

Scalable QoS for Distributed Storage Clusters using Dynamic Token Allocation

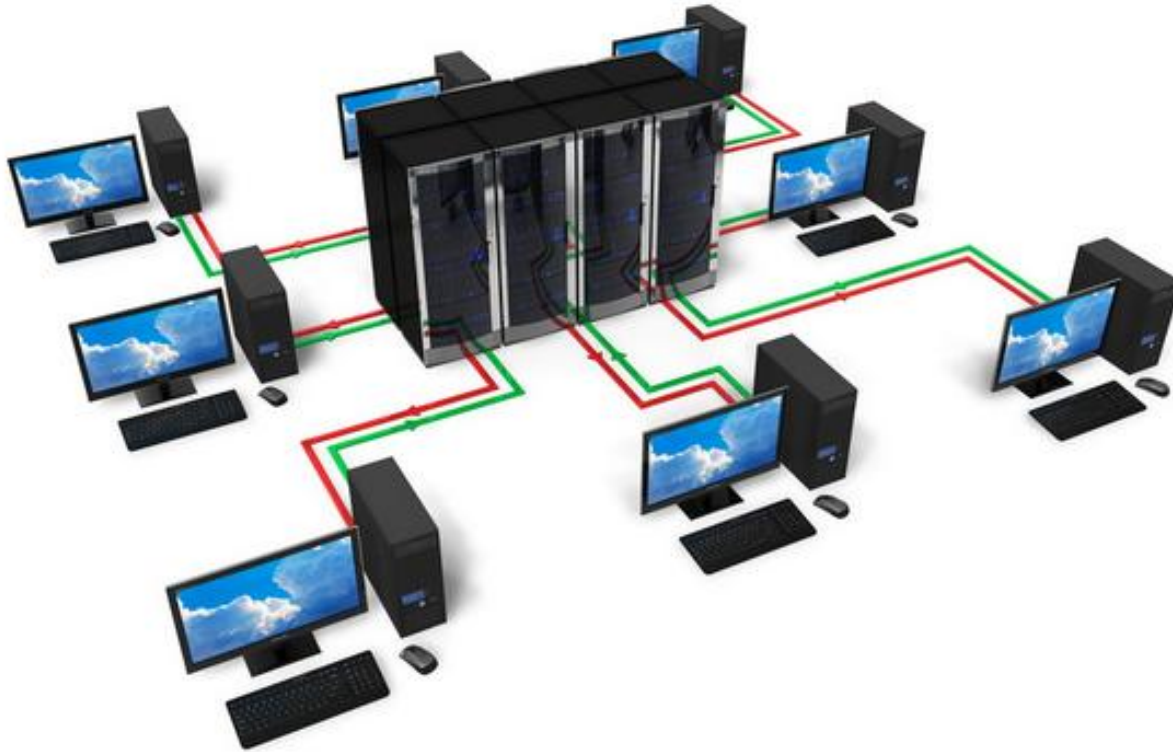
Yuhan Peng¹, Qingyue Liu², Peter Varman²

