Computer Science
&
Data Science

Capstone Report - Spring 2022

A Benchmark Platform for Failure Detectors

Ruhao Xin,
Juncheng Dong

supervised by
Olivier Gilles Marin

Preface

If you are researching failure detectors but have difficulties comparing the evaluations of your failure detector with others, or are tired of building a complete network environment to test your failure detectors, then this paper is exactly for you.

Acknowledgements

We acknowledge our mentor and supervisor professor Olivier Marin for giving us a lot of useful suggestions. Moreover, he also provides us with his experiment results so that we can build a virtual network environment based on his data.

Abstract

We study benchmark platforms for failure detectors. By Failure Detectors, we mean algorithms that can suspect whether a process crashes or not. By benchmark, we mean some methods that can evaluate the performance of failure detectors based on some metrics. What we did here is: 1. Determine some metrics that evaluate the quality of service of failure detectors, like their accuracy rate, speed and so on 2. Design a language system of failure detector algorithms, which helps users conveniently translate their failure detectors into executable codes 3. Build a benchmark system, which gives scores to each metric based on built-in standards of 'good performance' and 'poor performance' gathering from our experiments, and then calculates a total score based on weights of each metric 4. Visualization of the scores, which contains bar chart and line chart.

Keywords

Failure Detectors; Benchmark; Distributed Systems

Contents

1 Introduction

One of the most typical and important problems in distributed systems is failure detection. This seems simple, but according to [1], in an asynchronous distributed system, it is impossible to deterministically judge the state of a process. A process which seemingly crashed may just be very slow. As a result, Chandra and Toueg introduce a model called *unreliable failure detectors* [2], proved that consensus can be solved even with unreliable failure detectors that make an infinite number of mistakes. Moreover, they define two properties to evaluate the performance of a failure detector: completeness and accuracy. Based on this, more and more papers have built failure detectors. For example, Bertier, Marin and Sens proposed an implementation of failure detector which is adaptable and can support scalable applications [3]. Later on, they integrated this with a hierarchical network structure to present a *Hierarchical failure detector* [4], which allows to decrease the number of messages and the processor load. Based on this, [5] improved the initial models of failure detectors by providing a probabilistic evaluation about suspicion rather than a deterministic evaluation.

However, there are some problems about designing failure detectors. Firstly, we notice that previous papers need to build practical networks to evaluate their failure detectors' performance. For instance, [3] did the evaluation on a non dedicated cluster of six PCs. [4] used a platform called *DUMMYNET* [6] to simulate some virtual network environments. [5]'s experiment involves communications between two computers located on different continents. We think that it is unnecessary for researchers to build a complete and complex network structure for evaluations. With regard to failure detectors, the behaviour of processes is much more important than the entire network flow. Secondly, the current comparison processes between different failure detectors are not very efficient because different failure detectors are designed under different network environments. As a result, it can be more convenient for researchers to build, test and compare failure detectors on a specialized platform. That's the reason why we aim to design and implement a benchmark platform for failure detectors.

2 Related Work

This section first talks about different implementations of failure detectors, and then talks about benchmark designs.

2.1 Failure Detectors

When we talk about failure detector, it is impossible to neglect the foundation paper of nearly all existing failure detectors [2]. This paper introduces the concept of unreliable failure detectors and studies how they can be used to solve Consensus in asynchronous systems with crash failures. It considers a distributed structure of a failure detector: each process has access to a local failure detector module. Each local module monitors a subset of the processes in the system, and maintains a list of those that it currently suspects to have crashed. These detectors are called 'unreliable' because it is likely for them to erroneously put some correct processes into their suspect lists. In addition, the paper characterizes a class of failure detector by specifying two properties: completeness and accuracy. Completeness means there is a time after which every process that crashes is permanently suspected by some correct process. Accuracy means that there is a time after which some correct process is never suspected by any correct process. Based on this, the paper develops the weakest class of failure detectors that satisfies these properties.

One of the contributions of this foundation paper is that it abstracts the structure, which is used in nearly every failure detectors. For example, the concept of completeness and accuracy can be used as a basic test standard to evaluate the performance of any kind of failure detectors. This paper expands the concept of completeness and accuracy. It categorizes the completeness into two parts: strong completeness and weak completeness. The accuracy is also categorized into four parts: Strong accuracy, weak accuracy, eventual strong accuracy and eventual weak accuracy (The concrete explanation of each term can be seen in [2]). By doing this, all failure detectors can be characterized into 8 labels. Another contribution of this paper is that it proves there exists an algorithm such that it can transform any kind of weak completeness into strong completeness. This algorithm works as follows: Every process p periodically sends a tuple (p, suspectsP) — where suspectsP denotes the set of processes that p suspects according to its local failure detector module Mp — to every process. When Receiving a message of the form (q, suspectsQ), p adds suspectsQ to outputP and removes q from outputP. Based on this transformation, every failure detector can increase its completeness meanwhile preserving its accuracy. Because of this algorithm, we can now consider only 4 kinds of failure detectors rather than 8.

