Taskmaster

Jotham Wong
jw0771

Chai Mauliola
mauliola

December 15, 2023

1 Motivation

Serverless computing or the Functions-as-a-Service (FaaS) paradigm is a rapidly growing cloud application model that sees popularity among startups and small-mid scale companies due to its advantage in blackboxing architectural constraints. However, it is plagued by a problem known as cold starts where time required to start up containers with their dependencies dominates. Chris Munns proposed that "pinging" a serverless function was really the only way to mitigate the issue of cold starts [1]. We therefore design and build Taskmaster, a system that allows plugging in custom pinging strategies to investigate its effectiveness.

2 Methodology

2.1 Architecture

The architecture of Taskmaster can be seen in Figure 2.1. Taskmaster essentially consists of three components:

1. Gateway
2. Strategy
3. Openwhisk (FaaS framework)

The FaaS framework used can be replaced by another FaaS framework since all FaaS frameworks have a cli tool available for creating and invoking actions, a simple modification can be made to the existing codebase to handle them appropriately. Openwhisk was chosen due to ease of use, it's architecture making use of docker containers which suffer from cold starts, exactly the problem we aim to investigate. From here onwards, we refer to Openwhisk as FaaS to make it clear that we don't rely on any Openwhisk specificities for Taskmaster's usage.

The Gateway is a standard HTTP server that sits between the user and the FaaS framework. Users invoke serverless function requests using a HTTP get request to the Taskmaster through the "/receive" route with their specified function name and parameters. The Gateway will route these requests to the FaaS framework and return the results if specified. The Gateway will also periodically poll the Strategy component at a specified interval (through the configuration file) and update the Strategy of function invocation patterns.

The Strategy is implemented as an interface that supports two methods: Update and Predict. In Update, it receives a key-value map of information that it can possibly use to update its internal state. When the Strategy is periodically polled by the Gateway, it will return a function name and parameters to the Gateway for "pinging" as mentioned by Chris Munns through its Predict method. Both Update and Predict are implementation dependent and any Strategy that satisfies this interface can be swapped by specifying the strategy in the configuration.

