Adaptive Timeout Strategies for Microservice Applications

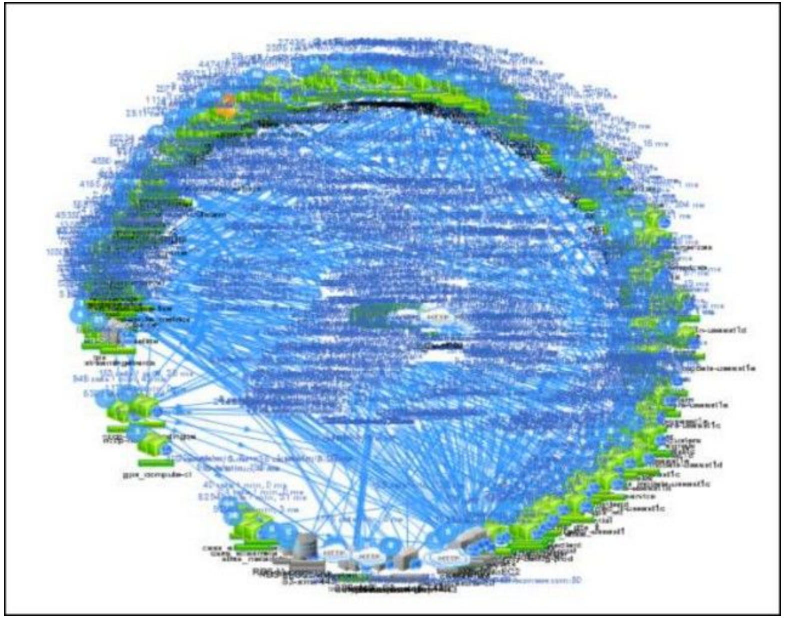
Adrita Samanta & Govind Velamoor

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

MIT PRIMES October Conference, 10/12/2024

Distributed Systems are powerful but complex

Netflix's distributed systems



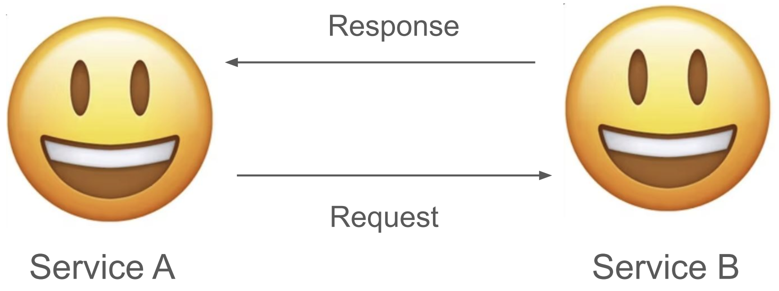
Benefits:

Scaling

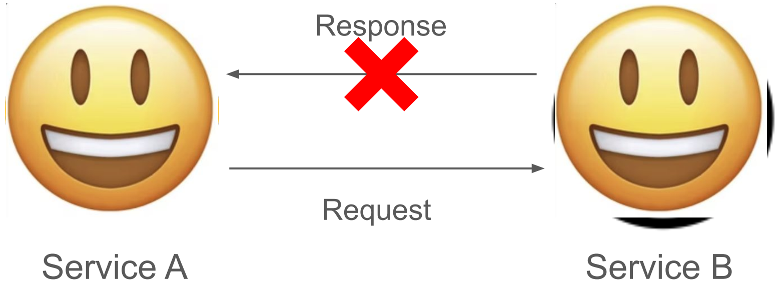
Performance

Challenge:
Debugging and failure-tolerance

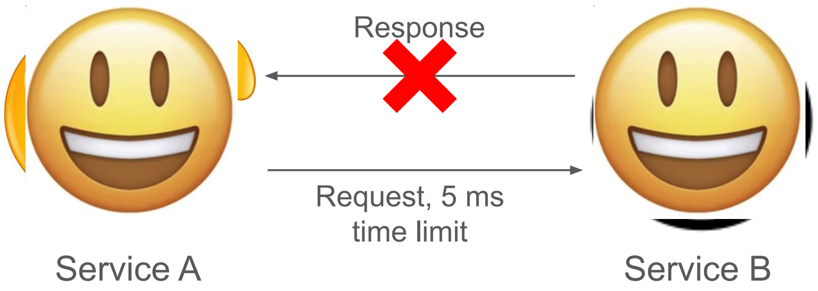
Why are timeouts important for distributed systems?



