OVERLOAD CONTROL FOR MS-SCALE RPCS WITH BREAKWATER

Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh,
and Adam Belay, MIT CSAIL
in *Proc. of USENIX OSID,* 2020

NSLab Seminar

Ga Eun Seo
Networking and Systems Lab.
20211133@sungshin.ac.kr
February 9, 2023

Contents

- Introduction
- Motivation and Background
- System Design
- Implementation
- Evaluation
- Conclusion

Introduction

- Modern data center applications consist of a series of microservices that interact using remote procedure calls (RPCs)
  - To satisfy the low latency requirements of modern applications, microservices often have strict Service Level Objectives (SLOs)
  - Maintaining tight SLOs remains challenging during overloads where a server's breakwater approaches or exceeds its capacity
- The goal of overload control is to shed excess load to ensure both high server utilization and low latency
  - Existing overload control methods are impractical because the server discards very short requests, so the overhead is comparable to the request's service time
- Breakwater, an overload control system for µs scale RPC, is presented
  - Breakwater relies on a server-based admission control scheme that allows clients to send requests only when they receive credit from the server
  - Server queuing delay is used as an overload signal
  - Breakwater uses demand guessing to minimize the communication overhead for servers to know which clients need credits

Motivation and Background

- Overload control is key to ensuring that backend services continue to function even when processing demands exceed available capacity
  - Overload identified as a major cause of cascading failures in large services
- Temporary overloads occur for a variety of reasons
  - Provisioning sufficient capacity for maximum breakwater may not be cost-effective
  - Services experience unexpected overload conditions despite capacity planning
- Without proper overload control, the system experiences a livelock where incoming requests starve to death as the server is busy handling interrupts for new packet arrivals
  - Doesn't produce useful work because most of the requests don't meet the SLO
- RPCs in microseconds are much more susceptible to performance degradation due to short-lived congestion than RPCs with longer service times
  - Even when the average demand from clients is less than capacity, the arrival of requests in short bursts lowers the latency of short requests

Motivation and Background

- RPC spans a variety of operations on data residing in memory or fast storage such as M.2NVMe SSDs.
- A single server handles µs-scale requests from thousands of clients at very high rates
- To cope with µs-scale RPCs, an ideal overload control mechanism provides the following properties:
  - No loss in throughput
  - Low latency
  - Scaling to a large number of clients
  - Low drop rate
  - Fast feedback

Motivation and Background

- Problem Definition and Objectives
  - No loss in throughput
    - RPC servers should handle requests at full capacity regardless of overload to avoid livelock scenarios
    - The overhead of performing overload control should be minimized
  - Low latency
    - An ideal overload control scheme ensures that every request processed minimizes the amount of time it spends waiting on the server
    - Low latency ensures that processed RPCs meet SLOs
    - Especially important for μs-scale RPCs where SLOs tend to be tight

Motivation and Background

- Scaling to a large number of clients
  - For short RPCs, clients with intermittent demands consume few resources on the server
  - An ideal overload control system is resilient to "incast" scenarios when a large number of clients send requests in a short amount of time
  - Overload control prevents queuing build-up caused by incasts without harming throughput
- Low drop rate
  - Dropping requests wastes resources on the server
    - Because it has to spend time processing and parsing the packets to be dropped
  - Overload control minimizes drop rates on servers
    - Dropping a request makes the RPC's tail latency more expensive when the network round-trip time (RTT) is similar to the RPC's execution time, resulting in higher retries
- Fast feedback
  - Clients have more flexibility in deciding what to do next if they can detect when within the SLO a request is unlikely to be served
    - If the server expects a request to violate an SLO, it should notify the client as soon as possible so that it can determine an alternative action rather than waiting for the request to time out

