

XX: A system

Anonymous Author(s)

Abstract

The promise of serverless currently remains an elusive and extremely attractive paradigm: get what you need as you need it, but only pay for what you use.

For example, building an elastic web server on top of a serverless infrastructure is not something that can be done easily today, but is in principle a good candidate for the serverless setting: its load is unpredictable and bursty, and it does not require managing its data locally.

Many challenges remain in making the ideal of serverless a reality; XX tackles scheduling: in a world where a large and heterogeneous set of jobs is being run on as serverless functions, there has to be a way to differentiate between functions' requirements and urgency.

XX is a distributed scheduler that allows developers to express priorities and enforces them. Developers express priorities to XX via assigning functions to fixed dollar amounts per unit of compute, and cap the overall usage by specifying a monthly budget. Memory costs per unit time used are the same across all priorities, and developers specify a maximum amount of memory for each function.
[hmng: Do I need the memory blurb? I added it because I mention memory below] XX places incoming jobs on machines that have memory available, and implements a machine scheduler that enforces priorities.

[hmng: TODO impl/eval blurb]

1 Introduction

A world where all cloud compute is run in the format of a serverless job is attractive to developers and providers: developers only pay for what they use while having access to lots of resources; and cloud providers have flexibility in scheduling and can achieve good utilization.

This paper focuses on building a usable and efficient priority mechanism for a world of universal serverless, and assumes that cold start times are small enough to be acceptable on the critical path — recent state of the art systems have been able to support cold start times in the single digit ms range[1], and we expect this trend to continue.

A driving example for this work is that of a web server. Its characteristics of unpredictable and bursty traffic make it well-suited for serverless, and the pay as you go model is particularly attractive to website developers who don't want to have to worry about provisioning.

Suppose a simple web server consists of two different page views, one for the landing page and another for users' profile pages, and also has map reduce jobs for data processing.

If the web developer wanted to host the web server in an existing serverless offering, a popular option is AWS lambda[1]. In order to influence the ways that different functions scale, AWS lets developers specify their desired 'reserved concurrency', ie the number of warm containers that are specifically reserved for a given lambda[1]. However, with that interface, the developer loses the benefit of the flexibility that was the initial draw: they are now paying for resources they are not using (because they have to pay to keep all those containers warm), and they can't burst when they need to (because once the number of invocations goes beyond the allocated warm containers it will contend with all the other functions for warm instances, or have to wait for a cold start).

Designing a scheduling system that enforces priorities while maintaining a purely on-demand usage structure faces some significant challenges.
[hmng: this is a bit of an awkward transition]

One of the challenges is that of multi-tenancy. Priorities alone are only relative: the developer of our web server knows that they want landing page views to run before profile page views to run before map reduce jobs; but have no way of expressing that priority relative to other peoples jobs. And, in fact, should not be trusted to: their own jobs will always seem more important than others', so if given a scale their high priority work will always have the highest priority available.

Another challenge is that of managing memory. Once a global ordering of all the processes' priorities has been established, implementing (un)-fairness in scheduling cpu time is somewhat straightforward. In a world with unlimited memory, it would be simple to preempt profile view jobs with landing page view jobs when they come in. However, in a memory-limited setting, it is not clear how to know whether the profile view job might be need to be killed, and if so whether it is worth it to do so, or better to just requeue the new job.

A third challenge is that of placing jobs. In a datacenter setting, where both the number of new jobs coming in and the amount of resources are extremely large, knowing where the free and idle resources are is a challenge central to any distributed system. There is generally an understanding that the total load does not fluctuate by any significant factor. This supported by the law of large numbers (it is very unlikely that all a cloud providers customers will burst at once), and can be seen in the ways AWS prices things: spot instances are sold at a market rate determined by the amount

of resources available in a given zone[1], and the market rate is experientially very steady over time[1]; the prices for lambdas and ec2 instances also only changes very slowly[1].

2 Design

Developers using XX write function handlers and define triggers just like they would for any existing serverless offering. Additionally, developers express priorities to XX, and XX enforces these priorities.

2.1 Interface

Developers express priorities to XX via assigning functions to fixed dollar amounts per unit compute. So in the example of the web server, the home page view might be assigned a high priority and cost 2c per cpu second, a the user profile view might be assigned a middle-high priority and cost 1.5c per cpu second, and finally the map reduce job can be set to a low priority which costs only 0.5c per cpu second.

To avoid unexpected costs in the case of for example a DOS attack or a bug, developers also express a monthly budget that they are willing to pay. XX does not guarantee that the budget will not be exceeded by small amounts, but can use it as a guideline and throttle invocations or decrease quality of service in the case that usage is not within reason given the expected budget.

Finally, developers are required to express a maximum amount of memory per function.*[hmng: But they only pay for what they use? Def would be in keeping with the goal. Is there a penalty for being very off? At that point we might as well just use profiling information? — let's just have this be part of the discussion]*

2.2 Internals

XX consists of a distributed global scheduler, which places new function invocations, and a machine scheduler, which enforces priorities on the machines.

Machine Scheduler. The machine scheduler has a simple paradigm: processes have fixed priorities, and higher priorities preempt lower priorities. Being unfair and starving low priority processes is a feature, not a bug: map reduce jobs should not ever interrupt a page view request processing.*[hmng: talk about our notion of how we expect that the amount of work in different priorities will relate to the resources available in the datacenter? That might fit better into the discussion section]*

Each machine is also running a dispatcher that is in charge of communicating with global scheduler shards. The dispatcher starts new jobs that have been assigned to that machine, and informs global scheduler shards in the event that it has idle resources (ie if memory utilization is low). It is also in charge of killing and then requeuing jobs under memory pressure. It chooses the job to kill by looking at both memory

used and money wasted if killed (lower priority jobs should be the ones to be killed if possible, but won't help much if they weren't using any memory to begin with).