Normal Request



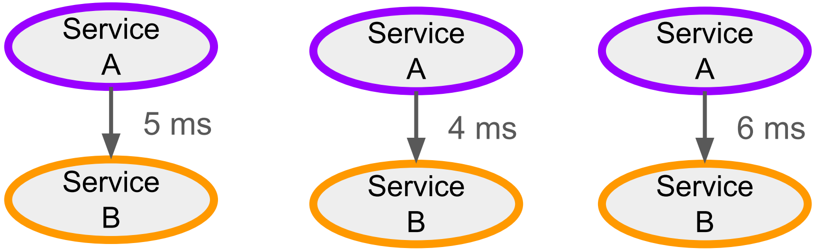
Failed Request



Request with a timeout

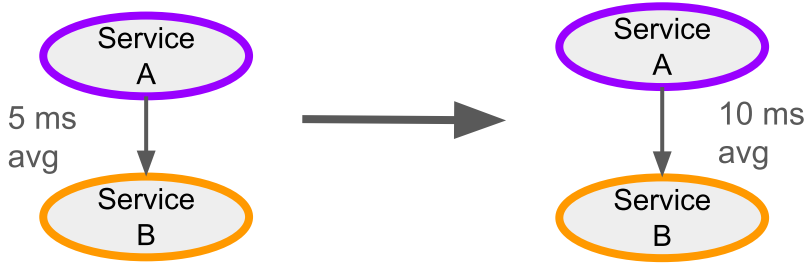
Challenges with setting timeouts

Latency is not the same every time



Time 1 Time 2 Time 3

Systems are changing

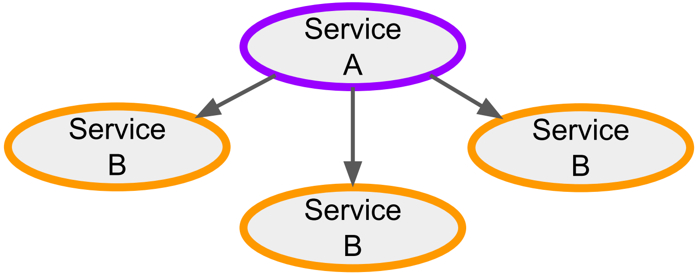


Original average latency

Request's average latency increases

Other challenges:

- Cache hit or miss.
- Systems can get overloaded.
- Multiple servers can accept the same request.



When is a timeout **optimal**?

- As systems evolve, timeouts change.

- An **optimal timeout** is a timeout that results in the minimal possible average amount of time before a response is received.
  - Too short → wasting work since we have to reissue requests
  - Too long → wasting time when request should have been discarded

- We **continuously update** the timeout values to adapt accordingly.

Table of Contents

- **Three algorithms**
  - **Dynamic Timeout Control**
  - Mathematical analysis
  - Reinforcement Learning
- Evaluation
- Evaluation on a larger-scale / Future Work
- Summary

A simple control-based approach saves resources

- Collecting data is costly
- Day-to-day variations are hard to account for in timeout values

Dynamic timeout control (inspired by TCP)

Goal: Update timeout value on a per-request basis without previous data

Design overview:

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

Dynamic timeout scheme: In the short/long term the system will adjust timeouts

Timeout Control Decision Process

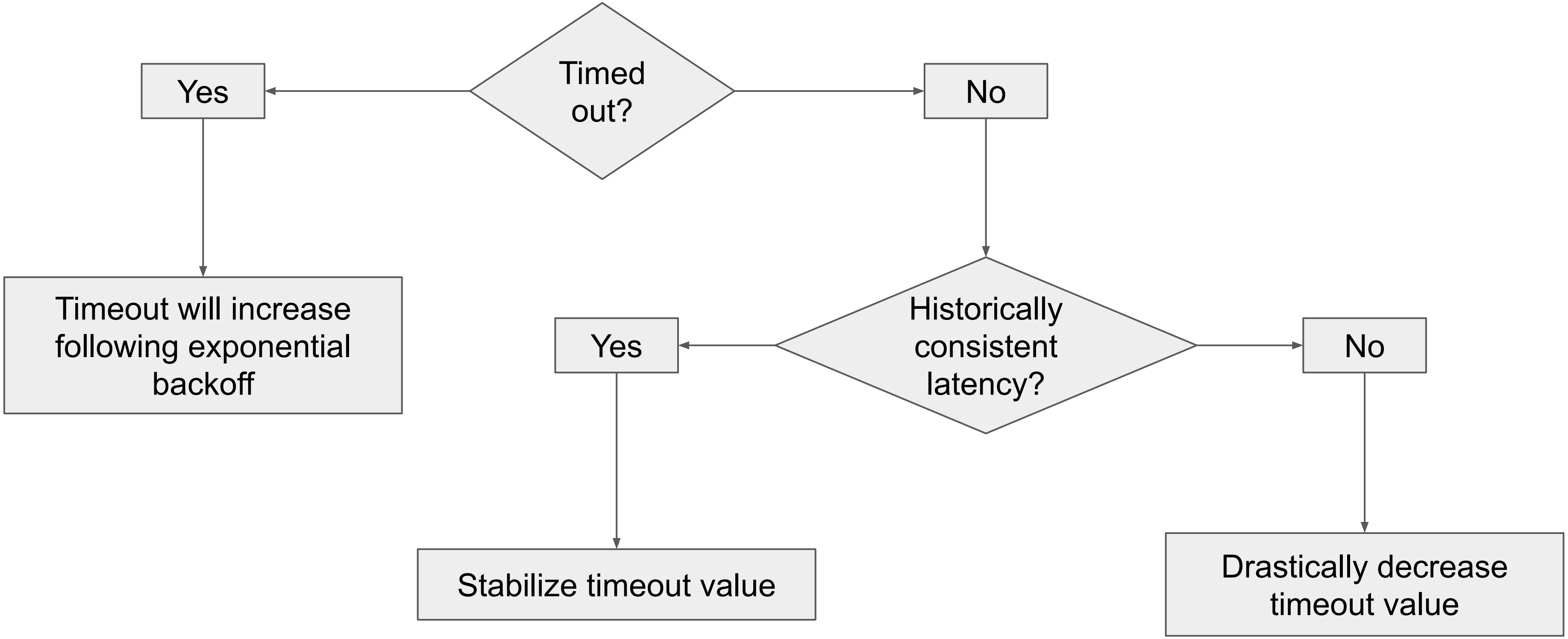
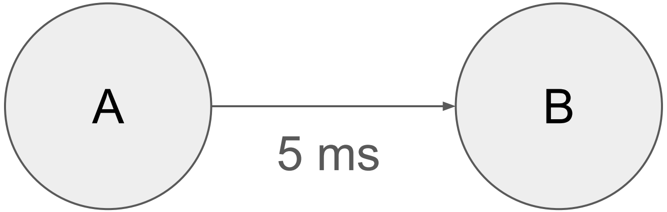


