Adaptive Concurrency Control for Mixed Analytical Workloads

Dan Kleiman
March, 2023
At Klaviyo, we do targeted messaging, data integrations, and analytics.
Analytics via APIs, Dashboards, and Reports

Which email domains have the best open rates?

Who are my most engaged customers?

How much revenue did my last campaign generate?
Fast, Flexible Query Services
Mixed workloads running on the same “shared calculator”
Healthy request processing for thousands of requests per second
Unhealthy request processing - waves of congestion
“My workload hasn’t changed. Why are my requests suddenly timing out?”
Better way to keep our service healthy for all our users?

Stop Rate Limiting!  
Capacity Management Done Right for APIs

Jon Moore  
Chief Software Architect & Senior Fellow  
Comcast Cable  
@jon_moore

Netflix Technology Blog  
Mar 23, 2018  ·  5 min read  ·  Listen

Performance Under Load

Adaptive Concurrency Limits @ Netflix

by Eran Landau, William Thurston, Tim Bozarth
Metrics Service
Request Flow
Metrics Service Request Flow

A. Metrics Service Client
   - Query Request
   - Processed Results

B. Envoy Proxy x 4
   - Request Routing
   - Processed Results

C. Metrics Service Server Pod x 24
   - Query Execution
   - Query Results

D. Clickhouse Node x 128

E.
Metrics Service Request Flow - gRPC Server

[Diagram of the Metrics Service Request Flow - gRPC Server]
Request Queuing and Concurrency
Healthy State - Queuing Balanced with Processing

4 requests in the queue, 4 second deadline each, processing 1 request per second, no timeouts
Unhealthy State - Queue Depth Exceeds Processing Rate

8 requests in the queue,
4 second deadline each,
processing 1 request per second,
last 4 requests to arrive time out
Unhealthy State - Processing Rate Slows Down

4 requests in the queue, 4 second deadline each, 2 seconds to process a request, last 2 requests to arrive time out
Concurrency is nothing more than the number of requests a system can service at any given time and is normally driven by a fixed resource such as CPU.

A system’s concurrency is normally calculated using Little’s law, which states: For a system at steady state, concurrency is the product of the average service time and the average service rate ($L = \lambda \cdot W$). Any requests in excess of this concurrency cannot immediately be serviced and must be queued or rejected. With that said some queueing is necessary as it enables full system utilization in spite of non-uniform request arrival and service time.

Systems fail when no limit is enforced on this queue, such as during prolonged periods of time where the arrival rate exceeds the exit rate. As the queue grows so will latency until all requests start timing out and the system will ultimately run out of memory and crash. If left unchecked latency increases start adversely affecting its callers leading to cascading failures through the system.

from Netflix’s Performance Under Load
Accept or Reject the Next Request?
Accept or Reject Before Queuing - Load Shedding

3 requests in the queue,
4 second deadline each,
processing 1 request per second,
accept or reject next request?
Accept or Reject Before Queuing - Load Shedding

4 requests in the queue, 4 second deadline each, processing 1 request per second, accept or reject next request?
Load Shedding at the Cluster Level
Load Shedding and Concurrency Control

1. How many requests are we already processing – *inflight requests*?
2. What our maximum number of requests we can process at once – *concurrency limit*?
3. If inflight request count < concurrency limit, accept the new request.
4. Otherwise, reject it.
Adaptive Concurrency Control
Adaptive Concurrency Control - Measuring Latency
Adaptive Concurrency Control - Record, Recalculate, React

1. Record latency (RTT) of each request
2. Calculate aggregate latency over a period of time
3. Adjust concurrency limits based on the aggregate latency value
Adaptive Concurrency Control - AIMD Algorithm

**Additive Increase**

When we are within our latency tolerance, we can increase the concurrency limit by 1.

**Multiplicative Decrease**

When we cross the latency threshold, we decrease the concurrency limit by a backoff multiplier.
# AIMD - Additive Increase, Multiplicative Decrease - Algorithm
# used to update concurrency limits, given current latency, number
# of inflight requests and whether there has been an absolute timeout

def _update(
    self,
    start_time: float,
    rtt: float,
    inflight: int,
    timeout_observed: bool
):
    # if we cross the latency threshold,
    # we backoff by our pre-configured backoff ratio
    if timeout_observed or rtt > self._latency_threshold_ms:
        self._current_limit = math.floor(
            self._current_limit * self._backoff_ratio
        )
    # otherwise, we can increase the limit if the current inflight
    # request count is approaching the limit
    # if they are far apart, we don't do anything
    elif inflight * 2 >= self._current_limit:
        self._current_limit += 1
    # finally, we make sure the limit is within the min/max bounds
    self._current_limit = min(
        self._max_limit, max(self._min_limit, self._current_limit)
    )
Python implementation of Netflix’s java version

```python
# AIMD - Additive Increase, Multiplicative Decrease - Algorithm
# used to update concurrency limits, given current latency, number
# of inflight requests and whether there has been an absolute timeout

def _update(
    self,
    start_time: float,
    rtt: float,
    inflight: int,
    timeout_observed: bool
):

    # Backoff Condition
    # if we cross the latency threshold,
    # we backoff by our pre-configured backoff ratio
    if timeout_observed or rtt > self._latency_threshold_ms:
        self._current_limit = math.floor(
            self._current_limit * self._backoff_ratio
        )

    # request count is approaching the limit
    # if they are far apart, we don't do anything
    elif inflight * 2 >= self._current_limit:
        self._current_limit += 1

    # finally, we make sure the limit is within the min/max bounds
    self._current_limit = min(
        self._max_limit, max(self._min_limit, self._current_limit)
    )
```
# AIMD - Additive Increase, Multiplicative Decrease - Algorithm
# used to update concurrency limits, given current latency, number
# of inflight requests and whether there has been an absolute timeout/

def _update(
    self,
    start_time: float,
    rtt: float,
    inflight: int,
    timeout_observed: bool
):
    
    # if we cross the latency threshold,
    # we backoff by our pre-configured backoff ratio
    if timeout_observed:
        self._current_limit
        self. current 1
    
    # otherwise, we can increase the limit if the current inflight
    # request count is approaching the limit
    # if they are far apart, we don't do anything
    elif inflight * 2 >= self._current_limit:
        self._current_limit += 1

    # finally, we make sure the limit is within the min/max bounds
    self._current_limit = min(
        self.max_limit, max(self._min_limit, self._current_limit))
}
“My workload hasn’t changed...”
After we derive a new Global Limit, we calculate Partition Limits as percentages of the Global limit.

Any caller can be mapped to a Partition.

Partition Limits guarantee throughput allocations on a per caller basis.
Going Live…
RTT increasing from 100ms to 400ms is a signal that we’re slowing down. Need to accept fewer new requests.
Reducing the limit in response to increased latency allows the system to recover gracefully.
Changes in RTT per server pod vary based on the query mix, so latency can vary considerably across the cluster.
When things did get bad? No more congestion, just spikes.
Thank you! Any Questions?

Blog post at klaviyo.tech

Ask me more @Dan_Kleiman