TAMPEREEN
AMMATTIKORKEAKOULU

3G NETWORK VISUALIZATION WITH SIGNALING MEASUREMENTS

David Spalding

Bachelor's Thesis
October 2018
Information Technology
Software Engineering

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikka
Ohjelmistotuotanto

DAVID SPALDING:
3G network visualization with signaling measurements

Opinnäytetyö 72 sivua
Lokakuu 2018

Lopputyö dokumentoi projektia, jonka aiheena on 3G-verkkojen signalointimittaukset. Tätä dataa on kerätty Nokian L3 Data Collector -ohjelmalla, ja kerättyä dataa on parsittu Nokian L3 Data Analyzer -ohjelmalla CSV-tiedostoiksi. Parsitusta datasta tarkastelemme murto-osaa, joka keskittyy GPS-palvelun heijastamaa osuutta. Tästä datasta, josta löytyy koordinaatiodataa, päätelaitteen mittaamaa signaalin vahvuutta, ja solun käyttämää pääsykoodia, rakennamme käyttötapauksen, jossa visualisoidaan signaalivahvuuksia ja solujen aluedominanssia.

Prosessi alkaa raakadatan työntämisestä tietokantatauluun, josta se rikastetaan toiseen tauluun. Rikastusprosessi summaa datan päivätasolle, ja samalla muuntaa mittaukset siten, että niitä tarkastellaan aluemittauksina, eikä pistemittauksina. Aluemuunnoksissa käytetään hyväksi konseptia, jossa aluekoordinaateista muodostetaan maantieteellinen tarkistussumma.

Datan analysointia varten on valjastettu ohjelmointikieli Python ja sen data-analysointi -kirjasto Pandas. Pandasia käytetään datan esikäsittelyssä sekä sen rikastamisessa. Rikastamisprosessin jälkeen kokonaissummattu data muunnetaan GeoJSON-tiedostoiksi, joihin sisältyy oleellinen alueinformaatio. Näitä tiedostoja hyödynnetään datan visualisoinnissa, jotka on toteutettu kahdella eri ohjelmointikehyksellä: Plotly ja deck.gl.

Tässä tekstissä käydään empiirisen osuuden lisäksi myös teoriaa 3G-verkoista, keskittyen 3G-verkkojen rakennuspalikoihin ja keinoihin, jolla verkko saadaan optimoitua. Lisäksi tarkastelemme verkkojen toimintaa teleoperaattorin näkökulmasta.

Tekstissä esitetty projekti on prototyyppi, ja se ei vastaa tulevan tuotantokelpoisen toteutuksen tasoa tai itse toteutusta. Projektin idea ja sen alla oleva data on saatu teleoperaattori Elisa Oyj:ltä.

Käyttötapausta voidaan käyttää mobiiliverkon reaaliaikaiseen valvontaan. Verkkosuunnittelijat ja -optimoijat saavat paremman kuvan verkon laadusta, kun tarjolla on reaaliaikaista dataa eri alueista. Valvonnan lisäksi käyttötapaus auttaa myös päätöksenteossa ja tulosten varmistamisessa.

Asiasanat: python, 3g, pandas, plotly, signalointi, geohash, visualisaatio

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Information Technology
Software Engineering

DAVID SPALDING:
3G network visualization with signaling measurements

Bachelor's thesis 72 pages
October 2018

This work centers around 3G mobile network signaling measurements collected by Nokia’s L3 Data Collector, which are further parsed to CSV files by Nokia’s L3 Data Analyzer. From these measurements we take a closer look at a fraction of the provided data. These measurements specifically are measurements reflecting the GPS service. Data gathered from the measurements contain coordinate data (latitude, longitude), signal strength in the UE, and primary scrambling code used by the active cell. From this data forms a use-case, in which we visualize signal strength quality and cell area dominance on map, with a fixed square grid from areas where measurements are available from.

Initially the wanted data is pushed onto a database table, where from the data is straight away enriched to the next table. The data enrichment process aggregates the raw data to a day level, and also transforms the measurements so that they are viewed as area measurements, instead of precise location measurements. This area aggregation is done by utilizing geohashing.

For data analysis purposes, Python and Pandas play a heavy role. Pandas is used for preprocessing and enriching the raw data. Python and its many libraries are used for orchestrating the whole process. Once data is enriched, it is converted into GeoJSON files, which contain area information aiding the visualization. Data visualization is implemented with two different frameworks: Plotly and deck.gl.

Before getting to the practical part and the actual implementation of this documentation, the reader is introduced the basics of 3G mobile networks, which includes going through different 3G network components and their methods for running the network smoothly. Things are also looked from the view of a network operator.

The project documented here is a proof-of-concept, and does not reflect the final production implementation. The project concept and the data for it was provided by Finnish mobile operator Elisa Oyj.

The use-case can mainly be used for real-time monitoring of a mobile network. When real-time data is offered for network planners and optimizers, troubleshooting and decision making is much more reliable. The observed changes will also aid with validating results.

Keywords: python, 3g, pandas, plotly, signaling, geohash, visualization

CONTENTS

GLOSSARY

3GPP	3G Partnership Project
API	Application Programming Interface
ASU	Active Set Update
BS	Base Station
CM	Configuration Management
CN	Core Network
CRNC	Controlling Radio Network Controller
DCH	Dedicated Channel
DRNC	Drifting Radio Network Controller
ETSI	European Telecommunications Institute
FACH	Forward Access Channel
FM	Fault Management
GSM	Global System for Mobile communications
HLR	Home Location Register
IMT-2000	International Mobile Telephony 2000
IRAT	Inter-radio Access Technology
ITIL	Information Technology Infrastructure Library
JSON	JavaScript Object Notation
KPI	Key Performance Index
LA	Location Area
MDT	Minimization of Drive Tests
NAS	Non-Access Stratum
NBAP	Node B Application Protocol
NMT	Nordic Mobile Telephone
PCH	Physical Channel
PDCP	Packet Data Convergence Protocol
PEP	Python Enhancement Proposal
PM	Performance Management
PoC	Proof-of-Concept
QoS	Quality of Service
RAB	Radio Access Bearer
RAN	Radio Access Network
RFC	Request For Comments

RLC	Radio Link Control
RNC	Radio Network Controller
RNS	Radio Network Subsystem
RRC	Radio Resource Control
RRM	Radio Resource Management
RRU	Radio Resource Utilization
RSCP	Received Signal Code Power
SC	Scrambling Code
SCM	Source Code Management
SDK	Standard Development Kit
SIM	Subscriber Identity Module
SRNC	Serving Radio Network Controller
SON	Self-Organizing Network
SQL	Structured Query Language
TMN	Telecommunications Management Network
UE	User Equipment
UMTS	Universal Mobile Telecommunication System
URA	UTRAN Registration Area
UTRAN	UMTS Terrestrial Radio Access Network
VLR	Visitor Location Register
WCDMA	Wideband Code Division Multiple Access
WSL	Windows Subsystem for Linux

1 INTRODUCTION

At the time of this writing, 5G (or 5th generation mobile technologies) has just quite recently entered the market, and talks and planning about 6G is already stirring loudly. This is a very fitting metaphor to the nature of mobile technologies which are constantly evolving and changing into something new. Despite the ever-changing nature, older generation of mobile technologies have not been evaporated by their newer counterparts. In fact, older technology generations are still being researched and developed in parallel with newer technologies, as they are still widely utilized to this day. A perfect example would be 3G, which is still utilized dominantly alongside with 4G, as 5G is still mainly waiting behind the curtains for its grand utilization.

While new upcoming technologies sit patiently behind the staging area, older technologies are being developed further through innovations. A common thing, on which developers can invest on, is optimization. In practice means investing on several processes which analyze the technology’s behavior. Once we are aware of its behavior on a deep enough level, we can plan its use in such an optimal way that we are able to squeeze all of its juice. This brings on the age old question of optimizing hardware versus upgrading hardware, which can also be interpreted as long-term investment versus short-term investment. With the pace of technology development, it is much more feasible to invest time and money in optimizing older hardware than constantly upgrade equipment with newer hardware, while the older expensive hardware rots as unusable.

All of the above holds very true also for mobile technologies and all network operators are constantly looking out for new innovations on how to get more juice out of their network equipment. Lucky for network operators as there is a whole business on enabling operators to monitor their networks very closely for performance. Offered are many types of data management models, which include configuration management and performance management to name two. Through observing data which these data management models produce, operators are able to perform resource management much more precisely on their networks and they can be even automated. Furthermore it’s possible to go even deeper with more low-level data, which enables deeper performance analysis and advanced network planning.

In this thesis I will be documenting my work with a use-case, which involved visualizing 3G network coverage and cell dominance areas. The base data for this work has been provided by a Nokia made software named MegaMon, which collects radio access network signaling measurements and passes them to a different software, which parses them into human readable CSV files. Signaling measurements are a type of data, which fall into the aforementioned category of more low-level data. The use of signaling measurements is a good method of validating network performance level, as trace data gathers statistics from signal data between user equipment and network elements. Within the scope of this work, a proof-of-concept was produced, and I will be specifically documenting the technology stack used to realize it. In the last chapter I will be pondering on what could be built from or on top of this PoC. This use-case work has been fulfilled and realized for the Finnish mobile network operator Elisa.

First chapter being the introduction for this work, on the second chapter we will take a brief look at 3G network technologies. The reader is introduced to the history and architecture of UMTS networks. The third chapter will delve into the role of a network operator and its position in the grand scheme of mobile networks. The aforementioned data management models will also be opened more to the reader. The fourth chapter expands more on telemobile management by introducing the concepts of performance monitoring and optimization. The reader is introduced on how performance is measured and what exactly is measured.

The fifth chapter opens up the second segment of this documentation. The previous chapters being an insight into mobile network technologies and management handling, the fifth chapter presents the use-case which is the actual project done for this thesis. Besides the project introduction, we will also take an extensive look at the different software technologies and tools used for this implementation. The sixth chapter contains the practical part of this document, meaning that the project implementation will be presented by looking at different figures, charts, and snippets of the actual code. The seventh and last chapter concludes this text with thoughts on what else could be accomplished with utilizing signaling measurements, and how this particular use-case could be advanced further.

2 3G NETWORKS

The history of mobile networks is a long and very eventful one. In the early days of NMT, or 1G as it's more popularly known as, mobile networks and telephones were somewhat niche products among tech enthusiasts. NMT networks were riddled with incompatibility issues and everything was experimental as no international standards existed back then. We have come a very long way from those days and in this chapter, we are going to take a deeper look at the 3rd generation of mobile communication, which back in the day opened the way for a global solution of mobile accessibility. (Kaaranen, et al. 2005, 3.)

2.1 3G specifications

As stated previously, back in the day NMT networks were problematic because of non-existent international standards. This caused incompatibilities with other networks, so network operation was mainly a nationwide service. On start of the 2nd generation of mobile communication technologies international bodies cooperated to specify a system, which would have an emphasis on compatibility on an international scale. Though a fully global solution was not achieved, the GSM standard was a big success both technical and commercial wise. (Kaaranen, et al. 2005, 5.)

The 3GPP collaboration was initially founded to further aid attaining global mobile communication system on the 3rd generation of mobile network technologies. Later this collaboration project has been extended develop and maintain technologies prior and beyond 3G (3G Partnership Project 2007). In many cases the term 3G covers regional terms like UMTS and IMT-2000. The European term UMTS describes 3G from the perspective of ETSI. IMT-2000 are more commonly used in Japan and the USA. Within 3GPP it has been decided that the official name of 3G is "3GPP System", however UMTS and IMT-2000 are still widely recognized and used. (Kaaranen, et al. 2005, 5.)

The first UMTS related 3GPP release was 3GPP Release 99. This release introduced the radio access method WCDMA (Wideband Code Division Multiple Access). Other key features included interoperability between 3G and 2G technologies. The next releases 3GPP Release 4 and Release 5 aimed to grow the technology stack so that multimedia content transfers became available for mobile users, and backwards compatibility was extended towards 2G. (Kaaranen, et al. 2005, 21-27.)

With 3GPP Release 10 and Release 11 MDT (Minimization of Drive Tests) functionality and support was added (Turkka 2014, 3). MDT is a vital element of this work and it will be discussed in more detail on section 4.5.

2.2 3G network architecture

When approaching telecommunication networks, the first approach should be by introducing two high-level layers. Namely the access stratum and the non-access stratum (NAS). The access strata can be defined as a group of protocols which enable the user the user to access the telecommunications network itself. To elaborate further, this stratum can be divided to two domains: the UE (User Equipment) domain and the access network domain. The UE domain contains the user, the mobile phone equipment belonging to the user and the SIM (Subscriber Identity Module) card within the mobile equipment which enables access to the access network domain. This latter domain's components are each different depending on the type of the telecommunications network. Later we will further examine an access domain network from the perspective of a UMTS network. The non-access stratum then groups protocols which handles communications between UE and the CN (Core Network). This layer mostly comprises of the core network domain where majority of the transmission functions happen. (Kaaranen, et al. 2005, 6–7.)

The grand idea behind this encapsulated layer structure is that this way single components can be swapped out and replaced. As technology rapidly evolves, the transfer mediums are under constant development and change. This way we can ensure that updates can be implemented in modular fashion, i.e. single modules can be swapped out without changing the whole network.

2.3 UMTS components

In terms of UMTS networks, the previously mentioned access network domain consists of UTRAN (UMTS Terrestrial Radio Access Network). The two main components in UTRAN are base stations (BS) and radio network controllers (RNC). A set of one RNC and multiple BSs can be further encapsulated as a RNS (Radio Network Subsystem). UTRAN usually consists of multiple RNSs. In this setting a base station (also known as Node B) is a component which is communicating between UE and RNC. A RNC is then the link towards the CN. (Kaaranen, et al. 2005, 99–101.)

2.4 Communication interfaces

The aforementioned "links" can be more precisely described as interfaces between UT-RAN architecture components (Figure 1). These interfaces secure the data flow between different components and each component connection uses a different kind of interface. We will not delve into technical specifics of these interfaces, as it is not the scope of this description, but rather just name them.

The interface between UE and BS is named as Uu. This enables user access to the access network. From the BS there is connectivity to the RNC. This interface is named as Iub. Towards the CN from the RNC the next connecting interface is named Iu. (Kaaranen, et al. 2005, 99–101.) There is also a fourth connection interface called Iur. This enables connectivity from RNC to RNC, or from RNS to RNS to be more specific. This connectivity allows multiple RNCs to manage RRM (Radio Resource Management), which enables better stability over the radio path. (Kaaranen, et al. 2005, 101.)

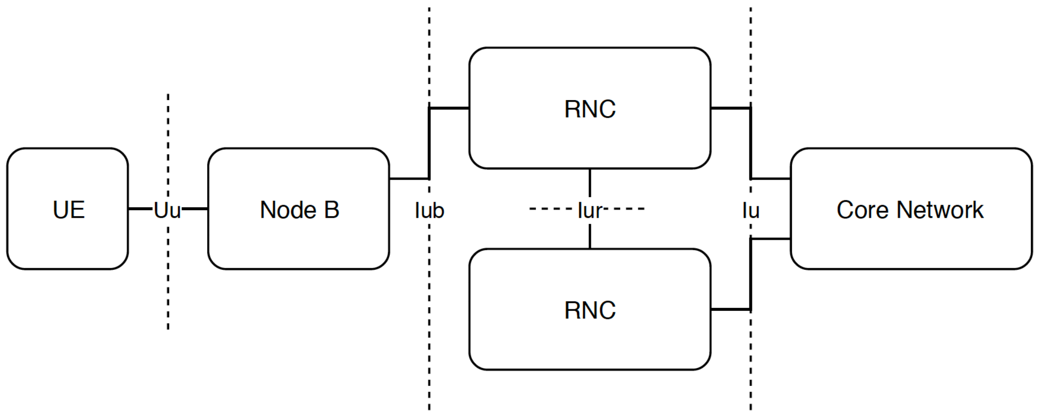


Figure 1. 3G architecture, with common components and their connecting interfaces.

2.5 RNC roles

RNCs have different roles depending on the context of how it is interacting with the network. Simply put, a RNC which is controlling the allocation of radio resources, is simply called a controlling RNC (CRNC). RNCs which allocate resources within the network for an UE are called serving RNCs (SRNC). There is also a third role for those more special occasions when a SRNC is already serving an UE, but the UE is moving towards a different coverage area, under a different RNC. In these cases the SRNC requests (through the Iur interface) the other RNC to allocate the UE context to itself. At this point the other RNC takes the role of a drifting RNC (DRNC). (Kaaranen, et al 2005, 111.)

2.6 Cells

The main function of a Node B is to offer radio coverage for mobile subscribers. Base station placement planning has to be done with a purpose, as UE reception deteriorates when the distance between UE and base station increases. A base stations resources are also limited, meaning that it is able to offer only so much radio links for subscribers, whom wish to use the network. These problems are resolved with the cellular concept, which fundamentally divides a large area into smaller sub-areas or cells. Each cell has its own base station and its own capacity, which to offer for subscribers (Figure 2). (Kaaranen, et al 2005, 37.)

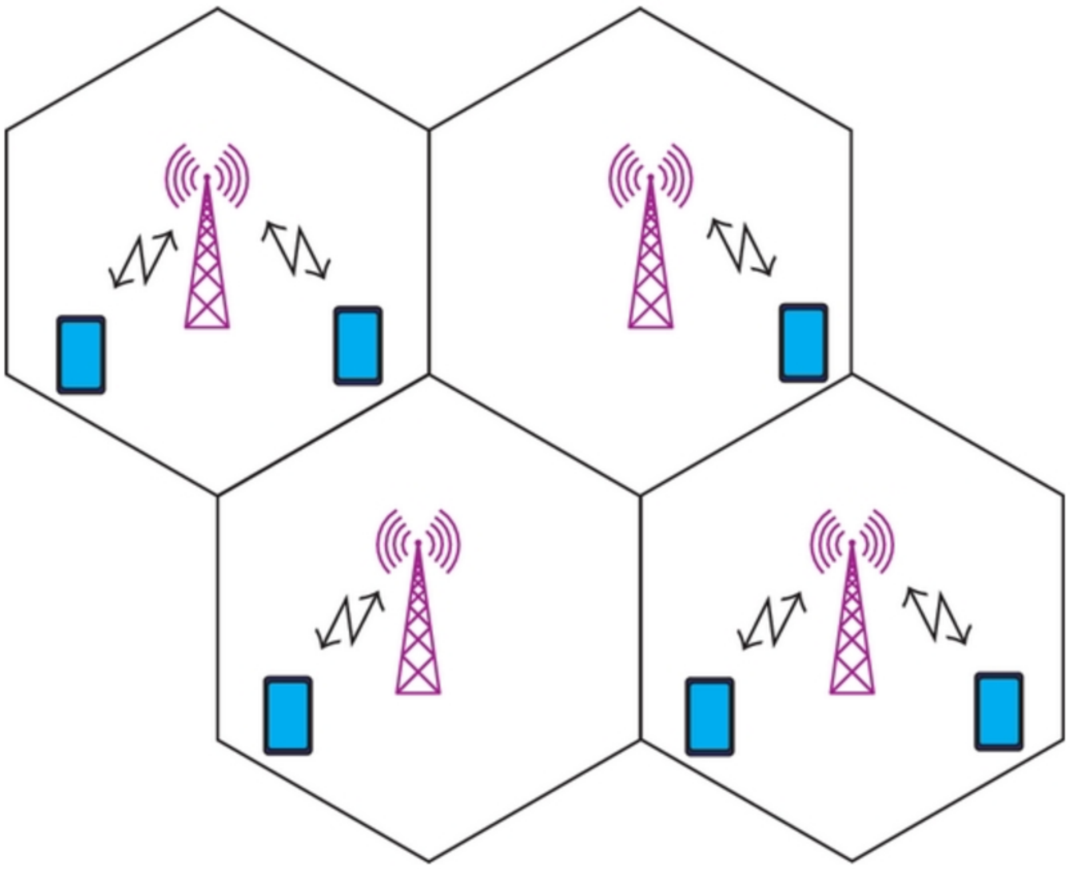


Figure 2. An area divided into cells with each their respective base station.

