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

The serverless ideal is beneficial to developers and providers in different ways: developers only pay for what they use while having flexibility to burst and still maintain performance; and cloud providers can pack more latency sensitive jobs because they don't need to have guaranteed reserved capacity for each client.

Today's world remains far from the ideal state: serverless functions have execution time limits, cold start times, and timeshare compute and i/o with many other processes.

It is certainly true that not all kinds of workloads are well suited for the serverless world: for instance, a KV store is not an easy fit for elasticity because of the large amount of state required to start up a new instance. However, applications that are stateless, or more specifically applications that are able to manage their data through a global storage system, really should be able to fully run as serverless work.

A driving example for this work is that of a web server. In principle its characteristics 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. However, it is rare to see web apps entirely hosted on serverless offerings today. One of the reasons for this is that there is no way to convey urgency or priority via existing serverless interfaces. If the developer cares about some functions more than others, the closest AWS lambda lets you do is what they call "reserved concurrency", which specifies a number of warm containers that are specifically reserved for a given function. However with that interface developers lose 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 while maintaining the desired performance (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).[hmng: feels a bit random to rig on lambdas here, but at the same time good to show state of the world (although I'm not doing a state of the art survey and talking about related work)]

We propose XX, a distributed scheduler for serverless workloads that enforces priorities on a per function invocation level.

[hmng: talk about challenges? Split into interface and internal design challenges? Design challenges kinda fall out of interface choices] [hmng: the split of content between intro and design feels like a bit of a mess, ig interface is more explained in the intro as motivated by how we think things should be, and then design is like the design of the actual system as motivated by the interface that we chose —- maybe intro and design are the wrong headings, and it should be motivation (the example) interface (the interface) and design (the actual scheduler)]

Developers express priorities to XX via assigning functions to fixed dollar amounts per unit compute; so in a web server a page view might be assigned a high priority and cost 2c per cpu second, whereas a map reduce job can be set to a much lower priority which costs only 0.5c per cpu second. The provider offers a fixed list of priorities and associated pricing, so as to avoid a bidding war among clients. Prices are not expected to change often, but this is in alignment with what AWS does today: spot instances are on demand, are sold at market rate (rather than to the highest bidder), and the market rate is experientially very steady over time. Similarly, currently in AWS developers can attach a number of vcpus and an amount of memory to a lambda, which is then associated with the price of the lambda. To avoid unexpected costs in the case of for example a DOS attack or a bug, developers can 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?]

Internally, XX uses a process-per-request model that assumes cold starts are fast enough to be tolerated on the critical path; recent state of the art systems have been able to support cold start times in the single digit ms realm, and we expect this trend to continue. [hmng: does this need to come earlier in the story? Certainly was more central in the last framing, and feels like it's a big assumption/scoping of the problem] XX consists of a distributed global scheduler, which places incoming function invocations on machines, and a machine scheduler, which runs functions by highest priority first. 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?] Global scheduler shards place incoming function invocations on a machine, prioritizing finding machines that have the available memory and are running low priority processes (in that order). If there appears to be no machine with the memory available, the shard uses power of k choices to potentially find a machine that can kill an existing low priority process, here the goal is to minimize sunk cost when killing.

[hmng: do we want/need worked out contributions? Seems a bit difficult at this point because I don't know if we have any other than like the collection of incidentals that makes up the design… scary]

2 Design

Developers using XX write function handlers and define triggers just like they would for any existing serverless offering. They then additionally attach the following pieces of information to each function:

(1) cpuPriority — An amount of money they are willing to spend per cpu second; this is chosen from a list of possible values
(2) maxMem — A maximum amount of memory the function will use; again chosen from a list of possible values

Additionally, developers specify a monthly budget.

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

When a new function invocation is triggered, that function is sent to a randomly assigned global scheduler shard. Each global scheduler shard maintains a multi-queue of functions, as well as a set of machines that have idle/available resources. Machines are tagged with the amount of memory they have available, as well as the priority of currently running jobs. The information may be outdated, but machines only add themselves and their availability information to one shard at a time so the information there will always be a pessimistic estimate of the current state of the machine. Shards choose what function to place next by the ratio of priority to amount of time spent in the queue, so high priority functions don't have to wait as long as lower priority functions to be chosen next. This mechanism is inspired by Shinjuku. [hmng: Do I need to say this? I don't mention Shinjuku in related work because its from another context/solving a different problem - maybe I should?] When placing a function, the shard will first look for a machine that has the function's maxMem 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 a power-of-k-choices mechanism: it polls k machines, giving them the cpuPriority and the maxMem of the function currently being placed. Each of the k machines responds with 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 function on the machine with the minimum value, or if all of them are too high the shard requeues the function, 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 the budget is done asynchronously: each time a function 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 function.

On the single machine level, the cpu scheduler enforces the functions' priorities via a simple highest-priority-first scheme. Under memory pressure, a daemon kills the process that has the highest ratio of memory used to money wasted if killed (reflecting that lower priority jobs should be more likely to be killed, but won't help much if they weren't using any memory to begin with). [hmng: Do we need the second part? Only if we oversubscribe, which probably we will — esp if we only charge for memory being used. I don't actually know how this would work like would we need a running daemon or can you put in some sort of handler or something; given that it would also need to notify someone that the process has been killed and restart it]

3 Implementation And Preliminary Results

To understand the dynamics of the single machine scheduler, we implement a prototype that uses linux' SCHED-FIFO to enforce priorities.[hmng: also talk about options that we thought about in terms of EEVDF/EDF, nice values, etc? Might come out of nowhere given that this wasn't a huge deal in the design, but on the other hand it is what I've spent the most time on]

< graph 1: representative microbench comparing straight CFS linux vs cgroups vs nice values vs priority >

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 Related Work

Many other projects have explored how to do better serverless 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.