Adaptive Timeout Strategies for Microservice Applications

Govind Velamoor, Adrita Samanta

MIT PRIMES 2024

Mentors: Lan (Max) Liu, Zhaoqi (Roy) Zhang, and Prof. Raja Sambasivan (Tufts University)

Abstract
Timeouts are critical in determining whether a request has succeeded or failed. Developers face several challenges when setting timeout values in distributed systems; the specific challenge we investigate being the systems' propensity to change, both over the short- and long-term. We propose a timeout-optimized policy targeting change over different time scales, assuming APIs that are both idempotent and atomic. We evaluate our approaches on a home-grown microservices testbed, by comparing the timeout percentage, total time taken, and closeness to actual latency when our approaches and the industry standard of Exponential Backoff are used in simulated environments with changing system performance.

1. Introduction

In the past, systems have traditionally been built in *monoliths*, with all of their components running on a single machine. With increasing speed and scale requirements, monolithic systems have faced several challenges. For one, the entire system must scale together, resulting in some components being over- or under-scaled. Monolithic applications are often hard to deploy because of their large size and are often tightly coupled, lack modularity, and have a limited development stack.

In the last two decades, organizations have been moving from using one monolith to many small independent applications called *microservices* [1] which communicate via *requests* to achieve more complex tasks. This architecture [2] allows for various performance and scaling benefits. Each individual service may be scaled to their own requirements rather than everything having to be scaled together. In addition, with distributed systems, work can be parallelized and spread over multiple nodes, improving performance.

Microservices commonly communicate with each other using request protocols such as RPC (Remote Procedure Call) and HTTP (Hypertext Transfer Protocol) which often result in long *tail latencies* [3]; for example in a system where the server usually responds in 10ms, the 99th percentile is often as high as 1 second. In such a case, timeouts are employed to determine whether a request is failed or simply taking longer than usual. After the client has waited a certain amount of time, the request times out, is deemed failed, and a retry is issued.

Correct timeouts are vital in distributed systems to determine whether information is flowing as intended. However, there has been little work so far on how to allocate them. Ultimately, poor timeouts result in additional latency for the user. Short timeouts will result in legitimate responses being prematurely abandoned by the system, causing unnecessary retries. Long timeouts will result in resources being allocated fruitlessly (requests that will never return a response would be kept alive for longer than necessary, taking resources away from other requests), slowing the system down for any requests which may actually provide responses. Thus, extensively long timeouts risk violating Service Level Agreements (SLAs) and in easier exploitation through Denial-of-Service attacks.

When individual services in a distributed system change, as they frequently do, it is important for every service to automatically update its timeouts to not cause user frustration or security vulnerabilities. In this paper, we investigate methods for assigning timeouts in dynamic (constantly changing) microservice applications.

2. Background

As we have seen, distributed systems offer many advantages to developers. However, with all the complexity they bring on the macroscopic scale comes a challenge: being able to debug and monitor the status of individual services. With developers frequently updating the implementation of their microservices, a request might take longer than usual. The central question we aim to answer is: if one service sends a request to another service but gets no response back in the expected time, what should it do?

Timeouts are a good strategy to diagnose whether a request has failed. There are, though, several challenges with setting timeouts in distributed systems. The simplest one is that each microservice's latency will vary from request to request. A commonly used structure is a cache, which results in fast response times for hits (when the request parameters are in the cache) and slow response times for misses (when the request parameters are not in the cache). In addition, multiples servers spread geographically can theoretically accept the same request, and different servers can have different response times. The primary issue we aim to address is that the implementations of individual microservices can also change in the long term. Setting timeouts in a once-and-done approach (hard coding them before ever running the system) is ineffective for dynamic systems where optimal timeouts may also change.

Thus, we aim to find methods to set timeouts in such a way that they can adapt to changes in the microservice application.

2.1. Prior Work

The following strategies are currently being used to reduce latency in distributed systems [3, 4]:

Setting better timeout values What is missing in the following options for setting timeout values is the capability for them to be updated as the distributed system is modified.

1. Retrying with constant time backoffs.
2. Retrying with exponential backoffs (with random jitter).
3. Using the greatest time response observed from a sample.

Request operation and retry policies The following options for manipulating how requests are handled are too general and do not necessarily align with the behavior of a particular service.

1. Tied requests: queuing the same request across multiple servers and allowing them to communicate results with each other. Once one server begins execution of a request, it is dequeued across all other servers.
2. Hedging: sending a second request when the first has reached 95% of max run time.
3. Disabling retries when the 99th percentile and 50th percentile are close together.
4. Continuously send requests until you receive a response
5. Retrying on 5xx (server-side) errors, but not 4xx (client-side) errors.

We propose retry policies that change timeout values dynamically, allowing us to have the generality and adaptiveness that comes with retry policies and the specificity that comes with setting accurate timeouts.

3. Algorithms

3.1. Assumptions

We assume that requests are both *atomic*, that they succeed fully or fail entirely, and *idempotent*, that they are repeatable: no matter the number of requests sent, the behavior is the same. In addition, we assume a *dynamic* system, one that is constantly changing. The following are examples of ways a system can change, resulting a change in latency distribution:

- Changes to server use
  - Using a different machine
  - Moving from using multiple servers to a single server, or vice versa
- Changes to service calls
  - Using different APIs
  - Calling new services
- Changing the programming language used
- Changing the hosting platform

Our major research goal was coming up with strategies to set timeouts in a dynamic system and to design an algorithm to determine the optimal timeout between any two microservices.

3.2. Timeout Principles

The algorithms we propose aim to satisfy the following principles:

Optimal An optimal timeout will result in the smallest feasibly possible amount of latency for a given client. Long latencies result in degraded throughput, and short timeouts result in long latencies. Balancing these two issues will result in an optimal timeout.

Dynamic The performance of microservices is constantly changing, both over the short- and long-term. In the short term, the microservice might experience higher latency due to increased load, for example, and in the long term, the implementation of the microservice might change. The algorithms we propose aim to be adaptive to both of these forms of change.

Consider the hypothetical scenario outlined in Figure 1, where after a change to the implementation of services C and D, B updates its timeout values, but A has not. Dynamicity would allow timeout values to change with the modified implementation, and timeout Optimality will ensure that it is within an acceptable range of the *observed latency*, what we define as the amount of time the client experiences the request taking (if the request times out, they only experience the timeout value amount of time).

3.3. Timeout Architectures

There are two proposed classes of solutions, Local and Global. They differ in who sets and calculates timeout values.

Local In the Local architecture, each service would spend its own resources determining all of its timeouts independently (see Figure 2a). Note that if a request times out, the service would not have the knowledge of how long the request would have taken. Parent services need to send requests without timeout values frequently to their children services to maintain an understanding of their performance. The pros and cons of this architecture are summarized below.

- Less accurate timeouts
- Repeated computation between independent microservices
- Less computationally intensive to calculate timeouts for one specific microservice
- New timeouts are regularly computed (services can afford to do this), so this solution is more responsive to change

Global In the Global solution, an independent body (an admin) would monitor all services within the system (see Figure 2b). It must be able to send requests without timeout values. If possible, Admin tests should be run in an API's shadow mode [4]. When this is not possible, because the Global option needs to send requests to the microservices while they are running normally, it cannot be run very often (or else users will face constant high latency due to the services being bombarded by requests by the admin). Having a large amount of data, the Global solution will be able to determine perfect timeout values. The pros and cons of the Global option are summarized below.