Department of Computer Science¹

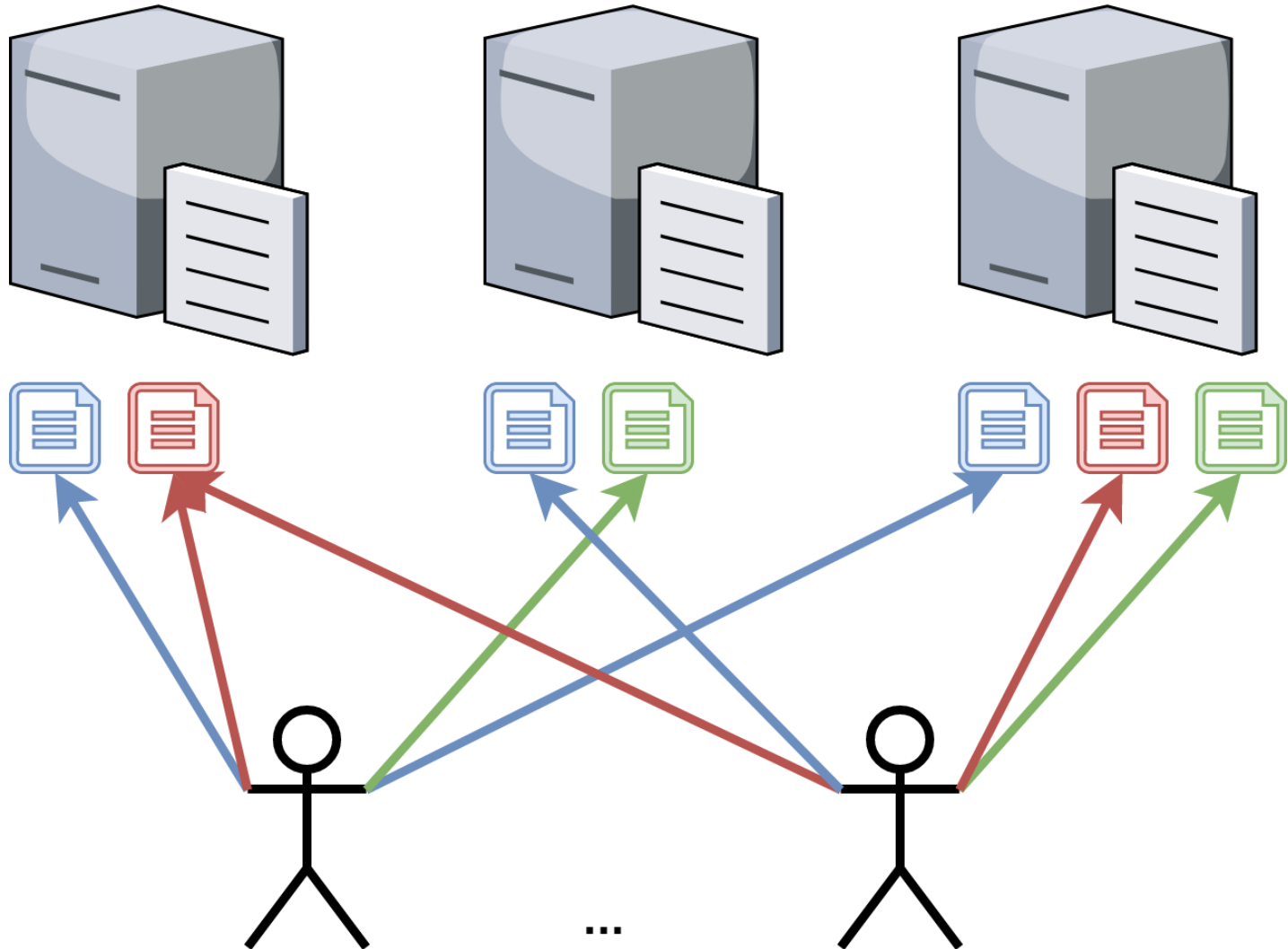
Department of Electrical and Computer Engineering²

Rice University

Clustered Storage Systems



Clustered Storage Systems






Bucket QoS

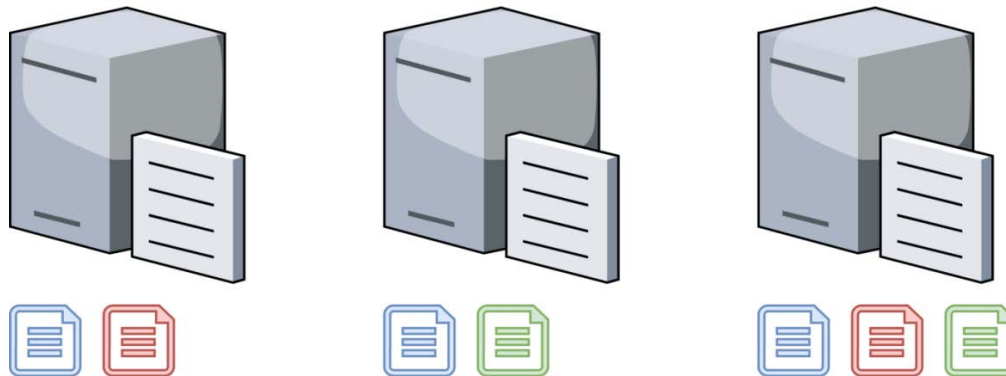
- Bucket: related storage objects
 - Considered as one logical entity
 - Several files or file fragments
- Bucket distributed across multiple storage nodes
- Bucket QoS
 - Differentiate service based on buckets being accessed

Problem Statement

- Provide throughput reservations and limits
 - Reservation: lower bound on bucket's IOPS
 - Limit: upper bound on bucket's IOPS
- QoS requirements are coarse-grained
 - Service time is divided into QoS periods
 - QoS requirements fulfilled in each QoS period

Why Bucket QoS?

