# STREAMLINING THE DEVELOPMENT OF WIRELESS EMBEDDED SYSTEMS USING CONTINUOUS INTEGRATION

Matevž Vučnik

**Doctoral Dissertation**
**Jožef Stefan International Postgraduate School**
**Ljubljana, Slovenia**

**Supervisor:** Prof. Dr. Mihael Mohorčič, Jožef Stefan Institute, Ljubljana, Slovenia
**Co-Supervisor:** Dr. Carolina Fortuna, Jožef Stefan Institute, Ljubljana, Slovenia

**Evaluation Board:**
Asst. Prof. Anton Biasizzo, Chair, Jožef Stefan Institute, Ljubljana, Slovenia
Asst. Prof. Boštjan Slivnik, Member, Faculty of Computer and Information Science, University of Ljubljana, Ljubljana, Slovenia
Asst. Prof. Valentin Rakovic, Member, Faculty of Electrical Engineering and Information Technologies, Ss. Cyril and Methodius University, Skopje, North Macedonia

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL


Matevž Vučnik

# STREAMLINING THE DEVELOPMENT OF WIRELESS EMBEDDED SYSTEMS USING CONTINUOUS INTEGRATION

**Doctoral Dissertation**

# POENOSTAVITEV RAZVOJA BREZŽIČNIH VGRAJENIH SISTEMOV Z UPORABO STALNE INTEGRACIJE

**Doktorska disertacija**

**Supervisor:** Prof. Dr. Mihael Mohorčič

**Co-Supervisor:** Dr. Carolina Fortuna

Ljubljana, Slovenia, November 2020

# Acknowledgments

When I started my PhD study I was not entirely convinced I am going to persist till the very end. Now that I am writing this I am glad I did and I have many people to thank for their encouragement to continue.

First, I would like to thank my supervisor Mihael Mohorčič for always giving me the right directions on how to proceed. He never showed any doubt that I am able to successfully finalize this thesis. His comments were always well thought out and to the point. I also appreciate the time he spent correcting my English.

Besides my supervisor, I would like to thank my co-supervisor Carolina Fortuna. She constantly gave me valuable advice and ideas on how to use my mostly applied work on different research projects to publish in scientific journals and conferences which was essential for finishing my PhD study.

I thank my coworkers at the Department of Communication Systems at the Jožef Stefan Institute for their insightful discussions which inspired me to expand my research from various perspectives.

I would also like to thank the members of the evaluation board for providing very constructive reviews that helped further improving the thesis.

Last but not least, I would like to thank my family and everyone close to me for their support throughout writing this thesis and my life in general.

# Abstract

Wireless communications are subject to rapid development cycles and ever-increasing application and user requirements. The next generation of mobile networks is thus expected to support industrial and traffic safety applications, enhanced multimedia applications and a large number of connected meters, sensors and other devices that will form living and working environments in the future. In order to meet the requirements of all these applications, mobile networks will accommodate both human-type as well as machine-type communications which will be enhanced through the inclusion of non-cellular short-range and low-power wide area technologies in the form of capillary networks. This will result in a multi-technology, high-interference radio environment that can already be found in some frequency bands in dense urban centers.

Wireless communication systems have been traditionally implemented straight into silicon chips leaving little opportunity for reconfigurability. The main reasons for this were the requirements for high power efficiency and strict real-time operation. With the recent advances in hardware platforms, however, we are now witnessing a trend towards softwarization of radio functions in wireless communications. This approach enables the design of highly adaptable and reconfigurable radio platforms as well as rapid prototyping. However, in the case of wireless embedded devices which are often characterized by restricted capabilities, this trend has yet to catch up and go beyond prototyping.

Software development and testing is a complex process involving highly skilled people and dedicated infrastructure. A mix of organizational practices, development practices and infrastructure contributes to the efficiency of the overall process which is relatively well understood for general purpose development. However, the development of wireless embedded firmware and its testing in a real environment is notably more challenging because the code needs to be uploaded and tested on target embedded hardware, requiring purposely developed testing infrastructure that needs to be optimally dimensioned.

In this thesis we investigated the dimensioning of embedded infrastructure as a function of developer activity to ensure predictable test completion times and reliably quantify the infrastructure size. By modelling developers activity, we gave better insights into the as yet unstudied part of the current development practices. With such knowledge, a reliable capacity dimensioning of supporting infrastructure is possible.

We further proposed a framework that can be used by wireless technology developers to enable continuous integration practices in their testbed infrastructure. We provided a proof-of-concept reference architecture and implementation of the framework for controlled testing of multi-technology wireless networks which blends web service technology and operating system virtualization technologies with emerging internet of things technologies enabling continuous-integration practices for wireless embedded development and testing.

Finally, we performed the evaluation and testbed dimensioning for different sizes of development teams. In particular, our capacity dimensioning simulation shows that for a team of 10 developers, only 1-2 parallel embedded test environments are needed to ensure test completion in less than ten minutes in 95% of the cases.

# Povzetek

Brezžične komunikacije so podvržene hitrim razvojnim ciklom in vse večjim zahtevam aplikacij in uporabnikov. Pričakuje se, da bo naslednja generacija mobilnih omrežij podpirala industrijske aplikacije in aplikacije za varnost v prometu, izboljšane multimedijske aplikacije in veliko število povezanih pametnih števcev, senzorjev in drugih naprav, ki bodo sestavljali bivalno in delovno okolje prihodnosti. Da bi izpolnili zahteve vseh teh aplikacij, bodo mobilna omrežja podpirala tako človeške kot tudi strojne komunikacije, ki bodo nadgrajene z vključitvijo neceličnih tehnologij kratkega dosega ter tehnologij nizkih moči in velikega obsega v obliki kapilarnih omrežij. To se bo odražalo v heterogenem radijskem okolju z različnimi tehnologijami in z veliko medsebojnimi motnjami, kakršno je že mogoče najti v nekaterih frekvenčnih pasovih znotraj mestnih središč.

Brezžični komunikacijski sistemi so bili tradicionalno vgrajeni neposredno v silicijeve čipe, ki puščajo malo možnosti za spremenljivo konfiguracijo. Glavni razlogi za to so bile zahteve po nizki porabi energije in delovanju v realnem času. Z nedavnim napredkom strojnih platform smo priča uvajanju programirljivih radijskih funkcij v brezžične komunikacije. Ta pristop ob uporabi zmogljivejše strojne opreme omogoča zasnovo zelo prilagodljivih radijskih platform ter hitro izdelavo prototipov. V primeru vgrajenih brezžičnih naprav, za katere so pogosto značilne omejene zmogljivosti, pa je treba ta trend še ujeti.

Razvoj in testiranje programske opreme je zapleten postopek, ki vključuje visoko usposobljene ljudi in ustrezno infrastrukturo. Kombinacija razvijajočih se organizacijskih praks, razvojnih praks in infrastrukture prispeva k učinkovitosti celotnega procesa, ki je razmeroma dobro razumljen za razvoj splošne programske opreme. Razvoj brezžične vgrajene programske opreme in njeno testiranje v resničnem okolju pa je bistveno zahtevnejše, saj je treba prevedeno programsko kodo naložiti in preizkusiti na ciljni vgrajeni strojni opremi, kar zahteva razvoj posebne infrastrukture za testiranje, ki jo je treba tudi optimalno dimenzionirati.

V tej disertaciji smo raziskovali dimenzioniranje vgrajene testne infrastrukture kot funkcijo aktivnosti razvijalcev, da bi zagotovili predvidljive čase dokončanja testiranja in zanesljivo določili zmogljivost testne infrastrukture. Z modeliranjem aktivnosti razvijalcev smo prispevali boljši vpogled v še neraziskani del trenutnih razvojnih praks in s pridobljenim znanjem omogočili zanesljivo dimenzioniranje zmogljivosti testne infrastrukture.

Nadalje smo predlagali okvir, ki ga lahko uporabljajo razvijalci brezžičnih tehnologij, da omogočijo prakse stalne integracije na svoji testni infrastrukturi. Zagotovili smo referenčno arhitekturo in izvedbo sistema za nadzorovano testiranje različnih tehnologij brezžičnih omrežij, ki združuje tehnologijo spletnih storitev in virtualizacijo operacijskih sistemov z vse bolj razširjenimi tehnologijami interneta stvari, kar omogoča uspešno vpeljavo prakse stalne integracije v razvoj in testiranje brezžičnih vgrajenih sistemov.

Na koncu smo izvedli še evalvacijo in dimenzioniranje testnih okolij za različno velike razvojne skupine. Rezultati simulacij dimenzioniranja testnih zmogljivosti so pokazali, da sta za skupino do 10 razvijalcev potrebni le 1-2 vzporedni vgrajeni testni okolji, da se v 95% primerov test zaključi prej kot v desetih minutah.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abbreviations

HTC ... Human-Type Communications
MTC ... Machine-Type Communications
MAC ... Medium Access Control
FOSS ... Free and Open-Source Software
CI ... Continuous Integration
LCSP ... Light-weight Client Server Protocol
CBD ... Component-Based Development
IT ... Information Technology
TDD ... Test-Driven Development
SaaS ... Software as a Service
FSF ... Free Software Foundation
GPL ... General Public License
LGPL ... GNU Lesser General Public License
VCS ... Version Control Systems
SSH ... Secure Shell
AWS ... Amazon Web Services
GKE ... Google Kubernetes Engine
LAN ... Local Area Network
MIT ... Massachusetts Institute of Technology
CD ... Continuous Delivery
SVN ... Apache Subversion
CVS ... Concurrent Versions System
SOA ... Service-Oriented Architecture
RUP ... Rational Unified Process
RAD ... Rapid Application Development
XP ... eXtreme Programming
SCM ... Source Code Management
MPL ... Mozilla Public License
FDD ... Feature-Driven Development
EPL ... Eclipse Public License
DSDM ... Dynamic systems development method
BSD ... Berkeley Software Distribution
ASD ... Adaptive Software Development
USD ... Unified Software Development
TCP ... Transmission Control Protocol
SSL ... Secure Sockets Layer
SDLC ... Software Development Life Cycle
API ... Application Programming Interface
COINS ... Continuous IntegratioN in wirelesS technology development
OMF ... cOntrol and Management Framework
NITOS ... Network Implementation Testbed Laboratory

BOWL ... Berlin Open Wireless Lab
UCI ... Unified Configuration Interface
SDN ... Software Defined Network
VNF ... Virtual Network Function
OAI ... OpenAirInterface
PDF ... Probability Density Function
CDF ... Cumulative Distribution Function
NAT ... Network Address Translation
VESNA ... VErsatile platform for Sensor Network Applications
VMS ... VESNA Management System
CRC ... Cyclic Redundancy Check
LQE ... Link Quality Estimation
ISM ... Industrial, Scientific and Medical
UWB ... Ultra-Wide Band

# Symbols

| | | |
|---|---|---|
| $\mathcal{N}$ | ... | normal distribution |
| $\mathcal{X}$ | ... | bimodal distribution, a sum of two normal distributions |
| $A_1$ | ... | amplitude of the first normal distribution in a bimodal distribution |
| $A_2$ | ... | amplitude of the second normal distribution in a bimodal distribution |
| $\mu$ | ... | median of normal distribution |
| $\sigma$ | ... | standard deviation of normal distribution |
| $\mathrm{e}^x$ | ... | natural exponential function |
| $\rho$ | ... | exponential distribution shift |
| $M$ | ... | exponentially distributed random variable |
| $G$ | ... | generally distributed random variable |
| $N$ | ... | number of parallel test environments |
| $\lambda$ | ... | Poisson process arrival rate |
| $A$ | ... | Erlang traffic |
| $h$ | ... | average execution time |
| $p$ | ... | percentage of expected contributions in the busy development hour |
| $r$ | ... | number of all considered daily contributions per developer |
| $k$ | ... | number of developers in the team |
| $W$ | ... | average wait time in the queue |
| $\mathcal{W}$ | ... | waiting time random variable |
| $w$ | ... | acceptable waiting time |
| $P_C$ | ... | Erlang C formula |
| $P_{GoS}$ | ... | probability of the grade of service |
| $L$ | ... | average number of tests in the system based on Little's Law |

# Chapter 1

# Introduction

Wireless communications are subject to rapid development cycles and ever-increasing application and user requirements. Mobile and wireless networks are thus expected to support industrial and traffic safety applications, enhanced multimedia applications on smartphones and a large number of connected meters, sensors and other devices that will comprise living and working environments of the future. In order to meet the requirements of all these applications, mobile networks will accommodate both human-type communications (HTC) and machine-type communications (MTC) [1]. MTC capabilities of existing predominantly HTC-oriented cellular technologies will be enhanced through the inclusion of non-cellular short-range and low-power wide area technologies in the form of capillary networks. This will result in a multi-technology, high-interference radio environment such as the one that can already be found in some frequency bands in dense urban centers [2].

Wireless communication systems have been traditionally implemented directly in silicon chips from physical layer up to at least the lower part of the Medium Access Control (MAC) layer. The main reasons for this were the requirements for high power efficiency and strict real-time operation. With the recent advances in hardware platforms, however, we are now witnessing a trend towards softwarization of radio functions in wireless communications [3]. This approach enables the design of highly adaptable and reconfigurable radio platforms as well as rapid prototyping. However, in the case of MTC systems, often involving wireless embedded devices characterized by restricted capabilities, this trend has yet to catch up and it is not clear if it will ever go beyond prototyping.

The increasing number of heterogeneous wireless technologies, their rapid development cycles and the trend towards full softwarization of radio functions [4] call for the introduction of more efficient and optimized practices in the design and development of wireless embedded systems that are suitable also for distributed teams of developers. In this dissertation we show that in the area of wireless embedded software development the continuous integration approach with appropriate adaptation brings similar benefits as in general software development.

## 1.1 Software Development

Typically, software development is organized in teams of developers, therefore it requires a sort of management and organization which can be professional or community-based [5]. Free and Open-Source Software (FOSS) development is becoming increasingly important and many developers get paid by companies and enterprises to develop FOSS software within those communities. However, the software development life cycle is independent of the type of development [6]. Rosove's software development life cycle model, depicted in Figure 1.1, defines a sequence of development stages within a feedback loop of each stage

[7]. The separate stages in the model are Requirements, Design, Production, Installation and Operations. These stages are executed in a timely manner. Each stage is followed by the next one. Furthermore, each stage can have an influence on other stages which were executed earlier in the cycle.

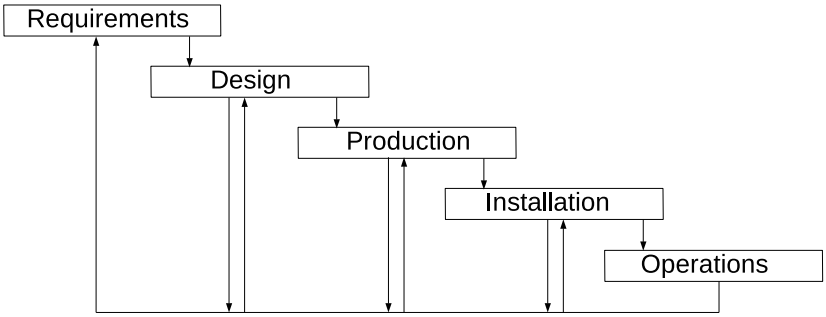



Figure 1.1: Rosove's software development life-cycle model [6].

The modern tools supporting software development are covering several of these stages with the intent to enable management, collaboration of developers as well as the development itself, making it more efficient by introducing automation in each stage and thus streamlining the development process. In software development, teams of programmers concurrently produce many lines of code, which needs to be integrated and tested. There are two distinct approaches to code integration, the first assumes a separate integration phase after the development and the second is based on Continuous Integration (CI) approach [8], [9]. Software development projects have practically without exceptions adopted the CI approach. Therefore, any bugs caused by the integration can be discovered already during the development phase, thus reducing the time needed for the testing phase and dismissing the need for a separate integration phase.

Compared to general software development, building wireless firmware is a slow process which requires complicated tool-chains and testbeds [10], [11]. Although complicated and expensive to build and maintain, distributed and heterogeneous wireless testbeds offer the most realistic experimental conditions [12]. Such long-lasting integration tests are typically performed during the night (i.e., off-work hours), therefore a common practice is to refer to those tests as nightly builds. However, we argue that by introducing incremental builds it is possible to introduce fully automated CI practices also in wireless communication systems development, and with suitable adaptation even to wireless embedded systems with restricted capabilities. The main challenge in this respect represents the testing phase of wireless firmware as it requires interaction with real world environment and may affect (i.e. interfere with) other test nodes, whereas general software can run in isolated parallel tests on a single computer or even in the cloud. In addition to adaptation of CI practices to wireless embedded systems this challenge calls for an in-depth investigation and understanding of wireless developers' activity, which in turn needs to be reflected in suitable dimensioning of wireless testing resources and scheduling of code tests.

Evidence from the literature suggests that a lot of work was invested during the years in the introduction of modern software development practices in embedded systems firmware development [10], [13], [14]. However, there are still opportunities for streamlining the development of embedded systems, particularly the wireless ones [15], [16]. Several challenges remain unresolved and are well understood by the community, i.e. hardware dependency, test environment configuration and process automation [17].

## 1.2 Motivation and Hypothesis

Due to the required testing on physical devices and inherent interaction with the real radio environment, current wireless embedded systems development is more challenging compared to traditional software development and calls for optimization and increased efficiency of procedures. The purpose of this dissertation is thus (i) to obtain a deep understanding of the specific needs of wireless developers; (ii) to adapt the CI practices to the specific characteristics and needs of wireless firmware development and testing; and (iii) to develop and implement a reference CI framework supporting firmware development for wireless embedded systems.

In this thesis we aim to confirm the following three hypotheses:

H1: By the analysis of large amounts of public metadata available from large-scale social coding platforms, it is possible to extract the information about the wireless developers' behavior in terms of frequency of committed source code, and use this information to build a data-driven probability distribution model.

H2: Continuous integration practices can be effectively adapted for streamlining the development of wireless embedded systems by the introduction of completely automated test build and execution framework for wireless testbeds.

H3: With the probability distribution model of commits it is possible to optimize the dimensioning of wireless testing resources and scheduling of code tests, and thus supporting the efficiency of the adopted CI approach.

To support these hypotheses we developed a new data-driven probability distribution model of commits per developer per day and introduced a new CI framework for the development of wireless embedded systems. We tested the hypotheses by defining the minimum testbed capacity required to ensure that developers obtain the CI feedback in 95% of the situations in under 10 minutes (i.e., in a so-called CI golden rule).

## 1.3 Contributions

The main goal of the dissertation is to streamline the development of wireless embedded systems by adapting and introducing continuous integration practices in the development and testing of wireless firmware. In this dissertation we achieved several original scientific contributions, which also represent partial goals in the process of achieving the main goal.

The main scientific contributions are:

C1: Data-driven probability distribution model of software developer code contributions.

C2: A reference framework for efficient introduction of continuous integration practices in wireless embedded systems development.

C3: Analysis of the impact of developers' activities on test execution in a continuous integration process with testbed dimensioning guidelines.

Additional contributions important for the broader research field include:

C4: An extensive analysis of related work and extension of current comparison methodology for wireless embedded testbeds.

C5: A discrete event simulator for evaluation of test execution times in a continuous integration process.

C6: A new light-weight application layer protocol to support the introduction of a continuous integration approach in wireless embedded devices with restricted capabilities.

By achieving the main goal and original contributions set forth in this dissertation we notably streamlined the development processes in wireless embedded software development. Results of this thesis are thus expected to advance the scientific knowledge in the field and benefit the broader community of wireless developers, and in particular the community of wireless embedded systems developers, both in industry and academia. Moreover, the model of developers' hourly and daily activities can be further used by the scientific community to investigate some of the remaining CI-related issues such as test coexistence in wireless environments, as well as professional community in optimization of organizational processes such as production planning or management of human and production resources.

## 1.4 Methodology

To confirm the first hypothesis, we analyzed and extracted the characteristics of wireless embedded system developers' activity by performing data mining of extensive metadata collected from large-scale social coding platforms, followed by statistical analytics and building of a data-driven probability distribution model of software developers code contributions per day.

To confirm the second hypothesis, we started with a thorough analysis of related research work and practical solutions, and identification of specific challenges for introducing CI practices in wireless embedded systems development. This preliminary investigation led to the definition of an architecture of a completely automated framework for introducing CI practices in a wireless embedded systems development process and its reference implementation based mostly on popular existing (open source) DevOps tools as an extension of existing wireless experimentation testbed LOG-a-TEC, both activities based on the advanced software engineering approach.

To confirm the third hypothesis, we carried out the evaluation of the CI framework in wireless embedded systems development. The evaluation is based on a combination of computer simulations, testbed experimentation and measurements using real wireless embedded devices, resulting in heuristic dimensioning of testbed capabilities.

## 1.5 Organization of the Dissertation

The content of this dissertation is structured in 7 main chapters with original contributions described in Chapters 3 to 6.

Chapter 1 gives the introduction to the topic of software development. It further presents the motivation, contributions and methodology of the dissertation.

Chapter 2 provides the necessary background and a brief overview of related work, introducing the field of software development from the perspective of development as well as organizational practices. We cover the state-of-the-art development and organizational practices and provide an extensive study of the tools and platforms supporting the software development. At the end of the chapter, we provide the current studies in the field of developers activity.

Chapter 3 provides an extensive analysis of the requirements related to the introduction of CI practices to the development of wireless embedded systems. Next, we identify the specific challenges related to the testbed-based wireless embedded systems firmware development. Finally, we introduce a reference framework for efficient introduction of CI practices in wireless embedded systems development (C2).

Chapter 4 describes the implementation of the framework and the definition of a new communication protocol named Light-weight Client Server Protocol (LCSP), which is a special purpose application layer communication protocol (C6). Furthermore, we propose and implement an example architecture for an experimental testbed which is able to execute wireless embedded CI tests. Finally, we include the comparison with the closely related work as well as an extension of the current comparison methodology for wireless embedded testbeds (C4).

Chapter 5 analyzes the developers' activities as important aspects for efficient introduction of CI practices in wireless firmware development, therefore a new developers activity probability model is proposed. First, we model the testing process based on a queuing theory. Next, we propose hourly as well as daily developer activity models (C1). Additionally, we provide the execution time measurements of tests on a real wireless testbed LOG-a-TEC.

Chapter 6 provides the performance evaluation of the proposed reference architecture and its example implementation. It describes the design of the simulator to analyse the affect of the parallel test execution capacity to test execution times (C5). It also provides the results of the simulations with the guidelines to achieving the optimal testbed capacity in the desirable time margins when introducing CI practices in the development of wireless embedded firmware (C3).

Finally, Chapter 7 concludes the dissertation by summarizing the contributions and providing the directions for future work.

# Chapter 2

# Background

Various development as well as organizational approaches emerged with the intent of streamlining software engineering within developers and management teams. Furthermore, various platforms and tools were developed to support and facilitate the programming process. They are based on different paradigms and methodologies, licensing terms or/and the targeting type of code. Within this chapter we will summarize the different aspects which are important in the state-of-the-art software engineering and thus essential for subsequent understanding of this thesis.

## 2.1 Software Development Practices

Software development is a social process where actors solve problems by writing computer code [18]. It is a constantly evolving process with new programming paradigms, methodologies, frameworks and tools [19]. The most important software development paradigms can be roughly divided into traditional and agile. The traditional so called waterfall software development is organized sequentially in several phases where the result of each phase is an input to the next phase. On the other hand, the agile software development paradigm favours inter team communication as well as consideration of customers' needs, and instead of rigid project planning it prefers rapid response to changes. For instance, lean software development is an agile methodology strictly focused on customer needs which tries to eliminate everything that does not directly benefit the customer [19], [20].

Other two important paradigms are iterative and component-based development (CBD). A typical software development life cycle (SDLC) includes 6 stages: planning, analysis, design, coding, testing and maintenance. An iterative development paradigm goes through the entire cycle several times. A CBD is a modular software development paradigm where the system is split into standalone modules which are supplied by different providers and integrated together to form a complete application.

DevOps is an agile approach to software development. DevOps combines the two formerly distinct processes, i.e. development and information technology (IT) operations, into a single process. The main idea is to bring together software development and IT automation, which is responsible to put the developed code in the production and to enable the complete software life-cycle monitoring [21].

A large study of software development methodologies is presented in [20]. The authors surveyed more than 150 project managers about their software development methodology. They concluded that traditional development using waterfall methodology is still very common in large companies. However, it is evident from the survey results that agile development is becoming increasingly important, especially in companies with a lower number of employees. The authors classified the development teams based on their size in

small with ten or less developers, medium between 10 and 30 developers and large with more than 30 developers. The results also suggest that for critical projects, traditional methods are still more common than the agile ones.

Implementation of the state-of-the-art software introduces new challenges, which require different development approaches. Some of the most notable challenges in current software development are: distributed systems, cloud, Software-as-a-Service (SaaS), scalability, and the use and development of FOSS [22]. Agile software development paradigm proved to be suitable for the state-of-the-art software development especially in startups [20]. Furthermore lean software development is gaining a great deal of adoption in many companies [23].

## 2.2 Organizational Practices

Organizational practices describe the structures and relations that enable the development of software systems. The important values in organizational practices are, among others, time, money and skills. The focus of this thesis is in the organization of skills or expertise in the software development team. Comprehensive research conducted in [24] shows that top management often plays an important role in software systems development. Furthermore, the development team cohesiveness is of significant importance for the project performance. The size and organization of the development team can also be responsible for the project outcome. The team developing and deploying a software system coexists in a process where interaction with people and technology is equally important for the success of the project.

### 2.2.1 Team Management

The team organization can be based on a top-down management approach, which is a more traditional approach, or a more modern one where the focus for the organization is the product itself. Here, the idea is to develop various parts of the software system at different sites using the concept known as distributed software development. The research published in [25] shows that distributed development does introduce some delay in the development process compared to the same-site work. However, the effect is indirect. The problem appears to be the lack of effortless communication between team members which could provide an overall knowledge of the expertise each team member brings to the project. Therefore expertise coordination can significantly influence the project's outcomes [26]. Furthermore, the investment in tools and practices proves to be justified for the organization to reach faster software release cycles [27].

Distributed software development is ubiquitous in FOSS development. Research shows how the challenge of assigning the work is done in FOSS projects. Many FOSS projects proved to be very successful and that knowledge can be transferred also in a corporate environment developing proprietary software. However, FOSS projects mostly rely on self-assignment of the work by developers themselves, whereas in a corporate environment, the top-down hierarchical management is much more common than a development organization based on a volunteering process. The task self-assignment process works by distributing the task description to all available developers who evaluate the task and volunteer to do it. In this case, the expertise management is purely self-motivated and there is no risk of assigning the task to someone without sufficient skills. In contrast, in proprietary software development, it is often a project manager who assigns a developer to the task. This process requires competent expertise management to optimally assign the tasks to individual developers. A form of self-assignment of tasks was introduced in CERN and is known by the name 'gentle' management [25].

### 2.2.2 Agile Development Organization

The developers are known to dislike a rigid management structure in their workplace, therefore initiatives appeared which provided an alternative to the traditional top-down management in software development. One of the more prominent initiatives was the so-called Agile Manifesto [28], [29]. It is a collection of four most important values which can lead to better software development, if applied correctly:

1. Individual interactions,
2. Working software,
3. Customer interactions,
4. Response to change.

The first value explains that individual interactions are more important than strictly following processes and using process management tools. The second one prioritizes working software over extensive documentation. The third one explains that customer interaction is more important than strictly following contracts which leads to the last one explaining that more important than following the plan is to be able to adapt to change which is inevitable in any software development process.

The agile software development organization puts more emphasis on people than project management tools, which is the case in traditional development organization. The managers organizing the software development following the agile approach focus on individual talent, expertise and team communication [30]. In a transition to the agile team organization, it is important to remove the rigid micromanagement structure and rather introduce a so-called macromanagement organization which gives the team space to approach problems more independently and contribute to the project success.

This kind of transition is not trivial, therefore researchers working in this field introduced frameworks to help with the agile development team organization. In [31], the authors propose an organizing framework for agile software development and provide a deeper understanding of agile practices. Furthermore, they identify the enablers and inhibitors of agility, and explain the expected capabilities required in a successful agile development team. The conclusion of the study is that for a successful agile development team a high level of self-initiative is required including a form of self-management and supportive structure of communication among team members.

Another study dealing with agile software development success factors was reported in [32]. The authors collected data in a survey from 109 agile projects from organizations of different sizes and locations. They came to similar conclusions as others that for a successful agile development process it is important to include agile project management, an agile friendly team environment and a strong customer involvement.

## 2.3 Infrastructure Support for Software Development

Professional software development is rarely done by a single programmer but rather by multiple programmers and developers collaborating in a team. Therefore, typically some kind of infrastructure support is required to enable efficient and maintainable development of software components implemented in the form of the source code which is constantly evolving [29]. The focus of this work is on the team software development practices described in the previous sections which historically first became possible with the introduction of online software source code Version Control Systems (VCS) often referred to as Source

Code Management (SCM) systems [33]. These systems enabled the distributed software development by introducing a centralized online code repository and management of code contributions by different team members. The idea is based on graph theory where the development happens in the mainline or the trunk with branches forming a directional tree structure [34].

### 2.3.1 VCS and SCM Systems

One of the typical features of the VCS is code checkout, which is used to obtain a working version of software from a remote server hosting source code repository. Each incremental change to the source code produces a differential history log which helps with the tracking of software evolution. In most VCSs, such change is referred to as commit which marks a change in the repository. Typically, modern VCSs keep commits locally until the developer is ready to update the server. This is done by initiating a remote server push command. A branch is used to create a copy of the mainline source code to enable independent development by programmers at different speeds. When the development in a separate branch is finished, it can be merged back in the mainline. On the other hand, commits from other developers can be downloaded from a remote repository by initiating a pull command. Releases of stable versions of codebase can be labeled with a tag which is similar to a branch. The tag enables release maintenance independent of the mainline development.

Among the most influential and historically important implementations of VCSs are CVS[1], SVN[2], BitKeeper[3], Git[4], Mercurial[5] and Bazaar[6].

### 2.3.2 CI/CD and DevOps Tools

CI/CD and DevOps initially appeared as development practices, however they soon became more about the infrastructure and tools supporting these practices. Historically, software integration was a separate phase which followed the development phase. The idea of CI is that the integration happens constantly during the development and that a separate integration phase is no longer needed. Typically, multiple developers work on the same code-base simultaneously. In such environments it can happen that a code fix from one developer breaks a feature of another. By introducing CI, such problems get discovered and fixed early in the development process.

CI practices should properly interface five main entities [9] described below:

**Repository** holds the code, configuration files, tests, documentation and scripts needed for the automated build. Generally, the repository is based on a kind of VCS.

**Build automation** typically includes a large set of tools and components to compile and build the source code in a working software.

**Tests** are included inside the repository and need to be executed on each build to ensure that the integration was successful.

**Fast build** is important so that each commit can be built and tested typically in under 10 minutes.

[1] CVS, `https://www.nongnu.org/cvs/`
[2] SVN, `https://subversion.apache.org/`
[3] BitKeeper, `http://www.bitkeeper.org/`
[4] Git, `https://git-scm.com/`
[5] Mercurial, `https://www.mercurial-scm.org/`
[6] Bazaar, `https://bazaar.canonical.com/en/`

**CD** of stable releases into production. It can be a part of the CI or an independent process.

To achieve a successful blend of software development, continuous production deployment and resource monitoring, a high degree of automation is required [21]. DevOps tools play an important role in this process. Traditionally, two separate teams are responsible first for software development and second for putting the software in production environment as well as later monitoring the production behavior. The first is a team of software developers and the second a team of IT engineers. Developers are familiar with writing code and IT engineers with all the required operating system tools ensuring that software runs fluently. IT engineers are also responsible for network administration and security. These are very important aspects of the software development life-cycle and DevOps paradigm is not trying to make IT professionals obsolete. On the contrary, the idea is to provide them with new tools so that they are able to automate and abstract these complicated procedures in a way that developers without the necessary IT skills can provide a high level description of how particular software needs to behave in production and from there everything else can be automated.