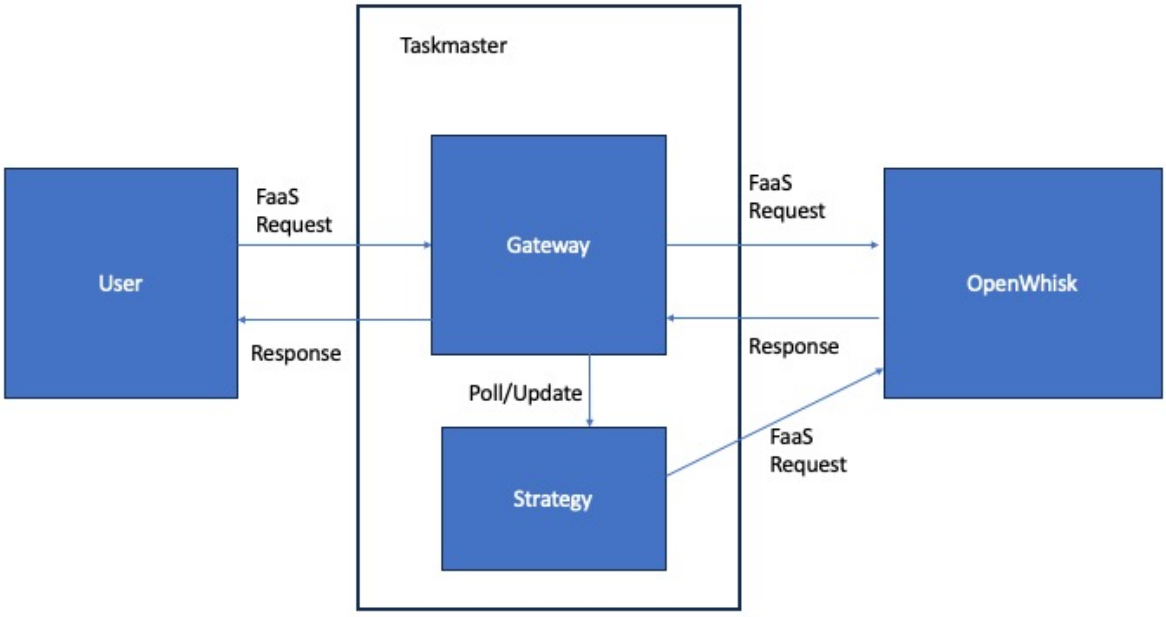

Figure 1: Taskmaster architecture

2.2 Workloads

To test our system, we artifically generate function request workloads using the python script "generator.py". Several probability distributions can be used to represent different patterns of workloads but we currently use the Gaussian distribution with a specified mean and variance to control the intervals at which the functions are invoked as well as a uniform distribution for the function calls. A wide variety of functions can also be used as the testbed functions. The scripts are easily extensible to support testing on a wide range of scenarios.

2.3 Strategy

In this section, we detail the different strategies that were employed and provide experimental results in the next section.

2.3.1 Naive

We employ a periodicity of 0 which means that the Strategy is never called. This serves as our experimental baseline. Although this does incur some penalty due to the additional HTTP call that needs to be made, we assume that the delay is negligible in the context of containers shutting down with respect to the function invocation workload.

2.3.2 LRU

This Strategy resembles the LRU cache under the following intuition: a function container that was least recently used is most likely to shut down and subsequent function calls will incur the cold start penalty. Therefore, a simple linked list is used to maintain the LRU order and Update will update the internal LRU ordering. When the Strategy is polled, it will ping the lru function and move it back to the front of the queue.

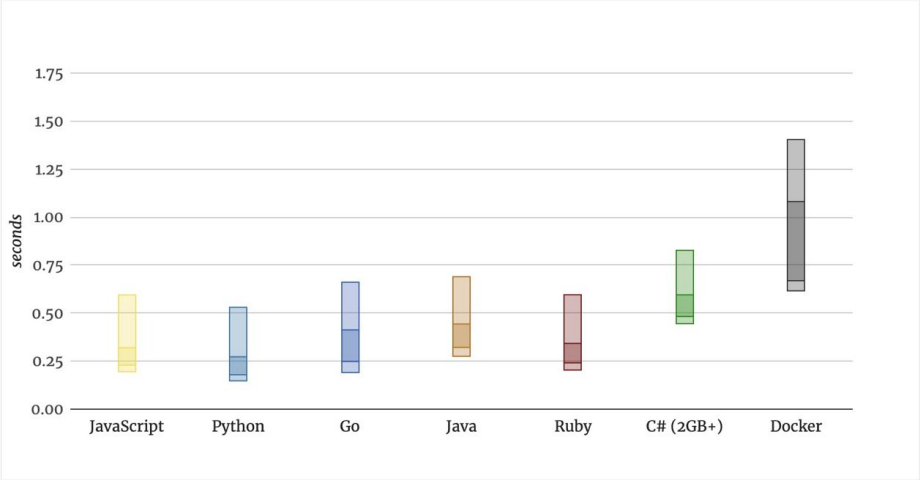

Figure 2: Figure obtained from Mikhail Shilkov

2.3.3 MRU

This strategy is the same as the LRU strategy, except that the most recently used function will be kept alive. This strategy is the opposite of the LRU and intuitively does not make sense since the most recently used function should not be pinged again but we run experiments on it to compare against the LRU.

2.3.4 PQ

We observe that different programming languages incur different penalties due to cold starts as seen in Figure2. We therefore want to minimize the penalties due to the worst case, Java, by assigning constant weights to the different programming languages proportional to their penalties in the figure, and ping the highest priority containers with a max heap. In this way, containers with lower penalties will incur the cold start penalty but the heavier penalties are less likely to be cold.