Chen's Failure Detector

Based on the notions introduced by Chandra and Toueg, [7] designs a new failure detector algorithm based on the idea of heartbeat strategy. The basic idea of heartbeat strategy is illustrated

in Figure 1:

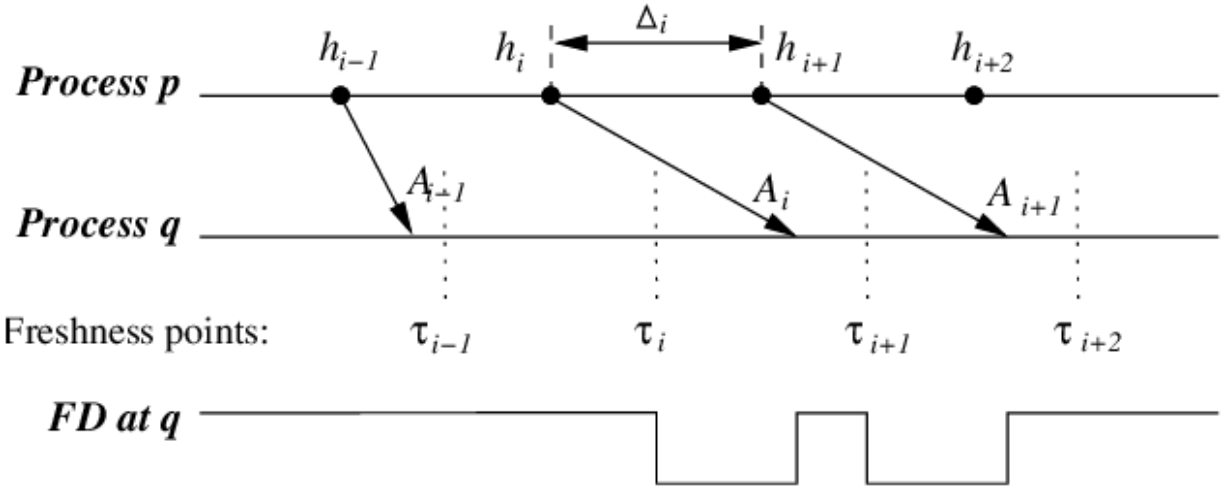

Figure 1: In this implementation, every process p periodically sends an "I'm alive" message to every other process. If a process q doesn't receive this message from p after the freshness points, q starts to suspect that p crashes. If later q receives this message from p, q will not suspect p any more. [4]

However, the problem of this heartbeat strategy is that even though the heartbeats are sent at regular intervals, because of the unstable network environment, the receipt time of process q can be very unstable too. To solve this problem, they improve the heartbeat strategy by using past arrival times to calculate the next expected arrival time EA. Moreover, considering the unstable network environment, they add a fixed parameter called δ as a safety margin for the next freshness points. As a result, the freshness points τ can be calculated by: $\tau = EA + \delta$. One key point of this algorithm is that it doesn't assume the existence of synchronized clocks for p and q. The process q can just use its local time to calculate the next expected arrival time, which not only makes it feasible in practice, but also makes it extensible for future improvement.

Bertier's Failure Detector

Based on Chen's work, [3] introduced a failure detector that satisfies strong completeness and eventual strong accuracy called 'adaptive failure detector'.

One contribution of this paper is that it compares two strategies that can be used in failure detectors: Heart-beat strategy and pinging strategy. The pinging strategy works as the following: A process p monitors a process q by sending "Are you alive?" messages to q periodically. Upon reception of such messages, the monitored process replies with an "I am alive" message. If process p times out on process q, it adds q to its list of suspected processes. If p later receives an "I am alive" message from q, p then removes q from its list of suspected processes. In this paper, it

demonstrates that the heart-beat strategy is better than pinging strategy because firstly, heart-beat strategy sends half as many as a pinging strategy, which decreases the pressure of the network flow. Secondly, it is easier to make a better estimation with a heartbeat message than by pinging another detector because in heart-beat strategy, the system only needs to estimate the transmission delay once. The meaning of this comparison is that in the future, people just need to consider heartbeat strategy as the foundation of their new failure detectors.

Another contribution of this paper is that it improves the failure detector algorithm from [7]. This paper uses the same algorithm basis as [7] to calculate the EA. However, instead of a fixed value, they use a dynamically changing value as a safety margin. This estimation supposes that the behaviour of the system is not constant. It adapts the safety margin each time it receives a message.

Asynchronous Failure Detector

A problem that comes with Chandra and Toueg's failure detector specification is that it usually requires the underlying system be implemented with additional synchrony assumptions [8]. In their paper, Achour Mostéfaoui et al. presented an asynchronous implementation of failure detectors based on a query-response mechanism that does not assume any kind of synchrony. The basic idea is to repeat the process of sending queries to all processes and receiving responses until they reach a certain number and compare the respondents of consecutive iterations to find out who is lost. The significance is that the failure detector need not have a clear time perception, and the number of failed nodes can be arbitrary. However, the model assumes reliable communication channels and the implementation is very likely to generate a large overhead in the network.