Such high level of automation requires the tools for:

- orchestration,
- container runtime and,
- monitoring.

The tools are typically interconnected through exposed APIs to form proper toolchains which enable end-to-end DevOps practices implementation. Orchestration tools are used for distributed system deployment and management. Example implementations of such tools are Ansible[7] and Puppet[8]. The first uses a popular Secure Shell (SSH) client as a foundation, so its architecture is agent-less, whereas the second is based on its own agent implementation.

Container runtime tools, such as Docker[9] and rkt[10] are widely used for systems deployment automation. The description of a container includes all the required dependencies as well as the required executable for running the service. Furthermore, only the required TCP ports for exposing the service APIs are exposed outside the container, therefore the usage of container technology can improve security if properly used [35]. The next step is to run the developed system in a production environment which for web applications typically requires a cloud provider such as Amazon Web Services[11] (AWS) and AWS Lambda or Google Kubernetes[12] Engine (GKE). AWS and GKE allow running containerized microservices in the cloud while AWS lambda adds another layer of abstraction and enables the development of cloud native applications based on a serverless architecture [36].

The last step is to enable continuous monitoring of services in production and provide alerts when misbehavior is detected. Some popular monitoring solutions are, among others, Nagios[13] and Munin[14], or a more modern implementation Prometheus[15].

[7] Ansible, `https://www.ansible.com/`
[8] Puppet, `https://puppet.com/`
[9] Docker, `https://www.docker.com/`
[10] rkt, `https://coreos.com/rkt/`
[11] AWS, `https://aws.amazon.com/`
[12] Kubernetes, `https://kubernetes.io/`
[13] Nagios, `https://www.nagios.org/`
[14] Munin, `http://munin-monitoring.org/`
[15] Prometheus, `https://prometheus.io/`

This kind of infrastructure support nowadays represents the state-of-the-art software development and automated production delivery for web-based applications. This section summarized the operations part of DevOps, whereas the next one covers the development part.

### 2.3.3 Software Development Platforms

While contemporary software mostly relies on cloud, it is important to recognize that there is a large variety of software being developed which is not-cloud based. This software development can also benefit from certain advanced practices described in the previous sections. Especially CI can be adopted by literally any software or even firmware development, where the latter can be particularly cumbersome and challenging, and will be extensively covered within this thesis.

Software development platforms which enable CI can be self-hosted on Local Area Network (LAN) or cloud-based, and they can integrate various CI and DevOps features. One of the first and to this date still the most influential cloud platform for hosting source code version control is GitHub[16]. There are also others from simple online Git repositories such as Gitea[17] to complete DevOps frameworks such as GitLab[18]. The goal of this section is not to provide an exhaustive list of frameworks, since this is a rapidly evolving field, but rather to provide different views on how software can be developed. Some of the currently most influential tools and platforms in distributed software development are summarized in Table 2.1 and briefly described below.

Table 2.1: Software development platforms.

| Platform | Local | Cloud | VCS | CI/CD | DevOps |
|---|---|---|---|---|---|
| GitHub | ✗ | ✓ | ✓ | ✗ | ✗ |
| GitLab | ✓ | ✓ | ✓ | ✓ | ✓ |
| Bitbucket | ✓ | ✓ | ✓ | ✓ | ✓ |
| Gitea | ✓ | ✗ | ✓ | ✗ | ✗ |
| SourceForge | ✗ | ✓ | ✓ | ✗ | ✗ |
| Travis CI | ✗ | ✓ | ✗ | ✓ | ✓ |
| Buildbot | ✓ | ✗ | ✗ | ✓ | ✗ |
| Jenkins | ✓ | ✗ | ✗ | ✓ | ✓ |

GitHub is a proprietary cloud platform for source code version control which currently still does not integrate support for CI/CD and DevOps features. However, it provides a very powerful system for integrating external services which can provide the missing features. GitLab, on the other hand, from the very start focuses on complete integration of DevOps features while itself also being FOSS. Furthermore, it also enables hosting a local instance of the platform. GitLab integrates features for automated testing and building of production releases by defining pipelines. However, for running the releases in production environment, one must rely on external cloud providers such as AWS or GKE. Bitbucket[19] includes similar features as GitLab, however it is proprietary software which can also be used locally through Bitbucket server. Gitea is a FOSS package for hosting Git-based software development and includes features such as issues tracking and pull requests.

[16] GitHub, `https://github.com/`
[17] Gitea, `https://gitea.io/`
[18] GitLab, `https://gitlab.com/`
[19] Bitbucket, `https://bitbucket.org/`

SourceForge[20] was one of the first platforms for centralized hosting of software repositories free of charge for FOSS development and in this way helped with popularizing FOSS development. It later gained bad reputation because of its monetization practices which included misleading download buttons as well as distribution of malware and changing owners. Travis CI[21] is a hosted platform for building, testing and deploying software projects hosted on GitHub. It is integrated with GitHub through its application API. Travis CI is technically FOSS, however not in the form that could be easily self-hosted by anyone. Buildbot[22] is a locally hosted FOSS tool for building, testing and releasing software located in a VCS. It has an architectural design of a single master and multiple slave nodes. The master node is responsible for checking out the code from VCS and the slave nodes are responsible for building it. Multiple slave architecture is practically useful when a build needs to be done for different target architectures. Jenkins[23] is nowadays one of the most popular self-hosted CI/CD platforms with numerous plugins which make it even more powerful. It can be integrated with different VCSs through its rich web-based configuration interface. Similar to Buildbot, the builds can be done on multiple worker nodes with different architectures.

## 2.4 Development Process Modeling

### 2.4.1 Embedded Wireless Testing Infrastructures

In the development of embedded systems, significant effort is being invested in the testing process [11]. The specifics of embedded systems require testing on real hardware to avoid the undesired bias which can be introduced when relying only on simulations [37]. An approach of combining real hardware with simulations for the purpose of system and verification testing is known as co-simulation of mixed hardware and software systems, also referred to as hardware-in-the-loop (HIL) testing [38]. The authors in [37] proposed HIL testing approach for the development of new wireless technologies and protocols, and the authors in [39] argued that the co-simulation of software and hardware provides a more realistic testing and evaluation environment to some extent, improving the efficiency of test and evaluation. However, it is still different from the real test environment. Therefore, real hardware test platforms and the environments are essential to the process of developing new wireless technology.

Over the last two decades, large efforts went into developing realistic wireless embedded and sensor experimentation and testing environments predominantly in academia. Perhaps, the first such larger wireless testbed was available at the University of Berkley [40], while in [41] they identify about 30 such facilities. With the initiatives such as FIRE [42] in Europe, of which LOG-a-TEC [43], [44] is part of, GENI [45] in the US and the emerging 5G wireless systems [39], significant technological advances in supporting more cost-effective, real-life wireless testing became possible.

### 2.4.2 Dimensioning of Testing Resources

There are two distinct problems in the case of testing resources dimensioning. The first one is the stationary problem and the second is the dynamic or elastic problem. In the case of a stationary problem, the resources can not be easily added on demand and need to be dimensioned in advance, typically for the worst case scenario estimate. In the case

[20] SourceForge, `https://sourceforge.net/`
[21] Travis CI, `https://travis-ci.org/`
[22] Buildbot, `https://buildbot.net/`
[23] Jenkins, `https://jenkins.io/`

of a dynamic problem, the resource demand can be estimated and adjusted during the execution such as the CPU resources or even virtual machine resources in the case of cloud services. Our problem of parallel testing environments for wireless embedded systems can be classified as a stationary problem which needs an optimal static dimensioning in advance since such test environments are typically complicated and expensive to set up and operate. The dynamic problems can be approached using past data feed into e.g. linear regression models while for the stationary ones queuing theory is more appropriate. Modelling approaches for the two groups also differ as reviewed in [46].

For instance, by using queuing theory the authors in [47] and [48] modeled the number of required debuggers in software testing to improve service time and optimize costs, while the author of [49] applies it to the aspects of software performance and scalability, namely waiting times for high volume requests on web platforms and databases, computing resource usage and API profiling.

The work most closely related to ours is that of [50] where the authors used queuing theory to model fault corrections in software development. Using the model, they investigate the optimal number of fault correction stations for NASA Space Shuttle. Their approach is similar to ours in terms of mathematical and simulation tools. However, the novelty of our work is in the application of queuing theory to modeling of CI for wireless embedded systems development.

### 2.4.3 Developers Activity

Queuing theory models rely on parameters describing the characteristics and distributions of incoming requests and service processes. Such parameters were traditionally modelled using statistical sampling, however, in our specific case, the requests for the CI are connected to the activity of developers. Since several types of developer activities are recorded in software repositories, we took a data driven approach of mining existing software repositories and extract knowledge in the form of models describing developer activity.

Knowledge about the developers activity can be extracted from various software development platforms and projects. Substantial amount of work has already been invested in mining different public software repository datasets, generated mostly from the largest social coding platform GitHub as well as from TravisCI[24], a large CI platform for testing software in the cloud. In the following, we summarize the relevant literature overview related to mining software repositories and the CI and DevOps development practices.

The authors frequently use GHTorrent dataset for their analysis and often combine it with different sources of data such as TravisTorrent, which holds metadata from the TravisCI platform. Furthermore, they use the StackOverflow metadata as well as their own survey data as a basis for their research.

In [51], the authors combined a survey conducted with 240 GitHub users with the GHTorrent dataset to study the quality of GitHub metadata. The authors suggest to be careful when analyzing GitHub metadata because of the varying quality. We took this into consideration and based our analysis only on non-empty pushes. This means that at least one commit was included in the push which is an evidence of an active user.

A different quality-related study was reported in [52]. The authors did a quality analysis of open-source software based on the number of stars which are given to a repository from other GitHub users as a sign of quality recognition. Additionally, they took into consideration the GitHub issues and combined the GH Archive[25] dataset and data obtained from the GitHub API. They concluded that the number of reported fixed bugs is negatively

[24]TraviCI, `https://travis-ci.org/`
[25]GH Archive, `http://www.gharchive.org/`

affected for the projects with many developers making many contributions to different projects. In [53], they analysed the impact of CI on development practices. They observed more pull requests being closed after CI adoption. They combined the GHTorrent and the TravisTorrent datasets and also carried out a survey among some of the CI adopters. A different approach is reported in [54] where authors surveyed 307 active GitHub developers in regard to security. They were interested in how years of experience affect their approach to security problems and found the relation to be statistically significant.

In [55], the authors designed and implemented a framework and a tool to identify repositories containing well-engineered software projects. They were looking for elements such as community, continuous integration, documentation, history, issues, license, and unit testing. As a proof of concept they used the GHTorrent database and they discovered that 24.07% of projects on GitHub are well-engineered. In [56], they were mining the GHTorrent metadata in combination with the StackOverflow metadata to analyze developers' behavior and discovered that active issue committers from GitHub are also question askers from StackOverflow. In [57], the authors obtained the GitHub metadata through the GitHub API to develop rules for measuring project success related to the the number of downloads. The authors in [58] did an interesting analysis of the GHTorrent dataset. They observed that by far the most frequent event on the GitHub platform is a push. They also depicted an hourly commit distribution, however they based it on the UTC timestamp since this is the time GitHub records for each event. Therefore, the resulting distribution does not accurately capture the real hourly activity from the developers' perspective.

In [59], the authors analysed how timezone dispersion or concentration of developers affects the distribution of work activity in projects. They collected their own dataset based on a subset of 223 projects from GitHub. They discovered that the developers usually come from the same timezone and that they commit at various hours. A slightly different approach was taken by the authors in [15] who collected the dataset through the Visual Studio IDE using the FeedBaG++ tool to do the empirical study on the relationship between developer's working habits and efficiency. The results show that weekends exhibit different behavior than workdays which we also took into consideration in our study.

Knowing the distribution of all commit frequencies is a fundamental part of understanding software development processes. In [60], the authors use the database of the Ohloh.net open source project index to analyze the time elapsed between two separate commits of the same developer. They discovered that the developers have small commit frequencies, on average one commit each 3.2 days. However, 50% of developers have less then 13.8 hours between their commits, which is more realistic for professional development and we also take this into account in our study.

Extensive research on the development activity was done also in relation to the use of CI and DevOps tools. In [61], the authors did an analysis on how adopting CI affects the time to deliver merged pull requests. They observed that only 51.3% of the projects deliver merged pull requests more quickly after adopting CI. The data for the research was collected through the GitHub API. In [62], the authors did an empirical study of predicting the result of a CI build using the TravisTorrent dataset. They concluded that in an on-line scenario, the prediction is not very successful. In [63], the authors were investigating the parallel scheduling for distributed CI by using multiple CPU cores. They observed an exponential speedup when adding multiple CPU cores. In [64], they analyzed the influence of the CI introduction in GitHub projects on the commit activity. The research showed that there is no significant impact on developers' commit activity after adopting the CI. The authors based their analysis on the GHTorrent and the TravisTorrent datasets. A different study is reported in [65] where the authors performed an analysis of a performance monitoring framework overhead. The framework was integrated in Jenkins CI [66]. The

authors concluded that runtime monitoring during CI cycle does not introduce substantial overhead, therefore it should not be omitted.

Based on a literature overview we can conclude that a lot of work has been done on analyzing the GHTorrent dataset. However, we found this dataset to be quite complicated for use, therefore we decided for another project which also provides public GitHub metadata in a year by year SQL table format and is hosted on the Google BigQuery cloud platform. The GH Archive is a project behind the archive of public GitHub activity and makes data easily accessible for further analysis. GH Archive has recorded all public activity on GitHub from 2011 to 2018, in total several TB of data. GH Archive is mostly focused on the metadata of events. On the other hand, GHTorrent is mostly focused on the content of events [51], [67].

## 2.5 Summary

Based on the research conducted for this thesis we have observed a trend in the software development which goes in the direction of agile organizational as well as development practices. Furthermore, the infrastructure and tools supporting the development are increasingly focused on automation of previously expert manual work, therefore sparing the developers' time to focus on actual software features rather than on integration, testing and building of releases. Finally, we provided the literature overview of the developer activity related-studies with the emphasis on suitable datasets for such investigation.

# Chapter 3

# Analysis of Requirements and Challenges for CI

In this chapter, we analyze the requirements of wireless technology developers in terms of testing to enable CI practices in their testbed infrastructure. Moreover, we provide and discuss open challenges for wider use of CI practices in wireless technology development. Software development and testing is a complex process involving skilled people and infrastructure. A mix of evolving organizational practices, development practices and infrastructure contribute to the efficiency of the overall process which is relatively well-understood for general purpose development. Development of wireless firmware and its testing in a real environment, however, is notably more challenging because the code needs to be compiled, uploaded and tested on target embedded hardware instead of a general purpose computer. This requires purposely developed testing infrastructure that needs to be optimally dimensioned to satisfy the needs of the development team. The conclusions of the analysis were subsequently used to define a proof-of-concept reference architecture of the framework for controlled testing of multi-technology wireless networks as a proposed upgrade of an existing wireless experimentation testbed with new software and hardware functionalities.

This chapter provides contribution C2 in Section 3.3, respectively, and is based on [68].

## 3.1 CI for Embedded Systems Development

Current development practices are moving from traditional towards agile methods [20]. In software development, the agile practices [30] are complemented by modern infrastructures such as the ones involving CI [9], that automate software integration and testing. Therefore, they increase the efficiency of developers. Developers require constant feedback of how their newly implemented code runs on target systems integrated with the code from other developers from the team. CI automates the integration, testing and reporting of software tests and should be designed and dimensioned in such a way as to ensure the test results are communicated to the developer in the shortest time possible.

There are other approaches to testing software and firmware [69]. One approach is to do manual tests executed by a test engineer. Manual testing is time-consuming and error prone. Furthermore, it is almost impossible to parallelize test execution. Another approach are the so-called nightly builds and tests, where the tests typically take a long time. Therefore, a single automated test is executed during the off-working hours, typically at night, and the report is finished by the morning for developers to start fixing potential glitches found during the nightly test.

Provisioning test infrastructure for general purpose development of PC or web applications is a relatively well-understood process. Figure 3.1 depicts the comparison between software development on a PC with a full operating system and a microcontroller where the developed firmware typically compiles in raw machine code. PC software can be tested on any general purpose operating system locally or in the cloud, as it does not require any real-world interaction. Multiple tests can run in parallel and virtualization or containerization tools can be used for better test isolation, hence general purpose software testing capacity is auto-scaling by default.

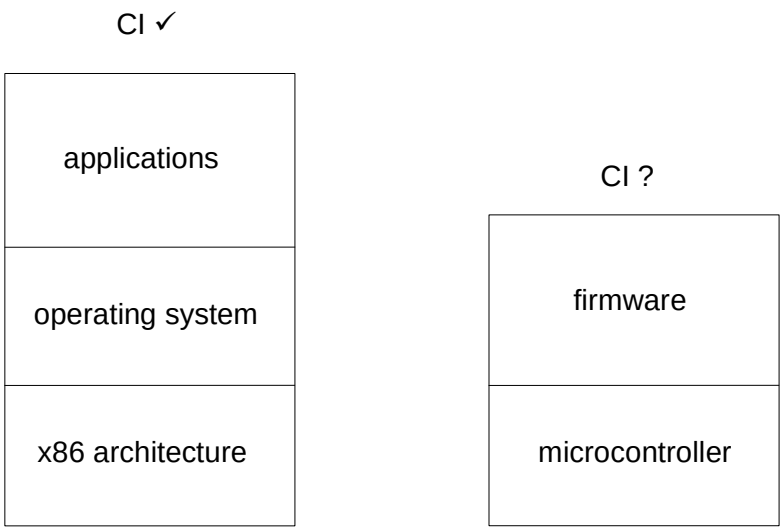



Figure 3.1: Existing CI usage on a PC and a new way of CI integration in an embedded microcontroller development process.

Firmware testing, on the other hand, is more challenging, because the code needs to be compiled, uploaded and tested on target embedded hardware instead of a general purpose PC. Moreover, for the case of wireless embedded hardware systems, the code needs to be tested on real hardware target devices deployed in a (close to) real operating environment testbed characterized by specific challenges. For instance, multiple distributed test nodes may be affected by a single wireless test, i.e. a transmitter and a receiver plus the nodes affected by the wireless interference caused by the test itself. Unlike for the case of general purpose computing, it is currently infeasible to provide on-demand and scalable test resources for wireless embedded systems, so they need to be carefully dimensioned and planned. While a recent trend towards softwarization of radio functions similar to Software Defined Networks (SDN) and Virtual Network Function (VNF) [70] might solve the on-demand availability of devices, the effects of propagation in real environment will still limit the scalability of test resources.

## 3.2 Challenges in Wireless Embedded Systems Development

For wider adoption of CI practices in the development and testing of wireless technologies, we identified three groups of challenges. The first group revolves around the constraints posed by the physical environment; the second one is concerned with process automation; and the third group is related to the lifecycle management of the wireless product.

### 3.2.1 Challenges Related to the Physical Environment

A CI system for wireless networks comprised of distributed hardware has to take into account that there may be unpredictable radio interference on the physical transmission medium or failures in hardware, and it must be ensured that the test reports to the developer are immune to them. In other words, from the test reports, it should be clear if the test failed because of software issues or because of conditions in the physical environment. This can be addressed by multiple repeated tests in time and space, which is related to reproducibility of tests and reliability of testing hardware.

- Reproducibility: A possible approach for ensuring reproducibility of the test results in the unpredictable wireless environment is for the CI framework to use spectrum sensing to identify a suitable test channel and thus avoid interference. Additionally, using controlled interference in test cases can sometimes prove useful to verify the interference-specific performance of a particular wireless algorithm.

- Reliability: A possible approach for ensuring reliability is to have redundancy in terms of distributed hardware and perform the same tests on more disjoint subsets of testbed devices.

### 3.2.2 Challenges Related to the Process Automation

With the adoption of CI practices in the proposed framework, it should be possible to realize automated system tests also in wireless embedded systems development. However, while the CI system and process are fully automated, the detection of errors caused by the physical environment is currently manual. This is an issue specific to wireless environments and has yet to be rigorously addressed. Additionally, the developer of the test currently needs to manually select one or more testbed devices to be used in the test.

- Automated tests: A possible approach for enabling fully automated tests is to design and develop a mechanism that would analyze spectrum sensing data in correlation with the tests and repeat the test if it failed because of a temporally jammed channel. This process could be made transparent to the developer.

- Automated resource allocation: A possible approach for enabling automated resource allocation is to develop a mechanism that would automatize the resource (i.e., testbed device) assignment for individual test execution. It would be interesting to see if the concurrent tests could be somehow categorized and how this information could be utilized by an algorithm for automatic selection of testbed devices to optimize test execution and testbed utilization while keeping test results reproducible as well.

### 3.2.3 Challenges Related to the Lifecycle Management of the Wireless Product

Continuous software development should also consider CD of the builds in production as the next step after adopting CI and thus make a step towards the lifecycle management of the embedded wireless devices, which are typically not being maintained after entering the market. CD could also solve security issues in devices on the market. Currently, every electronics producer addresses the problem of CD separately; however, it would be beneficial to find some standard method.

## 3.3 Infrastructure Framework for CI Process

There are four main functionalities that wireless network developers and testers need from a testbed environment:

1. Simple integration of software and hardware components that require testing; the priority is to minimize the overhead that the testbed usage introduces into the technology development cycle.

2. Controllable generation of traffic in the network under test, such as on-demand packet transmission according to custom patterns and distributions, to test new network protocols under realistic traffic conditions.

3. Controlled emulation of a radio operating environment using various technologies in order to identify potential coexistence issues between new and legacy technologies as early in the development cycle as possible.

4. Monitoring and analysis of network and radio spectrum metrics.

The introduction of the CI process into the embedded wireless technology development significantly lowers the barriers to the first and last of these functionalities, while relying on the other two to successfully carry out the CI tests. The framework we are proposing should seamlessly incorporate features of a wireless testbed with the CI development practices.

According to the existing software development practices, approaches and tools extensively covered in Sections 2.1 to 2.3, the proposed infrastructure framework for extending the CI practices with support for wireless firmware development should properly interface three main entities depicted in Figure 3.2:

**Repository** that holds the code, configuration files, tests and documentation, i.e., everything needed for the automated build. Generally, the repository is implemented using VCS tools as described in Section 2.3.1.

**Automation system** that performs all the instantiations and builds and transfers software components. Its implementation typically includes a large set of tools and components. In modern software development, the most efficient approach is to use web and containerization technologies.

**Testbed device** is used to run the tests and experiments. It can be off-the-shelf equipment, a custom-built device or a hybrid. The first approach is more suitable for improving existing technologies, while the second is for developing new ones. For the sake of generality, we assume that a testbed device comprises separate functional blocks represented by the infrastructure node for device management and the target node for test execution.

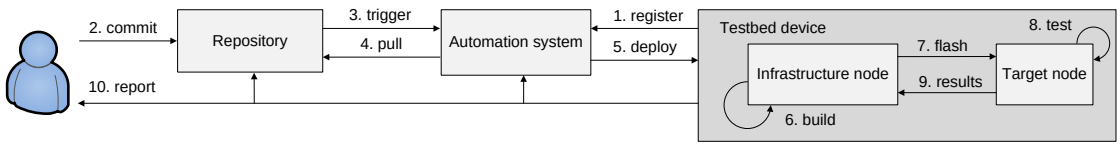



Figure 3.2: CI process framework.

As depicted in Figure 3.2, the CI process begins in step 1 when a new testbed device automatically registers itself in the automation system and becomes available for testing. In step 2, the developer commits new code to the repository, initiating step 3, which consists of triggering the automation system. In step 4, the automation system pulls the changes from the repository and executes the deployment scripts specifying the testbed devices required by the test case. Step 5 deploys the changes from the repository to the testbed device and instructs it to start the build process. In step 6, the testbed device initiates the build process on the infrastructure node. The build system is contained inside a container; therefore, the infrastructure part of the testbed device stays unaffected by any build failures or run-away processes. If the building process succeeds, the infrastructure node in step 7 flashes the target node with freshly built firmware and in step 8 executes the test process. In step 9, the infrastructure node obtains the results from the target node and examines them. In step 10, the automation system updates the repository with the result and communicates the result to the developer, typically via email.

## 3.4 Summary

In this chapter, we analyzed the requirements for the framework that can be implemented on a state of the art testbed suitable for experimentation with wireless embedded systems. Next, we discussed the challenges when introducing CI practices for wireless network development. Finally, we proposed the framework, which simplifies the code development process on real wireless testbeds and enables frequent code changes by developers.

# Chapter 4

# Implementation of a Framework for CI in Wireless Technology Development

Given the increasing complexity of wireless technologies, evaluation and testing of new devices, protocols or solutions in realistic environments is desirable but tedious and expensive. Although complicated and expensive to build and maintain, distributed and heterogeneous wireless testbeds offer the most realistic testing conditions [12]. While the cost of adding additional hardware functionality to support emerging technologies is difficult to overcome, the cost of developing software for the increased variety and number of wireless devices can be significantly reduced using continuous integration (CI) principles and tools [71]. In particular, FOSS software projects have benefited significantly from adopting CI principles that significantly reduce the workload of the core maintainers. Contributed code can be automatically tested for syntax, style and errors using sophisticated toolchains that automate previously manual development practices. Studies have shown that the CI methodology significantly increases productivity and code quality [72].

A recently proposed software-centric systematic testing methodology for mobile networks [73] incorporates unit testing, integration testing and system testing, which are central to software development and represent an important part of the CI process. Testbed support enabling such a methodology can be economically implemented using modern tools, such as software containers, that can be automatically deployed to test new code.

In this chapter, we present the reference architecture and implementation of the framework for ContinuOus IntegratioN in wirelesS technology development (COINS). This framework can be used by developers to enable conventional CI development practices on embedded wireless testing infrastructure. To illustrate and facilitate the discussion on framework capabilities, we upgraded the existing LOG-a-TEC testbed [43], [44] for controlled testing of emerging multi-technology wireless networks to enable CI. LOG-a-TEC is a combined outdoor and indoor heterogeneous wireless testbed for experimentation with sensor networks and machine-type communications[26].

This chapter provides contributions C4, based on [68] and C6 in Section 4.1.1.3, based on [74], respectively.

[26] LOG-a-TEC, `http://log-a-tec.eu/`

## 4.1 Continuous Integration Support System

The architecture of COINS is illustrated in Figure 4.1 and further explained in this chapter. It consists of the three main architectural components identified and described in Chapter 3.3: repository, automation system and testbed device, which are mapped on the elements of the existing LOG-a-TEC infrastructure [43], [44]. The COINS architecture is implementation independent with well defined modules which interact through specified interfaces for which we provide a reference implementation although another implementation using a set of different tools is certainly possible.

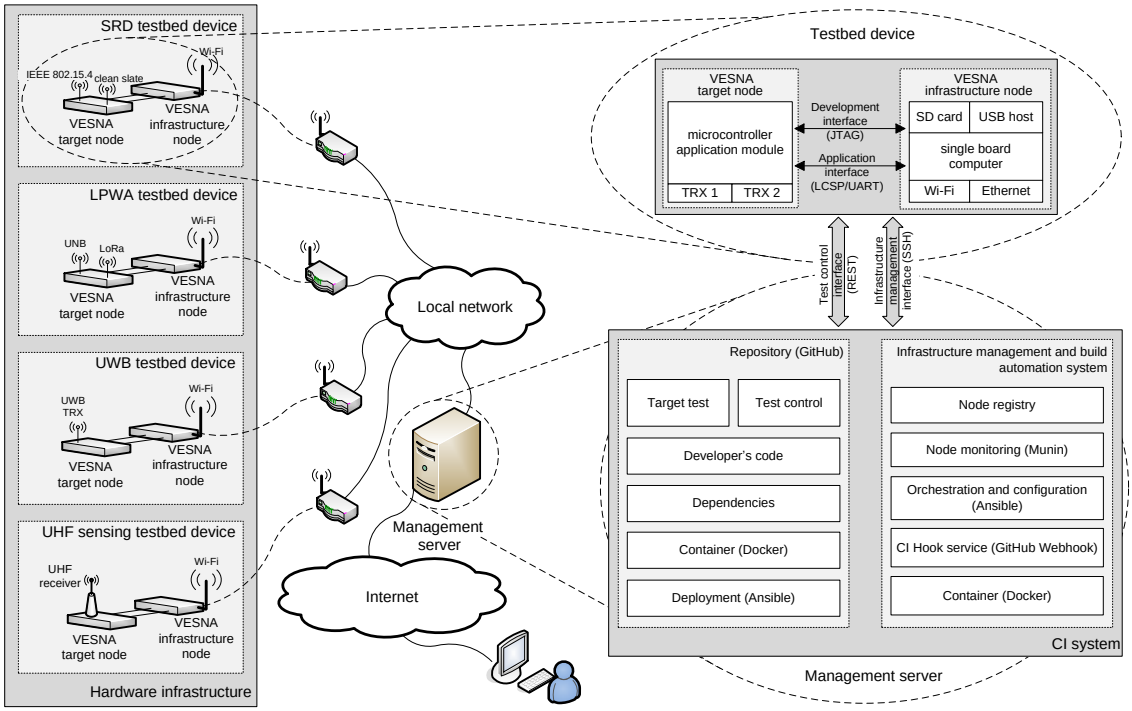



Figure 4.1: Architecture of the reference implementation.

For the reference implementation of COINS[27], we chose widely used FOSS tools. This way, COINS benefits from and contributes to the work of established communities and is more likely to be adopted. In particular, the container system is based on Docker[28], which we use to package software for distribution to nodes in the testbed. The CI hook service adds support for the GitHub WebHook[29] client. We use Ansible[30] to describe and automate tasks. Ansible is a popular automation engine for configuration management and software deployment. The networked resource monitoring tool Munin[31] serves for testbed device monitoring. We also implemented a custom node registry system and released it as free software under the AGPL license.

[27] COINS, `http://github.com/sensorlab/SensorManagementSystem/`
[28] Docker, `http://www.docker.com/`
[29] GitHub, `http://developer.github.com/webhooks/`
[30] Ansible, `http://www.ansible.com/`
[31] Munin, `http://munin-monitoring.org/`

### 4.1.1 Testbed Devices

The hardware infrastructure is composed of several types of testbed devices, as depicted on the left side of Figure 4.1. The testbed device comprises two VESNA (VErsatile platform for Sensor Network Applications) based nodes, as depicted in the top right part of Figure 4.1.

The testbed device depicted in Figure 4.2 can use several wireless technologies, i.e. short-range (SRD), low power wide area network (LPWA), ultra-wideband (UWB) and UHF spectrum sensing as depicted on the left side of Figure 4.1. Architecturally, we consider hybrid testbed devices with two separate functional blocks represented by the infrastructure node and the target node. The main reason for the separation of the testbed device into two functional blocks is to have a generic infrastructure node that can be combined with various target nodes to support a range of heterogeneous experiments with a unified way of management and reconfiguration. Another reason is to achieve hardware separation between robust and stable testbed management functionality on the infrastructure node and the experimental code, which is prone to errors and crashes on the target node. The infrastructure node can also support containerization, while the target node is often based on an embedded system with limited hardware resources.

