A Novel Failure Detection Solution with Multiple QoS in Data Center Networks

Kai Shen
School of Software
Shanghai Jiao Tong University
Shanghai, China
knshen@sjtu.edu.cn

????
????
Shanghai Jiao Tong University
Shanghai, China
????@sjtu.edu.cn

ABSTRACT

Large scale data center networks are complex due to hundreds of thousands devices and links in them. Failures of these devices and links sometimes can lead to user-perceived service interruptions. Thus automated failure detection plays a very important role in maintaining the reliability of data centers. Many researches have been performed on failure detection in networks or distributed systems, but few of them identifies quality of service (QoS) multiplicity of failure detection in data center networks.

In this paper, to tackle with this problem, we firstly divide network devices into two categories: imperative devices whose failures need to be detected in real time, and non-imperative devices. Subsequently, we leverage a co-detection approach named K-detectors to detect failures of imperative devices, while the key idea of non-imperative devices failure detection is to analyze network traffic changes over time.

We evaluate our approach on a simulated network built by ns-3. Experimental results show that for imperative devices, our approach has a better QoS except for limited detection time increase in the presence of normal message loss and different size of failures. For non-imperative devices, our approach outperform existing work in terms of accuracy and failure discrimination.

Keywords

Failure detection; Quality of Service; Data center networks

1. INTRODUCTION

Data center is a pool of resources interconnected using a communication network, a broad range of services (e.g., Web search, data analytic, storage backup) are hosted on data centers. Data center networks play a pivotal role in data centers as they integrate various resources and connect hundreds of thousands devices together (e.g. routers, switches, servers). For example, the Microsoft Live online services are supported by a Chicago-based data center spanning more than 700,000 square feet [14].

Due to the complexity and scale of data center networks, failures are common. Hence it is crucial to detect failures in the networks efficiently and effectively. Traditionally, domain experts are responsible for failure detection with the help of network monitoring data. But such a manual process is time-consuming, error-prone and not scalable. So automated failure detection approaches are useful in terms of helping reduce mean-time-to-recovery and service disruption.

However, failure detection in data center networks should not be simplex. On one hand, data center networks are reliable due to the built-in redundancy of data center networks like hardware redundancy and fault-tolerant routing protocols. Evidence shows that for about 80% of the links and about 60% of the devices, there is a more than four 9's of availability [6]. On the other hand, many applications hosted on data centers have very strict timing constraints [22], they require failure detectors to provide quality of service (QoS) with quantitative timeliness guarantees.

Unfortunately, few researches have noticed QoS difference in failure detection of data center networks. Some researches on failure detection architecture such as [5][23][7][24][24] focus on the scalability issue of process crash detection in large scale distributed systems without considering underlying network topologies, while other work on failure localization in data center networks such as [1][11][13] treat all the network devices equally and utilize a general detection approach.

To address this issue, we firstly divide network devices into two categories: imperative devices and non-imperative devices. Imperative devices mainly include servers, top of rack switches and links between them, while non-imperative devices include other routers, switches and links. The main difference between them is that imperative devices have strict timing constraints of QoS as the failures of them may affect application functionalities. Naturally, our detection approach is two-fold: for imperative devices, we propose a co-detection model called K-detectors model to ensure the timeliness, soundness and accuracy at the same time. For non-imperative devices, we utilize a data mining based model to produce a ranked list consisting of suspect devices by extracting network traffic features. To evaluate our work, we compare our approach with hierarchical detection architecture [5] in terms of detection for imperative devices, and the work of Herodotou et al. [11] in terms of detection for non-imperative devices. The experimental results prove the effectiveness of our work.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

This paper makes the following contributions:

- We identify the differences in QoS of failure detection in data center networks.
- We further divide network devices into two categories and propose effective failure detection approaches for these two kinds of devices respectively.
- We set up a series of evaluation metrics to evaluate different aspects of failure detection approaches.
- We implement our approach as well as other baselines on the same simulated data center network and prove the effectiveness of our approach.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents data center networks architecture. Section 4 provides an overview of our approach, while the details are described in Section 5. The experimental evaluation along with experimental results is presented in Section 6. Finally, Section 7 concludes this paper.

2. RELATED WORK

In this section, we will present various relevant works including failure detectors, failure detection architecture in distributed systems and failure detection in networks.

2.1 Failure Detectors

A failure detector (FD) is widely recognized as an oracle to intelligently suspect failed processes [3]. The monitored process periodically send heartbeat messages to its detector to prove its liveness.

Chen et al. proposed a FD that provides QoS [4]. It estimates the expected arrival time (EAs) of the next heartbeat message according to a slide window storing n most recent arrived messages. EAs determine the deadline that the detector will wait for the next heartbeat before suspecting the monitored process. *Chen FD* can adjust current network condition and set the timeout threshold adaptively by referring to recent heartbeats. Bertier et al. proposed a similar FD [2], whereas it uses a dynamic way to compute error margin of *Chen FD*. Other FDs like ϕ *Accrual FD* [9] and *Satzger FD* [19] can output a continuous accrual failure value rather than a binary value which stands for the failure probability of the process at any time.

