programmer
Multi-thread support	__	No	Yes	Yes	Yes	No	__	___	No
Additional programming work	In case of semantic changes	Yes	Yes	No	No	Not necessary	Yes	Yes	Yes
Implementation	Core kernel including runtime checking, trivial cores for pre-ans postcode and user space	Library	Runtime framework, source- to-source translator, and transformation code generator.	Compiler, runtime, patch generator, and dynamic updating tool.	Patch constructor, patch injector and registration and runtime libraries.	Compiler, patch generator and runtime system	Command interpreter, source code manager, editor, compiler and run-time support system	Programming language	Library

2.5 Security threats

An update process involves a download and acceptance of the new software on the device. Therefore, there is a security risk. Besides, the risk is higher when automatic updates are considered [42], [43]. As a user's role decreases, the power over the system an attacker can get increases. As a result, when designing an update system, is always essential to bear in mind possible threats and to work on the assumption of the Internet as an untrusted infrastructure.

Several security threats and attacks are identified and described. They are sorted into distribution security and acceptance security although these two categories are inherently related.

2.5.1 Update distribution security

When update files are distributed, integrity and authenticity must be ensured so that the update can be trusted by the receiving device. In order to the achieve this goal, update distribution requires a secure communication channel (Figure 4). Secure communication protocols, such as HTTPS, provide the necessary features for a safe and authenticated connection. However, as downloads are more and more distributed through decentralized system networks and mirror servers [13], the update download path becomes longer and therefore, it is more difficult to control and secure. Authenticated binaries are an alternative for them.

Even though common update systems generally include update distribution security features, results of the analysis of some of them have been disappointing [13], [42].

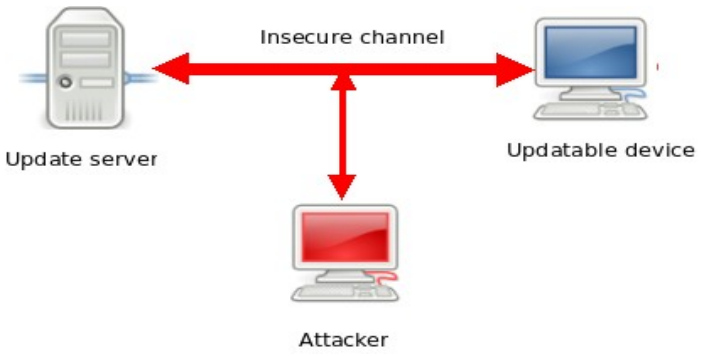

Figure 4: Update over insecure channel

Secure distribution is crucial because it ensures the content of the update files received. Several ways in which attackers can benefit from its failure are the following:

- Freeze attack. Attackers who are profiting from a security vulnerability want a fixing patch to remain unnoticed by users. Update notifications are intercepted or modified with this aim. [14] (Figure 5)

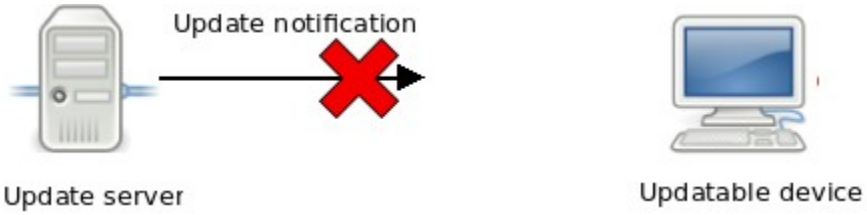

Figure 5: Freeze attack

- Rollback attack or downgrade attack. As the previous one, it also profits from a known security vulnerability and uses altered update notifications. In this case, an updatable device is urged to install an unsafe previous update as if it was a new release (Figure 6). The user thinks the software is been updated while it is actually been rollbacked. The release is totally valid and correctly signed by the provider so it can pass any acceptance check on the device. This attack is sometimes targeted at communication protocols [44], which are forced to establish a less secure connection.

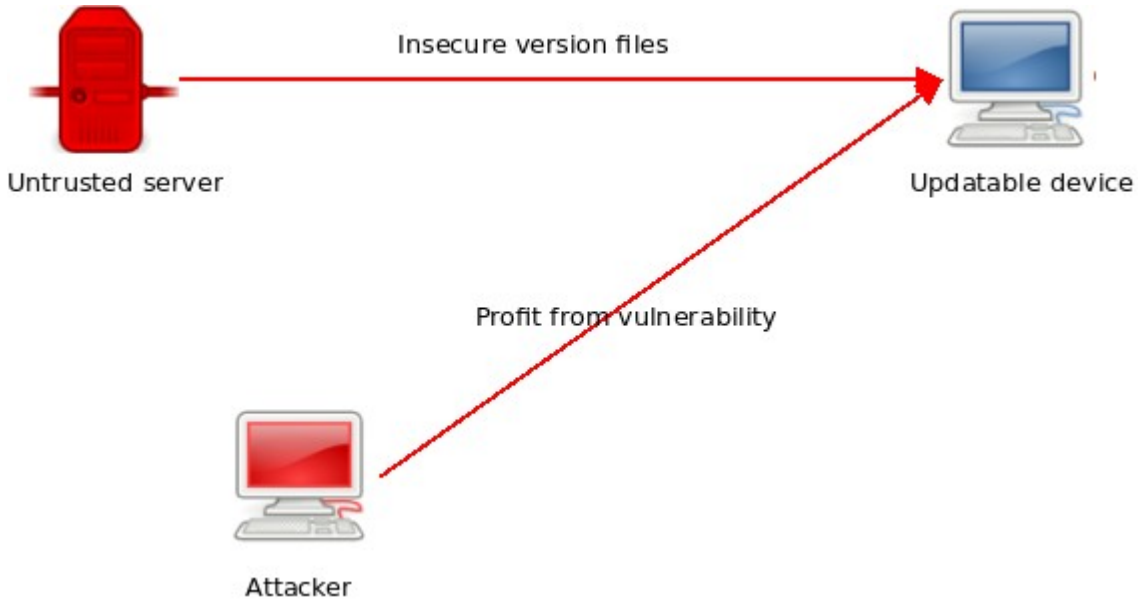

Figure 6: Downgrade attack

- Modification of the update files. Communication packets including the update files are intercepted and modified (Figure 7). Alterations may include malicious software (trojan, spyware), faulty configuration files or code with unexpected

behaviour that compromises the device performance.

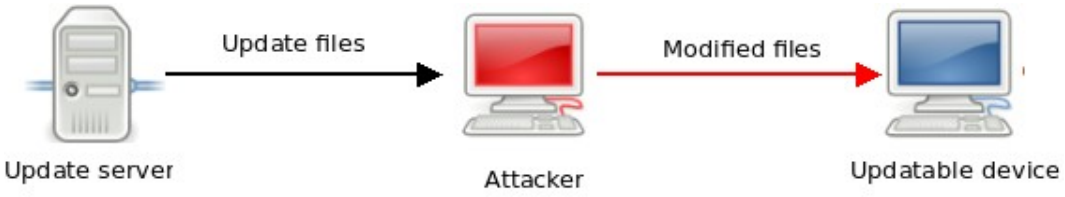

Figure 7: Modification of the update files

Those attacks are generally performed through man-in-the-middle (MITM), DNS poisoning or ARP spoofing schemas [42]. Packet sniffers are employed in order to intercept notifications or data.

Figure 8: Update over wireless network

Only some update-specific attacks have been described. Obviously, any other attack aimed at distribution or communication channels can affect update distribution. This is specially true for wireless networks (Figure 8) as they are more susceptible to attacks and packet losses. Because of their suitability to autonomous devices, the use of wireless communication links in updates is growing and therefore specific security policies must be considered [45]. They include restrictions on the commands that can be executed by the modified software and time limit for updates as well as criteria to choose update timing so normal performance is not compromised.

2.5.2 Update acceptance security

The distribution channel is not the only possible source of security vulnerabilities in the update process. It is pointless to establish a secure distribution channel if no more measures regarding the update source or correctness are taken. End-to-end authentication ensures a trusted update source and diminish the risk of spoofing (impersonation) attacks (Figure 9).

Figure 9: Update from untrusted source

Release code and related information (notifications, metadata) are signed by updater developers with valid private signatures. Their corresponding public signatures are available in the updatable devices to perform necessary checking.

Many update systems implement simple authentication that cannot survive a key compromise [14]. A key compromise takes place when some of the keys are under control of an attacker (Figure 10). Samuel, Mathewson, Cappos and Dingledine propose responsibility separation roles and the multi-trust signature to be implement in a system that would survive the compromise of some keys.

Apart from authentication, acceptance analysis that can be incorporated include checksums and resources and dependencies checking. In that way, update files are proved to be valid, complete and suitable to be executed on the device.

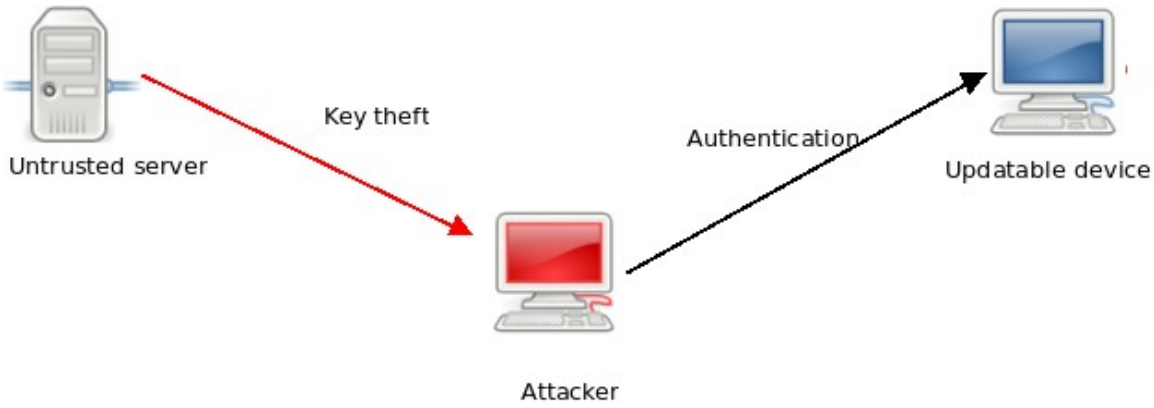

Figure 10: Key compromise

2.6 Update of embedded systems

Embedded systems present autonomy characteristics that complicate the update process. Their small size makes them suitable for controlling every kind of device but it also implies limited resources and access constraints. Due to the lack of graphical interface (GUI) in most of them, they can only be accessed through their network connection, usually wireless. As a result, if the network is compromised there is no other way to enter the system to deal with the attack. As it has been stated, wireless connections are susceptible to be attacked and therefore the device may end up under the attacker's control. That results in a serious threat with home appliances, but it is significantly worse in factory machines or even critical infrastructure systems.