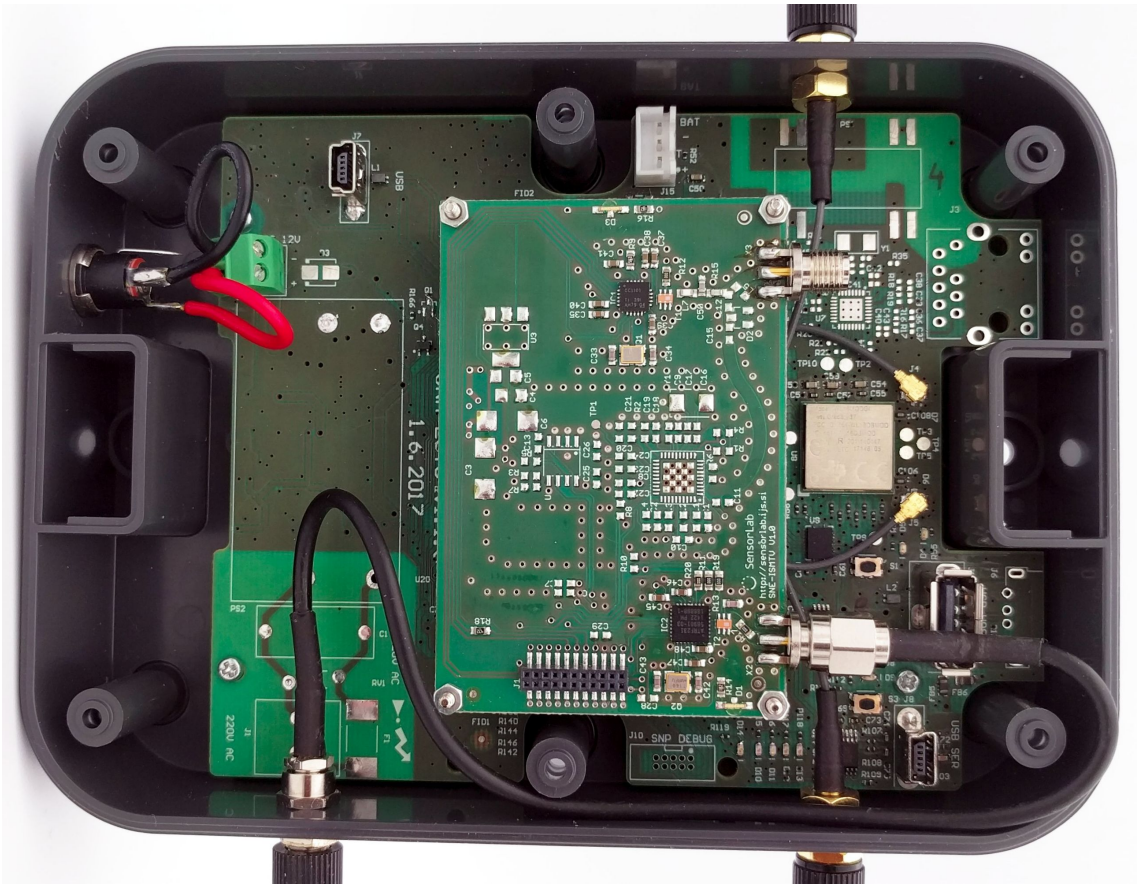



Figure 4.2: Testbed device: Infrastructure node with attached target node.

The integration of the two is done via custom hardware design and the adoption of selected interfaces and protocols, thus going beyond similar approaches such as [75]. Both nodes are interconnected via application and development interfaces, providing the exchange of application data as well as remote low-level application debugging. The reliable development interface is based on JTAG and the application interface on protocols such

as LCSP (Light-weight Client Server Protocol) or SLIP (Serial Line Internet Protocol) running on top of serial interface.

#### 4.1.1.1 Infrastructure Node

In our implementation, the infrastructure node is a custom-designed single board computer based on the BeagleCore module running the Debian Linux operating system. It features wired Ethernet and WiFi for infrastructure connectivity, an SD card slot for storage expansion, VESNA and USB interfaces for modular extensibility and a development/debug process as depicted in Figure 4.3.

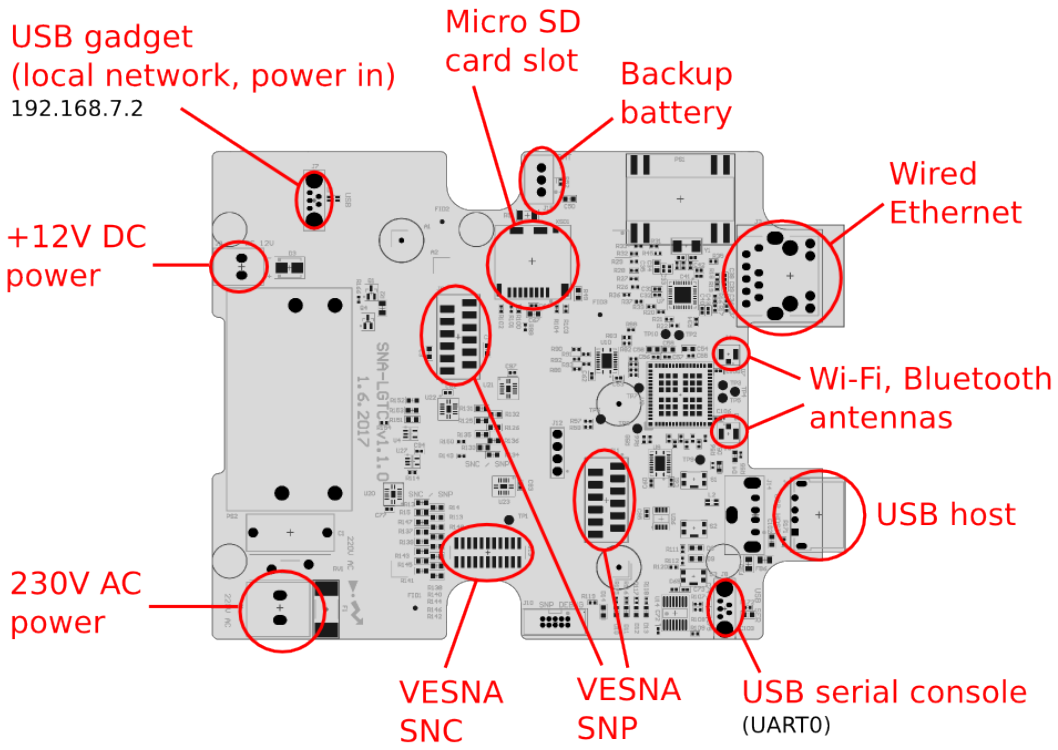



Figure 4.3: Infrastructure node connectors.

For the distributed system of infrastructure network, an OSGi approach is proposed in [75], however such a high level ecosystem represents unnecessary system lock-in which can be avoided by choosing a more general purpose approach as an integration level. We chose SSH protocol as an integration protocol that combined with the state of the art Ansible software provisioning system becomes a cornerstone for a scalable and adaptable distributed system. Authors in [76] recommended using similar systems, i.e. Chef or Puppet, as also described in Section 2.3.2. However, both of them require a special agent running on each node, which introduces a potential point of failure which we would like to avoid. Ansible is agent-less and therefore a more reliable system which is especially important in cases where nodes cannot be accessed easily.

#### 4.1.1.2 Target Node

The target node is a custom VESNA device with an ARM Cortex-M3 microcontroller application module and dedicated experimentation transceivers as depicted in Figure 4.4. VESNA device can run a dedicated OS (e.g., Contiki) or custom firmware.

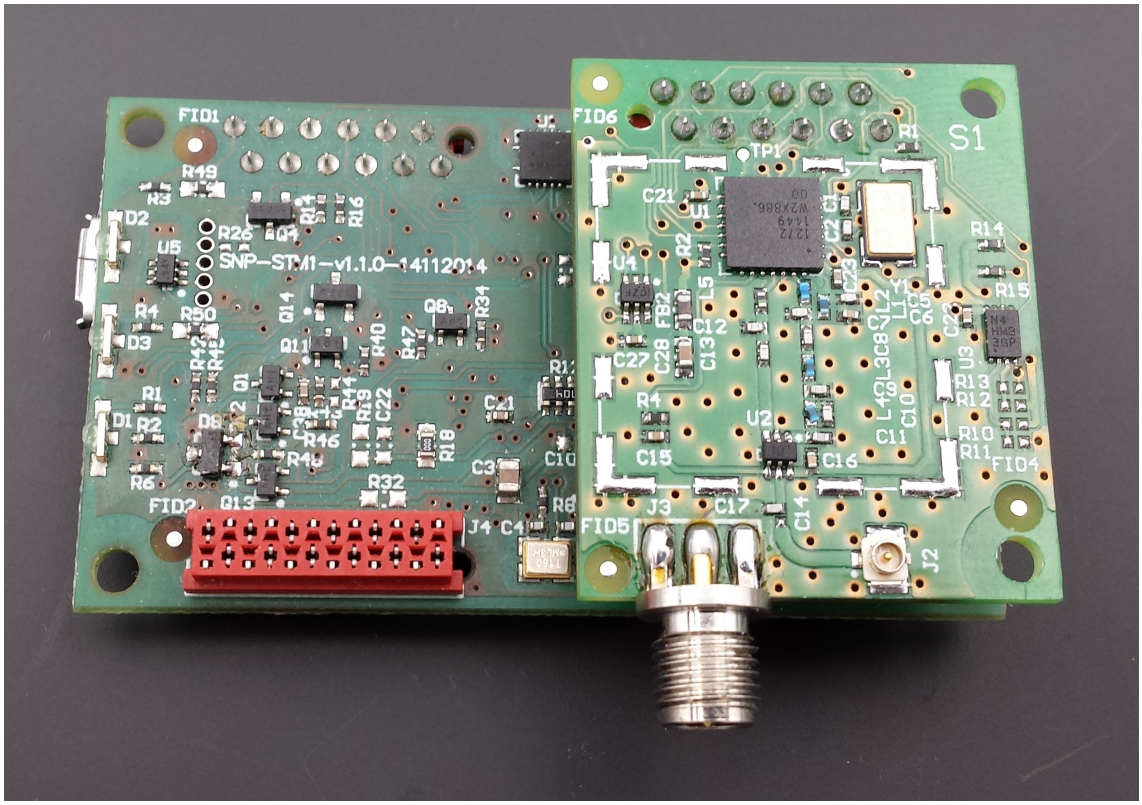



Figure 4.4: Target node with attached LoRa experimentation transceiver.

The target nodes host several different radio interfaces and thereby support testing in various frequency bands. In particular, we classify them according to supported experimentation with:

- LPWA communications,
- short range UWB communication,
- advanced spectrum sensing in sub-1 GHz bands,
- efficient lightweight protocols in 2.4 GHz and sub-1 GHz bands that are not based on IEEE 802.15.4 (clean slate), and
- spectrum sensing and signal/interference generation by reconfigurable transceivers.

#### 4.1.1.3 Communication Between Infrastructure and Target Node

In our proposed implementation the test description is based on the custom-developed protocol LCSP (Lightweight Client Server Protocol) for communication between the VESNA target and the VESNA infrastructure node as depicted in Figure 4.5. On top of this protocol we implemented an abstraction library called ALH Tools which serves as a basis for developing distributed wireless tests and experiments.

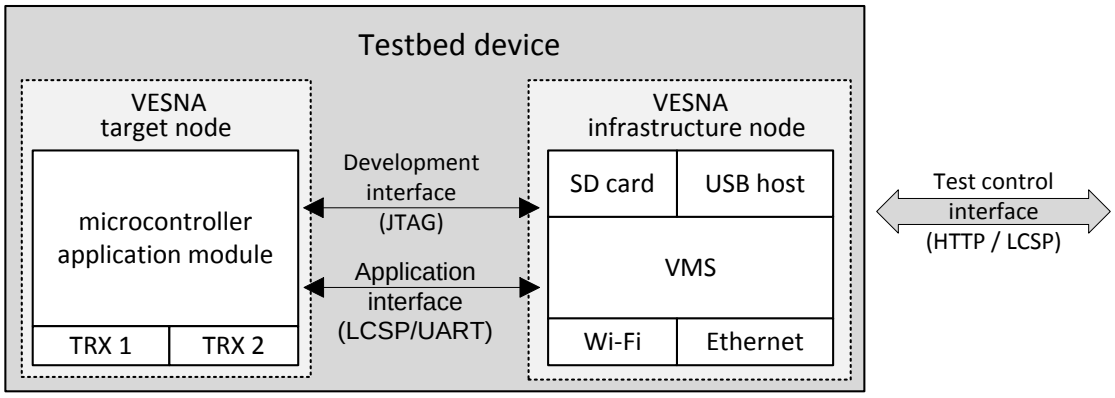



Figure 4.5: Communication within the testbed device over LCSP protocol and outside communication over REST API serving as HTTP / LCSP proxy.

The LCSP protocol is based on a client-server architecture. In our case the server side is on the target node and the client side is on the infrastructure node hosting the VESNA Management System[32] (VMS) tool, which supports the LCSP protocol and is not directly related to the management server from Figure 4.1. In order to access the exposed resources of target nodes, the connection with the infrastructure node needs to be established over a serial, TCP or SSL link. Resources that are pre-prepared on the target nodes are exposing different features.

LCSP protocol was initially developed for communication between a sensor network and a remote server. It was inspired by the HTTP protocol, but purposely kept simple for fast and easy implementation on VESNA nodes with restricted capabilities. The protocol defines two basic methods, GET and POST, which are understood by each VESNA node. The GET method is used for "safe" requests which do not change the state of the system. The POST method, on the other hand, is used for "unsafe" requests which can change the state of the system. The response is considered to be in a binary format although most messages are in a text format. Each response ends with a sequence OK\r\n to mark the end of the response. The protocol format of the request and response is shown in Listing 4.1.

The LCSP protocol includes simple and efficient error handling mechanism. There are two types of errors that can be encountered as a response to requests. The first type is JUNK−INPUT, which is the more common situation when someone mistypes the resource name and the parser on the node does not recognize it. After this response, the parser on the node expects 5 new lines which reset the parser before one can try to access the resource again. The second type of error is CORRUPTED−DATA, meaning that cyclic redundancy check (CRC) did not pass successfully. Thus we can conclude that the error happened somewhere on the link between the client and the server. The probability of this error is very low.

[32]VMS, `https://github.com/matevzv/vesna-management-system/`

Listing 4.1: LCSP request and response format

```
Client requests must be formated:

GET resource?arg1=val1&...&argN=varN\r\n

  resource: abstract resource identifier
  examples: - firmware/version
            - sensors/temperature
  arg1:     parameter 1 name
  val1:     value of parameter 1
  ...
  argN:     parameter N name
  valN:     value of parameter N

POST resource?parameters\r\n
Length=len\r\n
<data having len bytes length>\r\n
crc=crc_value\r\n

  resource:   abstract resource
              for example: firmware
  parameters: parameters given to the
              POST handler same format
              as GET parameters
  len:        length of the data that
              will be written to the
              specific resource
  data:       possibly binary data
              to be transmitted
  crc_value:  CRC value calculated on all
              the previous content except
              the line starting with crc=;
              value represented as an
              unsigned decimal number

Server responses must be formated:

<response to a specific request>\r\n
\r\n
OK\r\n

Error messages must be formated:

<response with error description>\r\n
<JUNK-INPUT or CORRUPTED-DATA>\r\n
\r\n
OK\r\n
```

Additionally, VMS implements a REST API which translates HTTP requests to LCSP requests as depicted in Listing 4.2. The call to the API has to meet the specified form for GET and POST request. In order to make a GET or a POST request we have to make a call to the handler called "communicator" located in the web server inside the management system. Requests that do not meet the template are rejected by the VMS. The call includes several parameters:

- Cluster - corresponds to the serial/TCP/SSL/ port which VESNA node is connected to.
- Method - can be GET or POST.
- Resource - corresponds to the resource name located on the target node.
- Content - a specific parameter of POST requests specifies the data transmitted to nodes.

Listing 4.2: LCSP / HTTP proxy request example

```
api?cluster=port&method=get&resource=name&
...method=post&resource=name&content=content
```

The ALH Tools[33] is a python library which provides utilities and modules for managing VESNA-based wireless sensor networks based on LCSP protocol in an object oriented manner. In a typical setup, VESNA nodes participate in the network with an infrastructure node connected over serial/TCP/SSL communication link. In this network, each sensor node exposes LCSP API, supporting two types of requests: GET and POST. The tunnel terminates in an infrastructure node hosting VMS that performs the translation between the LCSP protocol and a proper HTTP REST interface exposed on the web.

Alternatively, the target node can also be directly connected to the infrastructure node over a serial line. This setup is typically used for the development, debugging or testing. The ALH Tools with LCSP protocol transparently support both modes of operation. Optionally, either an URL of an HTTP REST endpoint is given or a character device for the serial line. The ALH Tools library is implemented in a way that abstracts the LCSP protocol complexity and enables fast composition of tests. The usage of the ALH Tools library on an example of LoRa firmware CI test can be found in Appendix A.

### 4.1.2 Repository

The repository support was added to the CI system to abstract the underlying complexity and enable the CI process described in Section 3.3. It includes the following features:

- the target test to be executed on the testbed device;
- the test control that runs on the management server and communicates with the testbed device;
- the developer's code that has to be tested;
- dependencies such as software libraries;
- the container file, which describes how to package the target test, test control, dependencies and developer's code into a container; and

[33] ALH Tools, `https://github.com/matevzv/vesna-alh-tools`

- the deployment configuration, which specifies where and how to build and run the container.

Our implementation of the repository, is based on cloud platform GitHub for hosting source code as depicted in Figure 4.6. The GitHub platform uses Git as a VCS, hence our implementation of repository is also based on Git. The target test and test control are realized using C and Python, but in general, the proposed framework is programming language independent. The container support is implemented through Docker, while the deployment is automated by Ansible. The repository that contains everything needed for the completely automated build is realized through a GitHub repository. Examples of more advanced tests and experiments are given in Section 4.3.

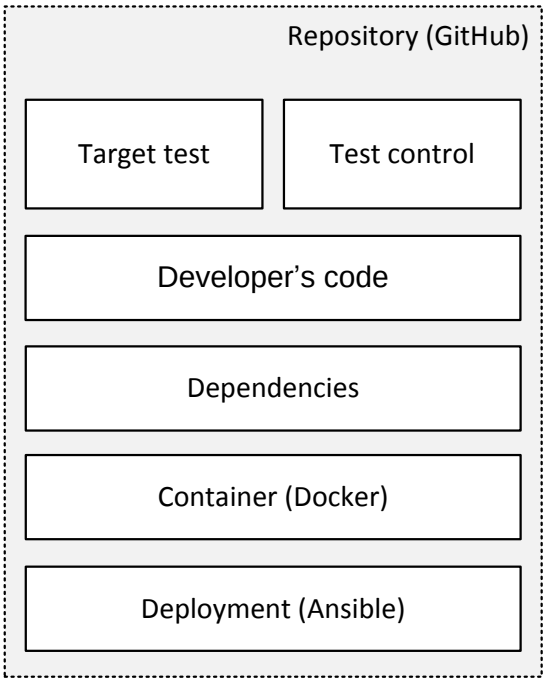



Figure 4.6: Repository implementation.

### 4.1.3 Infrastructure Management and Build Automation System

The infrastructure management and build automation system depicted in Figure 4.7 is implemented as a complex setup composed of self-sufficient systems packaged in a Docker container following the microservice architecture approach. Each container includes all dependencies with the purpose of being easily redistributable. In particular, the infrastructure management and build automation system contains:

- the node registry, which has a list with details about all testbed devices;
- the node monitoring service, which is in charge of monitoring the activity and health of testbed devices;
- the orchestration and configuration, which performs regular maintenance and reconfiguration of the testbed;

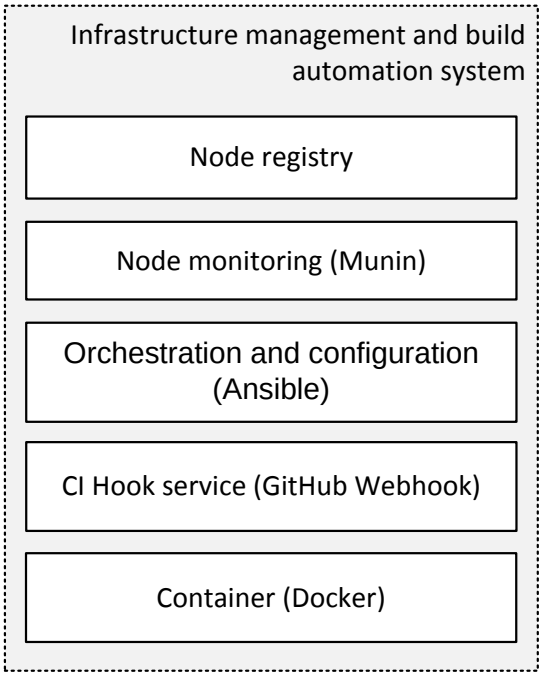



Figure 4.7: Infrastructure management and build automation system implementation.

- the CI hook service, which waits for signals of changes in the repository and initiates the build process; and
- the container file, which describes how to package all the previous components in a Docker container.

There are several periodic background jobs running on the management system that perform house-keeping tasks:

- Building the Ansible host file from the node registry database.
- Updating the information on the availability of individual nodes using the Ansible ping command.
- Stand-alone testbed device monitoring based on Munin.
- CI Hook service triggering the build process.

Externally, the management server exposes a user-facing web interface and an HTTP REST API [44]. The infrastructure management and build automation system allows browsing through clusters of individual devices and nodes in the database, positioning devices on a map, visualizing reports, monitoring device activity, etc. Thus, it gives an instant overview of the state of the testbed. Depicting the testbed devices on a map is particularly useful for the selection of devices for different wireless test cases.

For the reference implementation we decided to use FOSS tools and services whenever possible. The management part of the interface consists of a dashboard based on the Rundeck system[34] with the added support for Ansible orchestration[35]. The nodes need to

[34]Rundeck, `http://rundeck.org/`
[35]Ansible, `https://www.ansible.com/`

be provisioned before the deployment with a public SSH key of the management server, to later allow execution of system commands over Ansible SSH. The dashboard has options to either execute a single command on one or more nodes or define a job based on an Ansible playbook which can run once or periodically.

The web interface of the infrastructure management system also includes node monitoring information and statistics such as node uptime and system load provided by networked resource monitoring tool Munin[36] depicted in Figure 4.8. Warnings can be defined to trigger when specific monitored variables exceed a preselected threshold, such as running out of system memory or storage space. In such cases the system will send an email to the system administrator.

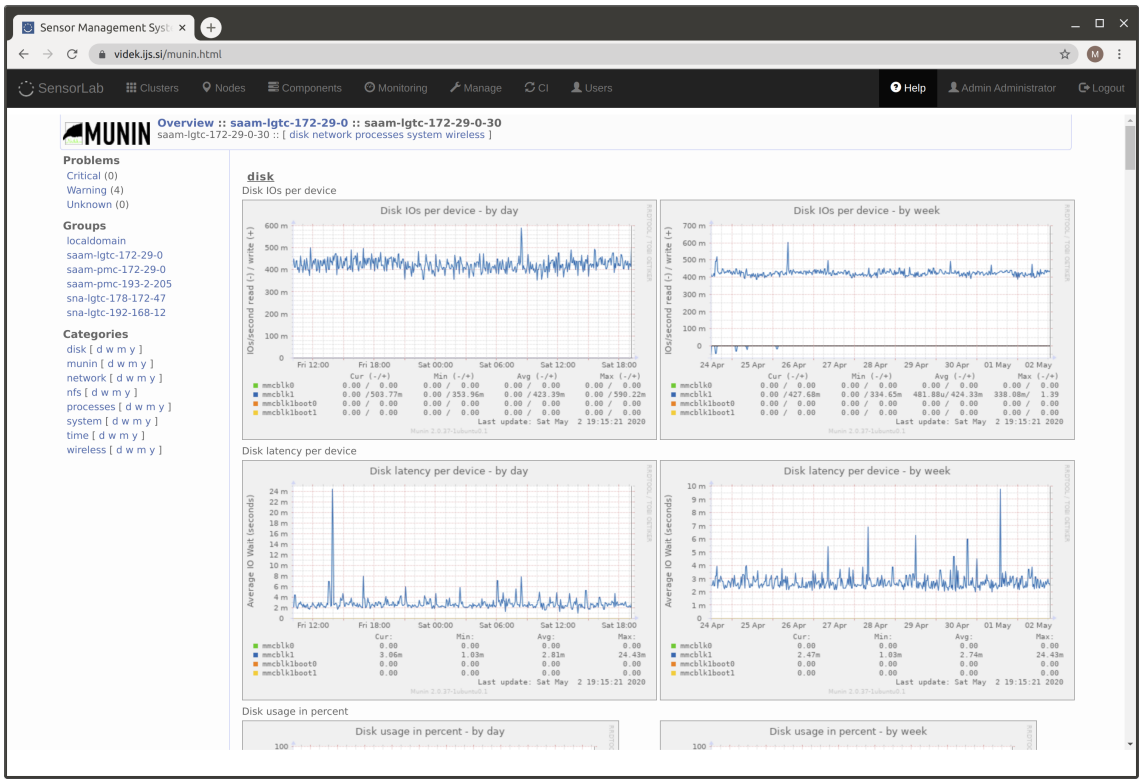



Figure 4.8: Node monitoring system Munin.

Programmatic access to the management system is performed through the REST API. For example, nodes use the REST API to make an automatic registration and update changes in configuration using a custom-developed node registry system Videk depicted in Figure 4.9. Users can also use the REST API to access information about the nodes and clusters programmatically.

Finally, the infrastructure management system runs the OpenVPN[37] virtual private network service. This service is useful when connecting nodes to the infrastructure management system from external networks. In a typical deployment on external networks, nodes are placed behind an IP network address translation (NAT) and/or a firewall that is not under the tester's control. This prevents direct IP connections from the infrastructure management system to the services listening on the nodes, such as the Ansible SSH, which is essential for node management. While this could be solved by modifying the firewall rules, this is often frowned upon by network administrators. Hence a more effective

[36] Munin, `http://munin-monitoring.org/`
[37] OpenVPN, `https://openvpn.net/`

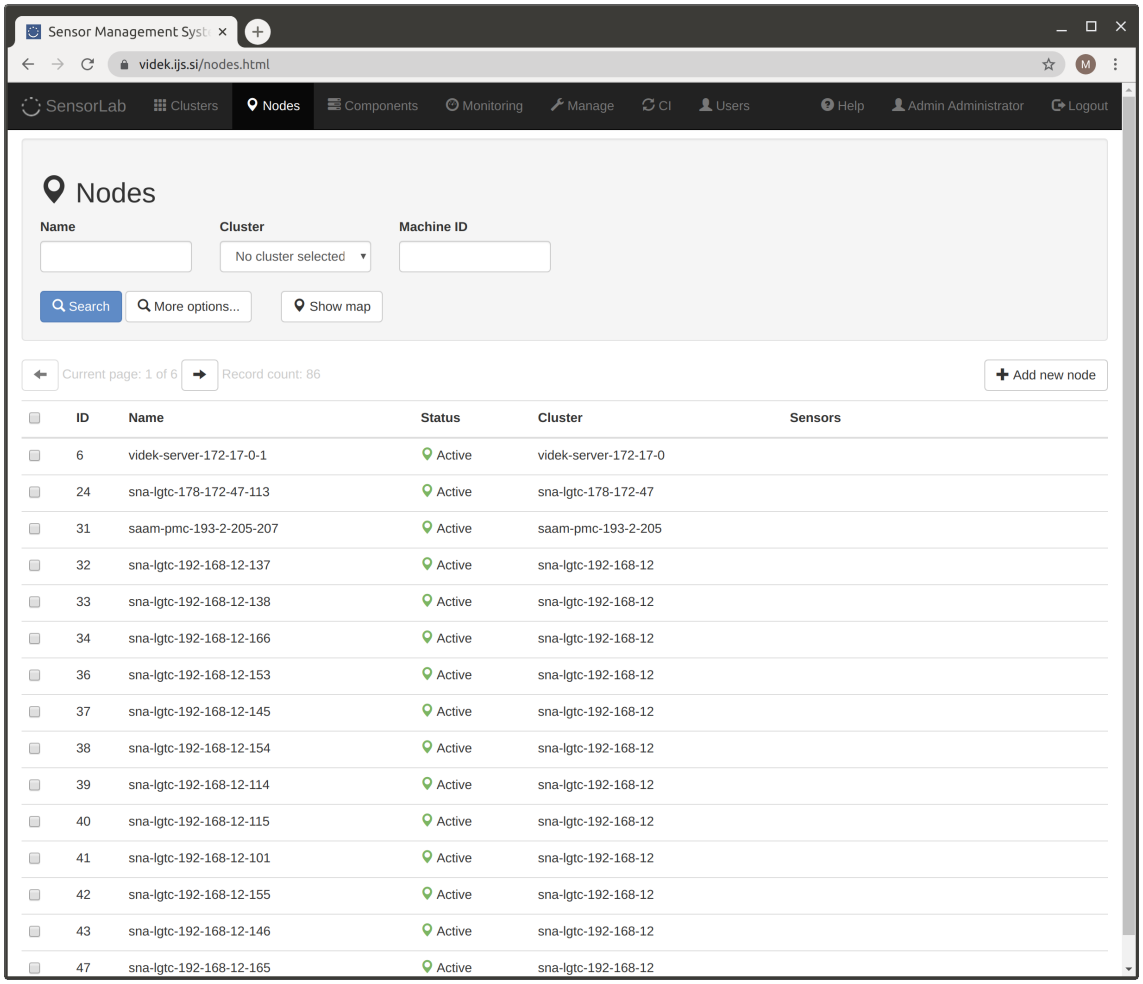



Figure 4.9: Node registry system Videk.

solution is to establish a VPN connection to external network nodes.

## 4.2 Automated Testing Implementation

The automated testing system was designed to abstract the underlying complexity and enable the workflow described in Section 4.1. Everything needed for automated tests is available in a GitHub repository[38] starting with the Ansible playbook which gets executed when new code is pushed to the repository by triggering the GitHub Webhook. It is listening on testing controller which is part of the central management server. The Ansible playbook contains a description of the test setup which will be executed on the target nodes and in the test controller. The testbed device will download the GitHub repository containing the Docker[39] file and the actual test code. The Docker image will not be built each time from scratch; the testbed generic image will be downloaded from Docker store, which has all the needed dependencies and services already installed and set up. Thus, the developer only needs to write the code for the actual test performed on the node and the part performed by the testing controller.

When the test is completed, the testing controller will automatically commit and push

[38] GitHub, `https://github.com/`
[39] Docker, `https://www.docker.com/`

the test results to GitHub, making them available to the developer for download and post processing. In the context of the testbed usage for CI, each time a developer contributes a change in the codebase, the test is executed on a testbed and results are returned to the developer, revealing if the test passed or failed. Complete testing orchestration is based on the popular open source automation server Jenkins[40].

An example test case for a wireless communication system includes software for three entities: the transmitter, the receiver and the test controller. The first two will run on the target node within the testbed device, while the third runs on the management server. The test controller instructs one testbed device to go in receive mode and another testbed device to transmit the test sequence. When the first testbed device receives the test sequence, it reports the results back to the test controller, which verifies if the received sequence matches the transmitted one as shown in Listing 4.3. If it does, the test succeeds; otherwise, it fails.