2.2 Failure Detection Architecture

A failure detection architecture aims to provide the service of monitoring nodes in large scale distributed systems in a scalable way. Roughly speaking, there are two kinds of architectures. The first one is hierarchical architecture [5], all the nodes are partitioned into different groups, each group has a leader node. Within a group, the leader node is responsible for monitoring all the other nodes. Leader nodes periodically send node status information of his group to other leader nodes. Hierarchical architecture can reduce the number of heartbeat messages effectively, but it exists *Single Point of Failure* problem, when the leader node crashes, it needs to select a new leader.

Another alternative solution is Gossip-Style architecture [23]. Each node maintains a list containing the *heartbeat counter* for each node in the system. Every T_{gossip} seconds, each node firstly update *heartbeat counter* of itself and then randomly select another node to send its list to. Upon receiving a message, the node will merge two lists and update each node's *heartbeat counter* with the bigger one. If the *heartbeat counter* does not update after T_{fail} seconds, the node will be marked as crashed. If the *heartbeat counter* does not update after $T_{cleanup}$ ($T_{cleanup} \geq T_{fail}$) seconds, it will be removed from the list. This approach has very low bandwidth occupation because it only produces n messages each epoch where n is the size of the system. However, with the increasing of crashing nodes, it needs more time to detect these failures.

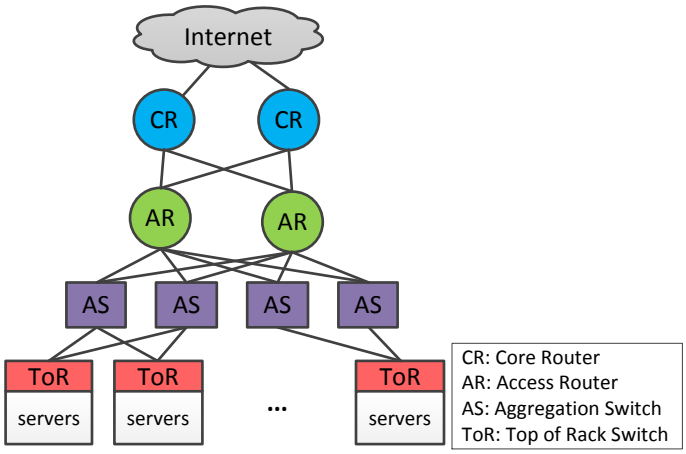

Figure 1: Conventional Data Center Network Topology

2.3 Failure Detection in Networks

Different from the work mentioned above, failure detection in networks take network topology into consideration and localize failure positions at IP layer. Gill et al. present the first large-scale survey on failures in data center networks [6].

Also, there exists a large body of work on applying statistical approaches to failure localization in networks. *Sherlock* [1] localizes failures to limited network components by capturing dependencies between components. *Shrink* [13] leverages Bayesian Network model to diagnose root cause of IP network failures. Herodotou et al. [11] actively inject ping traffic into network and analyze the ping results overlaid on top of the network topology, it generates a ranked list containing the suspected failed device and links along with their failure score that can best explain the observed ping data.

3. DATA CENTER NETWORK ARCHITECTURE

In this section, we will take a brief look at common data center network architecture.

A data center is a facility used to house computer systems and associated components, it usually contains hundreds of thousands devices and links. Modern data centers use hierarchical architectures. Figure 1 depicts a conventional data center network architecture [6][8][11]. At the bottom layer are racks of servers, typically there are 20 to 40 server-

s in each rack, and each server in the same rack connects to a Top of Rack (ToR) Switch via a 1 Gbps link. Each ToR then connects to a primary and a backup Aggregation Switch (AS) for redundancy. ASs are connected to Access Routers (ARs) forming a complete bipartite graph. Finally, ARs are connected to Core Routers (CRs). CRs are at the boundary of a data center network.

To limit overheads and also to isolate different services, servers are partitioned into virtual LANs (VLANs). All links in the data center use Ethernet as link layer protocol and physical connections are a mix of copper and fiber cables. In addition to the devices in Figure 1, there may exist other devices like Load Balancers (LBs) and Firewalls.

Modern data center usually employs *Equal Cost MultiPath (ECMP)* routing within the data center for fault tolerance, which means that given a source and destination, there are multiple paths from the source to the destination. Thus in this work, we will apply ECMP routing to the simulated network by enabling ECMP routing configurations in ns-3.

4. APPROACH OVERVIEW

In this section, we will introduce the whole design of our failure detection approach.

As we mentioned in Section 1, different network device should have different quality of service (QoS). Gill et al. [6] have shown the diversity of different devices' failure characteristics. Unfortunately, none of the work in Section 2 treats these devices differently. We firstly divide network devices into two categories.

- **Imperative Device.** This category includes servers, ToRs and links between ToRs and servers. Typically servers run various user applications and store user data. The crash of servers may cause user-perceived faults. The crash of a ToR will make servers under it unreachable. Also Gill et al. declare that the downtime of ToRs may be very long. Thus the failure detection for these devices must be fast enough besides other requirements.

- **Non-imperative Device.** It includes links, other routers and switches. A single failure of a link or a router does not affect the functionality of a data center because of built-in redundancy in data centers. So the data center administrators care more about whether the detection service is accurate enough to help them localize the real faults.

Figure 2 illustrates the entire architecture of our failure detection system. It mainly contains two modules. The upper half part illustrates the module for server failure detection. It is a combination of centralized and decentralized structures. A server can both play the role of a detector and a monitored object. Each server is monitored by multiple FD processes installed on different servers for redundancy. A server can also monitor a group of servers. Once a FD determines the crash of a server, it will send a crash report message to **Crash Decider**. **Crash Decider** will determine the final status of this server. On top of this, **Crash Decider** is also responsible for detecting ToR failures by combining all the server crash report messages together.

