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 serverless jobs is attractive to developers and providers: developers pay only for what they use while having access to many resources when needed; and cloud providers have control over scheduling and can use that to drive up utilization.

However, for many applications it is rare to see them entirely hosted on serverless offerings today[1]. 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. It is important that the page views finish quickly, with the landing page slightly taking precedence over the profile pages, and the map reduce jobs don't really matter.

If a web developer wanted to host their web in an existing serverless offering, a popular option is AWS lambda[1]. In order to ensure that page views have their required latency, developers can influence the ways that different functions scale[1]. They can used 'provisioned concurrency' (a number of warm containers kept for the given lambda) in order to ensure a request rate up to which page views will not experience cold start. They can also use 'reserved concurrency' (an amount of concurrency, not necessarily warm, reserved for the given lambda) to ensure that the map reduce job cannot use too much of the accounts concurrency, by reserving large portions for each of the two page views.

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 the containers warm), and they potentially 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, and if it's beyond the reserved concurrency will have to contend to run at all).

It becomes clear when looking at Lambda's interface that they are mostly concerned about slow cold start times. However, cold start latencies are a popular area of research and as a result have been reduced significantly — recent state of the art systems have been able to support latencies in the single digit ms range[1]. This is an opportunity: as cold start times go down, more and more latency sensitive jobs can support a potential cold start on their critical path. With low single digit ms cold start times, a page view that takes 20-30ms to run can be run as a serverless task without requiring a warm container to be able to meet its latency goals. For this paper, we assume that this is generally the case, ie that cold start times are small enough compared to the jobs being run that cold starts on the critical path are acceptable. It is then up to the scheduler to place and prioritize these jobs as desired.

This paper focuses on building a usable and efficient scheduling mechanism to support differentiating between jobs of different priorities in a world of universal serverless. In doing so, it faces multiple significant challenges.

One of the challenges is that of multi-tenancy. The fact that page views are more important to a web server than map reduce jobs is a relative statement: there is no way of mapping that prioritization to other peoples jobs, in order to be able to directly compare which job should run when. And, in fact, developers cannot be trusted to make such a comparison, their own jobs will always seem more important

than others': if given an absolute scale from 0-99 (0 being the highest priority), the highest priority job will always get a 0 and the rest will be relative to that. However, that does not reflect that in fact different clients do have different requirements and expectations, and need to be treated differently.

Another challenge is that of managing memory. Given that machines are limited in memory, placing a new high priority job invocation on a machine already running many jobs may or may not lead to enough memory pressure to require killing an already placed job (rather than just preempting other jobs). It is difficult to know whether that would be the case, and if so whether it is worth it, 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.

2 Design

Developers using XX write function handlers and define triggers just like they would for any existing serverless offering. Additionally, developers express jobs' 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 a 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 uses this budget as a guideline and throttles invocations or decreases quality of service in the case that usage is not within reason given the expected budget, though it does not guarantee that the budget will not be exceeded by small amounts.

Finally, developers are required to express a maximum amount of memory per function.[hmng: Should I have a forward reference to the discussion section here?]

2.2 XX Design

XX's main structure can be seen in Figure1: XX consists of a distributed global scheduler, which places new function invocations, a dispatcher, which runs on each machine and communicates with the global scheduler shards, and a machine scheduler, which enforces priorities on the machines.[hmng: should I draw the scheduler on there?]