Table of Contents

Mathematical Analysis uses historical data for prediction

- While systems are constantly changing, they remain structurally similar and are the same application.
- Assumption: historical data is representative of future latency.
- Allows us to precisely calculate the optimal timeout value.

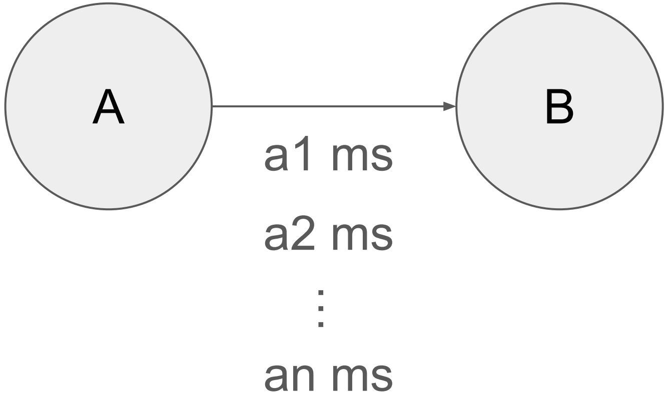
Increasing timeout values allow for precise hedging



Normal timeouts

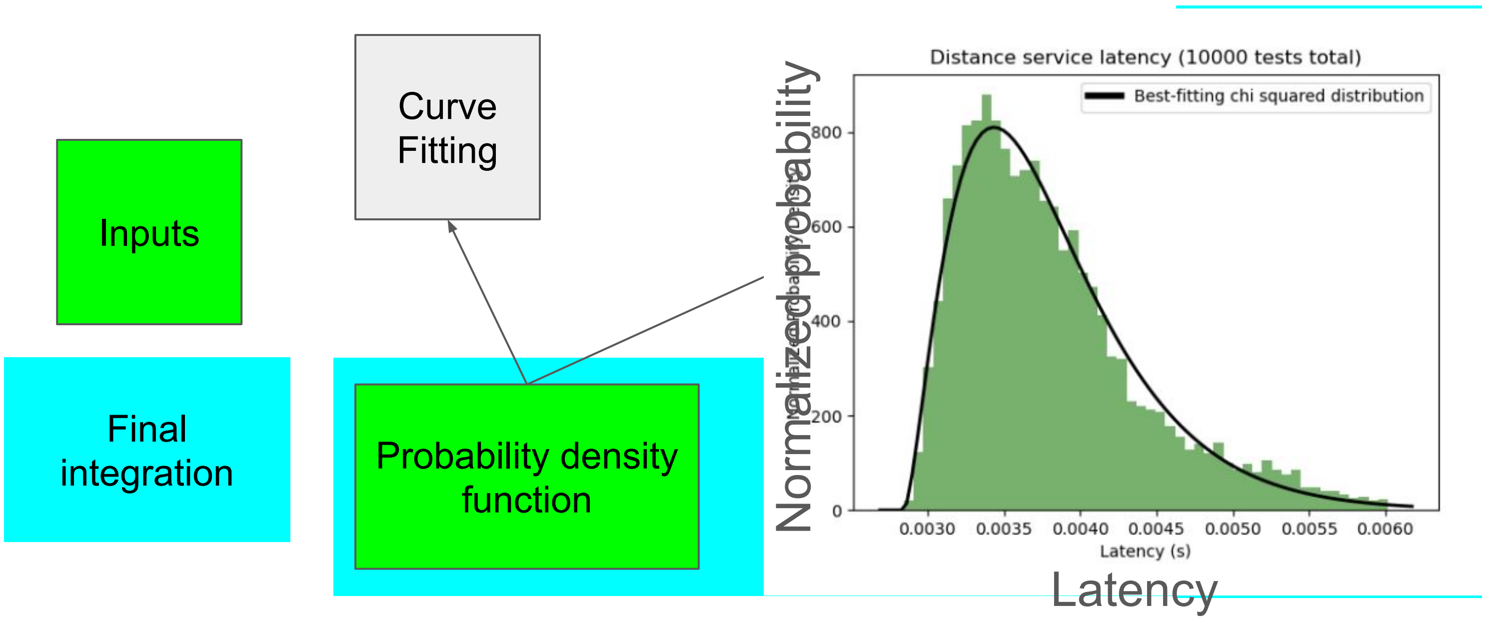
Failure conditions
- Temporary increase
- System failure