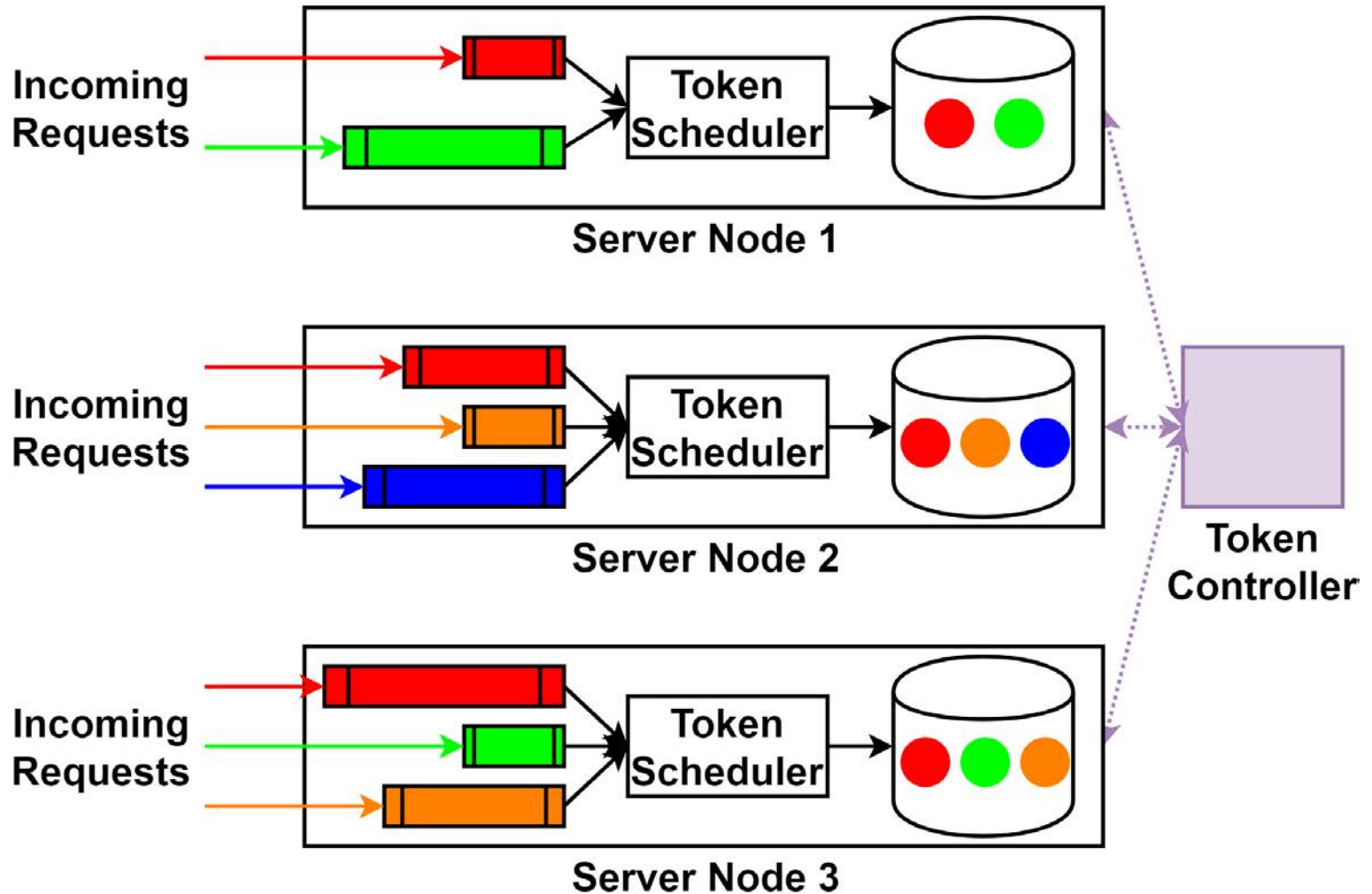
- Owners of the files pay for different services
-  Blue bucket: private files of a free user
 - Low limit
-  Green bucket: media files of a paid user
 - Low latency
-  Red bucket: database files of a paid user
 - High reservation



Challenges

- Buckets are distributed across multiple servers
 - Skewed bucket demands distribution on different servers
 - Time varying bucket demands
- Server capacities
 - May fluctuate with workloads
 - Load on servers can vary spatially and temporally
- QoS requirements are global across servers
 - Many servers can contribute to a bucket's reservations/limit
 - Reservations and limits applied to aggregate bucket service 7