Accrual Failure Detector

Then there is another kind of failure detector called 'accrual failure detector' introduced by [5]. There are two main differences of implementations between this model and the conventional failure detectors. Firstly, the outputs from conventional failure detectors are deterministic Boolean values, which means that the failure detectors can only either trust or suspect processes. However, the outputs from accrual failure detectors are on a continuous scale, which means that they can provide a probability whether a process crashed or not. Secondly, in conventional failure detectors, monitoring (i.e. gathering information from other processes) and interpretation (i.e. deciding on whether a process crashes or not based on the information grasped from 'monitoring' behavior)

are both done by the failure detectors, while in accrual failure detectors, the failure detectors only need to do the monitoring and provide a value to the applications, and applications will do the interpretation based on the given value.

The comparison between conventional failure detectors and the new version tells us that no matter how the model changes, some features are always solid in failure detectors equipped with a heart-beat strategy. For instance, in this paper, it abstracts the failure detector model as three layers based on all behaviors that can be done on the receiving side: Monitoring, interpretation and action. They follow the same steps: 1. Calculating the expected arrival time of the next heart-beat 2. Monitoring the next heart-beat 3. Based on the actual arrival time of the next heart-beat and other information, do the interpretation to determine the states of the processes. 4. Depending on the states, the applications start to take specific actions. This structure can be very helpful to researchers because they can refer to this to design new FD algorithms or rebuild existing FD algorithms much more easily.

In contrast, the difference between conventional failure detectors and accrual failure detectors shows how a failure detector can be extended based on some fundamental structures. For example, the algorithms used to calculate the expected arrival time of the next heart-beat can be different if we compare [3] and [5]. This kind of freedom and flexibility is very necessary for designing new FD algorithms.

2.2 Benchmark

The present subsection talks about benchmark platform design and metrics used in benchmark algorithms.

Benchmark Platforms

Unfortunately, there is little prior work on benchmark platforms for failure detectors. So to get some basic idea on how to design a benchmark platform, we study Exathlon[9]. This paper designs a benchmark platform for explainable anomaly detection over time series, which can be used to reproduce some previous experiments and help users to design and create new algorithms. This platform consists of three components: 1. A curated anomaly data set. 2. A novel benchmark methodology for AD (anomaly detection) and ED (explanation discovery). 3. an end-to-end data science pipeline for implementing and evaluating AD and ED algorithms based on the provided dataset and methodology.

This paper provides a referable structure for designing benchmark platforms for failure detectors. Firstly, the dataset in this model is generated from a real environment where the researchers disturb some job executions to create some anomaly of data. As a result, the benchmark platform can use this dataset as a virtual environment to test AD algorithms. We can also do this for failure detectors. A huge amount of data from [3] is accessible to simulate some real network structure, which simplifies the structure of the benchmark platform. Secondly, this benchmark platform evaluates AD and ED algorithms in terms of two orthogonal aspects: functionality and computational performance, which implies that we should also find some well-organized evaluation methodologies for FD algorithms. Thirdly, the pipeline here is actually a kind of compiler: It compiles the high-level algorithm into some code that this benchmark platform can run. In this way, it is much easier for researchers to implement their FD algorithms, which allows them to reproduce some results or create some variants of FD algorithms based on existing algorithms.

Metrics

The quality and performance of a failure detector are useful in applications. Many systems only work on pre-determined conditions, like time and space constraints. Also, quality and performance measurement can help developers make decisions about failure detector designs very quickly. To benchmark a failure detector, Wei Chen et al. in their paper [7] propose a set of metrics under the assumption that the message flow follows certain probability distributions and the failure detector eventually reaches a steady state. The metrics describe the speed (how fast it detects failure) and accuracy (how well it avoids making mistakes) of a failure detector. In other words, the goal is to speed up true positive detection and bring down false positive rate.

In their paper, Wei Chen et al. propose three primary metrics and four derived metrics [7]. As for primary metrics, "detection time" is the measurement of the speed, which measures the time period of a process crash and its suspicion on a monitoring process. The other two are used to specify the accuracy. They are "mistake recurrence time" that denotes the time between two consecutive mistakes, and "mistake duration" that denotes the time the failure detector used to correct a mistake. The four derived metrics are "average mistake rate", "query accuracy probability", "good period duration" (good period means mistake-free) and "forward good period duration" that can all be computed based on time series data. To satisfy the requirements formed by the metrics above, mathematical transformations of variables and arguments in the algorithms are needed, but they are widely applicable.

The paper provides a clear insight to FD algorithm benchmark design. It contributes to the trivial implementation of a benchmark algorithm for failure detectors. Also, the metrics are fully extensible to new algorithms that do not even assume probabilistic behavior of the system. However, a few other important or at least alternate metrics and factors are not taken into account in Wei Chen et al.'s paper, for example lack of speed measurements, timeouts, false negative rate, and hardware consumption. Besides, the accuracy metrics are under the constraint of failure-free runs of a process. To enhance the quality and performance analysis, we may want to incorporate the measurement of timeouts, 99 percent detection time, the normal process rate, and monitor CPU, memory, as well as network usage during the detection, into our design.

2.3 Positioning of our approach

As a result, we will refer to the above papers to design our experiments and build a benchmark platform for failure detectors. The three primary metrics we decide to use in our benchmark design are "detection time", "number of false detection" and "mistake duration average" which we adopt from [4]. Also, hardware resources, namely "CPU usage" and "memory usage", will be taken into account. In our platform design, we will adopt the Exathlon architecture that involves novel benchmark methodology and a data pipeline.

