JustInTimeout: Enacting Dynamic Network Timeouts via Distributed Feedback

Dylan Wilson
Nisarg Chokshi
Yash Trivedi
North Carolina State University
Raleigh, North Carolina

ABSTRACT

Timeouts are a very crucial yet often overlooked part of a distributed system. A timeout too high can lead to poor user experience, while one too low can lead to high system utilization and create availability and performance issues. In this paper we present JustInTimeout, a system to collect, distribute and inject timeouts dynamically based on the current state of the system, ensuring that future requests do not timeout pre-maturely. While our approach is language agnostic, we focus our client side work on Java. We try to make adopting our system trivial by using Aspect Oriented Programming; we provide a transparent layer between the application and Java native I/O.

ACM Reference Format:
Dylan Wilson, Nisarg Chokshi, and Yash Trivedi. 2020. JustInTimeout: Enacting Dynamic Network Timeouts via Distributed Feedback. In *Proceedings of Advanced Distributed Systems (CSC 724).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In a distributed system, when a request is sent to be fulfilled, the requestor must decide how long to wait before concluding that the response is not coming back. For example, when system A sends a request to another system B, system A needs to make an up-front decision on how long to wait and then, whether to retry if the response isn't serviced in time. While selecting timeouts and retry policies isn't complex, it is easily ignored by developers. During development, it can be very difficult to imagine all of the inter-dependencies, workload changes, failures and real-world conditions that might eventually occur. Thus, misconfigured or no configuration of timeouts do occur and can cause the server system to hang or experience significant performance issues. For example, a misused timeout bug caused Amazon DynamoDB to experience a five-hour service outage in 2015 [13].

From previous research [5], [13] it has been observed that most of the timeout problems involve misused timeout, missing timeout, improper timeout handling, unnecessary timeout and clock drifting. [14] presents an approach which provides recommendations on the timeout values for the misused timeout bugs. From those recommendations, the system can update the timeout value the next time it 'restarts'.

In this paper we try a more dynamic approach we call JustInTimeout, where the timeout recommendation is made online and can be leveraged in a running system. When a client sends a request to a server, a sidecar collects the performance metrics and sends the collected data to a data broker. The data broker then calculates the updated system state and provides a new value of timeout using its online learning algorithm. On the next application initiated request, our shim pulls the timeout value from the data broker and overrides the application set timeout before making the network call.

Section 2 describes in detail the design and components of the proposed solution and provides details on the approach to be used for doing the online calculation of new timeout, Section 3 talks about the implementation details, Section 4 about the Evaluation methodologies we followed, Section 5 displays the results that we got from our experimental setup, Section 6 sheds some light on previous related work that has been done, Section 7 details the future work that this paper could inspire and finally Section 8 mentions the conclusions we drew from the project/paper.

2 SYSTEM DESIGN

JustInTimeout consists of four logical components. A performance collector agent that sits alongside each networked service and enables continuous measurement of network calls, a data collector, that collects and durably stores the raw performance data, a data broker that summarizes the collected data into a set of timeout values, one for each service (at a minimum), and a universal shim that runs within each physical client needing one or more network services (including services that use other services). Please see Figure 1 and Figure 2 for an overview of the components and how they work together.

2.1 Client Shim: Aspect Oriented Timeout Injection

On the client side, we implemented the ability to request/receive appropriate network timeouts from the rest of our system in real-time. We apply those timeouts directly for the client application on the next network call (which could be the first network call). We created a generic implementation that can be used by Java based clients without modification using Aspect Oriented Programming (AOP) to dynamically inject timeouts into network calls (java.nio) from beneath the application level. This means, for applications that use Java Native I/O we are able to modify already compiled binaries without requiring original source via post-compilation or

CSC 724, Spring 2020, Raleigh, NC

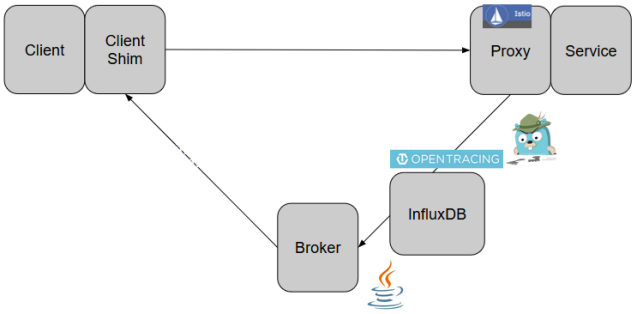

Figure 1: System Block Diagram. Each major component has been shown here.

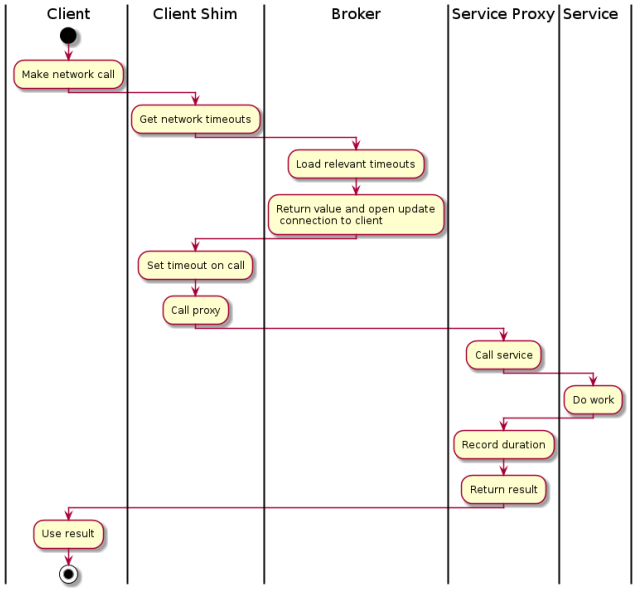

Figure 2: System Activity Diagram. Each major logical component and the interactions between are shown.

load-time AOP weaving. If a system is incompatible with our AOP approach, we could, in the future, modify a common networking library such as Jolly or Apache HttpComponent to consume and use our timeouts.

2.2 Proxy Sidecar: Performance Collector

This component collects and stores performance data automatically for all microservices in the Kubernetes cluster. Each microservice is fitted with an additional pod, termed a sidecar. This sidecar intercepts incoming requests to the worker pod, forwards the request to the worker pod and records metrics such as the response time. The performance data is forwarded to a storage system or directly to the Data Broker, depending on the Broker's design.

2.3 InfluxDB: Tracing Data

To enable ingestion, analysis and long-term storage across the entire data-set in real-time, we used the time series database InfluxDB that is able store the OpenTracing data format from our Istio/Jaeger setup. Influx is the source of truth for raw data used by our timeout brokers.

2.4 Broker: Timeout Predictor

The job of the broker is to consume and organize the tracing data, and then predict the current recommended timeout value for a given URL. Called by the client shim before making the actual network call, it informs the client on how long the network call might take and thus offers a recommendation for a timeout value. Client-broker communication will happen at random using the built-in round-robin load balancing in Kubernetes.