Their limited resources are also a restriction factor as it may be challenging to include advanced cryptographic protocols [13], [46] or any other kind of complex security checks in them. Low computational performance may also result in difficulties in running updaters in the background and they may interfere with the system operation.

The low cost of those systems acts also against the development of secure update systems, as sometimes it is simpler and cheaper to manufacture a new embedded device when a security failure is found than to develop complete analysis and evaluation tools to determine the security risks and measures to be taken [46]. Due to that and to the fact that sometimes they are a part of complex automation process, considerably expensive to stop, many embedded devices are never updated during their life time [36]. That is an additional argument for the suitability of dynamic updates. There where traditional updates have not been successful, they provide an alternative for the update of those systems that work non-stop. In fact, there has been some approaches to dynamic updates specifically designed to embedded systems [36], [47]. Both of them have been designed for real-time proprietary OS instead of choosing the most common Unix-based open-source ones.

Nilsson, Larson and Jonsson [45] provide a guideline for a secure wireless update infrastructure. Even though they do not consider explicitly dynamic updates, main criteria could apply to a dynamically-updatable system.

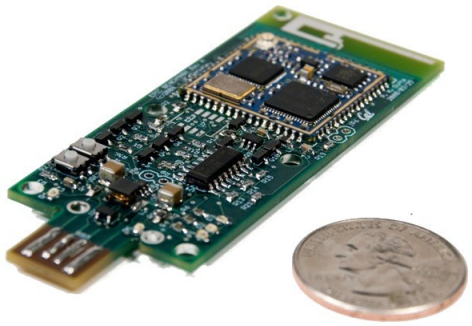

Figure 11: Embedded system

Chapter 3. Technical background

The embedded systems used are single-board computers Raspberry Pi model B, that are going to be used to control a small robot. The robot will be used by University of Oulu students and the aim of the update system is to allow them to write their own source code and test it on the robot. The code will be uploaded to the collaborated online repository GitHub [13] . Despite of these choices, the update system can be used with a different online repository and/or embedded device.

This chapter provides an overview of those two, together with a description of some other technologies also involved in the system.

3.1 Raspberry Pi and Raspbian

Raspberry Pi is a single-board computer developed by the Raspberry Pi Foundation [50].It has been designed to promote basic computer teaching at school and was launched on February 2012. It is a simple and small computer that includes CPU, GPU and 512 Mb RAM user's disk space /bandwidth. Besides, it contains audio, video and HDMI outputs, 2 USB ports and an Ethernet port. It can run complete operating systems. The best known is Raspbian [51] , a Debian-based OS optimized for its software.

Even though the update system has been developed and tested using Raspberry Pi and Raspbian, it can work in different embedded devices running a Unix-based OS.

Figure 12: Raspberry Pi

3.2 Git and GitHub

Git [52] is an open-source distributed revision control. It is a command line tool that allows a flexible source code management. Using the basis provided by git commands, several online repositories have appeared. They are web-based hosting services that permits developers to upload and handle their source code in collaboration with their colleagues. One of them is GitHub [49], that presents social network features so as users

can watch projects and be in touch with other developers, as well as organize team work. Besides, it also provides an API that offers the possibility to integrate its features on developers' applications.

In the update system development, Git and GitHub features are key in the implementation of the update control. Issue delivery for crash report is performed through GitHub API.

3.3 GPG

GPG [53] is a complete and free cryptography system. It is an open source implementation of OpenPGP standard as defined by RFC4880, alternative to PGP. It includes a key management system and tools for encryption and signing.

Updatable repositories require the use of GPG asymmetric key signature, as describe in 4.3.

3.4 Rpcgen and xdr

The external data representation (XDR) [54] is a data serialization standard developed by Sun Microsystems in the 1980's and defined in RFC1014. It allows data representation and transfer independent of computer architecture, operating system and transport layer. It is implemented as a library that allows information encoding/decoding as well as definition of data structures. Due to its portable characteristics, XDR is used in distributed applications through Sun's Open Network Computer (ONC) Remote Procedure Call (RPC) mechanism. As the interface generator of its mechanism, Rpcgen [55] acts as XDR parser.

Data for state transfer in dynamic updates is represented through XDR and parsed using Rpcgen.

3.5 Crontab

Cron is the job-scheduler for Unix operating systems. It manages processes periodically running in the background in order to automate maintenance of administration of the system. Scheduled jobs are defined in the crontab [56].

So as to the update system is executed at fixed intervals to check for updates, it is included in the device crontab with the interval fixed by the user.

Part II. Contribution

Chapter 4. A dynamic update system through collaborative repositories

The approach presented here comprises a complete generic update system. It consists of a client updater, resident on the embedded device that connects to an online repository, where developers have uploaded their software. Some of the features of the update system are already provided by the online repository itself, while the rest are included in the client updater. In order to support dynamic updates, the dynamic_c update library provided must be used by program developers.

Here follows a system description and a characterization according to the update steps defined in Chapter 2, together with a description of its main aspects. Technologies used in the implementation are explained in Chapter 3.

4.1 Update system description

Developers and collaborators involved in a software project upload their code in a collaborative repository online. Once it is ready to be used, it is made available for users. A client updater installed on the device to be updated is responsible for checking when new code can be obtained, for downloading it and for applying the necessary changes to the running software. Roles involved in the update system are shown in Figure 13.

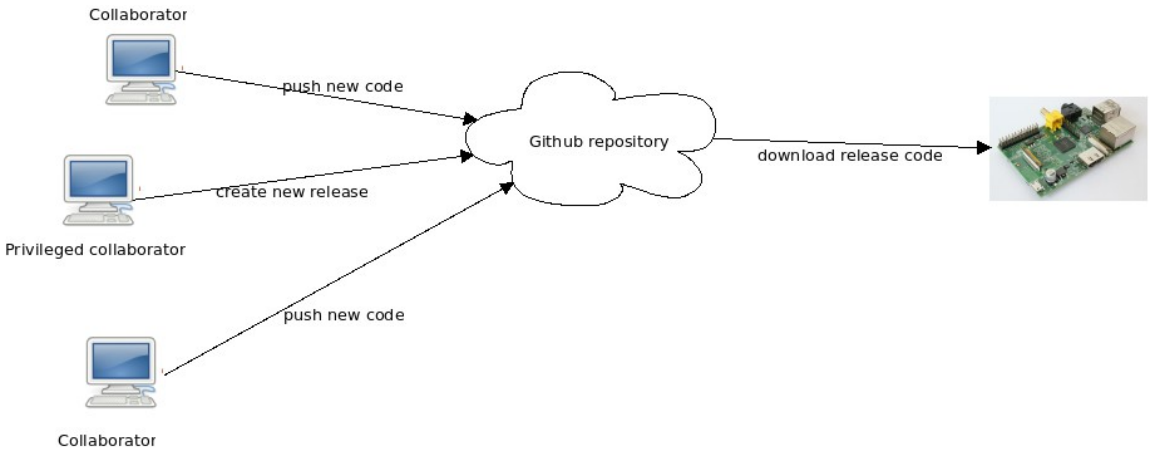

Figure 13: Update system architecture

4.2 Download mechanism

Developers taking part in a collaborative software project commit code that is not always in a working state. It may be to be completed by a different user or just waiting for a revision. As a result, not every modification on the repository code is an update to be downloaded on the device. Furthermore, not every collaborator may be allowed to create a release of software to de downloaded, as described in 4.3.

Receiving unwanted software on the client updater is both a security risk and a misuse of time and resources. In order to avoid that, the download mechanism only accepts a valid available update that has been tagged and signed by authorized collaborators (valid release). In that way, not-ready commits and non-authorized software are immediately rejected. The update client considers there is an update available if there is a valid release posterior to the version currently installed. In the repository state shown in Figure 14 the release v1.1 would be downloaded and the commit with an incomplete microphone code would be ignored. If several valid releases were available the one to be installed would be the most recent. Developers must take this into account as it is not possible to download an update other than the latest.

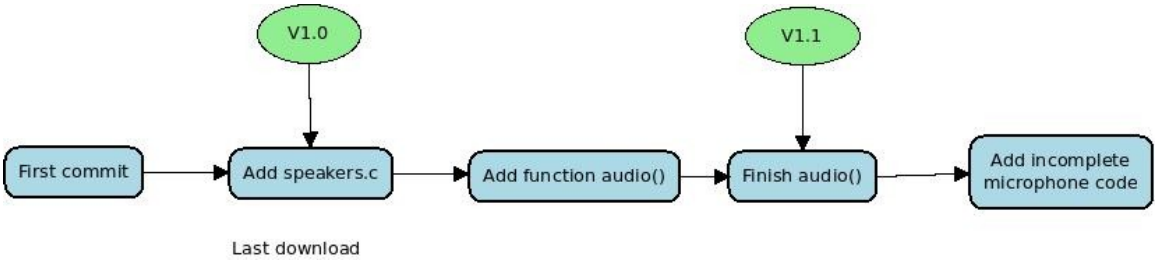

Figure 14: Structure of code in the repository

Once a valid release has been found, the new code is fetched and checked before it is installed. Only the files that differ from the version installed on the device are fetched. Git versioning control system provides features for the changed code delivery. Consequently, there is no need for the updater to include a patch generator or any other procedure to find out what has been.

4.3 Two levels of collaborators

In software projects, a considerable number of collaborators may be involved. As a consequence it may be useful to have two level of collaborators: those who are just cooperating and those who are allowed to create releases that would be actually fetched by devices, as shown in Figure 13. All of them are registered as collaborators on the online repository but only those authorized have their GPG public signature stored on the updatable devices keyring. This key is necessary for the asymmetric key signature explained in 4.5.Public key distribution is performed through public key servers such as *pgp.mit.edu* [48]. Main public keys can also be saved when the update system is installed. Another important role to take into account is the repository administrator, who manages collaborator roles.

4.4 Update procedure

Procedure to update a repository is shown in Figure 15.

4.4.1 Advertise Update/Receive Information

There is no advertise or notification mechanism. Once a new code release is ready it is pushed into the online repository. Using Crontab, the client updater is executed in periodic intervals specified by the administrator on the update configuration file. If no valid code releases are available (see 4.2) it finishes its execution. Otherwise, code is fetched from the repository.

4.4.2 Deliver/Receive Update

Owing to git versioning system, none of the delivery methods discussed in 2.2.2 are necessary in this update system. *Git fetch* provides the tool for easy modified-software retrieval. All modified software in the repository is fetched, election of applications to be updated is not supported. If an specific application is not wanted to be updated it may be selected using git tools, such as .gitignore.