For the detection of non-imperetative devices, **Traffic Collector** periodically collects traffic data of routers and switches and send telemetry data to **Traffic Analysis Worker**.

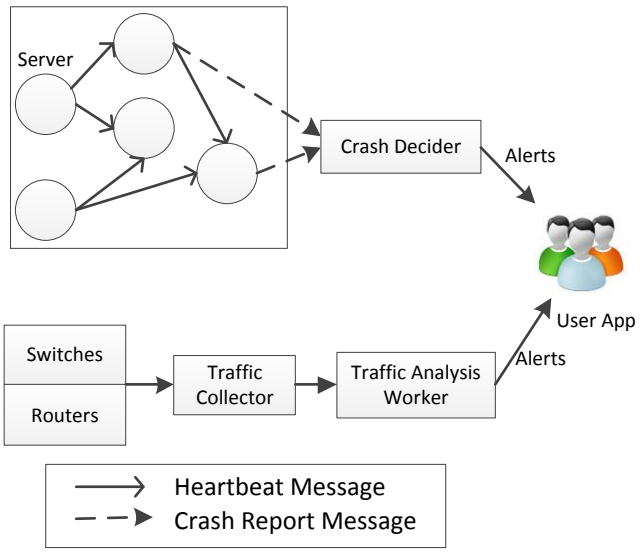

Figure 2: Failure Detection System Architecture

Traffic Analysis Worker finally generate failure alerts by performing analysis on these data.

5. FAILURE DETECTION APPROACH

In this section, we will discuss the detailed design of different aspects of the failure detection system respectively.

5.1 K-detectors Model

To detect the crash of servers and ToRs effectively and efficiently, this paper propose a co-detection model named K-detectors model. Our detection model is based on the assumption that once the server or ToR crashes, it never recovers by itself. We first refine the QoS of failure detection for servers:

- **Efficiency.** It measures how fast the system can report the crash after it crashes. The system is more efficient if it takes less time to detect a failure. In particular, server failures should be detected in real time.

- **Completeness.** It measures whether the system can detect all the crashed objects. The system is more complete if it rules out fewer true crashed ones.

- **Soundness.** In some scenarios, the behavior of the network may vary. For example, message loss and message delay may happen sometimes. The detection system should be sound enough to distinguish between such conditions and a real crash.

In K-detectors model, each server is monitored by K FDs on different servers where K is a parameter tuned by user applications. When a FD considers a server crashes, it sends a crash report message to **Crash Decider**. If the output of the FD transforms from suspect to trust, it re-sends a message to revoke the most recent crash report.

$$S_{s,t} = \begin{cases} \text{suspect} & \text{if } |CRM_{s,t}| \geq \lfloor (K+1)/2 \rfloor \\ \text{trust} & \text{otherwise} \end{cases} \quad (1)$$

where $CRM_{s,t}$ denotes the collection of crash report messages for server s until time t, $S_{s,t}$ denotes the status of server s at time t. If **Crash Decider** receives $\lfloor (K+1)/2 \rfloor$ crash messages of a server, it considers the server to have crashed. So a server is still under monitor even when a number of its FDs crash.

5.2 Monitoring Rule Inference

A monitoring rule defines the monitoring relationships among a group of servers. To form the K-detectors model and also to meet QoS to the greatest extent, we propose the following specifications of a monitoring rule:

1. Each sever monitors K servers. This balances monitor intensity of each server.
2. A server cannot monitor itself.
3. A server is monitored by K different servers.
4. $K-1$ of the K detectors are on the servers at the same rack.
5. The monitoring rule should ensure each link covered by heartbeat messages.
6. Each rack monitors servers of other racks which vary from different racks averagely.

Rationale

Intuitively, the use of K FDs is able to adapt to bursty-traffic conditions like message loss because the determination of a failure requires at least $\lfloor (K+1)/2 \rfloor$ FDs' agreement. Also, K-detectors can prevent crashes of FDs to some extent. Theoretically, a failure can still be detected even when $K - \lfloor (K+1)/2 \rfloor = \lfloor K/2 \rfloor$ of its detectors crash.

On the other hand, the fourth specification above can make the best of locality of LAN, which reduces the probability of degeneration on detection time in turn.

Specification five is the presupposition of link failure detection. It is easy to see that, some links cannot be distinguished from faulty links without this specification.

The last one is an extension of specification five. It is for the consideration of ToRs' failure detection, we will discuss it in detail in Section 5.4.

5.3 FD Algorithm

FD algorithm is used to decide the status of one monitored object by one FD. Our FD algorithm is almost the same as *Chen FD* [4] except for some minor changes.

$$EA_{l+1} \approx \frac{1}{n}(\sum_{i=1}^{n} A_i - \eta s_i) + (l+1)\eta \tag{2}$$

this formula estimates the expected arriving time of next message, where s_i is the sequence number of heartbeat message, A_i is the receipt time of message i, η is message send interval. Obviously, this formula normalizes each A_i by shifting it backward ηs_i time units.