2.5 Broker - Shim Communication Protocol

The Shim and the Broker communicate via a REST API endpoint exposed by the Broker. The client calls this as an http call and provides it with a url whose timeout needs to be computed. The server will then respond with an integer value which reflects the recommended timeout in milliseconds.

3 IMPLEMENTATION DETAILS

3.1 Client Shim

3.1.1 *Technology Stack.*

- Java [21]
- Guava [12]
- AspectJ [10]

3.1.2 *Aspect Oriented Programming.*

In software design, the best way of simplifying a complex system is to identify the concerns and then to modularize them. In Aspect Oriented Programming, the crosscutting concerns are modularized by identifying a clear role for each one in the system, implementing each role in its own module, and loosely coupling each module to only a limited number of other modules.

We are using Aspect Oriented Programming to deliver a portable run-time library that can be used with any Java application without the need to re-build the application or any implementation change.

3.1.3 *Advice Execution and recursion prevention.*

Pointcuts [9] generally weave the advice code in the target code which could result in an infinite recursion if not handled correctly. In our use-case, we are advising the *HttpURLConnection.getInputStream()* and calling the same function again in the advice for getting timeout from the broker, which will result in an infinite recursion in the advice.

To handle this situation, we leverage the *cflow* and *cflowbelow* functions provided by AspectJ. *cflow* and *cflowbelow* validates the pointcut conditions passed as parameter to be in the call stack and below the top of the stack in the call stack respectively. We add a *!cflow(adviceexecution()) && !cflowbelow(within(HttpURLConnection.getInputStream()))* condition on the pointcut for *HttpURLConnection.getInputStream()* in order to prevent an infinite recursion.

3.1.4 *Accessing this object.*

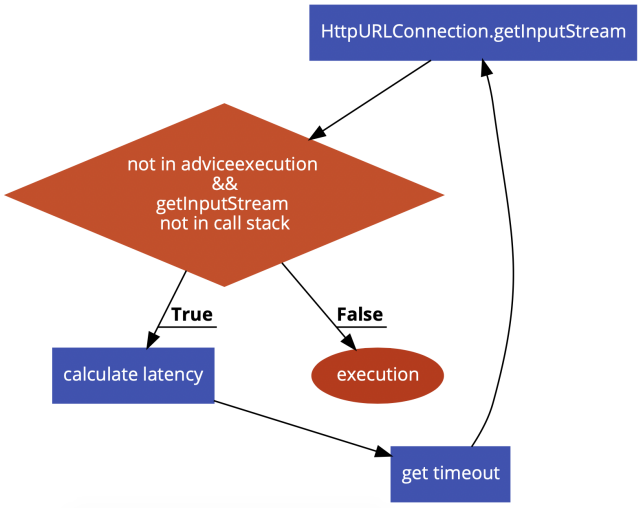

Figure 3: Code flow diagram. The possibility of infinite recursion and how it is handled is shown

The major use of Aspect Oriented Programming is to intercept method calls/executions. But with the timeout injection, we required to access the object on which the method intercepted by the *PointCut* was called in order to update the attribute. This was achieved by hardcoded unboxing to *HttpURLConnection* of the *target* attribute of the *PointCut*.

3.1.5 Calculating Latency.

The prediction provided by the Broker takes only the time required by the cluster to process the request and return a response under consideration. It doesn't include the latency between the client and the cluster, which is logically acceptable as every client would have a varying latency based on location and connectivity. Hence we need to calculate and add the latency to the predicted timeout by the broker.

The Java Networking library doesn't have built-in support for the ICMP protocol [22]. Hence we needed a workaround to calculate latency. We used the root endpoint on *port 80* to return a *404* and calculate the latency. This seems to be an efficient method of calculating the latency as the payload is almost negligible. An average of 10 latencies are returned as the value of latency.

3.1.6 Caching Broker Timeouts and Latency.

As discussed below, our RNN Broker makes new predictions on it's schedule and far less often than the client shim requests the timeout. Hence, it would be a bad idea to fetch the timeout value from the broker every time there is network call. A better approach is to cache the value for a short period and use the cached value until expiry. The GP broker, on the other hand, currently recalculates the timeout on every request, so a very short cache timeout on the shim is warranted.

For implementing this, we have used Google's Guava library to cache both the timeout values and the latencies. With the *LoadingCache* the cache executes the respective function to fetch the value, if the requested value is not present in the cache. This makes cache management easy, as we always lookup from the cache, and the cache fills itself as required.

3.1.7 Parameters and Environment Variables.

The client shim is designed to work out of the box by design. Hence the parameters that could change based on the system using the shim are taken as *System Property* or *Environment Variable*. Following are the variables available and required for the shim to work.

Broker URL

The Broker URL needs to be different for every system that uses the client shim. Hence, it is required to have a *System Property* that can be passed at the startup to the application, and remains unchanged during the execution. The broker URL can be provided using the VM Options and setting the property *brokerURL*.

For example, one would pass the following command line argument : *-DbrokerURL=<the url>*

Timeout Injection

There may be scenarios where there are expected anomalous behavior of the system, for example load testing or testing the accuracy of the broker predictions. To support this, we have introduced an environment variable to enable/disable the injection of timeouts. Furthermore, the client shim will dynamically read the value of this environment variable at runtime, hence a system restart is not required to reflect the changes.

The system needs to set the value of *injecttimeout* to *true* in order to enable the timeout injection. By default it will be disabled.

3.2 Failed Client Shim

The first approach worked well independently, but when integrated we discovered a huge design flaw. The flaw was then mitigated and this section discusses the implementation details of the failed approach and uncovers the things that went wrong.

3.2.1 Reading the Socket. With the first implementation, we tried going down to the low-level network calls to inject the timeout. The last procedure call in the Java Networking library is *java.net.SocketInputStream.socketRead* with *timeout* as one of the parameters. This method then calls the native (C++) code for doing the socket read.

The problem with this part of the code is that the *socketRead* method reads the data from a *FileDescriptor*. Socket Reading does wait for *FileDescriptor* to have some value to read, but it can also be the case that the FileDescriptor was pre-loaded and the *socketRead* function just did a system call to read from memory. This means that the time required for *socketRead* to execute would only be the time to read locally from the memory and not the network latency and the time taken by the server to respond.

Also, we noted that the FileDescriptor buffer is limited to 1000 bytes. Hence, if the response was going to be larger, multiple socketRead calls would be made, each with a different file descriptor. This means that the total timeout value of the server read we were applying, gets individual socket read timeouts, which was not the desired behavior.

3.2.2 File Descriptor and URL mapping.

When intercepting the *socketRead* function, we only get a *FileDescriptor* for the request, but to get the timeout prediction, we need to

have the endpoint. To address this, we found that the creation of *HttpClient* object, we can get what *FileDescriptor* is being used for what request. So we use a cache to store the mapping and later get the URL for the *FileDescriptor* in the *socketRead* pointcut.