- More accurate timeouts
- More computationally intensive because the system is processing more data simultaneously
- Timeouts are computed less frequently because more data is required

3.4. Sequential Timeout Values

A request may time out due to a fatal server error, or just a temporary increase in latency. Normally, when a request times out, the client will issue another request with the same timeout value. However, by increasing the timeout value on a retry request after a failed request, we can do both of the following:

- Be less sensitive to unfavorable network or load conditions on the server side
- Determine whether we actually have a server failure

We propose a new way of dynamically setting timeouts, something we call sequential timeouts. Define an increasing sequence of values, $a_1, \cdots, a_n$. The client will send its first request with a_1, and if we time out with some timeout value a_m, we will send a retry with the longer timeout a_{m+1}. This allows the system to be resilient over the short-term.

3.5. Testbed Creation

A preliminary goal of this project was to implement a new distributed system from the ground up. Other commonly used systems like DeathStarBench's Social Network system were evaluated, but because it was difficult to modify how each microservice functioned. DeathStarBench uses Thrift to run RPC calls, and Thrift did not have a mechanism to set or modify timeouts.

3.6. Statistical Analysis and Mathematical Modeling

While systems are constantly changing, they largely remain structurally similar and are still the same application. In this approach, we assume that historical latency is representative of future latency. Through a thorough statistical and mathematical analysis of an experimentally determined probability density function for latency among other metrics, we are able to precisely calculate optimal timeout values.

Consider the hypothetical scenario of a microservice continuously updating its timeout value for a particular service it calls. If it sets its timeout value to the 99th percentile of just the times it sees, then since the latencies it sees will be restricted, it creates a positive reinforcement loop where the next 99th percentile will be lower than the previous one. In this case, eventually the timeout value will go to zero. So, we need some mechanism to know or predict the top 1 percent latencies. One possible way is to use **curve fitting** to extrapolate the known to

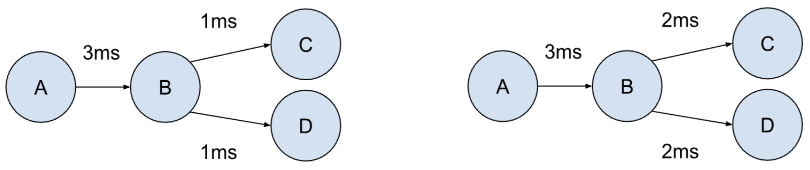
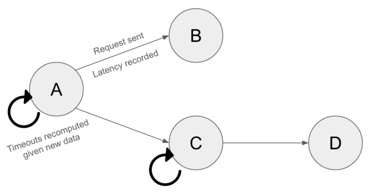
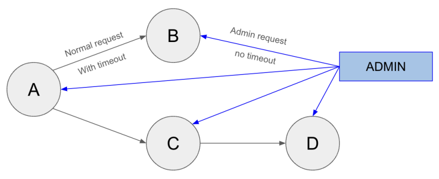


Figure 1: Left. A working system with acceptable timeouts. Right. After some internal modifications, services C and D now take 2ms each instead of 1ms. Timeout values are now not optimal.



(a) The local architecture



(b) The global architecture

Figure 2: Visuals of different timeout architectures

the unknown, though it fairly limited to specific distributions. As the general distributions of microservice latencies have not been well studied, we propose some possible curves. We will now discuss a more thorough analytical model.

Mathematical Model for Expected Latency First, we introduce two functions that take in timeout value. Define f to be the probability density function for latency, and define g to be a "cost" function: how much latency increases given a higher timeout value. We propose the following model E, the expected value of latency, given some timeout value t:

$$E(t) = \frac{\int_t^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x}(t + E(t)) + \frac{\int_0^t f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} \cdot \frac{\int_0^t xf(x)\mathrm{d}x}{\int_0^t f(x)\mathrm{d}x} + g(t) \tag{1}$$

See Figure 3 for the Markov chain used to construct this model. Let us break this model down into its parts.

$\dfrac{\int_t^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x}$: The probability that a request times out

$t + E(t)$: The amount of time taken by another request (we timed out, so we need to try again)

$\dfrac{\int_0^t f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x}$: The probability that a request succeeds

$\dfrac{\int_0^t xf(x)\mathrm{d}x}{\int_0^t f(x)\mathrm{d}x}$: The expected latency given that the request took under time t to complete

$g(t)$: Cost for sending an additional request

In order to use this model, we first derive E.

$$E(t) = \frac{t\int_t^\infty f(x)\mathrm{d}x + \int_0^t xf(x)\mathrm{d}x + g(t)\int_0^\infty f(x)\mathrm{d}x}{\int_0^t f(x)\mathrm{d}x} \tag{2}$$

We then minimize this function over t to determine the optimal timeout value.

Extensions to sequences This model is currently useful for changes in the long-term (simply recompute timeout values with new probability density function) but is ineffective in situations of short-term change; spikes in latency will always result in timeouts. To mitigate this issue, we use sequential timeout values. Define an increasing sequence $\{t_n\}_{0\leq n\leq z}$ for some z. We will always initially send a request with timeout value t_0, but whenever we encounter a timeout at t_i, we will send a retry with timeout value t_{i+1}. If we timeout with value t_z, the next request will be sent with timeout value t_z again. t_z is an input to the model.

We now modify the model to incorporate sequences.

$$E_n = \frac{\int_{t_n}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x}(t_n + E_{n+1}) + \frac{\int_0^{t_n} f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} \cdot \frac{\int_0^{t_n} xf(x)\mathrm{d}x}{\int_0^{t_n} f(x)\mathrm{d}x} + g(t_n). \tag{3}$$

For the sake of readability, for $n \neq z$, let $E_n = a_n + b_n E_{n+1}$. Note that when $n = z$, $E_z = a_z + b_z E_z$, or

$$E_z = \frac{a_z}{1 - b_z}$$

We now have the piecewise function

$$E_n = \begin{cases} a_n + b_n E_{n+1} & n < z \\ \dfrac{a_z}{1 - b_z} & n \geq z \end{cases}$$

Deriving a_n and b_n for future use, we have

$$a_n = t_n\frac{\int_{t_n}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} + \frac{\int_0^{t_n} f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} \cdot \frac{\int_0^{t_n} xf(x)\mathrm{d}x}{\int_0^{t_n} f(x)\mathrm{d}x} + g(t_n)$$

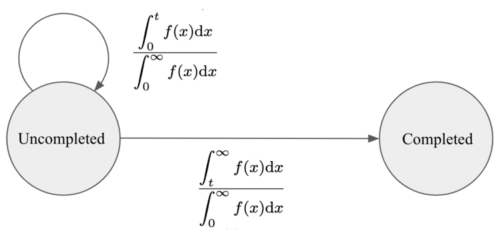
$$a_n = \frac{\int_0^{t_n} xf(x)\mathrm{d}x + t_n\int_{t_n}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} + g(t_n) \tag{4}$$

and

$$b_n = \frac{\int_{t_n}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} \tag{5}$$

In order to use this improved model, we need to minimize it over $t_0, t_1, \cdots t_z$. But this $z + 1-$dimensional minimization is very computationally intensive, and we propose a different implementation. To find minima, we require that $\nabla E_0 = \mathbf{0}$.

$$\frac{\partial E_0}{\partial t_0} = \frac{\partial E_0}{\partial t_1} = \frac{\partial E_0}{\partial t_2} = \cdots = \frac{\partial E_0}{\partial t_{z-1}} = \frac{\partial E_0}{\partial t_z} = 0$$



$\frac{\int_0^t f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x}$
Uncompleted
Completed
$\frac{\int_t^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x}$

Figure 3: Markov chain used in the mathematical model

$$\frac{\partial E_0}{\partial t_i} = \frac{\partial\left(a_0 + b_0\left(a_1 + b_1\left(a_2 + b_2\left(\cdots a_i + b_i E_{i+1}\cdots\right)\right)\right)\right)}{\partial t_i}$$

$$\frac{\partial E_0}{\partial t_i} = \frac{\partial\left(b_0 b_1 \cdots b_{i-1}\left(a_i + b_i E_{i+1}\right)\right)}{\partial t_i}$$

$$\frac{\partial E_0}{\partial t_i} = b_0 b_1 \cdots b_{i-1}\frac{\partial\left(a_i + b_i E_{i+1}\right)}{\partial t_i}$$

$$\frac{\partial E_0}{\partial t_i} = b_0 b_1 \cdots b_{i-1}\left(\frac{\mathrm{d}a_i}{\mathrm{d}t_i} + E_{i+1}\frac{\mathrm{d}b_i}{\mathrm{d}t_i}\right)$$