After that, we utilize Hyperbolic Tangent Function to model current suspicious level:

$$\tau(t_{now}) = |\, tanh(t_{now} - EA_{l+1}) \,| \tag{3}$$

Hyperbolic Tangent Function is one of the widely used activation functions in neural networks. The suspicious level $\tau(t_{now})$ always ranges from 0 to 1, when t_{now} grows bigger, the suspicious level is closer to 1, which implies a higher failure probability. A continuous valued rather than binary valued output decouples the interpretation of failure data from the failure monitoring mechanism, user applications can set their own threshold according to their demands.

5.4 Failure Detection for ToRs

Apart from determination of servers' crashes, **Crash Decider** is also responsible for failure detection for ToRs. The essential idea is that if most or even all servers in a rack is reported crash by FDs in other racks, it is more likely that the top of rack switch crashes rather than the servers.

$$S_{ToR,t} = \begin{cases} \text{suspect} & \text{if} \quad |\,\{s \,|\, s \in ToR, \\ & \qquad CRMO_{s,t} \neq \emptyset, \\ & \qquad |\, CRM_{s,t} \,| = 1\}\,| \geq \alpha \cdot m \\ \text{trust} & \text{otherwise} \end{cases} \tag{4}$$

where m is the number of servers per rack, α is a proportionality coefficient measuring how strict the criterion is to determine failure of a ToR. $CRMO_{s,t}$ denotes the collection of crash report messages of s coming from outside racks until time t. When **Crash Decider** gets more than $\alpha \cdot m$ crash report messages of servers under the ToR crash, it will report the crash of ToR. The last specification in Section 5.2 ensures that once a ToR is down, each other rack has the chance to find this failure. Suppose there are n racks in total, according to this specification, for each rack r, each other rack will monitor $m/(n-1)$ servers of rack r. An extreme counter-example is that if the outside detectors of rack 1 servers are all in rack 2, once ToR of rack 2 is down, **Crash Decider** never perceives the status of ToR of rack 1.

Upper Bound of α

Theoretically, a smaller α means a more aggressive detection. We can set an appropriate α empirically by looking back on historical data on ToR crashes. Suppose we have known the ToR crash probability p, there are n racks (each one has a ToR) and crashes of ToRs are subject to *binomial distribution*. When there are k ToR crashes, the number of server crash report messages we can observe m' should be

$$m' = \frac{m}{n-1}(n-k) = \frac{m(n-k)}{n-1} \tag{5}$$

Thus the value of α is at most:

$$\alpha = \frac{m'}{m} = \frac{n-k}{n-1} \tag{6}$$

Next, if given a confidence parameter p_0,

$$\begin{aligned} &\alpha \leq \alpha_0 = \frac{m'_0}{m} = \frac{n-k_0}{n-1} \\ &\text{where } k_0 \text{ is the minimum of } k \\ &s.t. \quad Pr(x \leq k) = \sum_{r=0}^{k} C_n^r p^r (1-p)^{n-r} \geq p_0 \end{aligned} \tag{7}$$

$Pr(x \leq k)$ denotes the probability of number of ToR crashes is no more than k. Hence the theoretical upper bound of α is α_0. Actually, because of server crashes, the number of

crash report messages **Crash Decider** can receive may less than observe m'_0, so the value of α should be less than its upper bound.

To sum up, Algorithm 1 illustrates the pseudo-code for overall detection algorithm of **Crash Decider**. It firstly

Algorithm 1 Detection Algorithm of **Crash Decider**

Input: K, η, n, α, m, τ_m
Output: $S_{i,t}$, $i \in$ Imperative Device

```
1: for i in Imperative Devices do
2:    S_{i,t_0} ← trust
3: end for
4: while True do
5:    List<Integer> failsEachRack ← ∅
6:    for each i in Servers do
7:       rackID ← getRackID(i)
8:       if |CRM_{i,t_now}| ≥ ⌊(K+1)/2⌋ then
9:          S_{i,t_now} ← suspect
10:      else
11:         S_{i,t_now} ← trust
12:         if only one CRM coming from other rack then
13:            failsEachRack.get(rackID)++
14:         end if
15:      end if
16:   end for
17:   for each i in ToRs do
18:      if failsEachRack.get(i) ≥ α · m then
19:         S_{i,t_now} ← suspect
20:      else
21:         S_{i,t_now} ← trust
22:      end if
23:   end for
24: end while
```

determines the status of each server (line 6 to 16), and then determines the status of ToRs (line 17 to 23). When detecting server failures, **Crash Decider** also records necessary information to find potential ToR crashes (line 12 to 14).

5.5 Failure Detection for Links

The failure detection for non-imperative devices is another important part of our detection system. Failures of routers and switches can be easily detected by checking traffic dump on them. So in this subsection, we will focus on detection for links. We firstly refine the following QoS:

- **Precision.** Precision indicates how many suspected links are true failed.
- **Completeness.** This measures whether our approach can find all the faulty links.
- **Discrimination.** Because we will provide a short ranked list of suspected objects, this measures to what degree can our approach distinguish between normal links and faulty links.