3.3 Proxy Sidecar

Targeted at a Kubernetes deployment, we used the popular Istio and Jaeger projects to implement our performance collection system. Consisting of a generic sidecar pod that is automatically injected by Istio into every "worker" or microservice pod to proxy requests into the worker pod. This sidecar, implemented using the Envoy project [11], proxies calls to the worker and captures performance and tracing data for calls into the worker pod. This raw data is collected by Jaeger collector pods running in the cluster and stored durably in InfluxDB. Since this is already available in an existing Open Source project, Istio, we used Istio for this component. We paired Istio with a second open source project, Jaeger, to enable the OpenTracing standard including collection and storage. OpenTracing is a vendor neutral standard for logging and tracing across network calls in a distributed system. The OpenTracing initiative is a descendent of the Google Dapper distributed system as in [25]. With these components available off-the-shelf, we selected and installed them in our Kubernetes clusters. We customized Jaeger to store all tracing data in the popular time-series database, InfluxDB.

3.4 Recurrent Neural Network Data Broker

3.4.1 Technology Stack.

1) Python3

The choice for using Python 3 as the language of choice for coding was mainly the variety of machine learning libraries that come with it. These libraries are a key component of the Data broker. We used Keras for all the tools needed to create our models. The reasons for using Keras are detailed in [20].

2) Flask

Flask is the perfect choice when it comes to creating a micro-framework using Python. Flask is an extremely lightweight framework for Python. It's not required, frameworks usually never are, but it makes development faster by offering code for all sorts of processes like database interaction or file activity.

This is very helpful when it comes to this sort of a problem, where you can shift the focus entirely to the problem statement and not have to worry about the server configurations and management which is managed by Flask under the hood.

*3.4.2 Why Machine Learning - **RNN**.*

This problem can be simplified as a time series prediction problem where the previous inputs within a short window span will most likely be the best candidates to predict future values. Keeping this in mind, a powerful type of neural network designed to handle sequence dependence is called **Recurrent Neural Networks** or RNN.

The Long Short-Term Memory or LSTM network [15], is a recurrent neural network that is trained using Backpropagation Through Time and overcomes the vanishing gradient problem. As such, it can be used to create large recurrent networks that in turn can be used to address difficult sequence problems in machine learning and achieve state-of-the-art results. A similar problem is solved in [26] where the tutorial is designed for weather prediction.

*3.4.3 Model of choice - **Convolutional LSTM**.*

Convolutional LSTM is optimal when you have no need for convolution operation in the input data all the while making use of CNNs along with LSTM. In contrast, CNN-LSTM have two different modules which are combined together. The CNN part is a regular CNN which acts as a spatial feature extractor. The output of the CNN is multiplied by the LSTM cell to learn the temporal features [24].

Convolutional LSTM is designed for image inputs in mind. But for this problem, we transformed the inputs into dataframes of sequences. Thus, we are able to better capture the variances in the input sequences and thus train our model with these features. This enables our model to identify them more naturally and make predictions based on sequences on our test data.

Hyperparameters:

- Sequence size = 2
- No of steps = 4
- No of features = 1
- Learning Rate = 0.008
- Optimizer = Adam [17]
- Training size = 5000 points

3.4.4 Additional Components.

1) InfluxDB Connector

InfluxDB is an open-source distributed time series database. As the raw endpoint data was coming from an Influx database, a small connector had to be written which interacted with this database to quickly fetch the relevant data. For this, Python provides a client [6] which is used to establish this connection and manage it. Relevant timeouts have been configured here to handle the database fluctuations.

2) Server API

An important endpoint is **/getendpoints** which is called by the clients when they wish to get the endpoints configured in the server for which timeouts are computed.

Sample usage: http://<SERVER_IP>:<PORT>/getendpoints
Returns: ['flight', 'booking', 'customer', 'auth', 'acmeair']

The main endpoint for the server is **/gettimeout** which is what clients need to call to get the most recent timeout values computed by the server. It requires a string that contains the endpoint whose value will be returned as an Integer in milliseconds.

Sample Usage: http://<SERVER_IP>:<PORT>/gettimeout?url=<ENDPOINT> **Return:** 68

3.4.5 Multiprocess mode vs Multidocker mode.

In the **MultiProcess** mode, there is just one server running which spawns off multiple processes which individually do the timeout computation for the url assigned to them. In the **Multi-Docker** mode, this idea is taken to the extreme where there are independent workers who register themselves with the running broker and get assigned tasks. These workers can register/unregister themselves at will and the computations are thus managed.

MultiProcess:

The flow diagram for the data broker in Multiprocess mode can be found in Figure 4. On startup, the system calls the **init** function that does the following things.

(1) Read endpoint.config and get the endpoints for which to configure timeouts.
(2) Spawn processes for each endpoint and separately call timeout computation module.
(3) Each process then calls the Influx database to get the raw timeout data for their endpoint.
(4) Each process preprocesses the data and creates a ConvLSTM RNN model.
(5) The model is then trained with this data and next value predicted.
(6) The main process does some postprocessing on the predicted values and updates the dictionary with the new values.
(7) At this point the computation is set to restart and the latest values are available to clients when they call.

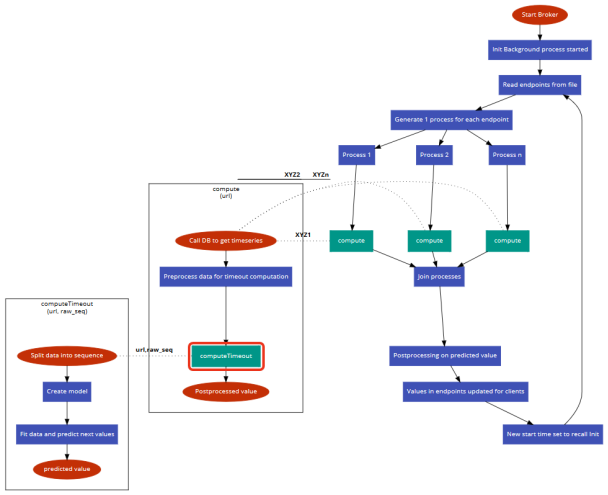

Figure 4: MultiProcess broker code flow diagram

Advantages of MultiProcess approach over a single process approach:

- Makes better use of multi core CPUs and GPUs as no single process can acquire full CPU
- Makes it easier to design architectures where individual processes are moved to different machines
- Processes have their own memory space unlike threads which means that there is no accidental overwrite in memory
- It does have a larger memory footprint than a single process or multi threaded architecture but it is the first step towards a distributed system.

MultiDocker:
Newly added interfaces to support MultiDocker:
This kind of an approach requires workers to register and unregister with the broker at will. So to set this up, we have added 2 new interfaces **/register** and **/unregister**. It is a call that does not require any parameter and the caller's IP will be added as a worker.