Solution Overview

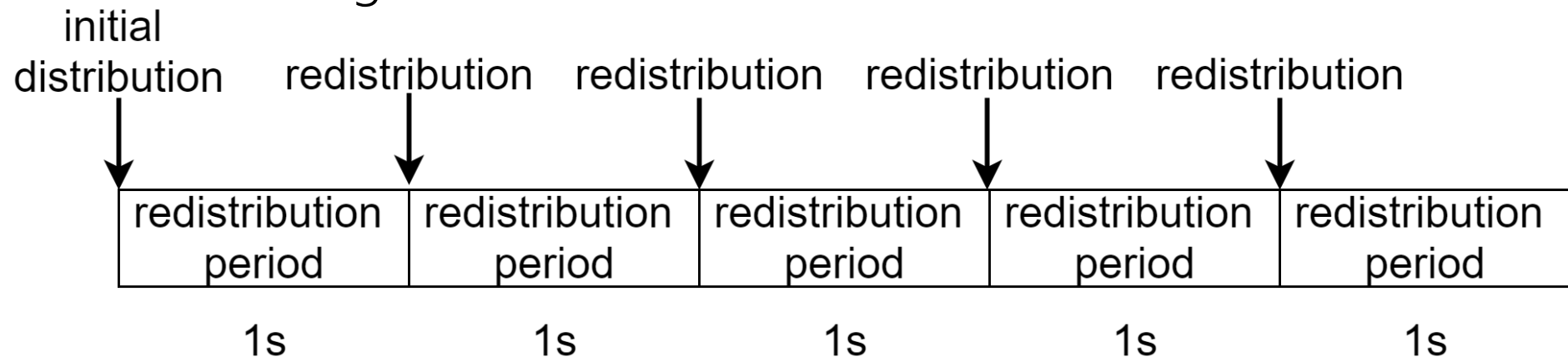


Coarse-grained Approach

- Use tokens to represent the QoS requirements
 - In each QoS period, each bucket allocated some number of reservation and limit tokens
 - Tokens are consumed when requests are scheduled
 - Scheduler gives priority to requests with reservation tokens
 - Requests which have no limit tokens are ignored

Coarse-grained Approach

- Divide each QoS period evenly into redistribution periods
- Controller runs token allocation algorithm to allocate the tokens at the beginning of each redistribution period
- Servers schedule requests during redistribution periods according to the token distribution



QoS period = 5s, 5 redistribution periods per QoS period.

Related Work

- Most existing approaches use fine-grained QoS
 - Request-level QoS guarantees
 - Compute scheduling meta-data (tags) for each request
 - Servers dispatches I/O requests based on the tags
- Our approach is coarse-grained
 - Guarantee QoS over a QoS period
 - Improves our earlier approach: bQueue¹
 - Uses max-flow/linear programming algorithm
 - High overhead, not scalable

¹ Yuhan Peng and Peter Varman, "bQueue: A Coarse-Grained Bucket QoS Scheduler", 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2018), Washington DC, USA.

pShift Algorithm

- Progressive Shift algorithm to allocate tokens
 - Smaller runtime overhead
 - Provably optimal token allocation
 - Can be parallelized
 - Can tradeoff accuracy and time using approximation

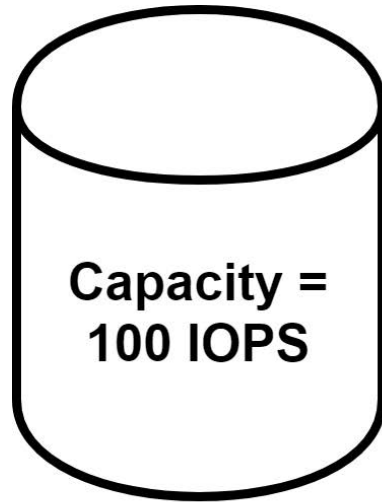
Token Allocation

- Input
 - Total Reservation and Limit tokens to be allocated
 - # reservation/limit tokens not yet consumed
 - Estimated demands
 - Estimated server capacities
- Output
 - Token distribution
 - For each bucket on each server the number of reservation and limit tokens allocated

Token Allocation

- Two basic constraints:
 - Tokens allocated for a bucket B on a server S should not exceed its demand on that server
 - Excess tokens are called strong excess tokens
 - Total number of tokens allocated to a server should not exceed its capacity
 - Excess tokens are called weak excess tokens
- Effective capacity
 - Tokens expected to consumed
 - # non-excess tokens