The key idea of our approach is to capture network traffic changes over time. If the network traffic changes significantly compared with traffic in normal conditions, we believe that there exists failures somewhere in the network. The insight is that network traffic distribution is associated with routing protocol and monitoring relationships of servers, the drastic changes of traffic probably imply failures. For example, if the network topology is just as Figure 1 shows, when

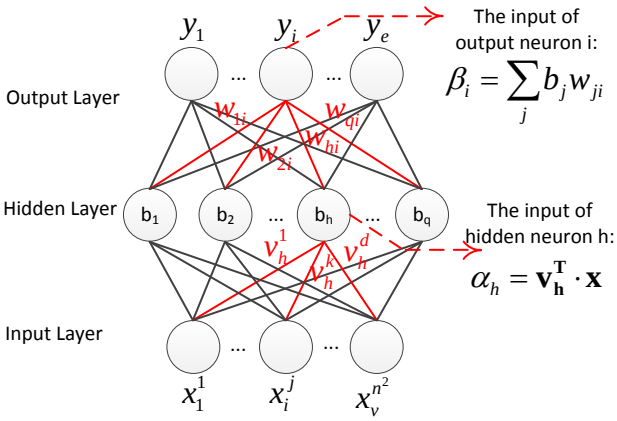

Figure 3: Multi-layer Perceptron Architecture

the link between the leftmost AR and the leftmost AS fails, the heartbeat traffic on the leftmost AR will decrease and the rightmost AR's heartbeat traffic will increase according to *ECMP* routing.

To collect required traffic data, we simply install *tcpdump* on each router and switch. These data will be collected to **Traffic Analysis Worker** for further analysis.

In order to model this data analysis problem, this paper proposes a Back Propagation Neural Networks (BPNN) based method to compute failure probability of each link.

Feature Extraction

The first step is to extract rational features from raw traffic data. To address this issue, we extract $v \cdot n^2$ features in total. v is number of devices (CRs, ARs, ASs), each device has n^2 features where n is the number of racks. In particular, each feature denotes traffic of one direction. Obviously, there are n choices of sources and n choices of destinations, so the total number of possible traffic directions is n^2. Formally, the traffic features during a time period ΔT is presented as $\boldsymbol{x} = (\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_v})$, $\boldsymbol{x_i} = (x_i^1, x_i^2, ..., x_i^{n^2})$ where x_i^j stands for direction j traffic of device i in ΔT.

BPNN Model

Suppose the number of links is e, for each time period ΔT, the output of our model should be $\boldsymbol{y} = (y_1, y_2, ..., y_e)$, where y_i denotes failure probability of link i. We apply a BPNN model to localize the faulty link during ΔT. As is shown in Figure 3, BPNN is a multi-layer perceptron architecture: To reduce the computational work of training, we only set one hidden layer, both the hidden layer neurons and output layer neurons utilize Sigmoid function as the activation function. Take the output layer as an example: $y_i = Sigmoid(\beta_i - \theta_i)$ where θ_i is a constant term of the perceptron.

Training of BPNN

To train the neural network, we apply the classical error BackPropagation (BP) algorithm [18]. BP algorithm is based on gradient descent strategy.

To get train data, we manually disable links to simulate real link failures. To be specific, each time we disable one link for some time. After that, we collect dump data from each router and switch, extracting direction features just

as mentioned above. The next step is to split the traffic data into different chunks, with each chunk representing the traffic data within one time unit (ΔT). Each traffic chunk forms one train instance.

The Detection Algorithm

BPNN model is only able to identify the faulty link during one time unit, in the following part, we will introduce how we can detect multiple faults over time.

First of all, we assume that there are at most one link failure during a time unit (ΔT). This assumption is partially supported by the finding proposed by Gill et al. They find that on an average, link failures tend to be separated by a period of about one week [6].

Algorithm 2 Traffic Analysis Worker Algorithm

Input: $\boldsymbol{x}^t = (\boldsymbol{x}_1^t, \boldsymbol{x}_2^t, ..., \boldsymbol{x}_v^t)$
Output: F_i for $i \in E$
1: divide traffic data $\boldsymbol{x}^t$ into $\lfloor t/\Delta T \rfloor$ chunks
2: **for** $i \in E$ **do**
3: $F_i \leftarrow 0$
4: **end for**
5: **for** each data chunk $\boldsymbol{x}$ **do**
6: $\boldsymbol{y} \leftarrow BPNN(\boldsymbol{x})$
7: **for** $i \in E$ **do**
8: $F_i \leftarrow F_i + y_i$
9: **end for**
10: **end for**
11: **for** $i \in E$ **do**
12: $F_i \leftarrow \frac{\Delta T \cdot F_i}{t}$
13: **end for**
14: sort $\boldsymbol{F}$ in descending order

Algorithm 2 shows the algorithm of **Traffic Analysis Worker**. The input of the algorithm is the refined time series traffic data over time t. Firstly, traffic data is divided into chunks. Each chunk represents a time unit ΔT of traffic (line 1). The last chunk will be dropped if its time period is less than ΔT. For each of traffic data chunk, we utilize the trained BPNN model to compute failure score of links during this time unit. The final failure score over t is computed as the arithmetic mean of failure scores of all time units (line 5 to 13). This solution inevitably benefits the early happened faulty links. For example, when there are only one faulty links in the network, the failure score of the sole faulty link will be very high, which in turn makes its final failure score higher. However, it does not matter because we care more about whether it can find more failures rather than the ranking.

6. EVALUATION

In this section, we will present how we evaluate our approach. Experimental results will also be shown in this section.

6.1 Experimental Settings

To build up a data center network, we deploy ns-3 on PC. ns-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use [10]. We install ns-3 on single PC with 2 cores, 3.4 GHz CPU and 4GB RAM. The version of ns-3 we use is ns-3.24.

