Study on High Availability and Fault Tolerance application

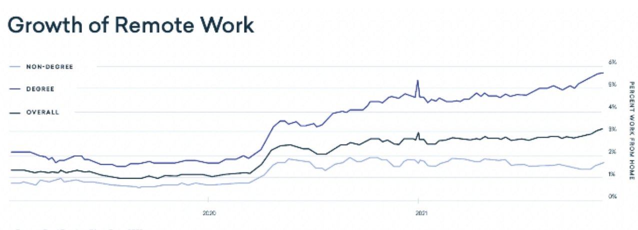
1st Norman Kong Koon Kit
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
kong.ko@northeastern.edu

2nd Michal Aibin
Khoury College of Computer Sciences
Northeastern University
Vancouver, BC, Canada
m.aibin@northeastern.edu

***Abstract*—Availability is one of the most important requirements for modern computing systems. This defined the maximum acceptable outage time for any given period. In cloud computing, it is common to use this metrics as a key factor in their decision to adopt a cloud service. This paper studies the breakdown in calculating the availability and proposes a conceptual model as a middleware in order to speed up the time for detection during system crash. Through simulations tests, we verified that the proposed model is able to detect the system crash in sub-seconds and improve the overall availability of the system.**

I. Introduction

Remote working and e-commerce have become the new normal since the pandemic [3, 10], it have changed our daily lives and the way how we consume services. With the growth of these online services, there is a high demand on non-functional requirements, especially in the domain of Availability and Fault Tolerance.



Availability [11] is considered as one of the main challenges of modern computing, especially for cloud computing since it can provide dynamic elasticity on computing resource. Therefore, as more missing critical applications are shifting towards this architectural pattern, availability has been considered as one of the major concerns on shifting to cloud platform.

High Availability defines the maximum allowed outage time, which usually refers to ”5-nines” of up-time, i.e. less than 6 minutes downtime annually. Thus, this is considered as one of the major challenges of on-premise or cloud computing as most of the critical services are migrating to this architecture [2].

In Fault Tolerance, this means that the system will operate continuously even some of the components fails. Instead of failing completely, application are designed to operate at downgraded service, while system recovery will be took place and try to resume the service automatically.

This paper will examine the current service availability and fault tolerance in cloud providers’ practices, together with other proposed in research papers and industries’s best practice. Then we will try to propose a system design that aim to improve the service availability and fault tolerance with a sample implementation.

The organisation of this paper is as follows: In Section 2, we will provide the Related Work with various definitions of availability and fault tolerance used. We will introduce the current challenge that faced in Section 3. In Section 4, we will propose and implement the solution using a sample application performing the evaluation. Evaluation and assessment against the metrics will be conducted in Section 5. In Section 6, we will summarise and conclude the pros and cons of current solutions before elaborating on future research directions.

II. Related Works

Understanding what availability is a prerequisite for the evaluation. The term ”Availability” was defined as ”the degree to which a system is functioning and is accessible to deliver its services during a given time interval.” [6]. This is the percentage of time that allows outage time for a given period. Some researchers further defined availability in the form of Service Level Agreements[5], and considered availability as the ”probability of providing service according to defined requirements”.

This term has been extensively used by service providers, especially cloud service providers [4]. In case any hiccups cannot fulfill this commitment, service providers will even issue credit for compensation[1]. Therefore, a lot of research effort has been dedicated to guaranteeing availability in the cloud [8].

In the domain of cloud computing, researcher [13] has proposed the cloud availability taxonomy using three criteria:

Availability mechanisms, Failures protected against, and Metrics, which categorized all the redundancy models accordingly Availability Management Framework [9].

We define "High Availability" as a service that is available at least 99.999% of the time [6]. For example, the maximum downtime under this scenario for service annually is only 5 minutes and 15 seconds.

In 2010, researcher [14] proposed a middleware "Low Latency Fault Tolerance" that uses different protocols for membership, messaging, and synchronization to achieve fault tolerance at the application level. This middleware provides fault tolerance for distributed applications using the leader/follower replication approach.

In 2011, another researcher [13] proposed a "heartbeat" approach to detect application failures. When there is no heartbeat received within a grace period, it is assumed that the application has failed and fail-over/recovery action will be triggered. For instance, the AWS load balancer monitors [12] the application status in such a way, but they do not protect against application failures. Although this mechanism does not detect all possible application failures, it is the most generic way as the application team is only required to provide an HTTP endpoint. However, this heartbeat technique is not an effective solution since it is a passive pull-based mechanism that takes more than 10 seconds to detect the failure.