Sample Usage:
Register: http://<SERVER_IP>:<PORT>/register **Return:** None
Unregister: http://<SERVER_IP>:<PORT>/unregister **Return:** None

The responsibilities of the broker change considerably in this configuration. The broker is still responsible for providing the RestAPI that the client calls for getting the timeout values, but it is no longer responsible for the computations. The flow diagram can be found in Figure 5

(1) Read endpoint.config and get the endpoints for which to configure timeouts.
(2) Worker systems can register with the broker using the newly designed **/register** interface.
(3) Spawn processes for each worker registered and call them for timeout computation.
(4) Coordinate between these calls and ensure that they respond with the required values and some sanity checking.
(5) The main process does some postprocessing on the predicted values and updates the dictionary with the new values.
(6) At this point the computation is set to restart and the latest values are available to clients when they call.

Figure 5: MultiDocker broker code flow diagram

The responsibilities of the worker on the other hand change considerably as that has become a microservice with it's own role and with it's own API. The flow diagram can be found in Figure 6

(1) On startup, it registers itself with the server by calling the broker's register endpoint.
(2) When the broker calls it's API for timeout computation, it gets a url endpoint with it.
(3) It then calls the Influx database to get the raw timeout data for their endpoint.
(4) It will also preprocesses the data and create a ConvLSTM RNN model.
(5) The model is then trained with this data and next value predicted.

(6) This value is then returned in it's raw nature for the broker to process.

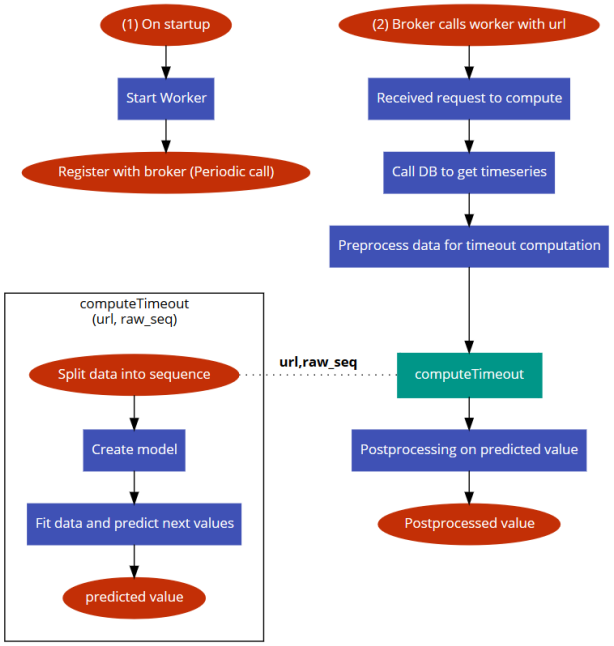

Figure 6: MultiDocker worker code flow diagram

Advantages of Mutidocker approach over single docker approach:

- Each docker can be moved to it's own individual machine
- Much faster with the whole CPU behind each process
- Granted the network calls add a overhead and wait time needs to be introduced with fallbacks in case of failures, but when created correctly, it generally gives a better performance and makes the system robust.

3.4.6 Performance.

An important metric when it comes to evaluating a any machine learning model is the shape of the loss curves. As can be seen from Figure 7 this curve is an exponential decay curve which means that the model is learning at every epoch and is constantly trying to improve its predictions. This means that with more epochs, the predictions will improve until it reaches a plateau on the curve at which time it will be performing at its best.

As expected the MultiProcess mode worked better than MultiDocker mode when using a single server. This is clearly visible in Figure 8 which compares the runtimes of the two. When we scaled up though, we immediately see the performance benefits of take over. In this approach, we can get max performance when we have one pod per url endpoint. So at n=5, we see the final drop after which the graph stabilizes and any more pod additions are just for backup and redundancy.

3.5 Gaussian Process Data Broker

3.5.1 Technology Stack.

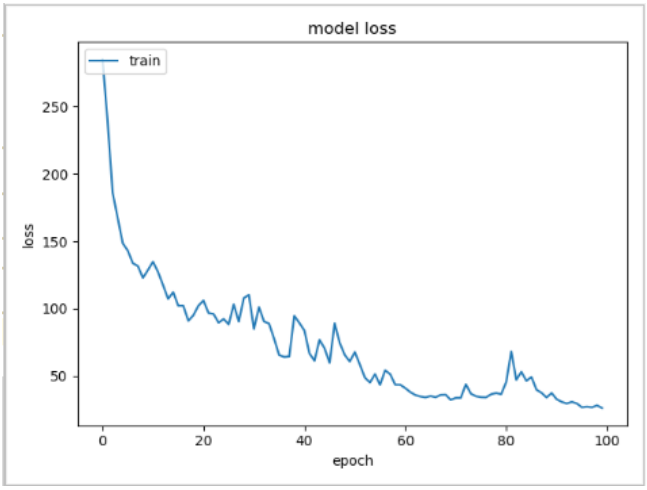

Figure 7: Sample Model loss (MSE) curve

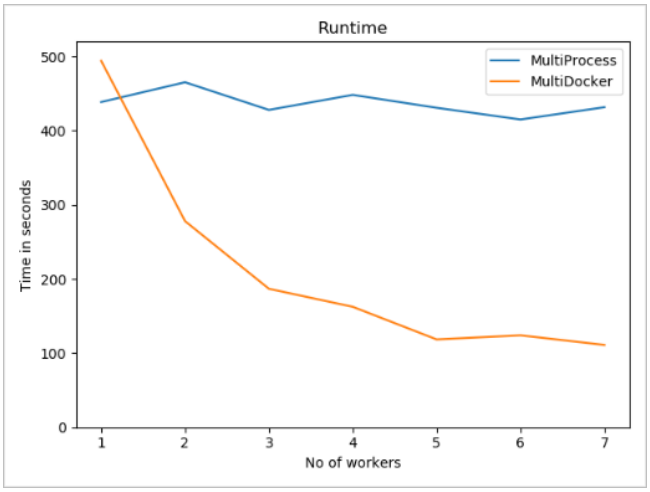

Figure 8: Runtime in two modes

1) Julia/Gen

Gen [4] is a flexible probabilistic programming system in the Julia language. Gen includes an example implementation of a Gaussian Process inference technique as described in [23]. We took the example code and modified it for use in our timeout prediction task.

3) InfluxDB

InfluxDB is an open-source distributed time series database. We are using it to collect and store the Jaeger tracing data, which includes response times. To speed development of the the Gaussian Process predictor, we have utilized InfluxDB's Continuous Query [16] feature to down-sample and regularize the raw data. After some experimentation, we settled on a 20 second interval as the granularity of the Continuous Query. Since our model is currently being trained online with each request, we wanted to balance our time-to-predict (fewer training data-points) with our ability to quickly react to changing response times (smaller time intervals) in the underlying system. The continuous query we are using is thus:

```
CREATE CONTINUOUS QUERY "cq_20s_max" ON "tracing"
BEGIN
SELECT max("duration") INTO "max_duration" FROM "span"
GROUP BY time(20s), service_name
END
```

This query produces a new entry every twenty seconds for each unique service being monitored dynamically (e.g. auth, customer, booking, flight) and the value is the max response time over the 20 second interval. Since our desire is to propose a useful timeout, aggregating using the maximum value is a natural way to use only the slowest responses recorded by the system.

