Machine Learning for Failure Detection in Distributed Systems

KONSTANTIN SOZINOV KIM HAMMAR
sozinov | kimham @kth.se

October 31, 2017

Abstract

Accurate failure detection can reduce operational costs and increase the reliability of distributed systems. However, detecting failures accurately is difficult. The difficulty can often be traced down to weak guarantees provided by the network and clock synchronization.

Accessibility of machine learning models and better computing resources have opened the research question whether machine learning is an effective approach for failure detection. We present a machine learning failure detector that uses an exchangeable machine learning model for predicting one-step ahead round-trip times. Our approach is novel in that we use a machine learning model for prediction, rather than estimating the probability distribution of round-trip times using formulas based on moving averages. The proposed detector uses incremental learning in combination with a dynamic safety margin based on the variance in recorded round-trip times. Our work opens up for experimenting with the proposed detector using different features and models depending on the characteristic of the contemplated environment for the failure detector.

We have quantified the behavior of the machine learning failure detector in simulated environments. Results show that it is suited for more aggressive failure detection than the eventual perfect failure detector and that it can adapt to different network conditions. Moreover, the decisive factors for accurate round-trip time predictions by the machine learning model were shown to be statistical patterns in the window of recorded round-trip times. Patterns in node features, such as geographic location and bandwidth, were shown to be of less importance.

Keywords— Machine learning, Distributed computing, Failure Detectors, Fault tolerant systems

Contents

List of Acronyms and Abbreviations

For clarity we summarize here the acronyms and abbreviations used in this document.

$\Diamond\mathscr{P}$	The eventual perfect failure detector [1].
λ_M	Average mistake rate of a failure detector [2].
MLFD	The machine learning failure detector.
MSE	Mean squared error.
NN	Neural Network.
QoS	Quality of Service.
RNN	Recurrent Neural Network.

RTT	Round-trip delay time. Originally this term is used to refer to round-trip delay time of IP-packages, in this report RTT typically will refer to the delay from when a failure detector sends a heartbeat request and the time a heartbeat response is received.
T_D	Average detection time, a metric to measure the time between a crash and the time it is detected by a failure detector [2].
PRNG	Pseudorandom Number Generator.
PCC	Pearson Correlation Coefficient.

1 Introduction

To decide if a remote computer is functioning, down, or temporary slow is an important task for distributed systems. Accurate decisions can reduce operational costs and increase reliability of algorithms. Often the guarantees provided by the network are weak and clocks of nodes in the system are not synchronized. These factors in combination with partial failures make failure detection challenging. Attributable to the weak guarantees, failure detection is impossible in completely asynchronous systems. This follows from the FLP-result by Fischer, Lynch & Paterson, 1985, that proves the impossibility of consensus in asynchronous systems [3].

Failure detection is provided as a service by the failure detector abstraction. Despite previous work on failure detection in distributed systems and research in machine learning, the integration of the two fields has not received much attention. We believe that access to modern machine learning models have opened the research question whether a failure detector based on machine learning is a viable approach.

The prior work that most resembles machine learning for failure detection in a distributed system is to use machine learning for round-trip time (RTT) prediction [4, 5, 6]. Also, machine learning has been used to identify reasons behind high RTT and data losses, to detect network anomalies, to detect denial of service attacks and to predict hardware failures [7, 8, 9, 10]. To the best of our knowledge, trained machine learning models have not yet been integrated into the failure detector abstraction.

In this report, we analyze the applicability of machine learning to the problem of failure detection. Our contributions include a theoretical definition as well as an implementation of a machine learning failure detector (MLFD) and evaluations of MLFD in simulated environments.

1.1 Problem Statement

Failure detection is a fundamental problem in distributed systems. Previous research on failure detectors has provided application developers with many choices. The contemplated environment of the failure detector has an impact on which failure detector is most suited. This project will probe the capabilities of using machine learning for failure detection and investigate in which environments it is appropriate.

1.2 Research Questions and Hypothesis

Building upon the described problem, this research will be concentrated around the following research questions:

Question 1. *How can the problem of failure detection be modeled as a learning problem?*

Question 2. *In which environments does the machine learning approach work well and what are its limitations?*

Question 3. *How does MLFD compare with the traditional eventual perfect failure detector ($\Diamond\mathscr{P}$)?*

Hypothesis 1. *We hypothesize that machine learning as a tool for failure prediction is useful when there exist patterns in the RTT based on features of nodes that are monitored. In such environments, the MLFD detector will outperform the traditional $\Diamond\mathscr{P}$ detector.*

2 Background

This section introduces the theoretical background. The section covers the failure detector abstraction, models of failure detection, metrics for evaluating failure detectors, and common machine learning models. Additionally, the section highlights relevant prior work on failure detection, machine learning, and RTT analysis.

2.1 Failure Detectors

In this section, we outline a summary of previous work on failure detectors, including a theoretical framework.

2.1.1 The Failure Detector Abstraction

As failure detection is such a fundamental problem for many applications, it is commonly provided as a service through the failure detector abstraction (figure 1). This results in a separation between applications and the logic of the detector, which is implementation specific. The application can at any point in time query the detector for the status of a process. As illustrated in figure 1, a failure detector monitors a set of processes by some means of network communication, most prevalent is the use of heartbeats.

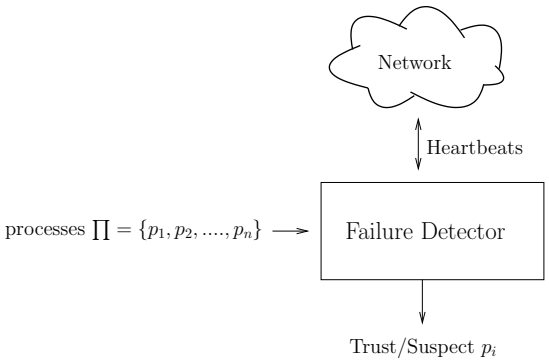


Figure 1: The Failure Detector Abstraction

2.1.2 Theory of Failure Detectors

We adopt the theoretical framework of unreliable failure detectors established by Chandra & Toueg, 1996 [1] that has been the basis for many theoretical results on failure detection [11, 12, 13, 2, 14]. Chandra & Toueg, 1996, propose to use abstract properties to define failure detectors. Abstract properties make it possible to reason about correctness and determine provable guarantees of failure detectors without having to rely on specific implementations [1]. Completeness is a requirement on the behavior of failure detectors regarding crashed processes. A complete failure detector should eventually detect a process that has crashed. Accuracy is a requirement on the behavior of failure detectors regarding alive processes. Accurate failure detectors will not suspect alive processes. In isolation, neither of these properties are very useful for characterizing failure detectors but together they can define the abstract behavior of failure detectors. Chandra & Toueg, 1996, define eight failure detectors based on variants of the accuracy and completeness properties [1]. $\Diamond\mathscr{P}$ is defined by being a complete and eventually accurate failure detector.

2.1.3 The $\Diamond\mathscr{P}$ Failure Detector

Algorithm 1 in appendix A implements a failure detector of class $\Diamond\mathscr{P}$. $\Diamond\mathscr{P}$ is designed for the partially synchronous system model where it has provable strong completeness and eventual strong accuracy. In a partially synchronous system, timing conditions will eventually hold. When the conditions will hold and for how long cannot be determined in advance. Hence, arbitrary timing assumptions can be violated before the conditions hold. Partial synchrony is a theoretical construction introduced by Lynch, Dwork & Stockmeyer, 1985 [15]. Despite being a theoretical construction many implementations assume a partially synchronous system [16, 17, 18].

2.1.4 Quality of Service Metrics

Completeness and accuracy properties are useful for contrasting failure detectors on a theoretical level, but are too vague for capturing the behavior of failure detectors in practice. Chen, Toueg & Aguilera,

2002, defines a set of quality of service (QoS) metrics for failure detection that will be used for comparing simulation results in this study [2]. The QoS metrics have been used extensively in related work [12, 16, 19, 20, 21, 22, 18, 23, 24, 25]. In particular, this report will focus on the QoS metrics average mistake rate (λ_M) and detection time (T_D). The rate of false suspicions made by the failure detector is measured by λ_M and T_D measures the elapsed time from the actual crash of a process and the time the process is suspected by the failure detector [2].

2.1.5 Adaptive Failure Detectors

The objective of a machine learning failure detector is to be adaptive to network conditions. Adaptive failure detectors are designed for environments where the conditions for failure detection are expected to change over time. Thus the idea of adaptive failure detectors is to adjust the time for failure detection based on prevailing network conditions, typically measured as a function of RTTs recorded in the past.