With the rise of microservice architecture, some researchers have begun to use Docker / Kubernetes to achieve better availability by shortening the reaction time to detect fault events [14].

III. Problem Statement

Availability is one of the important criteria in modern application. In order to improve the availability for better service level, we begin to perform reliability and availability analysis to characterize the problem.

In layman definition, the availability can be expressed by a mathematical equation :

$$Availability = \frac{uptime}{uptime + downtime} * 100\%$$

The number higher will have a better availability, which provides better service level agreement.

To further understand the problem, we can further define Reliability as the probability that a system continues to work as expected over a specified time duration. Thus, the terminology MTBF (Mean Time Between Failures) is used to represent the average time to the next failure, and is calculated as:

$$MTBF = \frac{Service\ Running\ Duration}{Number\ of\ Incidents}$$

This MTBF is not only used in Computer Science, it is also widely used as an important maintenance metric to measure performance, safety, and equipment design for mission critical systems [7]. Another point to note that MTBF only consider unplanned maintenance time, while planned maintenance events like scheduled software updates are not included.

Another term related to availability is MTTR (Mean Time To Recovery) which is the total duration starting from system malfunction, system detection and system recovery, which is denoted as

$$MTTR = \frac{Mean\ Time\ to\ Recovery}{Number\ of\ Incidents}$$

Based on MTBF and MTTR definition, the availability of a system can be used to measure the impact of failures on an application, and the availability can be defined as:

$$Availability = \frac{MTBF}{MTBF\ +\ MTTR} * 100\%$$

where we can further break down MTTR as

$$MTTR = MTD + MTF$$

where

- MTD = Mean Time to Detect
- MTF = Mean Time to Fix

Hence, we can express availability as a function of :

$$Availability = \frac{MTBF}{MTBF\ +\ MTD\ +\ MTF} * 100\%$$

Therefore, by the formula above, we can improve the Availability by either :

- Increase Service Up-time (maximize MTBF) or
- Decrease Mean Time to Detect (minimize MTD) or
- Decrease Mean Time to Fix (minimize MTF)

Optimization problem

The primary objective is to maximize the Availability of a system in order to achieve higher Service Level Agreements (which means more ninth). Thus, we can either perform :

1) Maximize the MTBF: There are a number of ways to increase the MTBF. For example, keeping track of the MTBF for each sub-system, especially those long run operation processes. Perform machine learning/data mining technique to identify any usage pattern that will lead to system failure, then arrange corresponding fixes during planned maintenance. While this can improve the MTBF and result in better availability, this paper will focus on how to minimize the denominator, i.e. to minimize MTTF in order to improve the overall availability.

2) Minimize Time to Detect (MTD): By Murphy's Law : "Anything that can go wrong will go wrong". While we could not completely eliminate the chances for system failures, we should try to minimize the time to detect the error such that we can discover the problem earlier in order to shorten the total time of outage. Then the overall availability will be increased.

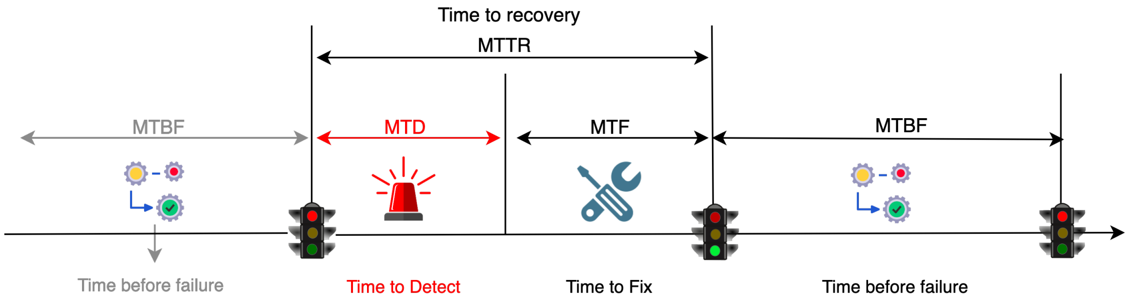


Fig. 1. Illustration on the relationship between various status

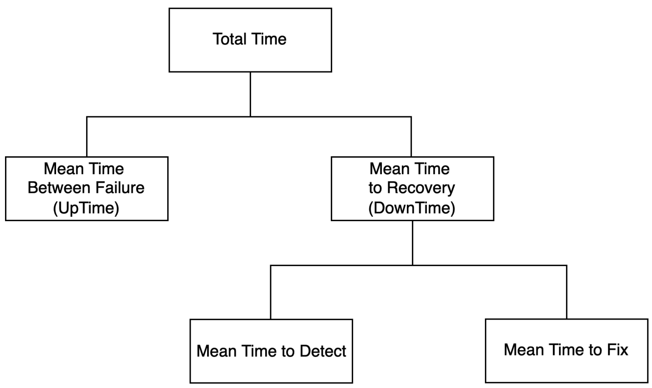


Fig. 2. Classification of Time component

3) Minimize Mean Time to Fix (MTF): By standardizing the procedure to restart / repair or replace the faulty component, the time to fix the problem can be reduced. On the other hand, by reserving additional spare component as redundancy, the time to repair can be minimal, providing that the failure cause is not due to logically fault. In this paper, the application we implemented will makes use of Docker and Publish/Subscribe mechanism to speed up the recovery.