Illustration: Basic Constraints



100 Red Tokens
50 Green Tokens

Demand(red) = 50
Demand(green) = 50

50 Red Strong Excess Tokens
50 Weak Excess Tokens
Effective Capacity = 100

pShift Algorithm

- Use graph to model the token allocation
 - Start from a configuration with no strong excess tokens
 - Distributing tokens according to the demands
 - Removing most # weak excess tokens while not introducing new strong excess tokens
 - Progressive shifting
- Goal: maximizing the effective system capacity

Progressive Shifting

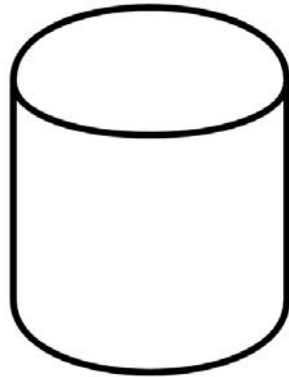
- Moving tokens between servers by shifts
 - Each shift reduce # weak excess tokens, i.e. alleviate the overloaded servers
 - Each shift does not introduce strong excess tokens
 - When no shift can be made, the resulting configuration has the globally maximized effective capacity

Token Movement Map

- Guide the token shifting
- How many tokens can be moved without violating demand restriction

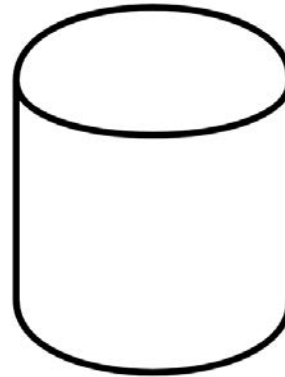
Token Movement Map: Illustration

100 Red Tokens
50 Green Tokens



Demand(red) = 150
Demand(green) = 50

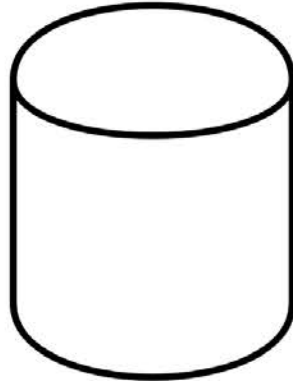
75 Red Tokens
25 Green Tokens



Demand(red) = 100
Demand(green) = 150

Token Movement Map: Illustration

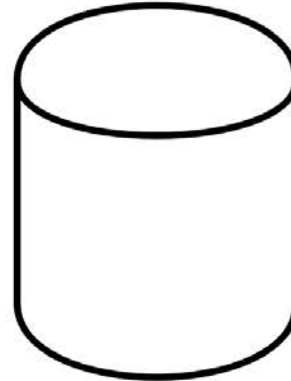
100 Red Tokens
50 Green Tokens



Spare Demand(red) = 50
Spare Demand(green) = 0

Demand(red) = 150
Demand(green) = 50

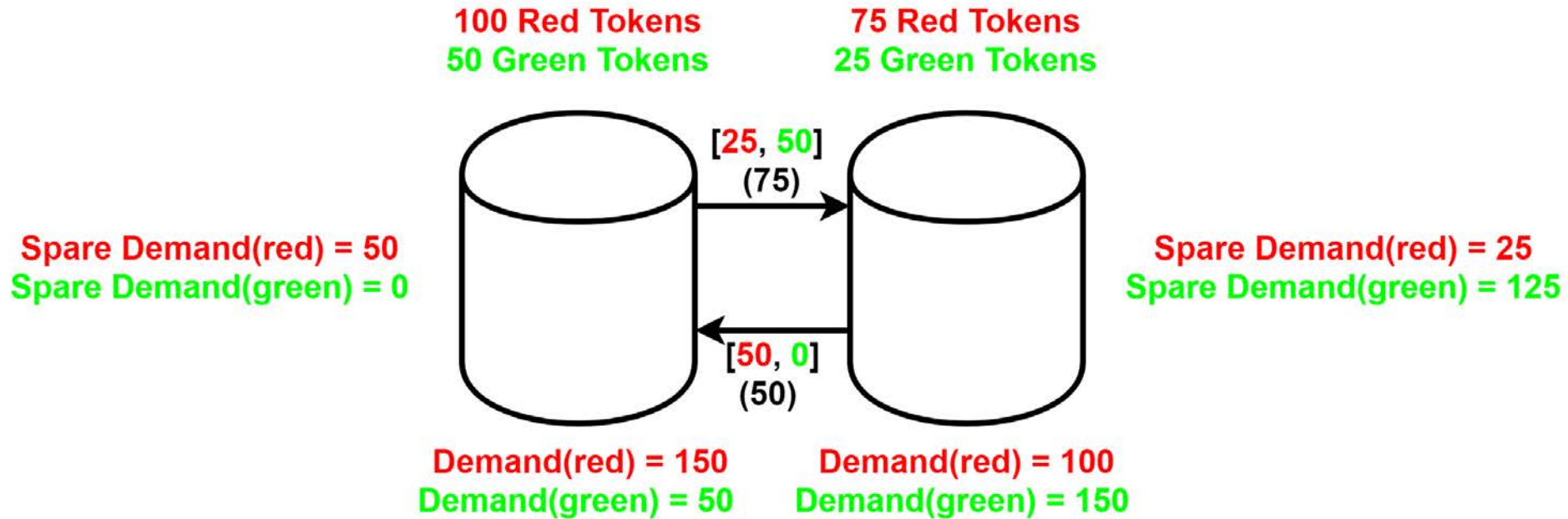
75 Red Tokens
25 Green Tokens



Spare Demand(red) = 25
Spare Demand(green) = 125

Demand(red) = 100
Demand(green) = 150

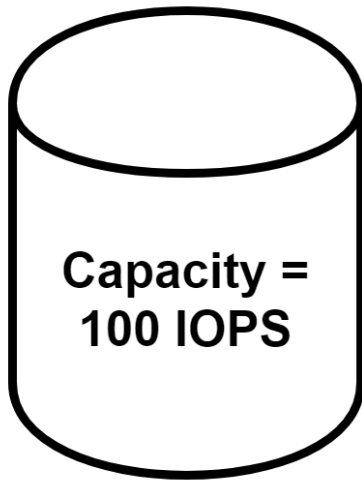
Token Movement Map: Illustration



min(the amount I have, your spare demand)