4.4.3 Configure/Reconfigure update

Configuration is characteristic of every program. Thus, it is up to the programmer to include configuration files or any other kind of configuration information. They will be downloaded from the repository together with the code.

Configuration of dynamic updates is usually essential but more complex. Yet, it is still individual to the very program. The dynamic update library tools to deal with configuration are presented in next chapter.

4.4.4 Install/Deploy Update

Once the fetched code is accepted as a valid release, it is merged with the device repository. In principle, the repository in the device is not meant to be modified itself, so as to be always a mirror of the newest release in the online repository. If it was modified and the merge failed, a crash report would be sent to the repository.

The merging overwrites the old files although it can be restored from the online repository in case of there is later some trouble or if a rollback is required by users. Despite of the overwriting performed by the update system, developers could design their software in an incremental way. In other words, the choice of incremental software falls on the programmer and not on the update system.

As the delivery is accomplished by the online repository, verification of completeness and synchronization are not required on the update system. Dependency checking is not applicable either, considering that in case of code from a previous release is necessary, it will be also included on the release or already present on the local repository.

In this stage source code is compiled and tested so as to be ready for execution. If there is

a failure the update is rejected. In that situation, the repository is rollbacked to previous state, previous version files are compiled again and a crash report is sent. Currently, source compilation is only supported for C language (using Makefiles or raw gcc if not provided) but it would be straightforward to incorporate more programming languages support. Testing is carried out with unittest files provided in the repository named with test+name of the program there are testing, e.g. a program called *loop.c* will have a unittest named *test_loop.*c.

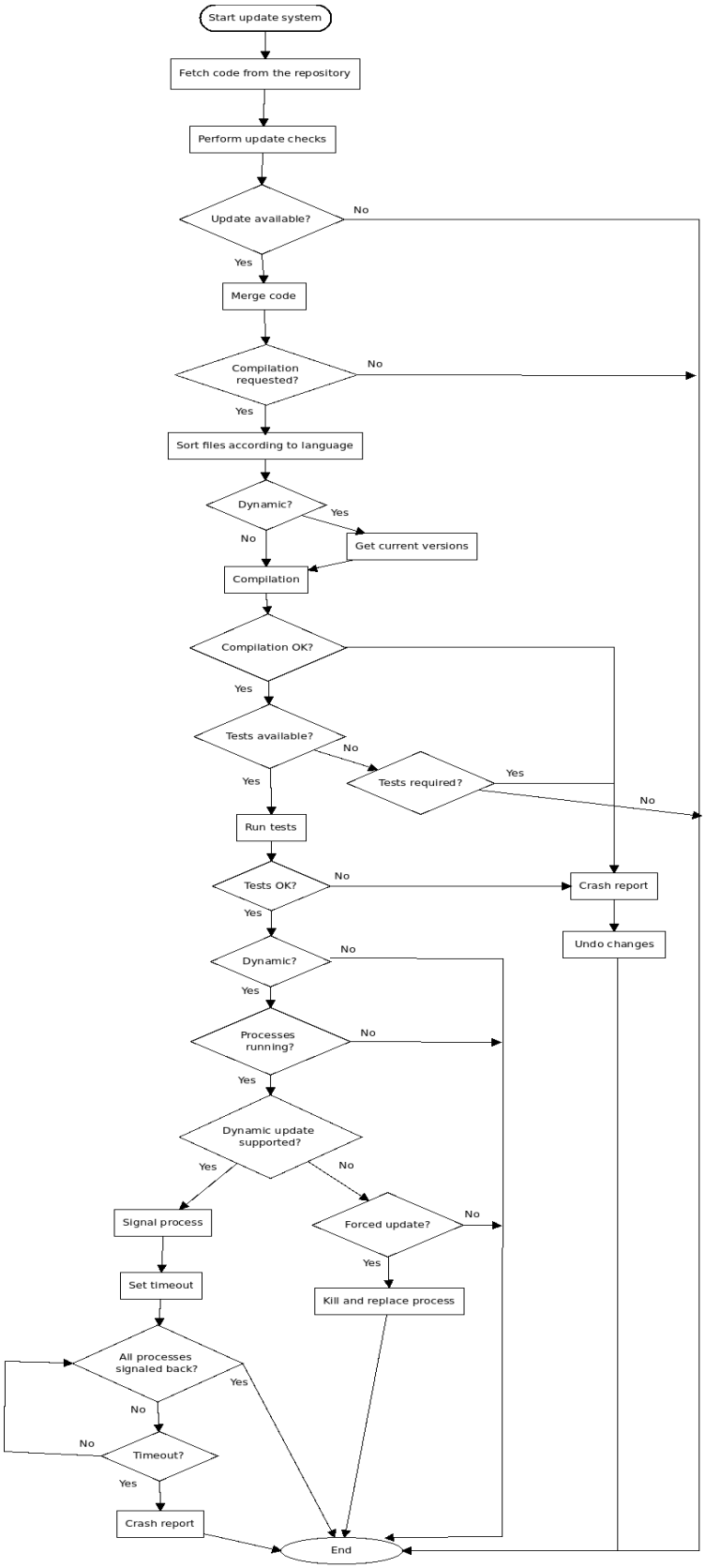

Figure 15: Flowchart of repository update

4.4.5 Apply Update

Several scenarios are considered depending on the current state of the updatable programs and the support for dynamic updates. If dynamic updates are disabled or the process to be substituted is not in execution, the updatable system has finished its task leaving the executable ready for use. If the process is in execution and supports dynamic updates, i.e. it has been implemented using the dynamic update library, the process will be signalled about the prepared update. In case dynamic updates are not supported, the update system won't do anything and will notify of dynamic update failure unless the force_update flag is active. In that case it stops the running process and runs the new one, even though no state recovery can be provided.

Dynamic updates procedure is explained in detail in Chapter 5.

4.4.6 Activate/Deactivate Update

As the update system has been designed for open-source software, this feature has not been considered.

4.4.7 Remove Update

Fetched code that has not proved to be a valid release is removed from the device. However, this action is performed by the system automatically and there is no option for the user to carry it out on request. In certain situation, when the system finishes the update process the user may be able to rollback, as explained in next section.

4.4.8 Rollback/Uninstall Update

Information on last correct version is saved by the update system so as user can rollback the repository to it on request. Nevertheless, rollback can only be performed once. Support for rollback to every previous version would imply saving information on every previous code release installed successfully. There is no support for dynamic updates rollback either, but it could be provided including more library functions. The release we have rollbacked from is considered as not wanted and will not be downloaded again.

4.4.9 Feedback

Feedback from users is accomplished through crash reports, as illustrated in 4.6.

4.4.10 Recovery of tainted state

Knowledge of tainted state of a program means having data about how a certain program should behave. That goes beyond the scope of this update system. Thus, no support is provided for this feature. However, the forced dynamic update (5.5) can be used as a recovery mechanism in some scenarios.

4.5 Security features

Most of the security threats described in the background are related to the content distribution from the repository to the updatable device. Issues such as certificate handling or communication protocols are controlled by the online repository. Even so, the update system provides an authentication mechanism based on asymmetric key signature. Authorized users sign code releases with their private GPG key while their public one is included on the device keyring. So as to a code release is accepted as valid, the signature must be correct and the email address that identifies the creator in the repository must be the same as the one included in the signature used on it. In practice, it represent a two-factor authentication scheme as privileged collaborators have to prove their condition both with their beloging to the repository developers and with a verified according signature. In that way, risk involved in key theft from authorized users (Figure 16) is dimished. Having a secure communication and a selection of trusted code releasers, along with the compilation and tests, constricts the possibility of downloading malicious or unexpected software.

In applications where availability is an essential, dynamic updates present a crucial security feature. Furthermore, the fact that the systems checks itself for updates instead of waiting for notifications makes it not vulnerable to rollback or freeze attacks.

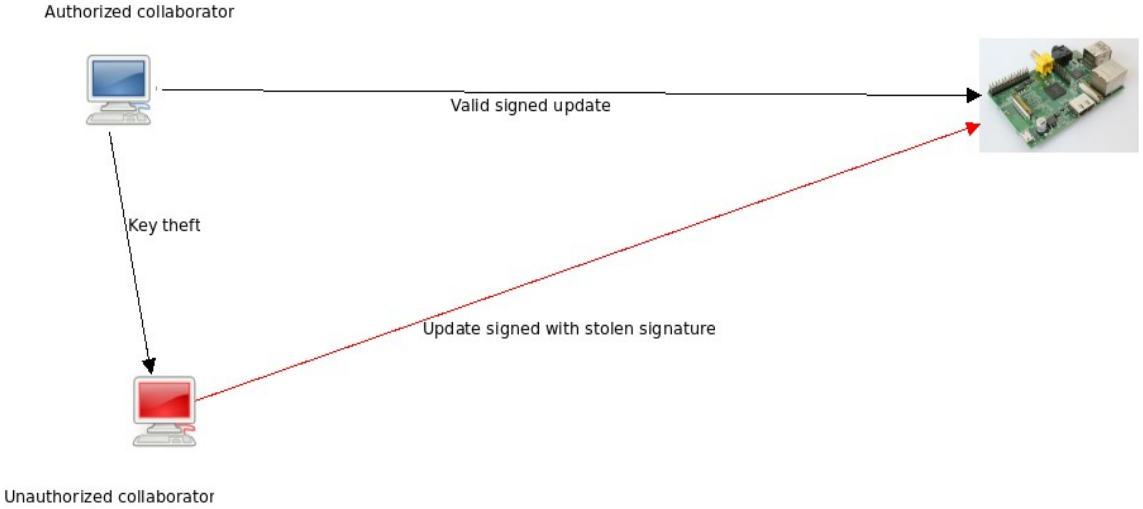

Figure 16: Unauthorized user may use stolen key to deceive the update system

4.6 Crash report

In the event of any failure during the update process, an issue is sent to the online repository. The issue is assigned to the creator of the code release and is tagged with the failure occurred. Besides, it includes the commit number (better than tag name as a tag is likely to comprise several commits) and sometimes more information on the failure, e.g. in case of compilation or test failure, the name of the source file whose compilation has failed is provided. Crash reports have been defined in the following scenarios: not signed or untrusted signature, key theft, compilation failure, test failure, dynamic update failure

and merging conflict. Assignees are then expected to act in consequence. A example of crash report is shown in Figure 17.

