Technische Universität München
Fakultät für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste

A Decentralized and Autonomous Anomaly Detection Infrastructure for Decentralized Peer-to-Peer Networks

Masterarbeit in Informatik

durchgeführt am
Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München

von

Omar Tarabai

Betreuer:	Dipl.-Inform. Matthias Wachs
Tag der Abgabe:	15. October 2014

I assure the single handed composition of this thesis only supported by declared resources.

Garching, 15 October 2014

Abstract:

In decentralized networks, collecting and analysing information from the network is useful for developers and operators to monitor the behaviour and detect anomalies such as attacks or failures in both the overlay and underlay networks. But realizing such an infrastructure is hard to achieve due to the decentralized nature of the network especially if the anomaly occurs on systems not operated by developers or participants get separated from the collection points. In this thesis a decentralized monitoring infrastructure using a decentralized peer-to-peer network is developed to collect information and detect anomalies in a collaborative way without coordination by and in absence of a centralized infrastructure and report detected incidents to a monitoring infrastructure.

We start by introducing background information about peer-to-peer networks, anomalies and anomaly detection techniques in literature. Then we present some of the related work regarding monitoring decentralized networks, anomaly detection and data aggregation in decentralized networks. Then we perform an analysis of the system objectives, target environment and the desired properties of the system. Then we design the system in terms of the overall structure and its individual components. We follow with details about the system implementation. Lastly, we evaluate the final system implementation against our desired objectives.

Contents

1. Motivation

Considering the decentralized nature of a peer-to-peer (P2P) network, services offered by the network (such as files sharing and instant messaging) rely on the resources made available by its individual peers and the underlay communication links between them instead of a centralized server(s) in case of the client-server model. Although this decentralization and distribution of tasks increases the robustness of the network as opposed to having single point(s) of failure, monitoring the P2P network becomes a challenge.

Monitoring is necessary to provide developers and operators with information about the behaviour of the network and its participating peers and to detect any issues or anomalies affecting them. For example, developers of a large-scale P2P network would be interested in monitoring the behaviour of the network after a new client software release and detecting any issues caused by bugs in the software.

The behaviour of the network can be expressed using a group of metrics that monitor various aspects of the network. For example, in a file-sharing P2P network, monitoring metrics such as the number of files shared per user, the average file lookup time, the number of overlay connections per peer, etc can help monitor the behaviour and the state of the network. An issue or anomaly affecting the network can be defined as a deviation from what is considered to be normal network behaviour and can be classified into anomalies affecting the overlay P2P network caused by attacks on the network or client software bugs and anomalies affecting the underlay network which can be caused by a variety of issues such hardware or network configuration problems.

1.1 Challenges

In a fully-decentralized P2P network, the absence of central or *supernode*-like entities makes monitoring the network a challenge since any information required for understanding the behaviour of the network is distributed over possibly hundreds or even thousands of participating peers. Gathering information from all peers becomes unscalable as the network grows and privacy concerns limits the nature of information that can be collected from peers.

In addition to monitoring the state of the network, to automatically and reliably detect anomalies, it is required to characterize and construct a model of normal network behaviour and identify abnormal behaviour as it occurs. The normal behaviour of a P2P network is expected to be constantly evolving and a present notion of normal behaviour might not be valid in the future.

1.2 Goal

The goal of this thesis is to design and implement a monitoring infrastructure for decentralized P2P networks that can monitor the behaviour of the network in a decentralized manner and automatically detect anomalies affecting it. The monitoring infrastructure should be able to operate in absence of a centralized authority while possibly report detected anomalies to monitoring points operated by network developers or operators.

2. Background

In this chapter we present an introduction into the main topics related to this thesis. We start with a brief introduction into peer-to-peer (P2P) networks and GNUnet which is a P2P framework that is used for implementing our work. We follow with an introduction into anomalies, its types and the problem of detecting anomalies.

2.1 Peer-to-peer Networks

P2P networks rely on the distribution of load and tasks between peers or users participating in the network. In contrast to a client-server architecture, P2P networks are fully decentralized which means that the peers do not rely on a central server or group of servers to supply certain resources.

P2P networks are used for a variety of applications such as file sharing (e.g. Gnutella, Bittorrent), instant message (e.g. Skype), media streaming (e.g. Spotify), anonymization (e.g. Tor) and digital currency (e.g. Bitcoin).

GNUnet[1] is a free and decentralized peer-to-peer framework published under the GNU Public License (GPL)[2] that aims to provide a reliable and censorship-resistance system of free information exchange. Multiple applications implemented on top of GNUnet include a Distributed Hash Table (DHT) for shared data storage among peers, file-sharing service, a distributed Domain Name System replacement titled GNU Name System (GNS) among others[3]. GNUnet identify peers in the network solely by their public key which is generated for each peer when it is started.

2.2 Anomalies

An *anomaly* (also referred to as *outlier* by Hodge and Austin [HoAu04]) is a pattern in data where that deviates from the expected or normal behaviour [ChBK09]. An anomaly detection method is a process that tries to detect such patterns by creating a model of perceived normal behaviour from a given dataset and identifying the patterns that do not conform to the modelled normal behaviour. For example, in the area of intrusion detection, a malicious behaviour might trigger an anomaly in network traffic or system

[1]https://gnunet.org/
[2]http://www.gnu.org/copyleft/gpl.html
[3]https://gnunet.org/gnunet-source-overview

call patterns that can be detected by anomaly-based intrusion detection systems. Other applications of anomaly detection methods include fraud detection, network performance monitoring, medical condition monitoring, satellite image analysis and detecting errors in text [ChBK09] [ZhYG09] [ThJi03].

In the context of P2P networking, an anomaly can be defined as a failure in the overlay P2P network or the underlay network that prevents one or more peers from participating in the network in the expected manner. Failures in the overlay network can be caused by software bugs that affect peer connectivity, performance or the ability to participate correctly in the network or an attack on the P2P network. Failures in the underlay network can be caused by network hardware or configuration problems, for example, an Internet sea cable damage causing a country to be disconnected from the global Internet.

Anomalies can be generally classified into *point anomalies*, *long-duration anomalies* and *contextual anomalies*. Following is a description for each of these categories.

2.2.1 Point Anomalies

Point anomalies are characterized by a single or a very small number of observations in a row significantly deviating from the normal previously observed behaviour before returning back to normal. Figure 2.1 shows an example of a point anomaly marked on the graph by the dotted square, other data points in the graph are considered within the range of normal values.

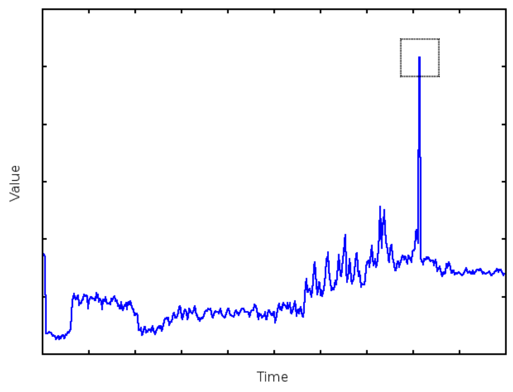


Figure 2.1: Point Anomaly.

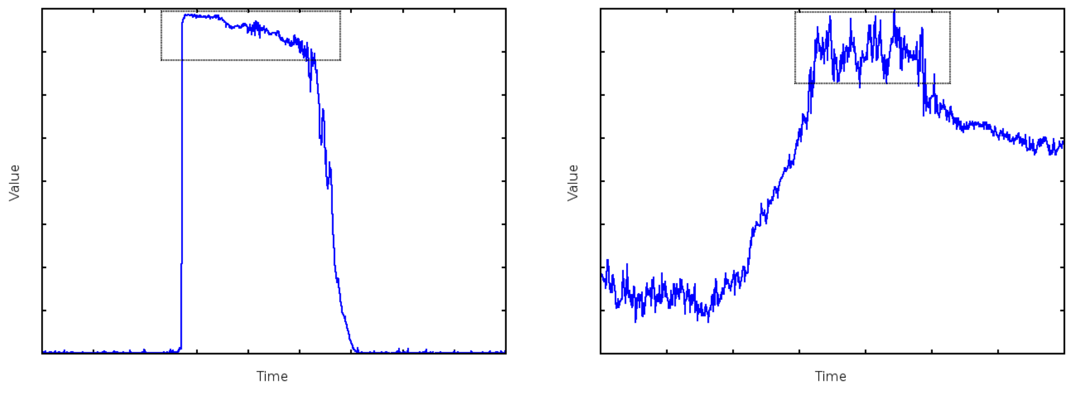
Point anomalies are caused by short-term faults or deviations in the system being monitored. For example, in a credit card fraud detection system, a transaction with a spent amount much higher than the normal range of expenditure would constitute a point anomalies. In other systems where measurements are collected using specialized hardware such as in a sensor network measuring temperature values, a point anomaly can be caused by a measurement fault caused by the hardware.

Point anomalies are relatively easy to detect and are the target of most anomaly detection methods such as statistical anomaly detection methods [MaSi03] and clustering-based methods [RLPB06].

2.2.2 Long-duration Anomalies

Long-duration anomalies are characterized by a deviation that persists across a high number of subsequent observations. The jump from normal to deviant observations can be sudden as shown in figure 2.2(a) or gradual as shown in figure 2.2(b). *Long-duration anomalies* can be caused by a general failure in the system under observation or a cyber attack on an observed network, etc.

The main challenge in detecting long-duration anomalies is differentiating between a long-duration anomaly and an evolution in normal system behaviour, particularly, in the case of a slow gradual change from normal to anomalous which can cause that the detection model be trained to identify the new (anomalous) behaviour as normal. The sensitivity to change of the training model determines the trade-off between the number of false positives and true negatives detected.



(a) Long-duration anomaly with sudden change (b) Long-duration anomaly with gradual change

Figure 2.2: Long-duration anomalies.

2.2.3 Contextual Anomalies

Contextual anomalies are observations that are considered anomalous due to the context they appear in but might not be considered anomalous otherwise. The context can be the spatial position of the observation or its position within a sequence of observations such as in the case of time-series data. Figure 2.3 shows an example of a point that is only anomalous due to its position in the sequence of points plotted by the graph.

An example of a contextual anomaly is a high surge in network traffic occurring a time of the day where low traffic is expected even though it is considered normal during another time of the day.

Detecting contextual anomalies is more challenging than detecting point or long-duration anomalies since detection methods need to take into consideration the contextual attributes in addition to the behavioural attributes of the observed system. In some cases, such as in time-series data, the separation in context is not straightforward.

A group of anomaly detection methods try to reduce contextual anomalies to point anomalies by identifying and isolating contexts and applying known point anomaly detection methods within the identified context. For example, Basu and Meckesheimer [BaMe07] propose a method that compares the observed value to the median of its neighbouring values for detecting contextual anomalies in time-series data.

Another group of anomaly detection methods focus on modelling the observed behaviour with respect to the given context. For example, applying a regression technique on sequential data by fitting a regression line and checking if subsequent values fit the constructed model [ABAB].

2.3 Anomaly Detection

Anomaly detection methods rely on comparing current behaviour of the monitored system to what is considered to be ”normal” behaviour. Normal behaviour varies according to

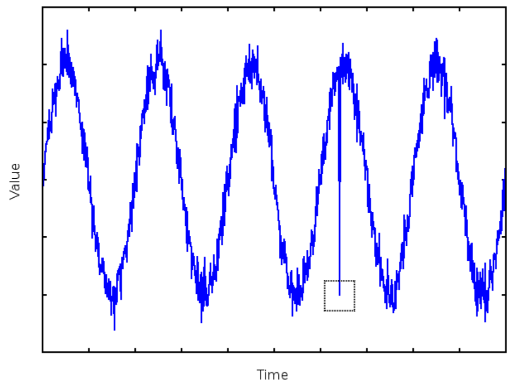


Figure 2.3: Contextual Anomalies.

the problem domain and the context in which the behaviour is observed, for example, normal volume of network traffic can vary according to the time of day. The behaviour is also likely to evolve as the system under consideration can evolve requiring that the model adapt itself automatically. In same cases, the unavailability of labelled training data requires that the model be robust enough to handle anomalies within the training set with minimum efficiency penalty. Therefore, it is required that the model "learn" normal behaviour from observation in case a predefined notion of normal behaviour is not available.

Anomaly detection methods are classified according to their *input type*, *output type*, *learning method* and *target anomaly type*. We look more into the each of these classification.

The expected input data type for anomaly detection methods falls into one of the following: *binary*, *categorical* and *continuous*. The input can be *univariate* where one variable is considered independently of any other variables, or *multivariate* where more than one variable (possibly having different data types) are assumed to be related and considered together by the anomaly detection method. In addition to a possible relation between multiple variables, a relation can exist between data points of a single variable. For example, in time sequence data, data points are related by their *temporal* component. A *spatial* or *spatio-temporal* relation between data points is also possible.