Notable work on adaptive failure detectors include (1) the $\phi - accrual$ failure detector (Hayashibara, Défago, Yared & Katayama, 2004 [12]); (2) the algorithm for failure detection presented by Chen et al., 2002 [2]; (3) the Low-Overhead accrual failure detector (LA-FD) (Ren, Dong, Liu, Li & Yang, 2012 [20]); (4) the Two Windows Failure Detector (2W-FD) (Tomsic, Sens, Garcia, Arantes & Sopena, 2015 [21]); (5) the weibull distribution failure detector (WD-FD) by Liu et al., 2017 that is explicitly designed for cloud computing environments [22]; (6) the exponential distribution failure detector (ED FD) (Xiong & Défago, 2007 [18]); (7) the self-tuning failure detector (SFD) in Xiong et al., 2012 [23]; (8) the Adaptare-FD that is based on the Adaptare framework (Dixit & Casimiro, 2010 [19]); and (9) in Bertier, Marin & Sens, 2002, an adaptive version of $\Diamond\mathscr{P}$ that uses a combination of the estimation algorithm of Chen et al., 2002 and Jacobsons's algorithm ([26]) is presented [16].

All of the adaptive failure detectors mentioned have the general idea in common. What characterize the detectors is the formula used for estimating the heartbeat arrival time and what assumptions are made on the probability distribution of RTTs. None of the listed detectors use a trained machine learning model for estimation. The most prevalent approach among prior work on adaptive failure detectors is to use a sliding window of RTTs and an equation estimating the arrival time based on moving averages.

2.1.6 Other Types of Failure Detectors

The adaptive failure detectors are the most relevant to the work in this project, other classes of failure detectors include (1) accrual failure detectors that outputs a degree of suspicion instead of a binary decision [17]; (2) lazy failure detectors that exploits messages already part of existing application protocols for failure detection and (3) gossip-style failure detectors where processes in a group cooperate by gossiping to detect failures [27]. The types of failure detectors are not mutually exclusive.

2.2 Machine Learning Approaches for RTT Prediction

This section discusses what patterns exist in RTT data and different machine learning models for predicting RTTs.

2.2.1 Existing Patterns for Failure Detection in Distributed Systems

A machine learning model performs best when it is trained with data that contains patterns. For failure detection in distributed systems, a salient pattern is a dependence of RTT on geographical distance. In Krajsa & Fojtova, 2011, it is shown that the dependence is linear [28]. Moreover, distributed systems can consist of heterogeneous computers where the knowledge of a computer's up-link and down-link speeds can be used to predict the RTT. Hu et al., 2015, use machine learning to diagnose factors behind high RTT and message losses. In traces from wireless cellular networks, they show that the channel quality is a strong factor behind RTT and message loss [7].

Diverse distributions have been observed when measuring RTTs in distributed systems of different nature. For this reason, different implementations assume different distributions. In Hayashibara et al.,

2004, the normal distribution is assumed [12]. In Liu et al., 2017, it is shown that in cloud computing environments, the weibull distribution is a better fit for RTTs than the normal distribution [22]. Adaptare-FD from Dixit & Casimiro, 2010, analyzes the RTT to find the best fit among a range of distributions, including the exponential, shifted exponential, weibull, pareto and uniform distributions [19]. Similarly, ED FD by Xiong & Défago, 2007, assumes the exponential distribution for estimating RTT timeouts [18].

In studies of end-to-end internet package delay, both Bolot, 1993 [29], and Mukherjee, 1992 [30], found that the probability distribution of end-to-end package delay was modeled closest by a shifted gamma distribution. This motivated Satzger, Pietzowski, Trumler & Ungerer, 2007, to use RTTs sampled from a gamma distribution when evaluating failure detectors in a simulated environment [24]. In results by Hooghiemstra & Van Mieghem, 2001, it is indicated that the end-to-end delay on the internet consists of two components, router processing delay and queuing delay caused by simultaneous traffic [31]. Similarly, Bogdanov, 2016, presents measurements across Amazon EC2 that too exhibits multiple elements of delay. In addition, Bogdanov contributes a novel technique for estimating network conditions that is capable of distinguishing between different cause of delay [32].

RTT estimation for failure detection differ from general network estimators in that the behavior of individual processes should be taken into account. For instance a process might routinely wait a number of seconds between receiving a heartbeat request and sending a reply, this delay is not owing to the network or the router processing delay but to the algorithm of the process.

2.2.2 Linear Regression

The output of the linear regression model is a linear function of the input. Linear regression can be defined mathematically as follows:

$$\hat{y} = W^T * X \tag{1}$$

where W is the weight matrix of parameters of the model and X is a matrix of input features to the model.

Karrer, 2007, used extensions of the linear model for predicting available throughput in a TCP connection [6]. By using extensions of the linear model, Karrer achieved an improvement of 40% for video streaming applications based on TCP.

Linear regression can be implemented as an online learning algorithm with minor modifications. Specifically, the Stochastic Gradient Descent Algorithm (SGD) can be used to implement linear regression for online learning.

2.2.3 Neural Networks

In a feed-forward multilayer Neural Network (NN) (often called multilayer perceptron) learning is done by matrix multiplications between the weight matrix W and the input matrix X. After performing this step, the output of the multiplication is passed through a non-linear function, such as *Sigmoid*, *Tahn* or *ReLU*. The algorithm for training NNs is iterative and minimizes the cost function.

Chong & Yoo, 2006, used a NN with one hidden layer for predicting one-step-ahead RTT [5]. The data for training consisted of RTTs and packet loss rates. The model demonstrated results for one-step-ahead prediction with a normalized error below 0.1 for 200 rounds of predictions.

Just as the standard linear regression, NNs for linear regression can be implemented using SGD.

2.2.4 Recurrent Neural Networks

Recurrent Neural Networks (RNN) differ from feed-forward NNs. RNNs maintains state from previous inputs that is used when updating the parameters of the model during training. For this reason, RNNs are able to take into account series of data when learning. As RTTs in distributed systems is a type of time series data, RNNs make a promising model for RTT prediction.

Using RNNs trained on real world data, Belgah & Tagina 2009, achieved low prediction error for forecasting RTT of time intervals [33]. Although RNNs show potential, they have the drawback that online

implementations are not widely available; as a consequence, we will not consider RNNs further in this study. We believe that this shortcoming is not intractable and as online implementations of RNNs become available we consider it to a be a promising model for failure detectors that use machine learning.

2.2.5 Other Machine Learning Models

Support Vector Regression (SVR) modeling was used in Mirza, Sommers, Barford & Zhu, 2010, for TCP throughput prediction [34]. With SVR they were able to predict TCP throughout with a margin of 10% most of the time. In consideration of this result, we see potential in using a SVR model for failure detection and leave it for future work to investigate.

Nunes et al., 2011, shows that a machine learning technique known as the Experts Framework can adapt quickly to changes in RTT [4]. They demonstrate that it can do better than the classical Jacobson's RTT estimate. In the context of failure detection, we consider the main limitation of their approach to be the heavyweight nature of the framework.

Hu, Wang & Sun, 2015, proposed a model that combines learning and modeling for predicting RTTs [35]. By using a decision tree for predicting RTTs over long distance links and a modeling approach for prediction over smaller distances, they achieved a high prediction accuracy. We have not considered this approach for our study due to the added complexity of using two models.

3 Method

This section outlines our strategy for answering the research questions in section 1.2.

3.1 Overview

The study was initiated with an exploration of the data available to failure detectors followed by an evaluation of machine learning models on the data. The outcome of the evaluation served as a basis for research question 1 and the design of MLFD, a failure detector based on machine learning. Evaluations of MLFD were based on Monte Carlo Simulations of a distributed system. The environments for simulation were designed in consideration of previous work on analyzing RTTs in distributed systems [28, 22, 18, 2, 29, 30, 32, 31]. To put evaluations in a context and answer research question 3, MLFD was compared with the traditional $\Diamond\mathscr{P}$.

3.2 System Model

We consider a partially synchronous system consisting of a finite set of processes $\prod = \{p_1, p_2, ..., p_n\}$. Processes are assumed to be in the crash-stop failure model extended with omission faults, defined in Cachin et al., 2011 [11]. Processes are non-byzantine and may crash at any time. Moreover, a crashed process does not recover, and restarted processes are considered by the system as new processes. Processes may exhibit omission faults. Finally, clocks of processes may drift at arbitrary rates, the only assumption is that each individual process clock is monotonically increasing and drifts at a constant rate.

3.3 Testbed

We did not find any existing testbed to match our requirements regarding machine learning models and configurable simulations. The environment for our custom testbed was Akka, an implementation of the actor model on the JVM [36]. As Akka runs on the JVM there exists a range of machine learning tools to leverage for prototyping. In particular, the implementation in this study leverages the support for machine learning in Apache Spark *. Actors in Akka can be viewed as lightweight processes and for the rest of the report, the terminology of processes is used.