While the cellular concept solves some problems, it also brings new ones. The two main problems are interference from neighboring cells, and problems with subscriber mobility. Interference is mainly caused when overlapping areas of two adjacent cells use the same frequencies for radio signals. This kind of interference can be though easily alleviated with the use of different frequencies per cell area. By increasing the frequency spectrum used by cells, capacity improvements can be reached. (Kaaranen, et al 2005, 38)

2.7 Spreading and scrambling

A single base station receives a multitude of signals constantly, and at the same time it has to be able to tell which signal come from which user. Also at the UE end, at the same time an UE also has to be able to distinguish the signals which are meant only for it. This is where the concept of spreading operations come in. In essence, spreading means increasing the signal bandwidth, and it consists of two operations: channelization and scrambling. (Korhonen 2003, 58-59.)

Channelization uses orthogonal codes, which means that (in an ideal environment) these codes won't clash with each other. Very simply explained, the generated code is used to separate users in the radio downlink (from base station to UE) of the same cell area, but the same separation cannot be performed for the uplink (UE to base station). The reason for this is that orthogonal codes in this context won't interfere with each other only if they are time-synchronized. A Node B can easily broadcast its signals in a synchronized manner, but for the other way around it isn't ideal at all. For the uplink solution there is scrambling codes, which are the second part of the spreading procedure. Scrambling codes are not orthogonal, but pseudorandom. Once a scrambling code is assigned, a spreading code is formed, which identifies different UEs in the Node B uplink. (Korhonen 2003, 112-115.)

Furthermore, scrambling codes can also be used with the radio downlink. Scrambling code utilization in the downlink direction reduces interference from base stations, which are too close to each other. Each base station is assigned with a single code as a primary scrambling code, and with it base stations can differentiate nearby cells. A base station can receive a primary code from 512 different primary scrambling codes. (Korhonen 2003, 116.)

2.8 Handovers

User mobility problems are solved with such functions as paging, location updating and connection handover. From these functions the handover is definitely the most important one, while at the same time is the cornerstone of cellular mobility. The basic idea of handovers are that when a subscriber moves from a cell to another cell, the radio link is handed over to the target base station without interruption. Handovers come in a few flavors. Soft handovers occur when an UE is connected simultaneously to more than one base stations. This sort of situation is typical when the UE is located within boundary areas of two overlapping cells. The network monitors constantly signal strengths between base stations. If the current active radio link goes below a threshold, the connection is handed over to a stronger radio link, if available. The procedure of monitoring active cells, which are connected to the UE, is called active set update (ASU) (Korhonen 2003, 38-39)

Along with soft handover, there is hard handover. While a soft handover is seamless (the user won't even notice that radio links have been swapped), hard handovers are not. Hard handovers occur when the used radio frequency of the UE changes. In these situations the UE momentarily stops transmitting, moves to the new frequency and starts transmitting again. A user might experience this with a noticeable drop in audio during calls. (Korhonen 2003. 44)

A third kind of handover is called relocation. These relocations occur when a subscriber moves within cells that are controlled by different RNCs. Extra measures are taken as the network now needs to monitor that two RNCs are successfully communicating with each other about serving radio links to the same UE. The concept of relocation was already introduced in section 2.5 along with the concepts of SRNC and DRNC. In this context, SRNC is the RNC which controls the cell where the connection was initiated. When the user moves to a different cell which belongs under a different RNC's domain, after the handover is done the new RNC takes the role of DRNC. (Korhonen 2003, 41-42)

The handover decision is made by measuring RSCP (Received Signal Code Power), which is the power measured by the receiver on a physical communication channel. RSCP is generally an indicator for signal strength. RSCP is measured from the radio downlink, meaning that measurement is performed by the UE, which then reports the measurement result back to the base station. (ETSI 2012.)

2.9 Location updates and paging

Location update and paging functions are for keeping track of the subscriber even when the UE has fallen outside of an active radio link. In order to do this, a geographical area is turned into location areas (LA), which consist of several pre-defined cells. Location updates take place periodically or the UE moves to a different LA. Location update procedure consists of requesting a location update from the network, obtaining the location information, and then deleting the old data. (Kaaranen, et al 2005, 44.)

Paging is a procedure which is used to reach an UE, when a call is made to it. Through paging, an UE's current location is determined by looking up the Home Location Register

(HLR) for local users, and the Visitor Location Register (VLR) for roaming users. Location areas are divided to sub-areas, which are called paging areas (PA). The relationships between these area concepts can be seen in Figure 3. (Kaaranen, et al 2005, 44.)

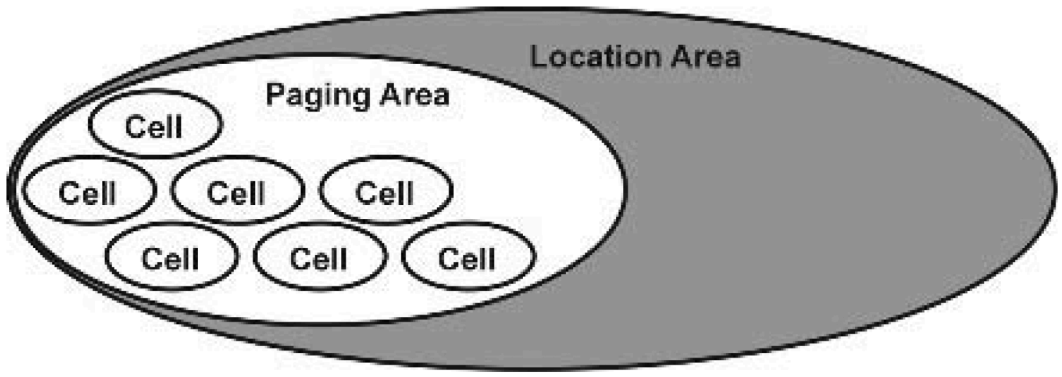


Figure 3. Relationships between cells, paging areas, and location areas (Kaaranen, et al, 2005, 44).

2.10 Node B Application Protocol (NBAP)

For the base station and RNC to communicate with each other, a protocol for this specific need has to be realized. For this there is NBAP. This protocol for configuring the management and setup of Node Bs (base stations) on the Iub interface. Base stations do not make any decisions on their own about radio resource management, instead all these decisions are made in the RNC, and BSs obey these commands without question. The control-plane is provided by the NBAP. Important duties of the NBAP are establishing and maintaining a control-plane connection over Iub, initiation and release of user-plane connections over Iub, and seeing that Node Bs active radio resources, when they are being requested over Uu. NBAP also is responsible for initiating specific measurements within base stations, and it has to ensure that these measurements are reported back. (Kaaranen, et al 2005, 321-323.)

3 NETWORK OPERATIONS

Designing and planning mobile technologies is a hugely resource consuming business, but equally resource consuming is running and maintaining network operations around them. In this chapter we will take a closer look at the challenges network operators have to tackle to build and maintain a functional mobile network.

3.1 Network Management

The biggest challenge in planning network operations lies in the fact that there is no ready-defined scheme for telecommunications-network-management. This is almost a polar opposite for mobile technology planning where the specifications are strictly defined by many large international specifications bodies. However in UMTS there do exist some specifications in TMN (Telecommunications Management Network, Figure 4), but these are meant mainly as guidelines. As UMTS networks inherently are multivendor networks, meaning that network element hardware are provided from many different mobile network companies, a somewhat common framework is a must for design compatibility. (Korhonen 2003, 279.)

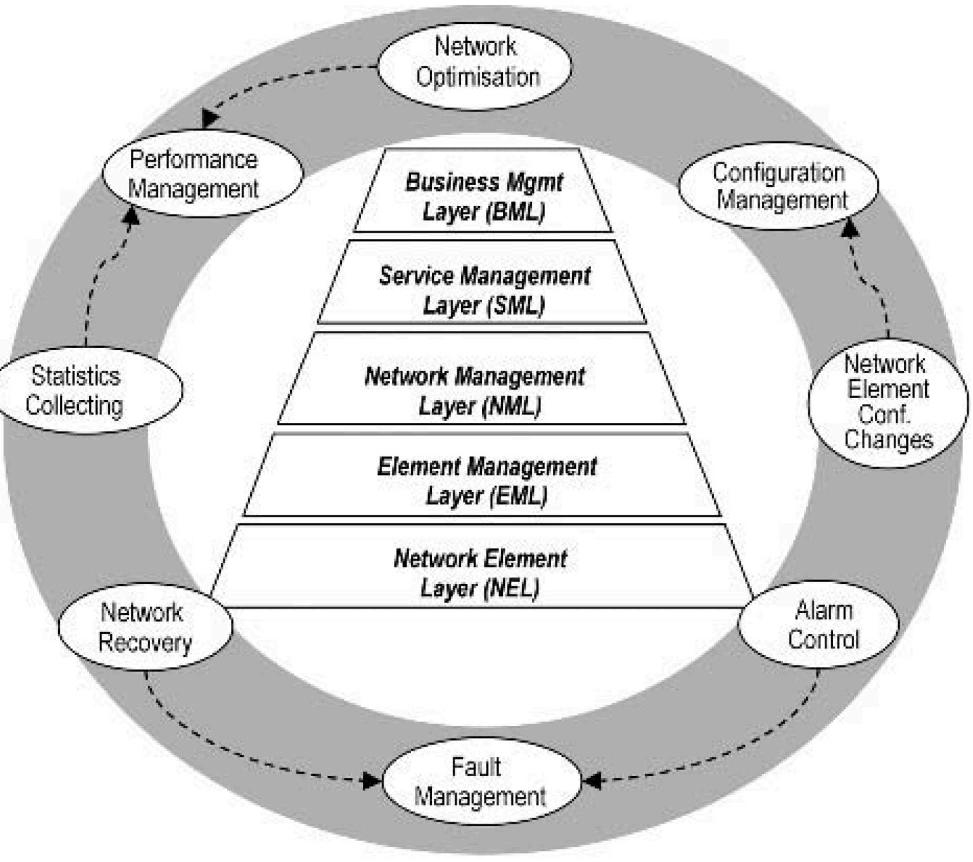

Network Optimisation
Performance Management
Configuration Management
Business Mgmt Layer (BML)
Service Management Layer (SML)
Network Management Layer (NML)
Element Management Layer (EML)
Network Element Layer (NEL)
Statistics Collecting
Network Element Conf. Changes
Network Recovery
Alarm Control
Fault Management

Figure 4. The TMN model (Kaaranen, et al 2005, 52).

The main subject in this section are the ITIL (Information Technology Infrastructure Library) processes which help define different kinds of data management. The three most used data types define the holy trinity: performance management, configuration management and fault management. These three are mainly responsible when it comes to configuring, optimizing and monitoring a mobile network, and they also play a huge role in self-organizing networks (SON) where human involvement is needed less (Turkka 2014, 19). We will now only briefly touch upon these different processes, as the next section take a closer look at these with the subject of optimization and fault handling.

3.1.1 Configuration Management (CM)

As network elements are in constant need of configuration and adjustment, hardware vendors have to enable a path to enable easy and effortless configuration control. Configuration changes are mainly short-term, because of the ever-changing needs of network resources. The most challenging aspect is in the form of network upgrades. As network equipment changes and evolves, old configuration changes might not be suitable anymore. For this reason, configuration management has to offer the aspect of reversibility. In case of upgrading related problems, changes have to be undone quickly, as breaks in telecommunication services are very quickly frowned upon by service subscribers. (Korhonen 2003, 282.)

3.1.2 Fault Management (FM)

As mobile networks are highly complex and advanced technical accomplishments, this also means that they are also highly error-prone. The main function of fault management is to detect the fault is soon as it manifests within the network. Many things can go wrong. For example electronic components can fail inside the network elements, equipment can contain buggy software which causes a system malfunction or small capacity due to traffic can block equipment from functioning correctly. (Korhonen 2003, 280.)

Korhonen in his book (2003) has neatly summarized the bare minimum what a fault alarm should contain:

> A fault report (also called an alarm) should contain (at least) the exact location of the fault, its type and severity, its probable cause, the time stamp of the fault detection, and the nature of the fault. All alarms generated by a network entity and not yet fixed and cleared must be stored in a list of active alarms in that entity. (Korhonen 2003, 281.)

The severity of the alarm is fully dependent on its type. Some faults are simple enough that the system itself can fix them, thus they can be cleared automatically at the same time when they are detected. Other alarms require manual input from the operator. For example faulty equipment has to be replaced to a functional one and in case of software related faults, reboots or updates are needed. (Korhonen 2003, 281.)

3.1.3 Performance Management (PM)

To start configuring network elements in a sensible way, first we need to see how elements in a mobile network are responding to their given configuration. The eyes in this setting are played by PM data. With performance measurements we are able to verify things like traffic levels, configuration change validity, resource availability and the quality of the network (Korhonen 2003, 283).

Later we will view performance management from a more practical perspective, when we reach the practical part of this thesis. As the use-case in this thesis involves displaying a real-time situation of a mobile network, it can be used by the network operator to troubleshoot and enhance the performance of the network.

3.1.4 Roaming Management

An operator's network coverage can reach only as far as the operator is able to invest in radio nodes and other mobile equipment. Normally all operators are able to cover all areas sufficiently within the country they are operating. This of course means that when subscribers are travelling abroad, they need to use radio resources of a different operator than their home operator. This sort of functionality is called roaming, and it addresses the issues related on roaming agreements between operators. Roaming agreement contracts usually define pricing, signaling interconnection, call detail records, and which side handles problems when they occur with a roaming subscriber. (Korhonen 2003, 284-285.)

3.1.5 Accounting Management

For charging and billing subscribers, there is accounting management. Part of accounting management is cost control which monitors and generates charges for the services which a subscriber has used. For local subscribers this is simple and not much interactions are needed, but for visiting users more actions are needed. Accounting management also works closely with fraud management, as dishonest use of network resources usually is done to avoid service fees. (Korhonen 2003, 285.)

3.1.6 Subscription Management

As with evolving technology service variety increases, users can tailor their subscription to have only services, which best fit their needs. Subscription management allows for operators to have access to their user's subscription data, and from this data the network can choose what services are provided for the subscriber. User subscription data is stored home subscriber server (HSS) and within the user’s SIM. With subscription management, 3GPP aims to standardize the interface to HSS, as proprietary interfaces only inconveniences operators which have hardware from multiple vendors. (Korhonen 2003, 285.)

3.1.7 QoS Management

Managing the network QoS consist of two areas: QoS policy provisioning and QoS monitoring. For configuration and maintenance of network elements with QoS policies, QoS policy provision is implemented. Policies are based on subscriber feedback and observed network performance. QoS monitoring handles the process of collecting QoS statistics and alarms, which is then used for analyzing if changes or upgrades are needed for the network. (Korhonen 2003, 286.)

3.1.8 User Equipment Management

UE management is used for activating and deactivating tracing functions for selected subscribers. When UE management is active, it reports trace activity back to the network management system. (Korhonen 2003, 286.)

3.1.9 Fraud Management

In essence, fraud management is fraud prevention. The aspect of mobility makes it more difficult to prevent fraudulent use of network resources in mobile networks, than with fixed networks. Mobility allows fraudulent use anywhere from the coverage area. Even outside of the home network’s coverage area, as fraudulence can be committed via roaming. Fraud prevention aims to prevent misuse from happening at all with several measures. For example, new subscriber’s credibility will be checked. If credibility is low, then the operator can refuse to provide services altogether. Network operators also monitor for fraud patterns. If a known pattern is detected, subscription is suspended. Visiting subscribers bring an extra challenge, as network operators are not able to test them in the same manner as local users. There exists however international registers, such as the Central Equipment Identity Register, which can be consulted for detection of fraudulent users.

For fraud detection to be as effective as possible, fraud detection should done as much as possible in real time. (Korhonen 2003, 287.)

3.1.10 Security Management

Network operators need to operate the network with O&M (operations and management) software, and this needs to be done with a secure way. When speaking of security management, two layers are brought up: the O&M IP network layer, and the application layer. The idea behind O&M IP network layer is that a private network is built alongside with the regular network for secure network management. This however is not a feasible concept as networks are usually large in size. For this reason, the O&M IP network layer usually just encrypts O&M related traffic. The application layer provides authentication and logging services. With these services the network can check if the user is authorized to perform requested operations on the network. With logging operators are able to check who has done what. (Korhonen 2003, 287.)

3.1.11 Software Management

Software management can also be divided into two areas: the actual software-management process and software-fault-management process. The software-management process handles new software releases and possible corrections for them. Software-fault-management then monitors and takes care of faults caused by problems with new software. (Korhonen 2003, 288.)

4 OPTIMIZATION AND FAULT HANDLING

To fully utilize network equipment, a network operator must learn how to optimize their hardware for maximum efficiency. It is not feasible to change network equipment at the same rate than new technologies and hardware pop up, so that’s why optimization is also a must for maximum hardware longevity. In this chapter we will delve deeper into performance measurements, key performance indexes, and what they mean in the perspective of network optimization.

Sections 4.3, 4.4 and 4.5 concentrate on different concepts about data collection. The information told there will function as a prerequisite when later we will take a look at Nokia’s MegaMon L3 Data Collector on section 5.4.

4.1 Performance measurements

Telecommunications Management Networks and performance measurement planning share one common thing: there are no ready-made specifications on how TMNs or performance measurements should be implemented. As the amount of attributes and events that can be measured is very high, it tough to say on what should a network operator focus their performance resources. For example operators can choose to favor network coverage instead of high data transfer speeds and measure performance with this perspective accordingly. An operator has to make the decision of where to invest their network capacity at. It would be wise though to take into large consideration the service subscribers’ wishes. Subscriber feedback is also an excellent validation on how well performance feedback is being utilized. (Kreher 2006, 2.)