The platform should not only be able to rebuild existing failure detectors, but also to test their performances based on network emulation and heartbeat messages generation. The goal is to define a set of languages that can explain the completeness and accuracy of FD algorithms, and provide scientists with an easy-to-use platform for designing and testing new algorithms. The features include separate network environment for different algorithms, customizable metrics, (virtualized) hardware monitoring, saving configuration and algorithms for later use etc. The time series dataset of the PlanetLab network from [4] will serve as a base line of evaluation, and generalization of different network settings will be our main issue of focus. The final result will be building a working software prototype that implements a clean user interface and integrates a built-in network environment and evaluation algorithms for benchmark.

3 Implementation of Failure Detectors

We plan to use Chen's failure detector, Bertier's failure detector, and the accrual failure detector as the foundation of our benchmark platform because all three failure detectors follow a similar method: calculating the next expected arrival time by using past arrival times of heartbeats. Therefore, to design our benchmark platform, we firstly implement these three failure detectors in Python. Section 3.1 introduces a new data structure that we design for our implementation. Sections 3.2 to 3.4 introduce how we implement the three failure detectors.

3.1 Record Data Structure

We designed a new data structure, Record, to store the arrival times of heartbeat messages. It requires the users to input the length of the Record, represented as n. This data structure only allows users to append new arrival times, like a queue until it is full. After that if users continue to append, it will dump the earliest arrival times and append the new arrival times into the structure. Moreover,this data structure supports additional operations. For example, users can calculate the sum of all arrival times in the Record by using the function get_sum(). Users can also calculate the difference between adjacent arrival times in the Record by using the function get_difference().

3.2 Implementation of Chen's Failure Detector

Chen's Failure Detector works in the following way: [7]

$EA_{(k+1)} \approx \frac{1}{n}\left(\sum_{i=k-n}^{k} A_i - \Delta_i * i\right) + (k+1).\Delta_i$

$\tau_{(k+1)} = \alpha_{(k+1)} + EA_{(k+1)}$

If we assume that process q receives heartbeat messages from process p, and considers the n most recent heartbeat messages, then the above parameters can be explained as the following: $A_1, A_2, ..., A_n$ are their reception times according to q's local clock. Δ_i represents the interrogation time of process p. α is a constant safety margin set by the user. τ_{i+1} is the estimation of next arrival date after receiving the ith heartbeat message from p.

According to this formula, we first transform this formula so that we can use the Record data structure. We transform it into:

$EA_{(k+1)} \approx \frac{1}{k}\left(A_1 + A_2 + ... + A_k\right) + \frac{(k+1)}{2}.\Delta_i$ when $k < n$.

$EA_{(k+1)} \approx \frac{1}{n}\left(A_{k-n+1} + A_{k-n+2} + ... + A_n\right) + \frac{(n+1)}{2}.\Delta_i$ when $k \geq n$.

By using the Record, they can be simplified into:

$EA_{(k+1)} \approx \frac{1}{n}.Record.get\_sum() + \frac{(n+1)}{2}.\Delta_i$ where n is the current length of the Record.

3.3 Implementation of Bertier's Failure Detector

Bertier's Failure Detector is an extension of Chen's. It follows the same way as Chen's to calculate $EA_{(k+1)}$. However, instead of the constant α, it uses a dynamic α, which is calculated in the following way: [3]

$error_{(k)} = A_k - EA_{(k)} - delay_{(k)}$

$delay_{(k+1)} = delay_{(k)} + \gamma.error_{(k)}$

$var_{(k+1)} = var_{(k)} + \gamma.(|error_{(k)}| - var_{(k)})$

$\alpha_{(k+1)} = \beta.delay_{(k+1)} + \phi.var_{(k+1)}$

Here the parameters $delay$, var, γ, β and ϕ are all set by the user.

3.4 Implementation of Accrual Failure Detector

Figure 2 from [5] demonstrates how the accrual failure detector works:

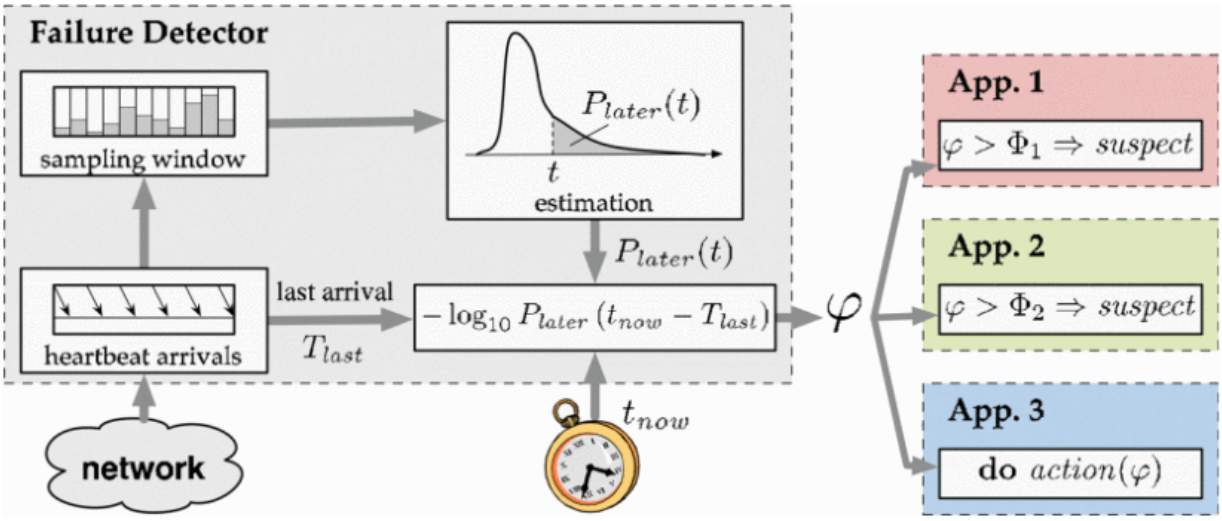

Figure 2: Information flow in the proposed implementation of the ϕ failure detector, as seen at process q. Heartbeat arrivals arrive from the network and their arrival time are stored in the sampling window. Past samples are used to estimate some arrival distribution. The time of last arrival T_{last}, the current time t_{now} and the estimated distribution are used to compute the current value of ϕ. Applications trigger suspicions based on some threshold (Φ_1 for App. 1 and Φ_2 for App. 2), or execute some actions as a function of Φ (App. 3).

However, we cannot directly apply this algorithm into our benchmark platform. One problem is that this algorithm requires a dynamic network environment to continuously detect the current

time t_{now}. But in our benchmark platform, we are using data from a real network to simulate an environment, which makes the detection of t_{now} impossible. As a result, we need to revise this formula a little bit through the following steps:

$-\log_{10} P_{later}(t_q - T_{last}) \geq \phi$

$P_{later}(t_q - T_{last}) \leq 10^{-\phi}$

According to the distribution of P_{later}, we can see there exists a value $t_q = t_{min}$ such that for any $t_{now} \geq t_{min}$, process q will suspect process p at time t_{now}. Therefore, we can understand the value of t_{min} as the next expected arrival date after q receives the latest heart-beat message at T_{last}, which satisfies:

$P_{later}(t_{min} - T_{last}) = 10^{-\phi}$

We notice that the notation T_{last}, t_{min} is equivalent to the notation A_i, τ_{i+1} respectively in Chen's and Bertier's failure detectors. As a result, the structure of accrual failure detector can be inherited from the previous two failure detectors.

To get the distribution P_{last}, we can use the following method:

$mean = mean(Record.get\_difference())$

$std = std(Record.get\_difference())$

$P_{last} = N(mean, std)$, N is the normal distribution

However, this method cannot work when there are only 1 or 2 arrival dates in the Record. To fix this issue, we define that when there is only 1 arrival date in the Record, we let $t_{min} = T_{last} + \Delta_i$ where Δ_i is the interrogation time of process p. When there are 2 arrival dates in the Record, we let $t_{min} = T_{last} + Record.get\_difference()$.

4 Language

After successfully implementing three failure detectors in Python, our next task is to design a language that allows future users to build their own failure detectors easily. Based on our structure, we find that there are two extensible parts for future users. The first part is the Record data structure. Future users can introduce new parameters and new functions to this data structure by inheriting it. The second part is the method to calculate the next expected arrival time. Future users can combine the existing parameters with new parameters they introduce to design their own calculation of the next expected arrival time. Based on this, we allow users to import two files. One is the extended data structure of Record. The other one is a txt file that

specifies how the users want to calculate the next expected arrival time. The format of the txt file is the following:

Outside: Because we use a loop to collect each arrival time from the virtual network environment, the meaning of 'outside' here is for parameters placed outside the loop, like some constant values that will not change within the loop (for example α in Chen's failure detector) and some initial values which will change(for example var and delay in Bertier's failure detector).

Inside: In contrast to the 'outside', the parameters here are placed inside the loop, which means that they will change continuously (for example the length and the sum of a Record class).

EA: This part allows the user to define their own method to calculate the next expected arrival time based on all the parameters they defined before and the built-in parameters.

We successfully translate all three failure detectors in this way. For example, to implement a Chen-based failure detector where the length of Record is 1,000 and α is 100,000, we can write:

```
Outside: N=1000,
alpha=100000;
Inside: length=object->get_length(),
sum=object->get_sum();
EA: alpha+sum/length+((length+1)/2) * delta;
```

In this case, whether the 'object' is the original Record or the inherited Record is determined by the user. The function after the arrow represents what kind of operation the user wants the Record to perform.

To implement Bertier's failure detector where the length of Record is 1000, delay is 0, var is 0, γ is 0.01, β is 1 and ϕ is 4, we can write:

```
Outside: N=1000,
delay=0,
var=0,
gamma=0.01,
beta=1,
phi=4;
Inside: length=object->get_length(),
sum=object->get_sum(),
error=A–E–delay ,
```

```
delay=delay+gamma*error,
var=var+gamma*(np.abs(error) - var),
alpha=beta*delay+phi*var;
EA: alpha+sum/length+((length+1)/2) * delta;
```

In this case, there are many built-in parameters. The 'delta' here represents the interrogation delay of this virtual network environment. It is not influenced by the users so it has a constant value 100,000,000. The 'A' here represents the latest arrival time based on q's local clock. The 'E' here represents the previous calculated expected arrival time before 'A'. Moreover, we allow users to write numpy (here is np) commands in the txt file.