4) Server API

The Gaussian Process predictor has the same primary endpoint "gettimeout" as the RNN predictor. This endpoint takes the url that is about to be called by the client as a parameter. In response, the predictor returns the predicted/suggested timeout value for the provided url. The Gaussian process predictor does not currently offer getEndpoints as the RNN predictor does. Eventually, we would expect to dynamically build this list from observed traffic and not require pre-configuration.

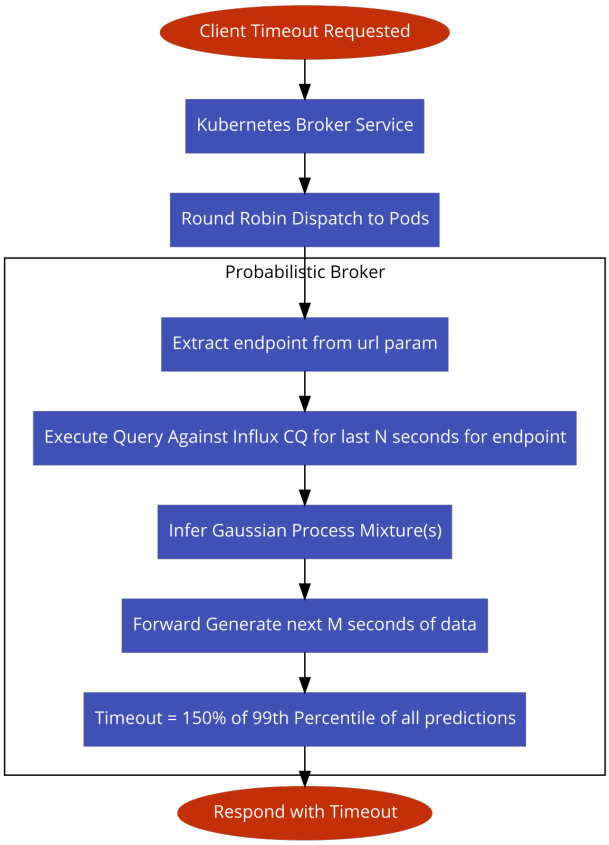

Figure 9: Gaussian Process Predictor Workflow

3.5.2 Gaussian Process Model Development. Following the initial implementation of the GP broker as shown in figure 9, we went back to study the various parameters available in the inference and

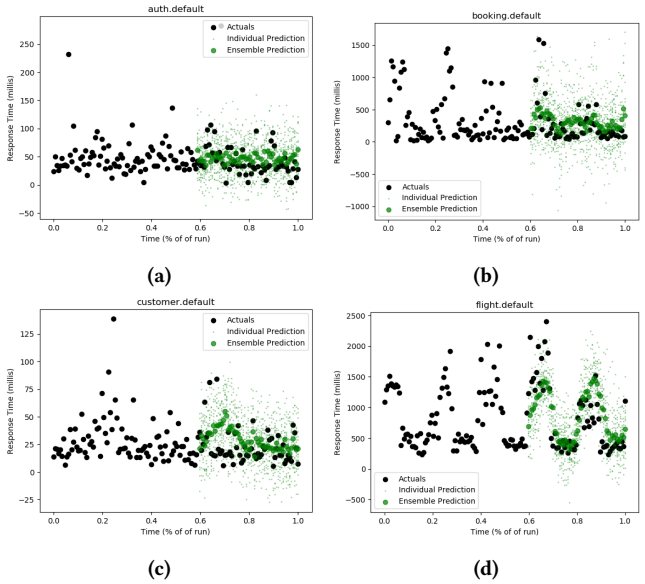

Figure 10: Example of Gaussian Process model development using High-Low cyclical scenario. Black points are actual data. Small green points are ensemble member predictions while large green points are ensemble predictions (Ensemble mean, 20 members, 1000 iterations per member).

prediction steps to evaluate the performance over a range of parameter values. In a future implementation, we would like to train regularly, but not with every call, and use the predictions over the next M seconds rather than our naive approach to train with every call to the service. To enable this experimentation, we developed three different system traces run via our JMeter load generation setup. The scenarios are "high-low cyclical", "long ramp" and "random". We executed each scenario and captured the timeout history for each. We then used them to train and test various parameter settings to understand performance characteristics of our approach. Figures 10, 12 and 11 show selected examples of the output from these scenarios. The Gaussian process method largely delivers on its promise of accurately predicting time series data. For example, in figure 10d we see a faithful reproduction of the cyclical nature of the training data. On the other hand, in figure 12d where the method is trying to reflect the steep slope at the beginning of the training data which leads to consistent under-prediction. In addition to the need for rigorous parameter tuning and further experimentation, we offer some ideas for improvement in the Future Work section.

4 EXPERIMENTAL EVALUATION METHODOLOGY

4.1 Acme Air Model System

To evaluate the full system we chose a model system called Acme Air as our subject. Acme Air [27] is a micro-service flight booking system in the spirit of Expedia. Customers login, search and book flights and can view and edit their profiles. In addition to the client web site, there are four main micro-services that make Acme Air

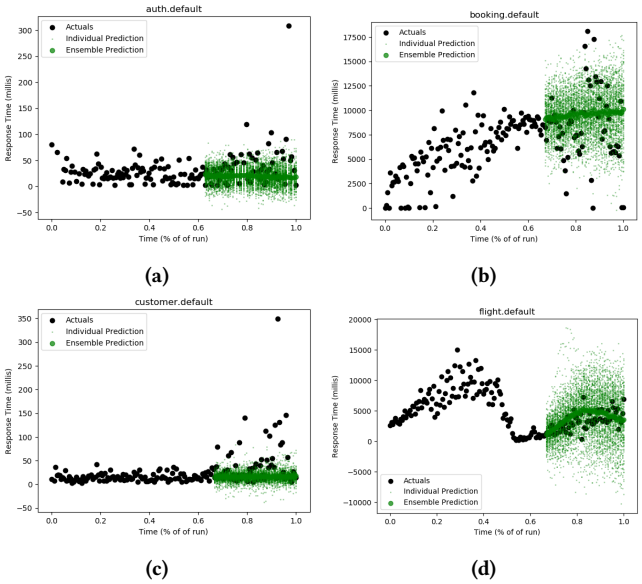

Figure 11: Example of Gaussian Process model development using Long Ramp scenario. Black points are actual data. Small green points are ensemble member predictions while large green points are ensemble predictions (Ensemble mean, 100 members, 100 iterations per member).