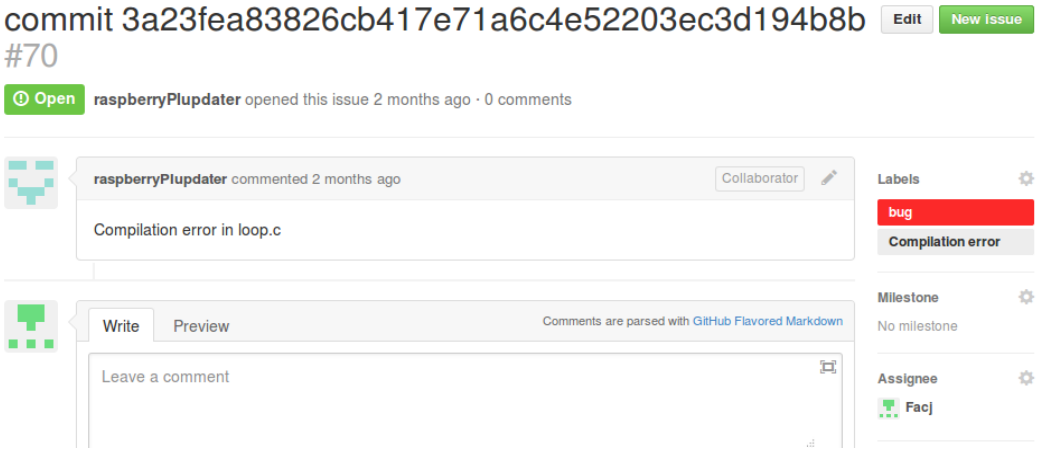

Figure 17: Example of crash report in Github

Whenever a failure different from the dynamic update one is encountered, the update process is aborted. Then the full code is discarded and the release is marked as invalid. In later update checks it is not considered, hence no duplicate issues are received in the online repository in a situation that may lead up to a self-DOS scenario. In case of dynamic update failure, the repository is not rollbacked so as to maintain the changes that may have already been applied to other running processes.

4.7 Configuration of the update system

The update system is highly flexible as most of the features can be disabled on user request. The list of alterable feature contains: compilation check, unittest, key check, force update, dynamic update together with automatic check for updates and its frequency. Like that, the update system can be turned into a manual update system that simply downloads from the repository, without any additional check. Obviously, disabling compilation check also implies disabling the unittest and dynamic update. The system can also be configured to consider unittest indispensable and send crash report if not found and to force dynamic updates even thought they are not supported.

```
Options for disabling features:
-c                      Disable compilation check.
-d                      Disable dynamic update.If the processes are running they
 will not be replaced
-k                      Disable key-authenticity check.
-t                      Disable unittest check.
-a                      Disable all checks.

More options:
-f <minutes>    [m <minutes>] [h <hours>] [d <days>]
Change update-check frequency to the value given.It takes minutes by default.
If the value is 0 automatic update check is disabled.
-r                        Rollback to previous version.
-x                        Force dynamic update, even if not supported.
-s                        Reject the update if not unittest is provided
-h                        Help on the usage
```

Figure 18: Command line options

The inhibition can be permanent or just used in a single execution. In the former case the update system configuration file is modified while in the latter command line options are utilized. Owing to that fact, single-execution update customization is only useful for devices with some kind of user interface. Anyway, every device can benefit from the customization through configuration file.

4.8 Self-update

It would be contradictory to advocate totally automatic and dynamic updates while the update system itself is required to be manually updated. Therefore the update system has been designed to be able to handle its own updates. In order to use always the most recent version, it checks for available update at the beginning of every execution. Similar analysis to the one performed for repository update is accomplished, as shown in Figure 19.

Safety is key factor in this step, as an attack to the update system itself would leave unprotected the security update of any other file. As a consequence, verifications related self-updates (authentication, tests) cannot be disabled.

In systems with no user interface, it may be used to configure the update process. A different configuration file is then included in the update files.

4.9 Connection checking

While developing and testing the system it was found out that on non-activity periods, network connections does not remain available. This can be easily fixed by bringing the (wireless) network interface up. Therefore on the start of the execution network connection is checked on and brought up if necessary so the update retrieval can be performed.

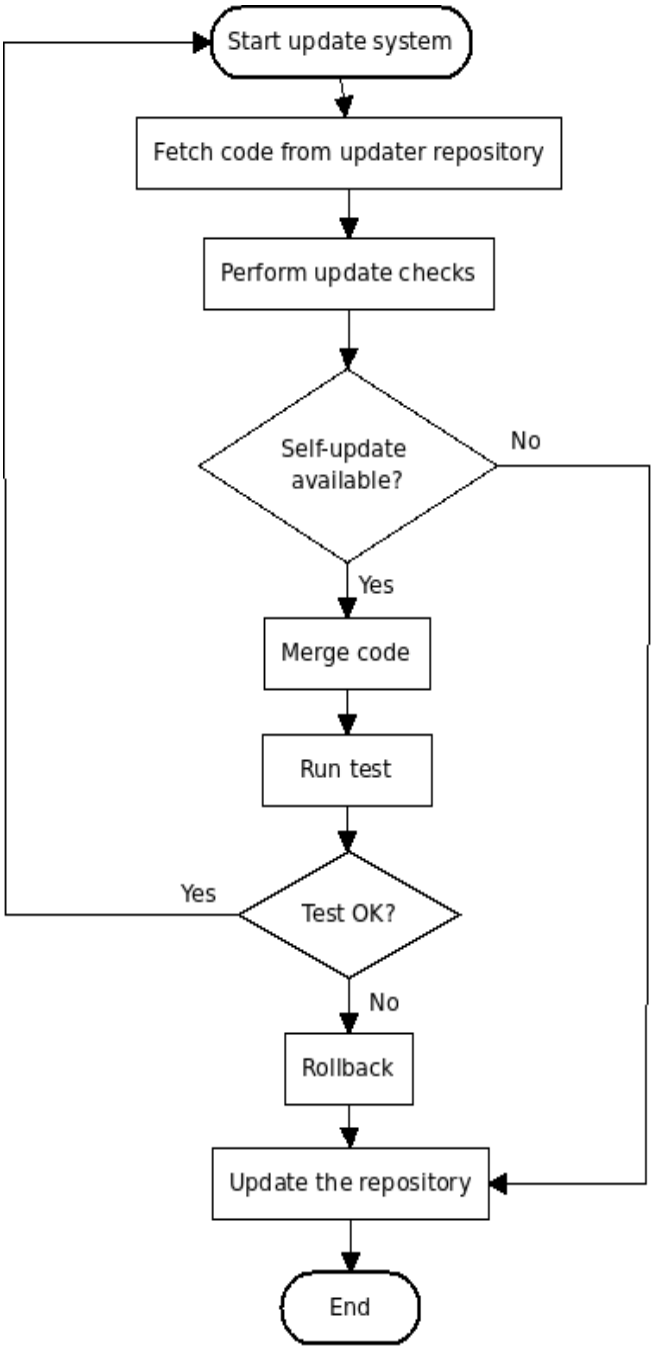

Figure 19: Flowchart focused on self-update

Chapter 5. Updatable development library

In spite of the research carried out over the last three decades, dynamic updates use is not widespread and no complete update system has been developed before [49]. Reviewed DSU approaches were tested on manual request. That is to say, no mechanism combining dynamic and automatic updates has been implemented. As a result, some of issues arisen during the design of the update system had not been studied in the referenced papers.

5.1 DSU library

The DSU approach presented here consists of a C library and the necessary handling functions in the update system script. Variables and function in the library are shown in Table 4 and Table 5 respectively. Only programs using library features can benefit from the mechanism.

A demonstration of usage of the library is described and illustrated in Chapter 6. The support is checked by the update client, using output obtained from running the compiled program with the --version option. If the option is not supported, the program is considered not dynamically updatable.

Table 4: Variables in the DSU library

Variable	**Description**
struct update_variables	Struct containing the variables to be serialized. Included variables are shown in italics.
int update_available	Flag that indicates if an update notification has been received
int update_in_progress	Flag that indicates if the running process is a dynamic update
int updated_from	Identifier of the update point where the previous version stopped its execution
int old_version_pid	Pid of the running process to be replaced
int up_system_pid	Pid of the update agent
int update_successful	Flag to indicare whether the new version execution has been successful or not.
char PROGRAM_NAME[100]	Name to be used to identify the serilization files

When an update is available, the program receives a SIGUSR2 signal. The signal handling function, included in the library, sets the *update_available* flag. When an update point is reached, this flag is checked and if set, the dynamic updates starts.

Firstly, a struct containing all data necessary for the program is serialized and saved in a .ser file.

Secondly, the update point number is saved in the *update_variables* struct and also serialized.

Finally, a new process is started and if its execution is correct, the old one dies.

Table 5: Functions in the DSU library

Function	**Behaviour**
void check_update_status()	It initializes the update status and sets the signal handler for SIGUSR2.
void save_update_status()	It saves the update variables status to be used by the new version.
int save_data(void *data)	Serializes the process data necessary for the execution of the new version. (To be implemented by developer)
void *restore_data(void *data)	Recovers the process data and modifies it if necessary. (To be implemented by developer)
void *update_point(int up_id, void **data)	Used as marker of safe update points. If an update is available, prepares its execution and waits for its notification.

5.2 Update points

Update points are included as a call to a function of the same name included in the library. When reached the following scenarios are considered:

- Update in process: Current execution is a update of a program that was running previously and that continues from this point on. Data from the old version is restored and flags are changed accordingly.
- Update available: Current execution is a program for which an update is now available and ready to be executed. Data from the program and update flags are

saved and a new process is launched. If it runs smoothly, output of the function indicates the program to finish its execution. In case no notification is received from the new process, the process continues as if nothing had happened and update is considered failed. Consequently, the update system is notified. All the notifications are implemented using the SIGUSR2 system signal.

- None of the above: nothing happens.

When an update is running, several if/else control the flow until the right update point is reached. This is rather laborious if many update points are inserted but ensures no inappropriate code is executed.

5.3 State transfer

So as the new process can neatly substitute the old one, it needs to be provided the values of variables the previous one was working with. Furthermore, update point number and flags containing information on the update status have to be also shared.

The new version is launched as a different process, totally independent from the old one and therefore, an interprocess sharing mechanism is required. Communication between them would be simpler if it was forked as a child. However, that would cause the process stay as a zombie in the system if the previous version does not wait for its end. The system could end up full of processes waiting for new processes that are, in turn, waiting for newer processes. Several interprocess communication system tools were considered, including message queues, shared memory and sockets. However, they do not allow to build a communication mechanism independent of the message format. As a result, serialization was chosen instead.