Listing 4.3: Wireless test example.

```
assertEqual(nodeTX.data, nodeRX.data)
```

The described testing approach was implemented in the testbed for the purpose of LoRa firmware development. The complete implementation of the test can be found in Appendix A. Especially in the industry, there are situations where multiple developers work on the same code-base simultaneously. In such environments it can happen that a fix from one developer breaks a feature of another. By introducing automated testing on a real testbed, the integration problems get discovered and fixed early in the development process.

## 4.3 Testbed Deployment and Supported Testing

In order to enable testing with the proposed framework in a dense and heterogeneous wireless environment, we upgraded the combined outdoor/indoor LOG-a-TEC testbed at the JSI campus as previously described. Table 4.1 summarizes the number and type of testbed devices in an indoor and outdoor environment, whereas the layout of devices in the outdoor environment is depicted in Figure 4.10 [68].

The outdoor part of the wireless testbed is located at JSI in and around the park area of 55 m by 60 m. Nodes are placed on light poles 3.5 m above the ground and on the surrounding buildings at heights from 2.0 to 9.3 m. To enable testing also in indoor as well as in mixed indoor/outdoor scenarios, the testbed is extended in the indoor environment with additional 20 UWB devices, three SDR devices and one LPWA device, deployed in second and third floors of the building with the dimensions of 28.4 m by 16.6 m. In addition, several portable devices of all types are at disposal for analyzing the use cases incorporating device mobility or further densification.

The diversity of types, relatively high number and density of testbed devices, and the possibility to host multiple radio technologies at most devices to transmit/receive/sense at the same time at the same location, enables testing scenarios in a very dense and heterogeneous wireless environment. The various types of tests include:

- investigation of assisted and non-assisted indoor/outdoor localization algorithms;
- custom and advanced spectrum sensing for dynamic spectrum access and cognitive radio networking;

[40] Jenkins, `https://www.jenkins.io/`

Table 4.1: Configuration and features of testbed devices.

| Node type | Radio | Frequency | No. of devices |
|---|---|---|---|
| Target node SRD A | Atmel AT86RF212<br>TI CC2500 | 868 MHz<br>2.4 GHz | 21 outdoor |
| Target node SRD B | TI CC1101<br>Atmel AT86RF231 | 868 MHz<br>2.4 GHz | 21 outdoor |
| Target node LPWA | Semtech SX1272 | 868 MHz | 3 outdoor<br>1 indoor |
| Target node UWB | DecaWave DW1000 | 3.5 - 6.5 GHz<br>(6 channels) | 11 outdoor<br>20 indoor |
| Target node<br>UHF spectrum sensing | NXP TDA18219HN | 470 - 866 MHz | 2 outdoor |
| Infrastructure<br>node | TI WL1837 | 5 GHz | 58 outdoor<br>21 indoor |

- link and/or network performance evaluation in controlled interference and/or heterogeneous radio environment;
- design and evaluation of new custom-developed wireless protocols and their benchmarking against the standardized protocols; and
- performance evaluation and controlled piloting of prototype IoT devices.

The testbed enables different testing with the dense IoT and MTC networks in the scope of the 5G networks. A wide range of radio interfaces supports testing with heterogeneous MTC technologies by transmitting/receiving/sensing at the same time in the same location using beyond state-of-the-art functionality like ultra-narrow band and ultra-wide band, packet-based experimentation, clean slate protocol design, composable and modular protocol stacks, custom and advanced spectrum sensing and signal generating functions in sub-GHz spectrum.

In order to support such diverse testing capabilities, the testbed comprises a set of libraries and tools that hide the implementation complexity from the developers. Thus, advanced software components and modules enable:

- out-of-the-box localization;
- Line-of-Sight/non-Line-of-Sight link classification;
- link quality estimation (LQE), classification and prediction;
- custom protocol stack development using modular components;
- spectrum sensing by energy detection, cyclostationary detection and eigenvalue- and covariance-based methods;

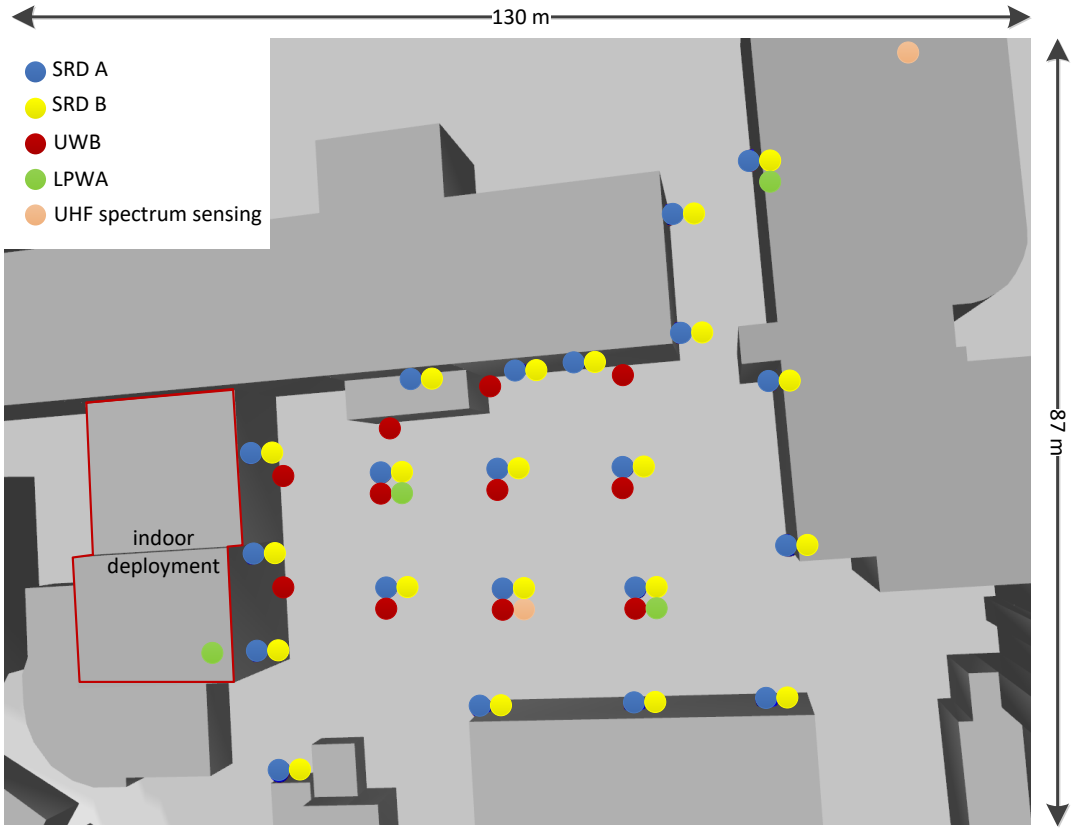



Figure 4.10: Locations of available testbed devices in an outdoor environment.

- signal generation including wireless microphone profiles; and
- game theoretic power allocation.

For evaluation and comparison of spectrum sensing algorithms the testbed enables generating controlled interference which also helps with reproducibility of tests and enables evaluation of the interference-specific performance of various higher-layer protocols. Such functionality is useful for any type of testing. Furthermore, localization and distributed spectrum sensing can be used to build geolocation spectrum occupancy databases, which in turn can be used in investigation of primary/secondary spectrum access regimes.

Through the availability of reconfigurable transceivers that are not limited to the IEEE 802.15.4 standard, new MAC technology can be developed. This is particularly relevant at the time when emerging non-IEEE 802.15.4 technologies such as Sigfox and LoRa for IoT are appearing on the market. In addition, the testbed enables the evaluation of the non-IEEE 802.15.4 MAC components alongside IEEE 802.15.4 MAC components which is particularly relevant for the developers who want to have a deep insight into what to expect when their technology has to co-exist with other standardized ones. It is also relevant for the large number of IoT companies developing their customized stack on top of IEEE 802.15.4 MAC.

## 4.4 Positioning and Comparison of Reference Framework

In the existing literature we did not find any other testbed framework addressing wireless experimentation with CI support. However, we identified three existing frameworks that are to some extent similar to our proposed framework COINS, i.e., the cOntrol and Management Framework (OMF) [77], the Network Implementation Testbed Laboratory (NITOS) [78] and the Berlin Open Wireless Lab (BOWL) [79]. We performed a feature-wise comparison of these frameworks and summarized it in Table 4.2.

In the conducted comparison we considered an extensive list of features that can be used to compare the experimentation systems [80]. The list of features, however, lacks the CI-specific properties that our work focuses on; therefore, we extended the list with four core CI concepts [9], [81]:

- Existence of a single source repository that contains everything needed for the completely automated build process.
- Support for completely automated build process executed on each commit.
- Self-tests included inside the repository, which run on each build.
- Fast build process so that each commit can be built and tested.

Table 4.2: Framework comparison.

| **Features** | **OMF** | **NITOS** | **BOWL** | **COINS** |
|---|---|---|---|---|
| Description language | OEDL | REST | UCI | Ansible/Docker |
| Experiment types | Any | Any | Wireless | Wireless |
| Operation | Testbed | Testbed | Prod./Testbed | Testbed |
| Interoperability | ✓ | ✓ | | ✓ |
| Reproducibility | ✓ | ✓ | ✓ | ✓ |
| Fault tolerance | ✓ | ✓ | | ✓ |
| Debugging | ✓ | ✓ | ✓ | ✓ |
| Monitoring | ✓ | ✓ | ✓ | ✓ |
| Data management | ✓ | | | ✓ |
| Source repository | | | | ✓ |
| Automated build | | | | ✓ |
| Self testing | | | | ✓ |
| Fast build | | | | ✓ |

For each feature presented in Table 4.2, we explain how we addressed the problem compared to the other frameworks. The last four features are CI-specific and are realized only in COINS.

### 4.4.1 Description Language

The description language is the language used to describe how and where to build and run the experiment. In OMF, the experiment is described using OMF Experiment Description Language (OEDL). The NITOS testbed is abstracted through a REST API. The BOWL system is based on the Open Wireless RouTer (OpenWRT) Unified Configuration Interface (UCI) scripts. In COINS we did not want to introduce a new language specifically for this purpose. Hence we used well-known tools that many developers already use in their development practices. In order to define what tests to run on which node and how to obtain test results we use the Ansible Playbook, a language for describing software deployment steps. For building our containers we use Dockerfile, a text format used by Docker to define container build procedures.

### 4.4.2 Experiment Types

The type of experiment describes the platform where the experiment is going to run. OMF and NITOS can support wired or wireless experiments, whereas BOWL is limited to WiFi-based wireless systems. COINS focuses on wireless systems and is suitable also for restricted capability devices. While OMF, NITOS and COINS are suitable for developing completely new as well as optimizing existing wireless solutions, BOWL is primarily suitable for optimizing existing solutions.

### 4.4.3 Operation

The operation refers to the ability of the testbed to also serve as a production system. OMF and NITOS are exclusively testbed frameworks, while BOWL is designed to work simultaneously as a testbed and production WiFi access network. In this respect, COINS is also exclusively a testbed framework.

### 4.4.4 Interoperability

Framework interoperability covers an infrastructure-independent design, support for existing infrastructure services, resource discovery and software interoperability. OMF and NITOS address interoperability issues, while BOWL UCI configuration scripts are implementation-specific. OMF can be extended, and NITOS specifies a testbed API. COINS addresses interoperability by offering a modular and implementation-independent design.

### 4.4.5 Reproducibility

Reproducibility refers to the ability to automatically run the same experiments multiple times. OMF enables complete reproducibility, while NITOS offers limited reproducibility since it implements automated node selection based on availability. BOWL also addresses the problem of making experiments reproducible. COINS supports reproducibility of test cases by creating tags inside the repository and manually choosing the devices used in the test. While declaratively all frameworks address reproducibility, we argue that it may be severely affected in case of missing repository support on the framework level and challenged by the ability to control the conditions in the physical environment.

### 4.4.6 Fault Tolerance

Fault tolerance addresses recovery capabilities when the experiment does not execute as expected. OMF and NITOS frameworks are designed to be fault-tolerant. In OMF, there is a Management Plane responsible for rebooting a resource and setting up particular disk

images on PC-based resources. NITOS uses an automatic monitoring tool to diagnose a malfunction in one of the resources. BOWL is not fully fault-tolerant as it also serves as a production WiFi network for actual users. COINS is fault-tolerant due to its architecture design having separated infrastructure and target node as well as separate development and application interfaces. The development interface provides low-level access to the target node. Therefore, when the application on the target node crashes and the communication over the application interference is lost, it is trivial to remotely reprogram the target node through the infrastructure node using the development interface, thus avoiding the need for physical intervention.

### 4.4.7 Debugging

The debugging process addresses the ability to identify problems and their causes during the experiment execution. All compared frameworks are designed to produce extensive log files during the build process. COINS also collects extensive debug log during the execution. Furthermore the debug log is automatically emailed to the developer in case the execution fails.

### 4.4.8 Monitoring

Monitoring is a feature that enables the developer or a maintainer to oversee the status of the system at any time. All compared frameworks have a system for monitoring. COINS uses the Ansible ping method, which works on top of SSH to identify which nodes are accessible. Furthermore, it integrates a standalone open-source monitoring tool Munin as part of the management system.

### 4.4.9 Data Management

Data management is responsible for the distribution and collection of data. OMF uses the Measurement Collection system based on a relational database. NITOS and BOWL do not prescribe any dedicated data collection mechanism on the framework level. COINS uses the same system for building data management as for debugging; i.e., each build produces the result files along with the debug files in the build repository.

### 4.4.10 Source Repository

The COINS framework is designed to keep everything needed for a completely automated build inside a single version-controlled source repository. This is in line with one of the fundamental CI concepts.

### 4.4.11 Automated Build

The automated build is another core CI concept. It ensures that a new code committed by the developer does not break the process, which is particularly important in CI. We showed that COINS defines the automated build system, which deploys the modified source code to the appropriate targets where it is built and tested.

### 4.4.12 Self Testing

Tests of each committed change are extremely important for the CI process. In COINS, all the tests are included inside the repository. The test process is executed automatically on each commit, and the result is communicated to the developer.

### 4.4.13 Fast Build

The last core CI concept requires a fast build process so that the developer does not have to wait long for the test results. To achieve fast builds, COINS relies on the container technology and incremental changes delivery to the testbed device. Therefore, only the changes are rebuilt within the container, thus making the build process fast.

## 4.5 Summary

In this chapter, we described in detail the implementation of the proposed reference architecture as well as the definition of a new efficient lightweight protocol for communication with embedded hardware, LCSP. Finally, we performed a qualitative comparison of the proposed framework with three related existing frameworks. The comparison considers a list of features already used in the community for comparing experiment management tools. However, to fully illustrate the advantage of the proposed framework, we extended the available list of features with four CI-related ones.

# Chapter 5

# Analysis and Modeling of Developers' Activity

In order to introduce fully automated CI practices in the development of wireless embedded systems with restricted capabilities, an in-depth investigation and understanding of wireless developers' activity is required which in turn needs to be reflected in a suitable dimensioning procedure of wireless testing resources and scheduling of code tests. In this investigation we represent the wireless test infrastructure as a queue-based system with physical capacity limitations, requiring carefully constructed models based on developer activity to properly dimension typically expensive tests infrastructure.

This chapter provides contribution C1 in Section 5.5, based on [82].

## 5.1 Modeling Software Development Testing Process

The use case of testing infrastructure consisting of one or more test environments used for wireless firmware integration testing during the development phase of embedded systems as described in Chapter 4 can be represented as a multi-server queuing system depicted in Figure 5.1. In such a system, requests for integration tests, represented as arriving jobs, are inserted in a single waiting queue until they are executed in one of the test environments, represented as servers. The test environment is defined as an independent setup able to execute integration tests isolated from others either in space or frequency.

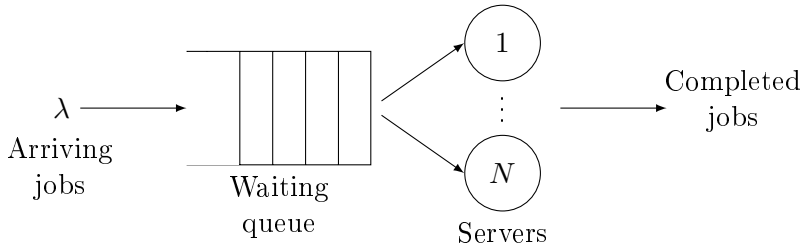



Figure 5.1: The multi-server queuing system.

The described system behaves according to a memoryless stochastic process and can be modeled as a Markov chain depicted as a state transition diagram in Figure 5.2 with transition probabilities associated with various state changes [83]–[86]. This is known as the Erlang delay system which can be described by Kendall's notation $M/M/N$ [87], [88]. The $M$ stands for Markovian random variable. The first $M$ represents arriving jobs according to a Poisson process with a mean rate $\lambda$, moving the process from state $i$ to $i+1$. The

second $M$ represents job service times with exponential distribution and a mean service rate $\mu$, moving the process back from state $i+1$ to $i$. $N$ is the number of servers able to simultaneously execute jobs, so the system is able to handle jobs at the rate $N\mu$ [86].

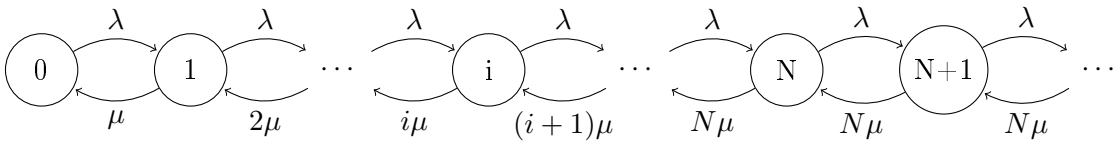



Figure 5.2: State transition diagram of the $M/M/N$ system having N servers and an infinite waiting queue.

The system is able to handle new jobs without waiting when $i \leq N$. If there are more jobs $i$ in the system than the available servers $N$, all the servers become occupied and the next arriving job is placed in a queue for a waiting time represented a by random variable $\mathcal{W}$. During the execution a job occupies one server for an average time $h$, which can be expressed as an inverse of the mean service rate:

$$h = \frac{1}{\mu}\,. \tag{5.1}$$

The $M/M/N$ system has an analytically traceable solution derived from the Markov chain theory while considering the normalization condition (sum of all probabilities must be equal to 1) of the state transition probabilities in equilibrium [83]–[85]. The solution, known as the Erlang-C formula [87], [89]–[91], has been successfully applied to various real-life queueing problems [49], [92]. The Erlang-C formula denoted by $P_C(A, N)$ gives the steady-state probability of a job being put in a waiting queue when all the servers are occupied and is defined as

$$P(\mathcal{W} > 0) = P_C(A, N) = \frac{\frac{A^N}{N!}\frac{N}{N-A}}{\sum_{i=0}^{N-1}\frac{A^i}{i!} + \frac{A^N}{N!}\frac{N}{N-A}}\,, \quad A < N\,, \tag{5.2}$$

where $N$ is the number of available servers and $A$ is the traffic load on those servers defined as the ratio of the mean arrival rate and mean service rate:

$$A = \frac{\lambda}{\mu}\,. \tag{5.3}$$

The mapping between the terminology generally used in queuing theory and the testing use case is summarized in Table 5.1.

In the rest of the thesis we will use queueing theory terminology applied to the use case of testing infrastructure for wireless firmware integration.

### 5.1.1 Testing Queue Wait Time

If there are more than N tests in the system, each new test is put in the queue until it can be executed by an idle server. The average queue waiting time $W$ is the time that test requests spend in the queue waiting for a free test environment. The waiting time in the queue is defined by Little's theorem [87], [90], [93] as

$$W = \frac{L}{\lambda}\,, \tag{5.4}$$

Table 5.1: Mapping of queueing theory terminology to the testing use case.

| Parameter | General | Applied to testing |
|---|---|---|
| $A$ | Busy hour traffic | Busy development hour traffic |
| $N$ | No. of servers | No. of test environments |
| $\lambda$ | Job arrival rate | Developer contribution rate |
| $h$ | Average job service time | Average test execution time |
| $W$ | Average job queue time | Average test queue time |

where $L$ is the number of tests in the waiting queue.

For the $M/M/N$ system, $L$ is defined as

$$L = P_C(A, N) \cdot \frac{A}{N - A}\,. \tag{5.5}$$

Now we can combine equations (5.1), (5.3), (5.4) and (5.5) to derive the average queue waiting time $W$ for the $M/M/N$ system as

$$W = P_C(A, N) \cdot \frac{h}{N - A}\,. \tag{5.6}$$

Using this equation we performed simulator verification by comparing the analytical and simulation results [90].

### 5.1.2 Waiting Time Distribution

For the provision of certain service level guarantees the average queue waiting time is not enough, so we need to determine also the distribution of waiting time for $M/M/N$ delay system [83], [94], [95] which has an exponential distribution if the waiting queue is infinite. $P(\mathcal{W} > t)$ is the probability that a job has waiting time $\mathcal{W}$ greater than $t$ and is defined as

$$P(\mathcal{W} > t) = P_C(A, N) \cdot \mathrm{e}^{-(N-A)\mu t}\,, \tag{5.7}$$

where $P(\mathcal{W} > t)$ is the probability distribution function of a random variable $\mathcal{W}$.

Complementary probability distribution function of a random variable $\mathcal{W}$ has the following property:

$$P(\mathcal{W} \leq t) = 1 - P(\mathcal{W} > t)\,, \tag{5.8}$$

where $P(\mathcal{W} \leq t)$ denotes the probability that the random variable $\mathcal{W}$ takes on a value that is less than or equal to $t$ [85]. Therefore, for a known acceptable waiting time $w$ we can define the probability of the grade of service $P_{GoS}$ for the Erlang delay system as

$$P_{GoS} = 1 - P_C(A, N) \cdot \mathrm{e}^{-\frac{w \cdot (N-A)}{h}}\,. \tag{5.9}$$

This formula gives the probability for a test waiting in the queue for less than the acceptable waiting time $w$. This model fits exceptionally well to the testing infrastructure capacity problem and is further used in Chapter 6 for the performance evaluation while the implementation of the presented formulas can be found in Appendix B.

### 5.1.3 Busy Development Hour Traffic

Busy development hour traffic can be expressed from Eq. (5.1) and Eq. (5.3) as

$$A = \lambda h\,, \tag{5.10}$$

where $\lambda$ represents the arrival rate of developers' code contributions during the busy development hour and $h$ the average duration of test executions. Each code contribution issues an integration test request, therefore it is expected to occupy one test environment.

The rate of developers' code contributions $\lambda$ can be extracted for instance from the GitHub metadata collection. We approached the problem by deriving the hourly commit distribution from data to detect the percentage of contributions issued during the busy hour as explained in Section 5.2. In the scope of this work the commits are triggers for software tests and thus also a proof of activity of each developer. For the dimensioning of testing environment we therefore wanted to include the number of developers in the calculation of $\lambda$ and defined it as

$$\lambda = p \cdot r \cdot k\,, \tag{5.11}$$

where $p$ is a percentage of a maximum number of developer's contributions occurring during the busy development hour, $r$ is the number of all considered daily contributions per developer and $k$ is the number of developers in the team. The intent is to provide a model where the only unknown is the number of developers which is different for every company or a development team.

As an example, to illustrate the calculation of code contribution rate in a busy hour, let us assume a team of 20 developers ($k = 20$), where each of them is expected to produce up to 5 commits in a day ($r = 5$). If 10% of all commits happen during the busy hour ($p = 0.1$), then the code contribution rate in a busy hour yields 10:

$$\lambda = 0.1 \times 5 \times 20 = 10\,. \tag{5.12}$$

## 5.2 Hourly Developers' Activity Distribution

Busy hour job arrival rate is typically statistically measured i.e. from a sample event log, however in the case of software development it is possible to analyze the developers' activity from other sources of data such as discussed in Section 2.4. In the case of this work the developers' code commits and repository updates in the form of so-called "pushes" are directly used as an indication of developers' contribution. Developers' activity can be measured in various ways as discussed in [96], however, software tests are typically triggered on commits and pushes.

In this thesis, we went beyond the mining of software repositories, which nowadays is a common practice as explained in Section 2.4. The intent is to use the knowledge obtained with mining as an input in the simulation to eventually dimension the optimal testing environment capacity.

For understanding the developers' activity, and particularly the percentage of contributions $p$ during the busy hour required for Eq. (5.11), we used well-known knowledge discovery methods [97] on development logs from different open source projects. Knowledge about developers' activity, however, can also be extracted by mining extensive software repository metadata collected from social coding platforms such as GitHub. We collected and analyzed the commit timestamps to obtain the frequency distribution of commits on an hourly basis. We then determined an appropriate fitting function that approximates the frequency distribution of commits. We built the frequency distribution model using

the fitting method based on Levenberg-Marquardt algorithm implemented in "curve_fit" function of the SciPy[41] library.

The metadata in git logs for popular open source projects is a high quality source of information since each contribution has to be approved by the project maintainer. For our analysis we used local timestamps in git logs, thus the obtained distribution is independent of the developer's timezone. All timestamps inside project logs are valid, therefore the considered datasets[42] do not require any cleaning. In our analysis we considered the following mainstream open source projects:

1. Linux kernel; all 768265 commit timestamps were used. Linux kernel is developed using their own git server infrastructure.

2. Firefox; all 429275 commit timestamps were used. Firefox is developed using their own Mercurial server infrastructure.

3. Docker; all 35448 commit timestamps were used. Docker is developed on GitHub.

4. Tensorflow; all 36334 commit timestamps were used. Tensorflow is developed on GitHub.

Linux and Firefox are representatives of mature open source community projects developed over two decades. Docker and Tensorflow, on the other hand, were open-sourced after initial development by commercial companies to engage a larger community of developers.

In addition to these mainstream projects we also considered three open source projects from the wireless domain, in particular the popular GNU radio project, the 5G OpenAirInterface (OAI) project and our in-house project for developing drivers for embedded wireless devices called VESNA:

1. GNU radio; all 11429 commit timestamps were used. GNU radio is developed on GitHub.

2. OpenAirInterface; all 6506 commit timestamps were used. Openairinterface is developed on GitLab.

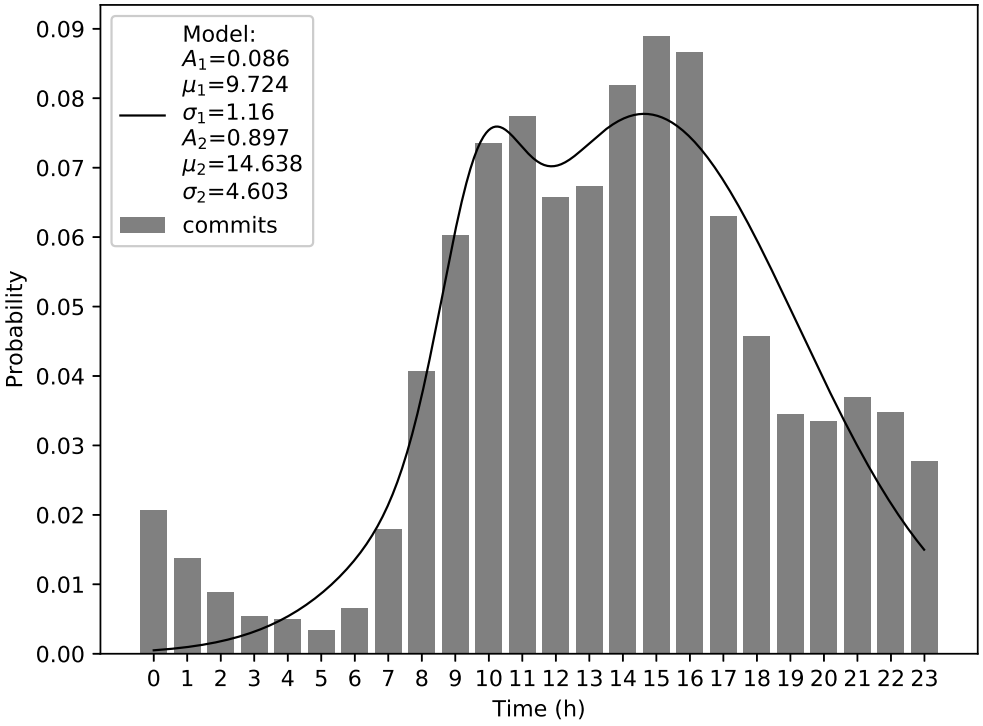
3. VESNA drivers project; all 2236 commit timestamps were used. Vesna drivers project is developed on GitHub.

The hourly commit activity frequency distributions for the mainstream projects are depicted in Figure 5.3. The combined count of commits is around 1.3 million. All four charts clearly exhibit two peaks, slightly differing in amplitude and time. In all cases one appears in the morning and the other in the afternoon, and they are separated by a region with decreased activity. The two peaks can be modeled by fitting a bimodal distribution function. Bimodal distribution function is a sum of two normal distributions which form a two-peak shape and is defined as

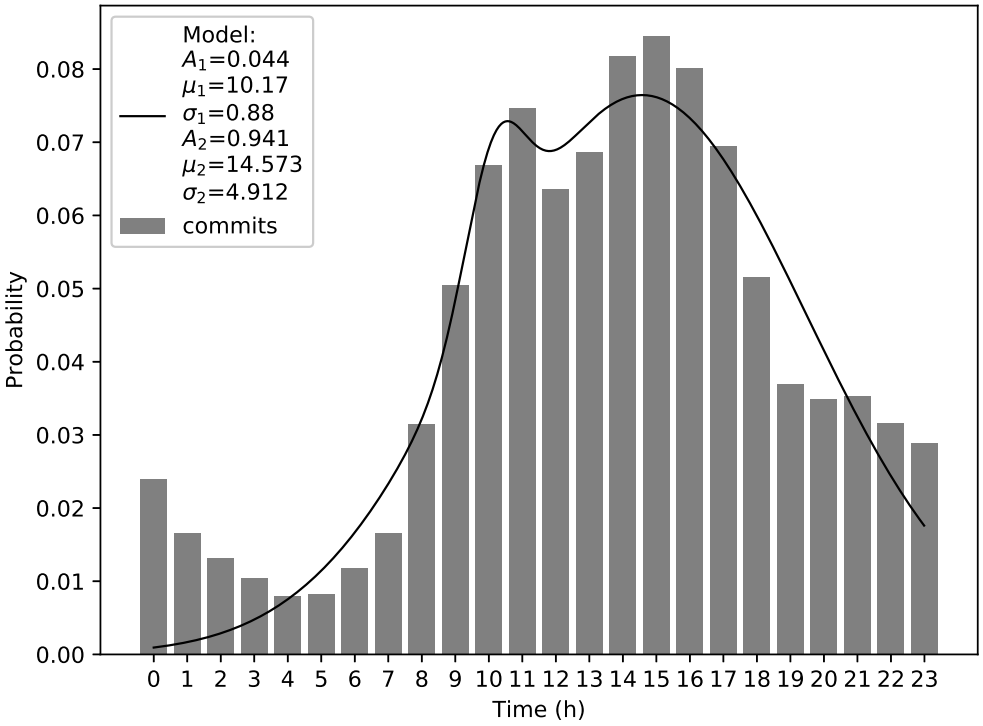
$$\mathcal{X} \sim A_1\mathcal{N}(\mu_1, \sigma_1) + A_2\mathcal{N}(\mu_2, \sigma_2)\,. \tag{5.13}$$

[41] SciPy, `https://docs.scipy.org/`

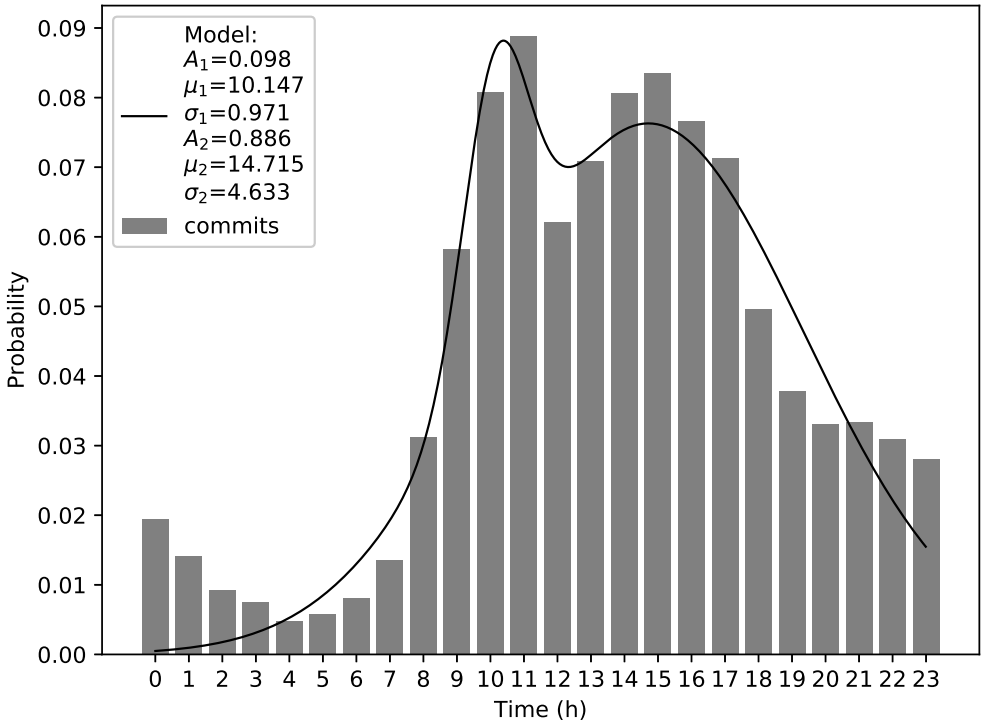
[42] The datasets were collected between Aug. 2018 and Jan. 2019.