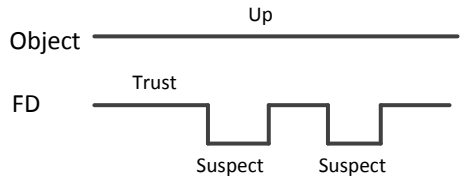

Figure 4: Average Mistake Rate

We set up a simulated data center network following the topology in Figure 1. There are a total of 120 servers (20 servers $\times$ 6 racks), 14 routers and switches and 140 links. For each link, data rate is 100 Mbps, delay is 100 ms.

6.2 Research Questions

The objective of our experiments is to validate the effectiveness of the proposed approach compared with baselines. For this purpose, we set up the following research questions:

RQ1: Can K-detectors model benefit server failure detection?

RQ1 evaluates QoS of failure detection for servers. To answer this question, we firstly set four metrics [4] corresponding to the QoS in Section 5.1.

- **Average Detection Time (AT_D)** is the average time that elapses from the crash of a server until the crash is reported.

$$AT_D = \frac{\sum_i (T_{D,i} - T_{C,i})}{n} \tag{8}$$

  where n is the number of detected crashes, $T_{D,i}$ is the detection time of ith crash i, $T_{C,i}$ is the crash time of ith crash.

- **Average Mistake Rate (AR_M)** measures the average rate at which a FD suspects a correct object. It equals the number of such mistakes per unit of time. For example, suppose Figure 4 depicts the monitoring of the object in one time unit, the mistake rate of this FD is two.

- **Query Accuracy Probability (P_A)** is the probability that the detection system outputs the correct results when queried at a random time.

- **Number of Undetected Crashes** measures the number of ignored real crash servers. This metric measures the completeness of the approach.

We compare our work with the traditional hierarchical architecture [5] in terms of the metrics above.

RQ2: Can K-detectors model benefit ToR failure detection?

The evaluation of failure detection for ToRs is similar to servers. We will evaluate the impact of different number of server crashes, that is to say, we want to see whether server failures affect the QoS of ToR failure detection. The evaluation metrics are the same as servers.

RQ3: Is the BPNN based approach effective?

This RQ evaluates whether the proposed approach is more effective in detecting link failures than existing work. We make another three metrics:

- **Precision@k.** We make the use of this metric in order to evaluate to what degree the detection is accurate. This metric is appropriate because it does not take into account the positions in the list which caters to our requirements [15].

$$precision@k = \frac{1}{n}\sum_{i=1}^{n}\frac{fail_{i,k}}{k} \tag{9}$$

  where $fail_{i,k}$ denotes the number of true failures for failure event i in the top k returned results. n is the number of failure events, k equals to the number of true failures. A perfect result is all the k failures are ranked at the top k positions.

- **MRR.** We also make use of Mean Reciprocal Rank (MRR), which is a statistic measure for evaluating a process that produces a ranked list to a sample of queries, the list contains items ordered by probabilities [17]. It is the average multiplicative inverse of the rank of the first faulty link:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \tag{10}$$

- **Discriminative Significance** (DS). DS is based on the notion of entropy. Entropy is a measure of unpredictability of information content [21].

$$DS = -\sum_{i=1}^{e} F_i \cdot log_2 F_i \tag{11}$$

  where F_i is a normalized failure score of link i. We leverage this metric to evaluate discrimination of different detection approaches. A smaller DS implies that the results of ranked list is more determinate, which makes it easier for further troubleshooting.

We will compare this part of our approach with the work of Herodotou et al. [11].

6.3 Experimental Results

RQ1: Server failure detection

We evaluate our approach in the presence of different degrees of message loss and server crashes. To be fair, FD parameters of all the methods are set to the same: $\tau_m = 0.001$, slide window size $n = 100$. Heartbeat message send interval η is 100 ms. The whole application is run for 1 min.

Figure 5 depicts P_A and AR_M of K-detectors and hierarchical architecture under different number of message loss without server crashes. Both the time and channel of message loss is chosen randomly. When loss rate is under 1%, 5-detectors has about an 8.2% improvement in P_A compared with hierarchical architecture. When loss rate is over 0.6%, 5-detectors makes more mistakes than others, this is probably because other methods make mistakes with relatively longer time. In a nutshell, with a larger parameter K, our approach behaves better in the presence of normal message loss. However, it cannot maintain with the increasing number of message loss.

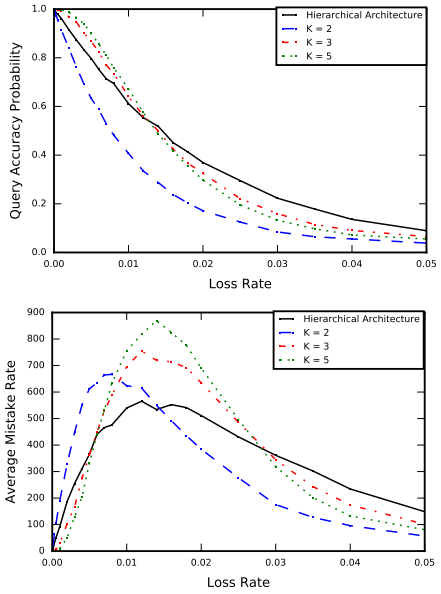

Figure 5: Effect of Message Loss on P_A and AM_R

Subsequently, we investigate it when there exists server crashes. We set message loss rate to 0.1%, both the time and server ID of crashes are selected randomly. As is shown in Figure 6, 5-detectors has a better P_A and AR_M than others in most cases. In terms of AD_T, a smaller K results in a quicker detection because it needs fewer FDs to reach an agreement. In general, AD_T of 5-detectors has a 22.4% increase, while 2-detectors only has a 7.3% increase compared with hierarchical architecture. Figure 7 shows the number of undetected crashes, hierarchical architecture tends to behave worse than K-detectors, which reveals the *Single Point of Failure* problem of hierarchical architecture.