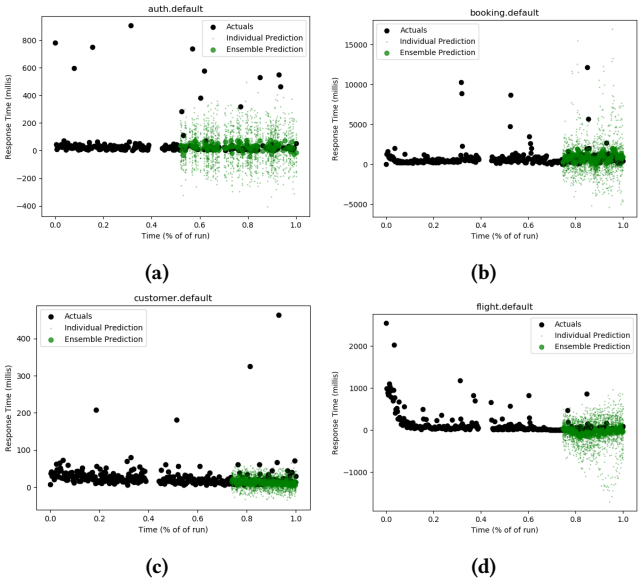

Figure 12: Example of Gaussian Process model development using Random scenario. Black points are actual data. Small green points are ensemble member predictions while large green points are ensemble predictions (Ensemble mean, 20 members, 1000 iterations per member).

work. They are "auth" for authentication, "booking" for lookup and creation of flight bookings, "customer" for customer profile information and "flight" for searching available flights. Each of the four micro-services is backed by a separate Mongo DB instance. To drive load to Acme Air, we used a JMeter script that simulates customer activity.

4.2 Full System Evaluation Method

To evaluate the integrated system, we initialized the Acme Air database using the configure/load database page on the website. We then started up a low-load JMeter script and started the brokers to allow them to warm up. The final step was to start the Acme-Air Java Client, running with the Client Shim attached. Once everything was running, we started the main JMeter script that simulated various load scenarios on the system. While the script was running, the Client Shim measured response times of the microservices and asked the broker's for their predictions. All data were recorded and analyzed after the run. The primary analysis consisted of comparing the predicted timeout with the actual response time with a focus on under-prediction as it guarantees timeout failures while even moderate over-prediction is essentially harmless. It should be noted, the YClient sampling method is relatively infrequent compared to the JMeter load being run. Thus, our sampling method may not uncover low-frequency events such as very long microservice calls. In comparison, the Brokers will see those events and make adjustments to predictions as necessary. Our final performance assessments are currently using the YClient samples and may be misleading. In the future, we would pull the raw Influx timeouts and use them to more fully assess our predictions.

4.3 Evaluating the portability of the library as a runtime

To show that the library we built with Aspect Oriented Programming can satisfy our claims of actually being portable, and getting the injection functionality activated in any given Java application without the need to rebuild or change any source of the application willing to adopt dynamic timeout injection, we created a simple Java application which would just fetch the flight data from the service. This application did nothing but open a connection, read the value and exit.

Without the library, the code would just print out the response received from the service. With the library the expected output should contain the call to our broker and the average latency to the host, and gets printed in the console.

5 RESULTS

5.1 Portability of the Client Shim

When we ran the above application along with the parameters mentioned in the implementation section, we saw that the task was successful. Only by adding the following parameters and setting the environment variable *injecttimeout=true*, we were able to get the application working with dynamic timeout injection.

-Xbootclasspath/p:<path to the shim jar> -DbrokerURL=<broker base URL>

5.2 Overhead of the Client Shim

The beauty of this shim is that it gets embedded deep inside the JRE. The major overhead of the shim is to compile the JRE, which could take up to a minute, but when using the output jar with any application, the runtime overhead is in nanoseconds. We see in Figure 13 how the client is able to adapt to the changes in service load.

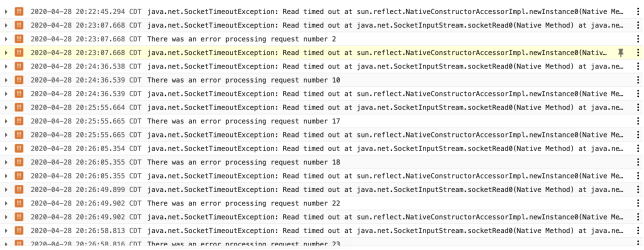

Figure 13: Auto recovery of the client

5.3 Broker Prediction Results

The system, overall, worked well. We had mixed success with our prediction techniques, somewhat dependent on the microservice endpoint. As expected, our predictions do not do so well in the start as these models are still learning the intricacies of the data. Once they get enough training data in place, their values improve considerably. Table 1 shows our early results. Surprisingly, both techniques averaged 22% under-prediction across all endpoints. The RNN performance was more consistent while the GP did fairly well on two endpoints and did poorly on the other two. The table also includes the RMSE split by under vs over prediction to help get a sense of the magnitude of the error.

Table 2 presents an updated result for the GP Broker on a similar run "cyclical", as Table 1. With these updated, performance has improved radically. Under prediction has been driven to below 1% on average while over-prediction RMSE has risen to only four seconds on average. See figure 15 for a plot of the run.

6 RELATED WORK

In this section, we discuss related work with a focus on describing the difference between our approach and previous approaches:

6.1 Similar Systems

Our proposal extends the ideas presented in [13] and attempts to apply estimated timeouts into direct use in the running system. Our method of timeout derivation is different. We are not deriving timeouts from system calls but rather measuring actual network call duration using a set of distributed proxies in front of all services offered by the target system. We did not focus on finding timeout bugs, but rather implemented a way to enable adaptive timeouts to hopefully avoid bugs and improve overall performance and robustness in the face dynamic system behaviors.

Aspects JustInTimeout are similar to a commercial feature in an ad-buying platform called Adaptive Timeout by IndexExchange. From their website:

> "Today, all wrappers available to Publishers use a fixed-value timeout, and most Publishers use a single fixed timeout for all of their header bidding supply, regardless of factors such as device type or the user's network conditions.... With Adaptive Timeout, our Publishers now have an adaptive machine learning feature that accounts for these variables.... Static timeouts led to a trade off, whereby publishers had to sacrifice user experience to wait longer for header bidding, or sacrifice revenue by setting aggressive timeouts."

While the marketing language claims machine learning, digging deeper we found the claims of machine learning for optimizing timeouts to be overstated. They are, in fact, using a basic statistical approach as we initially proposed using mean and standard deviation to pick a safe timeout. It appears they compute a few static timeouts based on a few coarse grained timeouts based on network quality (low, medium, high) and device type (phone, computer) and use those instead of static timeouts. In [8], Dixit et al. proposes the use of "Adaptare"[7] which attempts to match system behavior to any number of underlying statistical distributions. Our work is similar, but we designed a pluggable framework where we can test various and perhaps learning and prediction techniques, including Gaussian mixture models and Machine Learning models. The question of whether to use a machine learning model or a pure statistical model for our prediction is still an open question for us at this point. Makridakis et al[19]. provides some really good insights on this subject as to which would suit the data set better.