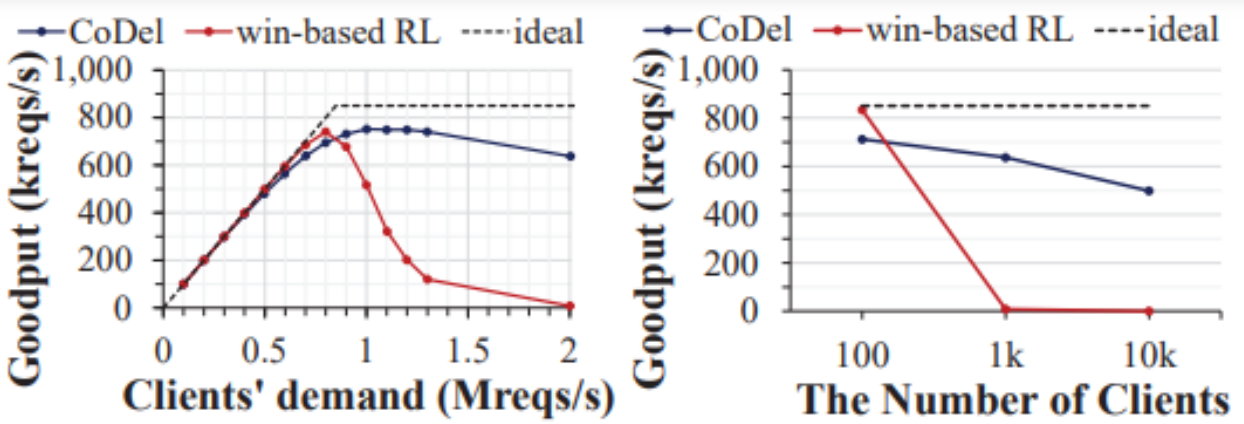
Motivation and Background

- Overload Control in Practice
  - To shed excess load before it consumes any resources
    - By dropping excessive load on the server or limiting the rate at which requests are sent on the client
  - The performance impairments of these two popular overload control approaches, developed for RPCs with long execution times, when used for μs-scale RPCs
    - Active Queue Management (AQM)
    - Client-side Rate limiting
- A hybrid approach combining client-side rate limiting and AQM is also proposed
  - Provides a more comprehensive evaluation of rate-based rate limiting and hybrid approaches

Motivation and Background

- Active Queue Management (AQM)
  - Acts as a circuit breaker, dropping requests from servers or separate proxies in certain congestion situations
  - Implemented an RPC server using CoDel for AQM to demonstrate the limitations of the AQM approach
    - The drop threshold parameter is tuned to achieve the highest goodput at an SLO of 200 μs
  - The greater the number of clients, the worse the goodput performance degradation
    - As demand from clients increases,
      more CPU is used to process packets, even though most requests are dropped by the server
    - As the number of clients increases,
      the overhead of sending failure messages increases with more clients because fewer messages can be merged

Goodput of CoDel and window-based rate limiting



(a) Goodput vs. clients' demand with 1,000 clients

(b) Goodput vs. # clients with clients' demand of 2M reqs/s

Motivation and Background

- Client-side Rate limiting
  - To eliminate the overhead of dropping requests on the server, some overload control mechanisms limit the transfer rate on the client
    - The client's response to overload is delayed by the network RTT, and a longer delay occurs when the RPC execution time is equal to or less than the RTT
    - The time it takes to receive feedback increases with the number of clients
  - As the number of clients increases, the load generated by each client becomes more intermittent, and messages are exchanged less frequently between individual clients and servers
    - Incast congestion occurs when many clients exceed server capacity, resulting in large queuing delays
    - AQM avoids long latencies by offloading excessive load from the server, so AQM outperforms client-based approaches for large numbers of clients, albeit with less than ideal goodput
- Implements the window-based rate limiting used by ORCA to account for client-side rate limiting with µs-scale execution times
  - Mechanism is similar to TCP congestion control
  - The client maintains a window size representing the maximum number of outstanding requests
  - If a response is received and the response time is less than the SLO, additionally increase the window size

Motivation and Background

- Challenge
  - Short average service times
  - Variability in service times
  - Variability in demand
  - Large numbers of clients

Motivation and Background

- Short average service times
  - Aim to support execution times of RPCs in microseconds
  - To devise an overload control scheme that can react in microseconds while keeping coordination overhead well below request service times
    - Errors in devising or implementing an overload control scheme can result in long queuing, overloading, or reduced server utilization
- Variability in service times
  - RPC execution times usually follow a long-tail distribution
    - The stochastic nature of RPC service times limits the accuracy of adjustments or scheduling on the client or server
    - Accurate scheduling requires knowledge of the execution time of each request, which is impossible in the presence of long-tail variability in execution time
    - Mutability is an ambiguity for overload detection because a single request can be long enough to cause significant queuing delays

Motivation and Background

- Variability in demand
  - Scheduling a client's access to a server requires some knowledge of the client's needs and the server's capacity
  - RPCs have variable arrival patterns and clients have sporadic demand during periods of inactivity
  - Variability in demand can lead to underutilization because clients given access to server capacity may not have enough demand to utilize it
- Large numbers of clients
  - All problems get worse as the number of clients increases
    - More difficult to fine-tune and higher overhead
    - Systems are more fragile when many clients generate demand simultaneously, increasing demand variability

Motivation and Background

Challenge

- Problems faced by server overload control systems
  - Network congestion control aims to maintain short packet queuing on switches while maximizing network link utilization
  - Packet processing time is constant,
    RPC processing sometimes shows high variance in request service time

- Issues Observed in Network Congestion Control
  - Overload control aims to maximize CPU utilization while keeping request queuing short on the RPC server
  - Client-side demand can fluctuate more at the RPC layer

- Designing an overload control system requires overcoming other challenges than network congestion control systems
  - Due to the high volatility of processing time and demand