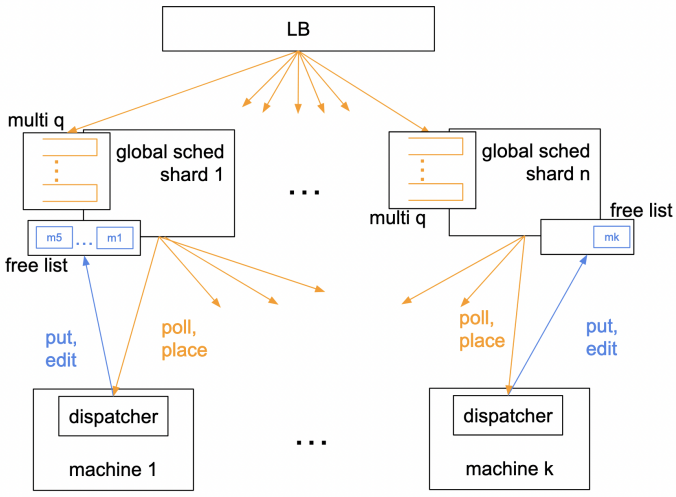

Figure 1: global scheduler shards queue and place jobs (in orange), on each machine a dispatcher thread keeps track of memory utilization and if it's low writes itself to a free list (in blue)

Attached to each global scheduler shard is a *free list*, which holds machines that have a significant amount of memory available; similarly to the way a free list in a local operating system maintains free blocks of memory, except distributed and at a much coarser granularity. In the shards free list each machine's entry is also associated with some metadata about compute pressure.[hmng: figuring out exactly what we want those values to be in the simulator right now, currently looking at max running priority and q size]

The global scheduler stores the jobs waiting to be placed in a multi queue, with one queue per priority.

Machine Scheduler. The machine scheduler is simple: jobs have fixed priorities, and higher priorities preempt lower priorities. Being unfair and starving low priority jobs is desirable, since map reduce jobs should not interrupt a page view request processing, but vice versa is expected.

Dispatcher. The dispatcher is in charge of adding itself to a shard's free list when memory utilization is low. If it is already on a list somewhere it will update that entry if it's value has changed significantly. Otherwise, the dispatcher chooses which list to add itself to using power-of-k-choices: it looks at k shards' free lists and chooses the one with the least other machines in it.

The dispatcher is also in charge of killing 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). Killed jobs are requeued at a randomly chosen shard.

Algorithm 1 Choosing a machine for a job j

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

The dispatcher also responds to probes by shards: given a potential job that a shard might want to run on the machine, the dispatcher computes the *time to profit*, which is the time it would take for the machine to start making a profit off of the decision of placing the job there. If there is enough memory free to fit the new job's max memory, that number is 0. If the dispatcher might have to kill a process due to memory pressure, it computes which job it would kill, which is the job with minimal price where $j.memUsg > newJ.maxMem$. The time to profit then is $(jToKill.timeRun \cdot jToKill.price)/(newJ.price - jToKill.price)$.

Global Scheduler.

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.

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 in its free list for a machine that has the job's maximum memory available. If there are multiple such machines, the shard looks at each machines compute pressure metrics; the goal being to minimize cpu idleness and job latency in low load settings.[hmng: playing with deatils of this in the scheduler right now] If a machine from the free list is chosen, the response from the dispatcher on placing the job will include updated info, which the shard will use to update the free list entry.

If there are no machines in the free list with the memory available, the shard switches over to power-of-k-choices: it polls k machines, sending the price and the maximum memory of the job currently being placed, and getting back the time to profit. 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.

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 understand the dyanmics of XX, we built a simulator.

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

4 Discussion

There remain a number of interesting open questions.

How is memory usage billed? Do developers only pay for the memory they use, or do they pay based on the max amount of the memory they stupilate? The former would result in gross overestimations because there is no downside to being on the cautious side, while the latter breaks the central maxim that developers only pay for what they use.

In the world where developers only pay for what they use, asking for a limit with no repurcussions if the estimate is off would be useless; everyone would just put the largest amount allowed. Adding penalties for being off seems unreasonable: some jobs simply have a high variance in their memory usage and penalizing developers that run such jobs is undesirable.

Ideally, memory would be a pay-per-use model and no further information would be requied; however that would require being able to reactively deal with memory pressure extremely quickly and well.