To implement an accrual failure detector where the length of Record is 1,000 and ϕ is 1, we can write:

```
Outside: N=1000&delta&1;
Inside: expected_interval=object->get_interval();
EA: A+expected_interval;
```

In this case, to calculate the interval, we design a new data structure inherited from the Record class. This new data structure now has 3 initial inputs (length, delta, ϕ) instead of 1 (only length). Moreover, this new data structure has a new function called *get_interval*().

5 Benchmark

In this section, we will discuss how we design a benchmark system in detail. First we explain the metrics we use to evaluate the QoS of a failure detector. Then we present and analyze our algorithm for calculating a failure detector's score based on its metrics.

5.1 Metrics and Weights

We decide on seven metrics to use in our benchmark algorithm: detection time (dt), probability of accuracy (pa), standard deviation of detection time (std_dt), standard deviation of probability of accuracy (std_pa), mistake duration (md), CPU time (CPU) and memory usage ($memory$). We set weights to the metrics to evaluate the failure detectors more fairly. The weights can be customizable by minor changes in the source code, but in this section, we fix the weights which we believe are optimized values. Detection time and probability of accuracy are the two primary metrics because the main job of a failure detector is to detect failures quickly and accurately.

Detection time measures the period from the time a failure occurs until the failure detector begins to suspect the node; the shorter the better. Probability of accuracy is the ratio of the number of correct detections to the number of total detection; the higher the better. They are assigned a weight of 20% each. The standard deviation of the two metrics are also taken into account because the benchmark runs on different links of nodes and the stability of failure detectors matters. Each of the standard deviation are assigned a 15% weight. One minor yet necessary metric is mistake duration that is the accumulated period taken by failure detectors to correct a mistake. The other minor metrics are CPU time and memory usage which are easy to understand by their names. We assign each of the minor metrics a weight of 10%. In total, the major metrics contribute 70% of the performance benchmark while minor metrics contribute the rest.

5.2 Algorithm of Calculating Scores

We decide to use Chen's failure detector and accrual failure detector as the baselines for our benchmark system. Figures 3 & 4 show the performance of the accrual and Chen's failure detector respectively. It is easy to observe that Chen's failure detector consumes the least hardware resources but is bad at probability of accuracy and mistake duration, while accrual failure detector provides the best probability of accuracy and mistake duration but runs very slow. Using the two opposites as baselines generates a balanced evaluation of performance.

parameters	mistake duration (ms)	detecion time (ms)	pa	cpu time (s)	memory (MB)	std of detection time (ms)	std of pa
n=100,phi=1	18101.63	101.07	94.10%	0.52	112.87	0.48	1.36%
n=200,phi=1	17502.34	101.01	95.00%	0.5	112.91	0.43	1.27%
n=300,phi=1	17380.42	101.02	95.23%	0.5	112.98	0.37	1.32%
n=400,phi=1	17291.43	101.06	95.37%	0.48	112.95	0.35	1.34%
n=500,phi=1	17185.43	101.12	95.53%	0.52	112.96	0.34	1.29%
n=600,phi=1	17142.25	101.14	95.60%	0.5	112.94	0.33	1.31%
n=700,phi=1	17049.61	101.15	95.67%	0.5	112.95	0.33	1.31%
n=800,phi=1	16945.94	101.15	95.75%	0.51	112.95	0.32	1.29%
n=900,phi=1	16920.91	101.15	95.79%	0.52	112.89	0.33	1.30%
n=1000,phi=1	16896.83	101.16	95.83%	0.49	113.12	0.34	1.30%
n=100,phi=10	9799.25	105.32	99.54%	0.5	112.95	2.39	0.19%
n=200,phi=10	9543.46	105	99.71%	0.49	112.96	2.11	0.13%
n=300,phi=10	9455.78	105.04	99.74%	0.48	112.89	1.85	0.11%
n=400,phi=10	9428.69	105.26	99.75%	0.49	112.93	1.73	0.10%
n=500,phi=10	9416.51	105.54	99.77%	0.48	112.96	1.69	0.10%
n=600,phi=10	9413.78	105.68	99.77%	0.51	112.97	1.64	0.09%
n=700,phi=10	9381.96	105.7	99.78%	0.49	112.92	1.62	0.09%
n=800,phi=10	9337.38	105.71	99.79%	0.5	112.94	1.6	0.09%
n=900,phi=10	9352.4	105.72	99.79%	0.49	112.99	1.66	0.09%
n=1000,phi=10	9364.9	105.75	99.79%	0.53	113.04	1.68	0.08%

Figure 3: The evaluations of metrics for accrual failure detector for ϕ=1 and 10. n is ranging from 100 to 1000 with step 100.