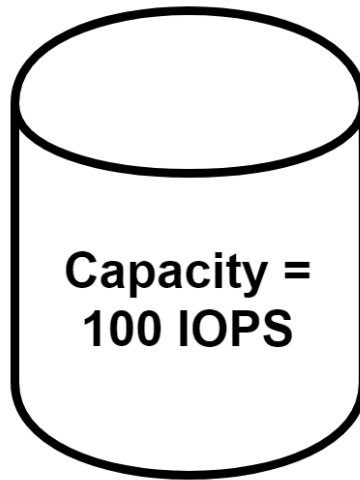
Progressive Shifting: Illustration

125 Red Tokens



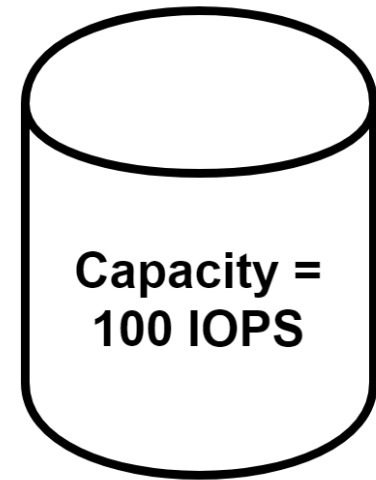
Demand(red) = 150

50 Red Tokens
50 Green Tokens



Demand(red) = 100
Demand(green) = 100

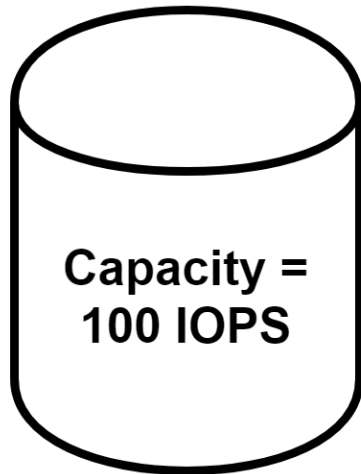