2.3.5 MFE

Maintain a counter for each function call and then keep alive the Most Frequently Encountered (MFE) function. With this strategy, we aim to make the common (most frequent) case fast which is a common design pattern in computer architecture.

3 Experimental Results

For evaluation, we used the programming languages Java, Javascript, Php, Ruby and Python as these were supported out of the box from Openwhisk. Simple functions that either had O(1) time complexity (the suite of Hello functions) as well as O(N) time complexity (the suite of factorial functions) were used to introduce some variance in the time it would take for the function completion. Golang was excluded from our testing suite as we ran into issues getting the go binary to execute on the serverless container. Instructions for setting up the workloads can be found in the README.

For each strategy in 2.3, we ran the simulation with polling periodicity set to 5, 10, 15 seconds and measured the number of cold, prewarmed, warmed and recreated container states using Openwhisk's logs. We initially wanted to evaluate on a polling periodicity of 1 second but noticed that it was overwhelming the FaaS framework on our machine and causing it to return multiple failures, therefore scrapping it. We also ran an experiment with polling periodicity set to 0 to reflect the baseline wherein no strategy is used. The experimental results can be seen below.

Strat	Period	Cold	Warm	Prewarmed	Recreated	Java	py	Js	rb	php
	5	422	119	156	303	11294.4	1252.22	238.653	5288.07	928.055
lru	10	356	151	149	344	10325.2	1107.04	209.56	4574.66	838.232
	15	342	148	146	364	12091.7	1242.46	278.87	4849.01	900.547
	5	216	212	157	415	10856.6	646.117	252.944	4615.13	892.022
mfe	10	240	213	145	402	10981.5	706.737	559.838	4907.16	966.254
	15	224	208	146	422	11648.2	843.991	294.625	5412.53	1020.82
	5	352	111	163	374	11371.9	1485.46	253.866	5023.53	912.354
mru	10	367	134	153	346	10373.3	1426.57	251.255	4754.13	1536.23
	15	335	139	150	374	11665	2537.04	536.574	5763.95	1567.71
	5	355	122	158	365	10750.1	1188.42	257.181	4833.99	1212.46
pq	10	386	135	163	316	10764.6	1163.77	211.574	4529.99	838.945
	15	385	140	160	315	11548.2	1279.84	286.301	5384.35	914.873
naive	5	301	200	128	371	11798.6	1099.46	497.421	4662.8	1187.24

Table 1: Experimental Results

We ran all experiments on the test workload file in the directory which used a uniform distribution on the 10 functions as well as a gaussian distribution with mean 5 and variance 1 for the delay between each successive function invocation in seconds. These formed 1000 function calls over a duration of 84 minutes to reflect slightly more realistic serverless use cases. The distribution of the simulated workload can be found in "Action counts.png" as well as Figure 4.

Our experimental metrics of interest are the number of recreated, cold, warm and prewarmed containers. Recreated and cold containers incur cold start penalties as the containers require some time to set up before they are ready to perform the job whereas warm and prewarmed containers reflect containers that are cached and ready to execute immediately. We also measure the average latencies (in milliseconds) for the function invocations per programming language which further act as a proxy for the impact of the cold or warm container states.

Our results can be seen in Table 1 and we can see the distribution of container states in more detail in Figure 3. Unfortunately, only the MFE strategy appears effective, succeeding in reducing the number of cold start containers compared to our baseline "Naive" at the expense of some recreated containers. We observe that the average latency of the python containers is significantly reduced in the MFE strategy compared to all other strategies which aligns with our distribution where the python language has the most number of occurrences.

Shockingly, the MRU strategy performs about as well as LRU implying that our intuition does not hold for this specific distribution of workload. Furthermore, the PQ strategy does not reduce the average latency of the Java language significantly even though it mostly only pings the Java and Php language as seen in 4 at the expense of the Python containers. Given the negative impact that a polling periodicity of 1 had on the standalone Openwhisk's server ability to handle incoming requests, we hypothesize that the reason the other strategies fail to benefit the workload is because the access pattern does not align with the function sampling and cause Openwhisk to allocate more resources to the wrong containers instead of the actual demand, thereby reducing the number of compute resources available. This seems to support why MFE works so well for this particular workload as there is more legitimate demand for the Python containers and therefore keeping them alive aligns nicely for the simulation. Evidently, the optimal strategy depends on the workload and further investigation is warranted.