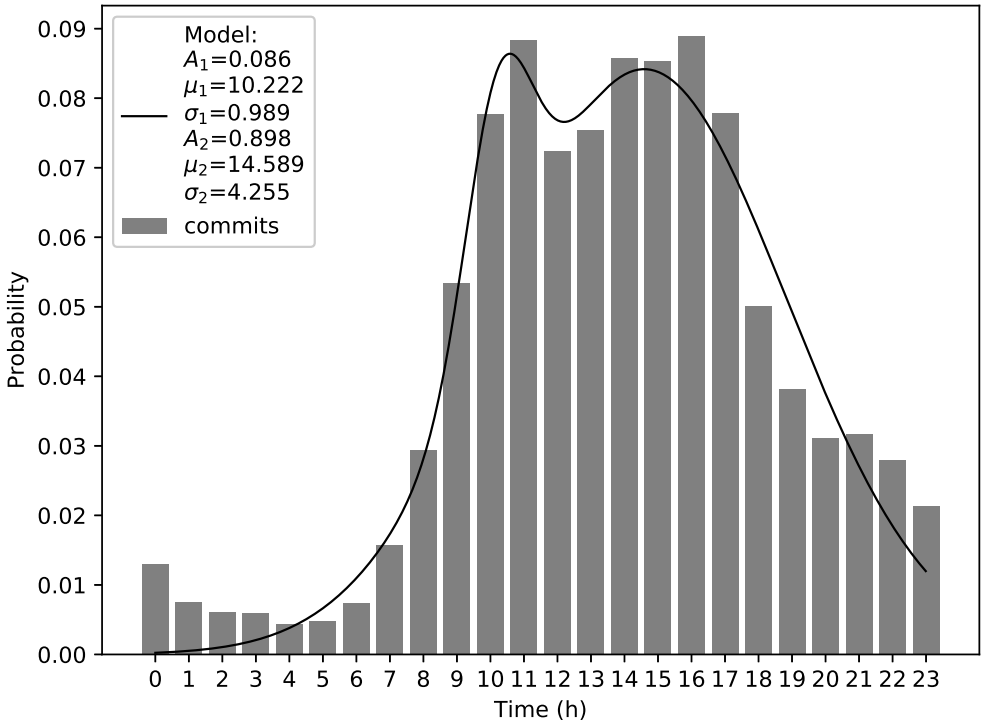

(a) Linux 768265 commits.




(b) Firefox 429275 commits.




(c) Docker 35448 commits.




(d) Tensorflow 36334 commits.

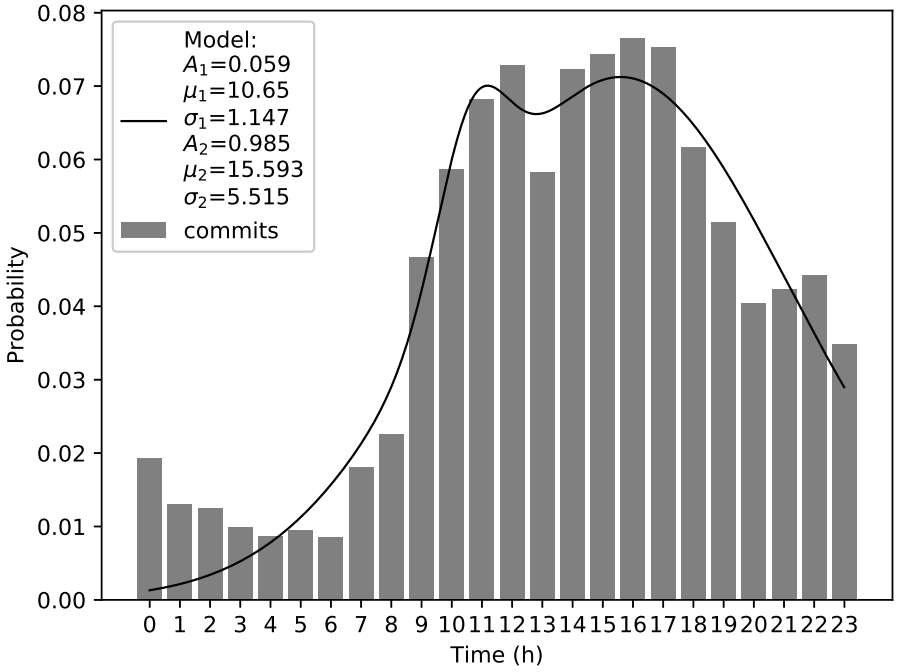
Figure 5.3: Mainstream projects hourly distributions of code contributions.

Considering the probability distributions from Figure 5.3, three observations can be made. First, different projects behave slightly different in terms of amplitude and times of the commit peaks. The morning peak in all cases appears at around 11 a.m. The afternoon commit peaks, however, appear at 3 p.m. in Figures 5.3 a, b, c and at 4 p.m. in Figure 5.3 d. Second, community-driven projects in Figures 5.3 a, b have higher commit activity in the afternoon whereas corporate-driven projects in Figures 5.3 c, d have a higher of very similar commit activity in the morning. The third and the most important observation for the calculation of busy hour contribution rate is that in all cases the peak hours contain less than 10% of all daily contributions. The percentages of contributions for the first and the second peak hour and their occurrence in time for the mainstream projects are summarized in Table 5.2.

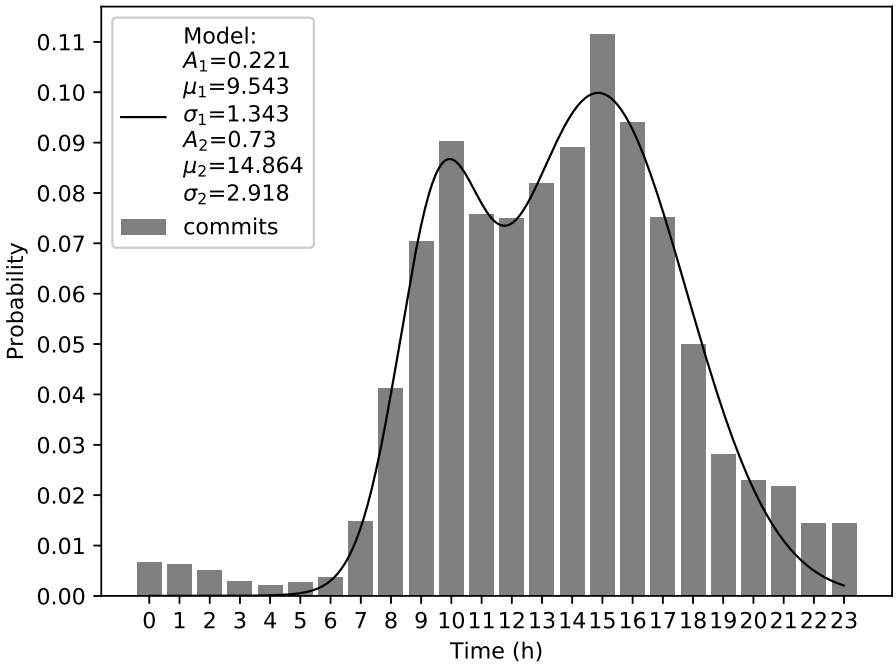
Table 5.2: Percentage of contributions $p$ [%] in peak hour and time T [h] of its occurrence for different mainstream projects.

| Project | $1^{st}$ p [%] | $1^{st}$ T [h] | $2^{nd}$ p [%] | $2^{nd}$ T [h] |
|---|---|---|---|---|
| Linux | 7.7 | 11 | 8.9 | 15 |
| Firefox | 7.5 | 11 | 8.5 | 15 |
| Docker | 8.9 | 11 | 8.4 | 15 |
| Tensorflow | 8.8 | 11 | 8.9 | 16 |

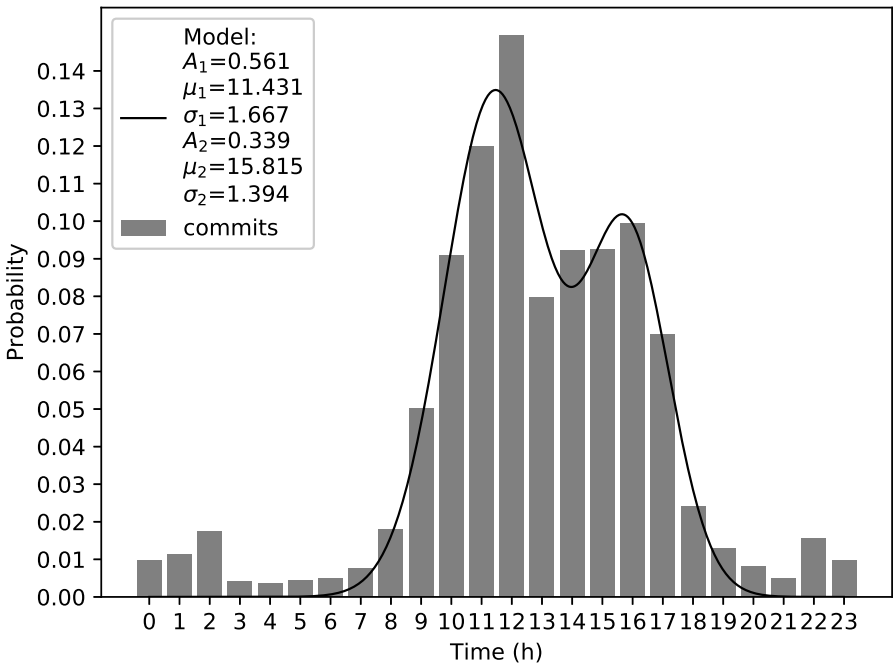
Figure 5.4 depicts hourly commit distributions for projects specific to wireless systems development. It is evident that wireless projects contribution activity distributions




(a) GNU radio commit distribution based on 11429 commits.




(b) OpenAirInterface commit distribution based on 6506 commits.




(c) VESNA drivers commit distribution based on 2236 commits.

Figure 5.4: Wireless projects' hourly commit distributions.

also broadly conforms to the distributions of the mainstream projects. Therefore, we can conclude that the observation from the mainstream projects applies to smaller projects. Furthermore, the pattern can also be used in a wireless domain. However, from the peaks observed in Figure 5.4 and summarized in Table 5.3 we can conclude that peak hours as well as the percentage of hourly contributions exhibit larger variations in smaller wireless projects than in the mainstream projects. Nonetheless, the dataset for these projects is much smaller combining only approximately 20 thousands commits. A very similar two peak distribution of human activity was also found in [98] while observing a completely different source of data. Their work was based on call center logs which they used as a basis for a comparison between modeling based on simulations and analytical approach with queuing theory.

Table 5.3: Percentage of contributions $p$ [%] in peak hour and time T [h] of its occurence for different wireless projects.

| Project | $1^{st}$ p [%] | $1^{st}$ T [h] | $2^{nd}$ p [%] | $2^{nd}$ T [h] |
|---|---|---|---|---|
| GNU radio | 7.3 | 12 | 7.7 | 16 |
| OpenAirInterface | 9.0 | 10 | 11.1 | 15 |
| VESNA drivers | 14.9 | 12 | 9.9 | 16 |

## 5.3 Team Size Analysis

With respect to testing resource parallelization requirements for the introduction of CI practices it is also important to take into account the typical sizes of developer teams. Using the GHTorrent dataset hosted on the Google BigQuery platform we extracted from the metadata the number of contributors per each project using the SQL query given in Listing 5.1.

Listing 5.1: SQL query used to download data on the number of contributors per project from the Google BigQuery database.

```
SELECT repo_id, COUNT (*) AS members
FROM [ghtorrent-bq.ght_2018_04_01.project_members]
GROUP BY repo_id ORDER BY members ASC;
```

This query returned 6,654,055 projects, but a vast majority of them have less than 10 contributors. In Figure 5.5, we depict the distribution of group sizes on a logarithmic scale, showing an exponential drop in the number of contributors per project and indicating that projects with 50 or more developers are extremely rare. Consequently, we decided to consider in subsequent simulations the group sizes of 10, 20, 50 and 100 developers, thus covering the most interesting and representative use cases.

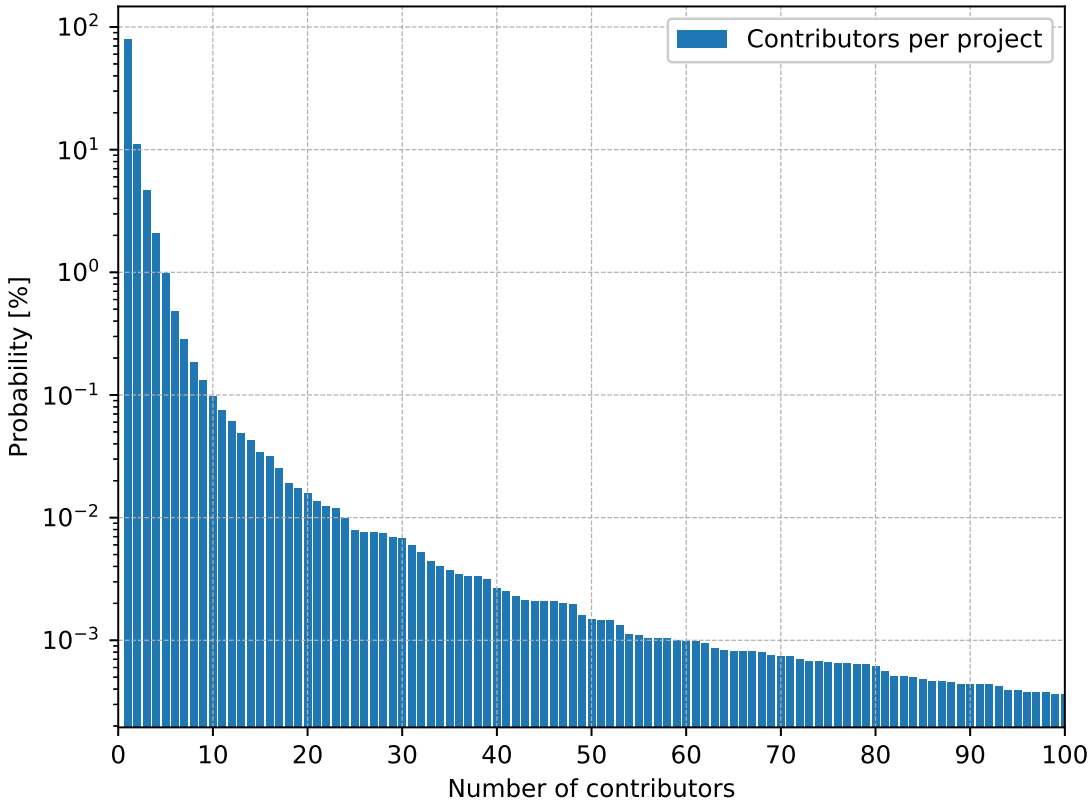



Figure 5.5: Probability of a number of contributors per project.

## 5.4 Utilization of Testing Resources

In addition to buffering the test requests, further dimensioning optimization of the wireless testing environment can be obtained by sufficient isolation of test cases in space and in frequency. Namely, unlike in pure software testing where a single device can support multiple parallel test environments isolated by the virtualization concept, in wireless firmware testing that includes physical communication a single test generally occupies multiple nodes. In a minimal test environment there are one transmitter node ($Tx$), one receiver node ($Rx$) and nodes in their vicinity that are affected by the radio interference ($I$). Mathematically, the number of occupied nodes by a single test can be represented by the following expression:

$$N_o = N_{Tx} + N_{Rx} + N_I \,, \tag{5.14}$$

where $N_o$ is the number of all the nodes which will be occupied by the test with $N_{Tx}$ transmitter nodes, $N_{Rx}$ receiver nodes and $N_I$ further nodes affected by the radio interference, which can be determined by distributed spectrum sensing in the testbed.

In such testing environment, a possible way of optimizing the testbed utilization is to support the execution of parallel tests in non-conflicting frequency-separated radio channels and frequency bands without the need to modify the tests or to depend on the user to provide extra metadata about the test. The LOG-a-TEC testbed that we used as a reference supports testing in three non-conflicting unlicensed frequency bands for industrial, scientific and medical (ISM) applications, i.e. in sub-GHz band at 868 MHz, in 2.4 GHz band and in 5GHz band, each supporting multiple frequency channels depending on the underlying technology. Moreover, the same testbed supports also multiple wideband

channels for ultra-wide band (UWB) technology between 3.5 GHz and 6 GHz.

Further optimization of the wireless testbed utilization can be achieved by space separation between multiple test environments making use of physical distribution of nodes and controlled transmit power, resembling the cellular approach in mobile communication networks.

## 5.5 Probability Distribution of Daily Code Contributions

After having investigated the daily activity (i.e., parameter $p$ in Eq. (5.11)) of a team or a community of developers to derive the peak hourly contribution percentile, we will now focus on an individual developer and in particular on push probability distribution as the maximum number of expected code contributions $r$ (see Eq. (5.11)) produced by a single developer.

In the context of this study, the push probability distribution is needed to determine the probability that a developer will push code contribution $r$ times per day. For the dimensioning of the testing infrastructure we need to find the parameter $r$ such that it covers preselected percentage of all pushes, for instance 95%, i.e. the probability of a number of all pushes per day beyond this limit is less than 5%. Since the shape of the probability distribution function is not known in advance we need to obtain it by analyzing the metadata on developers' activity and plot the push frequency distribution per developer per day.

### 5.5.1 Dataset Cleaning and Conditioning

There are multiple projects collecting public metadata from social coding platforms and making it searchable using query languages such as SQL. The largest collection of developers' metadata is collected from the social coding platform GitHub. The GH Archive dataset hosted on the Google BigQuery platform was created in 2017 and can be queried using SQL query provided in Listing 5.2. The dataset contains 206,210,836 non-empty push events from January 2017 to January 2018.

Listing 5.2: SQL query used to download data on non-empty push events from the Google BigQuery database.

```
SELECT FORMAT_TIMESTAMP('%Y-%m-%d',created_at)
AS date, actor.id
FROM `githubarchive.year.2017` WHERE type = 'PushEvent'
AND JSON_EXTRACT(payload, '$.size') <> '0'
```

GitHub is an open platform, so there is very little limitation of what can or cannot be uploaded. Therefore the dataset required cleaning, since not all push events are relevant for the analysis. The first data cleaning step was concerned with users that produce an unreasonable number of commits. These users are actually not human but rather different kinds of robots. One of the bigger numbers found in the dataset was a user producing close to 10,000 non-empty pushes in a single day. We confirmed that it actually was a robot, which was stated in the user description on GitHub. Therefore the threshold was set to the maximum of 50 pushes per day per developer and everything beyond that was ignored.

Another cleaning step aimed to remove the developers working during weekends and holidays since they behave differently than the developers working on workdays. In particular, there are fewer developers working on weekends and holidays, but on average they produce more pushes per day than those working on workdays only. The analysis was based only on workdays which are more critical for testbed dimensioning. Therefore, the

weekends and major US holidays will be excluded since previous studies indicated that the majority of GitHub events originate from US [58].

After cleaning the dataset as described above, there were still 42,080,406 non-empty push events left. The next step was to ensure that data is normally distributed, therefore P-value normality test was done for the first four most significant points, i.e. for one, two, three, and four pushes per day. If the P-value is more than 0.05, the dataset can be considered normally distributed. Figure 5.6 depicts the graphical representation of data as a Cumulative Distribution Function (CDF) as well as calculated P-values for four most significant push events.

The blue lines in Figure 5.6 represent the theoretical CDF of a normal distribution and the orange dots are the sum (on x-axis) of unique developers that made at least one push in that particular day. If they made exactly one push they get summed together and depicted as an orange dot in the top left corner of Figure 5.6, if they made two they end up in the top right corner, if they made three in the bottom left and four in the bottom right corner. This is repeated for every workday for the entire year where one dot represents the sum of unique developers for that day. The closer the orange dots are to the theoretical CDF, the closer the dataset is to normal distribution which is quantified by the P-value.

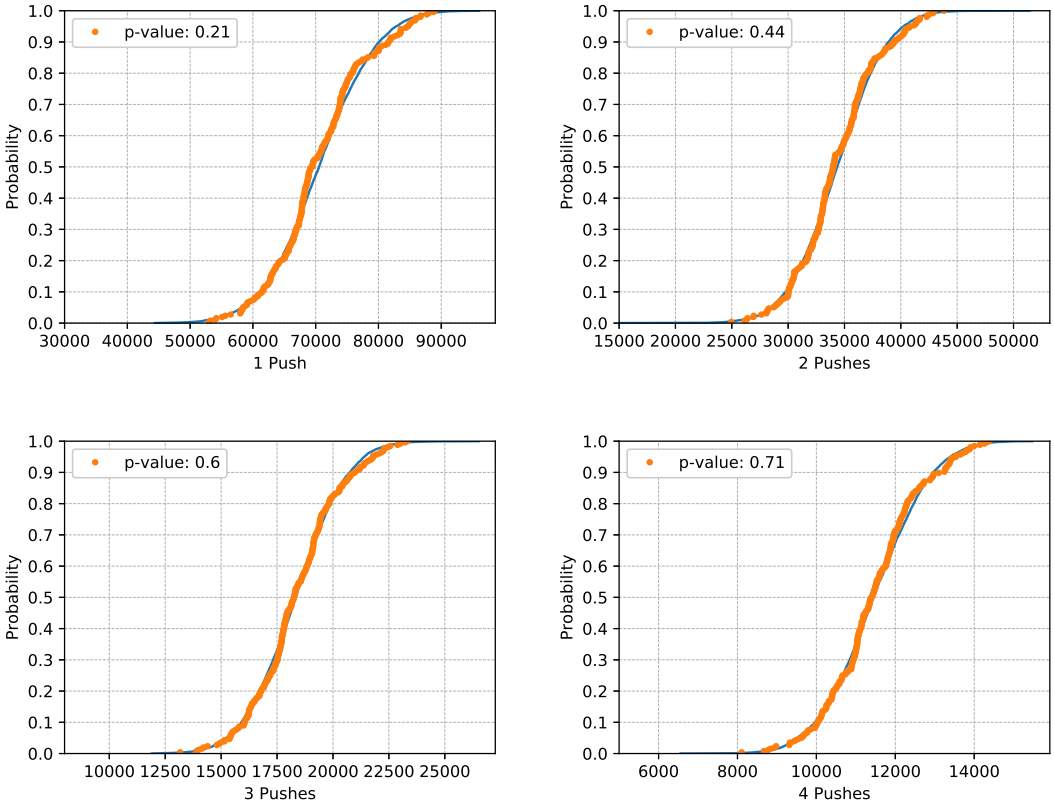



Figure 5.6: P-value normality test for the GitHub 2017 dataset after cleaning.

For a reference we also provide, depicted in Figure 5.7, the P-value normality test performed on data without any cleaning steps performed. It is evident based on the calculated P-values that without any cleaning this dataset cannot be considered normally distributed.

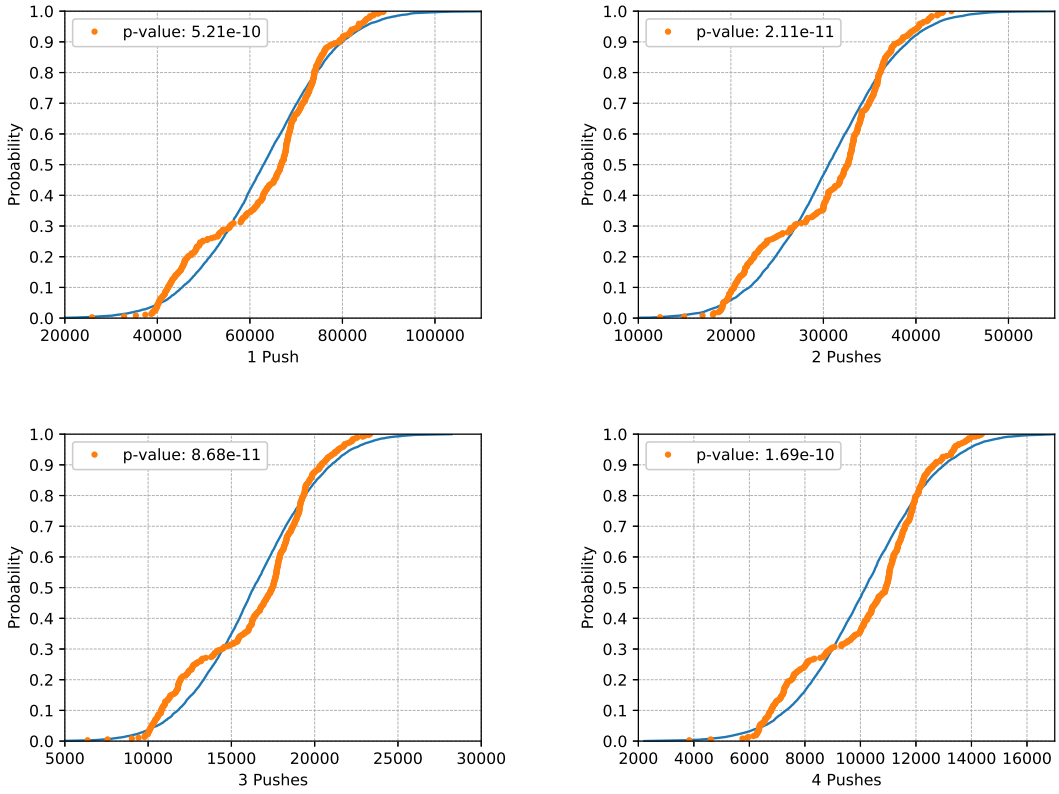



Figure 5.7: P-value normality test for the GitHub 2017 dataset before cleaning.

### 5.5.2 Model of Daily Developer Contributions

After the described cleaning steps it is possible to plot the data as push frequency distribution per developer per day averaged over a year with 95% confidence interval for each point as depicted in Figure 5.8. It can be observed that the number of pushes per developer per day drops exponentially, so it is possible to fit an exponential function to this data that represents a probability distribution model of contributions per developer per day. The exponential fitting function is defined as

$$f(x) = ae^{-bx} + c\,, \tag{5.15}$$

where the parameters of the fitted exponential function are a=131,334, b=0.643 and c=375. To obtain the resulting parameters, we again used the fitting method based on the Levenberg-Marquardt algorithm as described in Section 5.2.

The curve in Figure 5.8 depicts a fitting of an exponential function to data on the frequency of developer's pushes, indicating a very good match between the data points and the mathematical model.

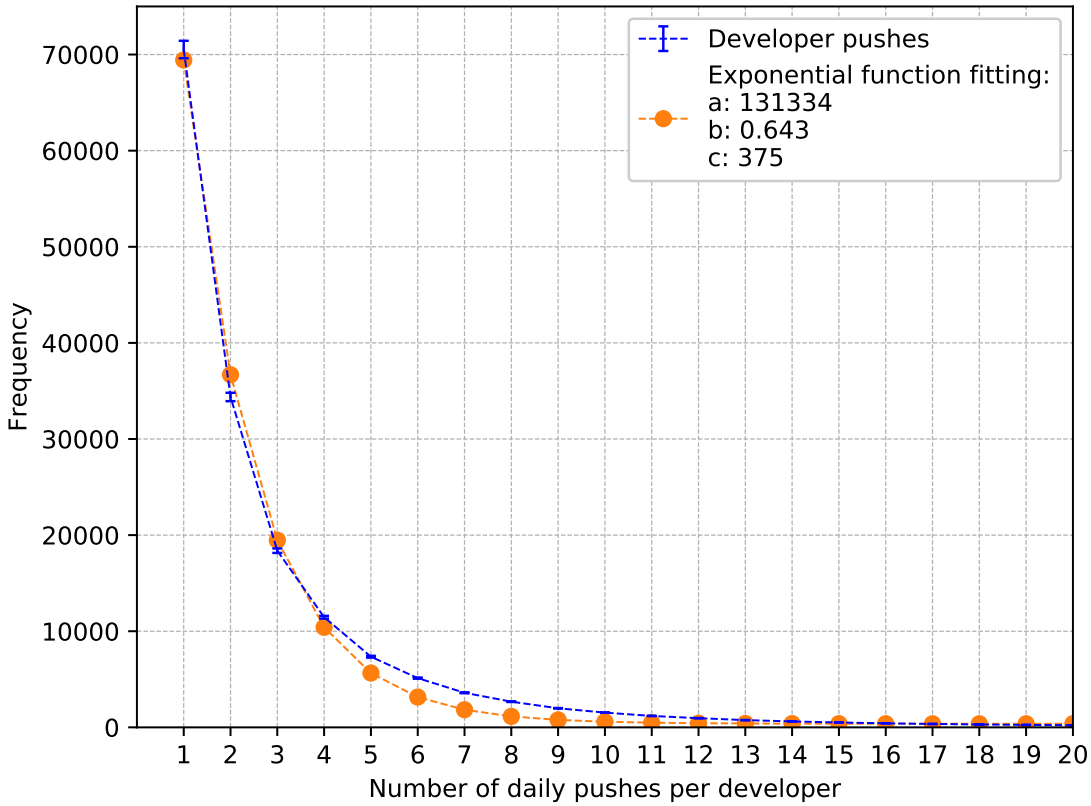



Figure 5.8: Daily push frequency distribution per developer for the GitHub 2017 dataset.

Next, we defined the developer contribution model as the probability density function (PDF) of the number of daily contributions per developer by shifting and normalizing the obtained fitting function so that the cumulative probability for the number of pushes on the interval $[1, \infty]$ equals 1. It can be written as

$$f(x) = \begin{cases} \lambda \mathrm{e}^{-\lambda(x-\rho)}, & x \geqslant \rho, \\ 0, & x < \rho, \end{cases} \tag{5.16}$$

where $\lambda = 0.64$ and $\rho = 1$. These parameters were obtained by fitting the exponential function defined by Eq. (5.15) to the GitHub dataset.