One practical example of a performance measurement could the counter *sho_succ_nb*, which measures the amount of successful soft handovers from a cell to another cell. Looking at Table 1, we see an example of what kind of data usually belongs to this counter. There are three columns belonging to this example, from which the first one “Source cell ID” tells from which cell the handover is initiated from. The second column “Target cell ID” tells to which cell the handover for was targeting at. The third column contains the count of handovers which were initiated successfully. Usually a performance measurement also contains columns for a timestamp and time duration, but these were omitted from this example.

Table 1. Example data belonging to the counter sho_succ_nb.

Source cell ID	Target cell ID	Successful attempts
100745	200044	230
166733	300300	3
100745	556441	14
100745	100654	0
245987	222330	0
166733	772211	512

PM data can be further constructed by containing different kind of performance counters to higher level indicators commonly called as KPI (Key Performance Indicator). The concept of a KPI is not tied to telecommunications only. KPIs are found plentiful as much from economic and business production as from technical development. (Kreher 2006, 4.)

Expanding on the idea presented on Table 1, the counter *sho_succ_nb* could have a complementing counter called *sho_att_nb*. While *sho_succ_nb* represents the amount of successful soft handovers, *sho_att_nb* counts each and every attempted soft handover, successful or not (Table 2). From these two counters we could devise a KPI which shows the success ratio of attempted handovers. This would be easily accomplished with this formula:

$$\frac{sho\_succ\_nb}{sho\_att\_nb} = handover\ success\ ratio \tag{1}$$

When the amount of successful handover is divided with total attempts, the success ratio is attained.

Table 2. Example data belonging to the counter sho_att_nb.

Source cell ID	Target cell ID	Handover attempts
100745	200044	255
166733	300300	5
100745	556441	54
100745	100654	0
245987	222330	6
166733	772211	512

4.2 Network optimization

As briefly mentioned previously, optimizing a network is mostly a question of how the network capacity is divided for the target goal. For the network to be in control of its own capacity, resource management has to be implemented for effectively utilizing resources. In terms of radio access networks, the expressions Radio Resource Utilization (RRU) and Radio Resource Management (RRM) are used. The latter mentioned RRM usually consists of functions like power control, handover control and congestion control. Briefly running through these functions, power control (as the name suggests) has the role of adjusting transmission power. Handover control is a function that is responsible for handing the connection over between network elements, when a mobile user moves from one coverage area to another. Congestion control ensures that the network stays in good shape even during high traffic. Congestion control can be further broken down into admission control, load control and packet data scheduling, which each have their own respective role in keeping up the quality of service (QoS). (Laiho 2006, 197.)

4.3 Trace data

Trace data (also known as subscriber and equipment trace or more simply as just trace) is low-level data at call level on user equipment. This meaning that via trace we can monitor call attempts on a signaling level. This further means that we are able to examine events which occur when the signal moves between different elements and their connecting interfaces (Uu, Iu, Iub, etc...) within the network. trace is vital when in need of advanced troubleshooting, resource optimization, coverage control, and capacity improvement. As performance measurements usually deliver aggregated values, trace is much more convenient when need of more specific data. 3GPP's TS (technical standard) 32.423 defines a standard of what a trace record should contain.

One solution to utilize trace measurements is the aforementioned Nokia MegaMon software, which will be detailed more in section 5.4. There are also many more solutions, from which one is Viavi Solutions' Signaling Analyzer family. The SART (Signaling Analyzer Real Time) solution promises highly effective troubleshooting with just a couple of clicks for multi-technology targets (GSM, UMTS, LTE, etc…) while multiple users are using it simultaneously. Call traces are visualized graphically with various KPIs, which help to discern poor areas with call success ratios, signaling and call load, user-plane performance, and various other metrics.

4.4 Radio Resource Controller and RRC measurements

Expanding on the concepts of RRM and RRU, for the RRM to be able to deliver information messages within the network, a radio path controlling protocol is needed. This is the role which the RRC protocol is designed for (Kaaranen, et al. 2005, 112). RRC is a sublayer of Layer 3 (the radio network layer) on UMTS radio interface, and it provides an information transfer protocol to the non-access stratum (NAS) and at the same time it also controls the configuration of UMTS radio interface Layers 1 (radio physical layer) and 2 (radio link layer). The main functions of RRC include creating and managing resources for RRC connections (i.e. establishing a link to the UTRAN), UE location management and connection handover between radio link nodes, power management, and setup and management of RRC measurements (Kreher & Rüdebusch 2007, 112-113).

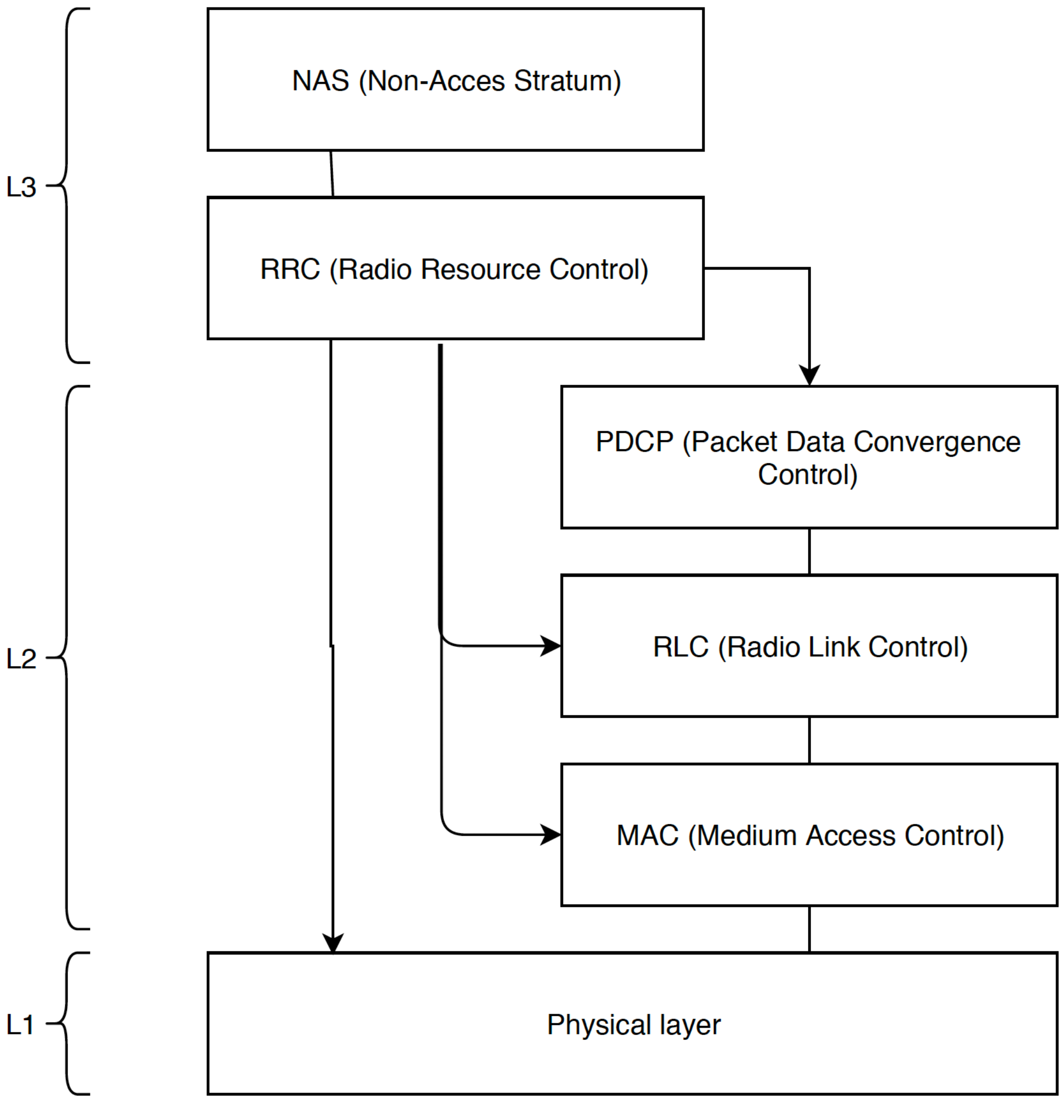

L3
L2
L1
NAS (Non-Acces Stratum)
RRC (Radio Resource Control)
PDCP (Packet Data Convergence Control)
RLC (Radio Link Control)
MAC (Medium Access Control)
Physical layer

Figure 5. 3GPP layers.

When a SRNC establishes a RRC connection with an UE, the SRNC sends a RRC Measurement Control message to the UE, which initiates a RRC measurement procedure. The initiating RRC Measure Control message contains information on what kind of measurement reports the SRNC is requesting to receive from the UE. Different types of measurements are categorized into different type of groups, and these different measurement groups can be described through different event-IDs. When a predefined threshold is reached, an event-triggered measurement report is sent towards the SRNC. (Kreher & Rüdebusch 2007, 239.)

Figure 5 shows a model of the 3GPP layers. As mentioned before already in this section, RRC is part of Layer 3. Sublayers of Layer 2 include PDCP (Packet Data Convergence Protocol) which offers transfer of user and control plane data, RLC (Radio Link Control) which is for transfer of PDUs (Protocol Data Unit), and MAC (Medium Access Control) which encapsulates data packets to be transferred from interface to interface.

4.4.1 RRC State Machine

To get a better understanding of RRC's signaling procedures, it is good to be aware of the RRC state machine, which handles the state of UE depending on its context towards the network. When an UE is turned on, it starts up with idle mode (RRC_Idle) until it requests an RRC connection from the network. In RRC_Idle state the UE is camping on a cell, but its individual presence isn't registered to the network. Location update procedures work normally however. (Kreher & Rüdebusch 2007, 114-115.)

For the connection to move onwards from idle mode, a RRC connection has to be requested from the network. From RRC_Idle state the UE enters either CELL_DCH (dedicated channel) or CELL_FACH (Forward Access Channel) states. The CELL_DCH state is characterized by conditions like when a dedicated physical channel is allocated for the UE, the UE is visible to the connecting cells, it possible to initiate a soft or hard handover, and the UE is sending RRC measurements to the RNC accordingly to the RNC's requests. In the state of CELL_FACH conditions like these occur: no physical channel allocated for the UE, no soft or hard handover will be initiated, and the network knows the position of the UE accordingly where the UE last made a location update procedure. (Kreher & Rüdebusch 2007, 115-116.)

The last two states are CELL_PCH (Paging Channel) and URA_PCH, which are somewhat identical. CELL_PCH state has features like: no dedicated physical channel allocated, no soft or hard handovers are possible, no uplink activity is possible, UE updates its location cell change, and UE sends RRC measurements to the RNC. The only difference with URA_PCH state is that location update procedures are not initiated by cell area changes, but with URA (UTRAN Registration Area) changes. (Kreher & Rüdebusch 2007, 116-117.)

4.5 Minimization of Drive Tests (MDT)

For an operator to ensure the QoS of its mobile network, constant monitoring has to be done in terms of network coverage. For the longest time, this was done with field tests, which included driving around with special equipment measuring signal coverage in various locations. Measurements done in this fashion are, for obvious reasons, very resource depleting, as drive tests require special equipment and people specialized using them. To overcome the need for drive tests, 3GPP started devising the concept of MDT. With 3GPP Releases 10 and 11, a standardized framework for gathering and fetching automated radio measurements was introduced (Turkka 2014, 3).

MDT reuses and extends the concepts of trace and RRC measurements to support its own functionality (ETSI TS 137 320 2016, 8). With MDT operators are able to collect measurements straight from user equipment. A collected measurement record usually contains the UE location (latitude and longitude), signal strength and a timestamp (i.e. when the record was logged) (Turkka 2014, 29-30).

MDT measurement retrievals come in two flavors: immediate MDT and logged MDT. As the name suggests, immediate MDT collects (extended) RRC measurements immediately when there is an active RRC connection. Logged MDT measurement mode allows for data retrieval of idle user equipment. When the UE moves to an RRC idle state (not actively connected to a radio node), the measurement data is logged into UE memory. Later when the UE reconnects within the network, it reports that it has logged measurements ready for retrieval. (Turkka 2014, 27-28.)

The use-case in this thesis utilizes data which is retrieved using MDT. As mentioned in the introduction, the use-case centers around network coverage and cell dominance. Network coverage can be easily measured just with coordination and signal strength measurements, which are basic MDT measurements. Cell dominance in the other hand requires a bit more data. This is covered more thoroughly within the next chapter. The information attained with the use-case data can be mainly used for verification of planned network coverage, but could also be used with troubleshooting customer service cases.

5 USE CASE INTRODUCTION

At this point we have arrived to the second part of this thesis. The first part being an introduction to 3G mobile network technologies, this second part solely focuses on my own implementation of this work. The focus on this work centers around a use case, which was handed to me by Elisa Oyj, which is a Finnish mobile network operator. This use case involves reading signaling measurements gathered from RNC devices, and aggregating them into performance and configuration statistics. These stats are further visualized by coloring areas in a map with different colors reflecting a different state of performance or configuration happening in each respective area. As mentioned in this thesis' introduction text, the given use-case involves visualizing network coverage and cell area dominance. The data for these measurements is provided by Nokia's MegaMon and Emil software, which will be looked at shortly.

Explaining the idea of this use-case a bit further, the visualization is implemented so that a map is divided into a square grid of roughly 75 x 75 meters per grid square. Going through square by square, each area is checked for available measurements. With network coverage visualizations, the available measurements are checked for signal strengths. All the signal strengths available within the same area are summed for an arithmetic mean. The outcoming number will decide the quality of network in that particular area and the area will be colored accordingly.

As explained in section 2.7, scrambling codes can be used to differentiate nearby cells from the UE perspective, and also to reduce inter-base-station interference. For reducing interference purposes, each cell is signed with a primary scrambling code. The measurement data provided for this work includes the primary scrambling code active at the cell at the time, when the measurement was retrieved. Using the same area grid, than with network coverage visualizations, we can group all measurements within one area, and calculate which primary scrambling code has the highest frequency count in said area. Once we know the most frequent primary scrambling code, we can deduce the cell which is the most dominant in that area. The most frequent primary scrambling code is also visualized with colors. With a wide color spectrum used, we are able to see how in different areas cell dominance is spread.

The square grid on top of the map is accomplished by with geohashing. Geohashing is a concept which is further explained in section 5.8. An example of network coverage visualization can be seen in Figure 6 and Figure 7.

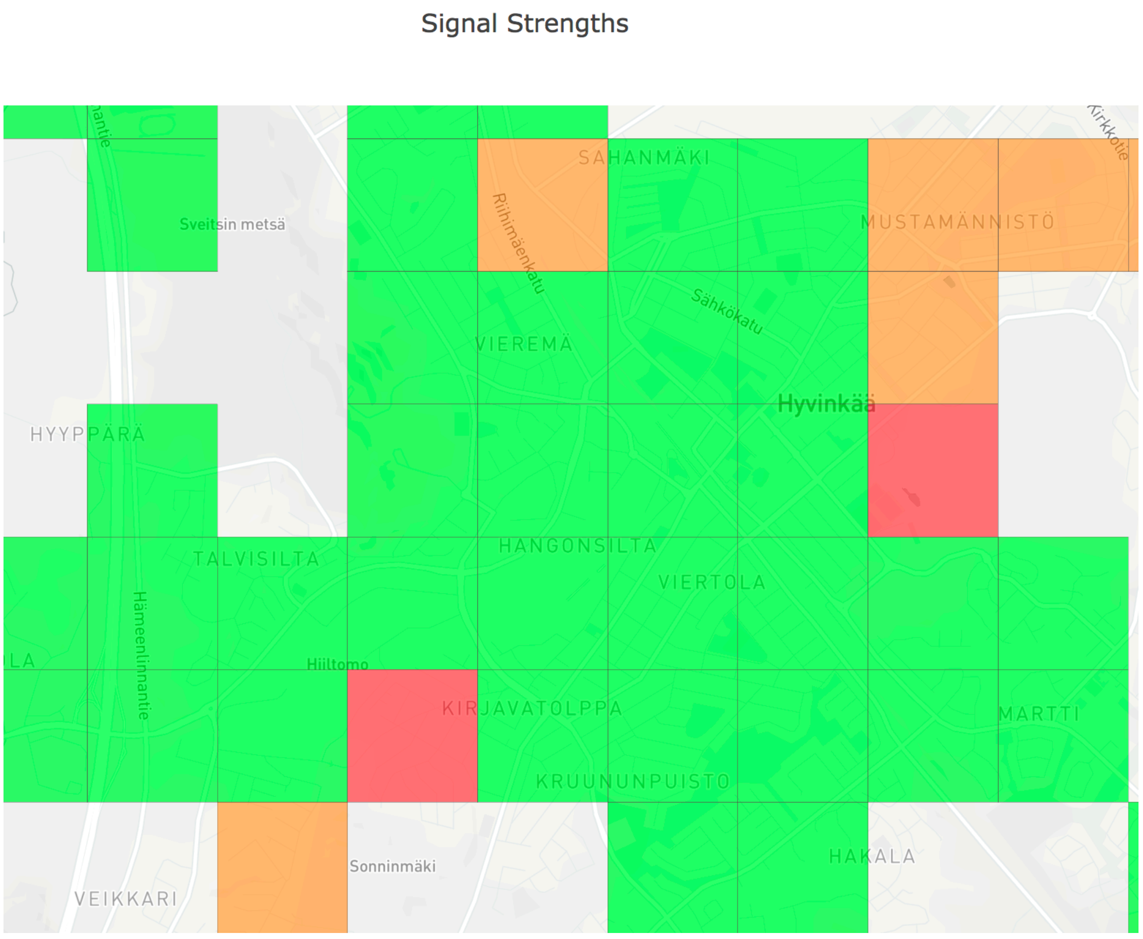


Figure 6. A screenshot of a map application featuring signal strength measurements from Hyvinkää.