In [18] Mace, et. al. describe a feature-rich distributed tracing framework that can dynamically add tracepoints to a running system using AOP, can correlate trace events across the system and assist in root cause analysis. Our system does far less, but we will use similar techniques in our Istio/Jaeger to then actively modify client behavior to improve it in real-time.

6.2 Related Technologies

The sidecar pattern as proposed in [2] will be used to measure the current timeout data from services hosted on Kubernetes pods and relayed to the performance collectors. The decision of using this pattern is for the simplicity it provides and the fact that we can make the code independent of any particular service.

In their paper, Cleveland et al.[3] focus mainly on using AspectJ to inject faults and weave in assessment criteria to assess the survivability of critical distributed systems. In our approach, we plan to use aspect oriented programming to inject the timeouts we compute into the client to improve future calls. As we wish to build a highly fault tolerant system, we decided to make use of Aspect oriented programming based on the results presented by Alexandersson et al.[1]. In their paper they concluded that having a AspectC++ based system which is very similar to AspectJ has a much higher fault coverage over C based systems especially with high optimizations.

7 FUTURE WORK

While we were successful in dynamically computing timeouts based on tracing data, we have just scratched the surface of what might be possible. Specifically, we would like to study the overhead of

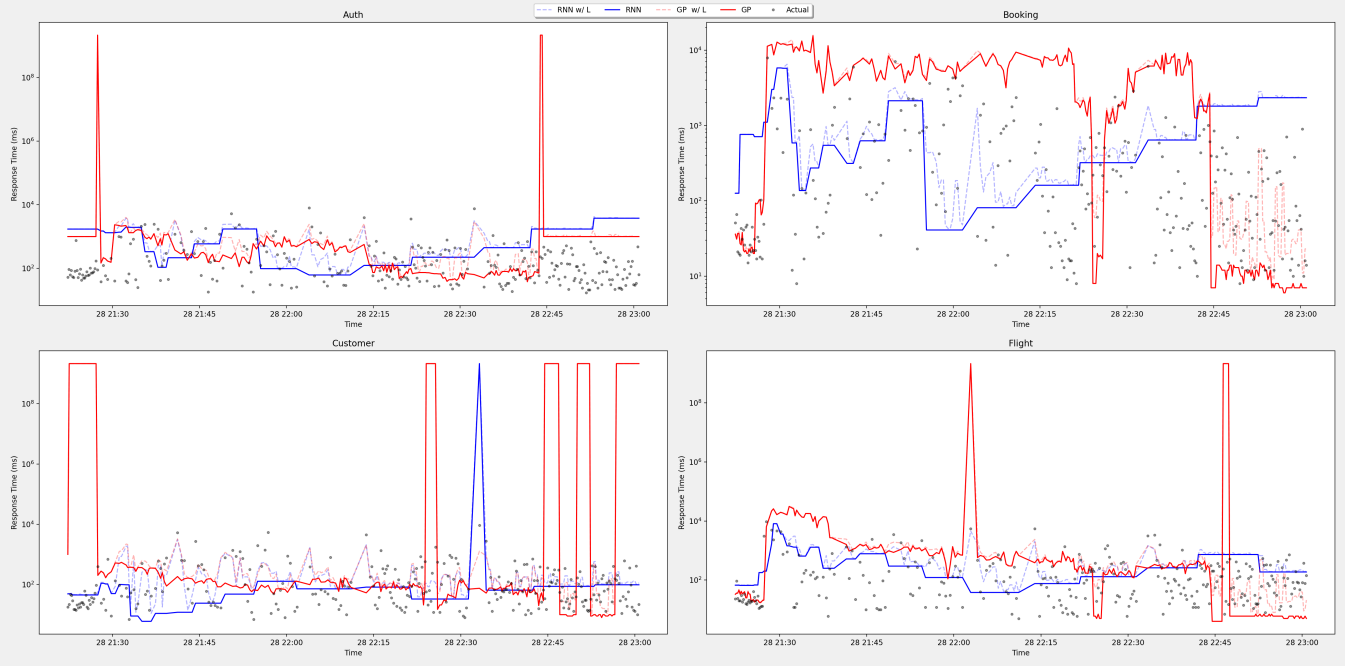

Figure 14: Example System Run

Sample run of about 90 minutes. Actual measured response times are the black points vs prediction methods as lines. The GP Broker suffered a technical failure after 22:45 which was removed during the analysis.

Table 1: Performance Metrics of Each Technique by Endpoint

	RNN			GP v.1		
Endpoint	Under-predict %	RMSE-Under (millis)	RMSE-Over (millis)	Under-predict %	RMSE-Under (millis)	RMSE-Over (millis)
auth	17%	1462	1808	30%	1253	820
customer	32%	1277	257	33%	1114	457
flight	19%	390	26	15%	547	5
booking	21%	1978	1455	11%	209	5656

Table 2: Updated Performance for GP Broker v.2

	GP v.2		
Endpoint	Under-predict %	RMSE-Under (millis)	RMSE-Over (millis)
auth	1.3%	1195	3558
customer	1%	170	3819
flight	0.4%	10	2762
booking	0.0%	NA	5471

our collection and prediction systems to aid in understanding how much overhead the system adds and under what circumstance we would recommend its use. Additionally, we would like the broker/predictors to be able to increase granularity and dynamically partition network calls based on url and payload. We would also like to extend beyond REST-based network calls to make the system more widely applicable. It would also be informative if we could train and test on more real-world workloads that extend beyond 1-2 hours. This will allow us to study longer-range predictions and assess and improve our prediction accuracy. Finally, we believe our tracing and prediction system could be utilized in concert with CPU/Memory profilers to better assess when a resource needs to be horizontally or vertically scaled. In particular, the promise of change-point detection in the Gaussian Process scheme is particularly interesting as this is a specific indicator that the underlying system underwent a drastic change in behavior and might need more (or less) resources. In addition, we could use the Istio/Jaeger data to identify system crashes and perform some operation to help the clients.

8 CONCLUSION

In this paper we have presented JustInTimeout which Enacts Dynamic Network Timeouts via Distributed Feedback. We had a lot of fun learning about new technologies and solving the various challenges we faced while implementing this project. Creating a

distributed system gave us in-depth knowledge of the various challenges that these systems have and how to design efficient systems to solve them.

Our system is aimed at solving the age old problem of what should a good timeout look like. This system dynamically computes timeouts based on recent state of the system and injects them into client applications with the hopes of fixing the problem of static timeouts being too small or too large. While this is still a very nascent approach, we believe we have laid the foundations on which work can be done to improve our solution.