Motivation and Background

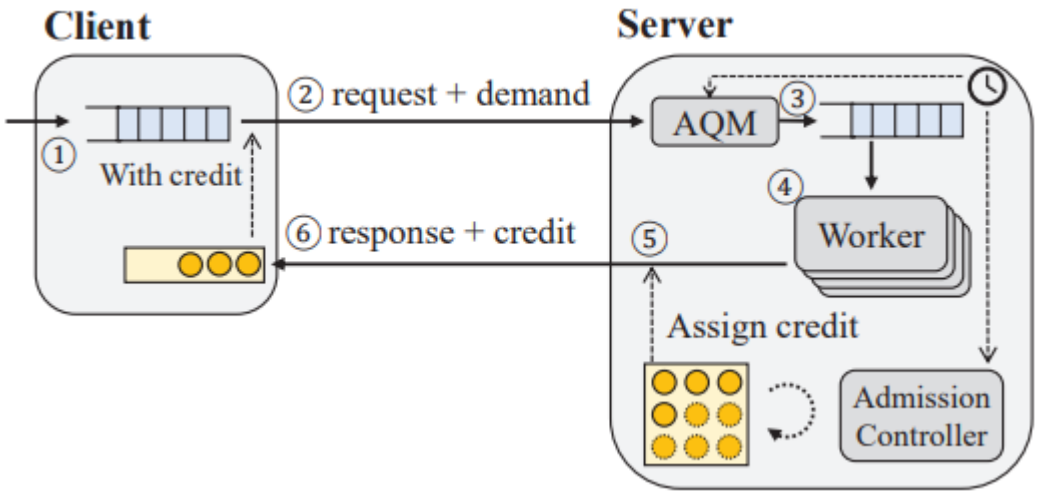
- Our Approach
  - Insights from receiver-centric mechanisms proposed in recent work on data center congestion control
    - In receiver-centric congestion control, receivers issue explicit credits to senders to control packet transmission, which provides better performance than traditional sender-based approaches
  - Explicit server-based admission control
    - A client can send a request only if it has been given explicit permission from the server
    - Server-based planning allows adjustments based on accurate estimates of server health
    - This allows for more precise control maintaining high utilization and low latency

Motivation and Background

- Demand speculation with overcommitment
  - The server needs knowledge of the client's needs to determine which client can send the request
  - Exchanging information introduces significant overhead as the number of clients increases
  - As the execution time of RPC decreases, resulting in greater overhead
    - Demand information is exchanged more frequently
  - The main difference between server-based and client-based schemes
    - The server can alleviate the need to have full information about the client's needs without hurting performance
    - Allow the server to guess about the needs of the client and prevent the server from under-utilizing the server by issuing more credits than it has capacity to allow overcommitment
- AQM(Active Queue Management)
  - Due to overcommitting, the server sometimes overload beyond its capacity
  - Relying on AQM to reduce excessive overload

System Design

- Breakwater
  - A new client joining the system sends a registration message to the server indicating the number of requests that are queued
    - The client piggybacks the first request on the registration message
  - The server adds the client to the client list and executes the request if it is not overloaded
  - The server responds to the client with an execution result or failure message
    - Piggybacks the credits issued to the client according to the request indicated by the client with the response
  - The client issues more requests based on the number of credits received
    - When the client no longer requests, it sends a deregistration message to the server that returns unused credits



Breakwater overview

System Design

- Several signals you can use to determine if a server is congested or not
  - CPU Breakwater
    - Popular congestion signal and is often used in cloud computing to make autoscaling decisions
    - CPU utilization represents only one type of resource contention that can affect RPC latency
    - Using CPU utilization as a signal, the overload controller cannot differentiate between real-time lock status and the ideal scenario of 100% utilization with no delayed RPCs
  - Server queue length
    - Similar signals are widely used to control network congestion
    - When RPC service times are highly distributed, queuing length is not a good indicator of request latency
  - queuing delay
    - Accurate even under RPC service time variability
    - Intuitive to map a target SLO to a target queueing delay at the server
- Breakwater uses queuing delay as its congestion signal
- Accurate measurement of the queuing delay signal is required for effective overload control
  - Signals must account for the sum of each phase of queuing delay experienced by the request and must ignore delays due to overload
    - Suppress incoming requests only when the system is overloaded

System Design

- Breakwater has two stages of queueing
  - Packets are in the queue waiting to be processed to generate the request
  - The thread created to handle the request waits for execution
- Breakwater tracks and sums the queuing delay in these two phases
  - For every queue in the system, each item is timestamped when enqueued
    - The timestamp is updated every hour as the queue dequeues
  - When the delay of queuing needs to be calculated, Breakwater calculates it using the difference between the current time and the queue's oldest timestamp
    - To keep track of the overall queuing delay as requests move from one queuing stage to another
- Existence of multiple causes of delay that are not caused by high utilization or overload
  - The biggest cause of delay is the threading model used by the system
  - For accurate queuing delay measurements, the system must avoid these delays
- Breakwater uses a dispatcher model for handling requests
  - Minimized when the dispatcher model is implemented using the recently proposed low-latency stack and lightweight threads in Arachne

Implementation

- Breakwater requires a network stack with low latency to ensure accurate estimation of the queuing delay signal
  - Using Shenango, an operating system
    - designed to deliver low latency to µs-scale applications with fast core allocation, lightweight user-level threads, and an efficient network stack
  - Implementation of Breakwater as an RPC library on top of the TCP transport layer
    - Handles TCP connection management, admission control with credits and AQM
    - Abstract the connection and provides a simple, individual RPC-oriented interface to the application
    - allowing the application to specify only the request processing logic
- Threading model
  - Breakwater relies on the dispatcher threading model to accurately measure queuing delay
  - The Breakwater server has a listener thread and an admission controller thread running
  - The receiver thread reads and parses incoming packets to generate requests
  - The sender thread is responsible for sending back a response (success or rejection) to the client
    - Multiple responses, the sender thread merges the responses to reduce messaging overhead
  - For all threads, use lightweight threads provided by Shenango's runtime library