How should XX deal with memory pressure? Ideally, XX would avoid the situation altogether, or if it occured would be able to solve it without killing, ie wasting resources.

Avoiding memory pressure situations would require knowing with high accuracy in advance how much memory a job will use. Profiling is improving as ML models improve, and might be a good use case (especially for jobs with a high variance of memory usage, using a model that is given the inputs to invocations could work well, since the inputs are likely to be the determining factor for memory usage).

On the other hand, profiling is still just an estimate that could always be wrong, and the system needs to be able to handle that. If we are able to come up with a mechanism that is reactive and at the right time scale, ie acts quickly enough that there is no buildup and the problem goes away, that would be ideal.

One option for this might be snapshotting: lower priority functions that we are now reactively killing might be allowed to snapshot themselves and then be placed somewhere else and re-started. In this case, the timescale would have to be such that the snapshotting does not (in its use of memory or compute) prohibivelly block the other jobs on the machine from running. Another option could be paging: the lower priority processes' memory are paged out; later when there is lower load and they start running again their latency will

be affected but they are lower priority so we don't care as much (since developers pay per usage for compute one could imagine some sort of recompense for the runtime that job pays for having been paged out).

Does priority increase with waiting? Preemptive priority scheduling means that at any moment in time the process running is the one with the highest priority on the machine. This ensures that the web server's page views run with high performance, and the map reduce jobs do not interrupt but rather wait for lulls in load to run. The flipside of this is that if there are no such lulls, ie if there are so many high priority jobs that they take up all the resources all the time, it is possible that nothing of any lower priority would ever run at all.

One solution to this problem is ensure that it can never be the case the there is so much load on high priority jobs that the data center will be full with them to such a degree that high-ish priority jobs can't run. There is evidence for and we expect that load will mostly be stable: this supported by the strong law of large numbers (it is very unlikely that all the jobs' random bursts align), 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]. This means that providers can mostly choose the rough breakdown of the load they will have at any given time (ie they can choose a percentage of how many jobs they want in each priority, and change it by adjusting prices or not sllowing users to select that level anymore).

[hmng: We discuss what a good breakdown would look like in the next section? Put eval next? Or just don't discuss and leave it at that, although that seems a little vague.]

5 Related Work

Many other projects have explored how to do better serverless scheduling.

Systems like Sparrow[1], Hermod[1], or Kairos[1] improve performance of scheduling in the distributed setting by trying out and using different scheduling policies. Unlike XX, they do not differentiate betwen individual types of jobs that come in.

Some systems generate information about jobs that are coming in to help placement decisions; for instance ALPS[1], which observes and learns the behaviors of existing functions and then makes scheduling decisions based on those; or Morpheus[1], which learns SLOs from historical runs, and then translates these to recurring reservations. XX instead gets the priorities directly from the developers as part of its interface.

Other existing systems ask developers to specify the desired behavior, like XX does. However, they don't change the actual underlying machine-level scheduling, and expressing priority is more complex than in XX, and/or has a different goal.[hmng: this feels a bit awk/not on the nail. Also hard because of unknown unknowns, ie there may be a related work that I don't know about for which this is not true. Hard to make specific enough so that is unlikely, while still actually going beyond just the literal design]

Sequoia[1], for instance, creates a metric of QOS for serverless functions. Unlike XX, Sequoia does not implement a new scheduler, but builds a layer in front of AWS lambda.

Another project[1] creates a language of Allocation Priority Policies (APP), which is a declarative language to express policies, then builds a scheduler around that. Unlike XX, the APP language is built around making load balancing decisions and ultimately defines a mapping of jobs to workers, rather than assocaiting priorities with the jobs and having the scheduler do the scheduling automatically.

AWS lambda itself also goes this route, by offering two different ways for developers to influence lambda function scaling: provisioned and reserved concurrency[1]. 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. However, these knobs are more centered around mitigating the effects of cold start rather than actually affecting the way the jobs are scheduled.

References

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