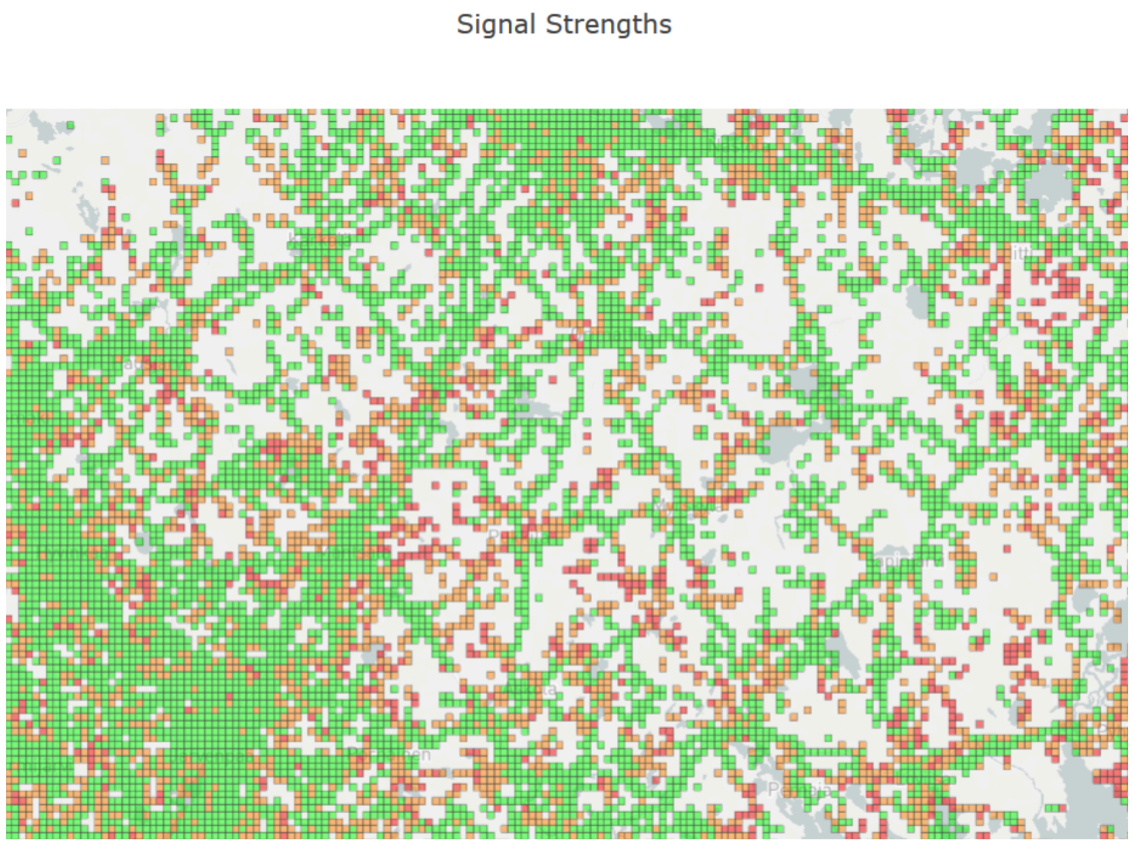


Figure 7. A screenshot of a map application featuring signal strength measurements from a broader range of Finland.

Now that the general idea of the use-case revolving around this work has been explained, the rest of this chapter will be kicked off by going through different technologies used in the software stack. Firstly, we will go through the main programming language used in this work. Different libraries and frameworks are discussed in sections. Technologies are sorted by their relevancy related to this work. The next section will take a closer look at the software stack itself and its underlying architecture.

5.1 Python

Created by Guido van Rossum, Python is a general purpose programming language first released in 1991 (Kuhlman 2009). Initially Python was a hobby project for Van Rossum who was looking to replace the ABC programming language (Van Rossum 1996). Python has two strong aspects, which make it a favored programming language by many. Firstly, Python aims for a simple, less-cluttered syntax. Python's core philosophy has been summarized in "PEP 20 -- The Zen of Python", which includes:

- *Beautiful is better than ugly.*
- *Explicit is better than implicit.*
- *Simple is better than complex.*
- *Complex is better than complicated.*
- *Readability counts.*
- *Errors should never pass silently.*
- *Unless explicitly silenced.*
- *If the implementation is hard to explain, it's a bad idea.*
- *If the implementation is easy to explain, it may be a good idea.*

Secondly, Python is a vastly extensible language with a large standard library, including tools for a large amount of uses. These include, and are not limited to, Internet applications, graphical user interfaces, databases and unit testing.

The first version of Python being released in 1991, Python 2.0 saw its release on October 16th of 2000. Python's second iteration saw many new features and by this time the development had shifted to a more community-based one. Python 3.0 was released on December 3rd of 2008. This release brought major revisions to the programming language. As shortly mentioned earlier, Python's development these days is highly community-based. New features to Python are brought in through a process known as the Python Enhancement Proposal (PEP) process (Warsaw, et al. 2000). Van Rossum himself has greatly contributed towards the development of Python as Python's "Benevolent Dictator for Life", but as of July 2018, he has stepped down (Van Rossum 2018).

Python's name is derived from the hugely popular sketch comedy show Monty Python's Flying Circus, which Van Rossum was enjoying at the time of Python's conception. Originally the name Python was just a working title, but it stuck beyond initial development. (Van Rossum 1996.)

In my work, Python is the main programming language, which was chosen for its extensive capabilities to process any kind of data, and its features to work with outside interfaces.

5.1.1 Pandas

As just mentioned, Python is very suitable for tasks including data manipulation and analysis, thanks to its vast library base. In terms of data processing, pandas is a very capable library. Pandas offers many data structures for handling data within typical use cases. Of these data structures, the two main are "Series" and "DataFrame", which can be thought as one-dimensional and two-dimensional data arrays. Typical use cases for pandas include fields like business, statistics and engineering.

Pandas was originally conceived by Wes McKinney on January 11th of 2008. Originally developed for the company, where McKinney used to work for, as a flexible tool for financial data, pandas was later released as free open source software under the BSD license (Kopf 2017). Pandas aims to be "the most powerful and flexible open source data analysis / manipulation tool available in any language", and the developing team claims to be well on its way toward that goal (Pandas 2018).

For this work pandas was chosen for data processing, as pandas plays well with CSV files. Furthermore, pandas has excellent support for interfacing with many database protocols, so inserting data from files to database is easy out of the package.

5.1.2 Plotly

For plotting graphs, Plotly Python framework is an excellent Python graphing library. Plotly's specialty are interactive graphs, which can be viewed through a web app, if desired so. Plotly offers a variety of graphs, as for example scatter plots, line charts, bar charts, pie charts, histograms, heatmaps, time series charts and many more! These can all be created easily within the realm of Python.

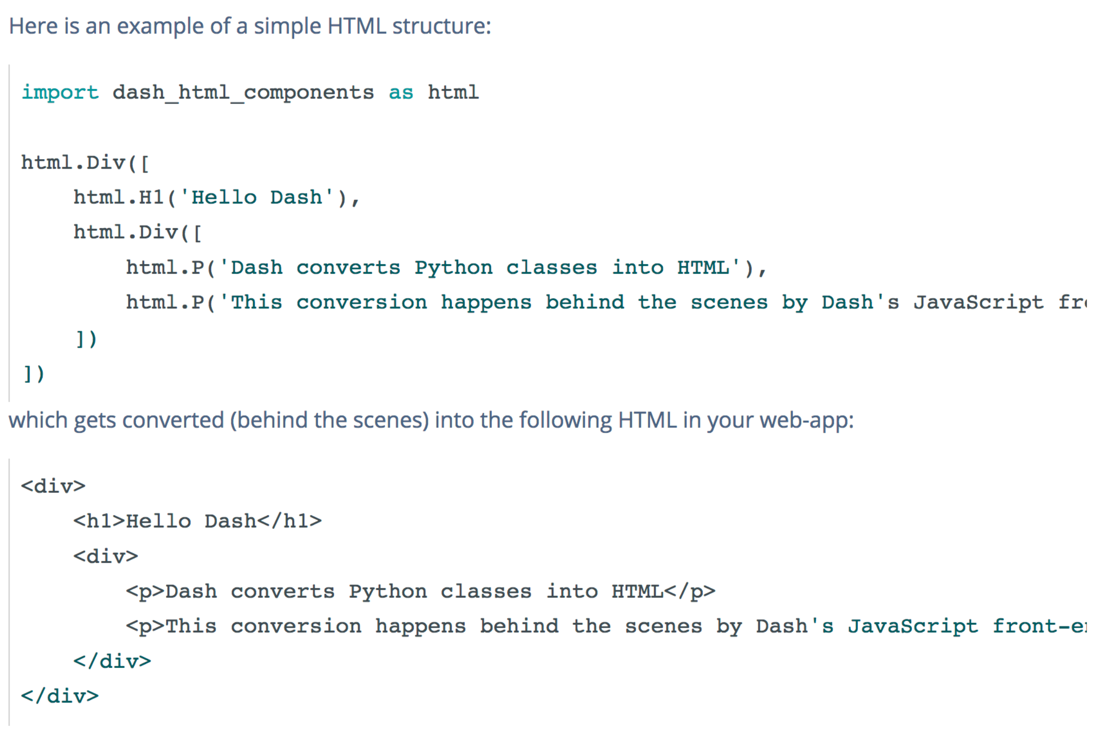
Along with Plotly's graphing library goes Plotly's Dash framework. Dash is an out-of-the box solution for displaying Plotly-made graphs within interactive web apps. Dash will be covered more soon.

Plotly's role in this work is to provide the interface for plotting areas on a map. Plotly is able to draw plots on top of Mapbox maps reading the data provided from GeoJSON files. Mapbox and GeoJSON will be covered in later sections.

5.1.3 Plotly Dash

Plotly's Dash is an open source Python framework specializing in data visualization. Dash aims to provide a simple, straight-out-of-the-box solution for building a web-based visualization app which can be viewed from a web browser. Dash abstracts HTML, CSS and JavaScript under single component libraries in such fashion, that a user can write a complete interactive web page with just writing Python code. Data can be visualized through graphs, tables, etc.

Dash mainly consists of two core libraries: Dash Core Components and Dash HTML components. The Dash Core Components library is for creating interactive interfacing components (such as dropdown menus, sliders, and input and text boxes) and visualization items (such as graphs). The Dash HTML Components library is (as its name suggests) for abstracting HTML implementations into Python. As seen in Figure 8, Dash takes in Python code which is converted into HTML when the constructed Dash app is launched into a web browser.


Here is an example of a simple HTML structure:

```
import dash_html_components as html

html.Div([
    html.H1('Hello Dash'),
    html.Div([
        html.P('Dash converts Python classes into HTML'),
        html.P('This conversion happens behind the scenes by Dash's JavaScript fr
    ])
])
```

which gets converted (behind the scenes) into the following HTML in your web-app:

```
<div>
    <h1>Hello Dash</h1>
    <div>
        <p>Dash converts Python classes into HTML</p>
        <p>This conversion happens behind the scenes by Dash's JavaScript front-e
    </div>
</div>
```

Figure 8. An example demonstrating Dash's conversion from Python to HTML.

Plotly released Dash on June 21st of 2017. Dash is a product of prototyping for two years with feedbacks from banks, labs and data science teams. As mentioned before, Dash aims to provide a solution which can be run straight out-of-the-box. This is achieved by having Dash run on other platforms such as Flask (a Python framework for constructing web services) and React.js, which renders all the frontend components. (Dash 2017.)

For this work, Dash provides means for displaying the Plotly-generated map graph, which displays network coverage information.

5.1.4 Venv

The problem of managing multiple Python projects is sometimes library dependencies. One project may require a very specific version of a certain library, while at the same time another project absolutely requires another version of the same library. Another problematic situation is where you can’t install new libraries on the global Python library directory, because you are working on a shared host. This is where Python virtual environments (or venv) jumps in. Usually Python libraries are stored within the path /usr/lib/python2.7/site-packages (on Linux, other platforms may have different locations), but with virtual environments the library directory is installed straight to the root of the project directory, which means that libraries are not shared between projects.

5.2 Deck.gl

Deck.gl is a Node.js library, which utilizes WebGL for generating map with visual overlays. Node.js is a JavaScript runtime environment for typically executing server-side JavaScript code for dynamic web pages. WebGL (Web Graphics Library) is JavaScript framework for rendering 2D and 3D graphics, which are viewable within a web browser. Deck.gl is designed for visualizations for large data sets.

In the context of this project, Deck.gl is used to visualize primary scrambling codes within mapped areas. Visualizing the scrambling code data needs a dynamic way to assign color information to a specific scrambling code, and at the time of developing this project, Plotly wasn’t able to deliver such a method. This is issue is viewed in more detail in subsection 6.3.5.

5.3 MariaDB

After the popular and free database MySQL platform was acquired by the Oracle Corporation in 2008, the development of MySQL was forked into a new development path to ensure open and free use. The new product which came out of this, is called MariaDB. Development led by the original developers of MySQL, MariaDB is intended as a 'drop-in' replacement for MySQL, meaning that MariaDB tries to be 'backwards compatible' with MySQL as much as it can. The lead developer of MariaDB, Michael Widenius, has expressed that as long as the amount of MySQL users surpasses the users of MariaDB, the drop-in compatibility is essential for making the transition to MariaDB as painless as possible. This however doesn't mean that development hasn't focused on making the actual product faster with more features. Widenius has bragged that he believes that MariaDB will always be more faster and secure than the Oracle Corporation's iteration of MySQL. (Pearce 2013.)

Like MySQL, MariaDB also attributes to the relational database model. Simply explained, the basic gist of the relational model is that data is organized into separate tables. Each table (or relation) has rows (or records) and columns (or attributes), each row having a unique key for separating it from the rest. This unique key could be for example a ID number, user name, product ID, or a timestamp depending on what kind of data the table is holding. Furthermore, different tables can be linked with a foreign key, which can be column which exists in two (or more) tables. For example, let's say we have a table which stores info about employee street addresses and another which has the same employees phone numbers. In this case a foreign key could be an employee ID. With this foreign key, we can search at the same time for a street address and phone number with the wanted employee ID simply by asking to return records from both tables with the wanted employee ID. Queries to relational databases are usually done with SQL (Structured Query Language), and this is also the case with MariaDB.

MariaDB was chosen for this project as the database for holding enriched data aggregated from the base data. MariaDB has a reputation for being fast with large amounts of data, so it was seen optimal for this cause.

5.4 Nokia MegaMon

Alternatively known as Nokia's L3 Data Collector, MegaMon is a Layer 3 data collecting software. MegaMon is designed to overcome the challenge of not having specific data to troubleshoot with when something has failed. How does it solve this problem? MegaMon simply records continuously all wanted events and stores the data for further inspection. MegaMon along with its needed hardware is installed and connected straight next to a RNC, where from the data is collected.

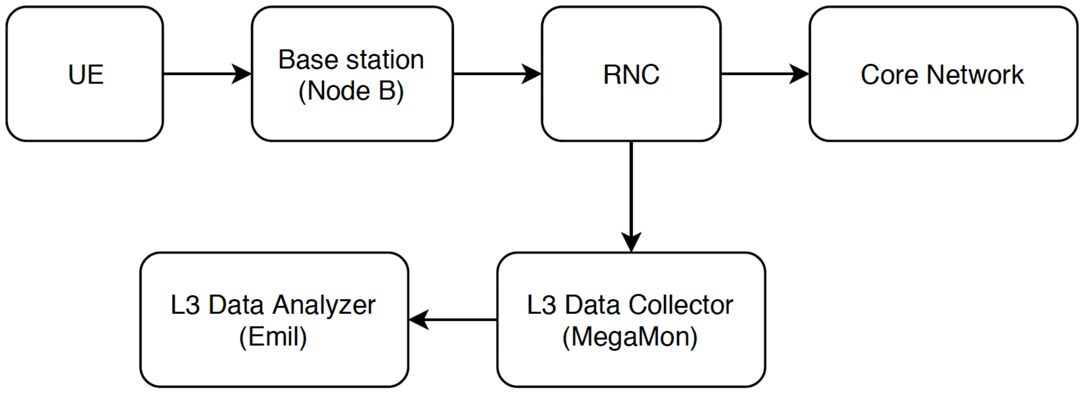
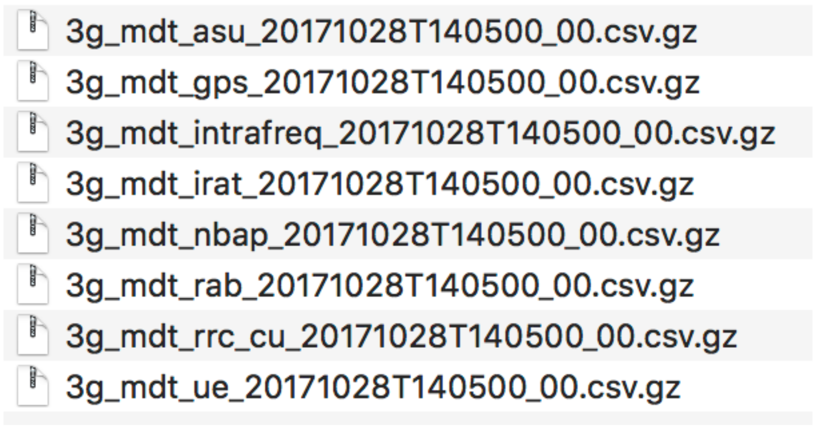


Figure 9. MegaMon data collection route.

As seen in Figure 9, MegaMon collects measurements from the RNC, and the measurements are passed onto Emil for analysis. The collected data in this context contains protocol traces from protocols such as RRC and NBAP.

5.5 Nokia Emil

As a counterpart for MegaMon, Emil (also known as L3 Analyzer / Viewer) acts as the element, which analyzes the collected data. The biggest function of Emil in this project is to parse collected data into human-readable CSV files, which are utilized as the base data. Measurements are dumped in intervals of five minutes into individual folders.



```
3g_mdt_asu_20171028T140500_00.csv.gz
3g_mdt_gps_20171028T140500_00.csv.gz
3g_mdt_intrafreq_20171028T140500_00.csv.gz
3g_mdt_irat_20171028T140500_00.csv.gz
3g_mdt_nbap_20171028T140500_00.csv.gz
3g_mdt_rab_20171028T140500_00.csv.gz
3g_mdt_rrc_cu_20171028T140500_00.csv.gz
3g_mdt_ue_20171028T140500_00.csv.gz
```

Figure 10. A screenshot of a directory containing CSV files dumped by Emil.

Figure 10 shows all files which were parsed by Emil within the test run. The file name indicates which kind of data is filtered into the file, and it also shows the timestamp when the file was dumped out. Seen in the screenshot is the file *3g_mdt_gps_20171028T140500_00.csv.gz,* which contains GPS measurements. These GPS measurements are the cornerstone of this whole project, as they contain location information, signal strength readings, and used primary scrambling codes of the measurements. With this data we are able to form information cellular coverage and cell area dominance.

Also worth mentioning is the file *3g_mdt_asu_20171028T140500_00.csv.gz*, which contains measurements about ASU (Active Set Update). As already discussed in section 2.8, ASU is used to initiate handovers, when the then active radio link goes below a threshold, and another monitored radio link has a more powerful connection to the UE. These files contain event data about handover initiations and connection successes or failures.

Going through rest of the files, *3g_mdt_intrafreq_20171028T140500_00.csv.gz* contains data regarding intra-frequency events (signals with same frequencies) in the same cell. File 3g_mdt_irat_20171028T140500_00.csv.gz is gathered with data from IRAT (Inter-radio Access Technology) events, which are recorded when for example a connection is handed over from a 4G cell to a 3G cell. File *3g_mdt_nbap_20171028T140500_00.csv.gz* contains records from NBAP events. File *3g_mdt_rab_20171028T140500_00.csv.gz* contains RAB (Radio Access Bearer) events. The RAB service creates and maintains radio resources for the end user to access the core network. File *3g_mdt_rrc_cu_20171028T140500_00.csv.gz* contains RRC events, and the file *3g_mdt_ue_20171028T140500_00.csv.gz* is gathered with records regarding UE data.