Implementation

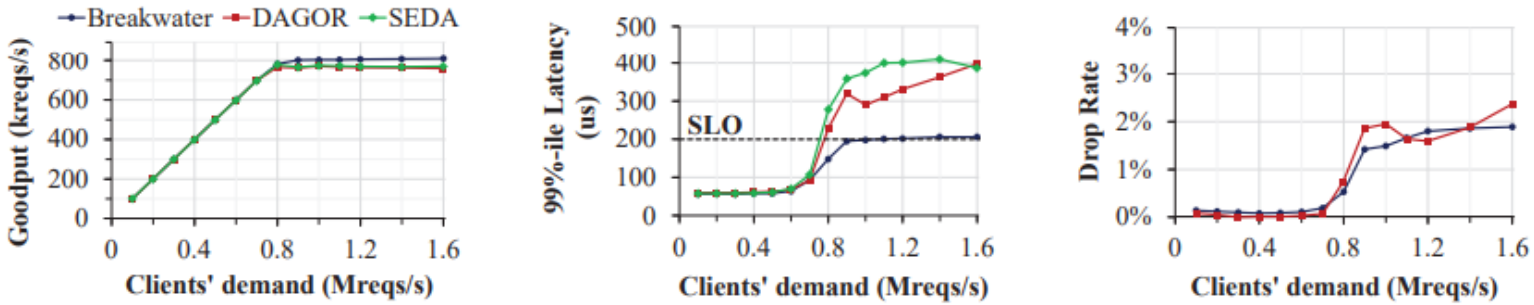
- Queueing Delay Measurement
  - Two major causes of queuing delays in Shenango
    - Packet queuing delay
    - Thread queuing delay
  - Instrument packet queuing and thread queuing
    - Each queuing keeps a timestamp of the oldest item
  - Modify Shenango's runtime library to emit queuing delay signal to RPC layer
    - When Shenango's runtime requests queuing delay, returns the maximum delay for packet queuing and thread queuing
- Lazy credit distribution
  - The admission controller must redistribute credits to clients to achieve max-min fairness based on up-to-date demand information
  - To reduce credit distribution overhead, Breakwater uses delay credit distribution to approximate max-min fair assignments
    - The sender thread decides whether to issue new credit, not issue credit, or cancel credit based on Cissued, Ctotal, and latest demand information

Evaluation

- Testbed
  - Using 11 nodes in a Cloudlab xl170 cluste
  - Nodes are connected via Mellanox 2410 25Gbps switches
  - Every node specifies a hyperthreaded pair dedicated to Shenango's IOKernel
- Baseline
  - Breakwater compared to priority-based overload control systems DAGOR and SEDA
    - DAGOR uses queuing delay to adjust the priority threshold at which the server drops incoming requests
    - SEDA uses a rate-based rate limiting algorithm, setting a percentage based on 90% file response time
- Setting end-to-end SLO
  - Set strict SLOs to support low-latency RPC applications
    - SLO budget based on server-side request processing time and network RTT
    - Our setup has an RTT of 10 µs and SLOs of 110 µs, 200 µs, and 1.1 ms for workloads with average service times of 1 µs, 10 µs, and 100 µs, respectively
- Evaluation metrics
  - Reports goodput, 99% file latency, drop rate and reject message delay
  - The reported latency captures any delay encountered by the request from the moment the request is issued until the response is received by the client
  - The drop rate is only reported by the server as it directly affects overall system performance

Evaluation

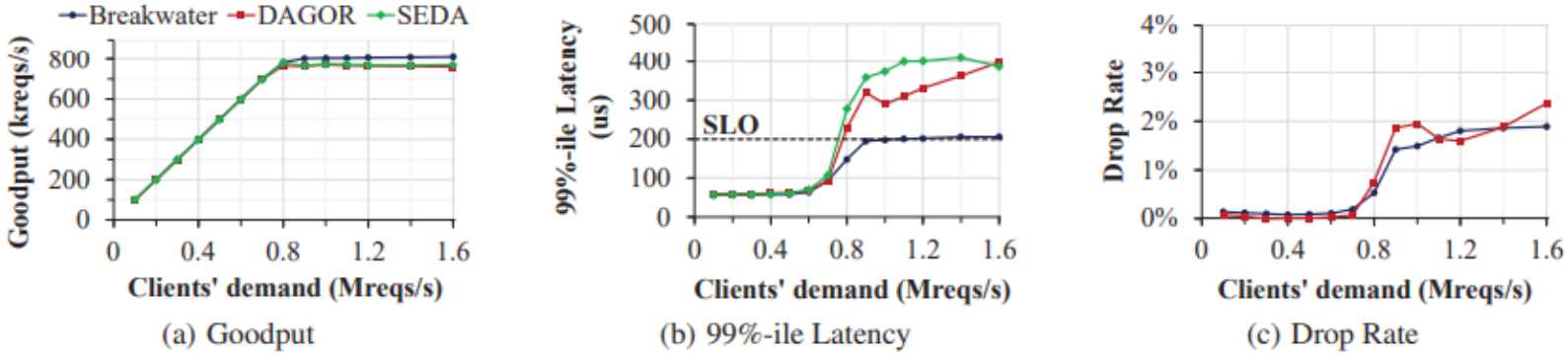
- Workload
  - Running 1,000 clients evenly split across 10 nodes in a CloudLab setup
  - Change the demand by changing the server's average request arrival rate between 0.1x and 2x the server's capacity
- Overall performance
  - When client demand is less than capacity, all three systems perform similarly in terms of goodput, latency and drop rate
  - At 700k reqs/s, SEDA has 99% higher file latency than Breakwater or DAGOR by 15%
    - Because SEDA doesn't drop the request on the server
  - Breakwater handles incasts well by preventing clients from sending requests unless they have credits and limiting the maximum queuing size
    - Breakwater achieves higher goodput with lower, limited tail latency