The output of an anomaly detection method can be either a label (*normal* or *anomalous*) or a score that describes the degree of certainty that the reported data point is anomalous. A threshold can later be applied by an expert on the score value to separate *normal* from *anomalous* data points.

Anomaly detection methods operate under one of the following learning methods: 1) *Supervised learning.* Data sets labelled (*normal* or *anomalous*) are used to train the model. 2) *Semi-supervised learning.* The model is trained with only normal data. During the detection phase, data points that do not conform to the learned normal behaviour is considered anomalous. 3) *Unsupervised learning.* No training data is required in this case. The input data is assumed to contain a high ratio of normal to anomalous points, otherwise, frequent anomalous behaviour will be considered as normal by the model.

2.3.1 Methods

We present a categorization of anomaly detection methods from literature with the advantages and disadvantages of each category.

2.3.1.1 K^{th} Nearest Neighbour

K^{th} nearest neighbour methods use the concept of proximity to identify outliers or anomalous points in a given data set. For each data point, distance to its K^{th} nearest neighbour defines its anomaly score. The data point is considered anomalous if its anomaly score exceeds a certain threshold. Different variations of this method were developed. Eskin et al. [EAPP$^+$02] use the sum of distances to the k nearest neighbours as the anomaly score. Knorr et al. [KnNT00] count the number of neighbours within a distance d from the data point, this is considered as the *density* of the data point neighbourhood, the anomaly score can be calculated as the inverse of that density.

Advantages:

- Purely data driven, no a priori knowledge required.

Disadvantages:

- The values of K and the anomaly threshold must be pre-specified.
- Computationally expensive because the distance to all other data points need to be calculated.
- Model must retain all (or at least recent) data points.

2.3.1.2 Clustering-based Techniques

Clustering-based techniques group similar data points into *clusters* and identify anomalous points by their relation to the established clusters. Clustering-based techniques generally go through two phases. The first phase applies a clustering algorithm such as K-means clustering or Expectation-maximization to the set of data points to detect clusters. The second phase detects anomalous data points by evaluating new points against existing clusters, this can be achieved by multiple methods. Data points can be considered anomalous if they do not belong to any cluster or belong to cluster with size below a certain threshold. Alternatively, an anomaly score for the data point can be calculated as the distance between the data point and the centroid of the nearest cluster [RLPB06].

Advantages:

- Computationally inexpensive since evaluation is performed against a limited number of clusters.

Disadvantages:

- The choice of clustering methodology depends on the type of data observed.
- Not effective if anomalies form a significant cluster together.

2.3.1.3 Gaussian Model

Gaussian model-based techniques assume that the observed data follows a Gaussian distribution. The *mean* and *standard deviation* values of the model are calculated. The method checks if a new data point falls outside a *confidence area* which is defined by Denning et al. [DeNe85] as d standard deviations away from the mean in both positive and negative directions. According to the three-sigma rule[Puke94], 68.27% of values in a Gaussian random variable lie within one standard deviation from the mean, 95.45% lie within two standard deviations from the mean and 99.73% lie within three standard deviations from the mean. Other methods such as the *box plot* [McTL78] and *Grubb's test* [Grub50] use different definitions of the *confidence area* to detect anomalies under the same assumption that the data follows a Gaussian distribution.

The model can be slightly modified to put more weight on recent data points [Denn87] thus making it more adaptable to slight changes over time. In NIDES (a popular intrusion detection system [JaVa93]), a "half-life" value is set by the security officer which specifies the time after which the weight of an audit record becomes half that of a recent audit record.

Advantages:

- Model is represented by only two parameters, no need to retain all data points.
- If the distribution assumption is true, the model can accurately detect anomalies.

Disadvantages:

- Assumes a specific distribution of observed data which is not accurate in all cases.

2.3.1.4 Histogram / Binning

Histogram-based methods sort data points into *histograms* or *bins* where each bin covers a range of values. New data points are assigned an anomaly score inversely proportional to the size of the bin it belongs to. The range size covered by the bins is critical to the anomaly detection process, a too small range size can lead to high rate of false positives and a too big range size can lead to high rate of true negatives.

Advantages:

- Does not make assumptions about the statistical distribution of data.

Disadvantages:

- Difficult to arrive at the optimal bin size.
- Requires a training phase with a minimal amount of anomalous data.

2.3.1.5 Kernel Density Estimation

Kernel density estimation (KDE) tries to estimate the probability density function (PDF) of a random variable by representing each data point as a local parametric model such as a Gaussian model (as used by [YeCh02]), the local model is also known as a kernel. Kernels are added to form a smooth estimated PDF function [Parz62]. Data points are declared anomalous if they lie in a low probability area of the PDF.

Advantages:

- Does not make assumptions about the statistical distribution of data.

Disadvantages:

- Requires a training phase with a minimal amount of anomalous data.

3. Related Work

3.1 Monitoring P2P Networks

We look at monitoring solutions implemented for Tor and I2P, two popular anonymity P2P networks to understand common methods used in practice to monitor the behaviour of P2P networks. We give an overview of the P2P networks and the methods implemented for monitoring them.

3.1.1 Tor Metrics

Tor[DiMS04] is a P2P network that uses relay nodes and layered encryption to achieve anonymity. Tor metrics[1] is a collection of tools to monitor and report user count and performance of the Tor network. Collected information can be classified into: 1) *Architecture specific*: number of relay nodes, relays by version/platform/type, relay available bandwidth, etc. 2) *User information*: total number of users, number of users per country, bridge users, etc. 3) *Performance specific*: average time to download files of three different sizes, timeout and failure rates, percentage of uni-/bidirectional connection.

The above information is collected from two main sources: bridges and relays publishing usage statistics to Tor directory nodes[LoMD10], and clients continuously trying to download files of three different sizes from the network[Loes09].

3.1.2 I2P

I2P[j(Ps03] is an anonymity P2P network that uses a variant of onion routing[GoRS96]. Communication *tunnels* are constructed through a number of *router* nodes with layered encryption. A *destination* correspond to an anonymous application endpoint. A peer typically runs one *router* and multiple *destinations*. Applications running on top if I2P include web browsing, web hosting, bittorrent clients and instant messaging.

I2P uses a simple DHT distributed among high speed *routers* (called *floodfill routers*) to store contact information of all *routers* (RouterInfos) and *destinations* (LeaseSets) in the network. Statistics about the network are gathered by deploying multiple *floodfill routers* into the network to collect RouterInfos and LeaseSets from the DHT [TiCF12]. Statistics are limited by the collected information about *routers* and *destinations*, they are visualized on internal websites such as http://stats.i2p/ and include: total number of routers over time, routers by version, number of destinations, etc.

[1]https://metrics.torproject.org/

3.2 P2P Aggregation

P2P aggregation methods deal with the problem of collecting and summarizing numerical information from a decentralized and highly dynamic P2P network. For example, retrieving the total number of files available in a file-sharing P2P network. The basic aggregate functions are *sum*, *average*, *minimum* and *maximum* [MBDG09]. More complex aggregate and data mining methods can be built on top of these basic aggregate functions.

P2P aggregation methods can be classified into three categories: gossip-based, tree-based and hybrid. Gossip-based methods are based on the exchange of messages between randomly chosen peers. Tree-based methods construct a logical tree of peers where data is aggregated from the leaves to the root of the tree. Hybrid methods try to combine both approaches [MBDG09].

3.2.1 Gossip-based

In gossip-based (also known as epidemic [EGKM04]) methods, during each round of the algorithm, each peer chooses one or more other peers at random to exchange information with [KeDG03]. The spread of information is similar to the spread of an epidemic. Due to the structureless nature of gossip-based methods, they provide high fault-tolerance and self-stabilization with simplicity at the cost of a higher communication and computation overhead than tree-based methods.

Kempe et al. [KeDG03] proposes *Push-sum*, a simple proactive gossip-based method for calculating sums or averages. At each time step, each peer chooses a target peer uniformly at random and sends the latest aggregate estimate and a weight value that represents the number of exchanged messages used to arrive at that estimate to the target peer. At the end of the time step, each peer calculates the new aggregate estimate from all previous messages received. They also propose other methods based on the *Push-sum* method for calculating more complex queries.

Jelasity et al. [JeMB05] proposes a similar but more generic method of computing aggregates. In case of computing the average, they do not assign weights to calculated estimates, instead, the latest local value is averaged with any received value with equal weights.

Other methods [KeKD01] replace the process of picking target peers uniformly at random with a non-uniform process such as a distance-based one in case close-by peers are more ”interesting”.

3.2.2 Tree-based

In the simplest version of the tree-based methods [BGMGM03][MFHH02], the querying node broadcasts a query message to the whole network, the broadcast message is propagated in a spanning tree structure with the querying node as the root of the tree and the last nodes that receive the broadcast message as the leaves of the tree. The information is aggregated bottom-up in the tree until the result reaches the root (querying node). This method is highly sensitive to faults in case of node failure, particularly in nodes higher up in the tree, however, achieving a lower convergence time and lower communication and computation overhead than gossip-based methods.

To address the issue of node failures, Bawa et al. [BGMGM03] propose *MultipleTree* where k (2 or 3) randomly-constructed spanning trees are used for the aggregation process. Thus, increasing the robustness by aggregation the information through multiple paths. Another method suggested by [DaSt05] relies on an underlying infrastructure that detects and reports node failure. On the event of node failure, the spanning tree is modified accordingly.

3.2.3 Hybrid

Hybrid methods try to combine the scalability of tree-based algorithms with the resilience of gossip-based algorithms. Renesse et al. [VRBV03] propose *Astrolabe* a method that uses a gossip protocol to propagate and update information about the structure of an overlay spanning tree used for aggregation. Artigas et al. [ArGS06] propose a method that organises nodes into a tree of clusters where each cluster is a grouping of nodes according to the *node-id*. Answers to a query are calculated within each cluster using gossip and then aggregated through the tree.

4. Analysis

In this chapter, we analyse the objectives of the thesis and present an overview of the target environment and the expected adversary model. We then describe our approach for the design of the monitoring infrastructure.

4.1 Objectives

The main objective is to establish a monitoring infrastructure for use by developers and operators of large-scale decentralized P2P networks to constantly monitor the behaviour of the network and automatically detect anomalies or failures affecting the network. The monitoring infrastructure should be: 1) *decentralized*: does not rely on a functionality provided by a central server(s) to perform its tasks, and 2) *autonomous*: can operate without user intervention or expert feedback.

The infrastructure will be based on GNUnet[1], a free and decentralized peer-to-peer framework that offers common P2P network functionality such as secure communication between peers, routing and a distributed hash table among others. GNUnet and the monitoring infrastructure can be deployed on the target network to achieve our required objectives.

4.2 Environment

Our target environment is a large-scale decentralized P2P network. The network is dynamic with a possibly high rate of peers joining and leaving the network. The network behaviour can be summarized using a set of metrics that describe the state of the system. The network is likely to evolve, for example, by new services being added. Therefore, the set of metrics that describe its state is likely to change.

4.3 Adversary Model

An adversary in the target environment controls a number of nodes or peers in the network. The adversary can participate and feed malicious information into the network. We also assume that the hardware capabilities of the adversary limits him from controlling a majority of peers in the network.

[1]https://gnunet.org/

4.4 Approach

The monitoring infrastructure is to be implemented using GNUnet P2P framework. The approach can be described in terms of: 1) information collection, 2) analysis and anomaly detection, and 3) reporting. All operations or subsystems implemented within the monitoring infrastructure can be enabled/disabled by the user participating in the system.

4.4.1 Information Collection

Information representing the state of the P2P network is to be collected locally by each peer participating in the monitoring infrastructure. Since the monitoring infrastructure targets any decentralized P2P network, the type of information or metrics that represent the network's state should be dynamic and easily modified by the developers or the users of the network. This will be implemented using *sensors*, where each *sensor* defines the source of information to be collected and other related definition fields such as the sampling interval.

Information to be collected can be classified into two categories: 1) network-based information: related to the P2P network itself. For example, number of neighbouring peers, peer uptime, DHT lookup time, etc. 2) host-based information: related to the host running the peer and its underlay connection. For example, memory and processor consumption of the P2P client software, ping RTT to a stable high-bandwidth server, etc.

4.4.2 Analysis and Anomaly Detection

Analysis and anomaly detection is performed locally on the peer side using the information collected by the *sensors*. Due to the dynamic nature of the observed network, it is difficult to determine a priori the expected values or behaviour of the input data. Therefore, for anomaly detection we need to rely solely on learning the normal behaviour from observation and detect the abnormal or anomalous behaviour using one of the anomaly detection methods established in literature. The output of the anomaly detection process should be a label (normal or anomalous) given a new observation, since an anomaly score will not be useful as an output due to the lack of expert knowledge required to determine a label from an anomaly score.