The testbed used for simulation is open source and available online†. The testbed uses a driver program

* `https://spark.apache.org/` † `https://github.com/Limmen/mlfd_prototype`

that can be parameterized by options for the simulation. The simulations consisted of a supervisor process that monitored a set of worker processes using a failure detector. The worker and supervisor processes follow primitive message-passing algorithms, where distribution is simulated through added network delay and a simulated probability of crashes and omission faults. Pseudo-code for the worker and the supervisor can be found in algorithm 3 of appendix C and algorithm 4 of appendix D, respectively. To sample RTTs pseudo-randomly from different distributions, implementations of probability distributions and random generators from the apache.commons.math3 package [37] were used.

3.4 Experiment Setup

The metrics average mistake rate λ_M and the detection time T_D, defined in [2] was used when evaluating failure detectors. Our method to compute the detection time of failure detectors is equivalent to the method used in simulations of Satzger et al., 2007 [24]. When a process crashes, the time of the crash is measured exactly after successfully sending a heartbeat message to the detector, which implies that the detection time is measured in the worst case scenario.

All experiments were conducted through the testbed on a local machine with 8 cores Intel i7-4790 p 3.60GHz processor, 16GB RAM and 64-bit Ubuntu 16.04.2 LTS with Linux kernel 4.10.0-33. Scala version 2.11.8 and OpenJDK 64-Bit Server VM, Java 1.8.0_131 was used for the testbed.

Similar to simulations of Satzger et al., 2007 [24] and Chen et al., 2002 [2], the inter-sending of heartbeat messages, message loss probability, and crash probability were fixed for each simulation. Message delays were simulated according to different probability distributions. Test parameters that were fixed for each simulation are listed in table 1.

Table 1: Test parameters

Parameter	**Value**
Test duration	30 minutes
Number of processes to monitor	100
Sample window size	200
Default mean	3000 ms
Default standard deviation	1000 ms
Crash probability	0.001
Heartbeat interval	2s
Machine Learning parameters	regularization = 0.3, $\eta = 0.0000001$

The test duration was decided based on observations. The duration was chosen to be long enough for the machine learning model to stabilize and long enough for a substantial amount of crashes to be simulated. 30 minutes of simulation with given simulation parameters accounts for ≈ 60 crashed processes out of a total of 100 processes and approximately 17000 heartbeats collected by the failure detector. 17000 is smaller than the period of the PRNG and have shown to be enough samples for the machine learning model to stabilize. The sample window size is the number of RTTs that MLFD buffers for each monitored process. Default mean and standard deviation is the values used by MLFD when the sample window is of size zero. Heartbeat interval refers to the period between heartbeat requests. Crash probability is the probability for a process to crash after sending a heartbeat reply. Machine learning parameters are specific to the chosen linear regression model, η stands for the learning rate. Regularization parameter and learning rate were chosen based on trial and error.

3.4.1 Experiments for Evaluating Machine Learning Models

Datasets from two simulations were used for data exploration and to evaluate machine learning models. The datasets were divided into train set (80%) and test set (20%) used for training and evaluation, respectively.

Table 2 describes the simulations.

Table 2: Experiments used to evaluate machine learning models for predicting RTTs

Name	Description
Experiment 1	A simulation where there is a correlation between geographic location, bandwidth, and RTT. The correlation is linear. RTTs were sampled from a normal distribution ($RTT \sim \mathcal{N}_p(\mu, \sigma^2)$).
Experiment 2	A simulation where there is no correlation between geographic location, bandwidth and RTT. RTTs were sampled from a normal distribution ($RTT \sim \mathcal{N}_{np}(\mu, \sigma^2)$).

The notation of subscript p (pattern) is used for RTT distributions where there is a linear correlation between RTT and geographic location and between RTT and bandwidth capacity. Subscript np (no pattern) is used to denote that there is no correlation. Depending on the nature of the distributed system it might be realistic or not to assume that there is a correlation between properties such as geographic location and RTT. Hence the experiments in table 2 were identified as two cases which should highlight how effective machine learning is depending on available features and their accuracy.

It was anticipated that data from simulations would have linear correlations and Pearson correlation coefficient (PCC) was used for measuring correlations in the data. PCC has a value between $+1$ and -1. A PCC of 0 indicates that there is no correlation, a PCC of $+1$ and -1 indicates that there is positive or negative correlation, respectively. To evaluate models, the R^2 metric was used. This measure corresponds to the squared correlation between the ground truth and the predicted values. Values of R^2 are in the range from 0 to 1 where a value of 1 indicates that the model fits the data perfectly. R^2 was preferred as it is a normalized measure that enables comparison with other models.

3.4.2 Experiments for Evaluating MLFD

MLFD was evaluated and compared with $\Diamond\mathscr{P}$ in two independent experiments listed in table 3. The experiments were designed in consideration of research question 2 and 3.

Table 3: Experiments used to evaluate MLFD and $\Diamond\mathscr{P}$

Name	Description
Experiment 3	Experiment to compare how MLFD and $\Diamond\mathscr{P}$ behave in increasingly unreliable network conditions. Five executions were run with a probability of omission fault being $0.001, 0.01, 0.1, 0.5, 0.75$. RTTs were sampled from a normal distribution ($RTT \sim \mathcal{N}_p(\mu, \sigma^2)$).
Experiment 4	Experiment for comparing how MLFD and $\Diamond\mathscr{P}$ behave when RTTs are sampled from different distributions. Four simulations were run using the following distributions. Normal distribution with correlation ($RTT \sim \mathcal{N}_p(\mu, \sigma^2)$). Normal distribution without correlation ($RTT \sim \mathcal{N}_{np}(\mu, \sigma^2)$). Exponential distribution with correlation ($RTT \sim Exp_p(\mu)$). Weibull distribution with correlation and shape 1.5 ($RTT \sim Wei_p(\lambda, 1.5)$). The probability of omission fault was fixed for each simulation at 0.001.

To measure stabilization of the machine learning model in MLFD, mean squared error (MSE) was used. MSE was the metric available to the online learning model of our implementation. The lower MSE the better the prediction. As it would be disadvantageous for MLFD to leave out training data, the data for prediction was used for training afterward.

4 Machine Learning for Failure Detection

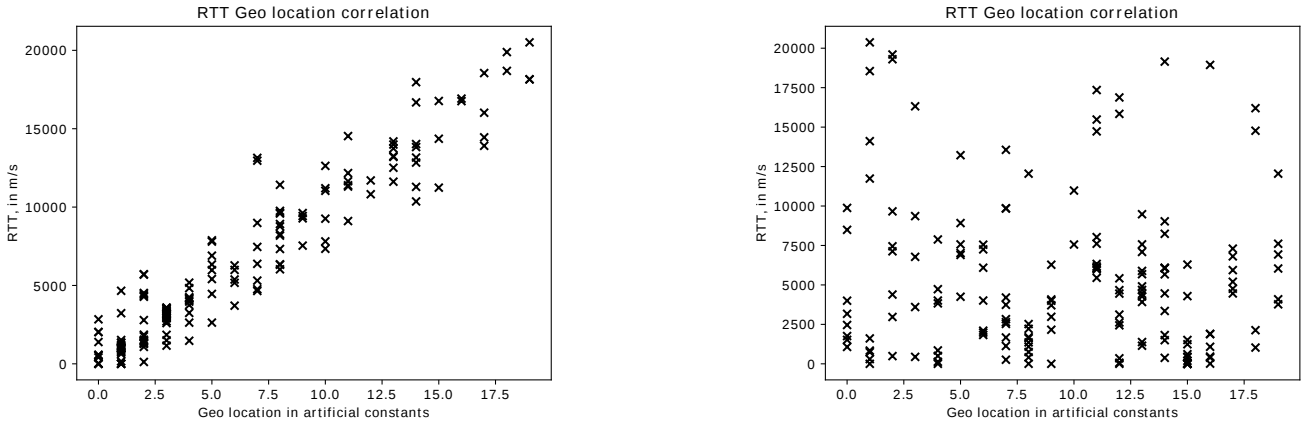
This section includes data exploration and an evaluation of machine learning models on the simulated data. Additionally, the section presents MLFD. We model failure detection as a learning problem where RTTs are predicted using incremental learning on data from recorded heartbeats.

4.1 Feature Exploration

Features for training the machine learning model depends on the environment where the failure detector operates. For experiments in this study, a feature vector v consists of the features listed in table 4.