Performance of Breakwater

Evaluation

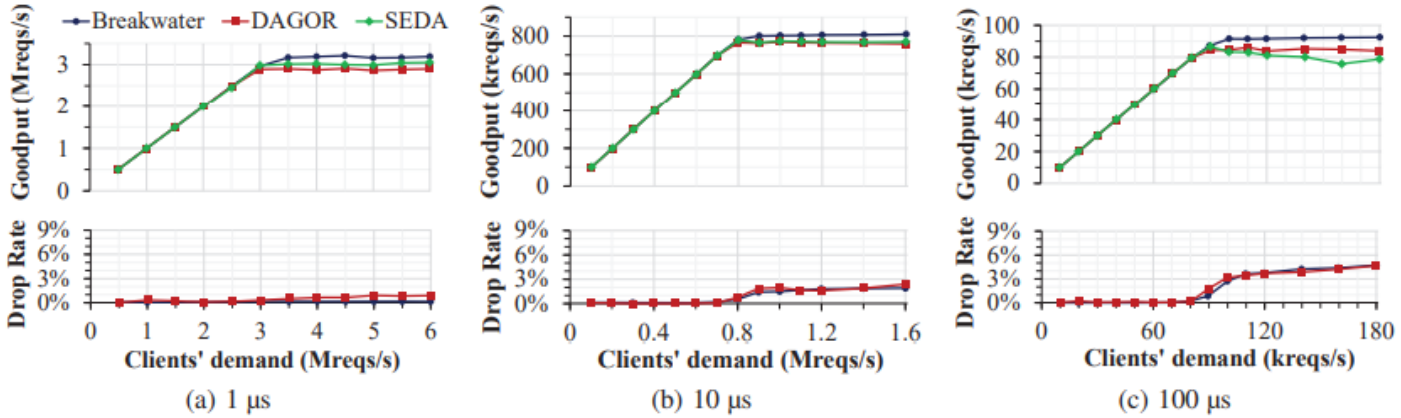
- Overall performance
  - When client demand is less than capacity, all three systems perform similarly in terms of goodput, latency and drop rate
  - At 700k reqs/s, SEDA has 99% higher file latency than Breakwater or DAGOR by 15%
    - Because SEDA doesn't drop the request on the server
  - Breakwater handles incasts well by preventing clients from sending requests unless they have credits and limiting the maximum queuing size
    - Breakwater achieves higher goodput with lower, limited tail latency
  - Breakwater is affected by incasts due to overcommitted credits, increasing tail latency and increasing drop rate due to overload
    - Breakwater relies on delay-based AQM to effectively limit tail latency while maintaining a comparable fall rate to DAGOR