Increase timeout value
- Sensitivity reduction
- Failure confirmation

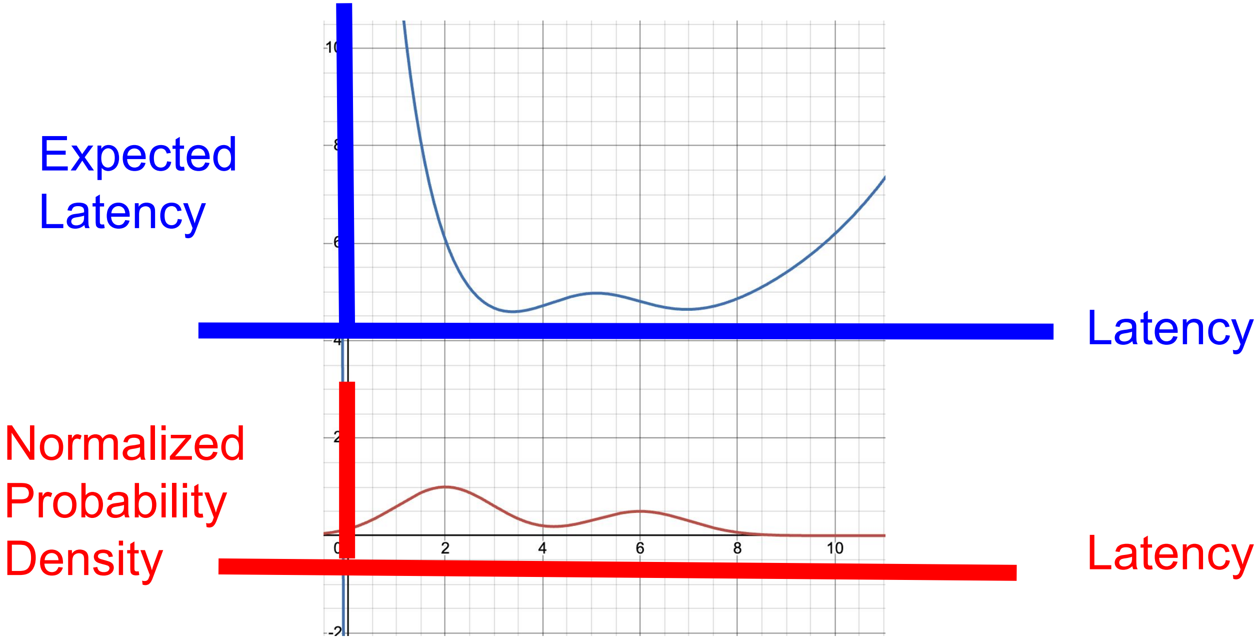


Sequential timeouts

Mathematical Model for Latency



Math behind mathematical model (1)



Math behind mathematical model (2)