Table 4: Feature vector for training machine learning models

Feature	Explanation
RTT_{μ}	Mean value of a window of the latest RTTs
RTT_{σ}	Standard deviation of a window of the latest RTTs
RTT_{min}	Minimum RTT of a window of the latest RTTs
RTT_{max}	Maximum RTT of a window of the latest RTTs
$Node_{loc}$	Geographic distance to the node in relation to the detector
$Node_{bw}$	Bandwidth capacity of the node



(a) Experiment 1

(b) Experiment 2

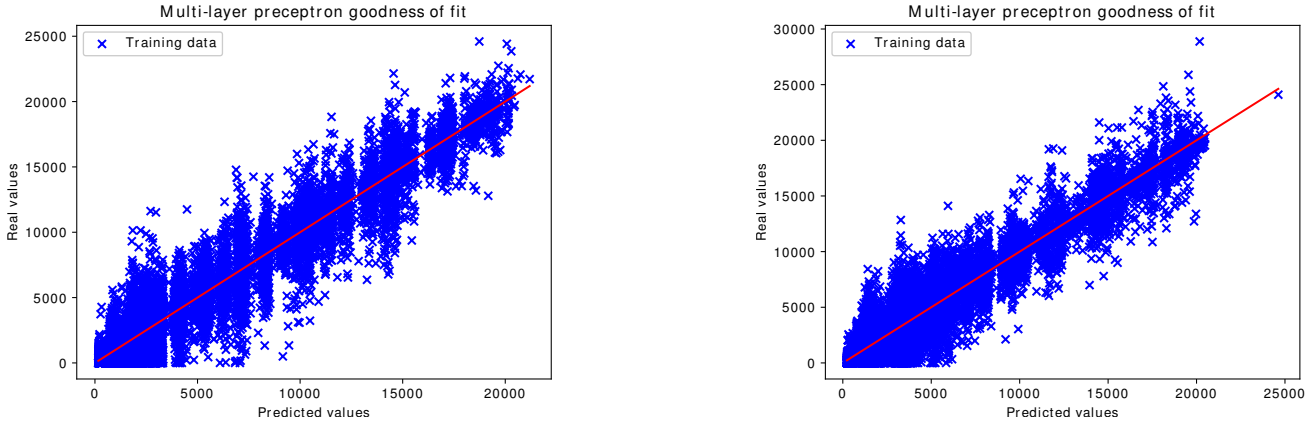
Figure 2: Correlation between RTT and geographical location for different experiment 1 and 2.

The effect size of features on the RTT have been computed with the Pearson correlation coefficient (PPC). Experiment 1 exhibited a near positive linear correlation between geographic location and RTT, giving a PCC of ≈ 0.96 (figure 2a); the same PCC in experiment 2 was ≈ 0.02 (figure 2b). The correlation between $Node_{bw}$ and RTT was weaker and outweighed by the correlation between RTT and $Node_{loc}$, yielding approximate PCCs of -0.26 and-0.09 for experiment 1 and 2, respectively. Furthermore, in both experiments the correlation between the features RTT_{μ}, RTT_{σ}, RTT_{max}, RTT_{min} and the RTT was prevalent, having approximate PCCs of $0.96, 0.25, 0.90, 0.90$ respectively.

4.2 Model Evaluation

A feed-forward NN with 4 hidden layers and 256 units in each layer was trained on the simulated data to predict RTTs. The NN used ReLU as its activation function. The network architecture was chosen based on

empirical testing and the hyper-parameters of the model were not tuned. We achieved a R^2 of $0.93 \pm 7.43 * 10^{-5} SE$ when training the model on data from experiment 1 (figure 3a) and a R^2 of $0.91 \pm 1.11 * 10^{-5} SE$ when training the model on data from experiment 2 (figure 3b). Thus, the goodness of fit for the two experiments was nearly the same.



(a) Experiment 1

(b) Experiment 2

Figure 3: Goodness of fit of multilayer perceptron regression

With linear regression, we achieved a R^2 score of $0.93 \pm 7.43 * 10^{-5} SE$ when training the model on data from experiment 1 and a R^2 of $0.91 \pm 1.59 * 10^{-5} SE$ when training the model on data from experiment 2. The results were near identical to the results achieved with the NN model.

The patterns between $Node_{loc}$ and RTT and between $Node_{bw}$ and RTT which is exhibited by experiment 1 had a modest impact on the RTT prediction. This indicates that the model was able to learn from statistical features of previous RTTs such as $RTT_{\mu}, RTT_{\sigma}, RTT_{max}, RTT_{min}$. Moreover, the results indicate no noteworthy differences between the NN model and the linear regression model.

4.3 MLFD

With the results derived above, we designed MLFD, an adaptive binary failure detector based on an online linear regression model. MLFD sends a liveness request to every process it is monitoring every heartbeat timeout δ. Monitored processes respond with a heartbeat message, indicating their liveness. When determining to detect or suspect a process, MLFD uses the concept of freshness points [16, 2, 21, 23]. A freshness point τ_p is an estimation of the time when the next heartbeat will be received from process p. In MLFD, τ_p is computed based on a prediction $pred_p$ from a machine learning model, and a safety margin α_p. MLFD stores heartbeat arrival times in a sample window S_p of fixed size for each process. S_p is updated dynamically. In addition, MLFD maintains a linear regression model M. MLFD uses S_p to dynamically compute the safety margin α_p for process p based on an estimate of the standard deviation σ_p, times a configurable constant k.

$$\mu_p = \frac{\sum_{s_i \in S_p} s_i}{|S_p|} \tag{2}$$

$$\sigma_p = \sqrt{\frac{1}{|S_p|} \sum_{s_i \in S_p} (s_i - \mu_p)} \tag{3}$$

$$\alpha_p = k \cdot \sigma_p \tag{4}$$

For each heartbeat reply from process p, MLFD adds the reply to S_p and updates M by computing a feature vector v. The feature vector is computed from the heartbeat reply and the current state of S_p. In this study, we assume the features listed in table 4. Statistical features of recorded RTTs are computed from the contents of S_p. The features $Node_{loc}$ and $Node_{bw}$ are assumed to be appended to heartbeat replies. The truth label lbl for vector v is the time difference from when the heartbeat request was sent (t_{sent}) and the reply was received (t_{rec}). The vector v together with the label lbl is used to update the model M.

$$v = [RTT_{\mu}, RTT_{\sigma}, RTT_{min}, RTT_{max}, Node_{loc}, Node_{bw}] \tag{5}$$

$$lbl = t_{rec} - t_{sent} \tag{6}$$

$$M = M.train(v, lbl) \tag{7}$$

For each heartbeat timeout δ, the model M is used to compute a prediction $pred_p$ for every process p. For all processes that have not yet replied to an outstanding heartbeat request, the current timeout C_t is computed by taking the time difference of the current time (t_{now}) and the time the request was sent (t_{sent}). The current timeout C_t, the prediction $pred_p$, and the safety margin α_p, constitutes as the freshness point for process p. The current timeout C_t and the freshness point τ_i are used by MLFD to make a binary decision d_p, either to trust or suspect process p.

$$pred_p = M.predict(p) \tag{8}$$

$$C_t = t_{now} - t_{sent} \tag{9}$$

$$\tau_p = pred_p + \alpha_p \tag{10}$$

$$d_p = \begin{cases} \langle suspect \rangle & \text{if } C_t > \tau_p \\ \langle trust \rangle & \text{if } C_t \leq \tau_p \end{cases} \tag{11}$$

An overview of the functioning of MLFD is shown in figure 4.

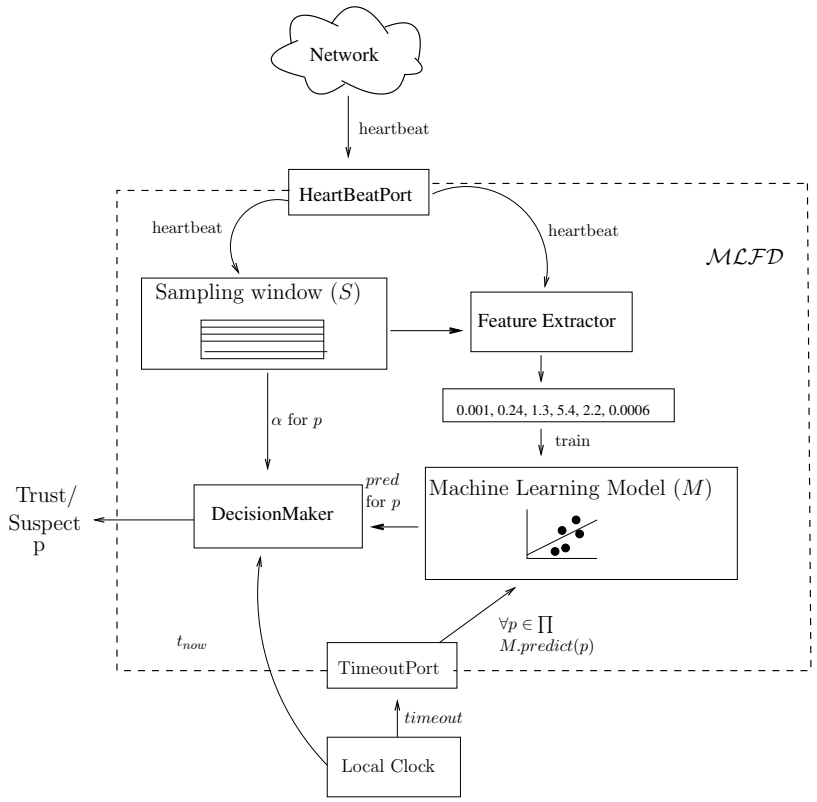


Figure 4: The functioning of MLFD, an adaptive binary failure detector that uses a sliding window of RTTs and a machine learning model that is updated incrementally to detect failures.