50 Green Tokens



Demand(green) = 100

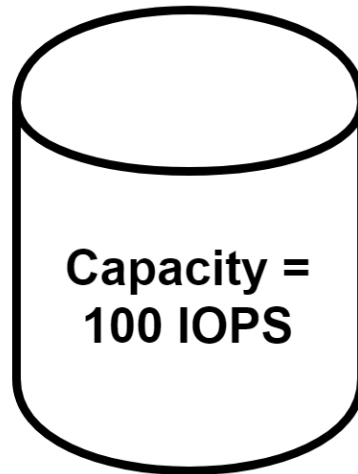
Progressive Shifting: Illustration

125 Red Tokens
(Overloaded by 25)



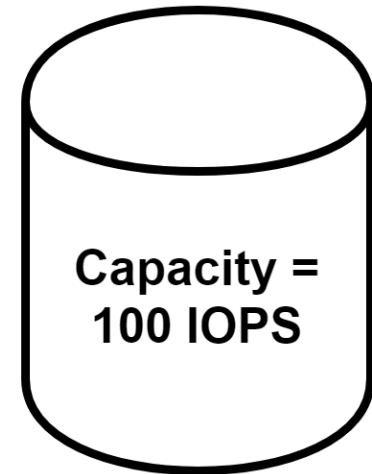
Demand(red) = 150

50 Red Tokens
50 Green Tokens
(Full)



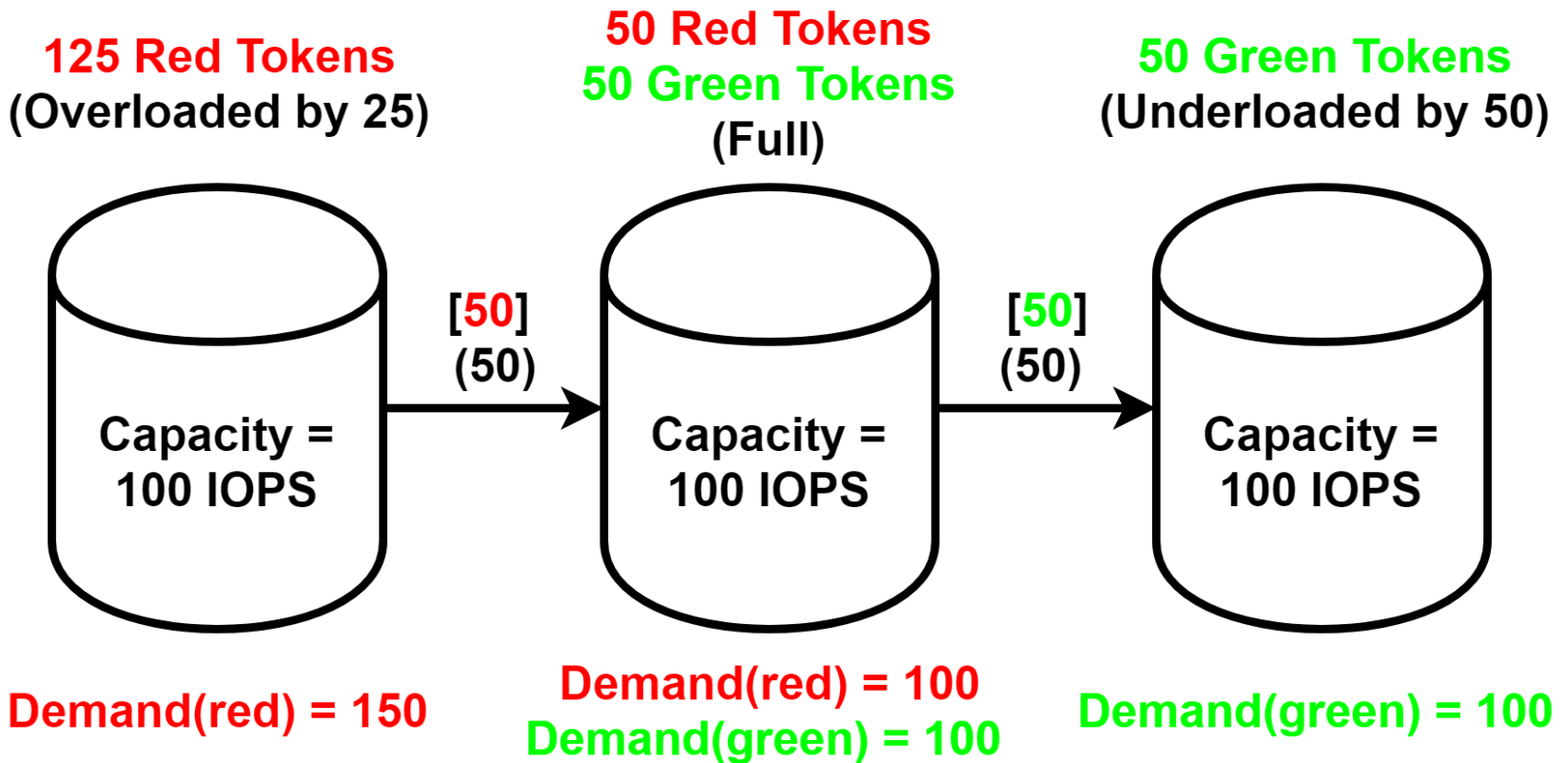