IV. THE ALGORITHM

As we have discussed in the previous section, we will focus on minimising the "Time to Detect" and "Time to Fix". The industry practice is to use pull-based health checks, i.e. load balancer keeps sending probe requests to each worker node every interval (t_h) . In case the worker node is unavailable immediately after the last health check, then the particular worker node will become unavailable until the load balancer detects it in the next health check $2t_h$. To make things even worse, the system will send additional retry health checks which means that the outage time can be up to $3t_h$. Therefore, in this section, we propose an push-based health-check mechanism such that system can detect node availability as earlier as possible.

Before we describe the algorithm, we introduces the below components:

A. Resource Agent

This is an embedded agent running on the worker load. When the worker node startup, it will register a "Alive" signal to the Load Agent. Once connected, it repeatedly sends a "Status Update" event which includes the CPU/Memory/Storage/Active thread count to the Load Agent and expects to receive an acknowledgement with "Time To Repost" (TTR). Then the agent will wait for this TTR and send the updated resource information again.

B. Load Agent

This is a standalone agent running independently from the service application. This daemon starts a socket and waits for connections from Resource Agent. Then it will keep a persistent session which expect to receive "Status Update" from Resource Agent. Then, it will update the status of Resource Agent into a distributed cache such that Load Balancer will use this information for client request dispatching. Since the connection is a socket connection, in case the connection is lost, this persistent connection will be terminated and the Load Agent will be notified immediately, then it will detach this worker node from the cache, such that Load Balancer will not dispatch client request to this crashed node accordingly. In order to enhance the fast recovery, this agent will also publish a recovery message such that Recovery Agent will perform recovery action accordingly.

C. Load Balancer

This is the client facing endpoint that it accepts external request. Then we implement a Resource Awareness routing algorithm to select a worker node, then dispatch the request to this node to handle. In case there is no node available, we will report outage and advise client to retry later.

D. Recovery Agent

This is another standalone agent work like Kubernetes master node to perform worker node life cycle management, including node creation and eviction. This agent subscribe to topic that Load Agent will publish recovery request in case worker any worker node was crashed.