REFERENCES

[1] Ruben Alexandersson and Johan Karlsson. 2011. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. 303–314. https://doi.org/10.1109/DSN.2011.5958244

[2] Brendan Burns and David Oppenheimer. 2016. Design Patterns for Container-based Distributed Systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

[3] Jeffrey Cleveland, Joseph Loyall, and James Hanna. 2014. An Aspect-Oriented Approach to Assessing Fault Tolerance. *Proceedings - IEEE Military Communications Conference MILCOM*, 1374–1381. https://doi.org/10.1109/MILCOM.2014.228

[4] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642

[5] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding Real-World Timeout Problems in Cloud Server Systems. 1–11. https://doi.org/10.1109/IC2E.2018.00022

[6] Influx Data. 2016. *Influx Data Getting started with Influx for Python.* https://www.influxdata.com/blog/getting-started-python-influxdb

[7] Mônica Dixit, António Casimiro, Paolo Lollini, Andrea Bondavalli, and Paulo Verissimo. 2012. Adaptare: Supporting Automatic and Dependable Adaptation in Dynamic Environments. *ACM Trans. Auton. Adapt. Syst.* 7, 2, Article Article 18 (July 2012), 25 pages. https://doi.org/10.1145/2240166.2240168

[8] Mônica Dixit, Henrique Moniz, and Antonio Casimiro. 2012. Timeout-based adaptive consensus: Improving performance through adaptation. (03 2012). https://doi.org/10.1145/2245276.2245371

[9] Eclipse. AspectJ. *Pointcuts.* https://www.eclipse.org/aspectj/doc/next/progguide/semantics-pointcuts.html

[10] Eclipse. AspectJ Runtime. *org.aspectj.aspectjrt:1.8.13.* https://www.eclipse.org/aspectj/

[11] Envoy. 2020. Envoy Proxy. https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy

[12] Google. Guava. *com.google.guava:28.2-jre.* https://guava.dev

[13] J. He, T. Dai, and X. Gu. 2018. TScope: Automatic Timeout Bug Identification for Server Systems. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*. 1–10. https://doi.org/10.1109/ICAC.2018.00010

[14] Jingzhu He, Ting Dai, and Xiaohui Gu. 2019. TFix: Automatic Timeout Bug Fixing in Production Server Systems. 612–623. https://doi.org/10.1109/ICDCS.2019.00067

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735 arXiv:https://doi.org/10.1162/neco.1997.9.8.1735

[16] Influx. 2020. InfluxQL Continuous Queries. https://docs.influxdata.com/influxdb/v1.8/query_language/continuous_queries/

[17] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:cs.LG/1412.6980

[18] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2018. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)* 35, 4 (2018), 1–28.

[19] Spyros Makridakis, Evangelos Spiliotis, and Vassilis Assimakopoulos. 2018. Statistical and Machine Learning forecasting methods: Concerns and ways forward. *PLoS ONE* 13 (03 2018). https://doi.org/10.1371/journal.pone.0194889

[20] Medium.com. 2017. *Why Keras.* https://medium.com/implodinggradients/tensorflow-or-keras-which-one-should-i-learn-5dd7fa3f9ca0

[21] Oracle. Java Runtime. *1.8.0_241.* https://www.java.com

[22] J. Postel. 1981. RFC0792: Internet Control Message Protocol. https://doi.org/10.17487/RFC0792

[23] Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. 2019. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 37 (Jan. 2019), 29 pages.

[24] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai kin Wong, and Wang chun Woo. 2015. Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting. arXiv:cs.CV/1506.04214

[25] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[26] Tensorflow. 2018. *Tensorflow Time series forecasting.* https://www.tensorflow.org/tutorials/structured_data/time_series

[27] T. Ueda, T. Nakaike, and M. Ohara. 2016. Workload characterization for microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10.

Appendix A GP V.2 UPDATED PLOT

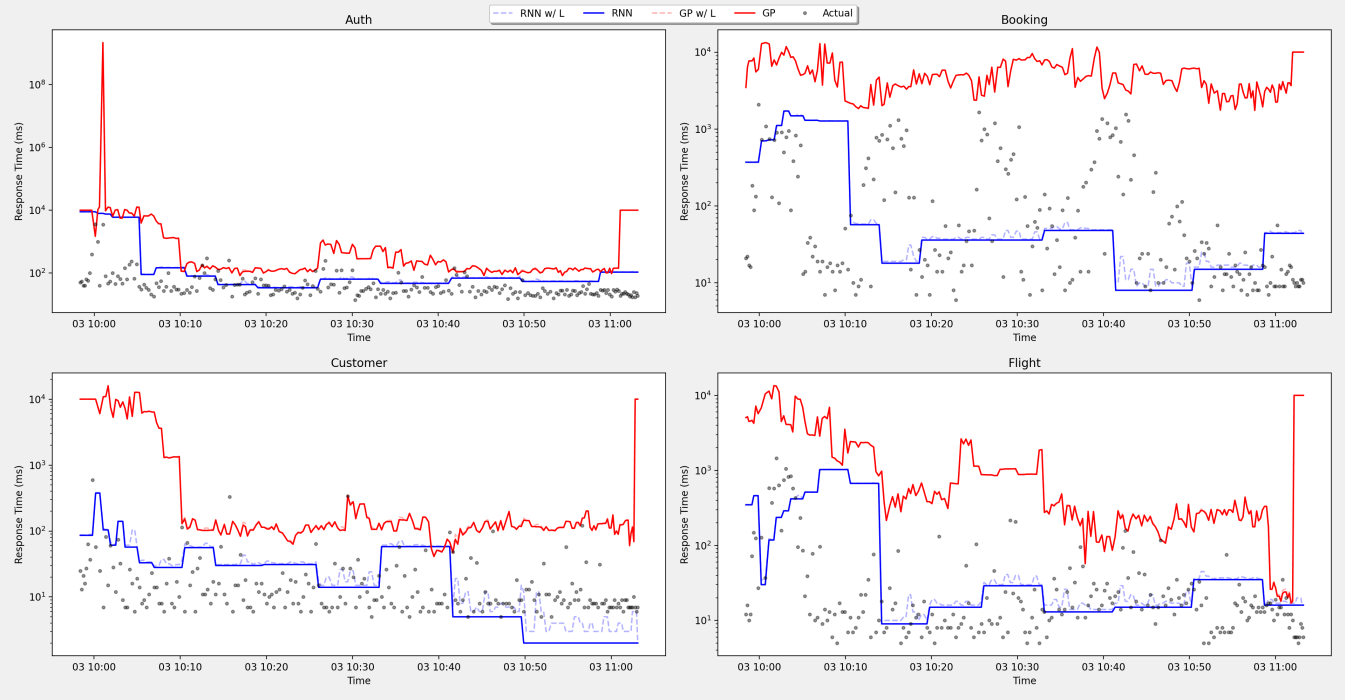

Figure 15: Cyclical Run 02 using GP Broker v.2

Sample run of about 60 minutes. Actual measured response times are the black points vs prediction methods as lines. This plot demonstrates the improvements made to the GP Broker as compared to the results in table 1 and figure 14.