Thus,

$$\frac{\partial E_0}{\partial t_i} = 0 \implies \frac{\mathrm{d}a_i}{\mathrm{d}t_i} + E_{i+1}\frac{\mathrm{d}b_i}{\mathrm{d}t_i} = 0,$$

As b_i are strictly positive. From the definitions in (4) and (5),

$$\frac{\mathrm{d}a_i}{\mathrm{d}t_i} = \frac{\mathrm{d}}{\mathrm{d}t_i}\left(\frac{\int_0^{t_i} xf(x)\mathrm{d}x + t_i\int_{t_i}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} + g(t_i)\right)$$

$$\frac{\mathrm{d}a_i}{\mathrm{d}t_i} = \frac{t_i f(t_i) + t_i(-f(t_i)) + \int_{t_i}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} + g'(t_i)$$

$$\frac{\mathrm{d}a_i}{\mathrm{d}t_i} = \frac{\int_{t_i}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} + g'(t_i)$$

$$\frac{\mathrm{d}b_i}{\mathrm{d}t_i} = \frac{\mathrm{d}}{\mathrm{d}t_i}\left(\frac{\int_{t_i}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x}\right) = \frac{-f(t_i)}{\int_0^\infty f(x)\mathrm{d}x}$$

So,

$$\frac{\int_{t_i}^\infty f(x)\mathrm{d}x}{\int_0^\infty f(x)\mathrm{d}x} + g'(t_i) + E_{i+1}\left(\frac{-f(t_i)}{\int_0^\infty f(x)\mathrm{d}x}\right) = 0.$$

$$\int_{t_i}^\infty f(x)\mathrm{d}x + g'(t_i)\int_0^\infty f(x)\mathrm{d}x - f(t_i)E_{i+1} = 0$$

$$E_{i+1} = \frac{\int_{t_i}^\infty f(x)\mathrm{d}x + g'(t_i)\int_0^\infty f(x)\mathrm{d}x}{f(t_i)} \tag{6}$$

As we know t_z, and have that $E_z = \frac{a_z}{1-b_z}$, we can work our way backwards to determine all t_i.

3.7. Dynamic Timeout Control

In certain cases, it can be expensive to collect large amounts of data. Additionally, day-to-day variations are hard to account for in systems that process a lot of data to arrive at a perfect timeout value. When the latency of a system spikes, it is important to increase timeout values immediately in order to prevent cascading failures. In this approach, we take inspiration from the TCP congestion control algorithms.

TCP keeps track of a value known as *cwnd*, or congestion window. It modifies the *cwnd* by an amount known as the *mss*, or minimum segment size. TCP congestion control algorithms all have three phases [5]:

- **Slow start**, when TCP is aggressive and increases the *cwnd* by 1 *mss* upon every acknowledgement.
- **Congestion avoidance**, when TCP is cautious and increases the *cwnd* by 1 *mss* per round trip time.
- **Fast recovery**, when TCP encounters a packet loss and reduces *cwnd* to 1 or by a factor of $\frac{1}{2}$.

We extend the key ideas here (upon a failure, be more lenient; upon a success, be more strict) to our dynamic timeout system:

- Decrease timeout value on a successful request
- Increase timeout value on failed request (timed out)

See Figure 4 for a full flowchart.

4. Implementation

4.1. Testbed

Design Principles The testbed was designed with the the following principles in mind:

- Simplicity: How lightweight is the system? How easy is it to create a new distributed system from scratch?
- Robustness: Can a variety of behaviors be modeled? Is it easy to implement each of these behaviors?
- Comprehensibility: Is it easy to understand the behavior of the system:

Being a key part of the goal of the project, these principles were especially important to be followed in the implementation of the timeout scheme.

Simplicity To ensure simplicity, systems were implemented in Python with a microservice library called Py-MS. Additionally, a ”microservice generator” was implemented, which takes in a configuration file containing details for each service and automatically creates a base for every service, allowing the developer to focus on the implementation of the service rather than the boilerplate.

It is also easy to define a new type of timeout scheme, simply by creating a new instance of the Timeout class below.

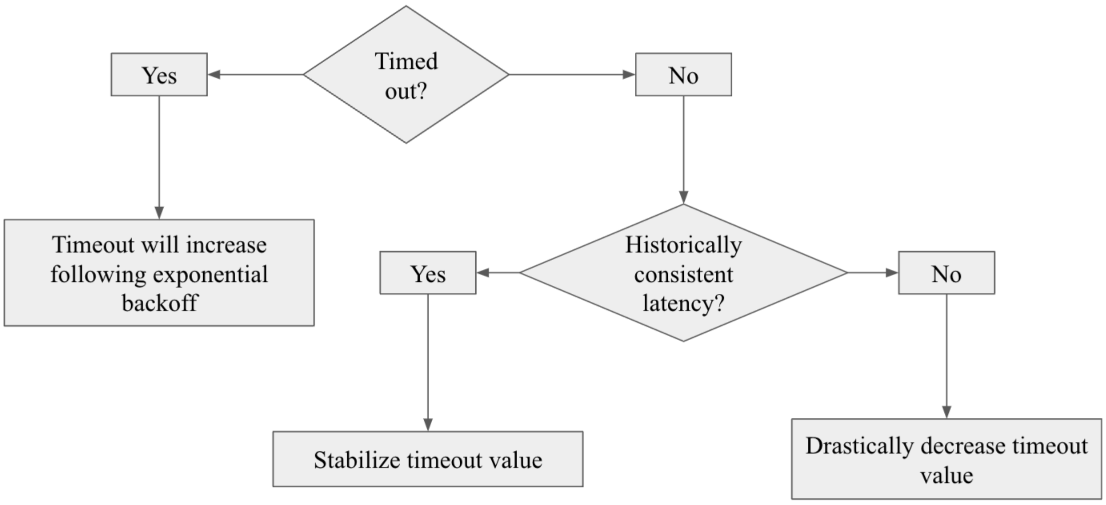

Timed out?
Yes
No
Timeout will increase following exponential backoff
Historically consistent latency?
Yes
No
Stabilize timeout value
Drastically decrease timeout value

Figure 4: A flowchart describing how the control-based approach changes timeout values following each request.

Robustness The implementation of each service is completely open to the developer. Each microservice system can either be asyncronous or synchronous.

The following is how the Timeout class is defined:

```
class Timeout(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def get(to):
        pass

    @abstractmethod
    def update(to, val):
        pass

    @abstractmethod
    def recompute():
        pass
```

The get method retrieves the timeout; the update method lets the Timeout class instance know current latency data, and the recompute method reruns any computation to come up with a new timeout.

Comprehensibility The Jaeger tracing agent was included in the design.

4.1.1. Purpose

The primary purpose of the service is to compute the area of a triangle using its coordinates using the formula $\frac{abc}{4R}$, where a, b and c are its side lengths and R is its circumradius.

Here is the detailed description of how each service works.

Triangle Area. The area service is given three coordinates, A, B, and C. It finds the distances between each pair of points through the distance service. It asks the circumradius service what the circumradius of this triangle is. It then computes and returns the area using the formula $\frac{abc}{4R}$

Circumradius. The circumradius service is given three coordinates, A, B, and C. It finds the perpendicular bisectors of pairs of points until it finds two that are not vertical and finds their intersection, using the perpendicular bisector service and the intersection service, respectively. This intersection is the circumcenter of the triangle. Then, using the distance service it finds the distance between any coordinate and the circumcenter, and returns this length.