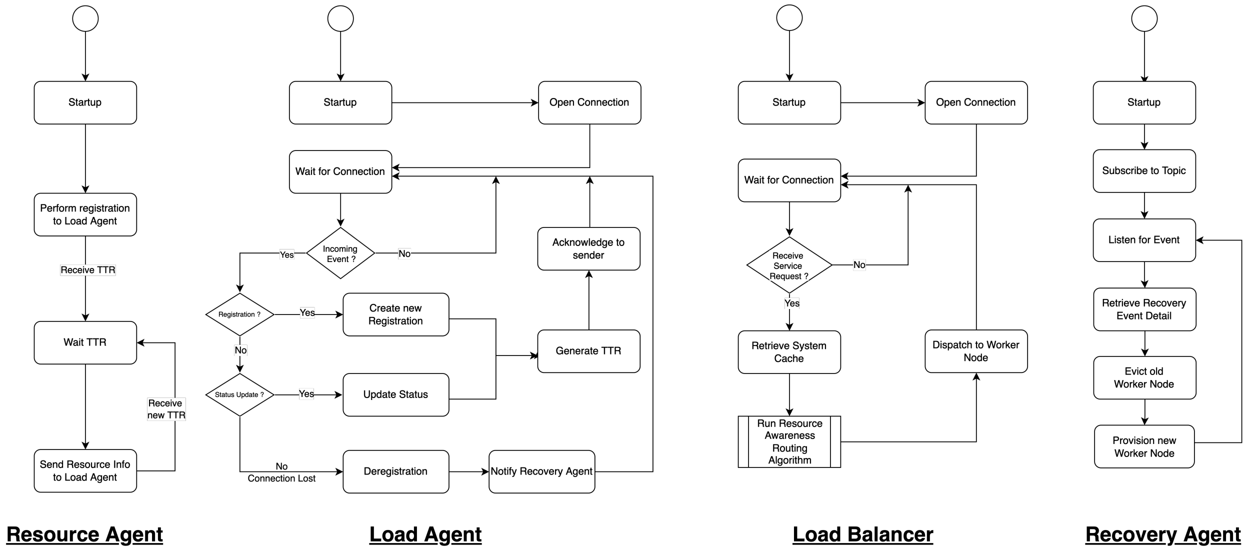


Fig. 3. High Level algorithm for each individual component

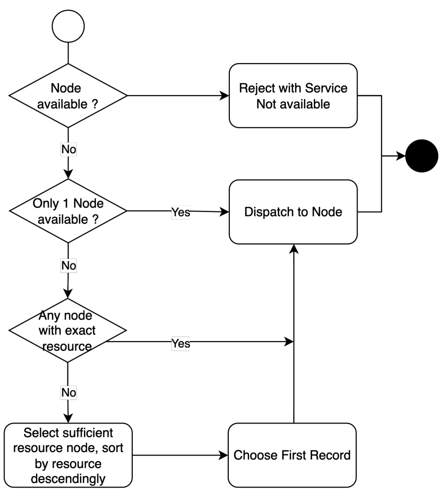


Fig. 4. Resource awareness routing algorithm

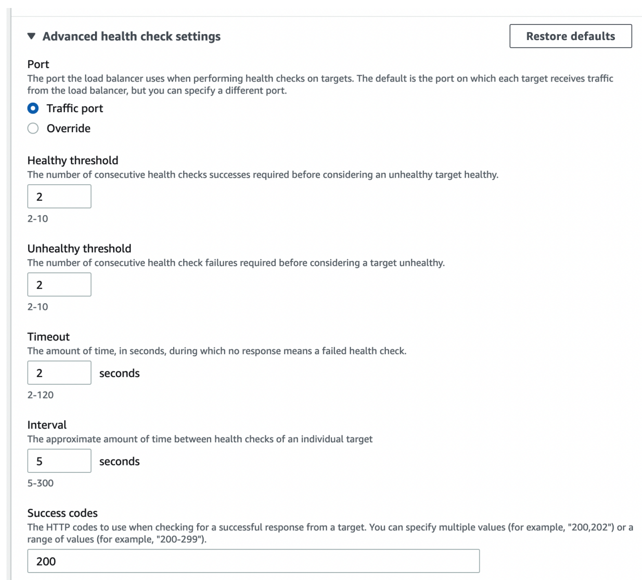


Fig. 5. AWS Application Load Balancer is using a pull based with minimum 2 seconds interval

Resource Awareness Routing Algorithm

This algorithm is a greedy algorithm that it will select nodes that match several criteria. The highest priority will be finding worker node that available resource that is ”Just Fit”, which means after servicing this new request, this node will be saturated. The rationale is that we would like to reduce the resource fragmentation across different worker nodes. In case we can fill up the capacity for one particular node, then we can keep other resourceful node for future demanding request. In the meanwhile, we proposed a thread level checking to ensure that the number of concurrent thread cannot be greater than the average, such that no worker node will be too heavily overload while some node is deep under-loaded. However, this is subjected to the programming language and the system capability.

Justification

The proposed methodology is considerably to enhance Availability since it uses persistent connection with push-based health check. In the traditional pull-based health check by Load Balancer, each probe requires a new TCP connection such that it will spend extra time on TCP handshaking. By using persistent connection, the Load Agent will receive ”Session Disconnect” signal immediately when any worker node crashes. Therefore, the ”Time to Detect” is expected to be much shorter than the pull based mechanism. However, the drawback is persistence push based mechanism will consume much more system resource than the pull based, so this