parameters	mistake duration (ms)	detecion time (ms)	pa	cpu time (s)	memory (MB)	std of detection time (ms)	std of pa
n=100,alpha=0	289146.39	99.98	0.6972	0.34	84.69	0.88	0.0605
n=200,alpha=0	515860.34	99.99	0.7071	0.36	84.68	0.9	0.0673
n=300,alpha=0	745535.06	99.98	0.7055	0.34	84.65	0.9	0.0679
n=400,alpha=0	975766	99.98	0.7024	0.35	84.68	0.9	0.0676
n=500,alpha=0	1205866.59	99.99	0.7009	0.33	84.7	0.9	0.0686
n=600,alpha=0	1436092.4	99.99	0.6975	0.36	84.7	0.9	0.0685
n=700,alpha=0	1666257.6	99.99	0.694	0.34	84.69	0.9	0.0686
n=800,alpha=0	1896238.79	100	0.691	0.35	84.69	0.9	0.0696
n=900,alpha=0	2126233.75	99.99	0.6873	0.34	84.7	0.9	0.0698
n=1000,alpha=0	2356274.1	99.99	0.6833	0.36	84.67	0.9	0.0703
n=100,alpha=100,000	286542.1	100.08	0.7729	0.35	84.7	0.88	0.0839
n=200,alpha=100,000	513317.82	100.09	0.7759	0.36	84.64	0.9	0.0841
n=300,alpha=100,000	742963.98	100.08	0.7719	0.35	84.68	0.9	0.0839
n=400,alpha=100,000	973156.07	100.08	0.7676	0.36	84.64	0.9	0.0836
n=500,alpha=100,000	1203231.82	100.09	0.7643	0.36	84.65	0.9	0.0833
n=600,alpha=100,000	1433417.95	100.09	0.7599	0.35	84.68	0.9	0.0831
n=700,alpha=100,000	1663542.3	100.09	0.7555	0.35	84.63	0.9	0.0829
n=800,alpha=100,000	1893488.9	100.1	0.7516	0.36	84.66	0.9	0.0829
n=900,alpha=100,000	2123441.39	100.09	0.747	0.35	84.69	0.9	0.083
n=1000,alpha=100,000	2353438.5	100.09	0.7423	0.36	84.7	0.9	0.0833

Figure 4: The evaluations of metrics for Chen's failure detector for α=0 and 100000. n is ranging from 100 to 1000 with step 100.

The two tables suggest the upper and lower bounds of the performance of the two failure detectors on different metrics. Based on these two tables, we can start to design benchmark algorithm for each metric. For each metric, we set two standards: high standard and low standard. Their values are determined by the min (or max) value of each metric. For example, the smaller the detection time is, the better the failure detector is. Therefore, we set the high standard of detection time as the minimum value, and the low standard of detection time as the maximum value. All the other metrics use the same method to determine their high standard values and low standard values. Then we set the score of high standard of each metric as 90 points and the score of low standard of each metric as 60 points. The results can be viewed in Figure 5.

	mistake duration (ms)	detecion time (ms)	pa	cpu time (s)	memory (MB)	std of detection time (ms)	std of pa	scores
high standard	9337.38	99.98	0.9979	0.33	84.63	0.32	0.0008	90
low standard	2356274.1	105.75	0.6833	0.53	113.12	2.39	0.0841	60
weights	0.1	0.2	0.2	0.1	0.1	0.15	0.15	1

Figure 5: The high standard and low standard of each metric and their corresponding weights.

By doing this, for each metric, we can design a function for calculating its score that takes in the actual performance data and outputs a score within the range of 0 to 100. For this project, we choose the linear function for all metrics for simplicity. In this way, the total score is given by

$$S_{total} = S_{dt} * w_{dt} + S_{pa} * w_{pa} + S_{std\_dt} * w_{std\_dt} + S_{std\_pa} * w_{std\_pa} + S_{md} * w_{md} + S_{CPU} * w_{CPU} + S_{memory} * w_{memory}$$

where S_i denotes the score of metric i described in subsection 5.1 and w_i denotes their corresponding weight.

6 Visualization

The final crucial part of the benchmarking platform is the visualization component. We design a horizontal bar chart and a line chart to visualize the results after running the benchmark algorithm. For this section, we run the benchmark on Chen's, accural and Bertier's failures detectors (the plarform is Windows 10 64-bit Enterprise Edition on i7-11800h@2.30GHz with 16GB DDR4-SDRAM) and will use the data as examples in the illustration. Due to the time and workload constraint, we do not make the visualization interactive. But with minor changes in the source code, the user can explore different views of the data.

Horizontal Bar Chart

In the horizontal bar chart view, the x-axis is the number of score, and the y-axis is the name of failure detector. The plot exhibits the scores of all metrics of all failure detectors grouped by name of failure detector as shown in Figure 6. Users can specify (in the source code) particular metric scores to plot.