Every variable in the program whose value needs to be stored (local variables may not be significant) has to be declared inside a struct in the structures.x file or in a different .x file. This file is used by Rpcgen to create serialization/deserialization functions for them, to be called to save/restore data. Any kind of variable can be included, except for pointers. However, *char* * pointers can be declared as strings and serialized. Structures.x is already created and contains the definition of the *update_variables* struct. It is created or deserialized at the beginning of every process so the variables can be used during the whole execution. So as a process can know what file is to be deserialized they are given specific names depending on the program name. Considering the previous example of a program called loop.c, the *update_variables* struct is serialised in *up_loop.ser* and the struct containing the process variables in *pr_loop.ser*. If the serialized file does not exist or cannot be accessed, the update is considered failed.

Whenever variables or types are modified in an update, the new data struct is also provided in the .x file. It cannot have the same name as the the old name. Common renaming pattern consists of adding the version number to the data struct. Conversions between the old and new data may be necessary. Developers include them in the *restore_data* function.

5.4 Crash report

Crash report is one of the issues not considered in the DSU literature, although Ginseng's creators mention the lack of a safe mechanism to handle failures [50].

The aim is to treat this failure as any other of the considered in 4.6. However, due to the nature of dynamic updates this is not straightforward. Error analysis is performed by the library and not by the update client. Besides the error handling takes place at the update point and at the start of the new version, which may not coincide with the execution of the update client. Owing to the time consumed before reaching the update point, the update client may have already finished its execution and the failed program would have nobody to notify and could not send the issue itself. Furthermore, in applications with high-availability requirements the failure of the update when the old process has already died would be fatal.

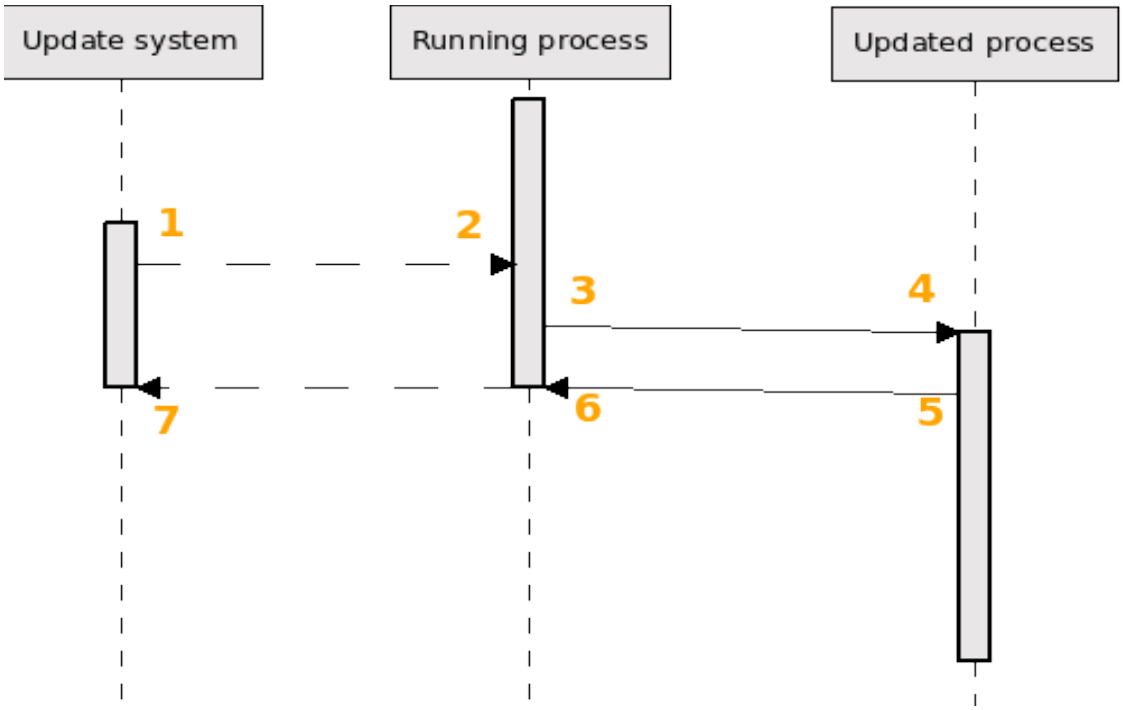

Figure 20: Dynamic update sequence

1) Notification of available update

2) Wait until update point

3) Prepare new version execution and launch the process

4) Restore date and starts the execution on the provided update point

5) Notification of success to the previous version.

6)Notification of success to the update system.

In the solution implemented the old version process does not stop on the update point, but waits until the new version process notifies its correct start. If no signal is received after a certain time the old process continues its execution as if the stop had never taken place. Regarding the crash report, another timeout mechanism is proposed. Timeout value can be specified in the configuration file. This is not an ideal solution as it restricts the update time, which cannot be know beforehand. Nevertheless, it is considered the least bad option as notification of failure is a basic security and feedback feature. Dynamic update sequence is shown in Figure 20.

In case non-dynamic update failure, the complete release is rejected and the repository is rollbacked. In case of failure of the dynamic update, the rollback would imply changing other processes already updated and running. Assuming independence of the processes, no consistency problems appear if some of the applications running have been updated and others have not.

When an update requires the dynamic modification of more than one process, the bash scripts waits until it receives as many signals as processes have to be replaced. Regrettably, the update client is not able to know which of the processes has failed as bash scripts lack tools to do so. Updating processes one by one would solve this problem yet the update process would be excessively slow. An alternative would be starting C processes in charge of the signal handling and the timeout wait. Their output would indicate whether the update was successful or not.

5.5 Forced update

If a program hangs and or for any reason never reaches the update point, the update can be forced. In that case, the process is simply killed and a new one is launched. No state transfer or update points can be considered. Assuming knowledge of the program hanging or other incorrect state, a forced update can be used as a recovery method. This alternative may also be used for the dynamic update of simple programs that do not require data exchange nor update points.

5.6 Version compatibility

Building a complete dynamic update system raises the issue of what to do when the updatable device has been offline for a while and the update system finds several updates available for download. In case of non-dynamic update, the most recent one is downloaded. In case of dynamic update, the new version knows how to convert data from the previous version but not from any other so the update will fail. Several solutions were studied:

- Including conversion for any previous version may be an excessive and often unnecessary work for the developer.
- Applying the updates one by one implies stopping the program several times and the resulting overhead.
- Download the update one by one but not substitute the process, just convert the data. This is implemented using a different signal (for instance SIGUSR1) in order to indicate the process to stop but not to start the new version. An option on

the new version is also required, so its execution consists only of converting and then serializing again the data. That would reduce the overhead as the process is not required to wait for the new one to notify its correct start. However, the process is still stopped when the first update is downloaded to keep data consistency. Furthermore, timeout crash report mechanism needs to be adapted.

- Building a version compatibility information system. Provided that previous versions have not involved data changes or those are minimal, conversion function provided can be applicable to update from several previous versions. Information on versions compatible for conversion appears on the output of the version option. If the running version is not compatible, the update fails and a crash report is sent.

The latter solution provides the most reliable alternative as it ensures data compatibility without increasing overhead. Even though update application is not always guaranteed, that scenario is unlikely to occur as updatable devices are thought to be always connected. One-by-one update solutions require more complex mechanisms and may imply downloading, and therefore reporting, defective releases that have already been fixed by following ones.

Part III. Example & Conclusion

Chapter 6. Example of usage

This chapter presents the procedure to use the update system. Explanations on how to build an updatable program and demonstration of the system's usage are provided.

6.1 Building an updatable program

So as to make the most of the DSU system, programmers must adapt their code and provide the files in the following description.

6.1.1 Files to be included

- Structures.x file. Definition of structs of data of the new and the old version, together with the one containing update variables, already provided. Structs can also be declared in a different .x file.
- Makefile. As well as compiling instructions it includes rpcgen statements to create the serialization functions. Any other library to link can be inserted.
- Unittest. Including the test considered by the developer.

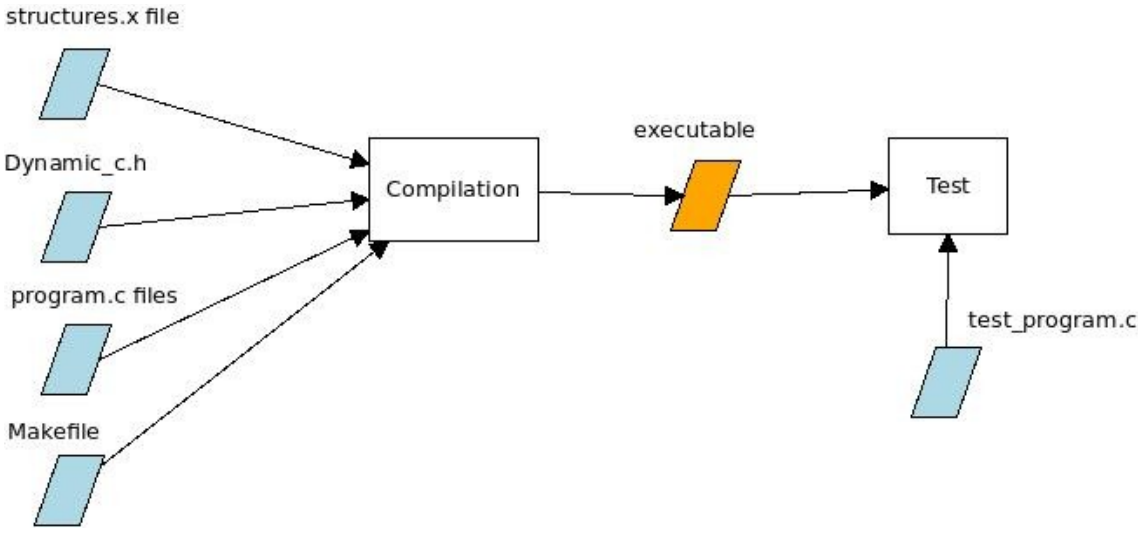

Figure 21: Optimal use of the update system

6.1.2 Code modifications

So as to obtain an updatable program, source code has to be modified:

- Implementation of the option –version so it can be recognised as updatable.
- Library headers: *Structures.c* and *dynamic_c.h* have to included in the source code.
- Initialization of the variable PROGRAM_NAME after which serialization files will be named.
- Update status checking at the beginning of the program. It is performed using the function of the same name implemented in the library.
- Update points and flow control with if statements.
- Implementation of functions *save_data* and *restore_data.* If no modifications to the data are required, the functions include just the serialization/deserialization.

The last three modifications are implemented using the library functions (Table 5). Figure 22 presents an example of simple updatable program with three update points.