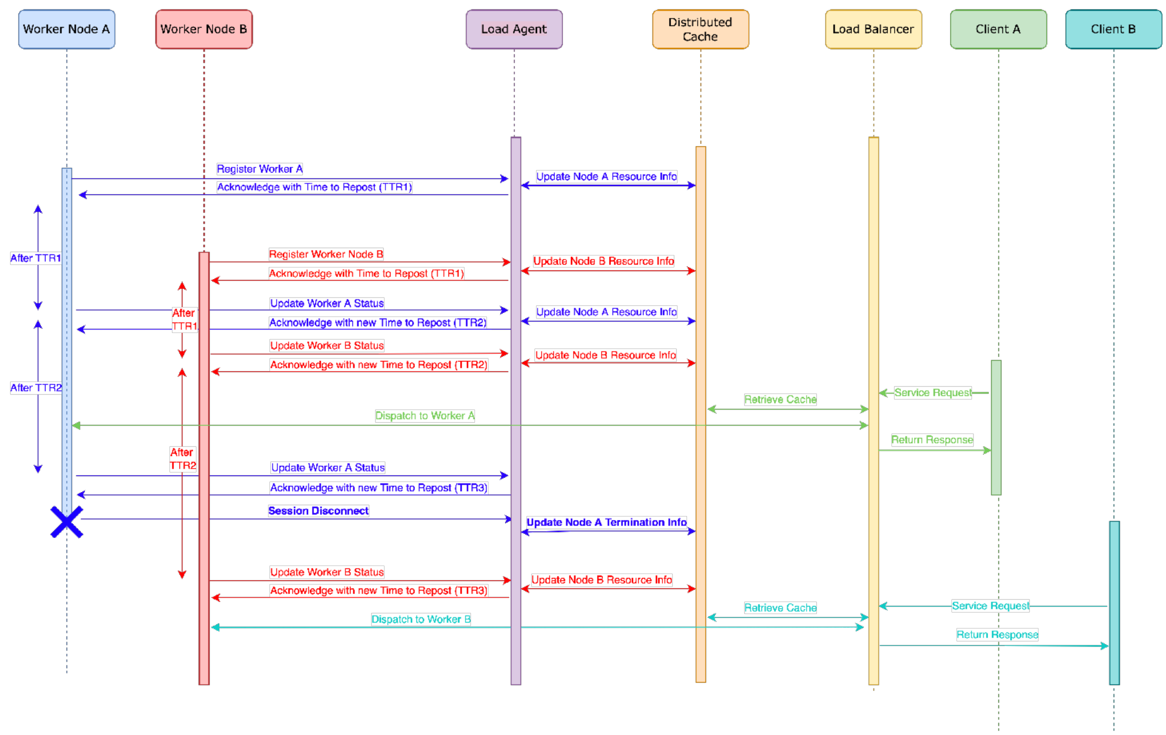


Fig. 6. Overall flow diagram to illustrate the fault event handling, recovery agent was not in scope

methodology should be used on mission critical or computing / networking resource is abundance application.

Figure 6 illustrated a high level flow diagram that when Worker Node A disconnects, Load Agent will receive "Session Disconnection" signal, then it will immediately update the system cache, so the subsequent client request will not route to this faulty node accordingly. In the meanwhile, the Load Agent will also trigger a "Recovery" message to Recovery Agent to perform the worker node eviction and re-provision.

In the next section, we will implement a sample application using this stack and compare the Amazon Application Balancer accordingly.

V. Simulation Plan

The proposed architecture was constructed, as shown in Figure 7, and the existing architecture was typically used by Cloud service providers in Figure 8.

For evaluation purposes, we have implemented the entire application using nodejs. Socket.io was chosen as the persistent connection middleware between Resource Agent and Load agent. Redis was adopted as the cached layer and also Redis pub/sub were adopted for the communication between Load Agent and Recovery Agent.

In order to demonstrate the Resource Agent middleware, various workload applications that embedded this agent were built and further dockerized as a docker image, such that the Recovery Agent can perform recovery by using docker command. Since we are trying to measure the performance of fault detection, we embedded a "Kill Switch" that will trigger the application to "crash" after providing service for a particular time.

Finally, ELK stack was selected as the log management layer such that we can capture all the status log centrally.

In this paper, we will analyse the performance of the existing cloud provider fault-detection method and the proposed fault-detection method via experiments. Subsequently, we compared how the performance varies depending on the change in workload and resource usage. We will setup 10 thread to run 1,000,000 iterations per scenario.

We will perform the simulation using a different workload against kill switch frequency.