To allow flexibility in choosing the proper anomaly detection method suitable for the type of collected data, the monitoring infrastructure should support implementing and changing the used anomaly detection method easily using a plugin system.

4.4.3 Reporting

Both information collection and anomaly detection are performed locally on the peer side with the results available to the user running the peer. Peers will have the ability to report collected or information or detected anomalies to one or more collection point peers that are operated by the network developers. For privacy concerns, the users should have control over this process by enabling/disabling the reporting on all or a subset of the *sensors* collecting information.

Additionally, peers can exchange anomaly reports with other peers that are directly connected in the overlay layer so each peer will keep track of the status of its neighbourhood and send the neighbourhood status information with any anomaly reports sent to the collection point. Thus reporting more information to the collection point with less number of messages.

Due to the possibility of having malicious peers disrupting the monitoring infrastructure by sending flooding collection points or its neighbouring peers with fake anomaly reports, the monitoring infrastructure should make this malicious behaviour harder for the adversary by expecting a computationally-expensive proof-of-work to be generated by the peer sending anomaly reports and the proof-of-work attached to the anomaly report to be verified by the receiver and the report rejected in the case of an invalid proof-of-work.

continue to function

5. Design

In this chapter we present the design for a decentralized and autonomous monitoring infrastructure (MI) realized using GNUnet P2P network. realized using the .... framework

The decentralized approach ensures that the MI can function in the absence of a central server, all peers will equally posses the ability to collect information about the network, analyse it and detect anomalies given the peer's point of view of the network (both P2P network and underlay network). For example, anomalies can appear in a subset of the P2P network in case of localized connection problem, a bug in a specific version of the P2P software, etc. The process of collection, analysis, detection and alerting is performed autonomously by components running on top of GNUnet, no user intervention is required. Detected anomalies are reported to the user and to the user's neighbouring peers on the overlay P2P network and optionally to a central collection point. Anomaly detection uses learning algorithms to construct a model of normal network behaviour, therefore, no pre-defined information about the network behaviour is required.

We also introduce the notion of *experiments* where a developer can publish new sensor definitions into the P2P network, peers who choose to trust the developer can download and execute the sensors and periodically report collected measurements to a *collection point* peer run by the developer.

The idea of a decentralized and autonomous monitoring infrastructure adds to the robustness of the P2P network by having the peers independently perform anomaly detection without relying on one or multiple points of failure. It also addresses users' privacy concerns, since measurements collected from the peer side are not reported to another party, the MI uses information that can be considered a privacy violation if published (e.g. peer host network usage, peer location, etc).

Each peer choosing to participate in the MI runs a set of predefined *sensors*, the sensors periodically collect information about the state of the P2P network and the underlay network from the peer's point of view, these information are stored at the peer-side. An *analysis component* uses the collected information to construct a model of what is conceived as normal network (P2P and underlay) behaviour. Using the constructed model and the constant feed of new measurements, the *analysis component* detects deviations from the normal network behaviour, these deviations describe an abnormal system behaviour and is thus considered an anomaly.

The MI consists of three core components: *sensor component*, *reporting component* and *analysis component*. Additionally, two more components perform special tasks, namely, *dashboard component* and *update component*. Configurations give the user the option to enable/disable any of these components.

Fig 5.1 shows a diagram of system components and data paths. The dashed border encloses the components running on the local peer and their interactions. The components are controlled by the user running the local peer who also receives feedback from the *analysis component* in case of detected anomalies. On the other side, the components interact with other peers in the P2P network represented by the cloud. Some of these peers perform special functions (*collection point* and *update point* peers) and are run by the network operators.

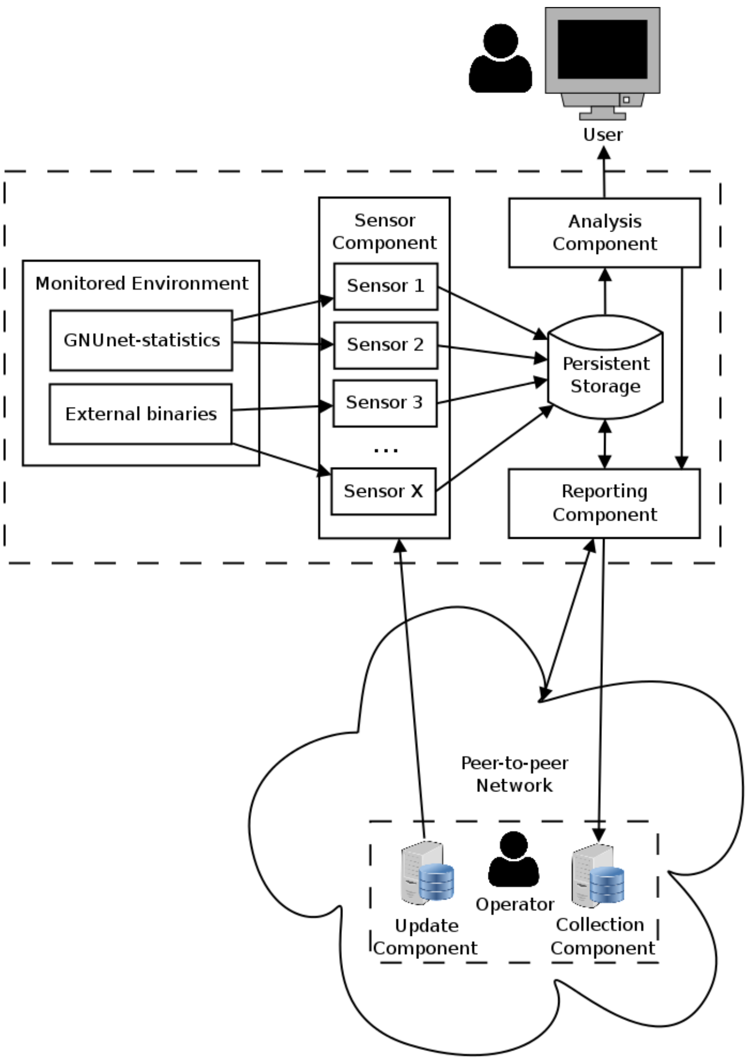


Figure 5.1: Component diagram

5.1 Sensor Component

The *sensor component* runs a group of sensors where each sensor defines the source of information for a single metric and other parameters that control the collection of information 5.1.1. To allow network developers or operators to implement new sensors in the case new functionality are added to the network or a need to observe other behaviour arises, sensor definitions can be updated from *update point* peers run by network developer or operators. *Update point* peers are defined in the component configuration.

5.1.1 Sensors Definition

Each defined sensor is uniquely identified by its name and version number. The version number helps with the update process in case a change to the sensor was made by the network developers, the system can replace the old version with the new version after update. The sensor definitions are stored in files to be easily editable by the user with one file corresponding to one sensor. An optional folder associated with the sensor contains any scripts/binaries required by the sensor. Table 5.1 describes the fields that can be found in sensor definition files.

Field name	Description	Optional
name	Sensor name	N
version	Sensor version number	N
description	Sensor description	Y
category	Category under which the sensor falls (e.g. TCP, datastore)	N
start_time	When does the sensor become active (format: %Y-%m-%d %H:%M:%S)	Y
end_time	When does the sensor expire (format: %Y-%m-%d %H:%M:%S)	Y
interval	Time interval to collect sensor information (in seconds)	N
lifetime	Lifetime of an information sample after which it is deleted from storage	Y
source	Either gnunet-statistics or external process	N
gnunet_stat_service	Name of the GNUnet service that is the source for the gnunet-statistics entry	If source is gnunet-statistics
gnunet_stat_name	Name of the gnunet-statistics entry	If source is gnunet-statistics
ext_process	Name of the external process to be executed	If source is external process
ext_args	Arguments to be passed to the external process	If source is external process
expected_datatype	The measurement data type to be expected (numeric/string)	N
collection_point	Peer-identity of peer running collection point	Y
collection_interval	Time interval to send sensor measurements to collection point (in seconds)	Y
report_anomalies	Flag specifying if anomalies are to be reported to collection point	N

Table 5.1: Sensor definition fields

The values of *description* and *category* fields are intended for users and network operators to understand the purpose of the sensor and the type of information being collected are ignored by the system.

Sensors collect information from two sources: 1) gnunet-statistics service which is a central service for collecting statistical values from different GNUnet services, entries are identified by the source GNUnet service responsible for the entry and a text description, the value returned is always an unsigned 64-bit integer. 2) External process executed by the sensor with optional given arguments. The process can be a system process (such as `ping` or `traceroute`) or a process/script that is bundled along with the sensor definition. The process exit code is checked and the returned output is discarded if an error occurred. Returned output is checked against the expected output data type defined in the sensor definition and the value is discarded if the data type mismatches.

Measurements are collected from the monitored environment according to the interval specified in the sensor definition. Measurements are stored in a persistent storage offered by GNUnet for a duration of time specified by the *lifetime* field in the sensor definition.

Since GNUnet is constantly evolving and more services are added, new information will need to be monitored. To allow developers to easily update the sensors, sensor definitions are stored in plain text files and read by the sensor component. We also introduce an update mechanism, where updates to the definition files are downloaded from *update point* peers run by a network operator and defined in component configuration. The updates can include changes to existing definitions or new definitions.

Additionally, the update mechanism in combination with the central *collection point* measurement reporting mechanism allows developers to create and run experiments on a group of peers, granted these peers trust the developer by including the peer identity of the developer's *update point* peer in the list of trusted update points in the component configuration. The experiment is expressed as a group of sensors that are created by the developer and requested by the target peers through the update mechanism. The sensors can run custom scripts or collect measurements for a duration bounded by the *start_time* and *end_time* defined in the sensor definitions file and the collected information reported back to the *collection point* which is also defined in the sensor definitions and controlled by the experiment developer.

We present a set of default sensors (table 5.2) to be initially defined that can be used to portray an overview of the network and its performance.

Name	Description
average-ping-rtt	Average ping latency to gnunet.org
core-peers-connected	Number of peers connected to GNUnet core service
datacache-bytes-stored	Bytes stored by GNUnet datacache service
dht-peers-connected	Number of peers connected to GNUnet dht service
fs-peers-connected	Number of peers connected to GNUnet fs service
gnunet-version	Installed GNUnet version number
known-peers	Number of GNUnet known peers
nse	GNUnet network size estimate
transport-bytes-received	Number of bytes received by GNUnet transport service
transport-http-connections	Number of GNUnet transport HTTP connections
transport-https-connections	Number of GNUnet transport HTTPS connections
transport-peers-connected	Number of peers connected to GNUnet transport service
transport-tcp-bytes-transmitted	Number of GNUnet transport TCP bytes transmitted
transport-tcp-sessions-active	Number of GNUnet transport service active TCP sessions

Table 5.2: Default sensors

5.1.2 Sensor Distribution and Update

Sensor definitions can be updated from *update point* peers run by a network operator and defined in component configuration. The component periodically (every 1 day) connect to the first defined update point and requests a list of available sensors. The component expects as reply a sequence of brief info messages, each corresponding to one sensor and containing the sensor name and version number. If that sensor does not locally exist or exists as an older version, the component sends a pull request for the new sensor definition and expects a single message with the full sensor definition and any associated scripts which is used to update the local sensor definitions repository.

If a failure occurs during connecting or communicating with the update point, the update point is marked as *failed*, and the process is repeated with the next defined update point if any. If all defined update points have failed, all *failed* flags are reset and the component retries with the first defined update point at the next interval (1 day).

5.2 Analysis Component

The *analysis component* is responsible for building models of network behaviour from collected sensor measurements, with one model instance built for each individual numeric sensor. Modelling methods are implemented as plugins to be switchable by only editing the component configuration. This offers the flexibility of implementing new modelling methods as plugins and exchanging them according to the method's suitability for analysing the type of information being collected. A *gaussian model* (described in 5.2.1) is implemented and used as the default modelling method.

The component monitors the system's persistent storage for any new sensor measurement values and feeds the measurement value to the model instance corresponding to the sensor which produced the measurement. The model (implemented in the model plugin) is expected to use the given value to perform two operations: 1) evaluate the value against the existing model to determine if the value is anomalous or non-anomalous and returning a flag with the result and 2) update the current model instance with the new value.

The component keeps a status flag (anomalous/non-anomalous) for each sensor. The flag is flipped only after X opposite subsequent results are returned by the model instance (i.e. X anomalous results in a row changes the value of the flag from *non-anomalous* to *anomalous*)). The value of X (called *confirmation count*) can be set in the component configuration and the default value is 1. When the sensor status flag flips, the *reporting component* is notified with the sensor name and the new status.