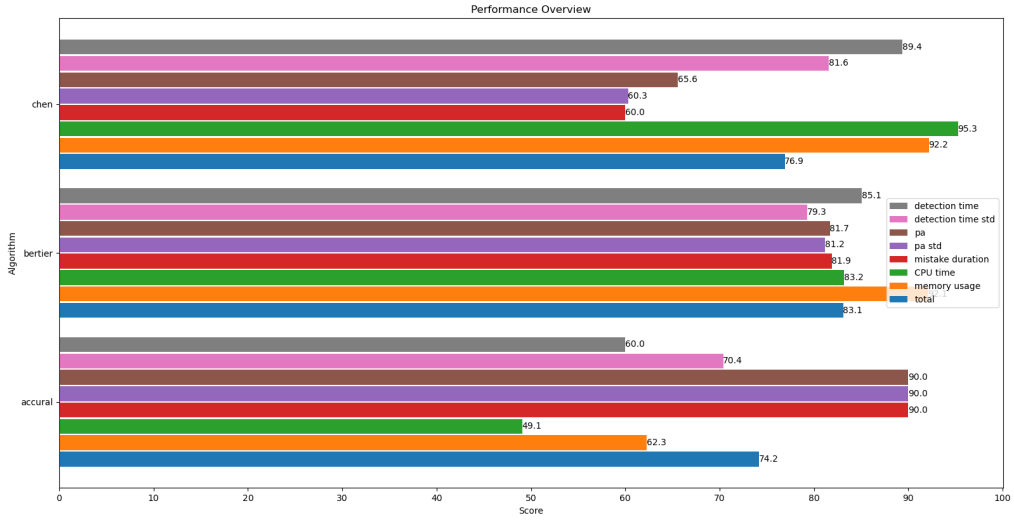

Figure 6: Horizontal bar chart plot of performance of Chen's, accrual and Bertier's failure detectors.

Line Chart

In the line chart view, the x-axis is the name of metric, and the y-axis is the number of score. The plot also exhibits the scores of all metrics of all failure detectors grouped by name of failure detector as shown in Figure 7. Users can specify (in the source code) particular metric scores to

plot.

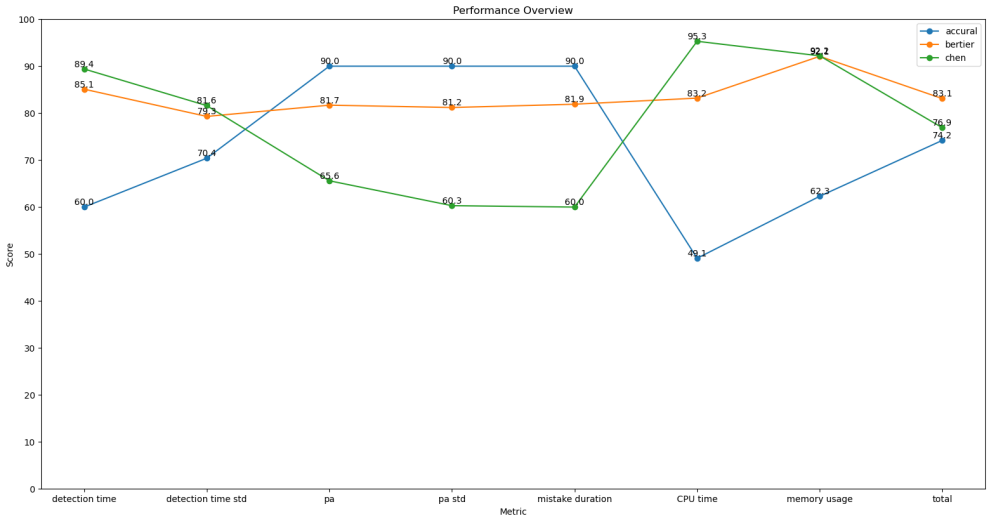

Figure 7: Line chart plot of performance of Chen's, accrual and Bertier's failure detectors.

7 Conclusion

Our project builds a novel benchmark platform for failure detectors. This system not only allows users to rebuild some existing failure detectors, but also allows them to design and evaluate new failure detectors without writing a whole program. Moreover, this system makes the comparison of different failure detectors much easier and more efficient. This project can be improved in the following ways: 1. We can design a better function for calculating the scores since currently we are using a linear function for each of the metrics 2. Maybe there are other metrics that can be used to evaluate the performance of a failure detector. If we can find any, it can make this benchmark system more complete and accurate.

References

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, p. 374–382, apr 1985. [Online]. Available: https://doi.org/10.1145/3149.214121

[2] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, p. 225–267, mar 1996. [Online]. Available: https://doi.org/10.1145/226643.226647

[3] M. Bertier, O. Marin, and P. Sens, "Implementation and performance evaluation of an adaptable failure detector," in *Proceedings International Conference on Dependable Systems and Networks*, June 2002, pp. 354–363.

[4] ——, "Performance analysis of a hierarchical failure detector," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, June 2003, pp. 635–644.

[5] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The ϕ accrual failure detector," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, 2004, pp. 66–78.

[6] L. Rizzo, "Dummynet: A simple approach to the evaluation of network protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 1, p. 31–41, jan 1997. [Online]. Available: https://doi.org/10.1145/251007.251012

[7] W. Chen, S. Toueg, and M. Aguilera, "On the quality of service of failure detectors," *IEEE Transactions on Computers*, vol. 51, no. 1, pp. 13–32, Jan 2002.

[8] A. Mostéfaoui, E. Mourgaya, and M. Raynal, "Asynchronous implementation of failure detectors," 07 2003, pp. 351– 360.

[9] V. Jacob, F. Song, A. Stiegler, B. Rad, Y. Diao, and N. Tatbul, "Exathlon: A benchmark for explainable anomaly detection over time series," 2021.