#	Type of Workload
1	Simple Web Workload
2	I/O Bound Workload
3	CPU Intensive Workload

TABLE I
WORKLOAD VARIATION

#	Kill Switch Frequency
1	5 minutes of execution
2	10 minutes of execution
3	20 minutes of execution

TABLE II
KILL SWITCH VARIATION

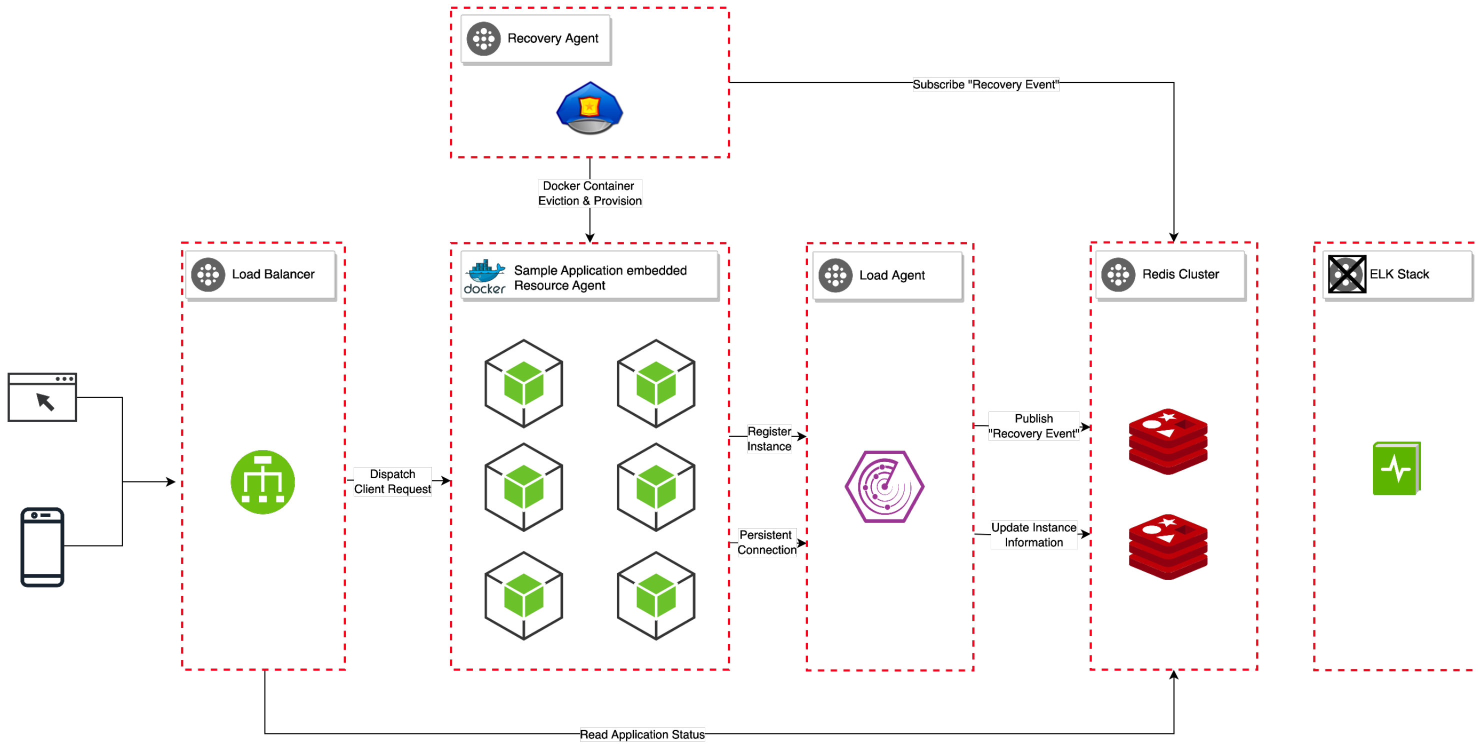


Fig. 7. Evaluation Implementation

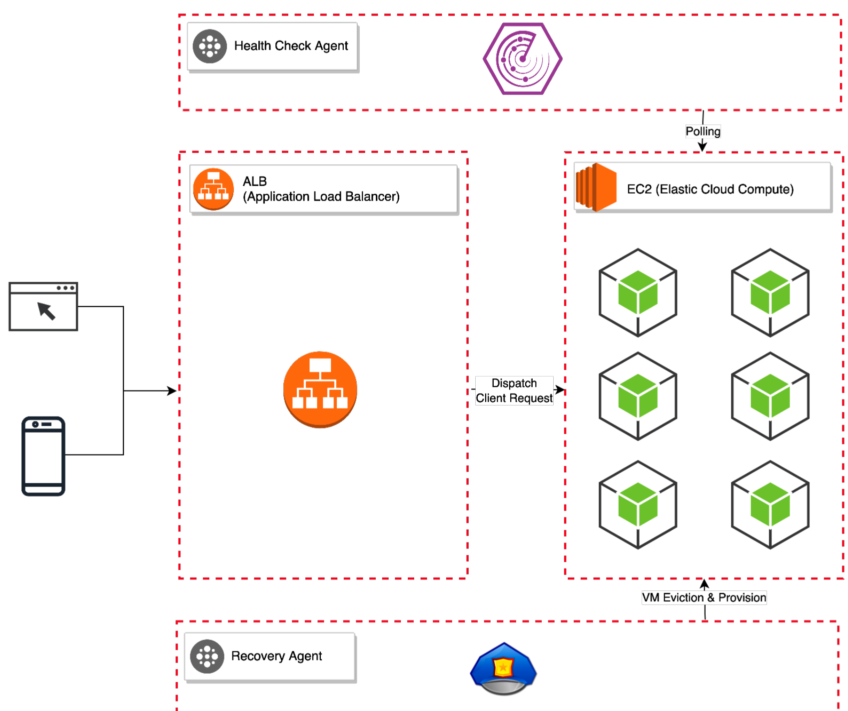


Fig. 8. AWS Load Balancer Logically Implementation

#	Kill Switch Frequency
1	2 seconds Connection Timeout
2	5 seconds Health Check interval
3	2 Unhealthy threshold

TABLE III
AWS LOAD BALANCER OPTIMIZED SETTING

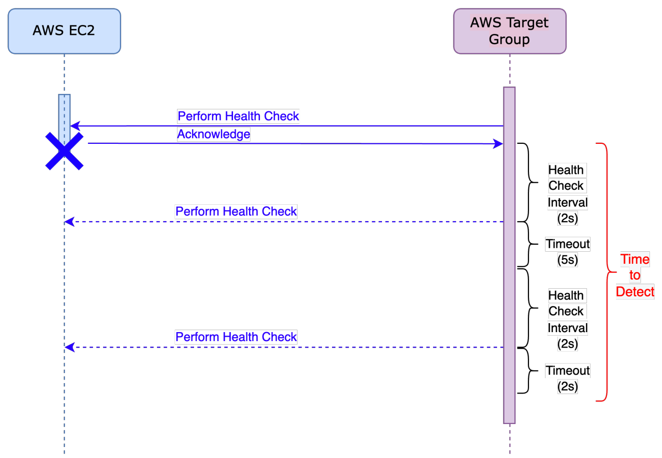


Fig. 9. AWS Load Balancer Worst Case scenario

#	Duration(s)	Up(s)	Down(s)	SLA(%)	Fail Cnt
AWS - 5	2,105.901	2,020.234	85.667	95.93204%	2,080
AWS - 10	2,203.670	2,131.404	72.266	96.72067%	1,518
AWS - 20	2,089.374	2,048.838	40.536	98.05992%	1,021
Pro - 5	2,943.944	2,936.602	7.343	99.75058%	3,990
Pro - 10	2,970.906	2,970.883	0.023	99.99923%	600
Pro - 20	3,005.539	3,004.165	1.374	99.95429%	308

TABLE IV
DETAIL BREAKDOWN OF THE SIMULATION

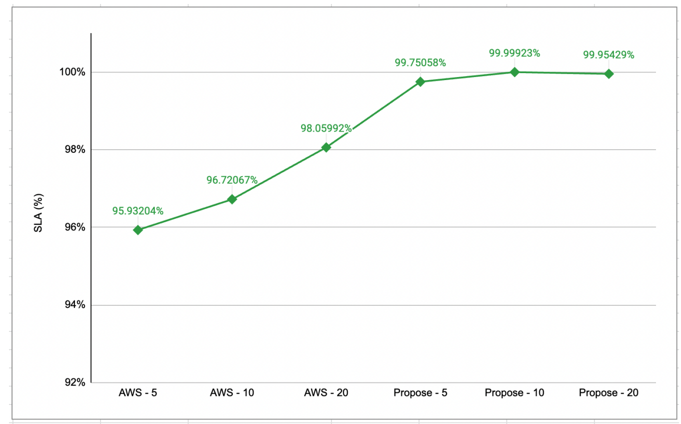


Fig. 10. Service Level Agreement with different scenario

VI. ANALYSIS

A. Analysis : Mean Time To Detect analysis

We calculated the average time to detect a system crash is around 24.5 millisecond. The data is consistent across lightweight web service tasks and I/O bound tasks. The reason is that these jobs did not affect our proposed algorithm since the Kill-Switch mechanism is running independently in nodeJS event-loop. However, the CPU bound jobs were greatly affected since nodeJS is a single thread programming model, which means that only one task can be executed at a time. Our implementation tried to compute the Fibonacci series so the JavaScript engine would execute the computation solely so our resource agent plugin was "paused" during execution. Therefore, we have dropped out the CPU bound jobs testing after several trial runs.

Compared with AWS application load balancer, the theoretical "time to detect" is 14 seconds. This figure is calculated based on the AWS Target Group Health Check setting, the minimum. The minimum unhealthy threshold is 2, while each health check interval takes 5 seconds with 2 seconds timeout. As illustrated in figure 9, the worst case mean time to detect is (5+2)* 2 = 14 seconds. Therefore, our proposed approach (24.5 ms) is 570 times faster.