Finally, we observe that there appears to be a non-linear relationship between the polling periodicity and the performance of the system. A 10 second polling periodicity appears to be most optimal with respect to the average latency of the various programming languages. Further work should be done to study this in more detail.

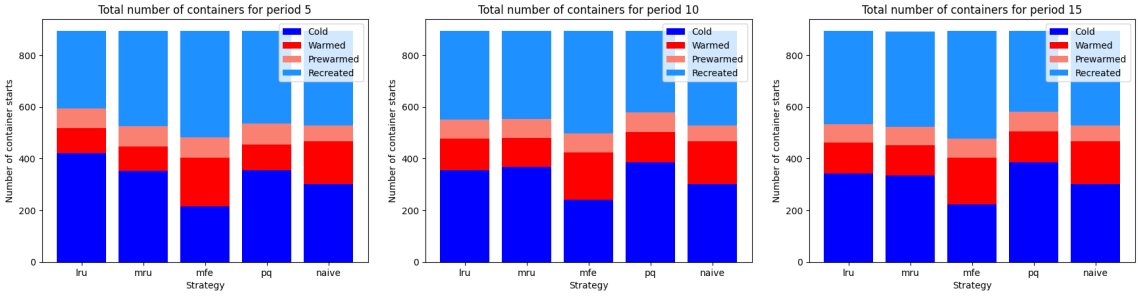

Figure 3: Stacked bar plots of container state

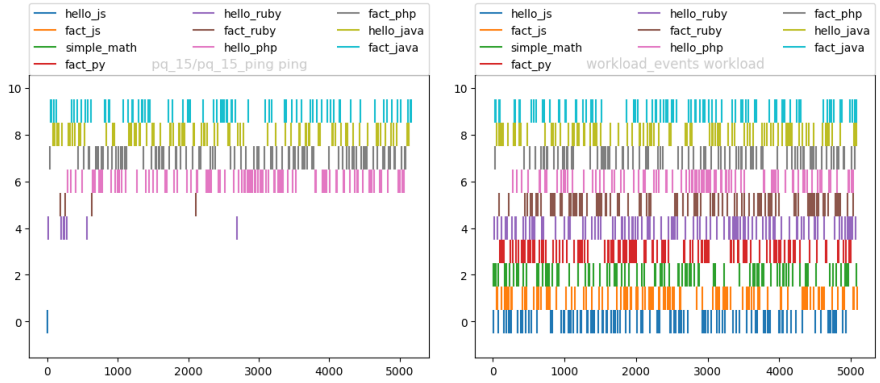

Figure 4: Timeline of PQ pinging vs Workload

4 Conclusion and Further work

In this research project, we have investigated the use of "pinging" serverless functions in an attempt to remedy the cold start problem and found the MFE strategy most effective for a uniformly distributed sample of functions following a gaussian distribution of delay.

Due to time limits, we were unable to add more complicated strategies that we initially had in mind, such as a ML forecasting approach that would train on the previous day's workload and use a Flask server as the endpoint for the prediction. We would also have explored a longer workload file with a larger suite of serverless functions with more complex dependencies per container lasting a full day instead of the current runtime of 1 hour 20 minutes for more realism in the experiments. We would also like to further explore the relationship between the polling periodicity and the impact it has on the performance of our system.

5 Acknowledgements

Thanks to Yue Tan, Yiwei Dai and Leo Chen for their feedback.

References

[1] Daly, J. Lambda Warmer: Optimize AWS Lambda Function Cold Starts. `https://www.jeremydaly.com/lambda-warmer-optimize-aws-lambda-function-cold-starts/`. Accessed: 2023-12-02.