5.2.1 Gaussian Model

The *gaussian model* incrementally calculates the *weighted mean* and the *weighted standard deviation* of all the measurements previously fed to it. To calculate these values, the model keeps track of the sums s_0, s_1 and s_2 which are initialized to 0 and incremented with each new measurement x using the following equation:

$$s_j = s_j + w * x^j \tag{5.1}$$

The weight value w is initialized to 1 (i.e. the first measurement will have a weight of 1) and incremented with each new measurement by the *weight increment* free parameter. If the *weight increment* is set to 0, all measurements will have the same weight of 1 and the model will be calculating the normal *mean* and *standard deviation*. The weighting is used to give more weight to newer measurements, thus making it more adaptive to change.

The model requires an initial *training period*, specified as a free parameter representing the number of initial measurements to be used for training the model. During this *training period*, the model updates its sums with any new measurements but does not perform anomaly detection. Outside the *training period*, the model starts by evaluating the new measurement received against the current model to detect any anomaly, the sums are then incremented using equation 5.1 only if the value is considered to be normal. The detection result is then returned to the component.

To evaluate a given measurement value x, the model starts by calculating the values of the *weighted mean* and *weighted standard deviation*:

$$\bar{x} = \frac{s_1}{s_0} \tag{5.2}$$

$$s = \sqrt{\frac{s_0 s_2 - s_1^2}{s_0(s_0 - 1)}} \tag{5.3}$$

From these values, a measurement x is considered to be normal if it falls within a *confidence interval* bounded by d standard deviations away from the mean in both directions:

$$\bar{x} \pm d * s \tag{5.4}$$

According to the three-sigma rule[Puke94], 99.73% of values in a normally distributed random variable lie within three standard deviations from the mean. We use this fact to the set the default value of d to 3.

5.3 Reporting Component

The *reporting component* is responsible for communicating sensor measurements and anomaly reports to a *collection point* and exchanging anomaly reports with neighbouring peers in the overlay network. As explained in 5.1.1, sensor definitions contains an optional collection point GNUnet peer identity and two other fields, *collection_interval* and *report_anomalies* that define the level of reporting to the collection point.

5.3.1 Measurement Reporting

if the *collection_interval* value is set to a valid value that is greater than or equal to the *interval* value, the component uses this interval to periodically send the latest collected measurement to the *collection point* peer. If a problem or a delay causes that the latest measurement has already been sent to the collection point, nothing is sent until the next interval. Table 5.3 lists the information sent in a measurement report.

In case a connection to the collection point could not be established or existing connection has failed, pending measurement reports are discarded.

Sensor name
Sensor version
Measurement timestamp
Measurement value

Table 5.3: Measurement Report

5.3.2 Anomaly Reporting

Anomaly reporting is triggered by a change in a sensor anomaly status detected by the *analysis component* which notifies the *reporting component.* Anomaly reports are sent to all neighbouring peers listening for anomaly reports. The report contains the related sensor name and version and a flag with the new anomaly status of the sensor.

The component keeps track of the anomaly status of all combinations of known sensors and neighbouring peers. Upon receiving an anomaly report from a neighbouring peer, the component checks if the sensor name and version matches a local sensor, if yes, the status flag corresponding to the sensor and source peer is updated.

If the *report_anomalies* flag is set in a sensor definition, anomaly reports are also sent to the defined *collection point.* The difference between reporting to a *collection point* and to neighbouring peers is that the anomaly report sent to the *collection point* contains an additional field which is the ratio of neighbouring peers who reported an anomaly status of *anomalous* to the total number of neighbouring peers. A change in this value later (i.e. by receiving a new report from a neighbouring peer) triggers re-sending of the anomaly report to the *collection point.*

In case a connection to the collection point could not be established or existing connection has failed, pending anomaly reports are queued and resent in the order at which they were introduced when a connection to the collection point is successfully established. The components retries connecting to the collection point every one minute. This is to ensure that even in the case of underlay network connectivity issues, any anomalies are reported later when the underlay network connectivity is restored.

5.3.3 Proof-of-work

To protect against malicious peers flooding the network or collection points with fake anomaly reports, each anomaly report to be sent require the generation of a memory and computationally expensive proof-of-work and a signature using the private key of the peer.

The proof-of-work of any arbitrary block is a number appended to the block, such that the scrypt hash [Perc09] of the number and the block that has a minimum number of leading zero's. Scrypt is a memory-hard hash function, the number of required leading zero's (defined in component configuration) sets the computational time difficulty of the proof-of-work process.

A proof-of-work is generated by the component before sending an anomaly report to either the *collection point* or neighbouring peers. Both the message and the proof-of-work are then signed using the peer's private key (GNUnet identifies peers by a Ed25519 public key, see **??**). Verification of the proof-of-work is cheap since it only requires a single hash operation of the message, comparing the result of the hash operation with the hash received and verifying that it contains the required number of leading zero's.

When the component receives an anomaly report from a neighbouring peer, the signature and the proof-of-work are verified. If a problem is detected, the report is discarded and the connection to the peer is killed.

5.4 Dashboard Component

As mentioned in 5.3, collected measurements and anomaly reports can be optionally reported to a *collection point* peer defined in the sensor definition. The *dashboard component* is run by any peer wishing to take the role of a *collection point.*

The component listens for report messages of both types sent by any peer in the network. Reports received are check to make sure that the sensor name and version matches a sensor that exists locally, otherwise, the report is discarded. In the case of a measurement report, the report is saved directly to the local persistent storage. In the case of an anomaly report, the proof-of-work and signature are verified (see 5.3.3) first, if the verification fails, the report is discarded and the connection to the peer is killed, otherwise, the report is saved to the local persistent storage.

Other components wishing to handle received reports can monitor the local persistent storage for any new reports and read them.

5.5 Update Component

As mentioned in 5.1.2, peers can request sensor updates from defined *update point* peers. The *update component* is run by peers wishing to act as an *update point* for other peers in the network. Requests for sensor list is handled by reading all local sensors and sending a message corresponding to each sensor with the sensor name and version number. Requests for the full sensor definition of a specific sensor is handled by serializing the sensor definition file and any associated scripts into a message and sending it to the requester.

To perform a network experiment on a group of peers, the developer of the experiment has to create the set of sensors and associated scripts that will be used for the experiment and add them to the local sensor repository, run the update component and instruct the peers that will run the experiment to set the developer's peer identity of the as one of their update points. To collect measurements and/or anomaly reports from the participating peers, the sensor definitions should contain the developer's peer identity as the collection point.

6. Implementation

- introduction:
in this chapter we present ..
- overview
the implementation is realized using the gnunet p2p framework.
the implementation consists of 2 major components...

6.1 GNUnet P2P Framework

GNUnet[1] is a decentralized peer-to-peer framework published under the GNU Public License (GPL)[2] that aims to provide a reliable and censorship-resistance system of free information exchange.

GNUnet architecture is divided into layers and services. Each service has an API to be used by other services or user interface tools to access the service's functionality. Each service has a single configuration file for configuration parameters related to the service. Figure 6.1 shows the typical interaction between GNUnet services.

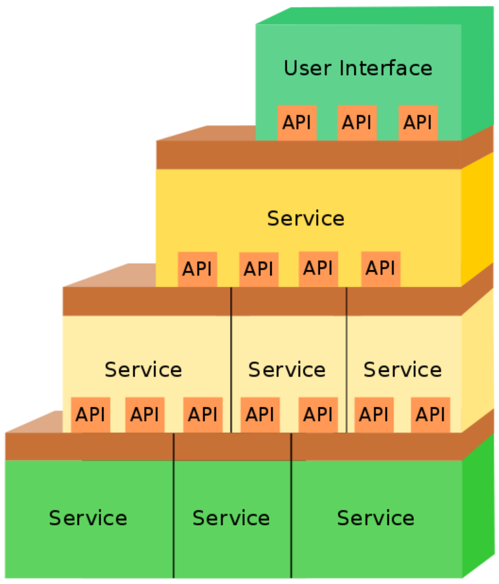


Figure 6.1: GNUnet service interaction. Source: gnunet.org

GNUnet identifies peers using a unique peer identity which is the public key of a key pair generated when the peer is started for the first time. The public key can be written as a 52-character ASCII string.

[1]https://gnunet.org/
[2]http://www.gnu.org/copyleft/gpl.html

used with this work

We use GNUnet version 0.10.1 for implementation. The following is a brief description of some of the main GNUnet services and other services used by our implementation.

Automatic Restart Manager (ARM).

ARM is responsible for system initialization and shutting down. It starts and stops single services and restarts a service in case of a crash. The command line interface `gnunet-arm` uses the ARM API to expose ARM functionality to the user, allowing the user to start/stop the whole system or single services.

is this required?

TRANSPORT service

TRANSPORT service is responsible for establishing connections to other peers. It uses plugins that implement communication protocols such as TCP, UDP, HTTP and HTTPS to exchange messages with other peers. It accepts incoming connections and notifies clients about connected and disconnected peers.

and of course establishes ... therefore write "maintains connections to other peers"

CORE service

CORE service builds on the TRANSPORT service to offer secure communication with directly connected peers. It achieves *confidentiality*, *authentication*, *integrity protection* and *replay protection* using various cryptographic primitives. CORE does not offer reliable or in-order communication. nor routing

Distributed Hash Table (DHT) service

The DHT service implements a generic distributed hash table (details in [EvGr11] by Evans and Grothoff) to store key-value pairs in the peer-to-peer network. The DHT service API offers methods to `PUT`, `GET` and `MONITOR` DHT records. The `MONITOR` functionality enables clients to get notifications when a new value is added for a given key.

store retrieve values in the dht and monitor...

CADET service

The CADET offers secure end-to-end communication between peers. It builds on the CORE service and implements a routing protocol to allow communication between peers that are not directly connected. It supports reliable and in-order transmission of messages and implements flow and congestion control.

STATISTICS service

The STATISTICS service offers a central repository for ~~other~~ GNUnet subsystems to publish ~~unsigned 64-bit integer~~ statistics about the system (e.g., number of connected CORE peers, number of `PUT` requests issued to DHT, etc). Each statistics record is a tuple:

- `subsytem`: Name of the GNUnet subsystem to which this record belongs.
- `name`: Record name.
- `value`: The unsigned 64-bit integer value.
- `persistence`: A flag that determines if the record persists after a service restart.

Using the STATISTICS API, clients can `STORE`, `RETRIEVE` and `WATCH` STATISTICS record. The `WATCH` functionality enables clients to get notifications when the value of a given (subsytem, name) combination changes.

PEERSTORE service

The PEERSTORE service offers persistent per-peer local storage of arbitrary data. Each PEERSTORE data record is a tuple:

- `subsystem`: Name of the GNUnet subsystem to which this record belongs.
- `peer-identity`: Identity of the peer to which this record relates.
- `key`: Record key string.
- `value`: Record binary value of arbitrary size.

you use quite a lot of different styles:
small \tt
capital \tt
italic
capital italic

confusing

PEERSTORE uses plugins to implement data storage and retrieval (e.g. using an sqlite database). The API offers clients the ability to `STORE`, `RETRIEVE` and `WATCH` PEERSTORE records. The `WATCH` functionality enables clients to get notifications when the value of a given (subsytem, peer-identity, key) combination changes.

TESTBED service

The TESTBED service offers functionality to run multi-peer deployments on a single or multiple hosts. It supports creating, starting, stopping and destroying peers, creating/destroying connections between peers and starting/stopping services on peers among other functions. These functions are exposed through the service API.

I think you should not use "MI" as abbreviation
but something like "monitoring system"

6.2 System Services

Our monitoring infrastructure (MI) is implemented using two GNUnet services: the SENSOR service and the SENSORDASHBOARD service. The SENSOR service is used by peers ~~wishing to~~ participate in the MI as monitoring peers, implementing the functions of sensor data collection, anomaly detection and reporting (design described in 5.1, 5.2 and 5.3 respectively). The SENSORDASHBOARD service is used by ~~peers wishing to~~ act as a collection point and/or update point for other peers in the network (design described in 5.4 and 5.5 respectively).

always
Section x

The following is a detailed description of the structure and implementation of both services.

6.2.1 SENSOR ~~Service~~

component

The SENSOR service is the core of the monitoring infrastructure, it is divided into the following main components:

mention that components will be described

- Main service implementation which runs the following sub components:

Remove one level of bullets

  - *sensor monitoring*: collects measurements from defined sensors and saves them in PEERSTORE.
  - *sensor analysis*: monitors PEERSTORE for measurements values collected by the *sensor monitoring* subcomponent and applies anomaly detection methods on the values.
  - *sensor reporting*: implements reporting sensor values and anomalies to collection points and other peers in the network as well as receiving anomaly reports from other peers.
  - *sensor update*: contacts update points for new sensor updates and pulls any necessary updates.

- API used by clients and other GNUnet services to access some of the service's functionality.
- Command line interface to the service API.
- Utilities library: a library of common sensor-related functionality that might be needed by other services.

As mentioned in 6.1, each GNUnet service has an associated configuration file for user configuration. Table 6.1 describes all configuration parameters for the SENSOR service and its default values. Configuration parameters are categorized into sections corresponding to the subcomponents that require the configuration value.

why empty?