Distance. The distance service is given the coordinates of two points x_1, y_1) and (x_2, y_2). It computes their Euclidean distance $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Perpendicular Bisector. The perpendicular bisector service is two points (x_1, y_1) and (x_2, y_2). It finds the line through their midpoint (midpoint service) and with a slope perpendicular to the segment through both of them (slope service).

Intersection. The intersection service is given the slopes (m_1 and m_2) of and a point ((x_1, y_1) and (x_2, y_2)) on each of two lines. It returns the intersection, which is $(\frac{m_1x_1 - m_2x_2 - y_1 + y_2}{m_1 - m_2}, m_1(x - x_1) + y_1)$.

Midpoint. The midpoint service is given two points (x_1, y_1) and (x_2, y_2). It returns the point $(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$.

Slope. The midpoint service is given two points (x_1, y_1) and (x_2, y_2). It returns the slope of the line passing through both, $\frac{y_1 - y_2}{x_1 - x_2}$.

A diagram of the structure of the system can be seen in Figure 5.

Implementation Specifics The testbed was implemented with the Python library pyms [6] and Docker, and instrumented with Jaeger. Each service ran in a separate container on Debian-Slim Linux, and communicated with another via HTTP requests. A workload generator (WRK2 [7]) was incorporated into the system. Python 3.8 was used.

4.2. Simple testbed design

Because our algorithms operate between exactly two services, we created an even simpler testbed with only two services, App1 and App2. Our algorithms run outside of both services, so requests are sent to them to set timeout values. App2 models changes in latency by sleeping for a certain amount of time on each request given by a latency model function. App1 sends a request to App2, and if it gets a response back without timing out, it returns the elapsed time. Otherwise, it errors, the

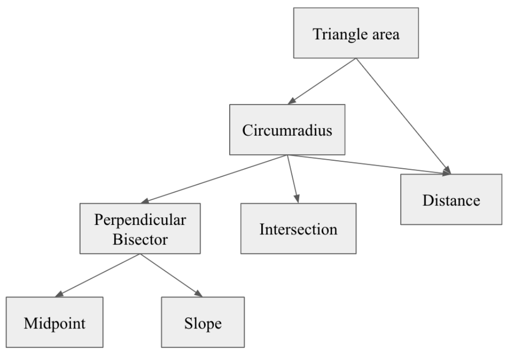


Figure 5: A visual of the implemented triangle are service. An arrow (for example from Triangle Area to Distance) represents that the former calls the latter.

response will be code 500, and the tester outside the services records the the elapsed time as the current timeout value.

4.3. Mathematical Modeling

In order to use the mathematical model outlined previously, we created a tree-based search algorithm to recursively find solutions to equation (6). First, we sampled regularly spaced values between an upper and lower bound (the minimum and maximum observed latencies, respectively) and determined the closest solution for each. For each of these solutions, excluding duplicates, we solve the equation again, ending up with a tree. When the tree depth (the number of elements in the timeout sequence, specified by the user) is reached, solutions are not found, and the sequence is ended by t_z. We call the algorithm that uses the mathematical model to determine timeout values the *Optimizer*.

4.4. Timeout Control

Timeout control alters the timeout value between two services following each request between them. It maintains a variable called *streak*, which keeps track of the current number of consecutive successful requests whose observed latency is within a percentage *tolerance* of the previous *streak* number of requests. On a failed request, the timeout value is multiplied by a fixed factor so that it will follow exponential backoff. On successful request, the timeout value will be lowered proportional to the distance to the observed latency the last request, divided by the logarithm of streak to stabilize it. So that the timeout value doesn't decrease so far that a request is likely to time out, a small value ϵ is added back onto the timeout value, defined as the average of the last *streak* number of requests multiplied by a factor called the *buffer*. Below is our implementation in Python. We call the algorithm that uses congestion control to determine timeout values the *Controller*.

```
class PID_Decay_Control:
    def __init__(self):
        self.timeout = 0.02
        self.pid = PID(0.1, 0, 0, setpoint=self.timeout)
        self.pid.output_limits = (-1, 1)
        self.timeout_factor = 1.5
        self.values = []
        self.times = []
        self.streak = 0
        self.tolerance = 0.2
        self.failures = 0
        self.eps = 0.005
        self.safe_buffer = 0.01
        self.eps_reset = 0.05

    def update(self, success: bool, time: float = None):
        if success:
            self.times.append(time)
            self.streak += 1
            self.pid.setpoint = time
            power = self.pid(self.timeout)
            ideal = sum(self.values[-self.streak:])/self.streak
            self.eps = sum(self.times[-self.streak:]) \
                        /self.streak * self.safe_buffer

            if abs(((ideal - self.timeout) / power) - 1) \
                    < self.tolerance:
                power = self.pid(ideal)/math.log(self.streak + 1)

        else:
            self.eps = self.eps_reset
            self.failures += 1
            self.streak = 0
            self.pid.setpoint = (self.timeout) * self.timeout_factor
            power = self.pid(self.timeout)

        self.timeout += (power + self.eps)

        self.values.append(self.timeout)

    def get(self):
        return self.timeout
```

4.5. Combined System

Following evaluation of the *Optimizer* and the *Controller* independently, we discovered that the former was under-responsive and the latter was over-responsive (see Conclusions). Given these opposite problems, we propose a combined architecture that resolves both. Given the rapidly correcting *Controller*, we use this system for timeouts on *diagnostic* requests, those sent on a regular interval to determine the true latency of the system (not cut off by the *Optimizer*'s timeouts). The *Optimizer* was also modified to consider newer requests more than older requests by skewing the input data: the ith oldest request in the buffer is repeated i times, then fed to the algorithm. The Combined System is a Local architecture.

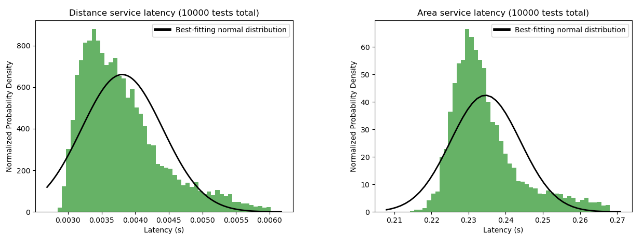
5. Results

We evaluated the Controller and Optimizer on three different metrics, comparing them with the current state-of-the-art, exponential backoff.

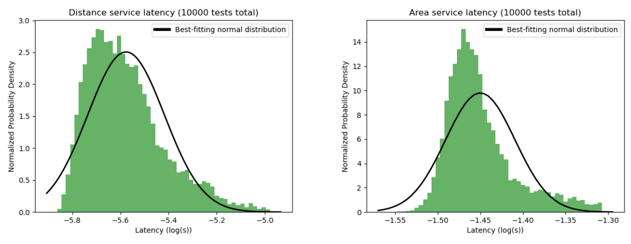
- **Speed.** How fast we can recompute timeout values?
- **Feasibility.** What is the resource usage (compute and memory) per request?
- **Robustness.** How adaptive is the timeout strategy to short-term fluctuations like high network load?
  - How many requests time out?
  - How close is the timeout value to the latency?

5.1. Curve Fitting

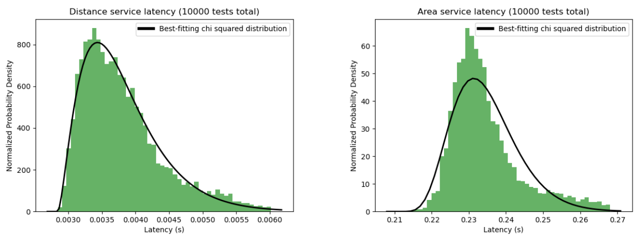
Normal Distribution The normal curve didn't fit the distributions well due its long tails and tall heads [8]. See Figure 6a.