RQ2: ToR failure Detection

To answer this question, we compute the metrics of ToRs failure detection in the presence of different server crashes. On top of this, we randomly make three ToR failure events with failure group size equals to 1, 2 and 3 respectively. In particular, α is set to 0.9, 0.6 and 0.4 when failure group size is 1, 2 and 3 respectively.

AR_M is all 0 for all the failure events. The P_A and AD_T are shown in Figure 8. Roughly speaking, size of server crashes has little impact on QoS of ToR failure detection. With the increasing of ToR crashes number, AD_T is better due to a smaller α, which means **Crash Decider** needs fewer crash report messages to determine the failure of a ToR.

The main drawback of our approach is that once occurring ToR failures, we cannot detect server crashes within the rack before the ToR recovers.

RQ3: Link failure detection

To answer this question, we wants to firstly explain why we select BPNN rather other popular classifiers. We select 3 other algorithms including Decision Tree (C4.5) [16], Bayesian Networks (BN) [12] and Support Vector Machine

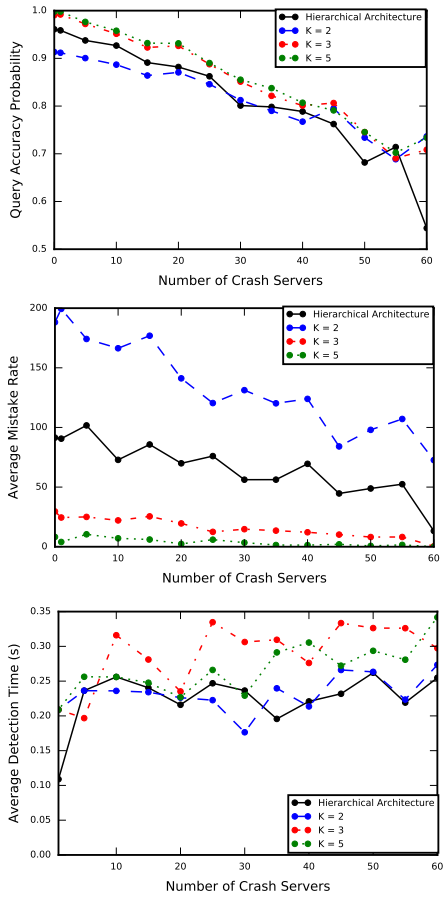

Figure 6: Effect of Crashes on P_A, AR_M and AT_D

Table 1: Results of Different Classifiers

Classifier	Precision	Recall	F-Measure
BPNN	98.8%	98.8%	98.8%
C4.5	96.4%	96.3%	96.3%
BN	96.9%	96.6%	96.7%
SVM	96.9%	96.6%	96.7%

(SVM) [20]. We apply these algorithms to train data by 10-fold cross-validation. Results are all shown in Table 1. Obviously BPNN outperforms other algorithms.

Empirically, parameters for BPNN is set as follows: learning rate equals 0.3, number of iterations equals 500, momentum equals 0.2, number of neurons at the hidden layer is $(\#attributes + \#classes)/2$, $\Delta T = 1$ s. The selection of ΔT is a tradeoff, when ΔT grows bigger, it may contain multiple failures which violates our assumption, while a smaller ΔT may add more occasionality to traffic data.

We manually make some link failures by disabling links in ns-3, we make 5 random failure events for different size of failure group respectively. Both the link ID and failure time are selected randomly. As Gill et al. finds that link failures tend to be isolated [6]. Only 41% of the link failure groups contain more than one faults, only 10% contain more than four faults. Consequently we only make failure events whose failure group size is no more than four.

Figure 9, 10 and 11 show the results of out experiments.

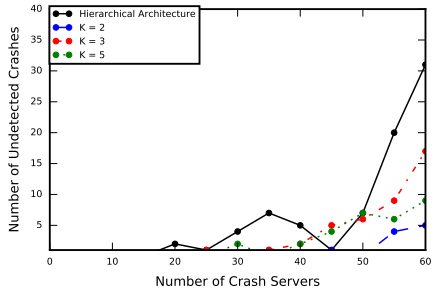

Figure 7: Number of Undetected Crash Servers

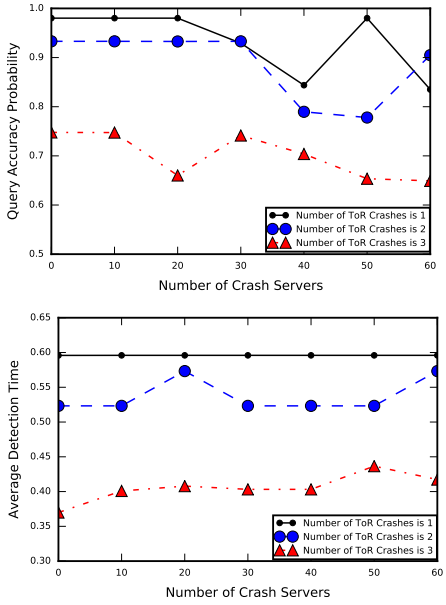

Figure 8: Effect of Server Crashes on P_A and AT_D of ToRs

Our approach has a better performance in terms of percision@k and MRR. In particular, there is an at least 73.3% precision@k performance, which indicates 73.3% of the failures can be found at the most top of the suspects averagely.