We have implemented *MLFD* using a streaming linear regression with stochastic gradient descent (SGD) model and the feature vector described in table 4 *. Pseudo-code for MLFD can be found in algorithm 2 of appendix B.

5 Results and Analysis

In this section, a theoretical analysis and evaluation of MLFD is presented. The partially synchronous system model is assumed.

5.1 Theoretical Analysis

Strong completeness of MLFD can be proved by relying on a weak assumption of the machine learning model M.

Assumption 1-$\mathscr{A}$. *Given training examples $\{x_1,...,x_n\}$ where the truth label $x_i(lbl)$ (i.e the RTT) for each example x_i is less than some real number y. Then $M.pred(p) \leq y, \quad \forall p$.*

Theorem 5.1. *Given assumption 1-$\mathscr{A}$, MLFD fulfills the strong completeness property. Eventually every crashed process p is permanently suspected by every correct process.*

Proof. Given that the model M is trained with new examples only when heartbeat replies are received by MLFD, all training examples will have labels that are less than some real number y. Eventually, the timeout since the last liveness-request to a crashed process p will exceed y. Given assumption 1-$\mathscr{A}$, this implies that p will be suspected by MLFD. As a suspicion for a process can only be revised in MLFD when a new heartbeat reply is received from the process, the suspicion is permanent under the assumption that crashed processes do not send heartbeats. □

For MLFD to guarantee eventual strong accuracy, like $\Diamond\mathscr{P}$ does, there must be an upper bound on how inaccurate a prediction for process p can be with respect to α_p. This assumption is not realistic for most machine learning models. In this sense, MLFD takes a probabilistic approach to the accuracy property.

5.2 MLFD Evaluation

This section presents an analysis of the configurable parameters of MLFD and $\Diamond\mathscr{P}$, as well as the results of experiments 3 and 4 described in table 3. The evaluation compares MLFD (algorithm 2) with $\Diamond\mathscr{P}$ (algorithm 1).

5.2.1 Trade-off Between Detection Time and Mistake Rate

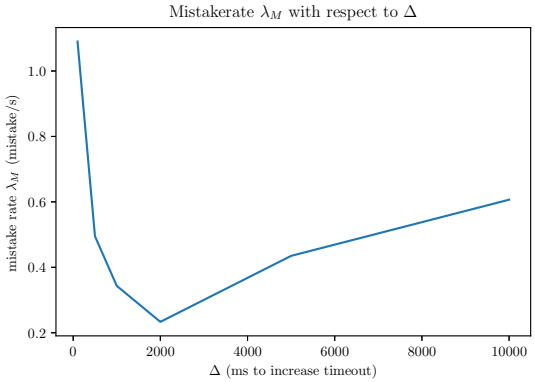
Both MLFD and $\Diamond\mathscr{P}$ have adjustable parameters to control the trade-off between the QoS metrics mistake rate (λ_M) and average detection time (T_D). For MLFD, the constant parameter k can be used to control the safety margin ($\alpha_p = k \cdot \sigma_p$). Likewise, $\Diamond\mathscr{P}$ allows tuning the parameter Δ which controls the increase of the detection timeout for each false suspicion.

* https://github.com/Limmen/mlfd_prototype

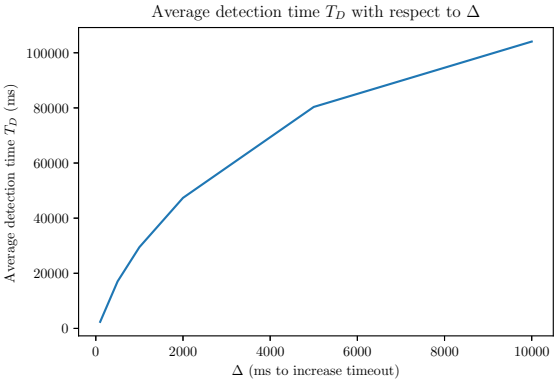
(a) Effect on λ_M when tuning k to control the safety margin α of MLFD.

(b) Effect on T_D when tuning k to control the safety margin α of MLFD.

Figure 5: Plots depicting the effect of tuning parameter k of MLFD. $RTT \sim \mathcal{N}_p(\mu, \sigma^2)$.



(a) Effect on λ_M when tuning Δ to control how much the detection timeout is increased for a false suspicion of $\Diamond\mathscr{P}$.



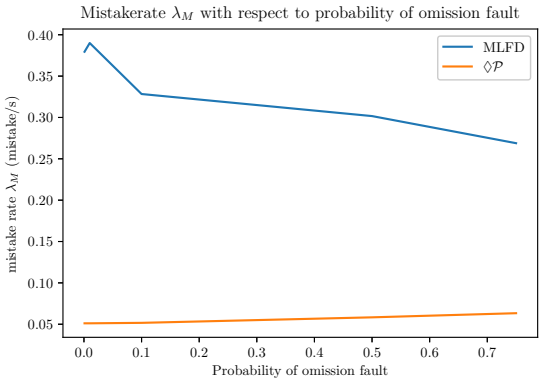
(b) Effect on λ_M when tuning Δ to control how much detection timeout is increased for a false suspicion of $\Diamond\mathscr{P}$.

Figure 6: Plots depicting the effect of tuning parameter Δ of $\Diamond\mathscr{P}$. $RTT \sim \mathcal{N}_p(\mu, \sigma^2)$.

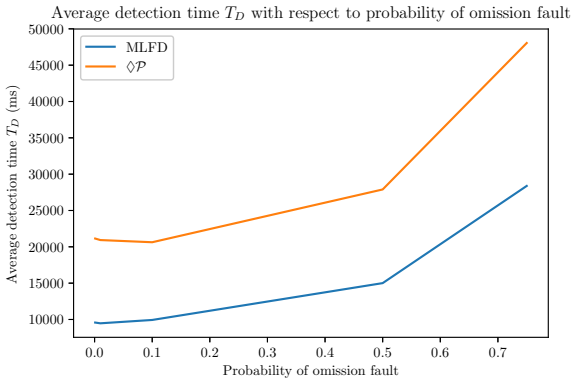
The higher k the higher T_D and the lower λ_M of MLFD (figure 5). Similarly, the higher Δ the higher T_D and the lower λ_M of $\Diamond\mathscr{P}$ (figure 6), with an exception that if Δ is too large to begin with, it takes longer for $\Diamond\mathscr{P}$ to find the right threshold for detection which causes it to make more mistakes initially. We can see in figure 6 that this behavior happens when $\Delta > 2000$. The effect of tuning k and Δ is independent of RTT distribution. To pick analogous trade-offs between λ_M and T_D for MLFD and $\Diamond\mathscr{P}$, the parameters $k = 3$ and $\Delta = 2000$ were chosen for the rest of the simulations.

5.2.2 Detection Performance in Unreliable Network Conditions

This section outlines the result of experiment 3 from table 3. The experiment tested how the detectors handle RTTs that are highly irregular. The results show that the higher the probability of omission fault, the higher value of T_D for both detectors (figure 7). The increase in T_D for both detectors show a similar growth, where T_D for MLFD was an order of magnitude smaller in general. As the probability of omission faults increased and T_D increased with it, λ_M of MLFD decreased. A steep decrease in λ_M occurred when a non-zero probability of omission fault was introduced and then it decreased linearly as the probability increased. Moreover, the steepest growth in T_D occurred when increasing the probability of omission fault from 0.5 to 0.75. Finally, it can be seen that MLFD had a larger mistake rate than $\Diamond\mathscr{P}$.



(a) Difference in λ_M when increasing number of omission faults.

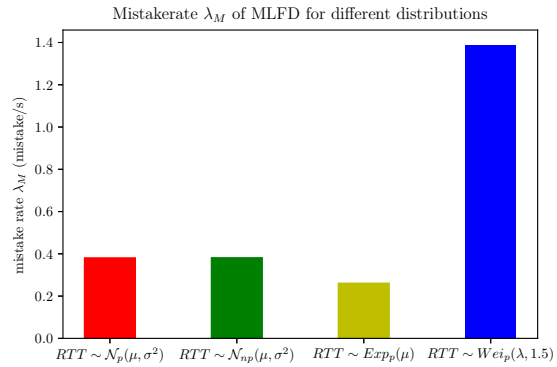


(b) Difference in T_D when increasing number of omission faults.