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

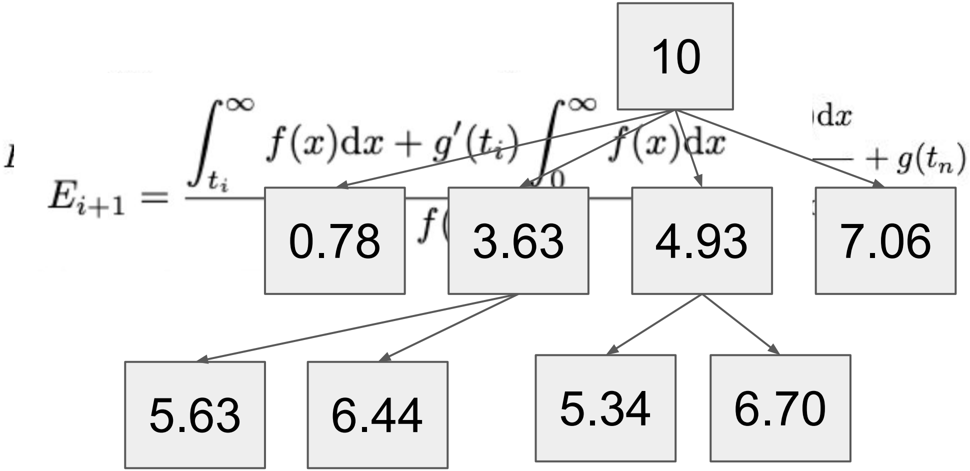
Case where we time out

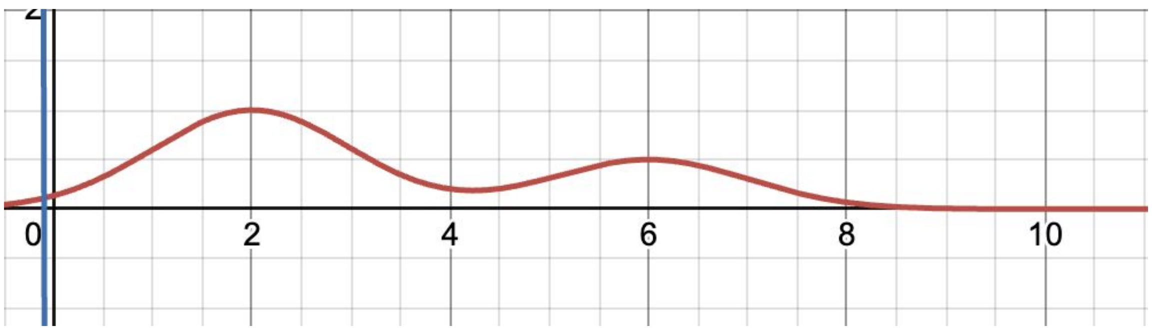
Cost function

Case where the request is successful

Math behind mathematical model (3)

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






```
0.78
    10
    ---> 6.33
3.63
    5.63
        10
        ---> 3.53
    6.44
        10
        ---> 3.49
    10
    ---> 3.84
4.93
    5.34
        10
        ---> 3.89
    6.7
        10
        ---> 3.84
    10
    ---> 4.16
7.06
    10
    ---> 3.93
10
---> 5.61
```

Table of Contents

Reinforcement Learning 101

An agent learns optimal moves by interacting with its environment

- Winning moves are rewarded
- Losing moves are punished
- Like someone learning to play a game

State used: array of latencies seen so far

Reward function used: -[total latency]

The complexity of latency might be understood by RL

- The systems are slow-changing, but latency itself is extremely complex
- Historical latency data may not be sufficient

Reinforcement learning

Goal: Train a Deep Deterministic Policy Gradient (DDPG) model to get a response back in the least amount of time possible.

Dynamic timeout scheme:

Short term: Agent will be trained on how to respond

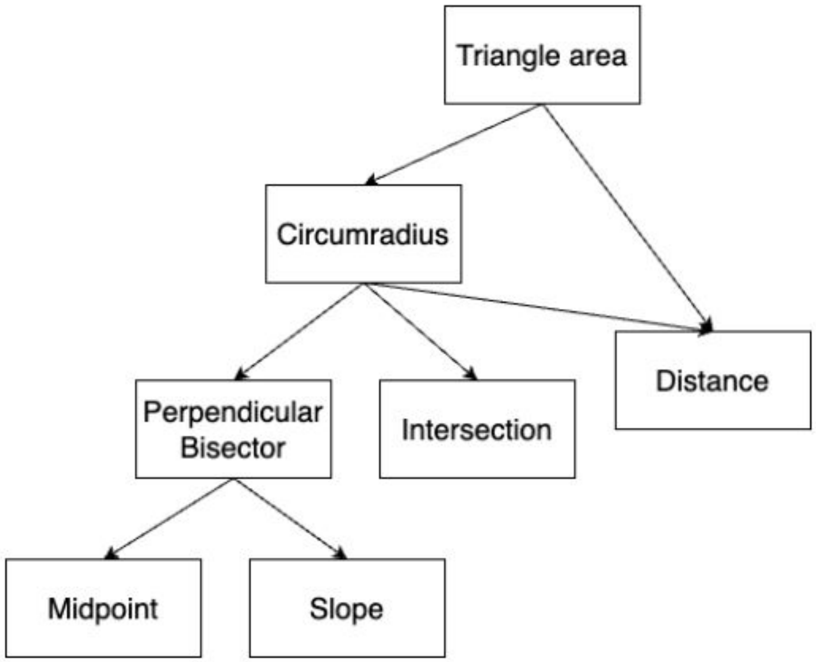
Long term: Agent will be retrained

Summary of strategies

Rank (Anticipated)	Speed	Feasibility	Robustness
1	Timeout Control Mathematical Modeling	Timeout Control	RL Model
2		Mathematical Modeling	Mathematical Modeling
3	RL Model	RL Model	Timeout Control