On the other hand, there is a huge improvement (nearly 12-folds improvement) in terms of DS. The approach proposed by Herodotou et al. tends to assign a wider range of failure scores to multiple links, while our approach is better at finding the relevant links. It mainly stems from the fact that our approach is based on a multi-layer perceptron architecture, it can precisely capture the characteristic of each link failure. But one of the drawbacks of our approach is that it requires much data and time to train BPNN offline to ensure its generalization ability, it cannot adapt to network topology changes either.

7. CONCLUSIONS

Failure detection plays a very important role in maintaining the reliability of data center networks. In this paper, we firstly identify the differences in QoS of failure detection for different network devices. We leverage K-detectors model to enhance the timeliness, accuracy and soundness of fail-

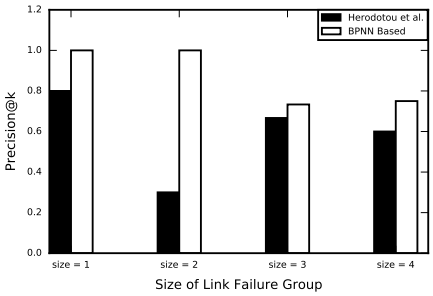

Figure 9: Precision@k of Different Failure Group Size

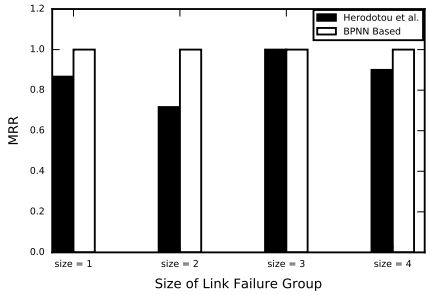

Figure 10: MRR of Different Failure Group Size

ure detection for imperative devices. We leverage a data mining based approach to produce a ranked list of the most suspect non-imperative devices by capturing network traffic features. We have implemented our approach as well as baselines on a simulated data center network built by ns-3. Our experimental evaluation has validated the effectiveness of our approach.

8. ACKNOWLEDGMENTS

This work is partially supported by the National High-Technology Research and Development Program of China under Grant No.2014AA01A301.

9. REFERENCES

[1] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 13–24. ACM, 2007.

[2] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 354–363. IEEE, 2002.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[4] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on computers*, 51(5):561–580, 2002.

[5] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Distributed Objects and Applications, 1999. Proceedings of the International Symposium on*, pages 132–141. IEEE, 1999.

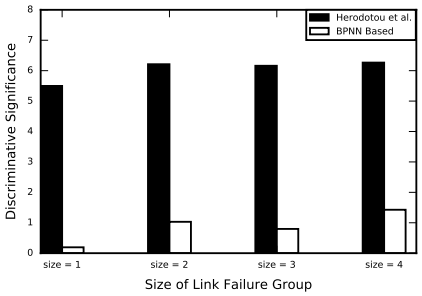

Figure 11: DS of Different Failure Group Size

[6] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 350–361. ACM, 2011.

[7] M. Gillen, K. Rohloff, P. Manghwani, and R. Schantz. Scalable, adaptive, time-bounded node failure detection. In *High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE*, pages 179–186. IEEE, 2007.

[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, volume 39, pages 51–62. ACM, 2009.

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

[10] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 15:17, 2008.

[11] H. Herodotou, B. Ding, S. Balakrishnan, G. Outhred, and P. Fitter. Scalable near real-time failure localization of data center networks. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1689–1698. ACM, 2014.

[12] F. V. Jensen. *An introduction to Bayesian networks*, volume 210. UCL press London, 1996.

[13] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A tool for failure diagnosis in ip networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 173–178. ACM, 2005.

[14] Y. Liu, J. K. Muppala, M. Veeraraghavan, D. Lin, and M. Hamdi. *Data Center Networks - Topologies, Architectures and Fault-Tolerance Characteristics*. Springer Briefs in Computer Science. Springer, 2013.

[15] I. Mogotsi. Christopher d. manning, prabhakar raghavan, and hinrich schütze: Introduction to information retrieval. *Information Retrieval*, 13(2):192–195, 2010.

[16] J. R. Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[17] D. R. Radev, H. Qi, H. Wu, and W. Fan. Evaluating web-based question answering systems. *Ann Arbor*, 1001:48109, 2002.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[19] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 551–555. ACM, 2007.

[20] B. Schölkopf and C. J. Burges. *Advances in kernel methods: support vector learning*. MIT press, 1999.

[21] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[22] A. Tomsic, P. Sens, J. Garcia, L. Arantes, and J. Sopena. 2w-fd: A failure detector algorithm with qos. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 885–893. IEEE, 2015.

[23] R. Van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Middlewareąf98*, pages 55–70. Springer, 1998.

[24] L. Xu, W. Chen, Z. Wang, H. Ni, and J. Wu. Smart ring: A model of node failure detection in high available cloud data center. In *IFIP International Conference on Network and Parallel Computing*, pages 279–288. Springer, 2012.