5.6 Docker

Common these days for development and production deployment is virtualization. This enables multiple products running in their own virtualized environment without affecting, or being affected by other environments. Environments communicate with each other only via predefined channels, which can be easily monitored for malicious activity. What Docker offers is operating-system-level virtualization, also known as containerization. Docker provides a way to run applications in containers, which are isolated and secured runtime environments for the application alone. Docker consists of the Docker daemon *dockerd*, Docker images (and containers which are instances created from images), and Docker repositories where from images can be downloaded. Containers run on top of Linux operating system's kernel and virtualization services provided by the kernel, which is why the container's themselves don't require their own virtual machines. Implementations of Docker for Macintosh and Windows both utilize their own lightweight virtualization platform for virtualization services. (Docker 2018.)

In this project, Docker is used for running the database. The MariaDB service is containerized in the context of this work.

5.7 Mapbox

Mapbox is a platform which provides open source application programming interfaces (API) and standard development kits (SDK) for displaying maps within mobile and web apps. The Mapbox company provides developers the means to display maps, search areas within maps, navigate around maps, and to create custom maps for use.

Mapbox was founded in 2010 as startup company, which offered mapping resources for non-profit customers. Throughout the years, Mapbox has grown into a multi-million business thanks to generous donations, and is now able to provide services for a wider audience, with a wider product range. (Maly 2013.)

The earlier mentioned Plotly framework for Python is able to utilize Mapbox's mapping service, and that exactly is its function inside this work. While Plotly plots the areas of interest, Mapbox provides the base map on which to draw.

5.8 Geohash

Geohash is a public domain geocoding system for encoding geographical location data (latitude and longitude) into strings (or hashes) of letter and digits. With geohashing we can present a degree of anonymity by presenting location in fixed areas rather than exact locations. The length of the hash equals with precision of the location. The longer the hash, the more precise the location. When we start taking dropping characters off, location precision goes down and the maximum error goes up. We can use this to our advantage by grouping many precise locations into groups based by the precision we want with geohashes.

As an example, say we have two geographical locations: (60.62522, 25.54888) and (60.62512, 25.54128). These can be converted to geohashes *udf0vkkps* and *udf0vhqwf*, which both have a (with a precision of 9 characters) maximum error of ±4,78 meters (Hill 2017). In our use-case where we are more interested in more larger areas, we can just take into consideration the first five characters of each geohash. This means that both of the previous geohashes are now considered as area udf0v, which means that now both of the locations belong to the same geohashed group with this precision. The maximum error with five characters is ±4,89 kilometers (Hill 2017).

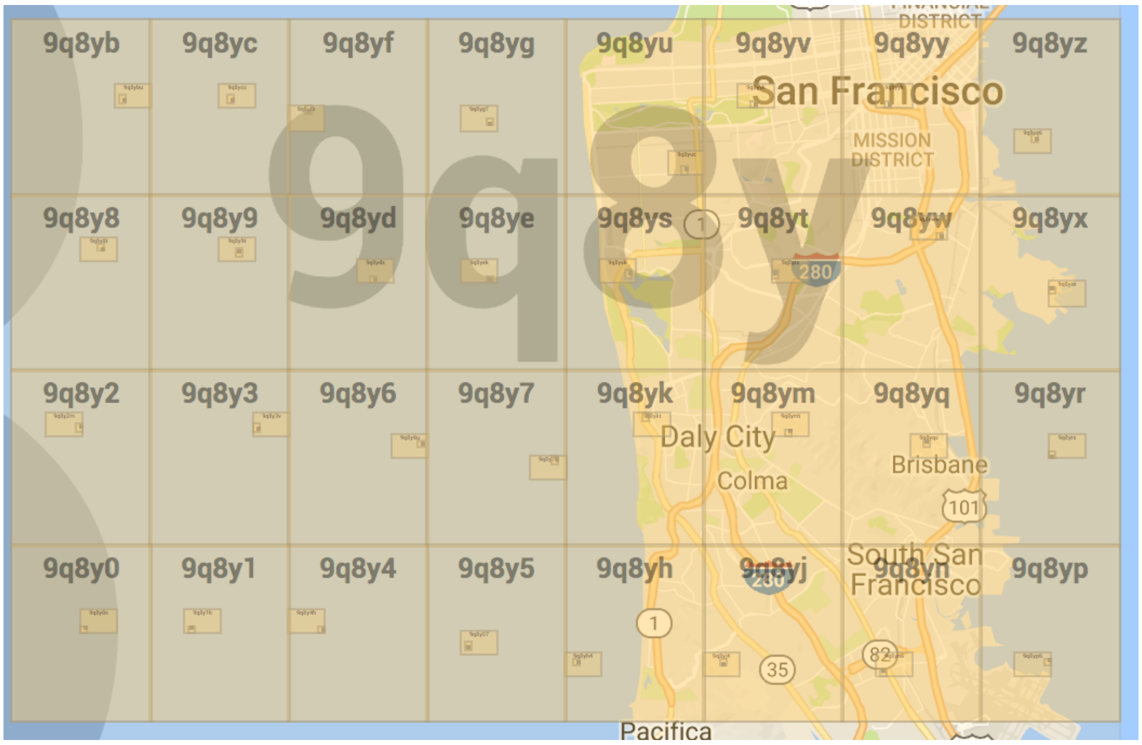


Figure 11. A Geohash grid (Hill 2017).

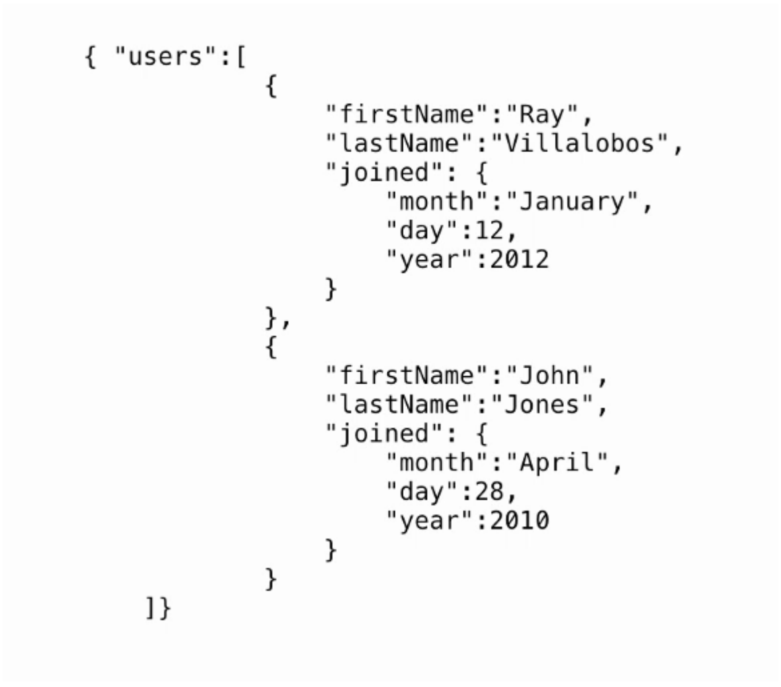
Looking at Figure 11 gives us a better picture of the area grids accomplished with geohashing. Depending on the geohash precision, grid areas are either rectangular or square-like.

5.9 GeoJSON

Based on the popular JSON (JavaScript Object Notation, Figure 12) format, GeoJSON is open standard format for displaying geographical forms and features. A GeoJSON file contains GeoJSON objects, which represent different spatial forms with coordinate mappings.

The standard was first finalized back in 2008, and later in 2016 released as RFC 7946 by the Internet Engineering Task Force (IETF 2016). GeoJSON is supported by many mapping technologies such as Google Earth and Bing Maps.

Within this work the GeoJSON format is used for drawing data layers on top of map services. From these layers the viewer is able to visualize what is happening with the network on that particular location of the map. GeoJSON is utilized by a Python module which enables dumping geospatial data into GeoJSON files with Python commands.



```
{ "users":[
            {
                "firstName":"Ray",
                "lastName":"Villalobos",
                "joined": {
                    "month":"January",
                    "day":12,
                    "year":2012
                }
            },
            {
                "firstName":"John",
                "lastName":"Jones",
                "joined": {
                    "month":"April",
                    "day":28,
                    "year":2010
                }
            }
    ]}
```

Figure 12. JSON objects and their attributes.

5.10 CSV

The comma-separated values (or CSV) file format is a delimited text file format, which uses commas for value delimitation. CSV files are commonly used as a temporary space for data storage, when data is moving between systems. For example, many systems utilize CSV files when exporting out their data. From here the data can be easily imported to any database, as practically all data manipulation libraries support parsing CSV files into databases. Even creating your own routine for CSV file parsing is a relatively easy task!

The CSV file standard is somewhat loosely defined. While there is a defined standard (RFC 4180), even it acknowledges that there are various specifications and implementations of the CSV format. Therefore RFC 4180 is defined by the implementation of CSV format which seems most frequent. This definition includes specifications like:

- *Each record is located on a separate line, delimited by a line break (CRLF). For example:*

  aaa,bbb,ccc CRLF

zzz,yyy,xxx CRLF

- *The last record in the file may or may not have an ending line break. For example:*

  aaa,bbb,ccc CRLF
  zzz,yyy,xxx

- *There may be an optional header line appearing as the first line of the file with the same format as normal record lines. This header will contain names corresponding to the fields in the file and should contain the same number of fields as the records in the rest of the file (the presence or absence of the header line should be indicated via the optional "header" parameter of this MIME type). For example:*

  field_name,field_name,field_name CRLF
  aaa,bbb,ccc CRLF
  zzz,yyy,xxx CRLF

(IETF 2005.)

In this work CSV files are utilized in the same fashion as described before; using them as storage containers when exporting data outside from a system to another. Aside from that, in this work CSV files are also used for storing in-between results, when aggregating data to a higher level. We will be looking into this on a later section.

5.11 Git

The Linux kernel is the most well-known product from Finnish software engineer Linus Torvalds. Not as well-known is that Torvalds is also the original developer of the very widely used version-control system Git. Released on April 7th of 2005, Git allows for tracking changes in computer files within in a project, and also coordinating changes when multiple people are working on the same project. Git's main use today is source code management (SCM or source control).

The development of git began on 2005 when a previously popular SCM system Bit-Keeper's owner started prohibiting the free use of BitKeeper. This saw a urgent need for a new SCM system, as the Linux kernel development was depended on BitKeeper (Barr 2005). The actual starting date for Git development began on 3rd of April 2005 and already on 16th of June, Git managed the release of Linux kernel 2.6.12. Very soon after, Torvalds handed down the maintenance responsibility of Git to Junio Hamano, who saw through the version 1.0 release of Git on 21st of December (Torvalds 2005).

Git's operations mainly revolves around repositories. There are local and remote repositories. A remote repository could be considered as a master repository which reflects the current state of an actual project. Local repositories are clones of the remote repository, and the actual work is done on the local repository (or actually on the working directory which is monitored by the local repository). Once a new feature is implemented on the local repository, it will be pushed on the remote repository for other developers of the project to obtain the new code also.

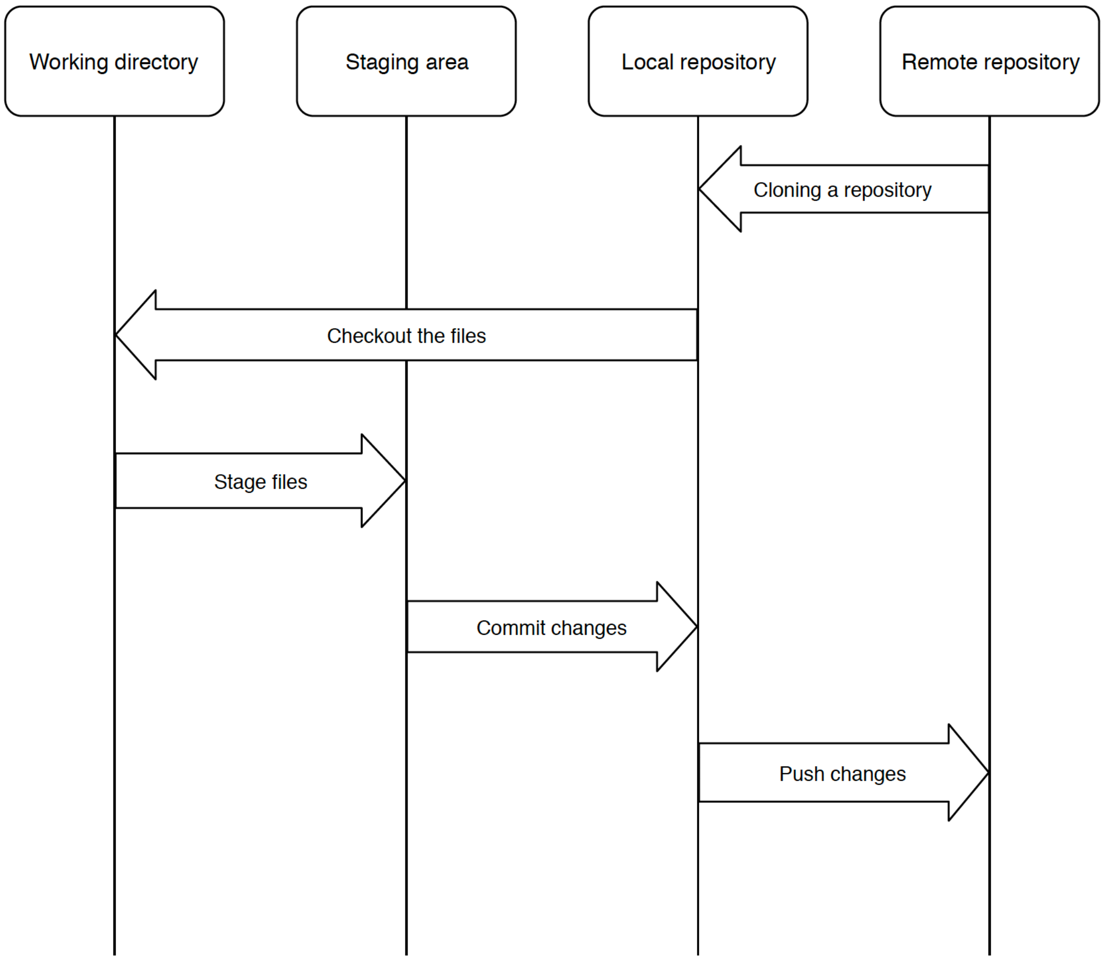


Figure 13. The different logical areas of Git.

Looking at Figure 13, we can grasp the basic flow of Git. If we imagine ourselves to a very common situation, where we are about to participate into the development of an already existing group effort, the first step is to clone the remote repository into a local one. At the same time when the repository is cloned, files from the local repository are checked out into a working directory, where they are visible for the local file explorer. The local repository itself is hidden under the root of the working directory in a directory named *.git*. The next natural step is to start working the files and start implementing changes. This is where the staging area jumps in. When we start making changes into files that are monitored by the local repository (i.e. files that already belong to the local repository), they are elevated into a stage which is between the working directory and local repository. When completely new files are created on the working directory, initially they are not automatically added to the staging area (or tracked by Git). Instead they have to be manually added a Git specific command. When the changes are ready, they can be committed to the local repository through a Git command. Once the implementation of a new feature in the code is ready, it can be pushed and merged with a Git command to the remote repository for the viewing of others.

The previous description of working with Git was of course overly simplified and it does not account for things like merge conflicts and repository branching. Merge conflicts can occur when pulling the latest changes from a remote repository to your own local repository. If the remote repository contains new changes in files which contain different kind of changes on the user's local repository, conflicts can occur. Git is able to solve minor conflicts quite sophisticatedly by itself, but situations where Git doesn't know what to do occur also. During these situations the user has to manually aid Git in deciding which merge conflict changes are kept and which are replaced by the remote changes. Branching is a concept where in essence alternate versions of the repository exists parallel in the same repository. Only one branch can be active at a time and files will be checked out from the active branch to the working directory. Commits only affect the active branch, so a different branch can have a totally different state. Branching is a quite common way of working, as making changes or pushing changes straight to the remote master branch is generally viewed as very bad practice. It is recommended that local changes are first made to a new branch and this branch is then pushed also to the remote repository. After when other working members of the project have compared the branch against the current master branch, upon approval the branch can be merged with the master branch.

For remote repositories, a Git server is needed. Git servers can be created out-of-box with just Git itself, and there are also many open source and proprietary solutions for this. The most popular Git server platform is GitHub and Bitbucket, which are both proprietary solutions.

In this project, Git is mainly used for having a backup on the remote repository, which is located in Elisa’s own GitHub Enterprise platform. As I am currently the sole contributor on this work, source code management isn’t needed in the same way as with group efforts. Besides having a backup, it is also a good feature to be able to rewind commits, if something goes very fundamentally broken with a new commit within this project. I will also mention that even this paper is stored in a separate remote Git repository for backup purposes!

6 USE CASE IMPLEMENTATION

In this chapter we will take a look at the actual implementation made for this use-case PoC. On this chapter we will start things by going through the underlying software stack of this project. After this I will explain the project architecture in more detail, meaning that we will go through each individual component and how they interconnect with each other. After this I will have a few words about setting the working environment for this project, and we will go through the project in even more detail, taking a closer look at the project in code-level.