Performance of Breakwater, DAGOR and SEDA

Evaluation

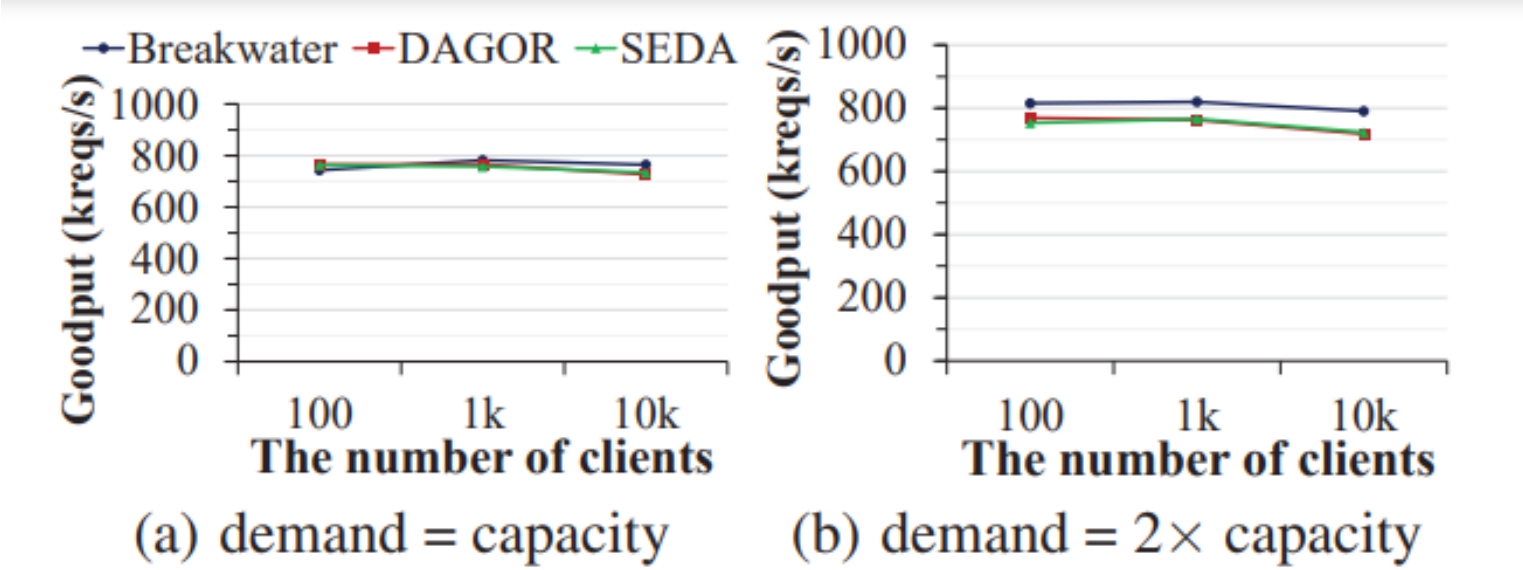
- Impact of Workload Characteristics
  - Experiments were repeated with different service time distributions and average service time values to ensure that Breakwater's performance benefits were not limited to specific workloads
  - Breakwater achieves the highest goodput regardless of load and service time distribution
  - As customer demand increases, Breakwater's good put benefit grows
    - Breakwater achieves 5.7% more goodput than SEDA and 6.2% more goodput than DAGOR with an exponential distribution at twice the capacity load
  - Breakwater Outperforms DAGOR and SEDA Regardless of Customer Needs and Average Service Times
    - Exposed the delayed response problem of SEDA and DAGOR
      - As average service time increases, client and server exchange messages less frequently
    - SEDA and DAGOR don't handle it well because incasts get larger as demand increases



Goodput and drop rate with difference average time

Evaluation

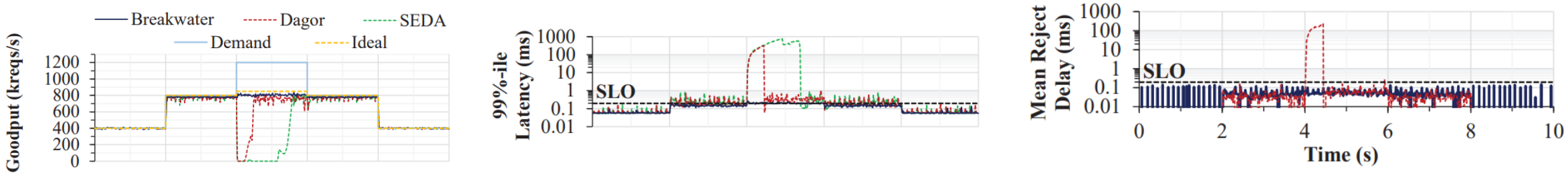
- Scalability to multiple clients
  - Varying the number of clients from 100 to 10,000 using synthetic workloads where service times follow an exponential service time distribution with an average of 10 μs
  - Goodput of all systems degrades as the number of clients increases as demand from clients approaches and exceeds capacity
    - Incast size increases as the number of clients increases, leading to poor performance
  - Performance of DAGOR and SEDA drops more than Breakwater as the number of clients increases
    - As the number of clients increases, each client exchanges messages with the server less frequently



Goodput with different numbers of clients

Evaluation

- Responding to Rapid Changes in Demand
  - RPC servers experience rapid changes in demand for a number of reasons, such as load imbalances, bursts of packets, unexpected user traffic, or redirected traffic due to server failures
  - When client demand is much less than capacity
    - Overload control schemes maintain similar goodput and tail latencies in steady state
  - When demand increases to near server capacity
    - Breakwater quickly converges and provides stable operation in both goodput and tail latency
    - Breakwater quickly converges while DAGOR and SEDA undergo stagnation collapse if the server suddenly spikes at time = 4 seconds and becomes constantly overloaded
    - Breakwater experiences a temporary tail latency increase (reaching 1.4x SLO) as client demand suddenly increases due to an increase in incasts due to overcommitted credits
      - But Credit cancellation and AQM quickly limit the impact of further incasts