Section name	Configuration name	Description	Value type	Default value
sensor	START_MONITORING	Start the monitoring subcomponent	YES / NO	YES
sensor	START_REPORTING	Start the reporting subcomponent	YES / NO	YES
sensor	START_ANALYSIS	Start the analysis subcomponent	YES / NO	YES
sensor	START_UPDATE	Start the update subcomponent	YES / NO	YES
sensor	SENSOR_DIR	Path to the sensor definitions directory	Path	<GNUnet installation directory>
sensor-analysis	MODEL	Name of the anomaly detection model to be used	string	gaussian
sensor-analysis	CONFIRMATION_COUNT	Number of subsequent analysis results required to change sensor anomaly status	unsigned integer	1
sensor-model-gaussian	TRAINING_WINDOW	Training period for the gaussian model	unsigned integer	400
sensor-model-gaussian	CONFIDENCE_INTERVAL	The number of standard deviations considered within normal	unsigned integer	8
sensor-model-gaussian	WEIGHT_INC	Increase in weight for each new measurement value	integer	0
sensor-reporting	POW_MATCHING_BITS	Minimum number of leading zero's required for anomaly report proof-of-work	unsigned integer	15
sensor-update	UPDATE_POINTS	List of peer identities of sensor update points	string	<empty>

Table 6.1: SENSOR service configuration

6.2.1.1 Main Service

The main service component starts by loading sensor definitions from the configured sensor definitions directory using the `load_all_sensors` function of the utilities library 6.2.1.4. It is responsible for for starting its subcomponents (according to the START_* configuration flags) and stopping them on service shutdown.

The service also listens for the following message types from clients:

- `GNUNET_MESSAGE_TYPE_SENSOR_GET`: Messages of this type contain a sensor name. The service responds with brief information about the requested sensor (name, version and description).
- `GNUNET_MESSAGE_TYPE_SENSOR_GETALL`: This is an empty message. The service responds with brief information about all defined sensors (name, version and description).
- `GNUNET_MESSAGE_TYPE_SENSOR_ANOMALY_FORCE`: This message is used for testing purposes and contains a sensor name. The service notifies the *sensor reporting* subcomponent that the status of the given sensor is now anomalous. No reply is sent to the client.

sensor monitoring

The *sensor monitoring* subcomponent starts by scheduling the execution of all defined sensors according to the *interval* defined in the sensor definition. When the time for executing the sensor arrives, the subcomponent checks if the sensor is enabled and that the current time is between the sensor *start time* and *end time* values. If so, the sensor is executed according to its *source* value which can be either *gnunet-statistics* or *process*.

For *gnunet-statistics* sources, a `GET` request is sent to the STATISTICS service with the STATISTICS *subsystem* and *name* set as the *gnunet_stat_service* and *gnunet_stat_name* defined in the sensor definition.

For *process* sources, the process name defined in the sensor definition as *ext_process* is checked if it exists either in the system `PATH` environment variable or in the folder associated with the sensor which exists in the sensor definitions directory and is named as `<sensor-name>-files`. The process is then executed with any arguments defined as *ext_args* in the sensor definition.

The result returned from both sources is checked if its data type matches the expected data type defined as *expected_datatype* in the sensor definition. The value is then saved in PEERSTORE with the `subsystem` name as "sensor", the `peer-identity` as the local peer identity, the `key` as the sensor name, the `value` as the execution result value and the `expiry` as the current date-time + the *lifetime* defined in sensor definition.

Any errors or failed checks throughout the above process triggers an error message and the related sensor is disabled.

respective

sensor analysis

The *sensor analysis* subcomponent uses plugins that implement the anomaly detection models. The `MODEL` configuration parameter specifies the plugin name to be loaded on subcomponent start. All plugins must implement the following functions to be usable:

- `create_model`: Creates a new model instance.

- `destroy_model`: Destroys a previously created model instance.
- `feed_model`: Feed a new value into the model instance. This function takes a value parameter of type `double` and returns a `boolean` value where 0 means that the fed value is normal and 1 means that it is anomalous.

The *gaussian*-model plugin 6.2.1.1 is currently the only implemented plugin.

The *sensor analysis* subcomponent starts by creating a model instance for each sensor that has the expected data type as *numeric* and sends a `WATCH` request to PEERSTORE to be notified with any new values saved by the *sensor monitoring* subcomponent. A flag representing the sensor anomaly status is initially set to *normal.* When a new value notification from PEERSTORE is received, the model instance corresponding to the sensor is fed with the received value. The feed function returns a `boolean` stating whether the fed value is considered normal or anomalous, it is then checked against the last $n-1$ (where n is the `CONFIRMATION_COUNT` configuration parameter) results received from the model, if all have the same value and are different from the current sensor anomaly status, the sensor anomaly status is flipped and a notification is sent to the *sensor reporting* subcomponent containing the sensor name and the new sensor anomaly status.

gaussian-model plugin

The gaussian-model plugin implements the algorithm described in 5.2.1, it implements the main functions required by the *sensor analysis* subcomponent as follows:

- `create_model`: a new model instance is created by initializing the model status values s_0, s_1 and s_2 to 0 and the initial weight w to 1.
- `destroy_model`: Cleanup any allocated memory for the model instance.
- `feed_model`: If the index of the fed value value passed is within the training window defined by the `TRAINING_WINDOW` configuration parameter, the model status values are updated and the function returns a 0 (no anomaly). Otherwise, the *weighted mean* and *weighted standard deviation* are calculated from the model status values and the fed value is checked if it lies within d standard deviations from the mean where d is defined by the `CONFIDENCE_INTERVAL` configuration parameter. This determines if the fed value is considered normal or anomalous and the result is returned.

sensor reporting

The *sensor reporting* subcomponent is responsible for sending value and anomaly report to a *collection point* and exchanging anomaly reports with neighbouring peers. Neighbouring peers are the peers that have direct connection with the local peer on level of the CORE service. The subcomponent keeps track of neighbouring peers by connecting to the CORE service and requesting notifications for any peer connection/disconnection events.

For value reporting, the subcomponent sends `WATCH` requests to PEERSTORE to monitor for new sensor values for sensors that have a `COLLECTION_POINT` defined and the `COLLECTION_INTERVAL` value set in the sensor definition. The *sensor reporting* subcomponent only keeps track of the latest sensor value received from PEERSTORE and the timestamp of receiving the value and sends a value report to the collection point every collection interval. The value report is a message of type `GNUNET_MESSAGE_TYPE_SENSOR_READING` and is sent to the peer identity of the *collection point* using CADET. The message carries the sensor name, sensor version number, timestamp and value. If the report fails to send, it is discarded and a new report is attempted at the next collection interval.

Table 6.2 shows the structure of an anomaly report message. The *sensor name* and *sensor version number* identify the sensor this report belongs to. The *anomaly status* is a boolean specifying whether the status of the sensor is normal or anomalous. *Anomalous neighbours* is the ratio of the number of neighbouring peers who reported an anomaly for the same sensor to the total number of neighbouring peers. The *proof-of-work* is a numerical value that when appended to the anomaly report (the part starting from *sensor name* up to *timestamp*), the scrypt hash [Perc09] of the resulting block has a minimum number of leading zero's defined by the `POW_MATCHING_BITS` configuration parameter. The whole block including the *proof-of-work* is signed using the local peer's private key. Generating a *proof-of-work* and *signature* for an arbitrary block of data and their verification (in case of a received block) is a function supplied by the sensor utilities library 6.2.1.4.

Sensor name
Sensor version number
Anomaly status
Anomalous neighbours
Timestamp
Proof-of-work
Signature

you can use latex bytefield package to create nice messages

Table 6.2: Anomaly report

2

The subcomponent listens for anomaly reports from neighbouring peers through GNUnet CORE service. On receiving an anomaly report, the *proof-of-work* and *signature* are verified using the utilities library. If they are valid, the sender peer identity is added to the list of anomalous neighbours for the sensor specified in the report.

When an anomaly notification is received from the *sensor analysis* subcomponent, an anomaly report is generated by assembling the main report information described above and generating a *proof-of-work* and *signature* using the utilities library. The final report is sent to all neighbouring peers through CORE. If the `collection_point` is defined and `report_anomalies` flag is enabled in the sensor definition, the report is also sent to the *collection point* using GNUnet CADET service.

In case connecting or sending reports to the *collection point* fails, anomaly reports destined to it are queued and the connection is retried every 1 minute. When the connection succeeds, previous reports are sent in the same order in which they were generated.

sensor update

The *sensor update* subcomponent reads a list of update points from the configuration parameter `UPDATE_POINTS`. If none are defined, the subcomponent shuts down immediately with a warning message.

Every update interval (hard-coded as 1 day), the subcomponent sends a message to the first functioning update point (all update points are initially assumed to be functioning) requesting a list of available sensors. The connection and communication with the update point is performed through GNUnet CADET service. It is expected that the update point replies with a series of messages carrying brief information (name and version) about the available sensors.

For each brief sensor information message received, the subcomponent checks if a sensor with the same name does not exist locally or exists but with a smaller version number. If so, a request for the full sensor information is sent to the update point which is a message containing the sensor name and version number being requested. A response message is expected which carries the content of the sensor definition file and the contents of any associated script files. These file are written to the configured sensor definitions directory and the sensor service is restarted for the updates to take place.

If an error occurs during connecting or communicating with the update point, the update point is marked as failed and the process is repeated with the next defined update point. If there are no more functioning update points defined, the status of all update points are reset to functioning and the subcomponent retries the update process on the next update interval (1 day).

6.2.1.2 API

The sensor API provides a method to access the functionality exposed by the sensor service to clients. It handles connecting to the service, sending, receiving and parsing messages and queuing client requests.

The sensor API implements the following functions:

- `iterate`: Iterate information about all defined sensors or a specific sensor given its name.
- `force_anomaly`: Used for test purposes to force an anomaly status on a given sensor name.

6.2.1.3 Command Line Interface

The CLI uses the sensor API to allow the user access to the service functionality through the command line. The tool supports the following command line arguments:

- `-a, -all`: Print information about all defined sensors.
- `-f, -force-anomaly SENSORNAME`: Force an anomaly on the given sensor name, used for testing purposes only.
- `-g, -get-sensor SENSORNAME`: Print information about a single sensor.

6.2.1.4 Utilities Library

The sensor utilities library implements common sensor-related functionality such as loading of sensor definitions from a given directory, generating scrypt proof-of-work and signature for an arbitrary block of data and verifying a given proof-of-work and signature.

Loading of sensor definitions is performed by parsing sensor definition files in a given directory and extracting expected sensor definition fields 5.1.1. A valid sensor definition file consists of one section beginning with a `[sensor-name]` header and followed by multiple `name=value` entries for the definition fields.

Generating a proof-of-work is a time-consuming process, thus it is performed asynchronously. Given an arbitrary block of data, the library looks for the first unsigned 64-bit integer (starting with 0) that when appended to the given block, the resulting block has an scrypt hash with a minimum number of leading zero's (the required number is passed as parameter to the generation function). When a valid proof-of-work is found, the whole message is signed with the given private key and the block along with the proof-of-work and signature are returned.

Verification is done by first checking if the signature is valid for the given public key. If yes, the data block and the proof-of-work are hashed using scrypt and the resulting hash is checked if it contains the minimum number of leading zero's required.

6.2.~~2 SENSORDASHBOARD Service~~ dashboard component

The SENSORDASHBOARD service is used by peer~~s wishin~~g to act as a collection point and/or update point for other peers in the network. Therefore, the service is divided into the following subcomponents:

- *collection point*: Receives value and anomaly reports from other peers and stores them persistently.
- *update point*: Receives and handles requests for sensor definitions from other peers.

collection point

The *collection point* subcomponent listens for the following message types arriving through CADET service:

- `GNUNET_MESSAGE_TYPE_SENSOR_READING`: sensor value message.
- `GNUNET_MESSAGE_TYPE_SENSOR_ANOMALY_REPORT`: anomaly report message.

On receiving a sensor value message, the subcomponent verifies the that the message is valid by checking that the given sensor name exists locally and the version number matches and the value data type matches the expected data type in the local sensor definition. If the report is valid, it is saved in GNUnet PEERSTORE.

On receiving an anomaly report message, the subcomponent verifies the given signature and proof-of-work using the sensor utilities library 6.2.1.4. If the verification succeeds, the report is saved in GNUnet PEERSTORE.

It is expected that other applications wishing to handle value or anomaly reports will issue `WATCH` requests to PEERSTORE to be notified when a new valid report is saved or `ITERATE` through previous reports saved in PEERSTORE.

update point

The *update point* subcomponent listens for the following message types arriving through CADET service:

- `GNUNET_MESSAGE_TYPE_SENSOR_LIST_REQ`: request for list of available sensors.
- `GNUNET_MESSAGE_TYPE_SENSOR_FULL_REQ`: request for full definition of a single sensor.