6.1 Software architecture

As previously mentioned, this chapter will focus on the project architecture, meaning that we will take a look at each component and at how they connect and cooperate with each other. The best way to go through with this, is to start off with Figure 14:

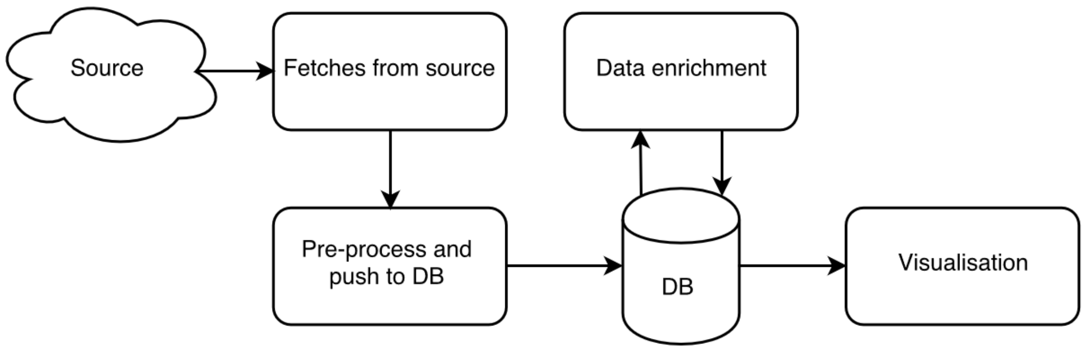
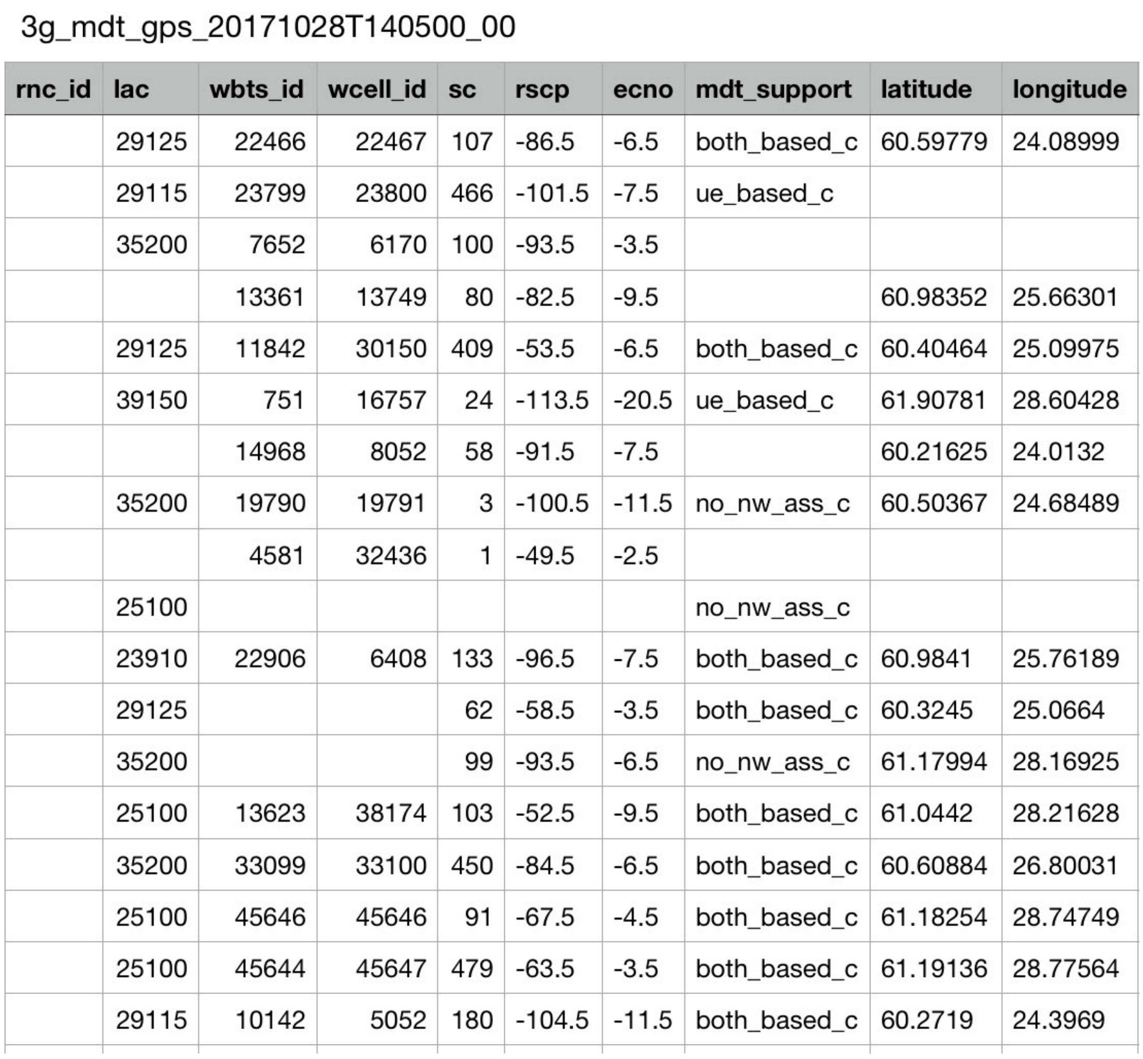


Figure 14. Project architecture.

Shortly explaining the above picture in its entirety, the process starts off with fetching the needed data from the data source. In this context, the data source is just a folder containing test run data spanning from one and a half days. Data is fetched from one whole day, preprocessed, and pushed to the local database. After the initial database push, the pushed data is exported for data enrichment. After this post-process, visualization data is created based on the just created enriched data. The next section will detail these actions in more detail.

6.1.1 Data source

As already discussed in section 5.4, Nokia's L3 Data Collector (MegaMon) is data collecting entity, which collects several types of signaling measurements between Node B and RNC equipment. These measurements are then passed on to Nokia's L3 Data Analyzer (Emil), which then parses the collected data from binary-coded data into human-readable CSV files. In this context we are only interested in GPS data, so we need to only access the files which start with the filename *3g_mdt_gps_*. These files offer a variety of data (Figure 15), but there are four columns, in which particular we are interested in. These columns are *sc* (scrambling code), *rscp* (received signal code power), *latitude*, and *longitude*.


3g_mdt_gps_20171028T140500_00

rnc_id	lac	wbts_id	wcell_id	sc	rscp	ecno	mdt_support	latitude	longitude
	29125	22466	22467	107	-86.5	-6.5	both_based_c	60.59779	24.08999
	29115	23799	23800	466	-101.5	-7.5	ue_based_c		
	35200	7652	6170	100	-93.5	-3.5			
		13361	13749	80	-82.5	-9.5		60.98352	25.66301
	29125	11842	30150	409	-53.5	-6.5	both_based_c	60.40464	25.09975
	39150	751	16757	24	-113.5	-20.5	ue_based_c	61.90781	28.60428
		14968	8052	58	-91.5	-7.5		60.21625	24.0132
	35200	19790	19791	3	-100.5	-11.5	no_nw_ass_c	60.50367	24.68489
		4581	32436	1	-49.5	-2.5			
	25100						no_nw_ass_c		
	23910	22906	6408	133	-96.5	-7.5	both_based_c	60.9841	25.76189
	29125			62	-58.5	-3.5	both_based_c	60.3245	25.0664
	35200			99	-93.5	-6.5	no_nw_ass_c	61.17994	28.16925
	25100	13623	38174	103	-52.5	-9.5	both_based_c	61.0442	28.21628
	35200	33099	33100	450	-84.5	-6.5	both_based_c	60.60884	26.80031
	25100	45646	45646	91	-67.5	-4.5	both_based_c	61.18254	28.74749
	25100	45644	45647	479	-63.5	-3.5	both_based_c	61.19136	28.77564
	29115	10142	5052	180	-104.5	-11.5	both_based_c	60.2719	24.3969

Figure 15. An excerpt from a CSV file featuring GPS measurements.

The data set provided for this use-case consists of a test run trying out MegaMon and Emil. During this test run about a one and a half days' worth of data was gathered from one RNC.

6.1.2 Data fetch and initial DB push

As just stated, in the context of this use-case, we are only interested in GPS related measurements. Therefore we only need to access measurement data files which contain the GPS measurements, and these files are the ones which start with the filename *3g_mdt_gps_*. The main component in this fetch and push implementation is done with Pandas. Responsible for the data handling, Pandas goes through one entire file, fetches only the wanted columns, drops rows with any empty data, and pushes the data into the database. Data in the raw data table looks as following:

```
MariaDB [usecase]> select * from mdt_3g_gps limit 25;
+----+------+--------+----------+-----------+---------+
| id | sc   | rscp   | latitude | longitude | geohash |
+----+------+--------+----------+-----------+---------+
|  1 |  107 |  -86.5 |  60.5978 |     24.09 | udc0sp  |
|  2 |   80 |  -82.5 |  60.9835 |    25.663 | udf4zv  |
|  3 |  409 |  -53.5 |  60.4046 |   25.0998 | ud9zeh  |
|  4 |   24 | -113.5 |  61.9078 |   28.6043 | ueh24v  |
|  5 |   58 |  -91.5 |  60.2163 |   24.0132 | ud9nd6  |
|  6 |    3 | -100.5 |  60.5037 |   24.6849 | udc81w  |
|  7 |  133 |  -96.5 |  60.9841 |   25.7619 | udf6fj  |
|  8 |   62 |  -58.5 |  60.3245 |   25.0664 | ud9z4m  |
|  9 |   99 |  -93.5 |  61.1799 |   28.1693 | uduh11  |
| 10 |  103 |  -52.5 |  61.0442 |   28.2163 | udu560  |
| 11 |  450 |  -84.5 |  60.6088 |   26.8003 | udg0cc  |
| 12 |   91 |  -67.5 |  61.1825 |   28.7475 | udukn1  |
| 13 |  479 |  -63.5 |  61.1914 |   28.7756 | udukng  |
| 14 |  180 | -104.5 |  60.2719 |   24.3969 | ud9qgh  |
| 15 |  391 |  -69.5 |  60.7597 |   24.7784 | udc9eu  |
| 16 |  410 |  -97.5 |  60.2907 |   24.4669 | ud9quz  |
| 17 |  439 |  -96.5 |  60.2678 |   24.4387 | ud9qu5  |
| 18 |  331 |  -86.5 |  60.3311 |   24.8608 | ud9xjw  |
| 19 |   33 |  -80.5 |  60.7518 |   26.0575 | udf98g  |
| 20 |  131 |  -75.5 |  60.4829 |   25.0758 | udcb4d  |
| 21 |  493 |  -86.5 |  60.9878 |   25.5333 | udf4vn  |
| 22 |  210 | -101.5 |  60.6015 |   24.6847 | udc8c8  |
| 23 |  132 |  -78.5 |  60.5911 |   24.7993 | udc8sq  |
| 24 |    4 |  -87.5 |  60.3888 |    25.705 | uddr8c  |
| 25 |  282 |  -62.5 |   60.625 |   24.8252 | udc8uu  |
+----+------+--------+----------+-----------+---------+
```

6.1.3 Data enrichment

In its initial form the data isn't suited for our particular use-case. That's why the data has to be enriched, or to be more specific with this context, the data needs to be aggregated.

As we are primarily interested in the sums of measurements within an area instead of the actual individual measurements, the data can be aggregated to a different database table, which already contains the wanted numbers for each area. This way performance resources won't be wasted on calculating the area specific numbers from raw data each time, when a client requests to see the visualizations, and having the aggregations stored on a different table takes very little extra space. As there is need for calculating an arithmetic mean from the signal strengths per area, and the most frequent primary scramble code per area, the aggregation process isn't exactly same for both numbers, but they only differ within the steps of each process. These processes will be looked in more detail in subsection 6.3.3.

It should be mentioned here that the aggregated table contains columns named as *geohash*, *sc_top*, *rscp_avg*, and *rscp_grade*. Shortly explaining these columns, *geohash* of course is the ID for each respective area, *sc_top* contains the most frequent primary scrambling code for that area, *rscp_avg* shows the average signal strength in that area, and *rscp_grade* holds the grade for that area based on the average signal strength. Grading is done with the following specs:

- Signal strengths that are equal to or more than -90 dBm, are considered as good in quality and get graded as *green*.
- Signal strengths that are less than -90 dBm or more or equal to -100 dBm, are considered as average in quality and get graded as *orange*.
- Signal strengths that are less than -100 dBm, are considered poor in quality and get graded as *red*.

Data in the aggregated data table looks as following:

```
MariaDB [usecase]> select * from mdt_3g_gps_agg_day limit 25;
+----+---------+--------+----------+------------+
| id | geohash | sc_top | rscp_avg | rscp_grade |
+----+---------+--------+----------+------------+
|  1 | 7zzzzz  |      6 |    -90.9 | orange     |
|  2 | ud8y7m  |     31 |    -92.5 | orange     |
|  3 | ud8y7q  |     31 |      -98 | orange     |
|  4 | ud8y7t  |     31 |    -98.2 | orange     |
|  5 | ud8y7v  |    225 |    -90.1 | orange     |
|  6 | ud8y7w  |     31 |    -92.5 | orange     |
|  7 | ud8ydp  |    187 |   -102.5 | red        |
|  8 | ud8ydr  |    187 |     -104 | red        |
|  9 | ud8ye3  |    490 |    -79.9 | green      |
| 10 | ud8ye9  |    490 |    -79.5 | green      |
| 11 | ud8yke  |     27 |      -65 | green      |
| 12 | ud8ykf  |     27 |    -60.5 | green      |
| 13 | ud8ykj  |    225 |    -93.4 | orange     |
| 14 | ud8ykm  |    225 |    -92.3 | orange     |
| 15 | ud8yks  |    225 |    -67.4 | green      |
| 16 | ud8ykt  |    225 |    -75.8 | green      |
| 17 | ud8ynp  |     79 |    -81.6 | green      |
| 18 | ud8yu9  |    490 |    -99.5 | orange     |
| 19 | ud8yub  |    307 |    -86.4 | green      |
| 20 | ud8yvw  |     73 |    -96.5 | orange     |
| 21 | ud8yxn  |     28 |    -86.5 | green      |
| 22 | ud8yxw  |    141 |    -81.5 | green      |
| 23 | ud8z7g  |      7 |    -79.5 | green      |
| 24 | ud8zju  |    400 |    -81.8 | green      |
| 25 | ud8zjv  |     18 |    -88.2 | green      |
+----+---------+--------+----------+------------+
```

6.1.4 Data visualization

After the data has been aggregated into a separate table, the aggregated data can be used for building files for visualization. The needed data for signal strengths and primary scrambling codes are fetched with two separate runs. Starting with signal strengths, the first step is to generate three separate GeoJSON files. These files are known as the green, orange, and red files. As one can guess, measurements are filtered into these three files based on their grading. These files are used by Plotly for drawing measured areas as seen in Figure 6, as the files contain all the coordinate information. The GeoJSON files contain a single JSON object for each area, and each JSON object contains properties which are the bounding coordinates of the area. The color layering itself is done with Plotly.

The process for visualizing primary scrambling codes is somewhat different. Reading the data from the aggregated data produces only a single GeoJSON file. This file also contains GeoJSON objects per area, but an object contains not only bounding coordinates for the area, but also the color information for that area along with the primary scrambling code. At the time of writing this, Plotly does not scale well with an increased amount of layers, therefore the visualization with this has been implemented with deck.gl.

6.2 Setting up the environment

Before we can run the project, the needed environment needs to be setup, so the project can run. Development was started on Windows 10 under a WSL (Windows Subsystem for Linux), but the project was later migrated to macOS 10.13.6 (High Sierra). The migration itself brought no extra steps, other than setting up the environment back up once more. This section will depict setting the environment up within macOS.

6.2.1 Python and Python libraries

MacOS operating systems come pre-installed with Python 2.7, but as the development scene is trying to swiftly move towards newer iterations of Python (3.6 and 3.7), installing Python3 is the first step here. The preferred way to install Python3, is with Brew, which is packet manager for macOS. Installation is simply done by typing `brew install python`, which installs Python3 and pip3, the packet manager for Python3 libraries. After Python3 and pip3 are installed, we want to install the virtual environment module, for creating and managing isolated Python environments. This is simply done with `pip install virtualenv`. After this we can move to our project root folder and type the command `python -m venv .`, which initiates a new virtual environment in the current folder. Now we can start listing the needed 3rd party libraries, and the preferred way is to create a file named *requirements.txt*, which holds the names and versions of our needed libraries. For example, here is a snippet of the contest of this project's *requirements.txt*:

```
geohash2==1.1
geojson==2.4.1
mysql==0.0.2
mysql-connector==2.1.6
pandas==0.23.4
sqlalchemy==1.2
```

Although the database itself will be inside a Docker container, at this point we will need to install the MySQL daemon, as the *mysql* and *mysql-connector* libraries require it for interfacing with MySQL databases, which means that it is needed for MariaDB interfacing. The MySQL daemon is simply installed with `brew install mysql`. At this point we can finally install the needed Python libraries for this virtual environment. First we need to activate the virtual environment by typing in the project root folder `source bin/acti-vate` and then install libraries with `pip install -r requirements.txt`, which reads the required libraries straight out of the *requirements.txt* file.

6.2.2 Docker and MariaDB

We could very well just use a database which sits on top of the OS, but for the sake of modularity and mobility, we are adding an extra layer by containerizing our MariaDB instance. This project has been designed to automatically initiate the database, so we can create and destroy our database instances as much as we want, without having to repeat setting up the database manually everytime.

Docker is installed by downloading an installer from Docker's official website. This project uses the Community Edition of Docker. After docker is up and running, we simply pull a MariaDB image with the command `docker pull mariadb:latest`. This downloads an official MariaDB image from the Docker repository, which can be used to initiate MariaDB instances (or containers). The string *latest* is a tag which points to the latest build of an image. We start a MariaDB container with the following command:

```
docker run --rm --name mariadb_poc --env MYSQL_ROOT_PASSWORD=<root_password> -
-detach --publish 3306:3306 mariadb
```

Going through this command, *docker run* is for creating a container from an image. The flag –rm sets the container to automatically remove itself when it's stopped, --name flag sets the container's name manually (by default they are drawn randomly from a list), the --env flag sets an environment variable inside the container (in this case we are setting the database root user password), --detach flag specifies that the container is run as a background process, the --publish flag forwards a local port towards a port inside the container (port 3306 is the MySQL/MariaDB default). The final argument is the name of the Docker image which from the container is created.

6.3 Project walkthrough

After the virtual environment and database are erected, the project is now ready for running. In this section we will take a detailed look at the project by going through snippets of code and flowcharts.

6.3.1 Initiating the database

As mentioned in subsection 6.2.2, with every run of the program, the database and its tables are created, if they do not exist already. For this we use the following function:

```python
def initialize_db():
    try:
        try:
            base_sql_engine =
create_engine('mysql+mysqlconnector://root:<root_password>@localhost:3306')
            conn = base_sql_engine.connect()
            conn.execute(db_schema.CREATE_DATABASE)
        finally:
            conn.close()
        try:
            conn = SQL_ENGINE.connect()
            conn.execute(db_schema.CREATE_RAW_TABLE)
            conn.execute(db_schema.CREATE_AGGREGATED_TABLE)
        finally:
            conn.close()
    except:
        raise ConnectionRefusedError('Failed to connect to database')
```

This function logins into MariaDB and creates the target database (if it does not exist already). After this the program logins into the database and creates the needed tables (if they do not exist already). This function is wrapped around with a *try-except* clause, which is Python's method for error handling. If an error occurs when performing code around a *try* clause, the *except* clause is checked for expected error objects and then code around the *except* clause is executed, if it is catch. The *except* clause in this function catches all errors, as no specific error is specified to it. If something goes wrong, a *ConnectionRefusedError* class raised and an error message is logged. This function also contains two *try-finally* clauses nested within the *try-except* clause. The *finally* clause is always executed, regardless of failure or success of its respective *try* clause. With these two clauses, we want to ensure that the database connection is always closed, so the database won't be clogged with dead connections.

Connection to the database is established with the Python sqlalchemy library's *create_engine()* function, which creates an engine for executing the SQL queries towards the database. Engines are created with connection strings which are formed with a database URL like: *dialect+driver://username:password@host:port/database* (SQLAlchemy 2018). In our case, the used dialect is *mysql* and the connecting driver is *mysqlconnector*. Initially we are connecting to the root of the MariaDB host to create the database, and for the second connection string the database is also defined. The second connection string is defined as global string at the start of the program, as it will be used throughout the program.