Table of Contents

Testbed Creation



Tools used

- Python
- Docker
- Modified version of wrk2
- Jaeger

Experiments/Evaluations

- Compare the median and 99th percentile observed latency of the three proposed solutions with a control
- Various situations will be tested

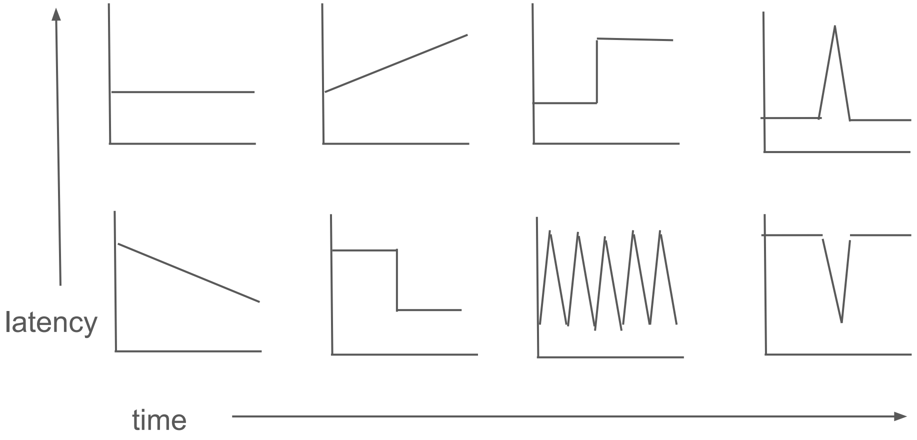


Table of Contents

Evaluating our methods on a larger-scale

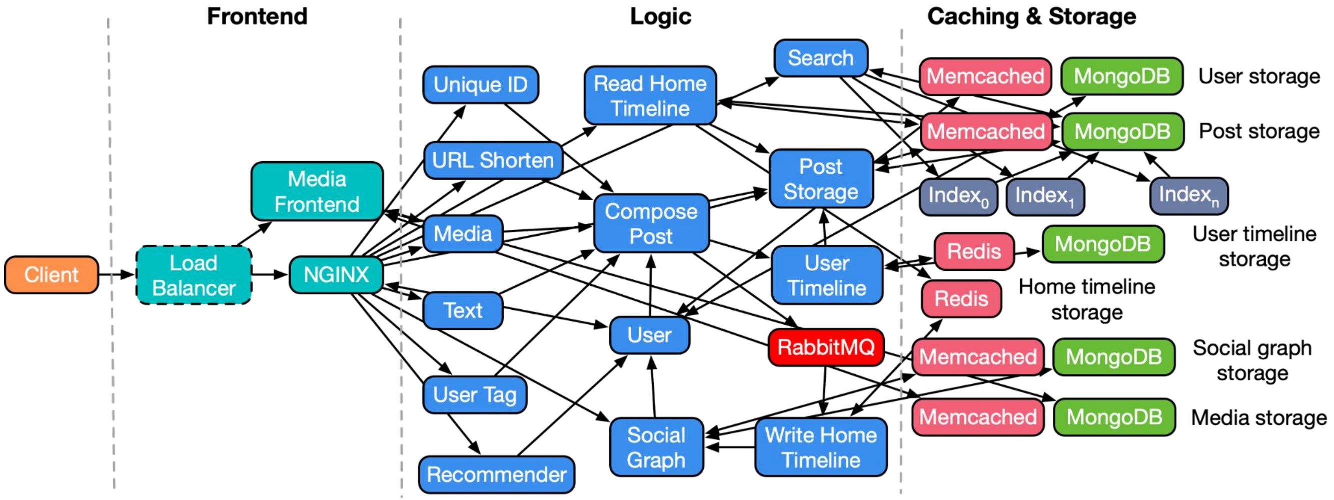
- The distributed system application we developed is very small and not representative of large-scale applications.

- DeathStarBench is an open-source benchmark suite modeled based on real-world applications.

- We used DeathStarBench’s socialNetwork application to create our evaluation testbed.

What is socialNetwork?

- socialNetwork is a networking application similar to Twitter and Facebook.
- The user can create posts, read posts, and follow/unfollow other users.
- The application also has many databases which increases the complexity of requests in the application.