B. Analysis : Overall Service Level Agreement

Instead of focusing on the Mean time to detect, we focused on the effectiveness of the overall service level agreement.

In our proposed solution, as we run in a local environment, we created 10 threads that triggered 100,000 requests individually, so the total execution is 1 millions requests for each kill switch iteration. The total execution time for each iteration is around 50 minutes.

We observed the number of fail count was 3,990 when kill-switch is 5 minutes, although this fail count is the highest amongst all scenarios, the downtime was only 7.343s so the overall SLA is 99.750%. When the kill switch increases to 10 minutes, the downtime becomes 0.023s only so the overall is 99.999% (five-ninth). This indicates that we can achieve "High Availability" even if our applications are scheduled to crash every 10 minutes. When the kill switch increased to 20 minutes, the SLA slightly dropped to 99.954%. All proposed scenarios have outperformed the AWS Application Load Balancer approach as described below.

In contrast, we have set up the AWS Application Load Balancer with fastest fault recovery. Since the stress test is running on a remote environment, we created 10 threads that only triggered 10,000 requests individually, so total execution is 100,000 requests. The total execution time for each iteration is around 35 minutes. As expected, we observed that the downtime decreased while the kill switch increased. The SLA was initially 95.93% when the kill switch was 5 minutes, then kept increasing to 96.72% and finally reached 98.06% when the kill switch was 20 minutes. The number of fail requests decreased by half from 2080 to 1021 during the 100,000 iteration.

Note : Although our proposed SLA outperformed the AWS current approach, the comparison did not take into account that AWS is using Virtual Machine while our proposed solution is based on Docker. As we are using Docker, the "Mean time to recover" is much faster than AWS since the Docker provisioning is much light-weight than virtual machines. Therefore, the comparison can only be used as a reference point only.

VII. Conclusion

High availability has been one of the biggest challenges in application design. There are various techniques that can be used to improve the availability of a service which depends on various use cases. This research paper proposes to use a push-based mechanism with persistent connection in order to reduce the "Mean Time to Detect" such that the overall Service Level Agreement can be improved.

VIII. Acknowledgment

I would like to express my very great appreciation to Peter Smith for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time so generously has been very much appreciated.

References

[1] *Amazon Compute Service Level Agreement*. en-US. URL: https://aws.amazon.com/compute/sla/ (visited on 08/15/2022).

[2] *AWS Application Migration Service (MGN) Service Level Agreement*. en-US. URL: https://aws.amazon.com/application-migration-service/sla/ (visited on 08/15/2022).

[3] *E-commerce in the time of COVID-19*. en. URL: https://www.oecd.org/coronavirus/policy-responses/e-commerce-in-the-time-of-covid-19-3a2b78e8/ (visited on 08/15/2022).

[4] *Google Cloud Platform Service Level Agreements*. en. URL: https://cloud.google.com/terms/sla (visited on 08/15/2022).

[5] D Davide Lamanna, James Skene, and Wolfgang Emmerich. "Slang: A language for defining service level agreements". In: *Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, Proceedings*. IEEE Computer Soc. 2003, pp. 100–106.

[6] Mina Nabi, Maria Toeroe, and Ferhat Khendek. "Availability in the cloud: State of the art". In: *Journal of Network and Computer Applications* 60 (2016), pp. 54–67.

[7] Duane Pettit, Andrew Turnbull, and Henk A Roelant. *General aviation aircraft reliability study*. Tech. rep. 2001.

[8] Alan Robertson. "{Linux-HA} Heartbeat System Design". In: *4th Annual Linux Showcase & Conference (ALS 2000)*. 2000.

[9] Pejman Salehi. "A model based framework for service availability management". PhD thesis. Concordia University, 2012.

[10] *The Remote Work Report by GitLab: The Future of Work is Remote*. en. URL: https://about.gitlab.com/company/culture/all-remote/remote-work-report/ (visited on 08/15/2022).

[11] Astrid Undheim, Ameen Chilwan, and Poul Heegaard. "Differentiated availability in cloud computing SLAs". In: *2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE. 2011, pp. 129–136.

[12] Jinesh Varia, Sajee Mathew, et al. "Overview of amazon web services". In: *Amazon Web Services* 105 (2014).

[13] Hyunsik Yang and Younghan Kim. "Design and implementation of fast fault detection in cloud infrastructure for containerized IoT services". In: *Sensors* 20.16 (2020), p. 4592.

[14] Wenbing Zhao, PM Melliar-Smith, and Louise E Moser. "Fault tolerance middleware for cloud computing". In: *2010 IEEE 3rd International Conference on Cloud Computing*. IEEE. 2010, pp. 67–74.