A request for list of available sensors is handled by creating a series of messages, one for each locally available sensor where each message carries the sensor name and version number. The messages are sent to the requesting peer through CADET.

A request for a full definition of a single sensor is first verified by checking that the given sensor name exists locally. If it does, the subcomponent reads the raw binary from the sensor definition file and any associated scripts/binaries in the `<sensor-name>-files` directory and compiles them into a single message which is sent back to the requesting peer.

more structure
subsection eval gauss
subsection eval Infrastructure

you describe the dataset? really?

7. Evaluation

in this section we evaluate the performance of the detection mechanism and the overall quality and performance of the monitor ing infrastructure

We evaluate several aspects of the monitoring infrastructure starting with analysing the complexity of the gaussian model anomaly detection method. We then look at the memory and storage consumption of the whole system which can be used as reference to determine if the system can be deployed on hardware with limited capabilities such as wireless sensor networks. We then describe the KDD 1999 dataset which is a large dataset of labelled network capture information mainly used for evaluating intrusion detection systems. We use the KDD dataset to arrive the optimum values for the free parameters in our system. Lastly, we perform large scale simulation experiments to determine if the monitoring infrastructure satisfies the objectives that we set earlier.

7.1 Complexity of Gaussian Model

The gaussian model anomaly detection method implemented within our system 5.2.1 uses an incremental approach, values fed into the model are used to update a fixed number of internal status variables. No record of previous values need to be kept and the number of internal status variables does not change. The anomaly detection process is performed by calculating two additional values (*weighted mean* and *weighted standard deviation*) and checking that the new value fed to the model lies within a pre-configured number of standard deviations from the mean. This process is constant over any number of values.

This property makes the complexity of the gaussian model constant with regard to processing, memory and storage requirements.

7.2 System Memory and Storage Requirements

To analyse the memory and storage consumption, we run a single GNUnet peer on a computer running Debian wheezy 64bit with Linux kernel 3.14. We run only the GNUnet services required by our system plus the sensor service which implements our monitoring infrastructure. Table 7.2 shows all the GNUnet services running with a brief description of each.

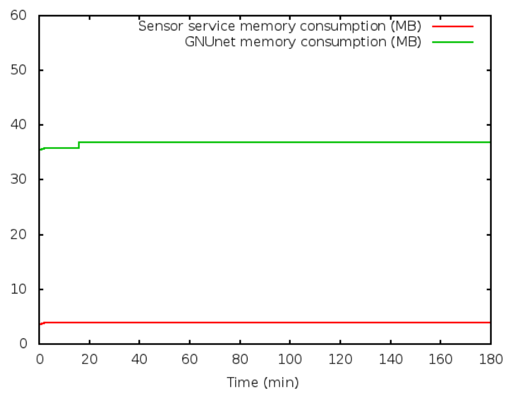
The sensor services runs with default configuration and all subcomponents 6.2.1.1 and default sensors 5.2 enabled. All sensors are configured to report anomalies to a remote collection point running on a different computer.

We run the peer for three hours, during which we monitor the memory consumption of the sensor service and the total memory consumption of all GNUnet services running. We also

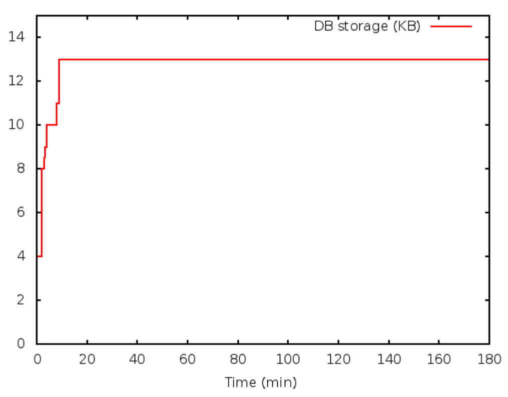
Service name	Description
arm	Automatic Restart Manager - handles starting/stopping other GNUnet services
statistics	Collects statistics from different GNUnet subsystems
core	Offers secure communication with directly connected peers
cadet	Offers secure communication and routing to any peer in the network
peerstore	Offers persistent storage of arbitrary data
transport	Establishes connections to other peers using different communication protocols
peerinfo	Stores information about known peers
dht	Implements a distributed hash table
ats	Automatic transport selection and outbound bandwidth determination
nse	Network size estimation
sensor	Monitoring and anomaly detection

Table 7.1: GNUnet services required by our monitoring infrastructure

monitor the storage the storage consumption by monitoring the size of the sqlite database which the peerstore service uses to store records passed to it by the sensor service. Tables 7.1(a) and 7.1(b) show memory and storage consumption over time.



(a) Memory consumption of sensor service and all GNUnet services



(b) Storage consumption of sensor service

From figure 7.1(a) we can see that the memory consumption of the sensor service stabilizes very quickly at a value of about 4 MB and the total consumption of all GNUnet services stabilizes at a value of about 37 MB. This also occurs with the storage consumption in figure 7.1(b) at about 13 KB. To understand why the storage does not increase even while collecting new measurements, we should note that measurements stored in peerstore has an expiry time that is by default the same as the collection interval unless otherwise specified in the sensor definition file.

Methodology 7.3 ~~KDD 1999 Dataset~~

you have to start why you use a dataset ... objective and motivation!!

The KDD dataset[1] was used for The Third International Knowledge Discovery and Data Mining Tools Competition to evaluate intrusion detection systems. The original dataset is a nine-week raw TCP dump data from a local-area network that is attacked with different types of attacks (e.g. denial-of-service, password bruteforcing, etc). The KDD organizers further processed the data by extracting around five million unique connection records from the TCP dump. Each record containing 41 different connection-related features such as duration, protocol type, service, etc. 311029 of these records were correctly labelled with the attack type if any.

[1]http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

We downloaded the corrected dataset and extracted the numeric features (34 features) from the records since our anomaly detection algorithm does not support symbolic input. We also reduced the attack labels to binary label of normal/attack. To evaluate the anomaly detection algorithm against the KDD dataset, we create a model for each of the features, feed the numeric values of each feature into the corresponding model and analysing the model output. If one or more of the models created detects an anomaly, we consider the connection record to be anomalous (or in this case, attack). The detected result is compared against the actual label in the dataset and the number of true positives, true negatives and the accuracy are determined.

7.4 Parameter Optimization

of what???

In this section we analyse the free parameters used in the system trying to arrive at the optimum value to be used as default. The parameters *standard deviations*, *training period* and *weight increase* are specific to the gaussian model anomaly detection method 5.2.1. The *confirmation count* parameter is related to anomaly detection in general but not to any specific anomaly detection method 5.2. The parameter *proof-of-work leading zero's* is related to the proof-of-work concept used to make attacks on the system more expensive 5.3.3.

7.4.1 Standard Deviations

This parameter represents the number of standard deviations away form the mean that is considered within normal. Any values lying outside this interval are considered anomalous. According to the three-sigma rule[Puke94], in a normally distributed random variable, 68.27% of values lie within one standard deviation from the mean, 95.45% of values lie within two standard deviations from the mean, 99.73% of values lie within three standard deviations from the mean. But since the underlying distribution of the data considered by the system is not necessarily normal or stationary, we try to arrive at the optimum value of normal standard deviations by testing the results of using values from the range of 1 to 10 which is assumed to be covering all possible efficient parameter values.

Table 7.2 shows the results of using different parameter values and evaluating them against the KDD dataset. TP, TN, FP, FN and Acc % refer to true positives, true negatives, false positives, false negatives and accuracy percentage respectively. Accuracy percentage is calculated using equation 7.1. Figure 7.1 shows a line-plot of accuracy percentage in relation to the parameter value.

$$\text{Accuracy} = \frac{\text{true positives} + \text{true negatives}}{\text{true positives} + \text{true negatives} + \text{false positives} + \text{false negatives}} \times 100 \quad (7.1)$$

The highest accuracy was achieved at a normal standard deviations value of 8 which is used as the default value during further evaluations.

7.4.2 Training Period

~~It is assumed tha~~t the model requires an initial period of readings to arrive at an optimum estimate of the mean and variance of the underlying dataset. During this period, evaluating new data points against the model in its current state will result in a lower accuracy of anomaly detection. This period represents the number of initial data points to be used for updating the model without performing anomaly detection.

The size of the training period is considered to be a compromise. A very small training period size might cause a higher rate of false positives due to a yet inaccurate estimate of

Parameter	TP	TN	FP	FN	Acc %
1	250333	174	60419	103	80.541364
2	250264	394	60199	172	80.589913
3	240592	13580	47013	9844	81.719711
4	240412	15418	45175	10024	82.252780
5	240409	19494	41099	10027	83.562304
6	240403	23111	37482	10033	84.723289
7	240259	30671	29922	10177	87.107633
8	240130	38608	21985	10306	89.618010
9	148862	43859	16734	101574	61.962389
10	148455	48437	12156	101981	63.303422

Table 7.2: Normal standard deviations evaluation results

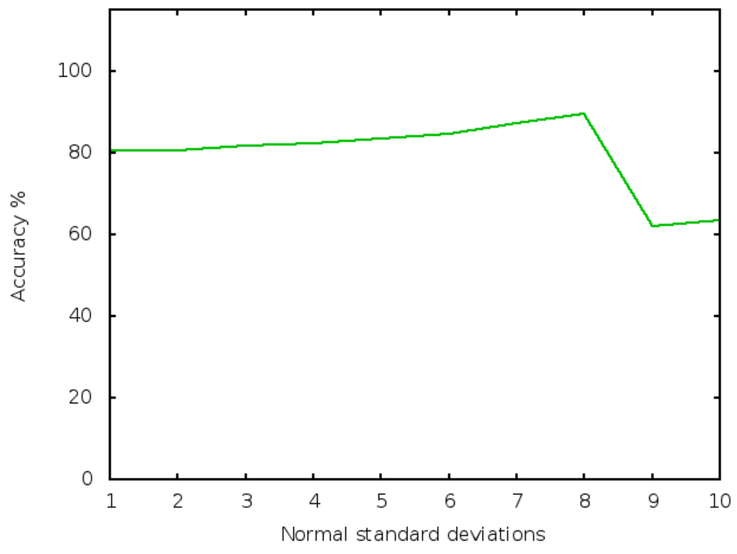


Figure 7.1: Normal standard deviations evaluation results

the mean and variance of the underlying dataset. A very large training period size might cause a higher rate of false negatives due to anomalies present within the training set going undetected.

For our purposes, we try to find the optimum value of the training period by evaluating different possible values against the KDD dataset since it best resembles our target environment. We evaluate the values starting with the smallest possible size of 1 up to 1000 data points with a step size of 10. Table 7.3 shows the evaluation results. Figure 7.2 shows a line-plot of accuracy percentage in relation to the parameter value.

The highest accuracy is achieved at a training window size of 400 which is used as the default training period during further evaluations.

7.4.3 Weight Increase

As described in 5.2.1, the gaussian model calculates *weighted mean* and *weighted standard deviation* values by assigning a higher weight to more recent values. The first value is assigned a weight of 1 and the weight is incremented by a fixed increase with each new value.

To find the optimum value of the weight increase, we evaluate different values in the range of 0 to 5 with an increment of 0.5 against the KDD dataset. Table 7.4 shows the

Parameter	TP	TN	FP	FN	Acc %
1	250435	3	60590	1	80.519180
100	240130	38608	21985	10306	89.618010
200	237644	43334	17259	12792	90.338200
300	237644	43335	17258	12792	90.338521
400	237644	43336	17257	12792	90.338843
500	233154	43399	17194	17282	88.915503
600	232509	43588	17005	17927	88.768893
700	232507	43590	17003	17929	88.768893
800	232502	45317	15276	17934	89.322539
900	68774	46646	13947	181662	37.109080
1000	68774	46646	13947	181662	37.109080

Table 7.3: Training period evaluation results

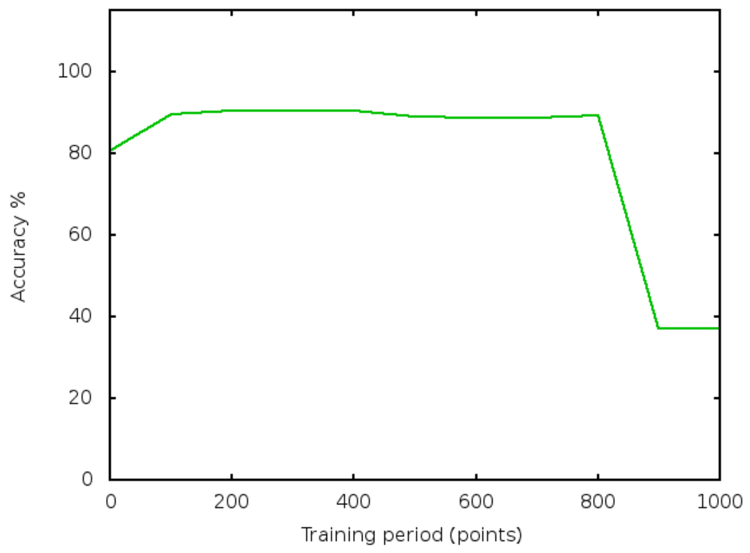


Figure 7.2: Training period evaluation results

evaluation results. Figure 7.3 shows the a line-plot of the accuracy percentage in relation to the parameter value.