The executed SQL queries are defined in an outside file named *db_schema.py*, which contains the schemas for creating the database and its tables. These schemas are:

```
CREATE_DATABASE = '''CREATE DATABASE IF NOT EXISTS usecase;'''

CREATE_RAW_TABLE = '''
CREATE TABLE IF NOT EXISTS mdt_3g_gps(
id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
sc INT NULL,
rscp DOUBLE NULL,
latitude FLOAT NULL,
longitude FLOAT NULL,
geohash varchar(10) NULL);
'''

CREATE_AGGREGATED_TABLE = '''
CREATE TABLE IF NOT EXISTS mdt_3g_gps_agg_day(
id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
geohash varchar(10) NULL,
sc_top INT NULL,
rscp_avg DOUBLE NULL,
rscp_grade varchar(8) NULL);
'''
```

The raw data is pushed to the table *mdt_3g_gps*. This table has all the same columns which are fetched from the source files as described in subsection 6.1.1. A column for the geohashes is also created, as geohashes are encoded from coordinates at the same when pushing the raw data to this table. The table *mdt_3g_gps_agg_day* is home for the aggregated data, and it contains columns for the geohashes, most frequent primary scrambling codes per geohash, the average signal strength per geohash, and the signal strength grade per geohash. Each table also contains a column for record ID, which is also the primary key for each record. All three SQL queries contain the *IF NOT EXISTS* clause, which means that the database or table will not be created if it already exists. Without this clause, the program would straight crash due to already existing database and tables.

6.3.2 Inserting raw data

After the database and its tables are initiated, we can start pushing data to them. The first step is to gather a list files and their full paths, which contain the raw data we want to push into our raw data table. After a list has been gathered, each item in the list is passed through this function:

```
def init_push_to_db(file):
    logging.info('Pushing file %s to DB.', file)
    for df in pd.read_csv(file, usecols=["sc", "rscp", "latitude",
"longitude"], compression='gzip', chunksize=CHUNKSIZE):
        geolist = []
        df = df.dropna().round({'latitude': 8, 'longitude': 8})
        for id, row in df.iterrows():
            geolist.append(geohash2.encode(row['latitude'], row['longitude'],
precision=GEOH_PREC))
        df['geohash'] = geolist
        df.to_sql('mdt_3g_gps', SQL_ENGINE, if_exists='append', index=False)
    logging.info('Pushed file %s.', file)
```

This is the first function (of many) to utilize Pandas. The CSV file is read into a Pandas DataFrame data structure. This is accomplished with the DataFrame's built-in method *read_csv()*. With this method we can filter in only the necessary columns, read from compressed files, and perform operations in smaller chunks. The argument *chunksize* defines how many rows from the source will be iterated before moving onto the next chunk of rows. Between swapping chunks the data is preprocessed and then pushed into the database. Preprocessing operations include rounding coordinate values, as there is a bug on Nokia's side, which can make some values several digits too long. All rows with null values are dropped, since we can't utilize records with any null values at all. At the same time an empty list named *geolist* is created, in which row by row geohashes are inserted. This is done utilizing the Python geohash library's *encode()* function. Since geohashes are generated in the same order as they appear in the source data, the *geolist* list can be inserted into to the DataFrame as-is, as there is no sorting functions performed at this point.

6.3.3 Aggregating data

Now the raw data is safe within the database. This means that the next step is to enrich data to be more suitable for our use-case. There are several ways we could approach the method for aggregating data. First option would be to perform the aggregation in-place within the database with SQL. However, SQL does not scale well with growing amounts of data, so performance would be poor, and it is wiser to save computing resources for answering incoming queries towards the database. The second option would be to download the data to an external machine and perform the aggregation with data manipulation tools like Pandas. However, using Pandas also brings a problem, which is the issue of Pandas being heavy memory-wise, and these are very large datasets we are processing. This is why we to implement a technique called *data sharding*. With sharding we divide the dataset into multiple shards, place the shards temporarily on the hard drive, and start aggregating them, while placing all intermediate results onto the hard drive also. As seen in Figure 16, aggregation rounds are performed recursively as long as we get results into one file.

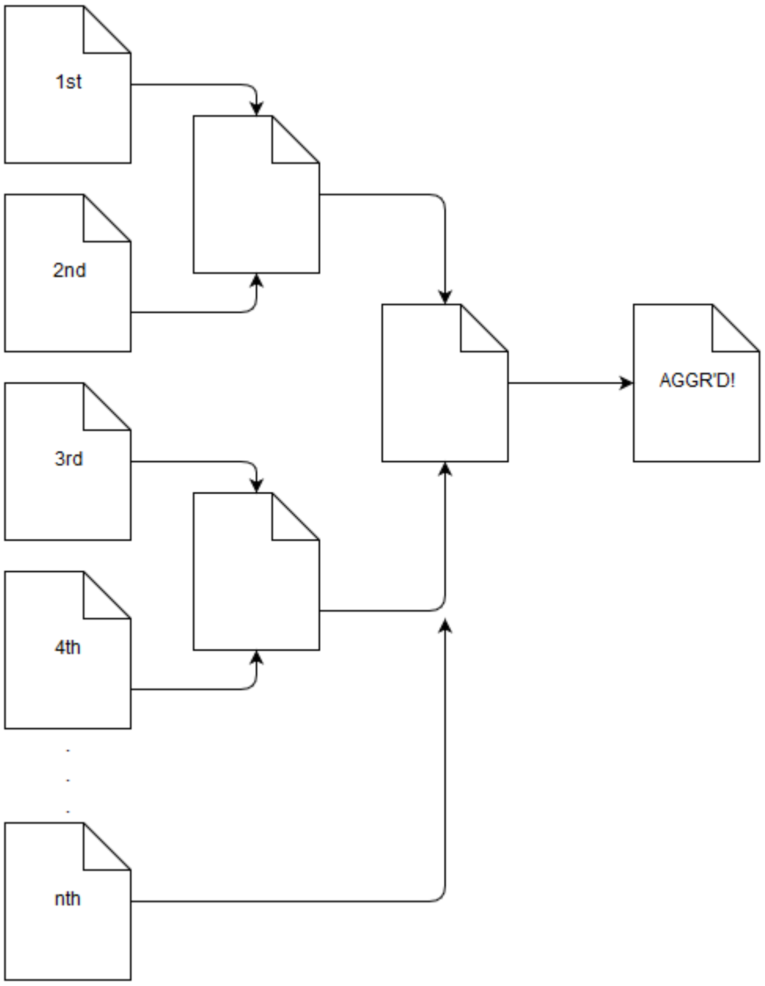


Figure 16. Sharding and aggregating the data.

The logic flow of the aggregation processes can be seen in Figure 17:

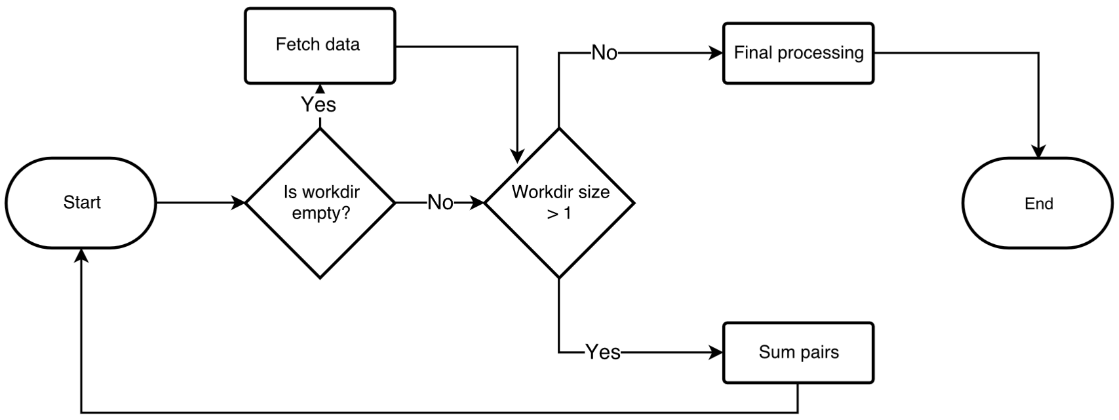


Figure 17. Flowchart for the aggregation processes.

As mentioned earlier in subsection 6.1.3, the aggregation processes for the average signal strength and most frequent primary scrambling codes are not exactly the same, but the data sharding flows presented in Figure 16 and Figure 17 are same for both. Differences come in how the intermediate results are aggregated before merging. Starting with the signal strength aggregation, let’s take a look at the following code:

```
def fetch_rscp():
    for i, df in enumerate(pd.read_sql_table('mdt_3g_gps', SQL_ENGINE,
chunksize=CHUNKSIZE, columns=['geohash', 'rscp'])):
        df = df.groupby('geohash',as_index=False)['rscp'].agg(['sum','count'])
        df.to_csv('temp/mean_' + str(i) + '.csv')
```

If the temporary working directory is empty, we have to start with fetching the needed data from the database. Data is fetched within chunks, and each chunk iteration is dumped onto the hard drive into its own CSV file. Initially the chunk is read into a DataFrame by performing a Pandas *groupby().agg.()* function. This function first groups all the measurements into groups of geohashes. At the same time for each geohash group, the column rscp (which holds signal strength values) is split into two new columns: *sum* and *count*. The *sum* column holds the sum for all RSCP measurements in geohash group and the *count* column holds the amount of measurements. The DataFrame is finally written out into CSV file, and the next chunk iteration is processed. When all chunk iterations are done, we have our first round of intermediate results, and we can move onto this code:

```
df1 = pd.read_csv(dirlist[i])
df2 = pd.read_csv(dirlist[i+1])
agg = pd.concat([df1, df2]).groupby('geohash', as_index=False).sum()
agg.to_csv('temp/mean_' + str(i) + '_' + str(level) + '.csv', index=False)
```

As long as the temporary working directory is holding more than one file, the above piece of code is performed. We read a pair of CSV files into two separate DataFrames and initiate a third one. The third DataFrame is an aggregation of the two previous ones, where we first concatenate the two DataFrames together, then perform a Pandas *groupby().sum()* function, which sums all the values within a group (the group being defined by the geohash in this case). Finally the DataFrame is stored on the hard drive in a new CSV file, and the code moves onto the next pair.

When the last file remains, we move onto this code:

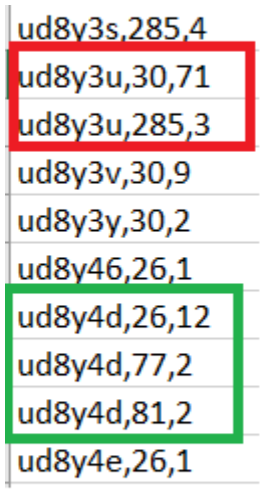
```
with open('output/mean_agg.csv', 'w') as file:
    file.write('geohash,rscp_avg,rscp_grade\n')
for df in pd.read_csv(dirlist[0], chunksize=CHUNKSIZE):
    df['mean'] = df['sum'] / df['count']
    df['mean'] = df['mean'].round(1)
    df.loc[df['mean'] >= -90, 'grade'] = 'green'
    df.loc[(df['mean'] < -90) & (df['mean'] >= -100), 'grade'] = 'orange'
    df.loc[df['mean'] < -100, 'grade'] = 'red'
    df.to_csv('output/mean_agg.csv', index=False, columns=['geohash', 'mean',
'grade'], header=False, mode='a')
```

In essence, what this code is doing, it creates a new CSV file outside of the temporary working directory and starts reading the aggregated data from the last intermediate result file by chunks into a DataFrame. With every chunk iteration, the quotients of the *sum* column divided by the *count* column are inserted into a new column named *mean*. The arithmetic mean is then graded into a new column named grade, as specified in subsection 6.1.3. Finally the chunk iteration is appended into the final CSV file before moving to the next chunk.

Next explaining how aggregation is done with the primary scrambling codes, we start with this piece of code:

```
def fetch_sc():
    for i, df in enumerate(pd.read_sql_table('mdt_3g_gps', SQL_ENGINE,
chunksize=CHUNKSIZE, columns=['geohash', 'sc'])):
        df = df.groupby(['geohash', 'sc'],as_index=False)['sc'].agg(['count'])
        df.to_csv('temp/freq_' + str(i) + '.csv')
```

Again, if the temporary working directory is empty, the needed data has to be fetched from the database. The process is quite similar as with aggregating signal strengths, but this time we are grouping measurements with both the *geohash* and *sc* columns. This means that Each geohash group lists every primary scrambling code belonging into it. At the same time we also keep a count of the amount of different measured primary scrambling codes in each geohash group. If we would take a look inside an intermediate result CSV file, each row would contain data in this order: geohash, primary scrambling code in that geohash group, count of that primary scrambling code in that geohash group (Figure 18).



```
ud8y3s,285,4
ud8y3u,30,71
ud8y3u,285,3
ud8y3v,30,9
ud8y3y,30,2
ud8y46,26,1
ud8y4d,26,12
ud8y4d,77,2
ud8y4d,81,2
ud8y4e,26,1
```

Figure 18. A snippet of rows in an aggregated primary scrambling code intermediate result file.

When the first round of primary scrambling code aggregation is done (i.e. data is fetched and aggregated once from the database), we get to this point:

```
df1 = pd.read_csv(dirlist[i])
df2 = pd.read_csv(dirlist[i+1])
agg = pd.concat([df1, df2]).groupby(['geohash', 'sc'], as_index=False).sum()
agg.to_csv('temp/freq_' + str(i) + '_' + str(level) + '.csv  ', index=False)
```

As with signal strength aggregation, from one CSV file pair each file is read into its own DataFrame and the data is merged into a third DataFrame. With primary scrambling codes this process is a bit simpler, as now we only have to sum the count of the primary scrambling codes in each geohash groups. Once again, when data is merged into the DataFrame, it is written out into a new CSV file, and we move onto the next file pair. When we get to the round with only one file, this code is performed:

```
with open('output/freq_agg.csv', 'w') as file:
    file.write('geohash,sc_top\n')
df = pd.read_csv(dirlist[0])
df = df.sort_values(by=['geohash', 'count'], ascending=[True, False])
df = df.drop_duplicates(subset='geohash')
df.to_csv('output/freq_agg.csv', index=False, columns=['geohash', 'sc'],
header=False, mode='a')
```

As with signal strength aggregation, as final CSV file is created outside the temporary working directory. Initially data from the last intermediate result CSV file is read into a DataFrame, and the DataFrame is sorted so that each geohash group lists the primary scrambling code with most counts as the first item in that geohash group. After this a *drop_duplicates()* function can be performed on each geohash group. By default, that function drops all duplicates which came after the first item, meaning that after deduplication we only have rows which show the most frequent primary scrambling code in each geohashed area. The final aggregation is done in one single chunk, as performing this in several chunks poses a risk that all rows belonging to a certain geohash group won't make it to the same chunk.

Now we have two CSV files in which one contains the average signal strength in each geohash area, and the other contains the most frequent primary scrambling code in each geohash area. At this point this function is performed:

```
def agg_to_db():
    df_a = pd.read_csv('output/mean_agg.csv')
    df_b = pd.read_csv('output/freq_agg.csv')
    df_a = df_a.merge(df_b, on='geohash')

    df_a.to_sql('mdt_3g_gps_agg_day', SQL_ENGINE, if_exists='append',
index=False)

    os.remove('output/mean_agg.csv')
    os.remove('output/freq_agg.csv')
```

The two CSV files are read into separate DataFrames. The data from the second DataFrame is merged into the first DataFrame by defining the *geohash* column as the common column. Now we finally have all the aggregated data in one data structure, which can then be pushed to the database. Finally the two CSV files are removed as the enriched data is now safe within the database.

6.3.4 Generating GeoJSON files

Now that we have the aggregated data for building our visualization, we still need to generate GeoJSON file out of this data. As with data aggregation, the process for signal strengths and primary scrambling codes is not exactly the same. As mentioned in subsection 6.1.4, visualizing average signal strengths requires three separate GeoJSON files. The process for this is quite simple, data is just filtered out into the three separate files based on the measurement grading. With primary scrambling codes, all the aggregated measurements are just dumped into one single file. However, before generating all the needed files, the geohashes associated with the measurements need to be converted into GeoJSON objects. For this task we have the following function:

```
def get_polygon(geohash, sc=None):
    bbox = geohash2.decode_exactly(geohash)
    width = bbox[2]
    height = bbox[3]
    polygon = geojson.Polygon([[(bbox[1] - height, bbox[0] - width),
                                (bbox[1] - height, bbox[0] + width),
                                (bbox[1] + height, bbox[0] + width),
                                (bbox[1] + height, bbox[0] - width),
                                (bbox[1] - height, bbox[0] - width)]])

    if sc != None:
        return geojson.Feature(geometry=polygon,
                               properties={'sc': sc, 'color':
colors.hex_to_rgb(colors.get_scl(sc))})
    else:
        return geojson.Feature(geometry=polygon)
```

For this we utilize the function *decode_exactly()*, which is found from the Python geohash library. This functions returns a tuple (a Python data structure, which is an immutable list), which looks like this:

```
>>> print(geohash2.decode_exactly('ezs42'))
(42.60498046875, -5.60302734375, 0.02197265625, 0.02197265625)
```

The first two items are the latitude and longitude of the exact center of the geohash. The latter two items of the tuple are the error margins of the latitude and longitude. With these values we can create a bounding box of the geohash. The created bounding box equals to the borders of the geohash. Looking at Figure 19 gives us a better idea how the center point and error margins are utilized. Once we now the border distances, we can calculate the corner coordinates of each corner of the bounding box. From these coordinates we can create a GeoJSON Polygon object, which is given the corner coordinates of the bounding box.

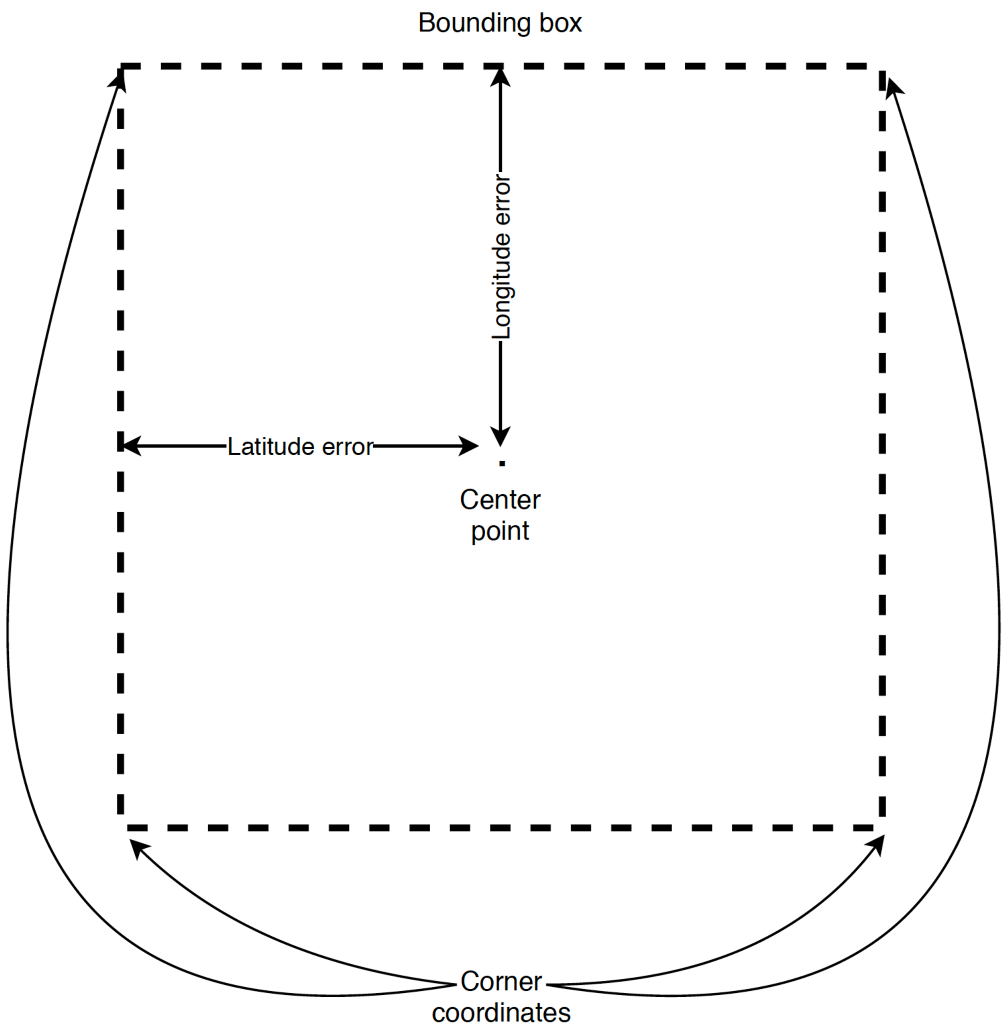


Figure 19. A coordinate and its bounding box.