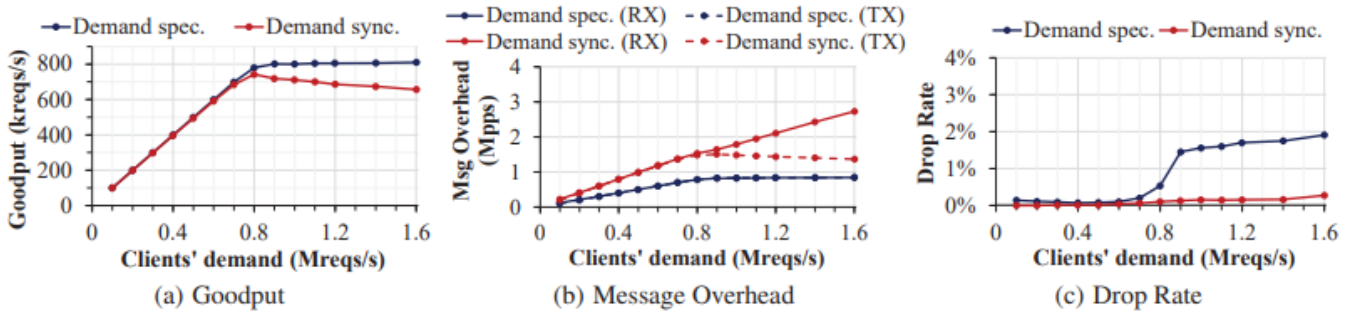


Goodput, 99%-ile latency and mean rejection delay

Evaluation

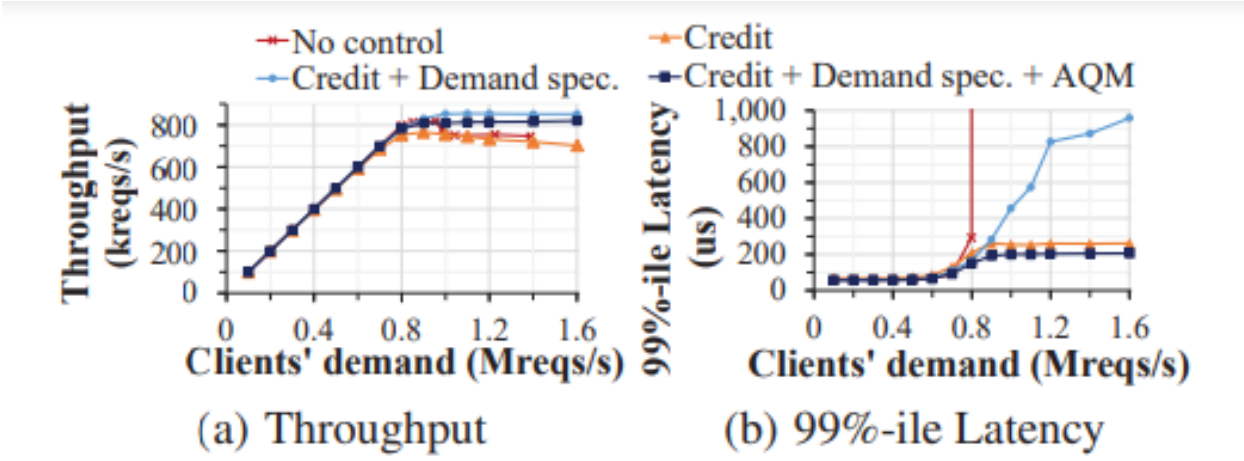
- The Value of Demand Speculation
  - Comparison of two strategies for gathering demand information: demand synchronization and demand guessing
    - To quantify the performance benefit of demand guessing
    - With demand synchronization, the client notifies the server whenever demand changes using explicit demand messages
      - The server generates an explicit credit message to the client if it cannot piggyback on the response
    - Demand guessing allows the server to speculatively estimate the client's demand based on the latest demand information and, where possible, incorporate credit for the response
  - Use demand synchronization, both RX and TX message overhead increases as demand from clients increases, resulting in goodput degradation
    - As the system becomes overloaded, the overhead of request messages continues to increase as client-specific requests change more frequently as client demands increase.
  - Overall, the cost of synchronization between client and server is high in terms of goodput degradation and network overhead, but the small benefit of lowering the drop rate on the server

Goodput message overhead, and drop rate



Evaluation

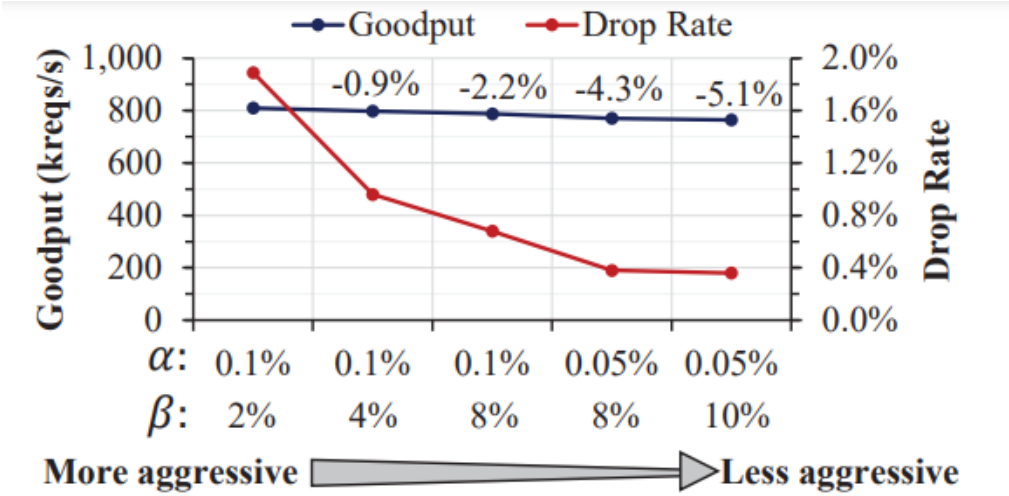
- Performance Breakdown
  - Measured throughput and 99% file latency after incrementally enabling the three main components: credit-based admission control, demand forecasting, and delay-based AQM
    - No overload control at all, throughput starts to degrade and tail latency spikes so that almost every request violates the SLO when demand rises above server capacity
    - Credit-based admission control effectively lowers and limits tail latency, but throughput still suffers due to messaging overhead
    - Demand guessing via message piggybacking reduces messaging overhead, but exacerbates tail latency due to credit overcommit incasts
  - Breakwater effectively handles incasts using delay-based AQM, providing high throughput and low tail latency