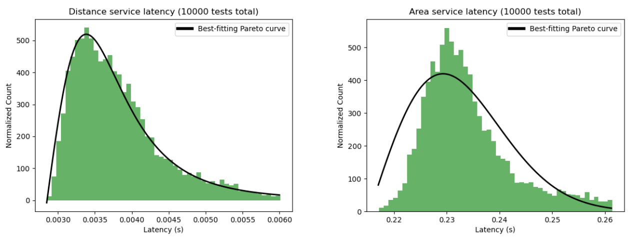
(a) Normal distribution.



(b) Log-normal distribution.



(c) χ^2 distribution.



(d) Pareto curve.

Figure 6: Histograms of distance (left) and area (right) service latencies best fitted to various curves and distributions

Log-Normal Distribution The log-normal curve fit the data better than the normal curve, but still suffers from not matching the long tails. See Figure 6b.

χ^2 Distribution This distribution fits the Distance latencies well, but is unable to meet the high peaks of the area service. See Figure 6c.

Pareto Curve The general form of a Pareto curve [9, 10] tested was $f(x) = \frac{abcx}{(1+b\cdot(x-s)^2)^{a+1}}$. This distribution fits the distance service well, though it does not fit the area service well. See Figure 6d.

Conclusions Overall, the χ^2 distribution was the best-fitting.

5.2. Evaluation of Algorithms

We first evaluated the latency on the simpler 2-service testbed, testing eight different scenarios, each 2000 requests long. Each algorithm was allowed a 200 request initialization period. We compared the performance of our two algorithms with exponential backoff, modeled as a sequence whose first element is 20% higher than the starting latency, whose factor is 2, and has 5 elements. Here are the scenarios we tested:

- A stable latency of 0.1 ms (Stable)
- A mean reverting random walk with mean 0.1 ms, volatility 0.01 and reversion strength 0.3 (Mean Reverting Random Walk)
- A slow increase from 0.1 ms to 0.4 ms (Slow Increase)
- A slow decrease from 0.4 ms to 0.1 ms (Slow Decrease)
- A fast jump from 0.1 ms to 0.4 ms (Fast Increase)
- A fast drop from 0.4 ms to 0.1 ms (Fast Decrease)
- Spikes from 0.1 ms to 0.4 ms (Spikes)
- Dips from 0.4 ms to 0.1 ms (Dips)

We plotted the timeout value, actual latency (obtained through diagnostic requests with no timeout value), observed latency, and marked where the systems timed out (Figures 15 -22).

We evaluated each solution based on three metrics:

- The sum of the observed latencies over all requests needed to fill 2000 responses divided by the integral of the latency model function (Time Fraction)
- The percentage of requests that timed out (Timeouts)
- The average percent change from observed latency to timeout value (Closeness)

5.3. Algorithm-Specific Test Implementation

Timeout Control We used a *tolerance* of 0.2 and a *buffer* of 0.1.

Optimizer Every 5 requests another was sent with no timeout to determine the actual processing time of the service. The tester kept track of a buffer of the last 200 requests sent with no timeout. Timeouts were recomputed every 20 requests. The cost function used was $g(t) = 0.4t$.

5.4. Tabulated Results

Here are the tabulated results for the eight scenarios, comparing the Combined solution with Exponential Backoff.

Random Walk	Time Fraction	Timeouts	Closeness
Combined	0.91	0.398%	38.3%
Exponential Backoff	1.04	0.00%	15.2%

Slow Increase	Time Fraction	Timeouts	Closeness
Combined	1.05	0.349%	18.9%
Exponential Backoff	1.57	49%	2.39%

Slow Decrease	Time Fraction	Timeouts	Closeness
Combined	1.05	0.1%	103%
Exponential Backoff	1.08	0.00%	104%

Fast Increase	Time Fraction	Timeouts	Closeness
Combined	1.02	0.843%	29.0%
Exponential Backoff	1.57	46.5%	7.05%

Fast Decrease	Time Fraction	Timeouts	Closeness
Combined	1.01	0.299%	77.0%
Exponential Backoff	1.05	0%	189%

Spikes	Time Fraction	Timeouts	Closeness
Combined	0.981	12.7%	32.7%
Exponential Backoff	1.20	10.7%	12.5%

Dips	Time Fraction	Timeouts	Closeness
Combined	1.04	0.249%	33.5%
Exponential Backoff	1.06	0%	37.6%

Stable	Time Fraction	Timeouts	Closeness
Combined	0.861	0.299%	29.1%
Exponential Backoff	1.07	2.77%	20.6%

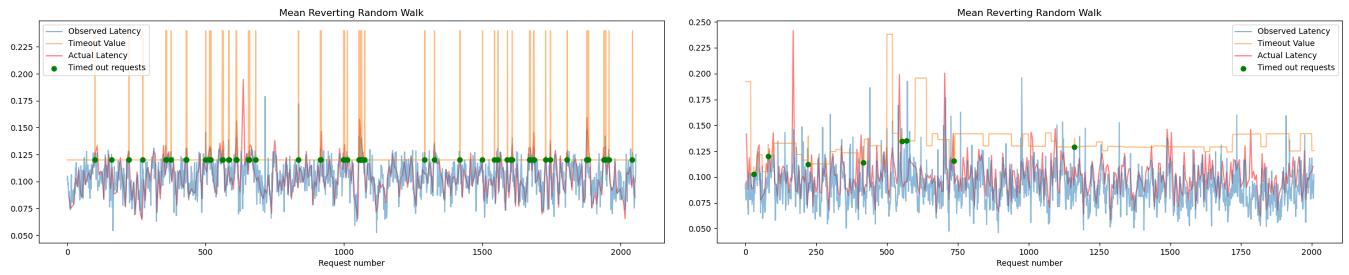


Figure 7: Comparison of Exponential Backoff (left) and Combined System (right) with latency modeled by a mean reverting walk

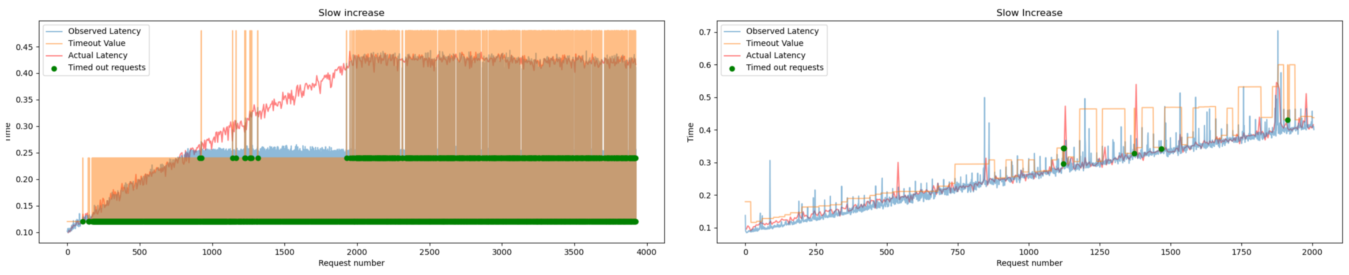


Figure 8: Comparison of Exponential Backoff (left) and Combined System (right) with latency modeled by a slow increase