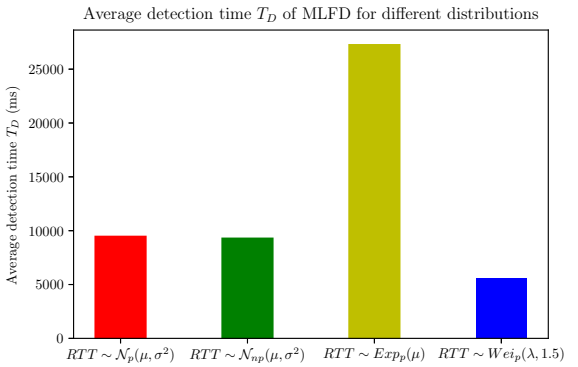
Figure 7: Comparison between QoS metrics of MLFD and $\Diamond\mathscr{P}$ in presence of omission faults.

5.2.3 Detection Performance for Different RTT Probability Distributions

This section outlines the result of experiment 4 from table 3. The experiment evaluated how MLFD and $\Diamond\mathscr{P}$ adapts to different RTT distributions.



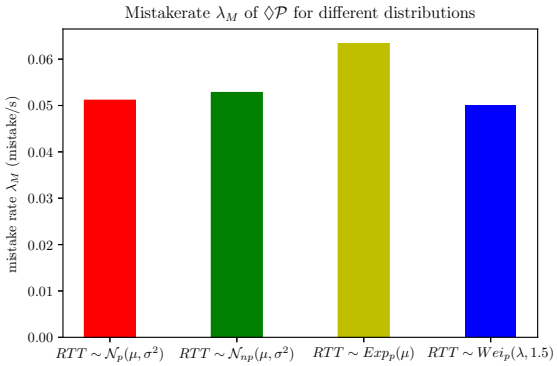
(a) Difference in λ_M when RTTs are sampled from different distributions.



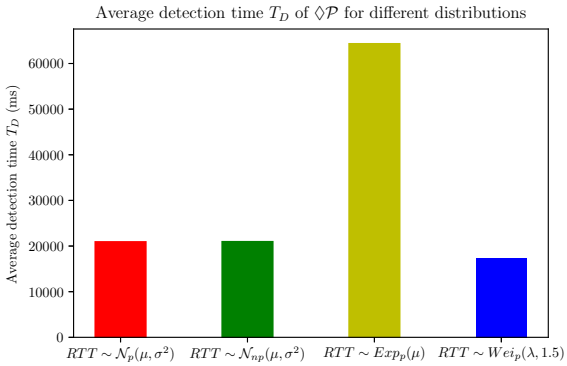
(b) Difference in T_D when RTTs are sampled from different distributions.

Figure 8: Comparison how MLFD adapts when RTTs are sampled from different probability distributions.

MLFD performed near equivalently regarding both λ_M and T_D for the normal distribution with and without correlation between node features and RTT (figure 8). For RTTs from the weibull distribution, the T_D was the lowest and the λ_M was the highest among all of the distributions. The exponential distribution caused the lowest λ_M, but at the cost of a higher T_D. Overall the results show that the higher T_D the lower λ_M. The exponential distribution had more outliers in the RTTs compared with the other distributions which caused $\Diamond\mathscr{P}$ to increase its timeout value significantly, increasing both λ_M and T_D (figure 9).



(a) Difference in λ_M when RTTs are sampled from different distributions.



(b) Difference in T_D when RTTs are sampled from different distributions.

Figure 9: Comparison how $\Diamond\mathscr{P}$ adapts when RTTs are sampled from different probability distributions.

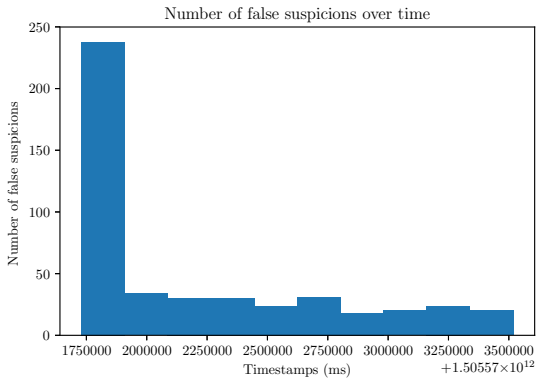
6 Discussion

There is a mismatch between the traditional computer science approach to focus on correctness and deterministic computations and the approach used in machine learning. In machine learning, a model that give accurate output 99% of the time is typically considered good. This makes it harder to reason about provable properties when machine learning is involved. As a consequence, MLFD only guarantees probabilistic eventual accuracy. Whereas $\Diamond\mathscr{P}$ have provable eventual accuracy. MLFD can be modified to guarantee eventual accuracy at the cost of higher detection times by making the safety margin α_p monotonically increasing for each false suspicion.

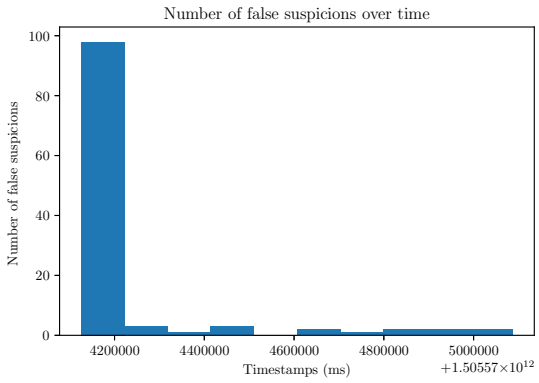
When compared with $\Diamond\mathscr{P}$, MLFD demonstrates consistently lower detection times but is also more prone to false suspicions. This is because $\Diamond\mathscr{P}$ monotonically increases its timeout until it does not do any premature suspicions while MLFD adapts its timeout for suspicion based on recorded RTTs. This has the fallout that MLFD uses a lower detection timeout for processes with consistently low RTTs and higher timeouts for processes with consistently higher RTTs, causing irregular RTTs to be detected quicker. On the contrary, $\Diamond\mathscr{P}$ uses a single timeout for all processes and is therefore not as responsive for irregular RTTs as MLFD.

Our hypothesis that MLFD would excel when there exist patterns in the RTTs based on features of the nodes was partially rejected. We found the reason that such correlations did not improve the prediction results to be that there existed strong correlations between RTT and statistical features such as $RTT_\mu, RTT_\sigma, RTT_{min}$, and RTT_{max}, which the model based its predictions on.

Both MLFD and $\Diamond\mathscr{P}$ made most of their inaccurate suspicions early in the simulations, before they had adapted their timeouts. This is visualized in figure 10 where the number of false suspicions is plotted with respect to time. The plot is from experiment 4 and the simulation where RTTs were sampled from the $Exp_p(\mu)$ distribution.



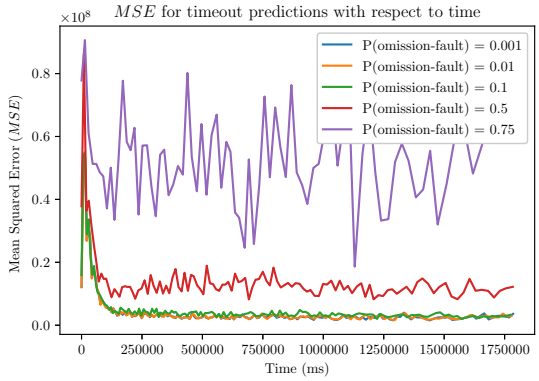
(a) Number of false suspicions made by MLFD with respect to time.



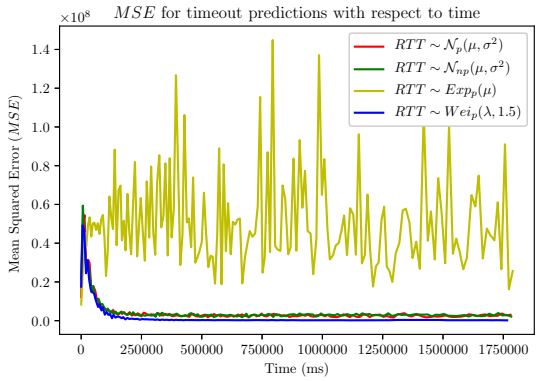
(b) Number of false suspicions made by $\Diamond\mathscr{P}$ with respect to time.

Figure 10: Number of false suspicions with respect to time. RTTs were sampled from the $Exp_p(\mu)$ distribution.

The machine learning model of MLFD stabilized early in the execution for most test cases (figure 11). For the simulation with a high probability of omission faults and for the simulation where RTTs were sampled from the exponential distribution, the prediction error was the highest. This indicates that the prediction performance is sensitive to wide RTT distributions that contain outliers.



(a) MSE with respect to time of the Linear regression model used in MLFD for different probabilities of omission fault.



(b) MSE with respect to time of the Linear regression model used in MLFD when RTTs are sampled from different probability distributions.