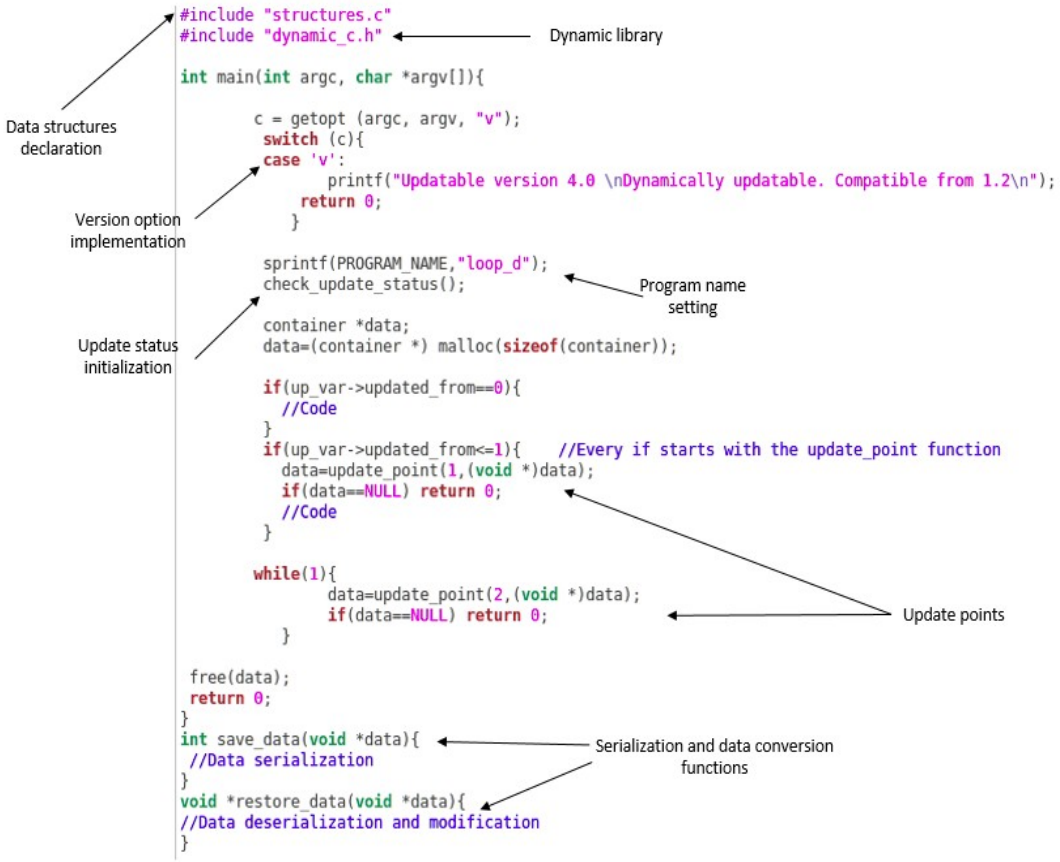

Figure 22: Updatable code

6.2 Demonstration

The program used for the demonstration is a simple C program that gets user name from command line and then writes its version number, the date, the user's name and the number of times it has been executed in the file *version_record.txt* every 5 seconds, as shown in Figure 24. For demonstration purposes, the updater is manually executed in all the scenarios below.

With the -v option we can check it the program is updatable, as shown in Figure 23.

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Repo1$ ./loop_d -v
Updatable version 4.2 extended
Dynamically updatable. Compatible from 1.2
```

Figure 23: Output of the version option

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Repo1$ ./loop_d fatima &
[1] 16540
fatima@fatima-Satellite-L500:~/Raspi_sw/Repo1$ more version_record.txt
Version 4.2 .Executed on fatima's device 1.000000 times on Fri May  2 16:13:46 2014
 Version 4.2 .Executed on fatima's device 2.000000 times on Fri May  2 16:13:51 2014
 Version 4.2 .Executed on fatima's device 3.000000 times on Fri May  2 16:13:56 2014
 Version 4.2 .Executed on fatima's device 4.000000 times on Fri May  2 16:14:01 2014
 Version 4.2 .Executed on fatima's device 5.000000 times on Fri May  2 16:14:06 2014
 Version 4.2 .Executed on fatima's device 6.000000 times on Fri May  2 16:14:11 2014
 Version 4.2 .Executed on fatima's device 7.000000 times on Fri May  2 16:14:16 2014
 Version 4.2 .Executed on fatima's device 8.000000 times on Fri May  2 16:14:21 2014
 Version 4.2 .Executed on fatima's device 9.000000 times on Fri May  2 16:14:26 2014
 Version 4.2 .Executed on fatima's device 10.000000 times on Fri May  2 16:14:31 2014
 Version 4.2 .Executed on fatima's device 11.000000 times on Fri May  2 16:14:36 2014
```

Figure 24: Output of the first version of the program

6.2.1 Dynamic update

Once the developer realize it makes no sense to use a float type to count the executions it changes the source code and uploads it to the online repository. On its next execution, the update system running on the device finds the available update (Figure 25). As we can see in Figure 26, the process was successfully replaced and the user's name and the number of executions were correctly sent to the new versions.

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh
SOFTWARE UPDATER  4.19
Fri May  2 17:27:45 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
Waiting for updated processes...
Dynamic update successful
```

Figure 25: Output of successful updater execution

In this screenshot, if we can figure out the update applying overhead by comparing the last printing by the old version and the first one by the updated one, that were on 17:58:45 and 17:48:51 respectively. That means just a second of overhead if we compare with a normal execution.

```
Version 4.2 .Executed on fatima's device 8.000000 times on Fri May  2 17:58:15 2014
Version 4.2 .Executed on fatima's device 9.000000 times on Fri May  2 17:58:20 2014
Version 4.2 .Executed on fatima's device 10.000000 times on Fri May  2 17:58:25 2014
Version 4.2 .Executed on fatima's device 11.000000 times on Fri May  2 17:58:30 2014
Version 4.2 .Executed on fatima's device 12.000000 times on Fri May  2 17:58:35 2014
Version 4.2 .Executed on fatima's device 13.000000 times on Fri May  2 17:58:40 2014
Version 4.2 .Executed on fatima's device 14.000000 times on Fri May  2 17:58:45 2014
Version 4.3  .Executed on fatima's device 15 times on Fri May  2 17:58:51 2014
Version 4.3  .Executed on fatima's device 16 times on Fri May  2 17:58:56 2014
```

Figure 26: Updated version output

6.2.2 Dynamic update not compatible

In case the device has been offline for a while and once online again, the update release may be not compatible with the running version of the program (Figure 27).

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Repo1$ ./loop_d -v
Updatable version 4.12 extended
Dynamically updatable. Compatible from 4.11
```

Figure 27: Output of the version option of non-compatible release

Then the updater could not apply the dynamic update (Figure 28).In that case we can choose to force the update, with the -x option (Figure 29).

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh
SOFTWARE UPDATER  4.19
Sat May  3 09:54:31 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
Dynamic update not supported
```

Figure 28: Output of updater when dynamic update not supported

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh -x
SOFTWARE UPDATER  4.20
Sat May  3 10:01:12 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
Dynamic update not supported
Forced update
```

Figure 29: Output of updater when forced update

Then, there is not state transfer and therefore the number of executions and the user's name are reinitialized (Figure 30).

```
Version 4.3  .Executed on fatima's device 16 times on Sat May  3 10:01:02 2014
Version 4.3  .Executed on fatima's device 17 times on Sat May  3 10:01:07 2014
Version 4.3  .Executed on fatima's device 18 times on Sat May  3 10:01:12 2014
Version 4.3  .Executed on fatima's device 19 times on Sat May  3 10:01:17 2014
Version 4.12 .Executed on user's device 1 times on Sat May  3 10:01:27 2014
Version 4.12 .Executed on user's device 2 times on Sat May  3 10:01:32 2014
Version 4.12 .Executed on user's device 3 times on Sat May  3 10:01:37 2014
Version 4.12 .Executed on user's device 4 times on Sat May  3 10:01:42 2014
```

Figure 30: No state transfer after forced dynamic update

6.2.3 Self-update

Once a valid self-update is found, the update system restarts before proceeding the update of the repository. In the case presented in the figure, the updated processes are not running, so dynamic update cannot be applied. This is the scenario shown in Figure 31.

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh
SOFTWARE UPDATER  4.19
Sat May  3 09:59:15 EEST 2014
Establishing internet connection...
Internet connection ready
Self-updating...
SOFTWARE UPDATER  4.20
Sat May  3 09:59:23 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
The updated processes are not running
```

Figure 31: Self-update of the update agent

6.2.4 Signature attack

The updater rejects a release if has not been signed or the signature is not on the trusted key ring (Figure 32) as well as in case of key theft. Figure 35 presents an example of security report sent to Github.

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh
SOFTWARE UPDATER  4.20
Sat May  3 10:16:56 EEST 2014
Establishing internet connection...
Internet connection ready
Not signed or untrusted signature
No update available
```

Figure 32: Untrusted update release

If key check is disabled the update is applied (Figure 33) but then the update process wouldn't be secure and security problems may be experienced (Figure 34).