Adding timeout functionality to socialNetwork

- The framework which socialNetwork uses to make requests, doesn’t have timeouts.

- We modified the services of socialNetwork, in C++, to be able to call timeouts. We used Docker to create another image and deploy our updated application.

- We tested sample workflows on our application using an HTTP workload generator (wrk2) and visualized their requests (and timeouts called) using Jaeger.

- Eventually, we will feed in the optimal timeouts from our algorithms into our testbed to test their performance.

Future Work

- We will evaluate the three algorithms on the smaller testbed we developed.

- We plan to connect the updated socialNetwork testbed with our algorithms to evaluate their performance on a larger scale.

Table of Contents

Summary

- When systems change, we need to update timeout values with them.

- We designed three novel algorithms to achieve this task.

- We implemented a new distributed systems testbed from scratch.

- We updated a large-scale application, socialNetwork, to have timeout functionality which we will use to evaluate the performance of our three methods.

Acknowledgements

We would like to thank

- Our mentors: Lan (Max) Liu, Zhaoqi (Roy) Zhang, and Prof. Raja Sambasivan for their guidance and time.

- MIT PRIMES: Dr. Slava Gerovitch and Prof. Srini Devadas for this great opportunity.

References

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

[2] E. Troubitsyna. Model-driven engineering of fault tolerant microservices. Fourteenth Int. Conf. Internet Web Appl. Serv, 2019.

[3] 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.

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

[5] 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.

[6] Tcp congestion control algorithms. https://www.tetcos.com/pdf/v13/Experiments/TCP-Congestion-Control-Algorithms.pdf

[7] B. Gregg. Frequency trails. https://www.brendangregg.com/FrequencyTrails/modes.html

[8] pyms. https://python-microservices.github.io/home/

[9] J. Richards. wrk2. https://github.com/giltene/wrk2

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

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

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

[13] Zhang Y. Cheng D. Shetty A. et al. Gan, Y. Death star bench repository.

[14] Zhang Y. Cheng D. Shetty A. et al. Gan, Y. An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems. ASPLOS ’19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019.

Notes

Overall: Speak slower; assuming too much knowledge form audience

Slide 2: what does the netflix distributed system mean; moves testbed to eval section

Slide 3: Rename title. Something like “proper timeouts help make system more efficient”

Slide 5: Define optimal

Slide 9: Animate different components as they are brought up

Slide 10: What does the graph actually say; talk about what each image means

RL Slides: There seems to be something missing for the intuition as to why RL was used

Slide 17: The diagrams need to be explained + add an example for how the systems will be compared

Slide 18/19: Continue from the evaluation testbed that Govind has and continue to say that it’s small so we use socialNetwork. socialNetwork doesn’t have timeout capabilities so we added this functionality. Walk through how we did that and mention future plan.

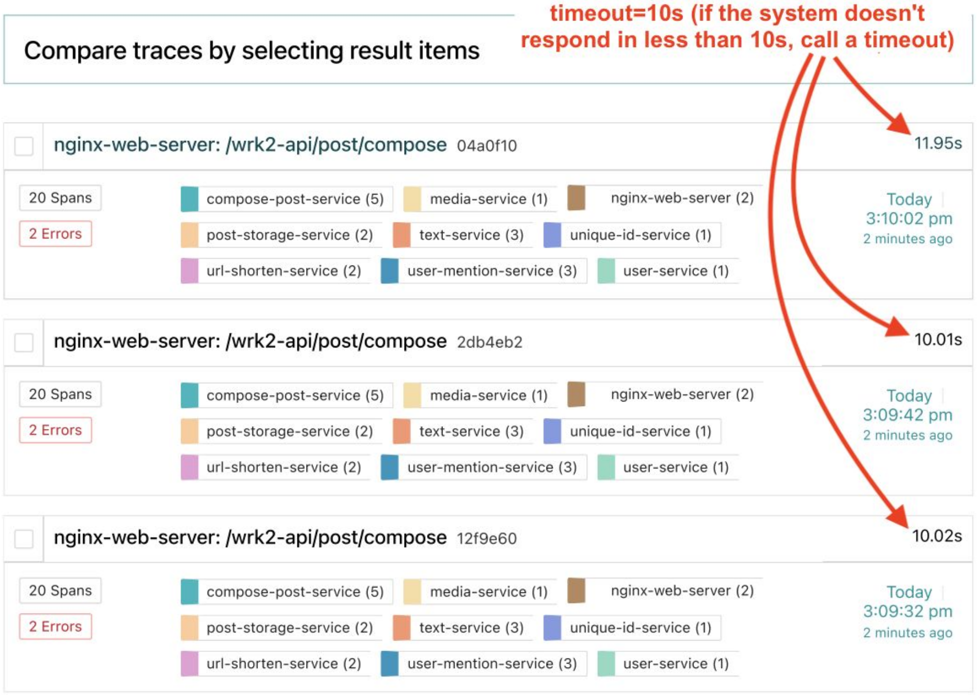
Add a conclusion slide

Example of timeout functionality in socialNetwork

- For one of socialNetwork's services, ComposePostService, we forced the service to take more than 20 s for each of its requests.