Figure 11: Stabilization of Mean squared error (MSE) over time for MLFD.

The safety margin α_p that MLFD computes dynamically for each process is supposed to cover up for inaccurate predictions when the RTTs are irregular. The results of experiment 4 demonstrates that this was effective. The λ_M for the simulation with RTTs sampled from the exponential distribution were the lowest despite having poor prediction performance, this was at the cost of a higher T_D. The same phenomenon is shown in the results of experiment 3, where λ_D of MLFD decreased when the probability of omission fault increased. Although MLFD adapts to RTT distributions, it showed alarmingly high mistake rate when the detection time decreased, as is shown in figure 8 and the result when RTTs were sampled from a weibull distribution. The low detection time and the poor mistake rate were due to a low variance in the RTT distribution, causing the safety margin to be unreasonably small. We believe that this scenario can be improved by introducing a lower bound on the safety margin.

We observed that the higher the maximum RTT of alive processes were, the more mistakes $\Diamond\mathscr{P}$ made and the longer it took for it to detect real crashes, which was expected. $\Diamond\mathscr{P}$ is only dependent on how large the highest RTTs are, which is why it performed the worst when RTTs were sampled from the exponential distribution, which had the most outliers.

6.1 Related Work

To the best of our knowledge, there are no prior attempts to use machine learning models inside a failure detector, which would have made for the most interesting comparisons. Prior results of failure detectors are not directly comparable to the results presented here as the environments and assumptions differ. It is left for future work to implement and evaluate the detectors in fair conditions.

In evaluations by Hayashibara et al., 2004, it is shown that the effect of increasing the ϕ parameter of the $\phi - accrual$ failure detector is similar to the effect of tuning the k parameter of MLFD [12]. Increasing ϕ gave a near linear decrease in λ_M and an exponential increase in T_D, which can be compared with increasing k of MLFD that resulted in a near linear decrease of λ_M and a linear increase in T_D. In our simulations the window size was fixed to 200, in Hayashibara et al., 2004 it is shown that increasing the window size up to 10000 can reduce the mistake rate of the $\phi - accrual$ failure detector drastically. Hence we anticipate that MLFD might see improved accuracy by increasing the window size as well. Finally, the trade-off between λ_M and T_D that we observed is in concordance with results of prior work such as [21, 24, 18, 23].

6.2 Conclusion

We have shown that it is possible to integrate machine learning into a failure detector with strong completeness and eventual probabilistic accuracy. The proposed detector, MLFD, uses an exchangeable machine learning model and opens up for experiments with different models on empirical data. The key idea to use machine learning for failure detection is to train the model on streaming data to predict RTTs based on recorded RTTs and node features. MLFD is able to adapt to different RTT distributions by using incremental learning of a machine learning model, in combination with a dynamic safety margin to compensate for inaccurate predictions. Evaluating MLFD in simulated environments indicate that it is suited for more aggressive failure detection than the traditional $\Diamond\mathscr{P}$. Finally, the decisive element for accurate RTT predictions by the machine learning model were shown to be statistical patterns in the window of recorded RTTs.

6.3 Future Work

A natural extension of our work is to evaluate MLFD further. Interesting evaluations to be done include to test MLFD in real-world scenarios, which have been left out in this study due to a restricted budget; evaluations to compare MLFD against other failure detectors, to evaluate MLFD with different window sizes and to evaluate MLFD with other probability distributions, all of which were left out in this study due to time limitations. Other measurements that were left out is to evaluate the overhead of using machine learning for failure detection, and evaluation of how quick MLFD can adapt to changing network conditions.

Our evaluation of machine learning models focused on linear regression and neural networks, future work may involve further evaluation of machine learning models. Finally, the MLFD algorithm presented in this study can inspire for new failure detectors based on machine learning. Such as making MLFD accrual, experimenting with pre-training, using different features for training and to evaluate the effectiveness of using individual machine learning models for each monitored process.

References

[1] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=226647

[2] W. Chen, S. Toueg, and M. K. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 561–580, May 2002. doi: 10.1109/TC.2002.1004595

[3] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985. doi: 10.1145/3149.214121. [Online]. Available: http://doi.acm.org/10.1145/3149.214121

[4] B. A. A. Nunes, K. Veenstra, W. Ballenthin, S. Lukin, and K. Obraczka, "A Machine Learning Approach to End-to-End RTT Estimation and its Application to TCP," in *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, Jul. 2011. doi: 10.1109/ICCCN.2011.6006098 pp. 1–6.

[5] K. T. Chong and S. G. Yoo, "Neural network prediction model for a real-time data transmission," *Neural Computing & Applications*, vol. 15, no. 3-4, pp. 373–382, Jun. 2006. doi: 10.1007/s00521-006-0042-1. [Online]. Available: https://link.springer.com/article/10.1007/s00521-006-0042-1

[6] R. P. Karrer, "TCP Prediction for Adaptive Applications," in *32nd IEEE Conference on Local Computer Networks (LCN 2007)*, Oct. 2007. doi: 10.1109/LCN.2007.145 pp. 989–996.

[7] Z. Hu, Y.-C. Chen, L. Qiu, G. Xue, H. Zhu, N. Zhang, C. He, L. Pan, and C. He, "An In-depth Analysis of 3g Traffic and Performance," in *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, ser. AllThingsCellular '15. New York, NY, USA: ACM, 2015. doi: 10.1145/2785971.2785981. ISBN 978-1-4503-3538-6 pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2785971.2785981

[8] T. Ahmed, B. Oreshkin, and M. Coates, "Machine Learning Approaches to Network Anomaly Detection," in *Proceedings of the 2Nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, ser. SYSML'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 7:1–7:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1361442.1361449

[9] S. Seufert and D. O'Brien, "Machine Learning for Automatic Defence Against Distributed Denial of Service Attacks," in *2007 IEEE International Conference on Communications*, Jun. 2007. doi: 10.1109/ICC.2007.206 pp. 1217–1222.

[10] J. Murray, G. Hughes, and K. Kreutz-Delgado, "Machine Learning Methods for Predicting Failures in Hard Drives: A Multiple-Instance Application." *Journal of Machine Learning Research*, vol. 6, pp. 783–816, May 2005.

[11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer Publishing Company, Incorporated, 2011. ISBN 978-3-642-15259-7

[12] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The phi; accrual failure detector," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, Oct. 2004. doi: 10.1109/RELDIS.2004.1353004 pp. 66–78.

[13] N. Hayashibara, A. Cherif, and T. Katayama, "Failure detectors for large-scale distributed systems," in *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*. IEEE, 2002, pp. 404–409. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/1180218/

[14] S. Kumar and J. Welch, "Implementing $\Diamond P$ with Bounded Messages on a Network of ADD Channels," *arXiv:1708.02906 [cs]*, Aug. 2017, arXiv: 1708.02906. [Online]. Available: http://arxiv.org/abs/1708.02906

[15] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988. doi: 10.1145/42282.42283. [Online]. Available: http://doi.acm.org/10.1145/42282.42283

[16] M. Bertier, O. Marin, and P. Sens, "Implementation and Performance Evaluation of an Adaptable Failure Detector," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02. Washington, DC, USA: IEEE Computer Society, 2002. ISBN 978-0-7695-1597-7 pp. 354–363. [Online]. Available: http://dl.acm.org/citation.cfm?id=647883.738261

[17] X. Défago, P. Urbán, N. Hayashibara, and T. Katayama, "Definition and specification of accrual failure detectors," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005, pp. 206–215. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/1467795/

[18] N. Xiong and X. Defago, "ED FD: Improving the Phi accrual failure detector," 2007. [Online]. Available: https://dspace.jaist.ac.jp/dspace/handle/10119/4799

[19] M. Dixit and A. Casimiro, "Adaptare-FD: A dependability-oriented adaptive failure detector," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*. IEEE, 2010, pp. 141–147. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/5623386/

[20] X. Ren, J. Dong, H. Liu, Y. Li, and X. Yang, "Low-Overhead Accrual Failure Detector," *Sensors (Basel, Switzerland)*, vol. 12, no. 5, pp. 5815–5823, May 2012. doi: 10.3390/s120505815. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3386713/

[21] A. Z. Tomsic, P. Sens, J. Coelho Garcia, L. Arantes, and J. Sopena, "2w-FD: A Failure Detector Algorithm with QoS," in *IPDPS 2015 - The 29th IEEE International Parallel and Distributed Processing Symposium*. Hyderabad, India: IEEE, May 2015. doi: 10.1109/IPDPS.2015.74 pp. 885–893. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01357777

[22] J. Liu, Z. Wu, J. Wu, J. Dong, Y. Zhao, and D. Wen, "A Weibull distribution accrual failure detector for cloud computing," *PLOS ONE*, vol. 12, no. 3, p. e0173666, Mar. 2017. doi: 10.1371/journal.pone.0173666. [Online]. Available: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0173666