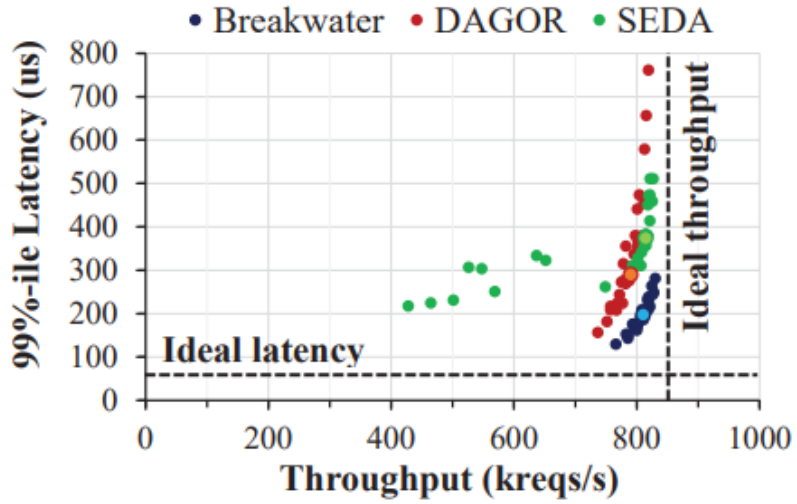
Breakwater performance breakdown

Evaluation

- Parameter Sensitivity
- The Breakwater parameter is set aggressively to maximize goodput, resulting in a relatively high drop rate
  - Less aggressive parameters, Breakwater can drop fewer requests at the expense of goodput
  - Breakwater sacrificed 2.2% of goodput (α = 0.1%, β = 8%) for a drop rate of 0.7% and sacrificed 5.1% of goodput (α = 0.05%, β = 10%) for a drop rate of 0.4%
- Breakwater delivers high throughput and low tail latency despite parameter configuration errors
  - Better throughput and latency tradeoffs and consistent performance with different parameter sets



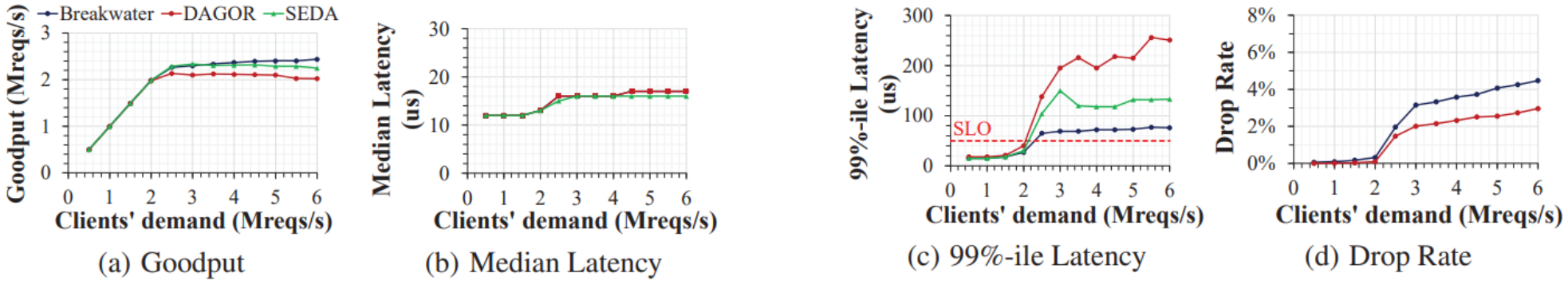
Goodput and drop rate



Throughput and 99%-ile latency trade-off

Evaluation

- Performance under Realistic Workload
  - To evaluate Breakwater in a realistic scenario, create a scenario where one memcached instance serves 10,000 clients
    - USR workload where 99.8% of requests are GET and another 0.2% is SET
    - Set SLO to 50μs considering that memcached's GET operation delay time is less than 1μs
  - Breakwater achieves stable goodput, low latency and low drop rates
    - DAGOR and SEDA have long tail latency due to incast when the server is overloaded, causing goodput degradation
    - With customers requesting twice the capacity, Breakwater achieved 5% more goodput and 1.8x lower file latency than SEDA at 99%
  - Breakwater shows a 99%-ile latency about 25μs higher than SLO and a drop rate about 1.5% points higher than DAGOR



Memcached performance for USR workload

Conclusion

- In this paper, we present Breakwater, a server-based credit-based overload control system for microsecond-scale RPC
- Breakwater achieves high throughput and low latency regardless of RPC service times, server load, and number of clients generating load
- Breakwater aims for non-zero queuing delay while avoiding queuing build-up while maintaining high utilization while generating credits based on the server's queuing delay
- Breakwater also significantly reduces the remaining messaging overhead by piggybacking demand and credits for requests and responses