By integrating the PDF, we can define the CDF as

$$F(x) = \begin{cases} 1 - \mathrm{e}^{-\lambda(x-\rho)}, & x \geqslant \rho, \\ 0, & x < \rho, \end{cases} \tag{5.17}$$

where $\lambda = 0.64$ and $\rho = 1$ same as for PDF. Figure 5.10 depicts the CDF and can be interpreted as the probability of the developer making $x$ or less contributions per day. We can search for the number of daily contributions per developer where the probability achieves the preselected percentage of all pushes, e.g. if we chose to cover 90% of development situations we need to consider up to 5 developers' contributions per day, for 95% up to 6 contributions per day and for 99% up to 9 contributions per day.

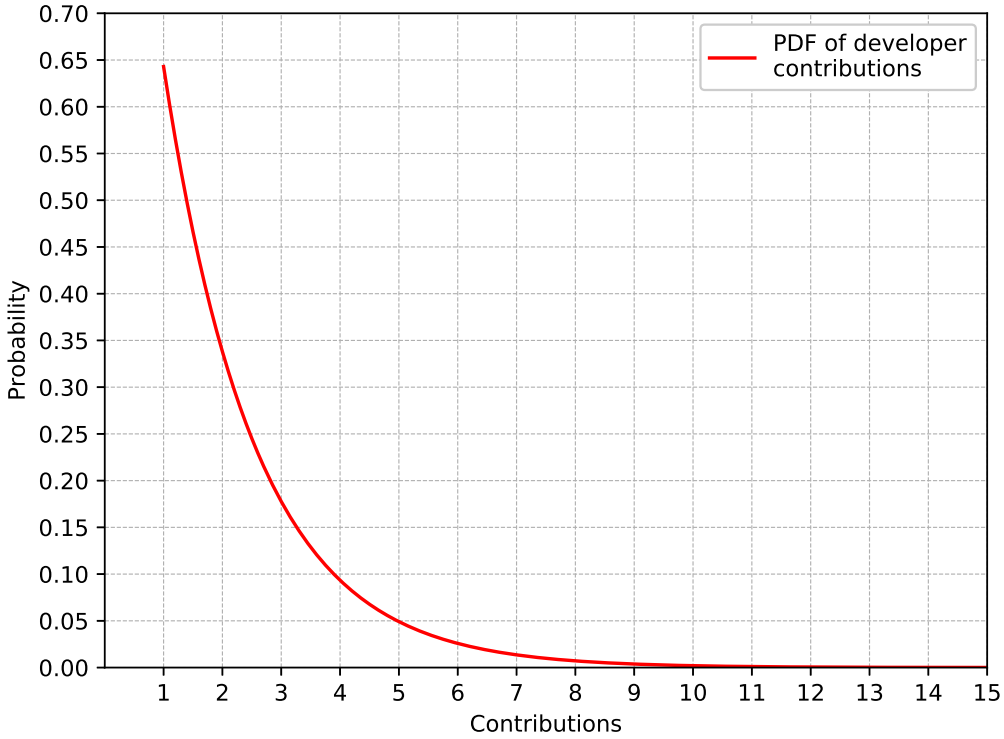



Figure 5.9: PDF of daily developer contributions.

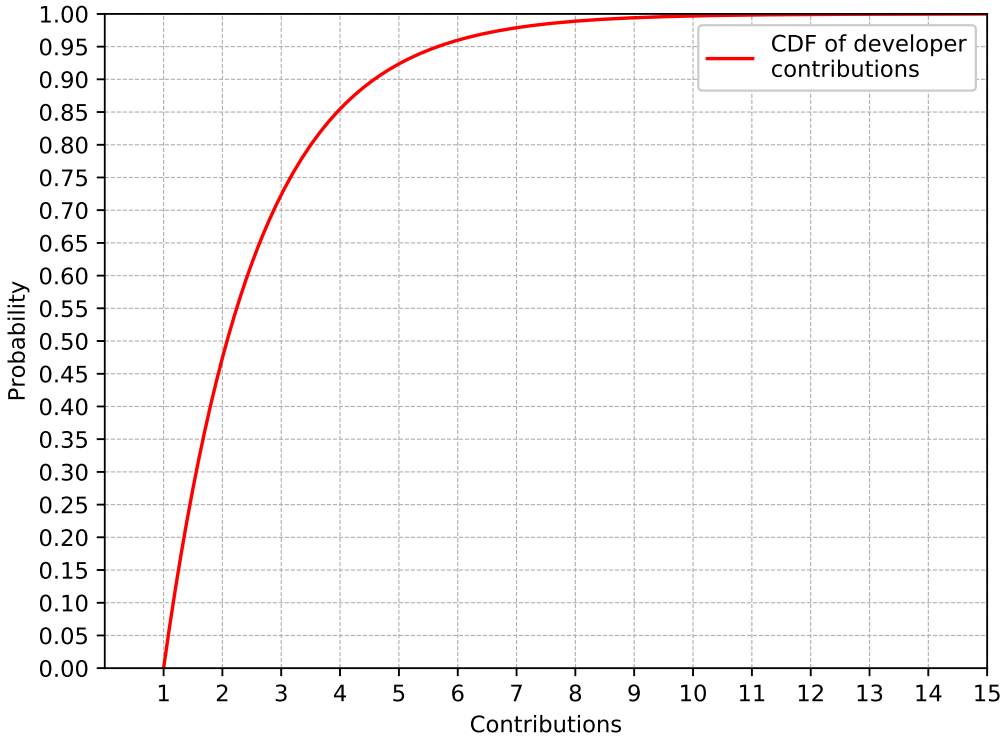



Figure 5.10: CDF of daily developer contributions.

## 5.6 Determination of Test Service Time

While analytically tractable Erlang-C based $M/M/N$ model assumes exponential distribution of test service times, the duration of tests in real systems does not obey such nice mathematical representations and depends on a number of steps. In particular, the process of firmware testing in a wireless distributed testbed includes automatic instantiation and preparation of builds of newly contributed code with other preexisting code, configuration files and dependencies, transfer of built software components to targeted nodes in the testbed, test running and collection of results. In order to determine the overall test execution times for wireless firmware contributions needed for the subsequent simulations in Chapter 6 we analysed the data obtained in our testbed for more than 200 CI tests of VESNA drivers project over the course of one year. The collected testbed build execution times are depicted in Figure 5.11 and summarized in terms of the average successful and failed test execution times in Table 5.4.

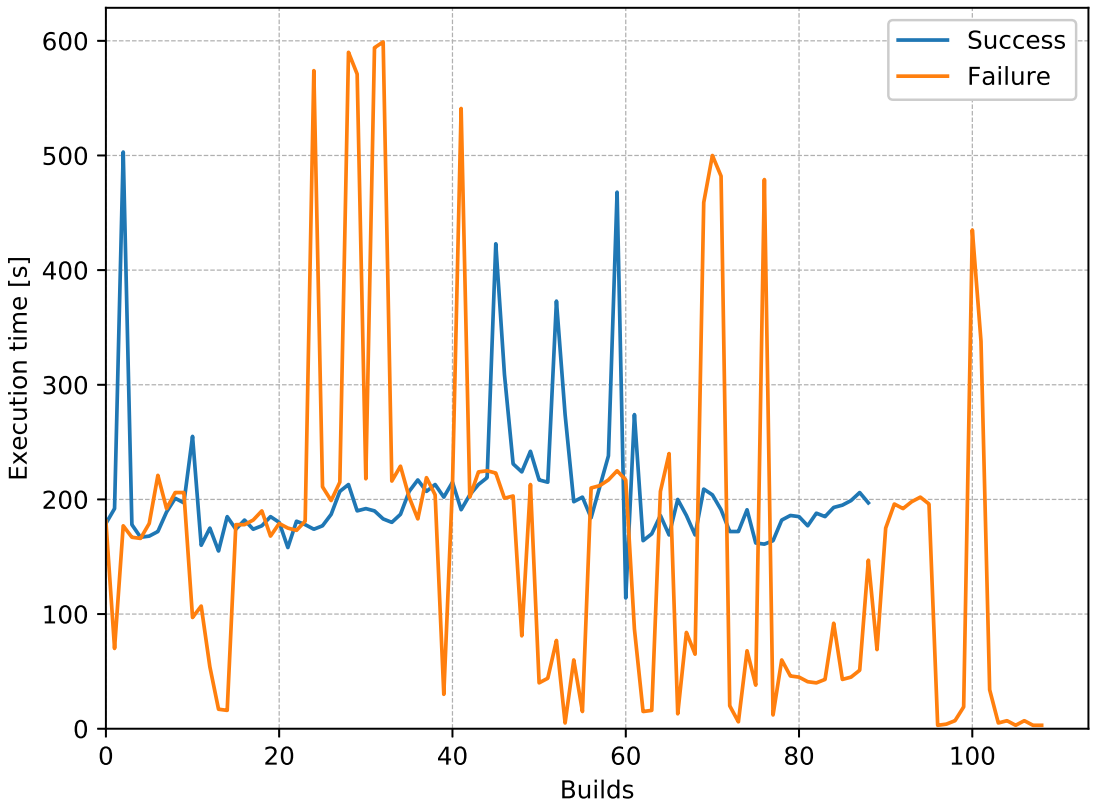



Figure 5.11: Testbed build execution times for successful and failed tests.

Table 5.4: Testbed build execution statistics.

| | Success | Failure |
|---|---|---|
| No. of tests | 89 | 109 |
| Avg. execution time [s] | $204 \pm 12$ | $164 \pm 28$ |

The average failed test execution time is shorter because in some cases, a test can fail immediately if, for example, there is an error in test delivery configuration. The interesting result is the average successful test execution time which is around three and a half minutes. The spikes in test execution times correspond to situations where some dependencies were missing, hence an installation procedure had to be invoked.

Based on the insight from the empirically obtained test execution times we can define a statistical model for generating test execution times in the simulation-based optimization of testbed capacity. In particular, we assumed a model generating random test execution times of duration distributed uniformly between two and five minutes, thus leaving an opportunity for optimization, given that the maximum acceptable waiting time of a developer to receive the test results in the CI-supported system is assumed to be 10 minutes. This assumption means that if separate tests last more than 10 minutes, it is not reasonable to use a CI approach.

## 5.7 Summary

In this chapter, we introduced a new concept of developer activity modeling based on queuing theory. We demonstrated that the problem of wireless embedded firmware testing fits well in a queueing theory model. Next, we presented a deep insight in developer activity and concluded that new models are required. A model of daily developer activity was extracted using data mining approach on large open source project repositories as well as on databases holding metadata of developer contributions collected from the GitHub platform. Additionally, we provided the execution time measurements of tests in the wireless testbed LOG-a-TEC which indicate that real CI tests execution duration is distributed uniformly between two and five minutes.

# Chapter 6

# Performance Evaluation and Testbed Dimensioning

In this chapter, performance evaluation is carried out using computer simulation which is based on a set of predefined assumptions. Where possible, simulations are complemented by numerical computations. The simulation assumptions are all derived either from real measurements or data mining of developer metadata to reflect the real situation as close as possible. First, we describe the design of the simulator which is able to model $M/M/N$ as well as $M/G/N$ queueing systems representing testing infrastructure integrated in a CI process for testing developer code contributions. Next, we describe the numerical algorithm which is used for determining the number of required test environments for the $M/M/N$ queueing system. Then, we present some preliminary results to narrow down the interesting use cases for running the simulations. Finally, we present a detailed analysis and discussion of simulation results for the testbed dimensioning focusing on the test execution times and optimization opportunities.

This chapter provides contributions C3 in Section 6.3 and C5 in Section 6.1.1, respectively, based on [82].

## 6.1 Tools for Testing Process Analysis

Queueing theory was proven to be very difficult to approach analytically and many problems have a closed form solution only under very specific assumptions. Nowadays queuing problems are typically investigated using computer simulations, which provide the approximate accuracy to mathematical solutions and simplify queueing problem analysis also for general distributions that are analytically unsolvable [99], [100]. With computer simulations it is thus no longer necessary to make unrealistic assumptions for the sake of a simple analytical solution [89].

When considering the $M/M/N$ type of queue, there are practical analytical tools available such as described in Section 5.1.1. However, they are often complemented by simulations as well as approached numerically. In our use case we are aware that the testing duration is not following the exponential but rather some other general distribution. Therefore, even if code contributions arrive according to the Poisson process, our problem actually corresponds to the $M/G/N$ queuing model that can only be efficiently solved by computer simulations.

### 6.1.1 Testing Process Simulator

We designed a discrete event simulator specifically to reflect the real situation in deploying tests on a physical wireless testbed. The simulator design is based on running several parallel processes, thus simulating the occurring events inserted in a queue and eventually scheduled to occupy a free resource. The simulation is performed "as fast as possible", however the termination parameter of the simulation reflects the real time in minutes. The simulator design is based on a popular Python library SimPy[43] which is a process-based discrete event simulation framework designed to implement multi-agent systems. The implementation of the simulator is presented in Appendix C.

Using the simulator, it is possible to model the behavior of $M/M/N$ as well as $M/G/N$ queuing systems. The simulator can be configured with separate simulation and test execution times as well as with the parallel test execution capacity. The design corresponds to a single queue and multiple servers which are being added until there is no more waiting in queue for incoming events. The simulator process workflow is represented with the flowchart in Figure 6.1.

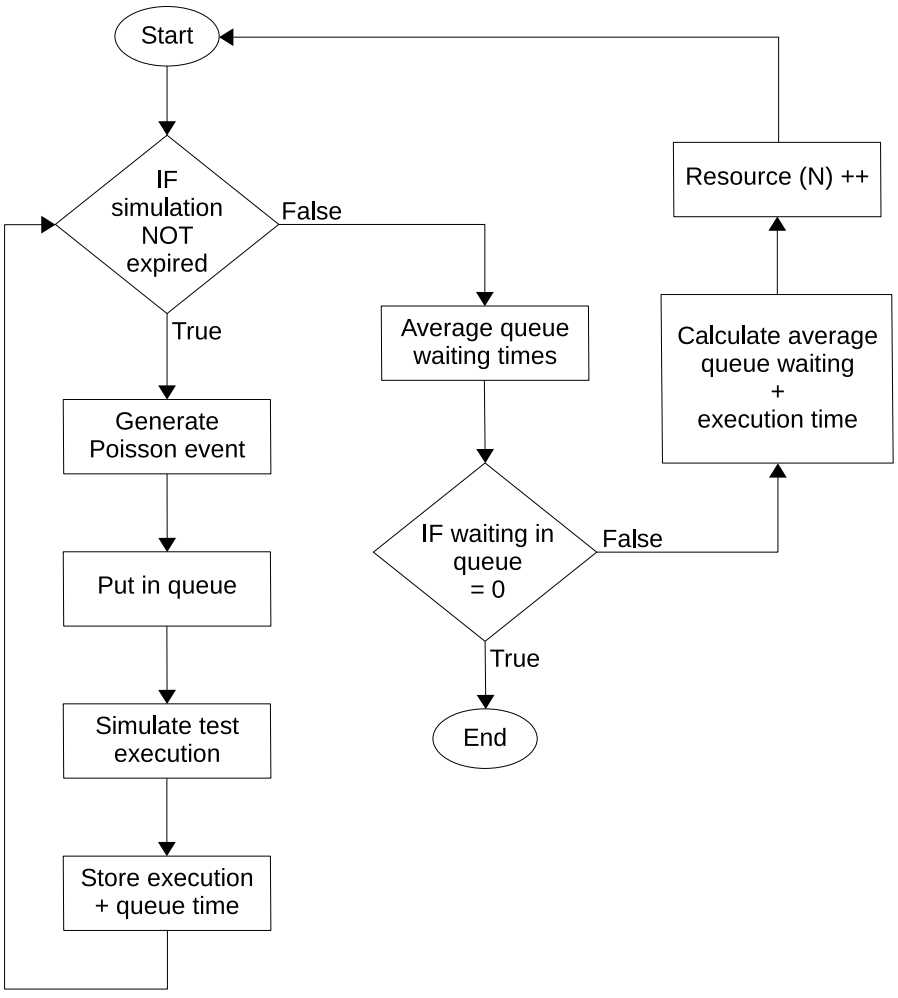



Figure 6.1: Simulator design flowchart.

The simulation workflow consists of two loops: the first is depicted on the left side of Figure 6.1, and the second is depicted on the right. When the simulation starts, the first

[43]SimPy, `https://simpy.readthedocs.io/`

loop is executed until the simulation timeout is reached. The loop starts by generating a Poisson event that is pushed into a waiting queue until a resource becomes available. Upon resource availability, simulation of the test execution is performed. The simulation of the test execution consists of computing a random variable that is used as the time for occupying one resource. The random variable can be either exponential if we are simulating an $M/M/N$ queuing system or a uniform random variable in the case of an $M/G/N$ queuing system as determined in Section 5.6. At the end of the iteration, the execution and waiting times in the queue are stored, and the cycle is repeated until the simulation termination time is reached.

When the simulation termination time is reached, the simulator enters the second loop. If the waiting time in the queue is zero, the simulation is ended since adding new resources would not change the result. If the waiting time is greater than zero, then the average waiting time in the queue is computed, the combined average waiting and execution times are stored, and additional processing resources are added. This cycle is then repeated until there are enough resources for the waiting time in the queue to become zero. Finally, the collected data of the combined waiting and execution averages are plotted on a chart for the number of resources that were used in a particular simulation cycle.

The input parameters of the simulator are:

1. test arrival rate $\lambda$,
2. average test duration in minutes h, and
3. simulation duration in minutes.

### 6.1.2 Numerical Computation of Testing Resources

Another technique that can complement and simplify searching for analytical solutions is the numerical approach. Since analytical solution for isolating $N$ from Eq. (5.9) is very complex if feasible at all [90], [101], we implemented Algorithm 6.1 to numerically compute the number of needed resources $N$ to satisfy the required $P_{GoS}$. This way it is easy to find the required number of resources in just a few iterations within the numerical algorithm. The resulting resources correspond to the number of parallel test environments.

**Algorithm 6.1:** Calculation of $P_{GoS}(n)$ until it is less than the required $P_{GoS}(N)$, thus identifying the number of required resources $N$.

**Input:** $0 \leq P_{GoS}(N) \leq 1$
**Result:** $N \leftarrow ?$
$n \leftarrow 1$
**while** $P_{GoS}(n) < P_{GoS}(N)$ **do**
    $n++$
**end**
**return** $n$

The algorithm is based on analytical formulas therefore only valid for the $M/M/N$ queuing model. The implementation of the numerical algorithm with supporting functions is presented in Appendix B.

## 6.2 Preliminary Results

### 6.2.1 $P_{GoS}$ for a Single Test Environment

Considering the peak hourly as well as daily probability of contribution per developer discussed in Chapter 5, we can calculate the average test arrival rate $\lambda$ by Eq. (5.11) and probability of the grade of service $P_{GoS}$ by Eq. (5.9). We are interested in what is the probability that a test will be executed on the testbed without being first placed in a waiting queue for different developer team sizes.

Assuming the goal for a testing environment is to handle 90% of the situations considering 5 daily contributions per developer or 99% of situations considering 9 daily contributions per developer, we can gather the results in Table 6.1.

Table 6.1: Average arrival rate and $P_{GoS}$ for a single test environment for different developer team sizes.

| Team size | 90% | | 99% | |
|---|---|---|---|---|
| | $\lambda$ | $P_{GoS}$[%] | $\lambda$ | $P_{GoS}$[%] |
| 10 developers | 5 | 95.7 | 9 | 87.5 |
| 20 developers | 10 | 84.4 | 18 | 28.7 |
| 50 developers | 25 | 0 | 45 | 0 |
| 100 developers | 50 | 0 | 90 | 0 |

We can conclude that no further optimization is required for the case of 10 developers with an average test duration of 3 minutes and maximum waiting time of 7 minutes since almost 90% of all the tests will be executed on a single test environment without any waiting for both 90% and 99% of developers' contributions. Whereas, for 20 or more developers, further investigation is required.

### 6.2.2 Simulator Performance Evaluation

To compare the simulated waiting times to those obtained through analytical computation based on Little's theorem, we ran a long-lasting simulation of the queue wait times and depicted the results in Figure 6.2. Since the numbers are hard to read from the figure, the results are summarized in Table 6.2. The first column of the table shows the number of test environments $N$ used in a particular stage of the simulation, while the next columns show the simulated as well as the analytical results for different values of the Erlang traffic. The results of the simulation provide the queue wait times with 95% confidence intervals.

Example values chosen for $A$ used for the simulator evaluation consider the following input parameters:

- 90% of daily developer contributions $r = 5$
- average test duration $h = 3$ min, and
- team size $k$ equal to 20, 50 and 100 developers.

Concretely, the first row in Table 6.2 is interpreted as follows: in the case where there is only one resource available, i.e. $N = 1$ and Erlang traffic $A = 0.5$, the analytical calculation of the wait time in the queue for a resource to be free based on Little's theorem (defined in Eq. (5.6)) results in $w = 3$ minutes. However, the simulation result with a 95% confidence

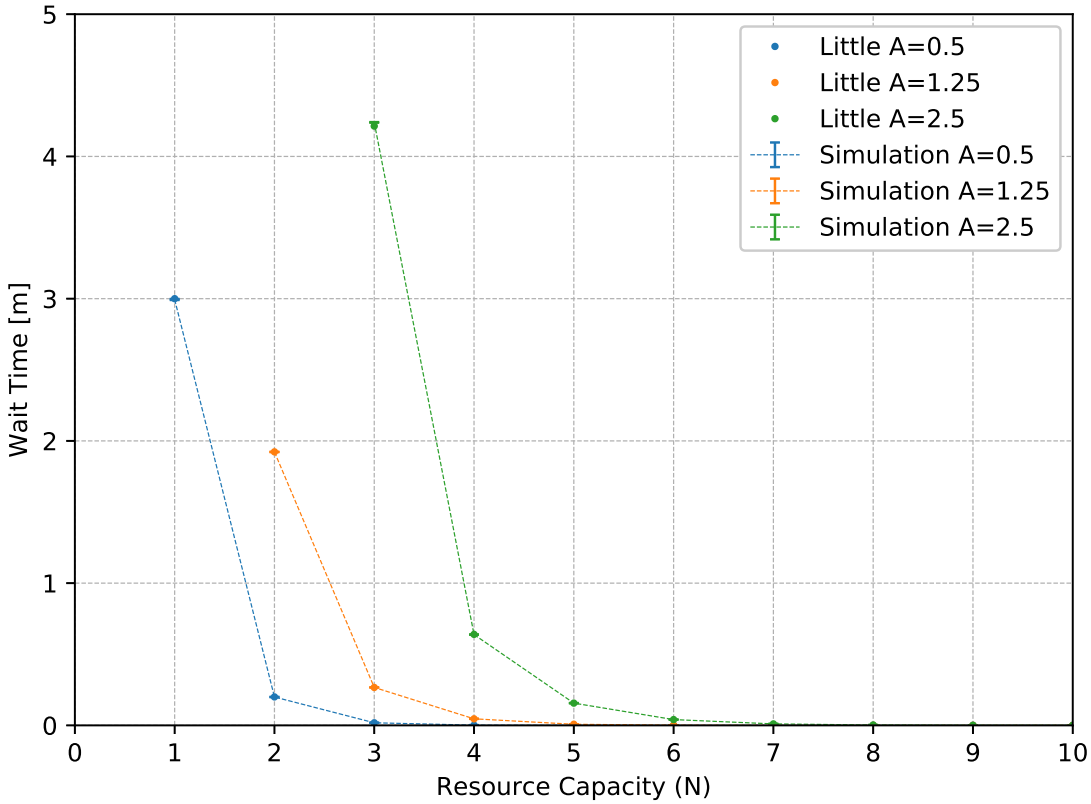



Figure 6.2: Comparison of queue wait times in minutes between the simulation results and the analytical calculation based on Little's theorem.

Table 6.2: Queue wait times in minutes obtained by the simulation and the analytical calculation using Little's theorem.

| N | A = 0.5 | | A = 1.25 | | A = 2.5 | |
|---|---|---|---|---|---|---|
| | Sim. [m] | Little [m] | Sim. [m] | Little [m] | Sim. [m] | Little [m] |
| 1 | 2.994±0.004 | 3.0 | NA | NA | NA | NA |
| 2 | 0.199±0.001 | 0.2 | 1.922±0.001 | 1.923 | NA | NA |
| 3 | 0.018±0.0 | 0.018 | 0.267±0.0 | 0.267 | 4.239±0.002 | 4.213 |
| 4 | 0.002±0.0 | 0.002 | 0.046±0.0 | 0.046 | 0.638±0.0 | 0.64 |
| 5 | 0.0±0.0 | 0.0 | 0.008±0.0 | 0.008 | 0.156±0.0 | 0.156 |
| 6 | 0.0±0.0 | 0.0 | 0.001±0.0 | 0.001 | 0.041±0.0 | 0.041 |
| 7 | 0.0±0.0 | 0.0 | 0.0±0.0 | 0.0 | 0.01±0.0 | 0.01 |
| 8 | 0.0±0.0 | 0.0 | 0.0±0.0 | 0.0 | 0.002±0.0 | 0.002 |
| 9 | 0.0±0.0 | 0.0 | 0.0±0.0 | 0.0 | 0.001±0.0 | 0.001 |
| 10 | 0.0±0.0 | 0.0 | 0.0±0.0 | 0.0 | 0.0±0.0 | 0.0 |

interval for the same input is $w = 2.994 \pm 0.004$ minutes. In the cases with increased Erlang traffic $A = 1.25$ and $A = 2.5$, the result $NA$ means that the queue gets full and the system never converges to a solution, and only 1 resource is not enough to handle traffics of these

sizes. The same logic applies to the rest of the table, where in most cases, the analytical calculation of the queue wait times is within the very narrow confidence intervals of the simulated times. Therefore, the simulation matches the analytical results exceptionally well. The simulation duration equivalent to 10 years in real time, which results in almost 5 million iterations, was intentionally chosen to be extremely long to demonstrate the simulated wait times converging to the analytical solutions.

## 6.3 Testbed Capacity Dimensioning

Within this chapter we provide the simulation results of test execution times at different test capacities for developer teams with 20, 50 and 100 developers assuming two different types of queues, $M/M/N$ with exponential distribution of test execution times and mean duration equal to 3 minutes, and $M/G/N$ with uniformly distributed test execution times between 2 and 5 minutes. The later parameters are based on empirical experience obtained from the VESNA firmware testing where compiling and flashing of nodes as the fixed part of the test takes almost 2 minutes. The parameters from the table had been previously defined in Table 5.1 and Eq. 5.11 and, where applicable, generic values had been derived for busy developer hour percentage $p$, team size $k$, daily developer contributions $r$, and test servicing time $h$, respectively. The last parameter is the simulation duration equivalent to 1 year in real time specified in minutes, which results in almost 500 thousand iterations. The specific values of parameters used for the following simulations with the corresponding Chapter 5 sections of the derivation are collected in Table 6.3.

Table 6.3: Simulation parameters for $M/M/N$ and $M/G/N$ type of queues as per Table 5.1 and Eq. 5.11.

| Parameter | Values | Derivation |
|---|---|---|
| $p$ | 10% | Section 5.2 |
| $k$ | 10, 20, 50, 100 | Section 5.3 |
| $r$ | 5, 6, 9 | Section 5.5 |
| $h$ | $M$ 3 min, $G$ [2, 5] min | Section 5.6 |

### 6.3.1 $M/M/N$ Queueing System Simulation

The results discussed could not be obtained without the discrete event simulator designed specifically to simulate testbed queueing execution, with which we simulated year's worth of testing executions in a matter of minutes. In this particular case we consider 90% of the total code contribution events that correspond to 5 daily contributions per developer. The averages are plotted with corresponding confidence intervals and standard deviations.

Figure 6.4 depicts a long-term average of testbed execution assuming $M/M/N$ type of queue. Focusing on the 10 minutes line marked with red, representing the upper limit for test waiting times, i.e. a golden standard time frame for CI, we can observe that the average waiting time is well below 10 minutes in case of:

- 20 developers served with 1 test environment.
- 50 developers served with 2 parallel test environments.
- 100 developers served with 3 parallel test environments.

However, we need to take into account also the standard deviation from the average waiting times if we want to ensure that none of the developers waits more than 10 minutes for the test results. Figure 6.4 depicts results for average waiting time and its standard deviation for the simulation assuming $M/M/N$ type of queue. Here the number of required test environments for the average waiting time below 10 minutes is as follows:

- 2 parallel test environments for 20 developers.
- 2 parallel test environments for 50 developers.
- 4 parallel test environments for 100 developers.

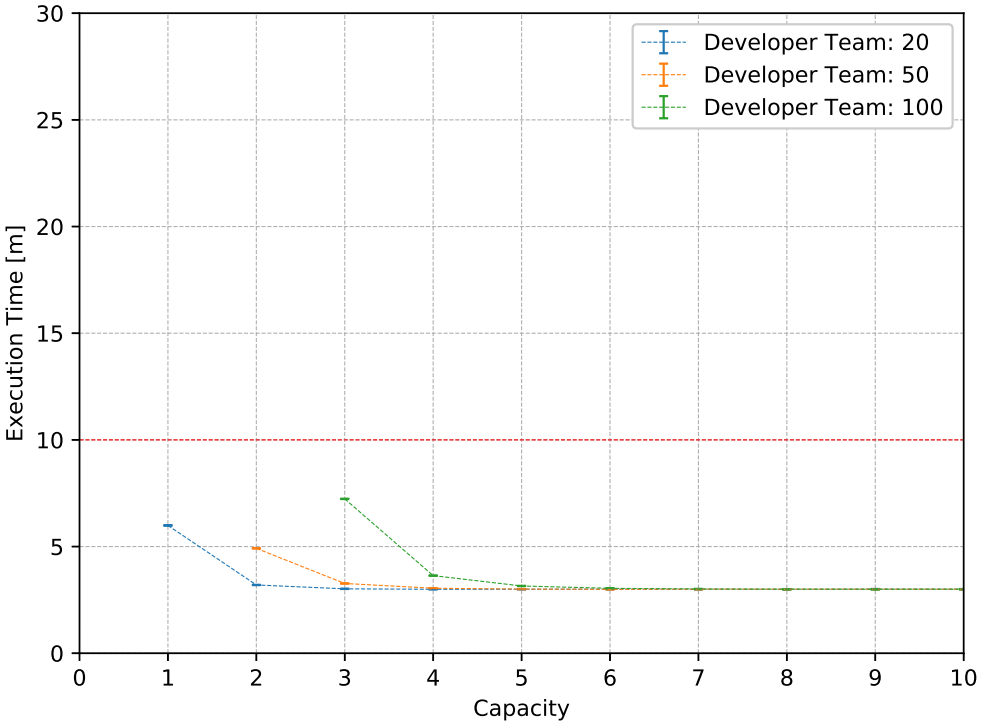



Figure 6.3: Simulation for $\lambda_{90\%}$ considering 5 daily developer contributions for $M/M/N$ with 95% confidence intervals.