[23] N. Xiong, A. V. Vasilakos, J. Wu, Y. R. Yang, A. Rindos, Y. Zhou, W.-Z. Song, and Y. Pan, "A Self-tuning Failure Detection Scheme for Cloud Computing Service," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012. doi: 10.1109/IPDPS.2012.126. ISBN 978-0-7695-4675-9 pp. 668–679. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.126

[24] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, "Variations and Evaluations of an Adaptive Accrual Failure Detector to Enable Self-healing Properties in Distributed Systems," in *Proceedings of the 20th International Conference on Architecture of Computing Systems*, ser. ARCS'07. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN 978-3-540-71267-1 pp. 171–184. [Online]. Available: http://dl.acm.org/citation.cfm?id=1763274.1763287

[25] ——, "A New Adaptive Accrual Failure Detector for Dependable Distributed Systems," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, ser. SAC '07. New York, NY, USA: ACM, 2007. doi: 10.1145/1244002.1244129. ISBN 978-1-59593-480-2 pp. 551–555. [Online]. Available: http://doi.acm.org/10.1145/1244002.1244129

[26] V. Paxson and M. Allman, "RFC 2988: Computing TCP's Retransmission Timer," http://www.rfc-editor.org/rfc/rfc2988.txt, online; Accessed 2017-09-17.

[27] I. Gupta, T. D. Chandra, and G. S. Goldszmidt, "On Scalable and Efficient Distributed Failure Detectors," in *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '01. New York, NY, USA: ACM, 2001. doi: 10.1145/383962.384010. ISBN 978-1-58113-383-7 pp. 170–179. [Online]. Available: http://doi.acm.org/10.1145/383962.384010

[28] O. Krajsa and L. Fojtova, "RTT measurement and its dependence on the real geographical distance," in *2011 34th International Conference on Telecommunications and Signal Processing (TSP)*, Aug. 2011. doi: 10.1109/TSP.2011.6043737 pp. 231–234.

[29] J.-C. Bolot, "End-to-end Packet Delay and Loss Behavior in the Internet," in *Conference Proceedings on Communications Architectures, Protocols and Applications*, ser. SIGCOMM '93. New York, NY, USA: ACM, 1993. doi: 10.1145/166237.166265. ISBN 978-0-89791-619-6 pp. 289–298. [Online]. Available: http://doi.acm.org/10.1145/166237.166265

[30] A. Mukherjee, "On the Dynamics and Significance of Low Frequency Components of Internet Load," *Internetworking: Research and Experience*, vol. 5, pp. 163–205, 1992.

[31] G. Hooghiemstra and P. Van Mieghem, "Delay distributions on fixed internet paths," 2001.

[32] K. Bogdanov, "Reducing long tail latencies in geo-distributed systems," p. 161, 2016, qC 20161101. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2:1042533

[33] S. Belhaj and M. Tagina, "Modeling and prediction of the internet end-to-end delay using recurrent neural networks," vol. 4, 08 2009.

[34] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A Machine Learning Approach to TCP Throughput Prediction," *IEEE/ACM Transactions on Networking*, vol. 18, no. 4, pp. 1026–1039, Aug. 2010. doi: 10.1109/TNET.2009.2037812

[35] W. Hu, Z. Wang, and L. Sun, "Guyot: a hybrid learning- and model-based RTT predictive approach," in *2015 IEEE International Conference on Communications (ICC)*, Jun. 2015. doi: 10.1109/ICC.2015.7249260 pp. 5884–5889.

[36] L. Inc., "akka," http://akka.io/, online; Accessed: 2017-09-12.

[37] T. A. S. Foundation, "Commons math: The apache commons mathematics library," http://commons.apache.org/proper/commons-math/, online; Accessed: 2017-09-18.

A Increasing Timeout Algorithm

Algorithm 1 Increasing Timeout algorithm for implementing the $\Diamond\mathscr{P}$ abstraction. Adapted from pseudo-code in [11].

Require:

$\prod$ ▷ Set of processes to monitor

Δ ▷ Value to increase timeout with

procedure INCREASING TIMEOUT($\prod$,Δ)

$timeout \leftarrow \Delta$

$alive \leftarrow \prod$

$suspected \leftarrow \emptyset$

$starttimer(timeout)$

upon event $timeout$ **do**

if $alive \cap suspected \neq \emptyset$ **then**

$timeout \leftarrow timeout + \Delta$

end if

for $p \in \prod$ **do**

if $(p \notin alive) \wedge (p \notin suspected)$ **then**

$suspected \leftarrow suspected \cup \{p\}$

output $\langle Suspect|p\rangle$

else if $(p \in alive) \wedge (p \in suspected)$ **then** ▷ Mistakenly suspected process p

$suspected \leftarrow suspected \setminus \{p\}$

output $\langle Restore|p\rangle$

end if

send $\langle HeartBeatRequest|p\rangle$

end for

$alive \leftarrow \emptyset$

$starttimer(timeout)$

upon event $received\langle HeartBeatRequest|p\rangle$ **do**

send $\langle HeartBeatReply|p\rangle$

upon event $received\langle HeartBeatReply|p\rangle$ **do**

$alive \leftarrow alive \cup \{p\}$

end procedure

B MLFD Algorithm

Algorithm 2 MLFD algorithm

Require:

$\prod$ ▷ Set of processes to monitor
$windowSize$ ▷ Size of sample window
δ ▷ Heartbeat period
k ▷ Constant to compute safety margin α_p

procedure MLFDALGORITHM()
$S \leftarrow \emptyset$
$alive \leftarrow \prod$
$suspected \leftarrow \emptyset$
$starttimer(timeout)$
$outstandingHBs \leftarrow \emptyset$
$M \leftarrow newStreamingLinearRegressionModel()$

upon event $timeout$ **do**
$t_{now} \leftarrow getCurrentTimestamp()$
for $p \in \prod$ **do**
if $(p \notin alive) \wedge (p \notin suspected)$ **then**
$t_{sent} \leftarrow outstandingHbs(p)$
$(\mu, \sigma, min, max) \leftarrow getStatisticalFeatures(S)$
$pred_p \leftarrow M.predict(\mu, \sigma, min, max, p.location, p.bandwidth)$
$\alpha_p \leftarrow \sigma \cdot k$
$C_t \leftarrow t_{now} - t_{sent}$
if $pred_p + \alpha_p < C_t$ **then**
$suspected \leftarrow suspected \cup p$
end if
else if $(p \in alive) \wedge (p \in suspected)$ **then** ▷ Mistakenly suspected process p
$suspected \leftarrow suspected \setminus \{p\}$
end if
if $p \in alive$ **then**
$outstandingHBs \leftarrow outstandingHBs.put(p \rightarrow t_{now})$
end if
send $\langle HeartBeatRequest|p \rangle$
end for

upon event $received\langle HeartBeatRequest|p \rangle$ **do**
send $\langle HeartBeatReply|p \rangle$

upon event $received\langle HeartBeatReply|p \rangle$ **do**
$t_{now} \leftarrow getCurrentTimestamp()$
$S \leftarrow S \cup (p \rightarrow (HeartBeatRebly, t_{now}))$
$(\mu, \sigma, min, max) \leftarrow getStatisticalFeatures(S)$
$lbl \leftarrow t_{now} - outstandingHBs(p)$
$M \leftarrow M.train((\mu, \sigma, min, max), lbl)$
$alive \leftarrow alive \cup \{p\}$
end procedure

C Worker Process of the Testbed

Algorithm 3 Algorithm of a worker process in the testbed

Require:
D ▷ Probability Distribution of RTTs
k ▷ Constant added delay based on geographic location and network topology
$crash$ ▷ Probability of crash
$omission$ ▷ Probability of omission faults
procedure WORKERALGORITHM(D,k,$crash$,$omission$)

upon event $received\langle HeartBeatRequest|p\rangle$ **do**
if $\neg messageLoss(omission)$ **then**
$delay \leftarrow D.sample() + k$
$starttimer(delay, p)$
end if

upon event $\langle timeout, p\rangle$ **do**
send $\langle HeartBeatReply|p\rangle$
$simulateCrash(crash)$

end procedure

D Supervisor Process of the Testbed

Algorithm 4 Algorithm of a supervisor process in the testbed

Require:
fd ▷ Failure Detector
$timeout$ ▷ Heartbeat period
$\prod$ ▷ Set of processes to monitor
procedure SUPERVISORALGORITHM()(fd,$\prod$,$timeout$)
$starttimer(timeout)$

upon event $\langle timeout, p\rangle$ **do**
$fd.timeout()$
for $p \in \prod$ **do**
send $\langle HeartBeatRequest|p\rangle$
end for

upon event $received\langle HeartBeatReply|p\rangle$ **do**
$fd.receivedReply(p)$
end procedure