The Polygon object is returned to a Pandas DataFrame, which contains signal strength gradings for geohashes, and the Polygon object is written into one of the three GeoJSON files depending on its grading. A single Polygon object looks like this:

```
{
    "geometry": {
        "coordinates": [
            [
                [
                    23.697509765625,
                    60.2105712890625
                ],
                [
                    23.697509765625,
                    60.216064453125
                ],
                [
                    23.70849609375,
                    60.216064453125
                ],
                [
                    23.70849609375,
                    60.2105712890625
                ],
                [
                    23.697509765625,
                    60.2105712890625
                ]
            ]
        ],
        "type": "Polygon"
    },
    "properties": {},
    "type": "Feature"
},
```

Coordinates are listed in order of drawing. Plotly starts drawing the borders starting from the first coordinate pair, moving on the second pair, and so on. There are five items listed, as drawing needs to end at the same point where it started from.

Primary scrambling code related Polygon objects are enriched with a bit more data. The *properties* attribute is assigned information about the actual primary scrambling code and it's also assigned a color based on the scrambling code value. The color is determined from a predetermined list which contains exactly 512 different color values. A color value is simply picked accessing the same index value as the scrambling code value (scrambling codes also start from value 0). The color value is converted a RGB value and then assigned to the Polygon object. A single Polygon object in this context looks like:

```json
{
    "geometry": {
        "coordinates": [
            [
                [
                    23.697509765625,
                    60.1885986328125
                ],
                [
                    23.697509765625,
                    60.194091796875
                ],
                [
                    23.70849609375,
                    60.194091796875
                ],
                [
                    23.70849609375,
                    60.1885986328125
                ],
                [
                    23.697509765625,
                    60.1885986328125
                ]
            ]
        ],
        "type": "Polygon"
    },
    "properties": {
        "color": [
            255,
            241,
            194
        ],
        "sc": 31
    },
    "type": "Feature"
},
```

6.3.5 Visualization with maps

Now we have finally arrived at the point where the actual visualization happens. As mentioned before, the signal strength visualization is implemented alone with Plotly's Dash. The code for this lies in a separate Python file. This file creates a Dash app which uses a Plotly map figure object as the map object. The three GeoJSON files are read into three separate JSON objects, which are further utilized by the Plotly figure object. When the Python file is run, it creates web page viewable by web browsers at localhost on port 8050. The code won't be looked in detail here, except for this part:

```
layers=[
    dict(
        sourcetype = 'geojson',
        source = green_file,
        type = 'fill',
        color='rgb(0, 255, 0)',
        opacity=settings['opacity']
    ),
    dict(
        sourcetype = 'geojson',
        source = orange_file,
        type = 'fill',
        color='rgb(255, 127, 0)',
        opacity=settings['opacity']
    ),
    dict(
        sourcetype = 'geojson',
        source = red_file,
        type = 'fill',
        color='rgb(255, 0, 0)',
        opacity=settings['opacity']
    )
],
```

This here depicts the *layers* argument of the Plotly figure object. This list defines what is drawn on top of the MapBox object. As we can see, we define three different layers which all source location data from their respective JSON objects, and each layer has its own color. This layer structure is the exact reason why we can’t use this same method for the primary scrambling codes. Having three different layers is just fine, but 512 different layers is out of the question with Plotly. Already at 50 different layers, loading the Dash app takes several minutes and scrolling the map is very far from smooth. At 100 different layers the web app does not load at all.

Unfortunately with the above method, there is no way to color the individual areas dynamically, meaning that the color data can’t be read from an individual GeoJSON Polygon object. The coloring has to be done on the layer level. Therefore, prototyping the primary scrambling code visualization was implemented with a different framework, which is the node.js library deck.gl. Like Dash, deck.gl also creates a MapBox object and draws on top of it by creating a separate figure layer. Deck.gl offers a *GeoJsonLayer* object, which is just like tailored for this project. Taking a look at the object, which used in the context of this work:

```
const layer = new GeoJsonLayer({
   id: 'geojson',
   data,
   opacity: 0.2,
lineWidthScale: 40,
   stroked: true,
   filled: true,
   extruded: false,
   wireframe: false,
   getFillColor: f => f.properties.color,
   getLineColor: f => [0, 0, 0]
 });
```

As we can see, this object offers many attributes for defining our figure layer. Most importantly, it offers a way to read the color data dynamically from each individual GeoJSON Polygon object. With this we are able to display areas with up to 512 different colors in our map view.

Results of the network coverage visualization have already been demonstrated in Figure 6 and Figure 7. Examples of most frequent scrambling code can be seen in Figure 20 and Figure 21. As mentioned before, one color responds to one particular primary scrambling code.

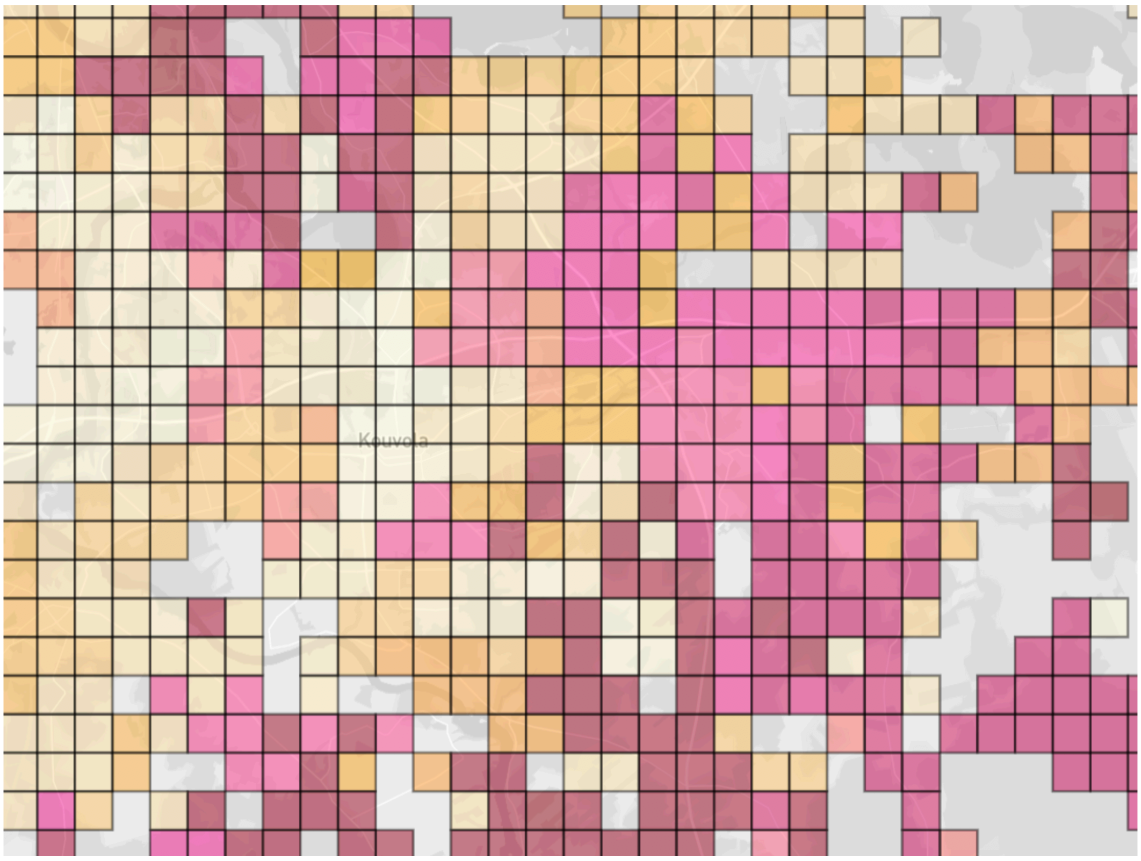


Figure 20. A screenshot of a map application featuring cell dominance within the Kouvola area.

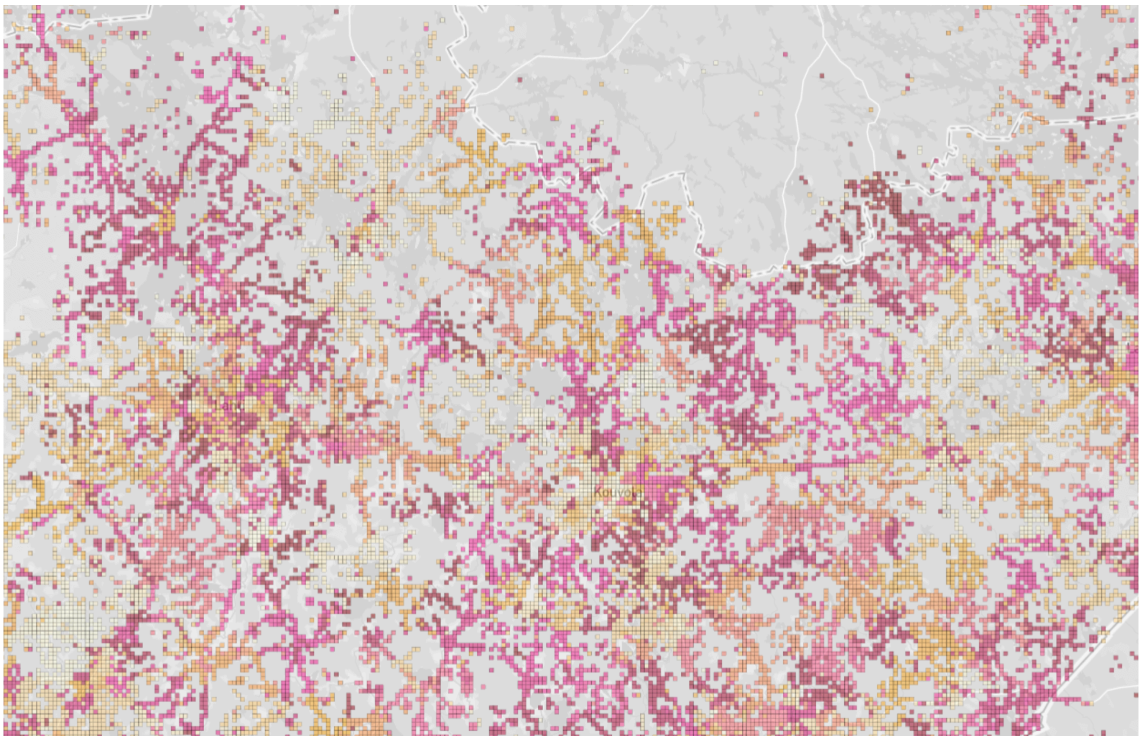


Figure 21. A screenshot of a map application featuring cell dominance within a broader range of Finland.

7 CONCLUSIONS AND DISCUSSION

As already clearly proved by presenting the use-case, utilizing signaling measurement data can bring many useful possibilities for monitoring network quality. The use-case presented here is utilizing only a small fraction of the offered data, so many more use-cases are possible for network planners, who are looking for automating network monitoring and optimization. One possible use-case is to utilize the ASU (Active Set Update) measurements. This use-case could monitor a cells which are initiating handovers to other cells way too often or cells which are not initiating handovers, when they should be initiated. Graphs can also be plotted for network planners, who can manually monitor that how handovers are behaving with certain cells.

The particular use-case presented here could also be automated further. At its current state it can provide network planners a view of how the network is performing at measured areas, but conclusions and actions still have to be made using one's own discretion. Automated handling of problem areas can be programmed easily, though an underlying logic has to be designed first.

As the work presented here is only a mere proof-of-concept, a prototype, the implementation needs refactoring before deployment for production. For starters, Pandas is great for quick results and prototyping, but it does not scale well with growing data sets. For the actual implementation something like Apache's Spark and pyspark could be utilized, as they are specifically designed for analytics with large-scale data. Pandas performs now well with sharding techniques and a with the provided static, limited test data set, but when a live data stream from several RNCs jumps into the picture, something faster and lighter is needed. Unfortunately, no live data stream was available during development, so this aspect was not able to make it into this documentation. Another part which needs refactoring is the visualization part. It is one thing to prototype an use-case with two separate mapping implementations, but for the actual productization this is far from feasible. One possibility is to use deck.gl for both visualizations, but JavaScript requires more maintenance with dependency libraries, because security issues are much more common with JavaScript. Deck.gl also isn't as easily managed as Plotly's Dash. Another option is to wait out, if the Dash framework would receive an update in the near-future, which would bring the feature utilize GeoJSON layers more effectively with Plotly figures. A third option would be to ditch custom web apps altogether and export the enriched data to KML format, which is a format used by Google Earth.

In the actual production implementation of this use-case, Docker could also play a bigger role. Now the majority of the code is running from the same file, but this could (and should) be cut into smaller, modular pieces. For example, the database initiation and raw data insert functions could be their own individual component. Next in line would be the data enrichment function within its own implementation. When data is enriched, the GeoJSON file generator would run independently on its own. All these different components would run as Docker containers having their own environments. When the live feed of MegaMon data becomes available, there is probably also need for a new component, which is responsible for fetching new data and storing it temporarily before it is pushed to the database.

All in all, the signaling measurements provided by MegaMon enable a lot of possibilities for optimization use-cases. The productization of the particular use-case presented here already offers a project, which would provide a basis for writing a similar document as this one.

REFERENCES

3G Partnership Project. 2007. 3GPP Scope and Objectives [PDF]. Published August 31 2007. http://www.3gpp.org/ftp/Inbox/2008_web_files/3GPP_Scopeando310807.pdf

Barr, J. 2005. BitKeeper and Linux: The end of the road? Linux.com. Published April 11, 2005. https://www.linux.com/news/bitkeeper-and-linux-end-road

Dash. 2017. Introducing Dash. Medium. Published June 21, 2018. https://medium.com/@plotlygraphs/introducing-dash-5ecf7191b503

Docker. 2018. Docker overview. Read October 29, 2018. https://docs.docker.com/engine/docker-overview/

ETSI TS 125 215. 2012. Physical Layer Measurements (FDD). Version 11.0.0. France: ETSI. Read October 30, 2018.

ETSI TS 137 320. 2016. Radio measurement collection for Minimization of Drive Tests (MDT); Overall description; Stage 2. Version 13.1.0. France: ETSI. Read October 8th, 2018.

Hill, W. 2017. Geohashing. Medium. Published April 22, 2017. https://medium.com/@bkawk/geohashing-20b282fc9655

Internet Engineering Task Force. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Read October 2, 2018. https://tools.ietf.org/html/rfc4180

Internet Engineering Task Force. 2016. The GeoJSON Format. Read September 30, 2018. https://tools.ietf.org/html/rfc7946

Pandas. 2018. pandas: powerful Python data analysis toolkit, Release 0.23.4 [PDF]. Published August 6, 2018. http://pandas.pydata.org/pandas-docs/stable/pandas.pdf

Pearce, R. 2013. Dead database walking: MySQL's creator on why the future belongs to MariaDB. Computerworld. Published March 28, 2013. https://www.computerworld.com.au/article/457551/dead_database_walking_mysql_creator_why_future_belongs_mariadb/

Peters, T. 2004. PEP 20 -- The Zen of Python. Published August 19, 2004. https://www.python.org/dev/peps/pep-0020/

Kaaranen, H., Ahtiainen, A., Laitinen, L., Naghian, S., Niemi, V. 2005. UMTS Networks: Architecture, Mobility and Services. 2nd edition. West Sussex, England: John Wiley & Sons.

Kopf, D. 2017. Meet the man behind the most important tool in data science. Quartz. Published December 8, 2017. https://qz.com/1126615/the-story-of-the-most-important-tool-in-data-science/

Korhonen, J. 2003. Introduction to 3G Mobile Communications. 2nd edition. Manhattan, United States: Artech House.

Kreher, R. 2006. UMTS Performance Measurement: A Practical Guide to KPIs for the UTRAN Environment. West Sussex, England: John Wiley & Sons.

Kreher, R., Rüdebusch, T. 2007. UMTS Signaling: UMTS Interfaces, Protocols, Message Flows and Procedures Analyzed and Explained. 2nd edition. West Sussex, England: John Wiley & Sons.

Kuhlman, D. 2009. A Python Book: Beginning Python, Advanced Python, and Python Exercises. Updated April 22, 2012. https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html

Laiho, J., Wacker, A., Novosad, T. 2006. Radio Network Planning and Optimisation for UMTS. 2nd edition. West Sussex, England: John Wiley & Sons.

Maly, T. 2013. A Cloudless Atlas — How MapBox Aims to Make the World's 'Most Beautiful Map'. Wired. Published May 14, 2013. https://www.wired.com/2013/05/a-cloudless-atlas/

SQLAlchemy. 2018. SQLAlchemy 1.2 Documentation: Engine Configuration. Read October 27, 2018. https://docs.sqlalchemy.org/en/latest/core/engines.html

Torvalds, L. 2005. Meet the new maintainer. Git mailing list. Published July 27, 2005. https://marc.info/?l=git&m=112243466603239

Turkka, J. 2014. Aspects of Knowledge Mining on Minimizing Drive Tests in Self-organizing Cellular Networks. Tampere University of Technology. Doctoral thesis.

Van Rossum, G. 1996. Foreword for "Programming Python" (1st ed.). Read September 29, 2018. https://www.python.org/doc/essays/foreword/

Van Rossum, G. 2018. Transfer of power. Published July 12, 2018. https://mail.python.org/pipermail/python-committers/2018-July/005664.html

Warsaw, B., Hylton, J., Goodger, D., Coghlan, N. 2000. PEP 1 -- PEP Purpose and Guidelines. Published June 13, 2000. https://www.python.org/dev/peps/pep-0001/