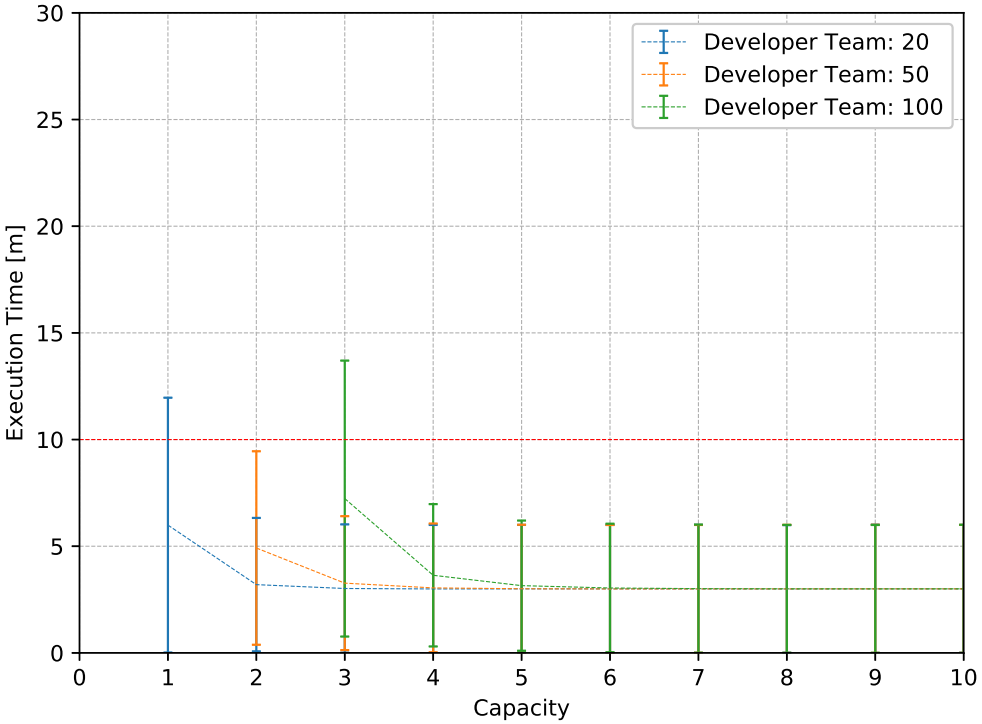



Figure 6.4: Simulation for $\lambda_{90\%}$ considering 5 daily developer contributions for $M/M/N$ with standard deviation.

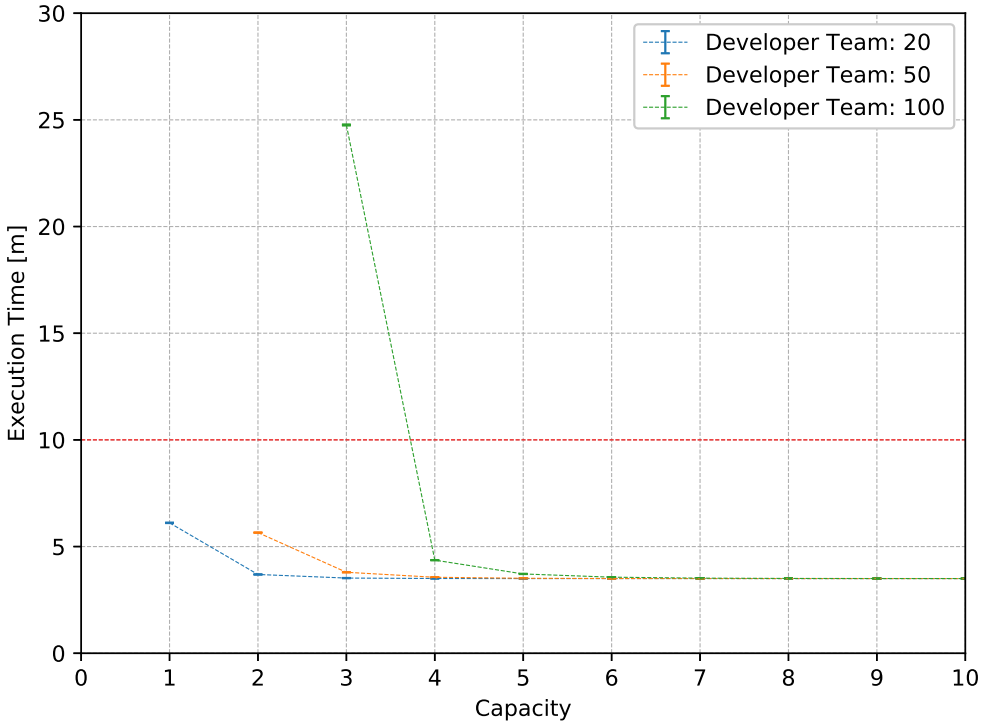



Figure 6.5: Simulation for $\lambda_{90\%}$ considering 5 daily developer contributions for $M/G/N$ with 95% confidence intervals.

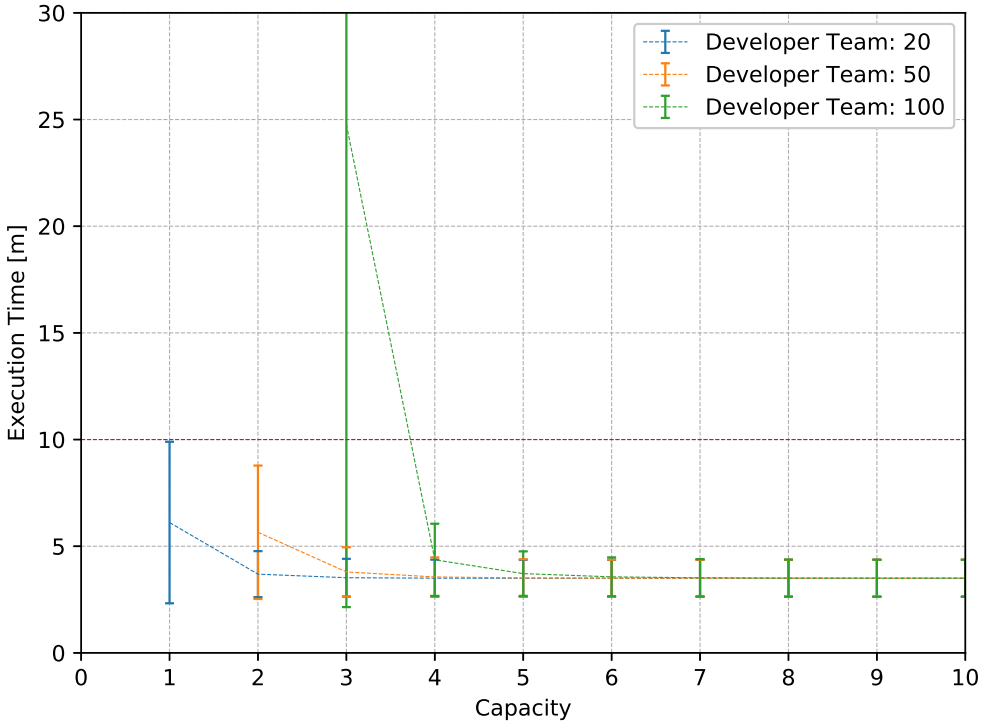



Figure 6.6: Simulation for $\lambda_{90\%}$ considering 5 daily developer contributions for $M/G/N$ with standard deviation.

### 6.3.2 $M/G/N$ Queueing System Simulation

Figure 6.5 depicts simulation results for average values with 95% confidence intervals for the $M/G/N$ type of queue and uniformly distributed random duration of tests in the interval between 2 and 5 minutes, which is a more realistic use case for CI-supported development of wireless embedded systems. Since the simulation termination is around 500 thousand, iterations the confidence intervals are accordingly narrow. Therefore, standard deviation provides better insight in the system behavior. We can observe that the behavior for 20 and 50 developers is very similar as for the $M/M/N$ queue, however, for 100 developers only 3 parallel test environments are not sufficient. Thus, in this case:

- 20 developers need 1 test environment.
- 50 developers need 2 parallel test environments.
- 100 developers need 4 parallel test environments.

Finally, Figure 6.6 also depicts the simulation for the $M/G/N$ type of queue with the standard deviation from the average waiting times. Here we can see that for 20 developers the average waiting time is far below 10 minutes and even the standard deviation stays below that mark, indicating that even development teams of 20 developers do not need to worry about parallel test environments. This greatly simplifies the test infrastructure for very common small to medium size teams, whereas the requirements for teams of 50 and 100 developers in terms of the number of required test environments for the average waiting time below 10 minutes remain the same as for the the $M/M/N$ type of queue:

- 1 test environment for 20 developers.
- 2 parallel test environments for 50 developers.
- 4 parallel test environments for 100 developers.

### 6.3.3 Comparison of Results

Table 6.4 summarizes simulation results and compares them with the analytical calculation for the necessary parallel testing environments to achieve the required $P_{GoS}$ calculated by implementing Algorithm 6.1. For analytical calculation we used a well-known rule for designing high availability systems, a so-called five-nines rule which is $P_{GoS} = 99.999\%$. This is a very rigorous rule and we can observe from Figure 6.4 that using this rule would introduce a substantial over-dimensioning of the test system.

The most important difference is that Figure 6.4 indicates that 2 parallel testing environments are required already for a team with 20 developers, whereas a more realistic simulation results in Figure 6.6 demonstrate that there is no need for the second test environment since all 20 developers will be served in under 10 minutes. Contrarily, the analytical result indicates that no more than $P_{GoS} = 84.4\%$ of developers will be served with a single testing environment in under 10 minutes in a team with 20 developers. The reason for different results are the two different distributions of test duration. Analytical approach considers the exponential distribution and our realistic simulation the uniform distribution.

For the sake of completeness of the study we also provide the results when we choose to support 95% of all development situations which corresponds to an average of 6 daily contributions per developer and 99% of all development situations which corresponds to an average of 9 daily contributions per developer. The interpretations of the results in

Table 6.4: Analytical and simulation results for determining required testbed capacity assuming $M/M/N$ and $M/G/N$ queues for $\lambda_{90\%}$ corresponding to 5 daily developer contributions.

| Dev. | Analytical Results $P_{GoS} = 99.999\%$ | Simulations $M/M/N$ | Simulations $M/G/N$ |
|---|---|---|---|
| 20 | 4 | 2 | 1 |
| 50 | 5 | 2 | 2 |
| 100 | 7 | 4 | 4 |

Figure 6.7 and Figure 6.8 are quite interesting. By looking at the graph for 20 developers in Figure 6.7 we could easily make a compromise and use only a single test environment since only a small part of the tests would take longer than 10 minutes in 95% of all development situations. However, when we take a look at Figure 6.8, we can see that in the case of increased development activity when we approach 99% of development situations, only one test environment would fail while the queue would fill up and the system would become nonresponsive. Finally, by adding a second parallel test environment, the system would easily handle 99% of all development situations.

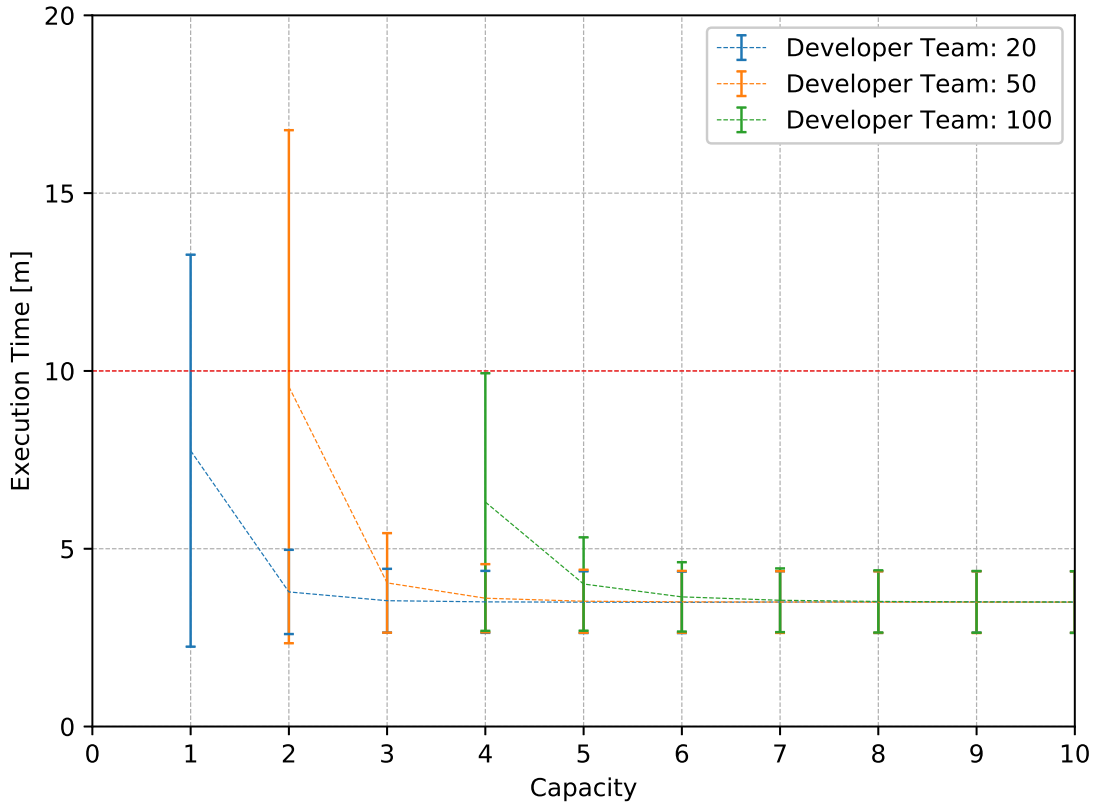



Figure 6.7: Simulation for $\lambda_{95\%}$ indicating 95% of development situations corresponding to 6 daily contributions per developer for $M/G/N$ with standard deviation.

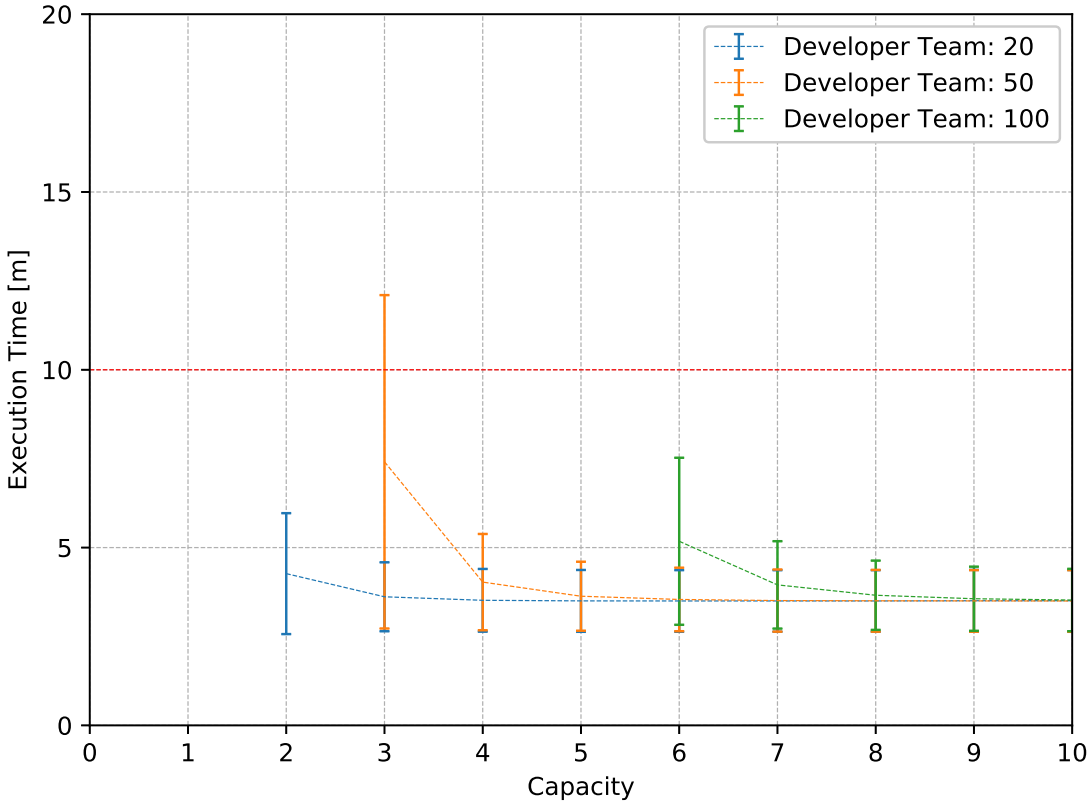



Figure 6.8: Simulation for $\lambda_{99\%}$ indicating 99% of development situations corresponding to 9 daily contributions per developer for $M/G/N$ with standard deviation.

Similar situation appears in case of 50 developers where having two test parallel environments indicates that only part of developers would have to wait more than 10 minutes while standard deviation stretches almost to 20 minutes. However, the average waiting time is still below 10 minutes. Then again, if there is an increased development intensity corresponding to 99% of development situations, only two parallel test environments would fail to deliver any test result since the queue would fill up. Increasing the test environments to 3 would make a testing system responsive even for 99% of development situations for most developers.

Having a large team of 100 developers working on the same project would require at least 4 parallel test environments which would handle 95% of development situations. However, if the development suddenly intensifies coming close to 99% of development situation having only 4 parallel test environments would fail to deliver test results. Actually it is evident from Figure 6.8 that even 5 parallel test environments would fail to deliver test results in this case. Therefore, in case of a development team of 100 developers and requirement of handling 99% of development situations, the required number of parallel test environments $N$ is 6.

## 6.4 Discussion

The results of the simulations as well as the numerical calculations are collected for easier comparison in Table 6.5. The first column gives the number of developers in a particular team, i.e. 20, 50 and 100. The next three columns group data based on the daily developer

contribution CDF defined in Section 5.5 considering 95% of all development situation which corresponds to 6 contributions per developer per day and is marked by $\lambda_{95\%}$. The last three columns give data for 99% of all development situations which corresponds to 9 contributions per developer per day and is marked by $\lambda_{99\%}$. The two groups of data use the corresponding $\lambda$ for subsequent calculations and simulations. The first column of the two groups each under its own $\lambda$ is the five-nines probability of the grade of service $P_{GoS} = 99.999\%$ which shows the number of required parallel test environments $N$ to achieve the desired goal of delivering test results to practically all developers within 10 minutes. The calculation is based on numerical Algorithm 6.1 defined in Section 6.1.2 modeling the $M/M/N$ queueing system.

Table 6.5: Analytical and simulation results for determining the required number of parallel test environments $N$ assuming $\lambda_{95\%} = 6$ and $\lambda_{99\%} = 9$ of developer contributions.

| Dev. | $\lambda_{95\%}$ | | | $\lambda_{99\%}$ | | |
|---|---|---|---|---|---|---|
| | $P_{GoS} = 99.999\%$ | $P_{GoS} = 95\%$ | Sim. | $P_{GoS} = 99.999\%$ | $P_{GoS} = 95\%$ | Sim. |
| 20 | 4 | 2 | 2 | 5 | 2 | 2 |
| 50 | 5 | 3 | 3 | 6 | 4 | 4 |
| 100 | 7 | 4 | 4 | 9 | 6 | 6 |

Since we previously demonstrated that the $M/M/N$ queuing system is not-well suited for modeling of the real testing situation, we repeated the calculation for the grade of service set to $P_{GoS} = 95\%$. Therefore, the next column is $P_{GoS} = 95\%$ which explains that only 95% of all developers will receive the testing results within 10 minutes considering the calculated number of parallel test environments $N$. The final column presents the results interpreted from the simulations performed for the $M/G/N$ queueing system presented in Section 6.3.2 based on the corresponding input parameter $\lambda$ same as the one used in numerical calculations.

The results in the first group under $\lambda_{95\%}$ corresponding to 6 daily developer contributions suggest that a team of 20 developers requires 4 testing environments if we only consider the less optimal $M/M/N$ queueing system and $P_{GoS} = 99.999\%$. If we use less strict probability of the grade of service $P_{GoS} = 95\%$ meaning that only 95% of considered tests will complete within 10 minutes, the resulting number of parallel test environments $N$ is 2. This is the same result as we observed in the simulation for $M/G/N$ system model. However, the more realistic $M/G/N$ queueing system simulation revealed that actually all the tests and not only 95%, as numerical results suggested, will complete within 10 minutes.

The results in the second group under $\lambda_{99\%}$ corresponding to 9 daily developer contributions suggest that a team of 20 developers requires 5 testing environments if we only consider numerical results for $N$ in case of $P_{GoS} = 99.999\%$. The less strict probability of the grade of service $P_{GoS} = 95\%$ results in required $N$ equal 2. This is again the same result as we observed in the simulation. However, the simulation considers all and not only 95% of executed tests to complete within 10 minutes.

For a team of 50 developers results are similar, only the number of required test environments increased in all cases. The same observation applies for the team of 100 developers. There are two additional observations. The first is that it is evident that by applying the five-nines rule in combination the with analytical approach for $M/M/N$ queueing system, which is a common practice when designing high availability systems, we would end up with an over-dimensioned testing system. The second observation is that when we compare

the analytical calculation for $P_{GoS} = 95\%$ with the simulated ones, they match perfectly. However, the more realistic testing process simulation reveals that actually all the tests, and not only 95% as the analytical results suggest, will complete within 10 minutes.

We can conclude that the calculations using standard equations can be used as a good first estimate. However, simulations provide a much deeper insight in real test execution times and therefore better determine the required number of parallel testing environments.

These results reflect a more realistic development situation when using the CI development approach for wireless firmware development and testing on real hardware in a real wireless environment. We have shown with simulations that a simple analytical approach comes very close to the real situation, if the input variables are carefully chosen. However, simulating a more realistic model can save a lot of effort and money to small and medium development teams under 20 developers since we demonstrated that they do not require parallel test environments for efficient agile and CI-supported development of wireless firmware.

# Chapter 7

# Conclusions

In this thesis, we addressed the problem of wireless firmware development from the perspective of development practices and supporting tools and environments, particularly during the testing and integration phase, and discussed how such development could benefit from modern practices present in general software development such as CI. As a possible solution we proposed COINS framework, which enables automated testing based on a real wireless testbed. Further challenges, especially test parallelization and isolation, which are not substantial for general software development, have been recognized as very important for the successful introduction of CI in wireless embedded systems development. In the case of wireless embedded systems, the source code of the firmware needs to be tested in a real environment and on the target hardware in a dedicated frequency band.

To obtain a deep insight in developer activity required for dimensioning of development tools and environments, we conducted a comprehensive analysis of developers' activity to ensure the feasibility of the streamlining approach we intended to implement. We have demonstrated that queueing theory is well suited for modeling of the testing process for wireless embedded systems development. There is an analytical approach which is useful for preliminary analysis, however, it is more practical and realistic to perform computer simulations to observe the behavior of the testing systems for supporting developers. It is important to differentiate between developers' behavior during the modeling process. Professional developers can behave different than FOSS developers. Another behavior observation which we came across during the analysis is that the developers working during weekends and holidays behave differently that the ones working during the working days.

To validate the proposed framework COINS, we implemented the CI-supported wireless firmware development system on top of the advanced heterogeneous outdoor wireless testbed LOG-a-TEC. We developed a new application layer protocol to enable efficient communication between the wireless embedded system and the infrastructure to enable the advanced testing on target nodes as well as the rapid development of new wireless technologies and protocols.

Additionally, we used the measurements from the real testbed to make an estimation of separate test durations and at the end combined all this knowledge in a simulation for optimization of a testing environment. The aim was to deliver test results to developers within 10 minutes while trying not to over-dimension the typically expensive test execution environments. For this purpose we introduced a new concept of developers' activity which takes into account the percentage of developers' contributions during the busy development hour, the number of all daily developers' contributions as well as the development team size to be able to model the testing process.

## 7.1 Summary of Contributions

For this thesis we carried out an extensive analysis of software developers' activity based on a data mining approach conducted on a large scale dataset collected from the most popular social coding platform GitHub. The result was a new data-driven probability distribution model off the frequency of committed software code per developer per day. Additionally, we showed the commit distribution for a developer team within 24 hours and used those findings as parameters for the testbed capacity evaluation making use of the queueing theory approach.

Next we proposed a completely new reference framework for efficient introduction of CI practices in wireless embedded systems development COINS, which simplifies the code development process on real wireless testbeds and enables frequent code changes. We realized a reference implementation of COINS by integrating existing open-source components and cloud services into a fully functional system supporting CI for wireless networks. We performed a qualitative comparison of the proposed framework with the existing related frameworks. The comparison considers a list of features already used in the community for comparing experiment and testing management tools. However, to fully illustrate the advantage of the proposed framework, we extended the available list of features with four CI-related ones.

We further identified a void in the field of embedded protocols for simple and efficient communication with the infrastructure. Therefore, we defined and implemented a new application layer communication protocol LCSP for this purpose.

Finally, we proposed and developed a simulator-based evaluation of test execution times in the CI process. It combines the inputs based on the analysis findings and the real testbed execution times measurements to determine the optimal testbed capacity for the specific developer team size.

## 7.2 Future Work

Future work can be done in several areas elaborated in this thesis. In the area of developers' behavior modeling, some further work is required in the field of fitting functions used to derive the developer activity models. A more sophisticated function to model the pushes per developer per day could be used instead of exponential function. The proposed fitting function does not provide a perfect accuracy in the lower part where the exponential function drops slightly too fast and does not model perfectly the data between 4 and 8 pushes. A possible candidate function which could improve the model could be a power-law function. However, for the work presented in this thesis, the deviation of the model is negligible.

Another possible future work would be a comprehensive comparison between workday and weekend developers, since the work presented in this thesis takes into consideration only workdays. We observed different patterns of behavior for the two distinct groups of developers.

There is further work required also in the area of wireless embedded devices lifecycle management. Continuous software development should also consider continuous delivery of the builds in production. Thus, further DevOps practices could be introduced in the proposed COINS framework to extend it with continuous delivery capabilities to solve the issue of wireless devices which are not maintained after entering the market. Continuous delivery could also solve security issues in devices on the market. Currently, every electronics producer addresses the problem of continuous delivery separately, however, it would be beneficial to find some standard method.

Finally, a more comprehensive study would be required to determine how the findings from this thesis in terms of testbed infrastructure dimensioning could benefit the software development organizational management teams to even further optimize the required testbed capacities, for instance by encouraging the developers to make timely contributions to achieve more deterministic test distribution. It is a topic closer to the field of social informatics, moreover, it can also be considered important in the management of teams of developers and development resources.

# Appendix A

# Python Code for LoRa Firmware CI Test Based on LCSP Protocol

```
import os
import unittest
import requests
from vesna import alh
from time import sleep

class LoraTests(unittest.TestCase):
    def setUp(self):
        port = 9000

        rx_ip = os.environ.get("LORA_RX")
        tx_ip = os.environ.get("LORA_TX")
        self.dev = "/dev/ttyUSB0"

        self.lora_rx = "http://%s:%s" % (rx_ip, port)
        self.lora_tx = "http://%s:%s" % (tx_ip, port)
        self.lora_rx_api = self.lora_rx + "/communicator"
        self.lora_tx_api = self.lora_tx + "/communicator"

        for x in range(100):
            try:
                requests.get(self.lora_rx)
                requests.get(self.lora_tx)
                break
            except:
                sleep(5)

    def test_lora_network(self):
        freq = 868100000
        bw = 125
        sf = 12
        cr = "4_5"
        pwr = 2
```

```
        test_msg = "hello lora network"

        node_rx = alh.ALHWeb(self.lora_rx_api, self.dev)
        node_tx = alh.ALHWeb(self.lora_tx_api, self.dev)

        print(node_rx.get("loraRadioInfo").text)
        print(node_tx.get("loraRadioInfo").text)

        prm = "frequency=%s&bw=%s&sf=%s&cr=%s" % (freq, bw,
            sf, cr)
        res = node_rx.get("loraRxStart", prm)
        print(res.text)

        prm = "frequency=%s&bw=%s&sf=%s&cr=%s&pwr=%s" % (freq
            , bw, sf, cr, pwr)
        res = node_tx.post("loraTxStart", test_msg, prm)
        print(res.text)

        for i in range(10):
            res = node_rx.get("loraRxRead")

            if "No packet received" not in res.text:
                break

            sleep(1)

        print(res.text)

        self.assertTrue(test_msg in res.text)

if __name__ == '__main__':
    unittest.main()
```

# Appendix B

# Python Code for Solving the $M/M/N$ Queueing System

```
from math import factorial, exp

# Erlang traffic for software development proposed in this
# thesis where p is 10%, r is the number of developer
# contributions, k is the number of developers and
# h is the average test duration in minutes

def erlang_traffic(r, k, h):

    p = 0.1

    A = p * r * k * h / 60

    return A

# Calculation of the Erlang-C formula where A is Erlang
# traffic and N is the number of resources

def erlang_C(A, N):

    C = (A**N / factorial(N)) * (N / (N - A))

    sum_ = 0
    for i in range(N):
        sum_ += (A**i) / factorial(i)

    P = (C / (sum_ + C))

    return P

# Calculation of the probability of the grade of service PGOS
# where A is Erlang traffic, N is the number of resources,
# h is average test duration and w is waiting time in
```

```
# the queue in minutes

def service_level(A, N, h, w):

    sl = 1 - erlang_C(A, N) * exp(-(N - A) * (w/h))

    return sl*100

# Waiting time in the queue based on Little's theorem

def queue_wait_time(A, h, N):

    W = erlang_C(A, N) * (h / (N - A))

    return W

# Calculation of the required number of resources N for
# achieving the desired probability of the grade of service

def resource_number(A, h, w, PGOS):

    i = 1
    while service_level(A, i, h, w) < PGOS:
        i = i + 1

    return i
```

# Appendix C

# Python Code of the Testing Simulator

```
import simpy
import numpy as np
from random import uniform, expovariate

# h is the average test duration in minutes
# MMN indicates exponential distribution with
# the average duration of 3 minutes and MGN
# indicates general distribution on the
# interval from 2 - 5 minutes

h_mmn = 3
h_mgn = (2, 5)

# Simulation duration in minutes equivalent of one week

duration = 7*24*60

# p used in this thesis is 10%

p = 0.1

# r is the number of developer contributions
# r = 5 for 90%, r = 6 for 95% and r = 9 for 99%

r = 5

# Developer team of 20, 50 and 100

k1 = 20
k2 = 50
k3 = 100

# Lambda calculation for software development proposed
```

```
# in this thesis: lambda = p * r * k

lambdas = [p*r*k1, p*r*k2, p*r*k3]

# Boolean variable to define MGN queue

MGN = True

# Poisson process test scheduler

def schedule(env, tests, nodes):
    while True:
        m = test(env, tests, nodes)
        env.process(m)
        yield env.timeout(expovariate(tests/60.0))

# M/M/N or M/G/N test execution simulation

def test(env, tests, nodes):
    arrive = env.now

    with nodes.request() as req:
        yield req
        q_times.append(env.now - arrive)

        if MGN:
            yield env.timeout(uniform(*h_mgn))
        else:
            yield env.timeout(expovariate(1.0/h_mmn))

    s_times.append(env.now - arrive)

# Loop executes simulations for different team sizes

for tests in lambdas:
    # Result arrays averages with standard deviation

    res_avg = []
    res_std = []

    # Simulation initialization where i represents N the
    # number of parallel test environments where tests
    # are scheduled according to the Poisson process

    for i in range(1, 10):
        # q_times is the array of times spent in the queue
        # combined with the service times s_times

        q_times = []
        s_times = []
```

```
        env = simpy.Environment()
        nodes = simpy.Resource(env, capacity=i)
        env.process(schedule(env, tests, nodes))
        env.run(until=duration)

        # Ignoring too long queue waiting and no waiting
        # and collecting the desired results

        q_mean = np.mean(q_times)
        if q_mean == 0:
            break
        elif q_mean > 30:
            continue
        else:
            res_avg.append(np.average(s_times))
            res_std.append(np.std(s_times))

print("No. of developers: " + str(int(tests/p/r)))
print("Averages:")
print(res_avg)
print("Standard deviations:")
print(res_std)
```

# References


[1] M. Condoluci, M. Dohler, G. Araniti, A. Molinaro, and K. Zheng, "Toward 5G densenets: Architectural advances for effective machine-type communications over femtocells," *IEEE Communications Magazine*, vol. 53, no. 1, pp. 134–141, Jan. 2015, ISSN: 1558-1896. DOI: 10.1109/MCOM.2015.7010526.