The highest accuracy is achieved at a weight increment value of 0 which means that all values are assigned equal weights. The weight increment default value is set to 0 during further evaluations.

7.4.4 Confirmation Count

Confirmation count is the number of similar and subsequent anomaly detection results that are required to change the anomaly status of a sensor. For example, if the *confirmation count* is 3, 3 subsequent anomalous values are required to change the related sensor status from *normal* to *anomalous* and vice versa.

The value of the *confirmation count* depends on the monitored environment and system user's requirements for anomaly detection. Smaller *confirmation count* decreases the possible granularity of detected anomalies. In some environments, short-term anomalies can be caused by measurement errors or brief system faults which might be uninteresting from the user's point of view. In such cases, a higher *confirmation count* is set.

Parameter	TP	TN	FP	FN	Acc %
0	237644	43336	17257	12792	90.338843
0.5	250069	772	59821	367	80.648750
1	250081	740	59853	355	80.642320
1.5	250081	740	59853	355	80.642320
2	250081	740	59853	355	80.642320
2.5	250081	740	59853	355	80.642320
3	250081	740	59853	355	80.642320
3.5	250081	740	59853	355	80.642320
4	250081	740	59853	355	80.642320
4.5	250081	740	59853	355	80.642320
5	250081	740	59853	355	80.642320

really exactly the same?

Table 7.4: Weight increase evaluation results

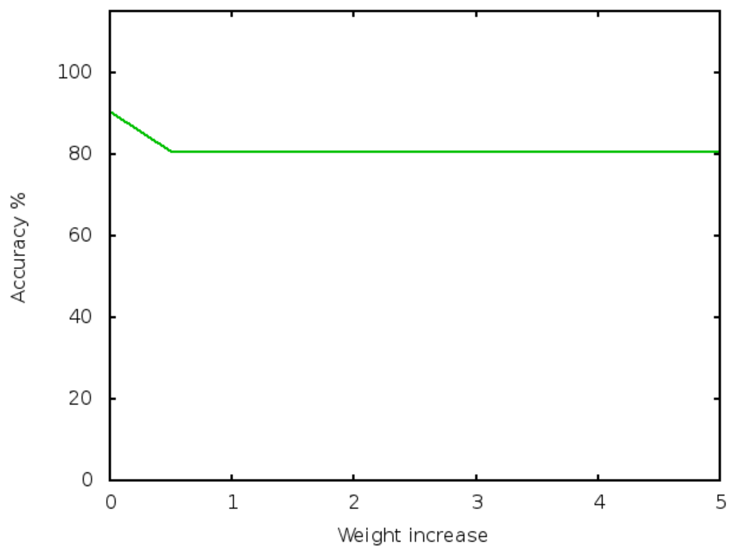


Figure 7.3: Weight increase evaluation results

We use a default value of 1 for the *confirmation count* to make the solution generic enough to cover all use cases even at the expense of possible uninteresting detection events.

7.4.5 Proof-of-work Leading Zero's

As described in 5.3.3, anomaly reports require a generated proof-of-work to be accepted by other peers or collection point. This is used to make it more difficult for malicious peers to flood the system with fake anomaly reports. The proof-of-work is a number which when appended to the anomaly report data block, the scrypt hash [Perc09] of the resulting block has a minimum number of leading zero's. The number of leading zero's required sets the difficulty (in terms of processing time) of generating the required proof-of-work.

We try to estimate the average processing time required to generate a proof-of-work for different values of *leading zero's*. The system used for this runs an Intel Core i7-4700MQ CPU with 8 GB of RAM. The operating system is Debian wheezy 64bit with Linux kernel 3.14. For each value of *leading zero's*, 10 random message blocks of size 1024 bytes are generated and a proof-of-work is calculated for each of these blocks. The average time needed to generate the proof-of-work for each of the 10 message blocks is recorded. Table 7.5 shows the results for required *leading zero's* from 1 to 20.

Leading zero's	Time (seconds)
1	0.006
2	0.0065
3	0.0103
4	0.0136
5	0.0304
6	0.0632
7	0.0651
8	0.1438
9	0.3917
10	0.4079
11	1.298
12	2.8741
13	3.8325
14	9.4033
15	15.6575
16	27.7812
17	39.2347
18	69.471
19	145.2158
20	311.1476

what is"behavior"? be precise

Table 7.5: Average processing time in seconds required to generate a proof-of-work

It is required that the processing requirements for generating the proof-of-work be expensive enough to discourage attacks that flood the system with fake anomaly reports but not too expensive as to discourage legitimate users or delay the anomaly reporting process by too long. For this purpose, we use a default value of 15 *leading zero's.*

7.5 Experiments

I thought you do not "global scale"?

To evaluate the behaviour of the system on a global scale, we perform an experiment by running multiple GNUnet peers randomly connected on a testbed, then introducing an artificial anomaly into the network and recording the anomalies detected on the peers.

7.5.1 Setup

The tool developed for running the experiment takes as input the number of GNUnet peers to start p and the total number of links l. Using GNUnet testbed service [2], the tool starts the required number of peers and randomly create the number of required links between them. All peers are started with the minimum GNUnet services required to perform connections and routing (we do not need some of the default GNUnet services such as file-sharing, revocation, etc). The *sensor* service is started on all peers with the default configurations and default sensor list and the *sensordashboard* service is started on peer 0 which acts as the collection point as well as a regular sensor peer. All sensor definitions are edited to enable anomaly reporting to peer 0.

After all peers and services are successfully started and the links between them created, the tool pauses for the length of training period required by the gaussian models built on top of the running sensors. The training period is calculated by multiplying the maximum measurement interval defined in the sensor definitions and the number of training data points required by the gaussian model as defined in the system configuration.

[2] https://gnunet.org/content/gnunets-testbed-subsystem

After the training period is over, the tool presents a choice for simulating two different types of anomalies into the system. The first method is by disconnection a number of existing links, for this, the tool prompts for a list of peer number pairs, each pair representing an existing link to be disconnected by the tool. The tool uses the API for the GNUnet TRANSPORT service 6.1 to force the disconnection. After the requested links are disconnected, the tool pauses for one minute to give the peers enough time for detecting and reporting anomalies. After which the tool prompts for a new list of peer number pairs, each pair representing a new link to be established. This is used in the case that due to the disconnections performed earlier, a group of peers now can not send any messages to the collection point. By restoring some connectivity, we allow the peers to send any previous anomaly reports to the collection point.

The second method of simulating anomalies is by changing the values written to the GNUnet statistics service 6.1 which acts as a repository for other local GNUnet services to publish runtime statistical information about the system and is a possible source of sensor measurements including multiple sensors that monitor peer connectivity. The tool takes as input a number A which is the number of peers to simulate an anomaly on, it then chooses A peers at random and writes a value of 0 to the GNUnet statistics entry that represents the number of connected peers on the GNUnet CORE level.

A visualisation wrapper is built around the tool that monitor its logs for the following events: 1) peers running and links established, 2) link disconnected, 3) link established, 4) anomaly report received by collection point. On each event, the wrapper draws a graph of the current state of the network and marks peers that reported anomalies.

7.5.2 Experiment 1

For the first experiment we start 10 peers that are randomly connected with an average of 4 links per peer (20 total links). Figure 7.4 shows the initial setup.

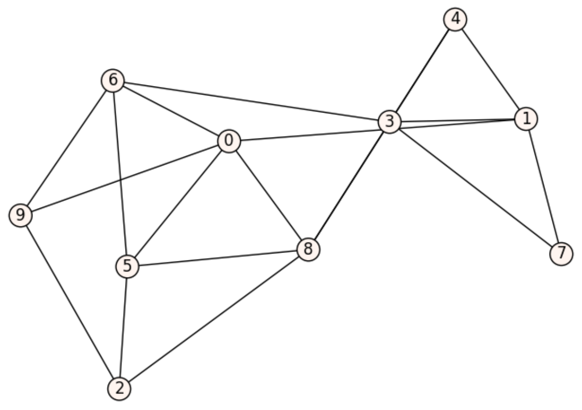


Figure 7.4: Experiment 1: initial setup

We simulate an isolation event where peers 1, 4 and 7 are disconnected from the rest of the network by disconnecting the links (4,8), (4,3), (1,0), (1,3) and (7,3). This simulates the case where an underlay connectivity issue causes a network to disconnect from the global Internet. Figure 7.5 shows the network after the disconnection.

After disconnection, the collection point receives an anomaly report from peer 3 with 0% neighbouring anomalous peers (figure 7.6). This is as expected since the number of peer

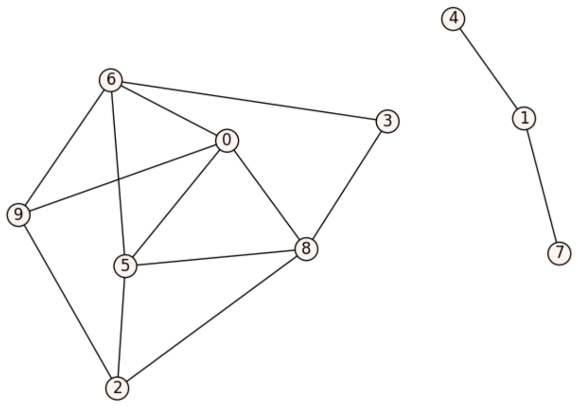


Figure 7.5: Experiment 1: after disconnection

connections for peer 3 went down from 5 to 2 peers. No more anomaly reports were received. At this point, any anomalies detected by peers 1, 4 and 7 are not reported since there is no path to the collection point running on peer 0.

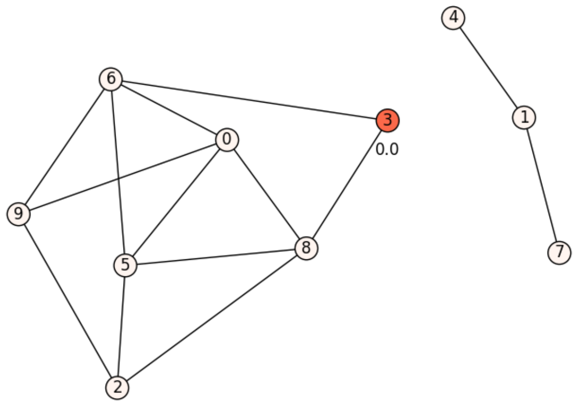
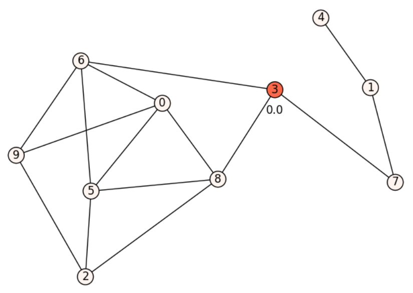
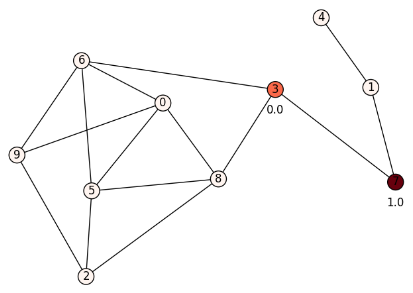


Figure 7.6: Experiment 1: first anomaly report

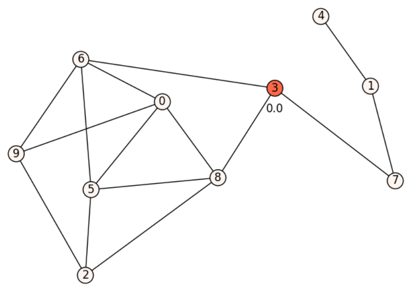
To receive anomaly reports from peers 1, 4 and 7, we re-establish the link (3,7). The series of figures 7.7 shows the anomaly reports received approximately 2 minutes after re-establishing the link. Figure 7.7(a) shows the network before receiving any new anomaly reports. In figure 7.7(b) we receive an anomaly report from peer 7 with 100% anomalous neighbourhood. This is the report generated before reconnection and peer 7 had only 1 neighbour which is peer 1. The next report (figure 7.7(c)) shows that peer 7 is out of the anomaly status, this is expected since the number of connections for peer 7 returned to a normal value of 2. We then receive two more old anomaly reports from peer 4 and peer 1 with 100% anomalous neighbourhood (figures 7.7(d) and 7.7(e)). The last report (figure 7.7(f)) shows that peer 1 received an update from peer 7 that its back to the normal status, therefore the anomalous neighbourhood is changed to 50% and a new report is sent to the collection point.