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh -k
SOFTWARE UPDATER  4.20
Sat May  3 10:29:40 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
Waiting for updated processes...
Dynamic update successful
```

Figure 33: Disabled key check

```
 Version 4.4  .Executed on fatima's device 90 times on Sat May  3 10:29:34 2014
 Version 4.4  .Executed on fatima's device 91 times on Sat May  3 10:29:39 2014
 Version 4.4  .Executed on fatima's device 92 times on Sat May  3 10:29:44 2014
 SYSTEM IS BEING ATTACKED Sat May  3 10:29:53 2014
SYSTEM IS BEING ATTACKED Sat May  3 10:29:58 2014
SYSTEM IS BEING ATTACKED Sat May  3 10:30:03 2014
SYSTEM IS BEING ATTACKED Sat May  3 10:30:08 2014
SYSTEM IS BEING ATTACKED Sat May  3 10:30:13 2014
SYSTEM IS BEING ATTACKED Sat May  3 10:30:18 2014
```

Figure 34: Security issues due to unsafe update

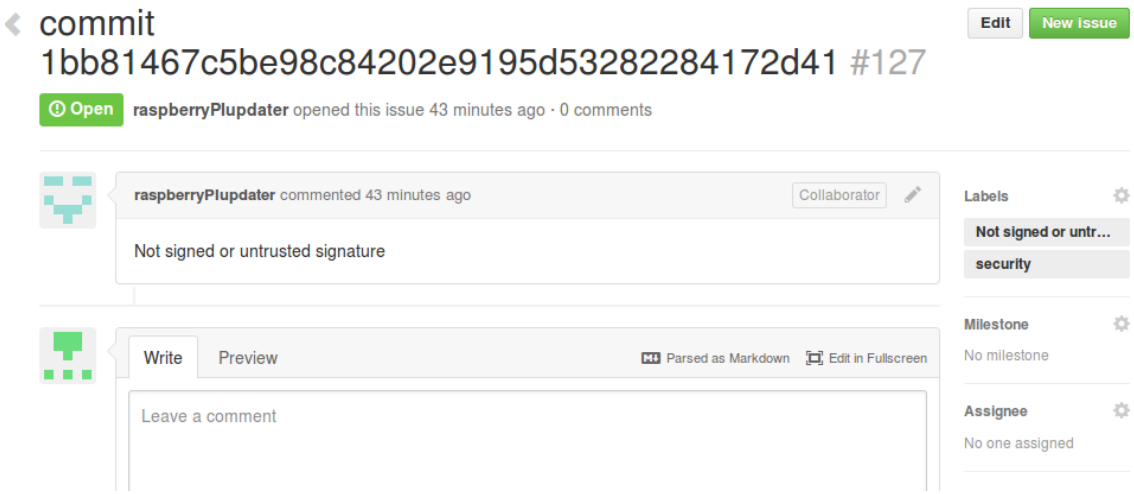

Figure 35: Example of security crash report

6.2.5 Unsuccessful dynamic update

In case a problem occurs while applying the dynamic update both the update agent and the old version process are not signaled. After a timeout (in this case is fixed to 10 seconds), the update agent considers the update failed (Figure 36).

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh
SOFTWARE UPDATER  4.20
Sat May  3 10:45:49 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
Waiting for updated processes...
update_failed Dynamic update failure
```

Figure 36: Dynamic update failure

And the old version process continues its execution as if nothing had happened. The only evidence of the failed update is the time difference between the executions while the timeout. As we can see in the screenshot in Figure 37, there was no printing from 10:45:57 to 10:46:09. That is a 12 seconds interval, which includes the serialization/deserialization, the timeout and the necessary checks as well as a possible wait due to the execution of sleep when the update was found.

```
Version 4.1 .Executed on fatima's device 7 times on Sat May  3 10:45:42 2014
Version 4.1 .Executed on fatima's device 8 times on Sat May  3 10:45:47 2014
Version 4.1 .Executed on fatima's device 9 times on Sat May  3 10:45:52 2014
Version 4.1 .Executed on fatima's device 10 times on Sat May  3 10:45:57 2014
Version 4.1 .Executed on fatima's device 11 times on Sat May  3 10:46:09 2014
Version 4.1 .Executed on fatima's device 12 times on Sat May  3 10:46:14 2014
Version 4.1 .Executed on fatima's device 13 times on Sat May  3 10:46:19 2014
Version 4.1 .Executed on fatima's device 14 times on Sat May  3 10:46:24 2014
Version 4.1 .Executed on fatima's device 15 times on Sat May  3 10:46:29 2014
Version 4.1 .Executed on fatima's device 16 times on Sat May  3 10:46:34 2014
```

Figure 37: In case of update failure, the old version continues its execution

6.2.6 Compilation failure

Failure of the compilation check of a single file results in update failure and rejection of the release, as shown in Figure 38.

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh
SOFTWARE UPDATER  4.20
Sat May  3 10:22:50 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
loop_d.c compile failed
Compilation error
```

Figure 38: Compilation failure

6.2.7 Test failed

Test failure of one of the updated files results in update failure (Figure 39) and therefore an issue is sent to the online repository (Figure 40).