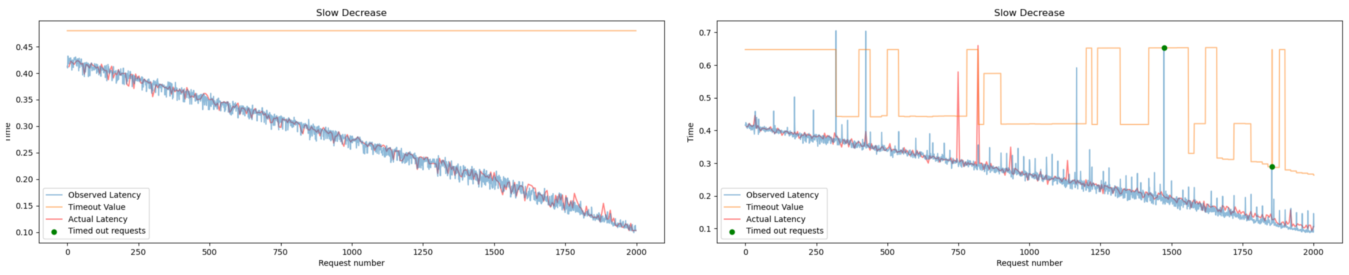


Figure 9: Comparison of Exponential Backoff (left) and Combined System (right) with latency modeled by a slow decrease

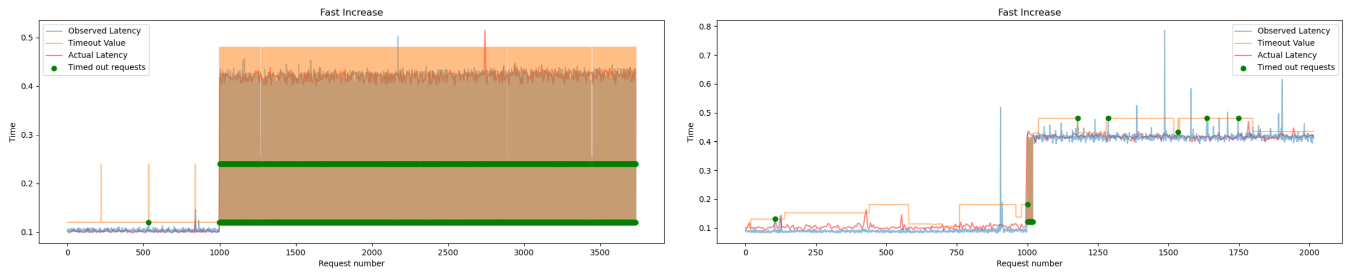


Figure 10: Comparison of Exponential Backoff (left) and Combined System (right) with latency modeled by a fast increase

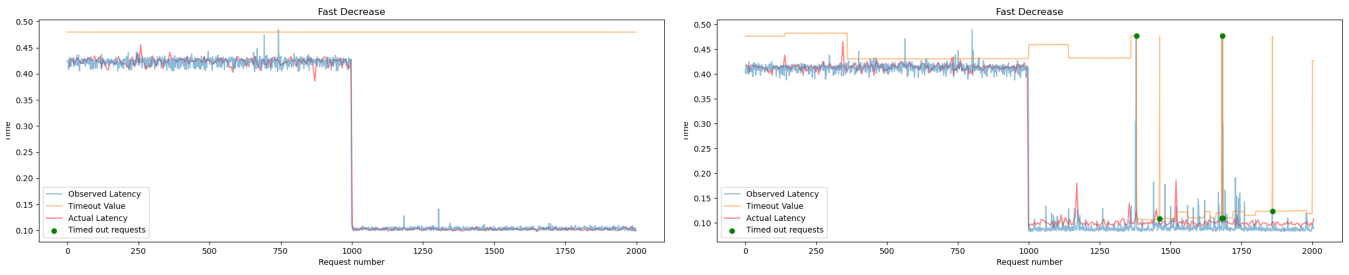


Figure 11: Comparison of Exponential Backoff (left) and Combined System (right) with latency modeled by a fast decrease

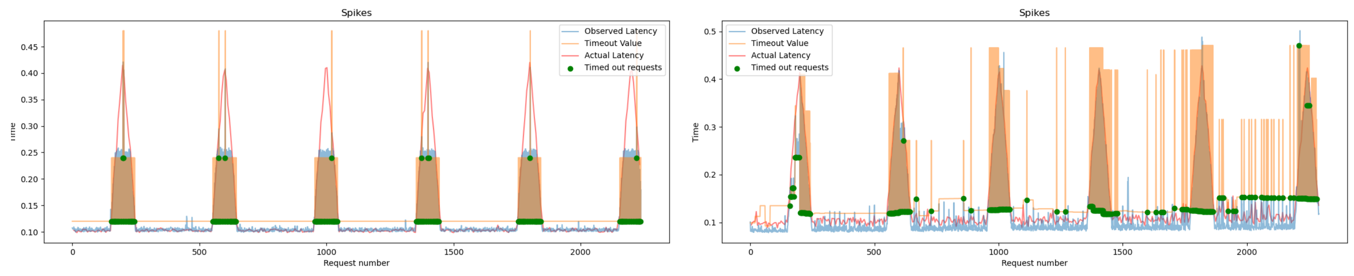


Figure 12: Comparison of Exponential Backoff (left) and Combined System (right) with latency modeled by spikes

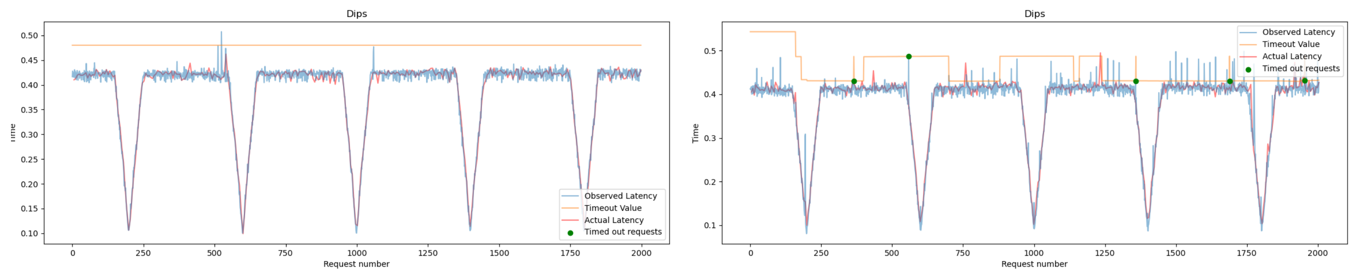


Figure 13: Comparison of Exponential Backoff (left) and Combined System (right) with latency modeled by dips

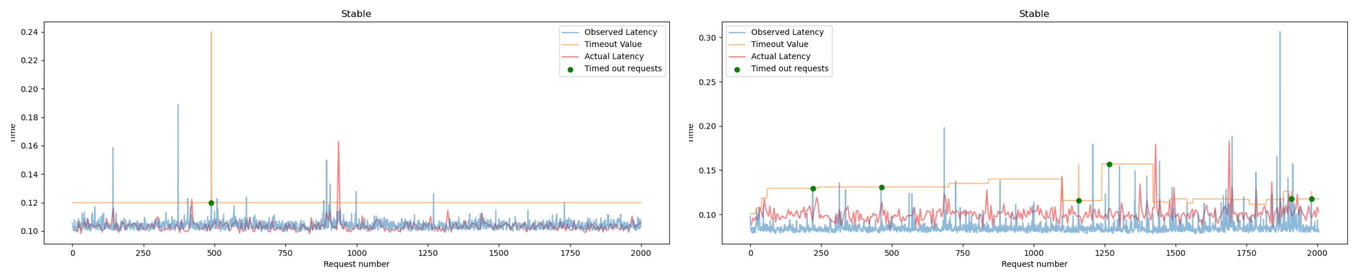


Figure 14: Comparison of Exponential Backoff (left) and Combined System (right) with latency that is stable

6. Conclusions and Future Work

The Controller and Optimizer significantly outperform Exponential Backoff on scenarios where the latency increases (Figure 16, 18). Exponential Backoff performs equivalently to our Controller and Optimizer when latency is decreasing (Figure 17, 19). The Controller performs better on Spikes than the Optimizer and Exponential Backoff, which perform equally (Figure 20). On Dips, the Optimizer performs equivalently to Exponential Backoff, both performing better than the Controller (Figure 21).