Global Scheduler. The global scheduler is sharded, and each shard maintains a multi-queue of functions assigned (one queue per priority), as well as a list of machines that have idle/available resources.

Machines are responsible for adding themselves to that list; and are tagged with the amount of memory they have available, as well as the highest priority of currently running jobs. Machines only add themselves and their availability information to one shard at a time, so although the information may be outdated, it will always be a pessimistic estimate of the current state of the machine.

Algorithm 1 Chooses a machine for a job j

```

 $N = \{ \text{machines with memAvail} > j.\text{maxMem} \}$ 
if  $|N| > 0$  then
  return  $\min(N.\text{maxPriorityRunning})$ 
end if
 $M = \text{timeToProfit of } k \text{ polled machines}$ 
if  $\min(M.\text{timeToProfit}) < THRESH$  then
  return  $\min(M)$ 
else
   $\text{reQ } j, \text{ with priority} -= 1$ 
end if

```

Shards choose what job to place next by the ratio of priority to amount of time spent in the queue, so high priority jobs don't have to wait as long as lower priority jobs to be chosen next.*[hmng: are we expecting that there will be queue buildup at GSSs?]*

When placing a job, the shard finds a machine to run the it, as also shown in the pseudocode in Algorithm 1.

The shard will first look for a machine that has the job's maximum memory available; if there are multiple such machines, the shard chooses the machine last seen running the lowest priority; this choice minimizes cpu idleness under low load settings.

If there are no machines with the memory available, the shard switches over to power-of-k-choices: it polls k machines, sending the priority and the maximum memory of the job currently being placed, and getting back a number that represents the time it would take for it to start making a profit off of the decision of placing the process there. This number is influenced by what process it would kill, how long that has been running, and what the price differential is between the two. This value captures the sunk cost of killing a process, as well as the implicit opportunity cost in that the lower priority process being killed would otherwise keep running.

The shard then can choose to place the new job on the machine with the minimum value, or if all of them are too high the shard re-queues the job, this time with a lower priority. [hmng: Is that actually necessary? But otherwise the shard would just pick it right back up, right? Given how it's picking the next proc to place from the multi q]

Checking in with clients' budgets to ensure that usage is not significantly outpacing a rate compatible with the budget is done asynchronously: each time a job for a client is triggered a global counter is asynchronously updated. If the counter's rate of change increases absurdly with regards to previous behavior as well as the budget as a whole, this triggers a throttling for that job.

3 Preliminary Results

To model the whole systems' dynamics, we built a simulator. [hmng: List assumptions and variables?]

< graph 2: graph of latency (and matching graph of utilization?) as load increases; per priority >

4 Discussion

— talk about max proportions of each priority and how it relates to how lower priority have latency degradation vs just never running

— talk about memory/killing and overprovisioning; if we want good memory utilization we should overprovision, and be prepared to kill something. Ideally have a pay-per-usage model, but then have a problem of right-sliding. Do profiling? But that will affect our decision of whether to place or not, whether we end up needing to kill or not. Maybe have conditional placements or something?

5 Related Work

Many other projects have explored how to do better server-less scheduling.

Some projects simply change the mechanisms and try to optimize observed behavior by trying out different scheduling policies. Sparrow does this on the distributed level, and takes away that a distributed scheduler using power-of-k-choices has performance close to a centralized global scheduler. Hermod performs a from-first-principles analysis of different scheduling paradigms through a simulator, along three dimensions: late-vs early binding, PS vs FCFS vs SRPT, and least loaded vs locality based vs random load balancing. Kairos does a distributed approximation of Least Attained Service, as a way of avoiding trying to estimate future behavior based on past performance.

Other projects do just that: generate better scheduling decisions by getting access to more information about the workload. There are two different ways of going about this: learn the behaviors, or ask developers to specify.

ALPS, for instance, observes and learns the behaviors of existing functions and then makes scheduling decisions based on those. Another example is Morpheus, which learns SLOs from historical runs, and then translates these to recurring reservations.

Other projects ask developers to specify the desired behavior. Each has come up with a different way of expressing priorities or policies, and then built a scheduler to implement those priorities. Sequoia, for instance, adds a metric of QOS, and then builds a prioritization in front of AWS lambda. Another project creates a language of Allocation Priority Policies (APP), which is a declarative language to express policies, then builds a scheduler around that. [hmng: should I talk about the actual contents of the languages and how they express priorities? Some are pretty complex, but this is also the category that we fall into, and I talk about it for AWS so; but AWS's is also way simpler] AWS lambda itself also goes this route, by offering two different ways for developers to influence lambda function scaling: provisioned and reserved concurrency. Provisioned concurrency specifies a number of instances to keep warm, and reserved concurrency does the same but ensures that those warm instances are kept for a specific function.

XX goes the route of asking developers to provide information about desired behavior, but has a much simpler interface than Sequoia or APP. Lambda also has a simple interface, but it is focused on mitigating effects of cold starts, rather than actually prioritizing the scheduling of the functions.

References

- [1] TODO. "TODO". In: 2020.