```
fatima@fatima-Satellite-L500:~/Raspi_sw/Update_sw$ bash update.sh
SOFTWARE UPDATER  4.20
Sat May  3 10:57:15 EEST 2014
Establishing internet connection...
Internet connection ready
Available update
Checking updated files...
Compiling...
Test for loop_d failed
```

Figure 39: Test failure

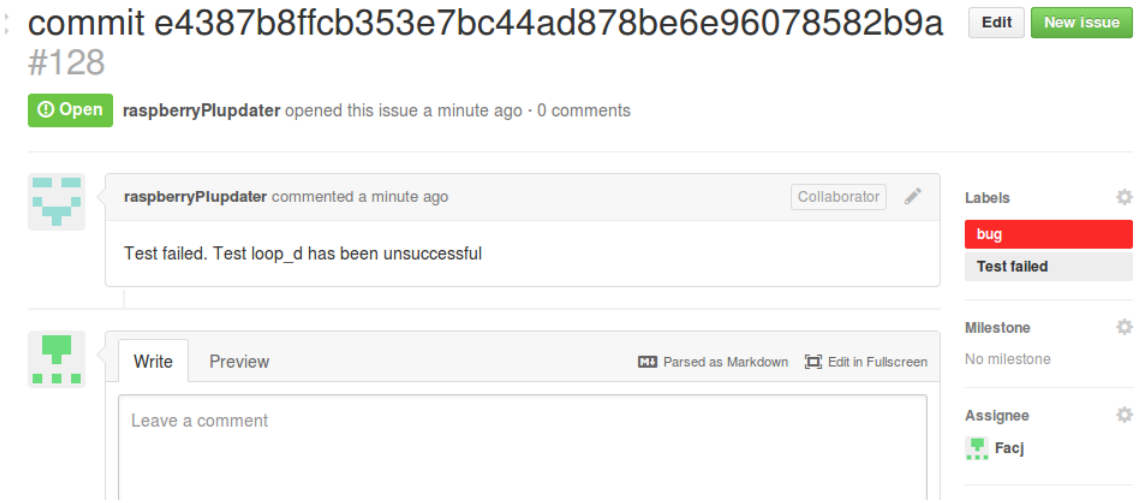

Figure 40: Example of test failure crash report

Chapter 7. Conclusion and future work

A review of software updaters shows reveals a trend towards automation. Security being provided, this is good news both for users and developers. For the former, it means straightforwardness in their software maintenance. For the latter, it involves a more predictable update model to work on.

The update process has been analysed in detail with a special focus on dynamic applying. Main concerns and approaches related to its development have been discussed. A complete update system has been thereon implemented and demonstrated. Demonstration can be regarded as a guide of usage for developers. Description and discussion of design and features have also been enclosed.

The system has been targeted at embedded devices as their inherent autonomy characteristics make them require an approach different to those used in traditional update schemes. Automation and self-update features are valuable tools for them.

On the whole, this Master Thesis intends to provide an usable tool that brings together some of the current issues related to update development. It represents a step towards the adoption of dynamic updates and serves as example of the potential of embedded devices and collaborative software.

7.1 Discussion

The system is complete because it covers every step of the update process. Because of that, it goes beyond previous DSU systems and represents an advance in the spread of dynamic updates. In fact, apart from Ksplice kernel updater [7], no DSU has been used out of academic scenarios.

Table 2 shows a comparison with proposed DSU systems. The approach is similar to those in Kitsune [3] and Ekiden [5] as they are also based on functions provided to developers. However, their methods to annotate variables to be transferred to the updated version are more complex. Kitsune's one even requires an specific compilator-like tool to create the annotations. The struct-based state transfer method is simpler and usable. Moreover, as it does not need annotations in the code it diminishes the demand for code modifications and eases the code lecture.

The use of a library may seem to be cumbersome over the DSU proposals based on compilation tools. Nevertheless, it gives developers more control of the update process, making it more predictable. There is no reason why dynamic update cannot be implemented in a library as any other functionality.

Performance cannot be compared with other DSU systems because only simple tests have been carried out. In spite of that, this proposal should be enough for updating larger applications. Even if more complex demonstrations had been taken, it would have been

difficult to contrast, as it has been stated in previous work [23].

Together with dynamic updates, the system has been based on availability and flexibility. The first is crucial since in every case where dynamic updates are particularly useful that is due to the availability requirements of the application. As a consequence, it is counterproductive if the update systems brings into work disruption. Even though this Master Thesis advocates totally automatic update processes as long as security features are included, it is also true that there is not a good solution for every circumstance. Consequently, many features have been made flexible so as to be able to adapt to the update conditions.

Considering security, it is mostly built on the online repository distribution even if the two-level-collaborators scheme supplies extra protection. This should not been regarded as a lack of the system but as a practical issue. It is not necessary to implement again working features that can be easily integrated.

The fact it has been designed for an Unix-based OS makes the system portable since the Unix software family is the most common one. As a result, the same update system can be used in different devices part of the same project or distributed network. Or in every device owned by an user.

It has been targeted at C language not only because of its spread, but also for its common use in embedded systems as it is more adaptable to simple architectures. Undoubtedly, the more programming languages are supported the more practical the systems. Following the C example, libraries for every language can be implemented.

7.2 Future work

Main components of the proposal in this Master Thesis (dynamic updates, embedded devices and collaborative software) will certainly play an important role in the future of software updates. However, there is still work ahead to consider this approach a finalized and completely usable mechanism. Some possible improvements and alternatives have been discussed in the previous chapter and here follows a description of several lines of future work in the software update field.

7.2.1 DSU spread

As next step in the automation trend to assist users in the maintenance of their software, dynamic updates are bound to change the field of dynamic updates. However it is still unclear when that will happen. The study of DSU systems started decades ago and yet it has not advanced enough to allow the adoption in common update mechanisms. Valid DSU schemes have already been presented but more research is needed to accomplish their effectiveness in other scenarios, particularly those involving coordination between processes or applications.

Besides, technology adjustment is also required. On the hand, some existing tools can be

employed. For example, mobile phones and other devices include hibernation or power saving modes on which working state is saved. This very capability could be exploited for state transfer. On the other hand, more preparation, such as OS support, is needed. Operating systems should be aware of these updates as it affects parameters they are working with, such as processes' pids.

7.2.2 Embedded devices

Due its growing influence on multiple applications, embedded systems are a major target for software developer. The implementation of the Internet of Things will promote the spread of those device and therefore, the amount of software required. As a fundamental part of software life cycle, updates for them are to be considered. Considering their autonomy characteristics automation is a must for many of them. In that way, the development of dynamic updaters involves a good start point for the massive development of embedded updater. Moreover it reinforces their autonomy.

Due to its inexpensive manufacture, embedded devices are sometimes designed to be thrown away as soon as their features no longer meet the user's requirements. Totally functional devices may end up in electronic waste because they cannot be updated, which implies an considerable environmental impact. Even when users are aware they cannot avoid the situation because, as it has been mentioned earlier, updates are not just a craving for something new but also a crucial security issue. Thus, users have to evaluate the trade-off between security and environmental consciousness. As a result, it can be stated that embedded updates are environmentally-friendly since they allow users to keep the same hardware for longer periods of time.

7.2.3 Distributed updates

When considering the update of embedded devices, we should bear in mind the fact they are often designed to work collectively. Sensor networks are one example of that. On those scenarios is important to keep version consistency between devices so as the collaboration is always successful. In order to accomplish date, mechanisms similar to those used in multithread update can be implemented. An alternative to those may be connecting just one of them to the Internet and build a distribution network from it. In that case, this device would act as central node and would control the status of the software on all the others. Distribution protocols for sensor networks have already been considered [57].

7.2.4 Software maintenance tool

Once a report functionality has already been established, it could be applied to more aspects of the software maintenance. Its use does not have to limit to update crash notification, but may spread to runtime errors or any kind of user feedback. A particular tool for runtime error detection should have to be implemented with that aim as it goes beyond the update system features. Consequently, the update system could be turned into a full maintenance tool, strengthening its capabilities to assist users in software handling.

7.2.5 Collaborative software

Factors such as international collaboration, remote work and short deadline requirements are the cause of the birth and growth of online collaborative tools. As those are expected to spread in the future, so will do collaborative software proposals. As a result, they may end up overtaking traditional repositories and becoming the main source of source code and therefore, the main source of updates. In consequence, appropriate mechanisms should be considered in update system

References

[1] T. Kristensen and S. Frei, “The security exposure of software portfolios,” *Secunia Whitepaper*, no. February, 2010.

[2] T. Duebendorfer and S. Frei, “Why silent updates boost security,” *TIK, ETH Zurich, Tech. Rep.*, no. April, 2009.

[3] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, “Kitsune: Efficient, General-purpose Dynamic Software Updating for C,” *ACM SIGPLAN Not.*, vol. 47, no. 10, p. 249, 2012.

[4] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, “Practical dynamic software updating for C,” in *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’06*, 2006, vol. 41, no. 6, p. 72.

[5] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, “State transfer for clear and efficient runtime updates,” in *2011 IEEE 27th International Conference on Data Engineering Workshops*, 2011, pp. 179–184.

[6] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, “POLUS: A POwerful Live Updating System,” *29th Int. Conf. Softw. Eng.*, pp. 271–281, May 2007.

[7] J. Arnold and M. F. Kaashoek, “Ksplice: Automatic Rebootless Kernel Updates,” in *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009, pp. 187–198.

[8] K. Makris and R. Bazzi, “Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction,” in *Proceedings of the 19th USENIX Annual technical conference*, 2009, pp. 31–44.

[9] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” *Comput. Networks*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[10] L. Hattori and M. Lanza, “Syde,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE ’10*, 2010, vol. 2, p. 235.

[11] S. Jansen, “A process model and typology for software product updaters,” in *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, 2005, pp. 265–274.

[12] M. VojnoviC, T. Karagiannis, and C. Gkantsidis, “Planet scale software updates,” *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4. p. 423, 2006.

[13] A. Bellissimo, J. Burgess, and K. Fu, “Secure software updates: disappointments and new challenges,” *Proc. 1st USENIX Work. Hot Top. Secur.*, pp. 37–43, 2006.

[14] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, “Survivable key

compromise in software update systems," in *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*, 2010, p. 61.

[15] "Yum (Yellowdog Updater, Modified) ." [Online]. Available: http://www.phy.duke.edu/~rgb/General/yum_HOWTO/yum_HOWTO/yum_HOWTO-1.html.

[16] A. Tridgell and P. Mackerras, "The rsync algorithm," *Jt. Comput. Sci. Tec. Rep. Ser.*, vol. TR-CS-96–0, no. June, p. 6, 1996.

[17] M. Ewing and E. Troan, "The RPM packaging system," in *Proceedings of the First Conference on Freely Redistributable Software*, 1996.

[18] P. Hosek and C. Cadar, "Safe software updates via multi-version execution," in *Software Engineering (ICSE), 2013 35th International Conference on*, 2013, pp. 612–621.

[19] M. Vouk, "Back-to-back testing," *Inf. Softw. Technol.*, vol. 32, no. 1, pp. 34–45, Jan. 1990.

[20] J. Tucek, W. Xiong, and Y. Zhou, "Efficient online validation with delta execution," *ACM SIGPLAN Not.*, vol. 44, no. 3, p. 193, Feb. 2009.

[21] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis, "Band-aid Patching," *Third Work. Hot Top. Syst. Dependability 26 June 2007, Edinburgh, UK*, pp. 102–106, 2007.

[22] G. Altekar, I. Bagrak, Burstein P., and A. Schultz, "OPUS: Online patches and updates for security," in *Proceedings of the 14th USENIX Security Symposium*, 2005, p. 19.

[23] E. K. Smith, M. Hicks, and J. S. Foster, "Towards standardized benchmarks for Dynamic Software Updating systems," *2012 4th Int. Work. Hot Top. Softw. Upgrad.*, pp. 11–15, Jun. 2012.

[24] R. a. Bazzi, B. Topp, and I. Neamtiu, "How to have your cake and eat it too: Dynamic software updating with just-in-time overhead," *Proc. Fourth Int. Work. Hot Top. Softw. Upgrad. - HotSWUp '12*, pp. 1–5, Jun. 2012.

[25] S. R. L. Jansen, G. Ballintijn, and S. Brinkkemper, *Software Release and Deployment at Exact: a case study report*. 2004, pp. 1–40.

[26] "Sony pulls buggy PS3 update after user complaints," *CNET News*, 2013.

[27] R. L. Jansen, G. C. Ballintijn, S. Brinkkemper, and A. van Nieuwland, "Integrated SCM/PDM/CRM and delivery of software products to 160.000 customers," *Softw. Eng. [SEN]*, no. R 0416, pp. 1–10, Jan. 2004.

[28] "Windows Installer." [Online]. Available: http://technet.microsoft.com/en-us/library/bb457071.aspx.

[29] "Apple Software Update." [Online]. Available: http://www.apple.com/softwareupdate/.

[30] G. N. Silva, "APT HOWTO." [Online]. Available:

http://www.debian.org/doc/manuals/apt-howto/.

[31] R. Hall, D. Heimberger, A. Van Der Hoek, and A. Wolf, “The Software Dock: A Distributed, Agent-based Software Deployment System,” 1997.

[32] “Omaha Overview.” [Online]. Available: http://omaha.googlecode.com/svn/wiki/OmahaOverview.html. [Accessed: 20-May-2014].

[33] D. Gupta, P. Jalote, and G. Barua, “A formal framework for on-line software version change,” *Softw. Eng. IEEE Trans.*, vol. 22, pp. 120–131, 1996.

[34] I. Lee, “Dymos: a dynamic modification system,” *SIGSOFT Softw. Eng. Notes*, vol. 8, no. 4, pp. 201–202, 1983.

[35] R. Bazzi, K. Makris, P. Nayeri, and J. Shen, “Dynamic software updates: the state mapping problem,” *Proc. Second Int. Work. Hot Top. Softw. Upgrad. - HotSWUp ’09*, 2009.

[36] M. Wahler, S. Richter, and M. Oriol, “Dynamic software updates for real-time systems,” *Proc. Second Int. Work. Hot Top. Softw. Upgrad. - HotSWUp ’09*, p. 1, 2009.

[37] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, “Live updating operating systems using virtualization,” in *Proceedings of the 2nd international conference on Virtual execution environments - VEE ’06*, 2006, p. 35.

[38] M. Hicks and S. Nettles, “Dynamic software updating,” *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005.

[39] C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster, “Efficient systematic testing for dynamically updatable software,” in *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp ’09*, 2009, p. 1.

[40] L. Pina and M. Hicks, “Rubah: Efficient, General-purpose Dynamic Software Updating for Java,” *Proc. 5th Int. Work. Hot Top. Softw. Upgrad. - HotSWUp ’13*, 2013.

[41] S. Martinez, F. Dagnat, and J. Buisson, “Prototyping DSU techniques using Python,” *Proc. 5th Int. Work. Hot Top. Softw. Upgrad. - HotSWUp ’13*, pp. 1–5, 2013.

[42] B. M. Luettmann and A. C. Bender, “Man-in-the-middle attacks on auto-updating software,” *Bell Labs Tech. J.*, vol. 12, no. 3, pp. 131–138, Nov. 2007.

[43] K. Dunn, “Automatic update risks: can patching let a hacker in?,” *Netw. Secur.*, pp. 5–8, 2004.

[44] A. Ornaghi and M. Valleri, “Man in the middle attacks Demos,” *Blackhat [Online Document]*, 2003. [Online]. Available: http://www.smarttech.ie/wp-content/uploads/2013/12/bh-us-03-ornaghi-valleri.pdf. [Accessed: 23-Apr-2014].

[45] D. K. Nilsson, U. E. Larson, and E. Jonsson, “Creating a Secure Infrastructure for Wireless Diagnostics and Software Updates in Vehicles,” in *Computer Safety, Reliability, and Security*, 2008, pp. 207–220.

[46] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, “Security in embedded systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, Aug. 2004.

[47] W. Lund and J. Lilius, “Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems,” *Proc. 5th Int. Work. Hot Top. Softw. Upgrad. - HotSWUp ’13*, 2013.

[48] “MIT PGP Key Server.” [Online]. Available: http://pgp.mit.edu/. [Accessed: 11-Mar-2014].

[49] “Github.” [Online]. Available: www.github.com. [Accessed: 20-May-2014].

[50] “Raspberry Pi Foundation.” [Online]. Available: http://www.raspberrypi.org/. [Accessed: 20-May-2014].

[51] “Raspbian.” [Online]. Available: http://www.raspbian.org/. [Accessed: 28-Apr-2014].

[52] “Git.” [Online]. Available: http://git-scm.com/. [Accessed: 28-Apr-2014].

[53] “The GNU Privacy Guard.” [Online]. Available: https://www.gnupg.org/. [Accessed: 28-Apr-2014].

[54] R. Srinivasan, “XDR: External Data Representation standard,” *Sun Microsystems*, 1995. [Online]. Available: http://tools.ietf.org/html/rfc1014. [Accessed: 14-Apr-2014].

[55] Sun Microsystems, “Rpgen Programming guide.” [Online]. Available: http://docs.oracle.com/cd/E19683-01/816-1435/rpcgenpguide-21470/index.html. [Accessed: 28-Apr-2014].

[56] “UNIX man pages : crontab.” [Online]. Available: http://unixhelp.ed.ac.uk/CGI/man-cgi?crontab+5. [Accessed: 28-Apr-2014].

[57] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava, “Sensor network software update management: a survey,” *Int. J. Netw. Manag.*, vol. 15, no. 4, pp. 283–294, Jul. 2005.