The Controller is able to maintain all of its statistics. However, it has a fundamental problem illustrated by how it times out on the way up from dips (Figure 21): it is much too dynamic, and its timeouts are not really useful in understanding the state of a microservice. On the contrary, the optimizer solution is too static, seen in how it takes a very long time to increase its timeout values on the fast increase (Figure 18).

The Combined solution remedies both of these issues, and by skewing data to newer results, it also outperforms the Optimizer, specifically in the case of a Fast Increase in latency (Figure 10 versus Figure 18). As expected, there are not significant differences between the Combined solution and Exponential backoff in Stable latency (Figure 14) and Mean Reverting Random Walk (Figure 7). When latency decreases, the Combined solution's timeout value also falls, giving it an advantage in Timeout Closeness (Figures 9, 11). In situations of Dips and Spikes, the two algorithms perform similarly (Figures 12, 13). When the latency increases, the Combined solution significantly outperforms Exponential Backoff (Figures 8, 10).

In addition, the current testbed is very simplistic as it consisted of only two services. We will extend our methods to a full distributed system, evaluating both the local and global architectures, starting with the Triangle Area distributed system, then SocialNetwork, a significantly larger distributed system with more diverse processes commonly used for benchmarking.

SocialNetwork does not have a mechanism for individual services to time out on requests, and neither does Thrift, which provides the mechanism for services to send requests to each other. We have so far modified SocialNetwork so that requests can timed out. We plan to create an implementation for timeouts in SocialNetwork like we did in the Triangle Area system by manually implementing timeouts, cutting off requests after the given wait time.

One limitation with out current testbed is that the time fraction is not entirely accurate, and has significant margins of error, seen in how the time fraction for Timeout Control in Dips is less than 1. We may need to develop a different metric to test how long the requests take on average relative to how long they are expected to take.

References

[1] Martinek P. Al-Debagy, O. A comparative review of microservices and monolithic architectures. *18th IEEE International Symposium on Computational Intelligence and Informatics*, pages 000149–000154, 2019.

[2] Ojdowska A. Przybyłek A. Blinowski, G. Model-driven engineering of fault tolerant microservices. *Fourteenth Int. Conf. Internet Web Appl. Serv*, pages 1–6, 2019.

[3] Barroso L. A. Dean, J. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[4] Vishal Varshney Anton Ilinchik. All you need to know about timeouts: How to set a reasonable timeout for your microservices to achieve maximum performance and resilience. *Zalando Engineering Blog*, 2023.

[5] Tcp congestion control algorithms.

[6] pyms.

[7] J. Richards. wrk2.

[8] The second law of latency: Latency distributions are never normal.

[9] Li Q. Yang, B. Enhanced particle swarm optimization algorithm for sea clutter parameter estimation in generalized pareto distribution. *Appl. Sci.*, 2023.

[10] He J. Zhang, W. Modeling end-to-end delay using pareto distribution. *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*, 2007.

A. Full Tabulated Data

Random Walk	Time Fraction	Timeouts	Closeness
Timeout Control	1.04	1.48%	26.0%
Optimization	1.11	0.891%	50.9%
Combined	0.91	0.398%	38.3%
Exponential Backoff	1.04	0.00%	15.2%

Slow Increase	Time Fraction	Timeouts	Closeness
Timeout Control	1.06	0.150%	7.28%
Optimization	1.12	1.28%	6.54
Combined	1.05	0.349%	18.9%
Exponential Backoff	1.57	49%	2.39%

Slow Decrease	Time Fraction	Timeouts	Closeness
Timeout Control	1.05	0.200%	12.9%
Optimization	1.10	0.00%	77.7%
Combined	1.05	0.1%	103%
Exponential Backoff	1.08	0.00%	104%

Fast Increase	Time Fraction	Timeouts	Closeness
Timeout Control	1.05	0.695%	14.2%
Optimization	1.14	8.13%	22.2%
Combined	1.02	0.843%	29.0%
Exponential Backoff	1.57	46.5%	7.05%

Fast Decrease	Time Fraction	Timeouts	Closeness
Timeout Control	1.04	0.150%	19.4%
Optimization	1.08	0.150%	121%
Combined	1.01	0.299%	77.0%
Exponential Backoff	1.05	0%	189%

Spikes	Time Fraction	Timeouts	Closeness
Timeout Control	1.26	1.04%	22.3
Optimization	1.36	10.3%	41.7%
Combined	0.981	12.7%	32.7%
Exponential Backoff	1.20	10.7%	12.5%

Dips	Time Fraction	Timeouts	Closeness
Timeout Control	0.986	0.498%	15.5%
Optimization	1.07	0.00%	37.3%
Combined	1.04	0.249%	33.5%
Exponential Backoff	1.06	0%	37.6%

Stable	Time Fraction	Timeouts	Closeness
Timeout Control	1.03	0.249%	14.1%
Optimization	1.07	0.100%	33.3%
Combined	0.861	0.299%	29.1%
Exponential Backoff	1.07	2.77%	20.6%

B. Timeout Control and Optimizer Graphs

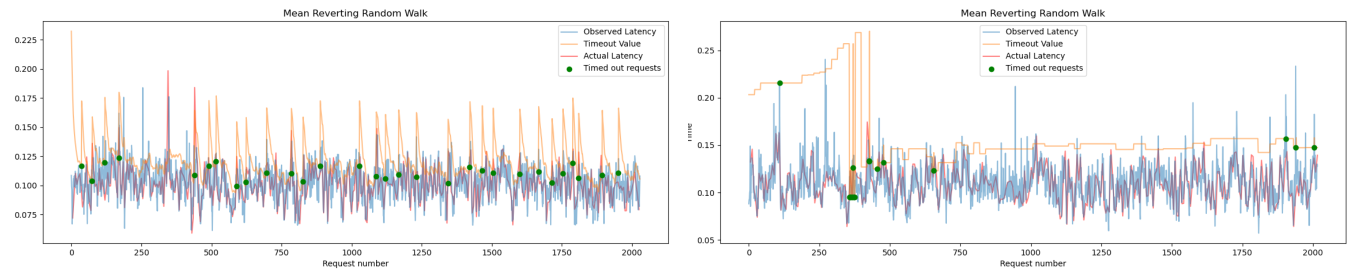


Figure 15: Comparison of Controller (left) and Optimizer (right) with latency modeled by a mean reverting walk

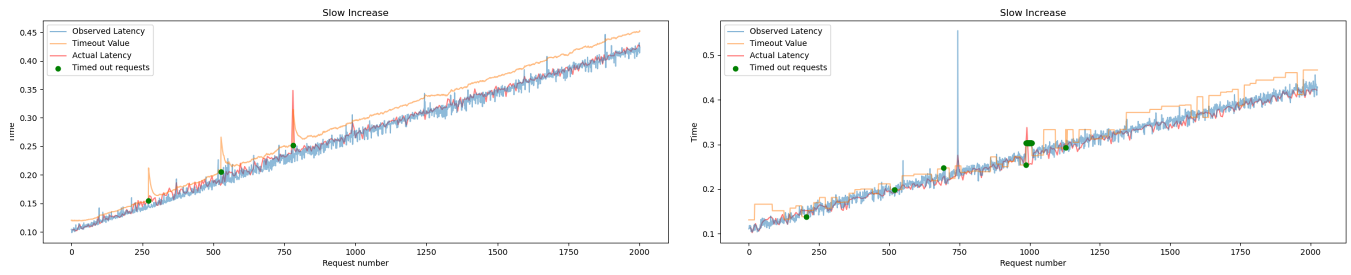


Figure 16: Comparison of Controller (left) and Optimizer (right) with latency modeled by a slow increase

