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. Supposing no multi-tenancy and 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 running the landing page view would mean the profile view job would 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. Because the scheduler has access to priority information about processes, this can give the scheduler a notion of how quickly things will need to be placed, but not how many resources they will use. It also interacts with the memory management challenge: placing a job requires balancing the tradeoffs between potentially needing to kill already running jobs, or putting the new job on a machine where it may have to wait for other high priority jobs to

finish. Additionally, if a machine will preempt low priority processes with high priority ones, then it may even be desirable to place a high priority job on a machine with a longer queue (as long as they are mostly low priority jobs), rather than on a machine with few high priority processes.[hmng: this challenge definitely feels the least worked out]

[hmng: we also never get back to these challenges yet. Maybe have discussion section centered around them? The existing discussion 'subsections' could probably be made to fit into the challenges]

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 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 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: Should I have a forward reference to the discussion section here?]

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.

On each machine is also running a dispatcher that is in charge of communicating with global scheduler shards. The dispatcher is in charge of informing a gloabl scheduler shard in the event that it has idle resources (ie if memory utilization is low). Each shard has an attached 'free list', the dispatcher chooses which list to add itself to using a power-of-k-choices

Algorithm 1 Chooses 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
```

mechanism: it looks at k shards and chooses the one with the least other machines in it.

The dispatcher is also in charge of starting new jobs assigned to its machine, as well as killing and then requeing 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 the aforementioned 'free list'.

Because machines only add themselves and their availability information to one shard at a time, the information in the 'free list' may be outdated, but it will always be a pessimistic estimate of the current state of the machine.

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 highest currently running priority and number of jobs. The goal is to minimize cpu idleness and job latency in low load settings.[hmng: playing with how to relate these to each other and how to make this decision in the scheduler right now]
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 priority and the maximum memory of the job currently being placed. The shard gets back a number that represents the time it would take for the machine to start making a profit off of the decision of placing the job there. This number is influenced by what job the machine would kill, how long that has been running, and what the price differential is between the new and the

potentially killed job. This value captures the sunk cost of killing a job, as well as the implicit opportunity cost in that the lower priority job 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.

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

There are still a number of open questions that the design only partially addresses, or does not have a completely satisfactory answer to.

Economics of memory. One of the key pieces of the design is the notion that jobs have an associated amount of memory, and when the job is placed that estimate is used in order to understand the consequences of placing a job (whether this will result in a kill of not). However, we did not discuss the payment model: do developers only pay for the memory they use, or do they pay based on the max amount of the memory they give? 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.

Dealing with memory pressure. Whether the memory usage per job is capped or not, because actual memory usage varies between job invocations, having high memory utilization will requite overprovisioning machines on memory. The current design deals with high memory pressure by simply killing the process that is the most convenient: lowest priority and using enough memory to make a difference.

This is not really a satisfying solution. Ideally, the system would avoid the situation altogether, or if it occured would be able to solve it without killing, ie wasting resources.

Well-know solutions for this are profiling or reactively snapshotting/paging. 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) prohibively 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. One of the observations the system design is built on is that Processor Sharing (PS) is not the right approach in a serverless setting, where jobs run once and then are done. Instead, XX uses preemptive priority scheduling, where at any moment in time the process running is the one with the highest priority on the machine. This means that if it is possible for load in the highest priority class to take up all the resources, nothing of any lower resource would ever run at all.

It is not necessarily clear that this is desirable, for instance in the web server example if a user profile view has waited for long enough it seems fine for a landing page view to be blocked for the time it takes while the profile view job runs. This would point to it being desirable that processes gain priority as they wait. On the other hand, this would not be true of a map reduce job: page views should never be interrupted because a map review job had been waiting for a long time.

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.