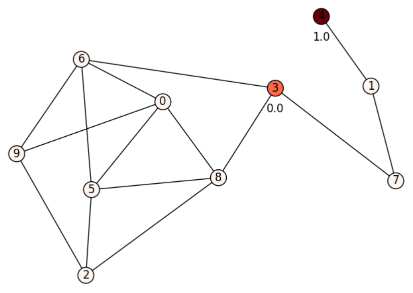
(a)



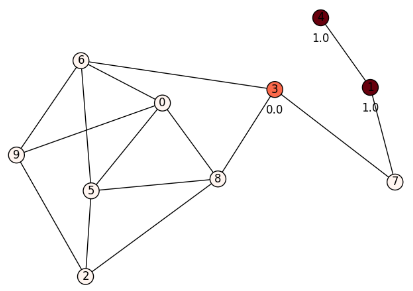
(b)



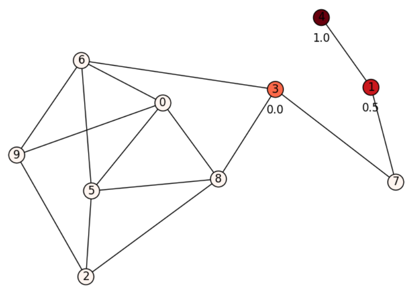
(c)



(d)



(e)



(f)

Figure 7.7: Experiment 1: subsequent anomaly reports

7.5.3 Experiment 2

For the second experiment we start 100 peers that are randomly connected with an average of 8 links per peer (400 total links). Figure 7.8 shows the initial setup.

We simulate anomalies by picking 20 random peers and writing a value of 0 to the peer's GNUnet statistics entry that represents the number of connected peers on the GNUnet CORE level. This is expected to immediately trigger an anomaly on the chosen peers. Figure 7.9 shows the total number of anomaly reports received over time (seconds) where

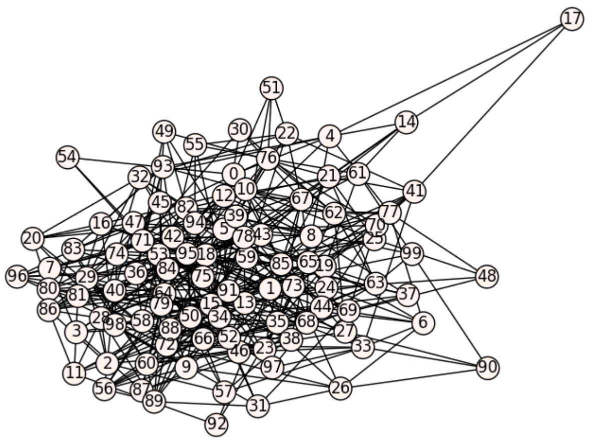


Figure 7.8: Experiment 2: initial setup

time $t = 0$ is the time of triggering the first anomaly. The first report was received at time $t = 13$ and the last report was received at time $t = 268$. The total number of reports received is 76. Figure 7.10 shows the status of the network after all the anomaly reports have been received. The red peers are the anomalous peers with the shade of red and the number written below the peer representing the percentage anomalous neighbourhood.

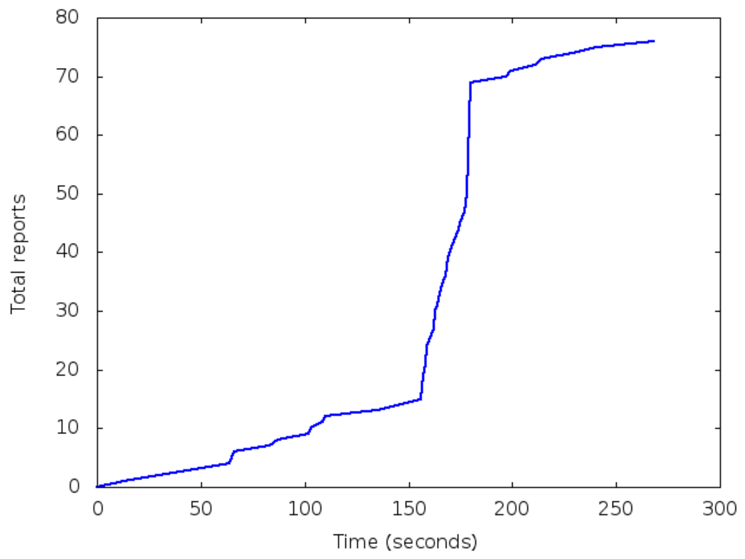


Figure 7.9: Experiment 2: number of anomaly reports received over time

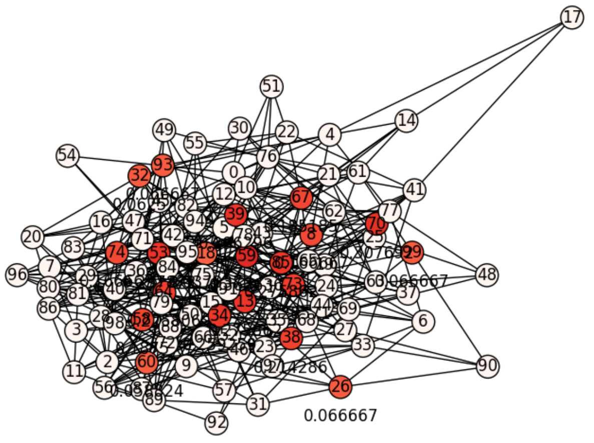


Figure 7.10: Experiment 2: final status

8. Conclusion and Outlook

Bla fasel…

(Keine Untergliederung mehr!)

Bibliography

[ABAB] Yuan Yao A, Abhishek Sharma B, Leana Golubchik A und Ramesh Govindan B. Online Anomaly Detection for Sensor Systems: a Simple and Efficient Approach.

[ArGS06] Marc S. Artigas, Pedro García und Antonio F. Skarmeta. DECA: A Hierarchical Framework for DECentralized Aggregation in DHTs. In *Proceedings of the 17th IFIP/IEEE International Conference on Distributed Systems: Operations and Management*, DSOM'06, Berlin, Heidelberg, 2006. Springer-Verlag, S. 246–257.

[BaMe07] Sabyasachi Basu und Martin Meckesheimer. Automatic Outlier Detection for Time Series: An Application to Sensor Data. *Knowl. Inf. Syst.* 11(2), Februar 2007, S. 137–154.

[BGMGM03] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis und Rajeev Motwani. Estimating Aggregates on a Peer-to-Peer Network. Technical Report 2003-24, Stanford InfoLab, April 2003.

[ChBK09] Varun Chandola, Arindam Banerjee und Vipin Kumar. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41(3), Juli 2009, S. 15:1–15:58.

[DaSt05] Mads Dam und Rolf Stadler. A generic protocol for network state aggregation. In *In Proc. Radiovetenskap och Kommunikation (RVK*, 2005, S. 14–16.

[DeNe85] Dorothy E Denning und Peter G Neumann. Requirements and model for IDES-a real-time intrusion detection expert system. *Document A005, SRI International* Band 333, 1985.

[Denn87] D.E. Denning. An Intrusion-Detection Model. *Software Engineering, IEEE Transactions on* SE-13(2), Feb 1987, S. 222–232.

[DiMS04] Roger Dingledine, Nick Mathewson und Paul Syverson. Tor: The Second-generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, Berkeley, CA, USA, 2004. USENIX Association, S. 21–21.

[EAPP$^+$02] Eleazar Eskin, Andrew Arnold, Michael Prerau, Leonid Portnoy und Sal Stolfo. A Geometric Framework for Unsupervised Anomaly Detection: Detecting Intrusions in Unlabeled Data. In *Applications of Data Mining in Computer Security*. Kluwer, 2002.

[EGKM04] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec und Laurent Massoulieacute;. Epidemic Information Dissemination in Distributed Systems. *Computer* 37(5), Mai 2004, S. 60–67.

[EvGr11] N.S. Evans und C. Grothoff. R5N: Randomized recursive routing for restricted-route networks. In *Network and System Security (NSS), 2011 5th International Conference on*, Sept 2011, S. 316–321.

[GoRS96] David M. Goldschlag, Michael G. Reed und Paul F. Syverson. Hiding Routing Information. In *in Information Hiding*. Springer-Verlag, 1996, S. 137–150.

[Grub50] Frank E Grubbs. Sample criteria for testing outlying observations. *The Annals of Mathematical Statistics*, 1950, S. 27–58.

[HoAu04] Victoria Hodge und Jim Austin. A Survey of Outlier Detection Methodologies. *Artif. Intell. Rev.* 22(2), Oktober 2004, S. 85–126.

[JaVa93] Javitz und Valdes. The NIDES Statistical Component: Description and Justification. mar 1993.

[JeMB05] Márk Jelasity, Alberto Montresor und Ozalp Babaoglu. Gossip-based Aggregation in Large Dynamic Networks. *ACM Trans. Comput. Syst.* 23(3), August 2005, S. 219–252.

[j(Ps03] jrandom (Pseudonym). Invisible Internet Project (I2P) Project Overview. Design document, August 2003.

[KeDG03] D. Kempe, A Dobra und J. Gehrke. Gossip-based computation of aggregate information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, Oct 2003, S. 482–491.

[KeKD01] David Kempe, Jon Kleinberg und Alan Demers. Spatial Gossip and Resource Location Protocols. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, New York, NY, USA, 2001. ACM, S. 163–172.

[KnNT00] Edwin M. Knorr, Raymond T. Ng und Vladimir Tucakov. Distance-based Outliers: Algorithms and Applications. *The VLDB Journal* 8(3-4), Februar 2000, S. 237–253.

[Loes09] Karsten Loesing. Performance of Requests over the Tor Network. Technischer Bericht 2009-09-001, The Tor Project, September 2009.

[LoMD10] Karsten Loesing, StevenJ. Murdoch und Roger Dingledine. A Case Study on Measuring Statistical Data in the Tor Anonymity Network. In Radu Sion, Reza Curtmola, Sven Dietrich, Aggelos Kiayias, JosepM. Miret, Kazue Sako und Francesc SebÃ© (Hrsg.), *Financial Cryptography and Data Security*, Band 6054 der *Lecture Notes in Computer Science*, S. 203–215. Springer Berlin Heidelberg, 2010.

[MaSi03] Markos Markou und Sameer Singh. Novelty Detection: A Review - Part 1: Statistical Approaches. *Signal Processing* Band 83, 2003, S. 2003.

[MBDG09] Rafik Makhloufi, Grégory Bonnet, Guillaume Doyen und Dominique Gaïti. Decentralized Aggregation Protocols in Peer-to-Peer Networks: A Survey. In *Proceedings of the 4th IEEE International Workshop on Modelling Autonomic Communications Environments*, MACE '09, Berlin, Heidelberg, 2009. Springer-Verlag, S. 111–116.

[McTL78] Robert McGill, John W Tukey und Wayne A Larsen. Variations of box plots. *The American Statistician* 32(1), 1978, S. 12–16.

[MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein und Wei Hong. TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks. *SIGOPS Oper. Syst. Rev.* 36(SI), Dezember 2002, S. 131–146.

[Parz62] Emanuel Parzen. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics* 33(3), 1962, S. pp. 1065–1076.

[Perc09] Colin Percival. Stronger key derivation via sequential memory-hard functions. *Self-published*, 2009, S. 1–16.

[Puke94] Friedrich Pukelsheim. The Three Sigma Rule. *The American Statistician* 48(2), 1994, S. 88–91.

[RLPB06] S. Rajasegarar, C. Leckie, M. Palaniswami und J.C. Bezdek. Distributed Anomaly Detection in Wireless Sensor Networks. In *Communication systems, 2006. ICCS 2006. 10th IEEE Singapore International Conference on*, Oct 2006, S. 1–5.

[ThJi03] M. Thottan und Chuanyi Ji. Anomaly detection in IP networks. *Signal Processing, IEEE Transactions on* 51(8), Aug 2003, S. 2191–2204.

[TiCF12] Juan Pablo Timpanaro, Isabelle Chrisment und Olivier Festor. A Bird's Eye View on the I2P Anonymous File-sharing Environment. In *Proceedings of the 6th International Conference on Network and System Security*, Wu Yi Shan, China, November 2012.

[VRBV03] Robbert Van Renesse, Kenneth P. Birman und Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Trans. Comput. Syst.* 21(2), Mai 2003, S. 164–206.

[YeCh02] Dit-Yan Yeung und C. Chow. Parzen-window network intrusion detectors. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, Band 4, 2002, S. 385–388 vol.4.

[ZhYG09] Weiyu Zhang, Qingbo Yang und Yushui Geng. A Survey of Anomaly Detection Methods in Networks. In *Computer Network and Multimedia Technology, 2009. CNMT 2009. International Symposium on*, Jan 2009, S. 1–3.