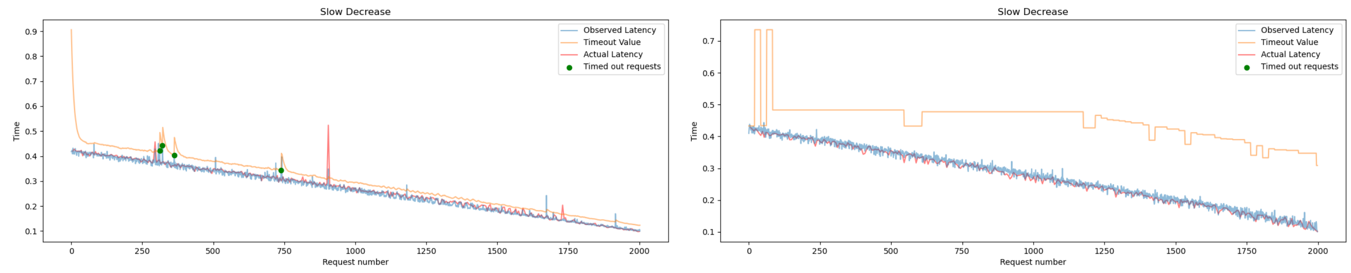


Figure 17: Comparison of Controller (left) and Optimizer (right) with latency modeled by a slow decrease

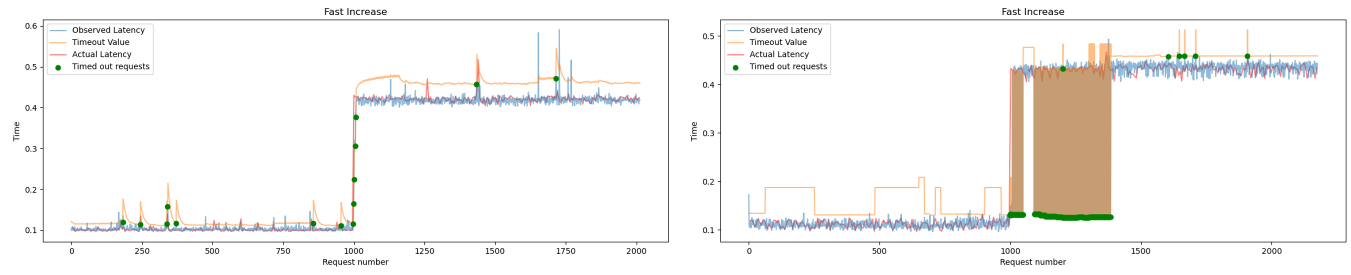


Figure 18: Comparison of Controller (left) and Optimizer (right) with latency modeled by a fast increase

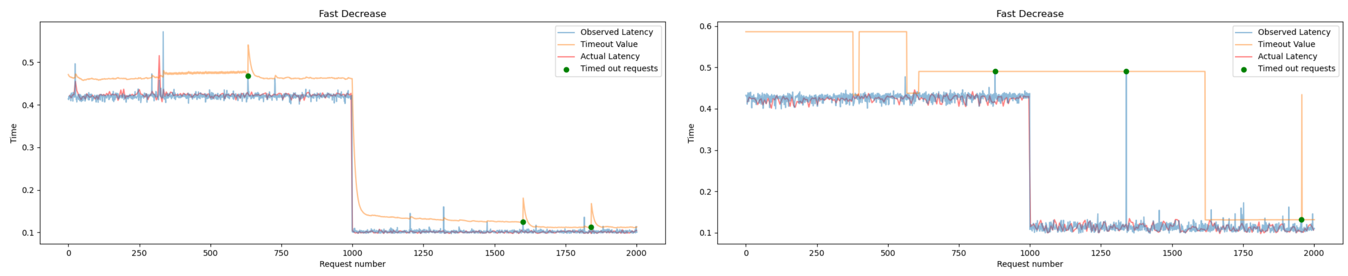


Figure 19: Comparison of Controller (left) and Optimizer (right) with latency modeled by a fast decrease

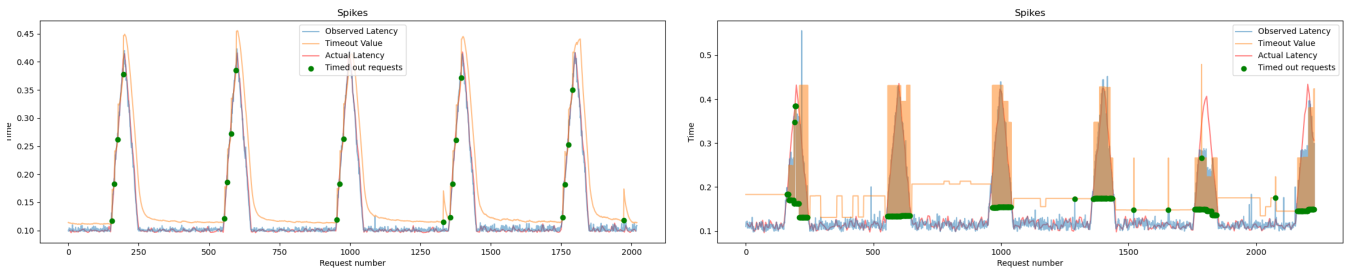


Figure 20: Comparison of Controller (left) and Optimizer (right) with latency modeled by spikes

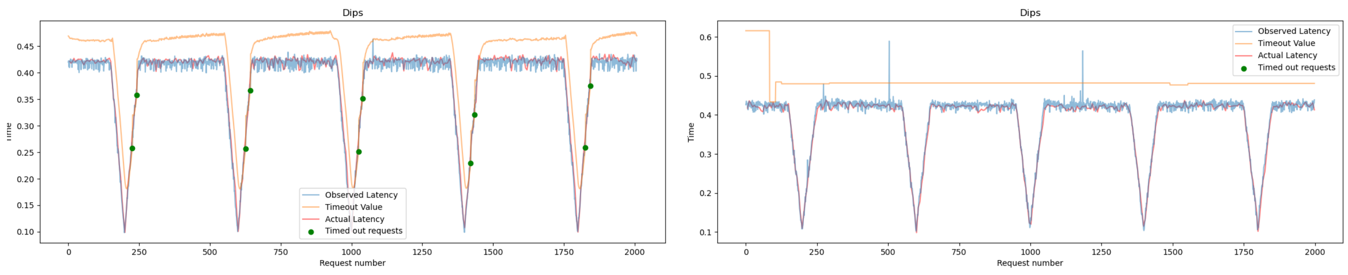


Figure 21: Comparison of Controller (left) and Optimizer (right) with latency modeled by dips

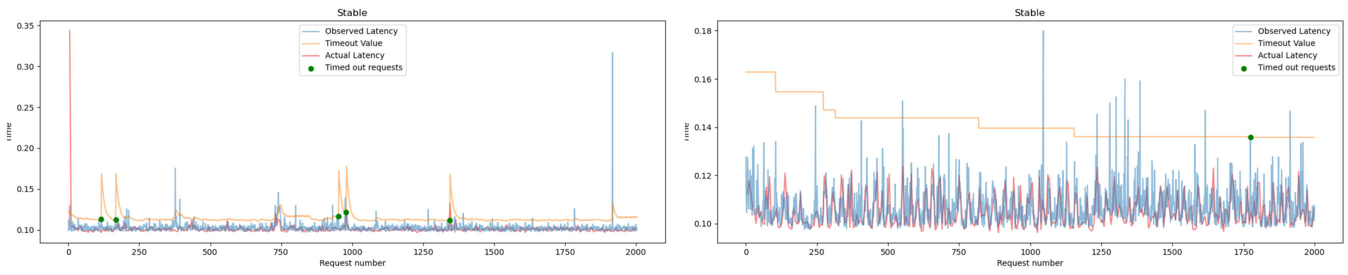


Figure 22: Comparison of Controller (left), and Optimizer (right) with latency that is stable