- To test our implementation of adding timeout values, we set the timeout value as 10 s.

- Eventually, we will feed in the optimal timeouts from our algorithms into our testbed to test their performance.



Setting a good timeout value is hard

- Latency is not the same every time
- Systems are changing
- Systems can get overloaded (metastable failure)
- Cache miss/hit
- There are multiple servers that can accept a single request, causing greater complexity
- Collecting data is costly
- Timeouts that are too short result in valid requests not being considered
- Timeouts that are too long result in resources being allocated to requests that will never return a response

Resilience to change in the system: Dynamic timeouts

- Sometimes, it may be useful to set a timeout once and for all.
- However, when systems evolve, **optimal** timeouts (neither too short nor too long) may change with them.
- We want to create an algorithm that can **continuously update** timeout values to adapt to the changing distributed system.

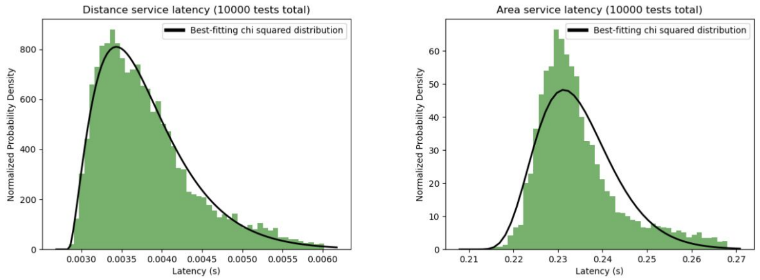
We have two levels of dynamicity

- Changes over long periods of time (e.g. an implementation for a service changes).
- Changes over short periods of time (e.g. latency momentarily spikes).

are we have timed out
- We send another request with a higher timeout value to hedge our bets
- Explain in terms of curves & bumps

Explain each p the equation intuitively

Model Iteration



Curve fitting

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

Expected latency function

n between waiting longer to get , or has it already failed?

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

Extensions to sequences

Organize the screenshot better

Practical implementation of mathematical model

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

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

```
0.78
    10
    ---> 6.33
3.63
    5.63
        10
        ---> 3.53
    6.44
        10
        ---> 3.49
    10
    ---> 3.84
4.93
    5.34
        10
        ---> 3.89
    6.7
        10
        ---> 3.84
    10
    ---> 4.16
7.06
    10
    ---> 3.93
10
---> 5.61
```

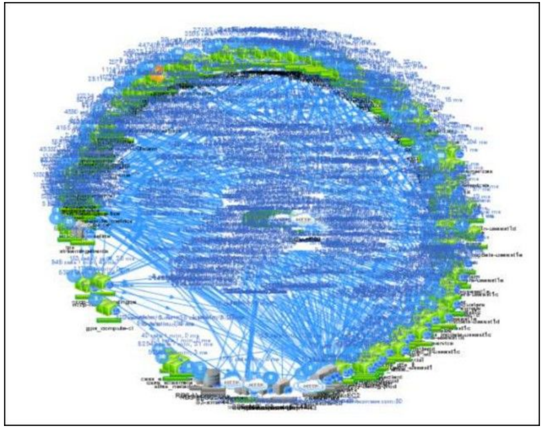
Distributed Systems are powerful but complex

Benefits:
Independent scaling of systems

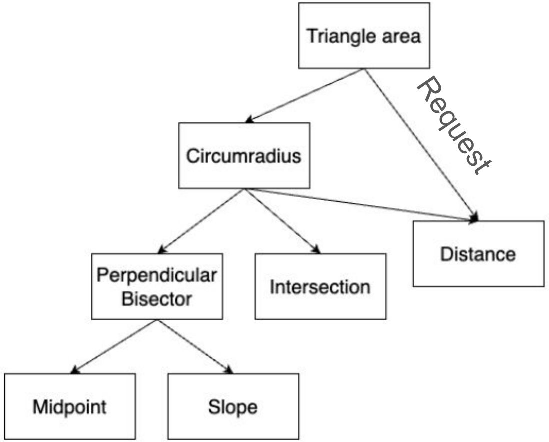
Parallelism

Language Heterogeneity

Netflix's distributed systems



Testbed we built



Challenge: debugging and failure-tolerance

Adding timeout functionality to socialNetwork

- socialNetwork makes requests using a framework called Thrift which doesn't have timeouts.

- We modified the services of socialNetwork, in C++.

- With our modified application we used Docker to create another image and tested sample workflows on our application using an HTTP workload generator called wrk2.