[2] E. De Poorter, J. Hoebeke, M. Strobbe, I. Moerman, S. Latré, M. Weyn, B. Lannoo, and J. Famaey, "Sub-GHz LPWAN network coexistence, management and virtualization: An overview and open research challenges," *Wirel. Pers. Commun.*, vol. 95, no. 1, pp. 187–213, Jul. 2017, ISSN: 0929-6212. DOI: 10.1007/s11277-017-4419-5. [Online]. Available: https://doi.org/10.1007/s11277-017-4419-5.

[3] G. Sklivanitis, A. Gannon, S. N. Batalama, and D. A. Pados, "Addressing next-generation wireless challenges with commercial software-defined radio platforms," *IEEE Communications Magazine*, vol. 54, no. 1, pp. 59–67, Jan. 2016, ISSN: 1558-1896. DOI: 10.1109/MCOM.2016.7378427.

[4] A. Gudipati, D. Perry, L. E. Li, and S. Katti, "Softran: Software defined radio access network," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013, pp. 25–30.

[5] D. Riehle, P. Riemer, C. Kolassa, and M. Schmidt, "Paid vs. volunteer work in open source," in *2014 47th Hawaii International Conference on System Sciences*, Jan. 2014, pp. 3286–3295. DOI: 10.1109/HICSS.2014.407.

[6] P. Rosove, *Developing Computer-based Information Systems*, ser. Information Sciences. John Wiley, 1967. [Online]. Available: https://books.google.si/books?id=tNUDAAAAMAAJ.

[7] R. Kneuper, "Sixty years of software development life cycle models," *IEEE Annals of the History of Computing*, vol. 39, no. 3, pp. 41–54, 2017, ISSN: 1934-1547. DOI: 10.1109/MAHC.2017.3481346.

[8] M. Fowler, *Continuous integration*, http://www.martinfowler.com/articles/continuousIntegration.html, Accessed on 22.1.2020, 2006.

[9] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007, ISBN: 0321336380.

[10] B. Greene, "Agile methods applied to embedded firmware development," in *Agile Development Conference*, Jun. 2004, pp. 71–77. DOI: 10.1109/ADEVC.2004.3.

[11] B. Broekman and E. Notenboom, *Testing Embedded Software*. Addison-Wesley, 2003, ISBN: 9780321159861.

[12] S. Wahle, C. Tranoris, S. Denazis, A. Gavras, K. Koutsopoulos, T. Magedanz, and S. Tompros, "Emerging testing trends and the panlab enabling infrastructure," *IEEE Communications Magazine*, vol. 49, no. 3, pp. 167–175, Mar. 2011, ISSN: 1558-1896. DOI: 10.1109/MCOM.2011.5723816.

[13] K. Könnölä, S. Suomi, T. Mäkilä, T. Jokela, V. Rantala, and T. Lehtonen, "Agile methods in embedded system development: Multiple-case study of three industrial cases," *Journal of Systems and Software*, vol. 118, pp. 134–150, 2016, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.05.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216300413.

[14] M. Kaisti, V. Rantala, T. Mujunen, S. Hyrynsalmi, K. Könnölä, T. Mäkilä, and T. Lehtonen, "Agile methods for embedded systems development - a literature review and a mapping study," *EURASIP Journal on Embedded Systems*, vol. 2013, no. 1, p. 15, Nov. 2013, ISSN: 1687-3963. DOI: 10.1186/1687-3963-2013-15. [Online]. Available: https://doi.org/10.1186/1687-3963-2013-15.

[15] P. Rodríguez, J. Partanen, P. Kuvaja, and M. Oivo, "Combining lean thinking and agile methods for software development a case study of a Finnish provider of wireless embedded systems," Jan. 2014, pp. 4770–4779. DOI: 10.1109/HICSS.2014.586.

[16] C. Huygens, D. Hughes, B. Lagaisse, and W. Joosen, "Streamlining development for networked embedded systems using multiple paradigms," *IEEE Software*, vol. 27, no. 5, pp. 45–52, Sep. 2010, ISSN: 1937-4194. DOI: 10.1109/MS.2010.93.

[17] L. E. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch, and M. Oivo, "Towards DevOps in the embedded systems domain: Why is it so hard?" In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, Jan. 2016, pp. 5437–5446. DOI: 10.1109/HICSS.2016.671.

[18] D. Robey, R. Welke, and D. Turk, "Traditional, iterative, and component-based development: A social analysis of software development paradigms," *Information Technology and Management*, vol. 2, no. 1, pp. 53–70, Jan. 2001, ISSN: 1573-7667. DOI: 10.1023/A:1009982704160. [Online]. Available: https://doi.org/10.1023/A:1009982704160.

[19] P. Bourque and R. E. Fairley, Eds., *SWEBOK: Guide to the Software Engineering Body of Knowledge*, Version 3.0. Los Alamitos, CA: IEEE Computer Society, 2014, ISBN: 978-0-7695-5166-1. [Online]. Available: http://www.swebok.org/.

[20] L. R. Vijayasarathy and C. W. Butler, "Choice of software development methodologies: Do organizational, project, and team characteristics matter?" *IEEE Software*, vol. 33, no. 5, pp. 86–94, Sep. 2016, ISSN: 0740-7459. DOI: 10.1109/MS.2015.26.

[21] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," *IEEE Software*, vol. 33, no. 3, pp. 94–100, May 2016, ISSN: 1937-4194. DOI: 10.1109/MS.2016.68.

[22] T. Rauber and G. Rünger, "Enabling scalability, adaptivity, and resilience in cloud applications by software-defined m-task-based programming," in *2019 Sixth International Conference on Software Defined Systems (SDS)*, Jun. 2019, pp. 88–95. DOI: 10.1109/SDS.2019.8768599.

[23] X. Wang, K. Conboy, and O. Cawley, "'Leagile' software development: An experience report analysis of the application of lean approaches in agile software development," *Journal of Systems and Software*, vol. 85, no. 6, pp. 1287–1299, 2012, Special Issue: Agile Development, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.01.061. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121212000404.

[24] L. McLeod and S. G. MacDonell, "Factors that affect software systems development project outcomes: A survey of research," *ACM Comput. Surv.*, vol. 43, no. 4, 24:1–24:56, Oct. 2011, ISSN: 0360-0300. DOI: 10.1145/1978802.1978803. [Online]. Available: http://doi.acm.org/10.1145/1978802.1978803.

[25] K. Crowston, Q. Li, K. Wei, U. Y. Eseryel, and J. Howison, "Self-organization of teams for free/libre open source software development," *Inf. Softw. Technol.*, vol. 49, no. 6, pp. 564–575, Jun. 2007, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2007.02.004. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2007.02.004.

[26] S. Faraj and L. Sproull, "Coordinating expertise in software development teams," *Management Science*, vol. 46, no. 12, pp. 1554–1568, 2000. DOI: 10.1287/mnsc.46.12.1554.12072. eprint: https://pubsonline.informs.org/doi/pdf/10.1287/mnsc.46.12.1554.12072. [Online]. Available: https://pubsonline.informs.org/doi/abs/10.1287/mnsc.46.12.1554.12072.

[27] J. D. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally distributed software development," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 481–494, Jun. 2003, ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1205177.

[28] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, *Manifesto for agile software development*, 2001. [Online]. Available: http://www.agilemanifesto.org/.

[29] I. Sommerville, *Software Engineering*, 9th. USA: Addison-Wesley Publishing Company, 2010, ISBN: 0137035152, 9780137035151.

[30] A. Cockburn and J. Highsmith, "Agile software development, the people factor," *Computer*, vol. 34, no. 11, pp. 131–133, Nov. 2001, ISSN: 0018-9162. DOI: 10.1109/2.963450.

[31] R. Vidgen and X. Wang, "Coevolving systems and the organization of agile software development," *Information Systems Research*, vol. 20, no. 3, pp. 355–376, 2009. DOI: 10.1287/isre.1090.0237. eprint: https://pubsonline.informs.org/doi/pdf/10.1287/isre.1090.0237. [Online]. Available: https://pubsonline.informs.org/doi/abs/10.1287/isre.1090.0237.

[32] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects," *Journal of Systems and Software*, vol. 81, no. 6, pp. 961–971, 2008, Agile Product Line Engineering, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2007.08.020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121207002208.

[33] D. Spinellis, "Version control systems," *IEEE Software*, vol. 22, no. 5, pp. 108–109, Sep. 2005, ISSN: 1937-4194. DOI: 10.1109/MS.2005.140.

[34] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "A fundamental approach to model versioning based on graph modifications: From theory to implementation," *Software & Systems Modeling*, vol. 13, no. 1, pp. 239–272, Feb. 2014, ISSN: 1619-1374. DOI: 10.1007/s10270-012-0248-x. [Online]. Available: https://doi.org/10.1007/s10270-012-0248-x.

[35] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2911732.

[36] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, Sep. 2017, ISSN: 2372-2568. DOI: 10.1109/MCC.2017.4250939.

[37] L. Zu, P. Wang, J. Han, F. Liu, Q. Ai, and Y. Ji, "A study on test platform for verifying new wireless networking technologies via hardware-in-loop simulation," in *2010 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2010, pp. 75–78.

[38] C. A. Valderrama, A. Changuel, P. V. Raghavan, M. Abid, T. Ben Ismail, and A. A. Jerraya, "A unified model for co-simulation and co-synthesis of mixed hardware/-software systems," in *Proceedings the European Design and Test Conference. ED TC 1995*, 1995, pp. 180–184.

[39] Y. Yang, J. Xu, G. Shi, and C.-X. Wang, *5G Wireless Systems: Simulation and Evaluation Techniques*. Springer International Publishing, 2018. DOI: 10.1007/978-3-319-61869-2. [Online]. Available: https://doi.org/10.1007/978-3-319-61869-2.

[40] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, California, USA, May 2003, pp. 14–27.

[41] J. Horneber and A. Hergenröder, "A survey on testbeds and experimentation environments for wireless sensor networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 4, pp. 1820–1838, 2014.

[42] A. Gavras, A. Karila, S. Fdida, M. May, and M. Potts, "Future internet research and experimentation: The fire initiative," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 3, pp. 89–92, 2007.

[43] T. Šolc, C. Fortuna, and M. Mohorčič, "Low-cost testbed development and its applications in cognitive radio prototyping," in *Cognitive radio and networking for heterogeneous wireless networks*, Springer, 2015, pp. 361–405.

[44] C. Fortuna, A. Bekan, T. Javornik, G. Cerar, and M. Mohorcic, "Software interfaces for control, optimization and update of 5G machine type communication networks," *Computer Networks*, vol. 129, pp. 373–383, 2017, Special Issue on 5G Wireless Networks for IoT and Body Sensors, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2017.06.015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128617302578.

[45] A. Gosain and I. Seskar, "Geni wireless testbed: An open edge ecosystem for ubiquitous computing applications," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, IEEE, 2017, pp. 54–56.

[46] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, Dec. 2014, ISSN: 1570-7873. DOI: 10.1007/s10723-014-9314-7. [Online]. Available: https://doi.org/10.1007/s10723-014-9314-7.

[47] Shu-Cheng Chang, C. Huang, and Jhih-Sin Lin, "Applying express-queue-based approach to software reliability and cost analysis," in *2016 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC)*, 2016, pp. 1–6.

[48] C. Huang and T. Kuo, "Queueing-theory-based models for software reliability analysis and management," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 540–550, 2017.

[49] H. H. Liu, *Software Performance and Scalability: A Quantitative Approach*. Wiley Publishing, 2009, ISBN: 0470462531.

[50] N. Schneidewind, "Software development queuing model," *Journal of Aerospace Computing, Information, and Communication*, vol. 7, no. 10, pp. 308–321, 2010. DOI: 10.2514/1.40133. eprint: https://doi.org/10.2514/1.40133. [Online]. Available: https://doi.org/10.2514/1.40133.

[51] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, Oct. 2016, ISSN: 1573-7616. DOI: 10.1007/s10664-015-9393-5. [Online]. Available: https://doi.org/10.1007/s10664-015-9393-5.

[52] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki, "GitHub projects. quality analysis of open-source software," in *Social Informatics: 6th International Conference, SocInfo 2014, Barcelona, Spain, November 11-13, 2014. Proceedings*, L. M. Aiello and D. McFarland, Eds. Cham: Springer International Publishing, 2014, pp. 80–94, ISBN: 978-3-319-13734-6. DOI: 10.1007/978-3-319-13734-6_6. [Online]. Available: https://doi.org/10.1007/978-3-319-13734-6_6.

[53] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2017, pp. 60–71. DOI: 10.1109/ASE.2017.8115619.

[54] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl, "Security developer studies with GitHub users: Exploring a convenience sample," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, Santa Clara, CA: USENIX Association, 2017, pp. 81–95, ISBN: 978-1-931971-39-3. [Online]. Available: https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar.

[55] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec. 2017, ISSN: 1573-7616. DOI: 10.1007/s10664-017-9512-6. [Online]. Available: https://doi.org/10.1007/s10664-017-9512-6.

[56] Y. Xiong, Z. Meng, B. Shen, and W. Yin, "Developer identity linkage and behavior mining across GitHub and StackOverflow," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 09n10, pp. 1409–1425, 2017. DOI: 10.1142/S0218194017400034. [Online]. Available: https://doi.org/10.1142/S0218194017400034.

[57] F. Chatziasimidis and I. Stamelos, "Data collection and analysis of GitHub repositories and users," in *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*, Jul. 2015, pp. 1–6. DOI: 10.1109/IISA.2015.7388026.

[58] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, Jun. 2012, pp. 12–21. DOI: 10.1109/MSR.2012.6224294.

[59] P. Devanbu, P. Kudigrama, C. Rubio-González, and B. Vasilescu, "Timezone and time-of-day variance in GitHub teams: An empirical method and study," in *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics*, ser. SWAN 2017, Paderborn, Germany: ACM, 2017, pp. 19–22, ISBN: 978-1-4503-5157-7. DOI: 10.1145/3121257.3121261. [Online]. Available: http://doi.acm.org/10.1145/3121257.3121261.

[60] C. Kolassa, D. Riehle, and M. A. Salim, "The empirical commit frequency distribution of open source projects," in *Proceedings of the 9th International Symposium on Open Collaboration*, ser. WikiSym '13, Hong Kong, China: ACM, 2013, 18:1–18:8, ISBN: 978-1-4503-1852-5. DOI: 10.1145/2491055.2491073. [Online]. Available: http://doi.acm.org/10.1145/2491055.2491073.

[61] J. H. Bernardo, D. Alencar da Costa, and U. Kulesza, "Studying the impact of adopting continuous integration on the delivery time of pull requests," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 131–141.

[62] J. Xia and Y. Li, "Could we predict the result of a continuous integration build? an empirical study," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2017, pp. 311–315. DOI: 10.1109/QRS-C.2017.59.

[63] O. Beaumont, N. Bonichon, L. Courtès, E. Dolstra, and X. Hanin, "Mixed data-parallel scheduling for distributed continuous integration," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 91–98. DOI: 10.1109/IPDPSW.2012.7.

[64] S. Baltes, J. Knack, D. Anastasiou, R. Tymann, and S. Diehl, "(no) influence of continuous integration on the commit activity in GitHub projects," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, ser. SWAN 2018, Lake Buena Vista, FL, USA: ACM, 2018, pp. 1–7, ISBN: 978-1-4503-6056-2. DOI: 10.1145/3278142.3278143. [Online]. Available: http://doi.acm.org/10.1145/3278142.3278143.

[65] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including performance benchmarks into continuous integration to enable DevOps," *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 2, pp. 1–4, Apr. 2015, ISSN: 0163-5948. DOI: 10.1145/2735399.2735416. [Online]. Available: http://doi.acm.org/10.1145/2735399.2735416.

[66] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2685629.

[67] G. Gousios, "The GHTorent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 233–236, ISBN: 978-1-4673-2936-1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132.

[68] M. Vučnik, T. Šolc, U. Gregorc, A. Hrovat, K. Bregar, M. Smolnikar, M. Mohorčič, and C. Fortuna, "Continuous integration in wireless technology development," *IEEE Communications Magazine*, vol. 56, no. 12, pp. 74–81, Dec. 2018, ISSN: 0163-6804. DOI: 10.1109/MCOM.2018.1800107.

[69] T. J. W. I. R. Center and R. Chillarege, *Software Testing Best Practices*, ser. Research report. IBM T.J. Watson Research Center, 1999. [Online]. Available: https://books.google.si/books?id=ooCBHAAACAAJ.

[70] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2429–2453, thirdquarter 2018, ISSN: 1553-877X. DOI: 10.1109/COMST.2018.2815638.

[71] M. Meyer, "Continuous integration and its tools," *IEEE software*, vol. 31, no. 3, pp. 14–16, 2014.

[72] B. Vasilescu *et al.*, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015, pp. 805–816.

[73] J. Pinola, J. Perälä, P. Jurmu, M. Katz, S. Salonen, J. Piisilä, J. Sankala, and P. Tuuttila, "A systematic and flexible approach for testing future mobile networks by exploiting a wrap-around testing methodology," *IEEE Communications Magazine*, vol. 51, no. 3, pp. 160–167, Mar. 2013, ISSN: 1558-1896. DOI: 10.1109/MCOM.2013.6476882.

[74] M. Vučnik, C. Fortuna, T. Šolc, and M. Mohorčič, "Integrating research testbeds into social coding platforms," in *2018 European Conference on Networks and Communications (EuCNC)*, 2018, pp. 230–234. DOI: 10.1109/EuCNC.2018.8443242.

[75] J. S. C. Xiao, S. K. V. Ragavan, and M. Shanmugavel, "OSGi-based, embedded, distributed, telematics framework for flexible service provisioning in cyber-physical production systems," in *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, Dec. 2016, pp. 1–5. DOI: 10.1109/ICCIC.2016.7919566.

[76] M. Vögler, J. Schleicher, C. Inzinger, S. Nastic, S. Sehic, and S. Dustdar, "LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments," in *2015 IEEE Symposium on Service-Oriented System Engineering*, Mar. 2015, pp. 78–87. DOI: 10.1109/SOSE.2015.23.

[77] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "OMF: A control and management framework for networking testbeds," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 54–59, Jan. 2010, ISSN: 0163-5980. DOI: 10.1145/1713254.1713267. [Online]. Available: https://doi.org/10.1145/1713254.1713267.

[78] D. Stavropoulos, A. Dadoukis, T. Rakotoarivelo, M. Ott, T. Korakis, and L. Tassiulas, "Design, architecture and implementation of a resource discovery, reservation and provisioning framework for testbeds," in *2015 13th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*, May 2015, pp. 48–53. DOI: 10.1109/WIOPT.2015.7151032.

[79] T. Fischer, T. Huehn, R. Kuck, R. Merz, J. Schulz-Zander, and C. Sengul, "Experiences with BOWL: Managing an outdoor wifi network (or how to keep both internet users and researchers happy?)" In *Proceedings of the 25th International Conference on Large Installation System Administration*, ser. LISA'11, Boston, MA: USENIX Association, 2011, pp. 24–24.

[80] T. Buchert, C. Ruiz, L. Nussbaum, and O. Richard, "A survey of general-purpose experiment management tools for distributed systems," *Future Generation Computer Systems*, vol. 45, pp. 1–12, 2015, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2014.10.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X14002003.

[81] D. Stahl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, no. Supplement C, pp. 48–59, 2014, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.08.032.

[82] M. Vučnik, M. Mohorčič, and C. Fortuna, "Developer activity modeling: Implications on wireless embedded testing," *IEEE Internet of Things Journal*, under review.

[83] H. Takagi and B. Walke, *Spectrum Requirement Planning in Wireless Communications: Model and Methodology for IMT - Advanced*, ser. Wireless Communications and Mobile Computing. Wiley, 2008, ISBN: 9780470758953.

[84] V. B. Iversen, *Teletraffic engineering and network planning*. DTU Fotonik, 2015.

[85] O. Ibe, *Fundamentals of Stochastic Networks*. Wiley, 2011, ISBN: 9781118092989.

[86] E. Chromy, J. Diezka, M. Kavacky, and M. Voznak, "Markov models and their use for calculations of important traffic parameters of contact center," *WTOC*, vol. 10, no. 11, pp. 341–350, Nov. 2011, ISSN: 1109-2742.

[87] I. Moscholios and M. Logothetis, *Efficient Multirate Teletraffic Loss Models Beyond Erlang*, ser. Wiley - IEEE. Wiley, 2019, ISBN: 9781119426882.

[88] G. Kesidis, "Introduction to queueing theory," in *An Introduction to Communication Network Analysis*. IEEE, 2007, pp. 83–110, ISBN: null. DOI: 10.1002/9780470168684.ch3. [Online]. Available: https://ieeexplore.ieee.org/document/5237294.

[89] G. Bolch, S. Greiner, H. d. Meer, and K. S. Trivedi, *Queueing Networks and Markov Chains*. New York, NY, USA: Wiley-Interscience, 2006, ISBN: 0471565253.

[90] E. Chromy, T. Misuth, and M. Kavacky, "Erlang C formula and its use in the call centers," *Advances in Electrical and Electronic Engineering*, vol. 9, no. 1, 2011, ISSN: 1804-3119. [Online]. Available: http://advances.utc.sk/index.php/AEEE/article/view/34.

[91] W. C. Chan, *Performance Analysis of Telecommunications and Local Area Networks*, ser. The Springer International Series in Engineering and Computer Science. Springer US, 2002.

[92] L. C and S. A. Iyer, "Application of queueing theory in health care: A literature review," *Operations Research for Health Care*, vol. 2, no. 1, pp. 25–39, 2013, ISSN: 2211-6923. DOI: https://doi.org/10.1016/j.orhc.2013.03.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2211692313000039.

[93] J. D. C. Little, "A proof for the queuing formula: $L = \lambda W$," *Operations Research*, vol. 9, no. 3, pp. 383–387, 1961. DOI: 10.1287/opre.9.3.383. eprint: https://doi.org/10.1287/opre.9.3.383. [Online]. Available: https://doi.org/10.1287/opre.9.3.383.

[94] G. Giambene, *Queuing Theory and Telecommunications: Networks and Applications*. Springer, 2014, ISBN: 9781461440857.

[95] W. .-. Chan and Y. .-. Lin, "Waiting time distribution for the M/M/m queue," *IEE Proceedings - Communications*, vol. 150, no. 3, pp. 159–162, 2003.

[96] C. Treude, F. Figueira Filho, and U. Kulesza, "Summarizing and measuring development activity," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, 2015, pp. 625–636.

[97] L. A. Kurgan and P. Musilek, "A survey of knowledge discovery and data mining process models," *The Knowledge Engineering Review*, vol. 21, no. 1, pp. 1–24, 2006. DOI: 10.1017/S0269888906000737.

[98] M. A. C. Bouzada, "Dimensioning a call center: Simulation or queue theory?" *Journal of Operations and Supply Chain Management*, vol. 2, pp. 34–46, 2009.

[99] J. F. C. Kingman, "The first Erlang century—and the next," *Queueing Systems*, vol. 63, no. 1, p. 3, Nov. 2009, ISSN: 1572-9443. DOI: 10.1007/s11134-009-9147-4. [Online]. Available: https://doi.org/10.1007/s11134-009-9147-4.

[100] T. Mišuth, E. Chromý, and I. Baronák, "Method for fast estimation of contact centre parameters using Erlang C model," in *2010 Third International Conference on Communication Theory, Reliability, and Quality of Service*, Jun. 2010, pp. 181–185. DOI: 10.1109/CTRQ.2010.38.

[101] S. Qiao and L. Qiao, "A robust and efficient algorithm for evaluating Erlang B formula," Tech. Rep., 1998.

# Bibliography

## Publications Related to the Thesis

## Original scientific articles

1. M. Vučnik, T. Šolc, U. Gregorc, A. Hrovat, K. Bregar, M. Smolnikar, M. Mohorčič, and C. Fortuna, "Continuous Integration in Wireless Technology Development," *IEEE Communications Magazine*, vol. 56, no. 12, pp. 74-81, 2018, ISSN: 1558-1896, DOI: 10.1109/MCOM.2018.1800107.

2. M. Pesko, M. Smolnikar, M. Vučnik, T. Javornik, M. Pejanović-Djurišić, and M. Mohorčič, "Smartphone with Augmented Gateway Functionality as Opportunistic WSN Gateway Device," *Wireless Personal Communications*, vol. 78, no. 3, pp. 1811-1826, Oct. 2014, ISSN: 0929-6212, DOI: 10.1007/s11277-014-1908-7.

3. M. Vučnik, M. Mohorčič, and C. Fortuna, "Developer Activity Modeling: Implications on Wireless Embedded Testing," *IEEE Internet of Things Journal*, 2020, under review.

## Published scientific conference contribution

1. I. Boškov, H. Yetgin, M. Vučnik, C. Fortuna, and M. Mohorčič, "Time-to-Provision Evaluation of IoT Devices Using Automated Zero-Touch Provisioning," *2020 IEEE Global Communications Conference (GLOBECOM)*, Taipei, Taiwan, 2020.

2. D. Rivera, E. M. D. Oca, W. Mallouli, A. R. Cavalli, B. Vermeulen, and M. Vučnik, "Industrial IoT Security Monitoring and Test on Fed4Fire+ Platforms," *Testing software and systems : 31st IFIP WG 6.1 International Conference, ICTSS 2019*, Paris, France, 2019, pp. 270-278, 2019, ISBN: 978-3-030-31279-4, DOI: 10.1007/978-3-030-31280-0_17.

3. M. Vučnik, A. Švigelj, G. Kandus and M. Mohorčič, "Secure Hybrid Publish-Subscribe Messaging Architecture," *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Split, Croatia, 2019, ISSN: 1847-358X, DOI: 10.23919/SOFTCOM.2019.8903868.

4. M. Vučnik, C. Fortuna, T. Šolc and M. Mohorčič, "Integrating Research Testbeds into Social Coding Platforms," *2018 European Conference on Networks and Communications (EuCNC)*, Ljubljana, Slovenia, 2018, pp. 230-234, ISSN: 2575-4912, DOI: 10.1109/EuCNC.2018.8443242.

# Other Publications

## Book chapter contribution

1. M. Vučnik, "AMI-Based Continuous Power Quality Assessment System". In J. N. J. Davies and C. Fortuna, *The Internet of Things: from data to insight.* Hoboken, NJ: Wiley, 2020, ISBN: 978-1-119-54526-2.

## Published scientific conference contribution

1. T. Javornik, A. Hrovat, A. Vilhar, M. Vučnik, I. Ozimek and M. Pesko, "Radio environment map (REM): An approach for provision wireless communications in disaster areas," *2014 1st International Workshop on Cognitive Cellular Systems (CCS)*, Germany, 2014, DOI: 10.1109/CCS.2014.6933796.

2. C. Fortuna, M. Vučnik, B. Fortuna, K. Kenda, A. Moraru and D. Mladenić, "Towards Building a Global Oracle: A Physical Mashup Using Artificial Intelligence Technology," *Proceedings of the Third International Workshop on the Web of Things*, Newcastle, United Kingdom, 2012, ISBN: 9781450316033, DOI: 10.1145/2379756.2379763.

3. A. Moraru, D. Mladenić, M. Vučnik, M. Porcius, C. Fortuna, and M. Mohorčič, "Exposing real world information for the web of things," *8th International Workshop on Information Integration on the Web: in conjunction with WWW 2011 (IIWeb '11)*, Hyderabad, India, 2011, DOI: 10.1145/1982624.1982630.

# Biography

Matevž Vučnik received his B.Sc. in electrical engineering from the University of Ljubljana, Faculty of Electrical Engineering at the telecommunications department in 2010. He joined the Department of Communication Systems at the Jožef Stefan Institute (JSI) the same year as a research assistant. In 2010, he also enrolled in a PhD study at the Jožef Stefan International Postgraduate School study programme Information and Communication Technologies. He currently has a position of a senior researcher at the Department for Communication Systems at the JSI.

During the years working at the JSI he has participated in several national and European research and development projects, where he has been involved in proposal and deliverable writing, yet having the most important contributions in applied tasks. He has been responsible for design and implementation of various software components and systems required for successful projects piloting and realization. His main research interest is thus concerned with the design and development of practical distributed systems and internet protocols.

The author started his work at the JSI by designing and implementing scalable databases for sensor data collection and distributed communication protocols for wireless embedded systems. He has developed in-depth knowledge of internet communication protocols and has designed several communication systems specifically for static as well as mobile wireless sensor networks. He has practical knowledge of the Linux operating system and has designed several embedded devices running embedded Linux. He has practical experience with the Linux kernel driver development and has contributed to Linux kernel project by extending the driver to support a new LTE module. Furthermore, he has developed several web applications, designed and implemented several APIs for sensor data management, and released them under free software licenses. He was one of the first researchers using popular container technology Docker in the field of Internet of Things (IoT). He introduced Docker to IoT to enable ad hoc experimentation with the distributed system of synchro-phasor measurement devices designed and implemented for a successful FP7 project Sunseed. For the same project he designed a framework and implemented a modular, web-based distributed system management platform which is currently being used in multiple projects.

He has been involved in several FP7, H2020 and national projects (given below in chronological order) with the following responsibilities:

- FP7 PlanetData – Designed and implemented a scalable big data SQL database for sensor data.
- National Competence Center Opcomm – Designed and implemented an Internet protocol-based pipeline for sensor data.
- FP7 CREW – Contributed to establishing an open federated test platform for LOG-a-TEC testbed developed at JSI.

- FP7 CITI-SENSE – Designed and implemented a database for mobile air quality sensor data, web-based APIs, data proxy service and frontend for data acquisition.
- FP7 Fed4FIRE – Contributed to adapting LOG-a-TEC testbed experimentation procedures to enable full integration with the Fed4FIRE federation of testbeds.
- National project OSU – Contributed to proposal writing; designed and implemented a teaching web platform to simplify the development of air quality and other sensor applications.
- FP7 ABSOLUTE – Designed and implemented a database for Radio Environment Map (REM) and APIs for mobile cell transmit power calculation.
- FP7 SUNSEED – Designed and implemented distributed Internet protocol-based data pipeline for time critical synchro-phasor device, supported safe ad hoc experimentation and implemented a centralized management system for distributed measurement testbed.
- H2020 Fed4FIRE+ – Supporting the experimentation using the LOG-a-TEC testbed and upgraded the testbed using contemporary IoT and container technologies based on Linux OS.
- H2020 WiSHFUL – Implemented Unified Programming Interfaces (UPIs) on LOG-a-TEC wireless sensor nodes.
- H2020 SAAM – Serving as Task leader for System Technical Specifications, User-Sided IT Infrastructure and System Component Verification.
- H2020 DEFENDER – Implementing a remote management system for PMU testbed and designing a data transfer pipeline for centralized fault processing.
- H2020 NRG-5 – Implementing a distributed data processing pipeline for smart energy-based use cases using IoT sensor data.

The author has published several peer reviewed journal and conference papers. The result of his work on distributed IoT system management and automation resulted in a publication in a highly respectable magazine IEEE Communications Magazine, ranked 2nd in the field of Telecommunications.