Demand(red) = 100
Demand(green) = 100

50 Green Tokens
(Underloaded by 50)

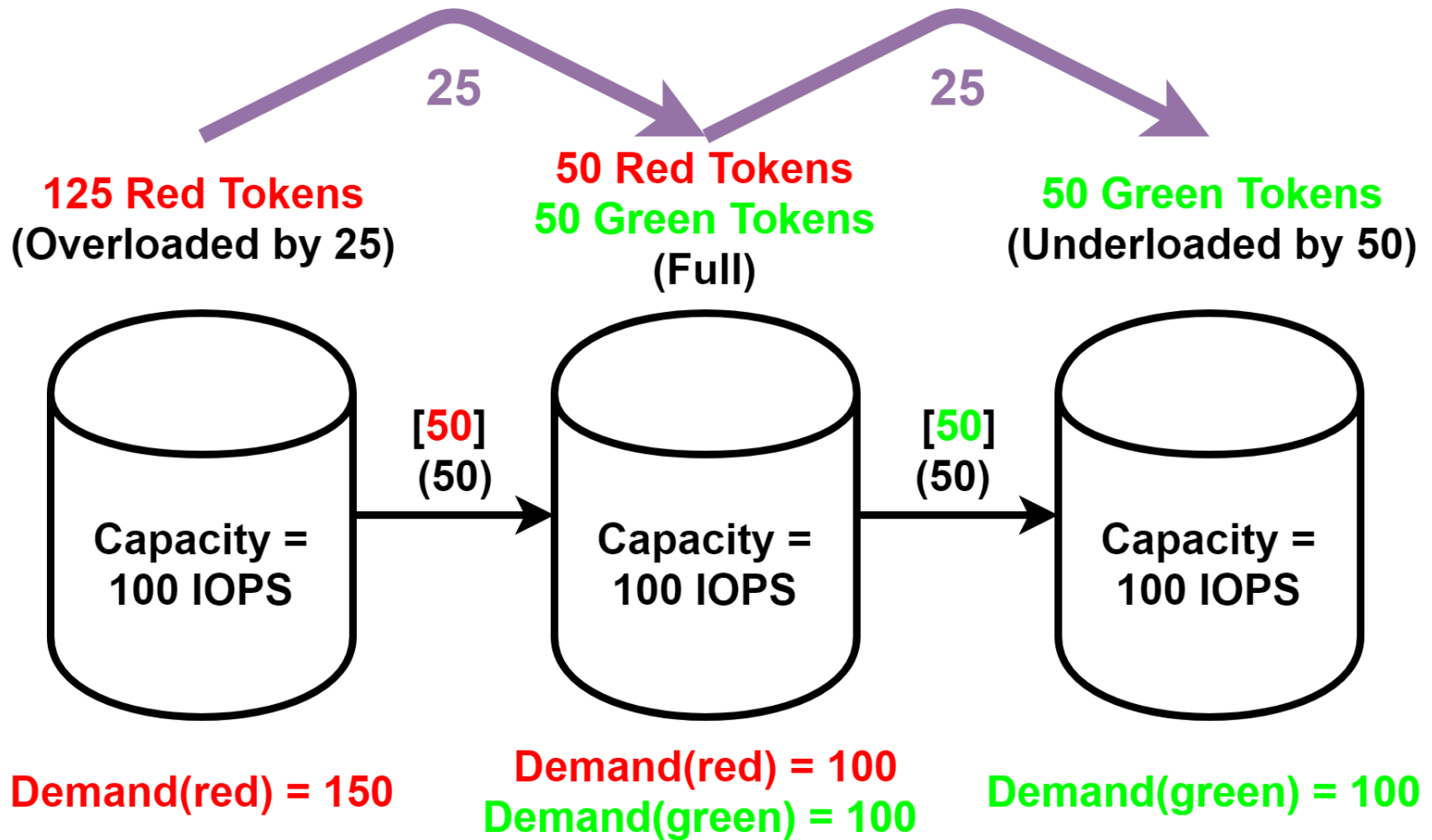


Demand(green) = 100

Progressive Shifting: Illustration

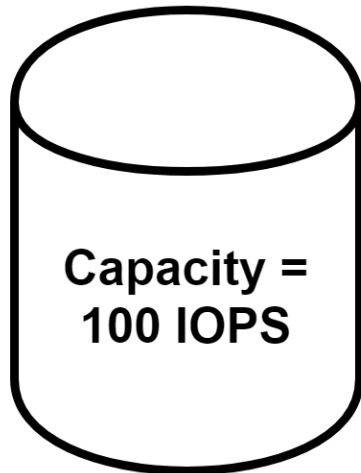


Progressive Shifting: Illustration



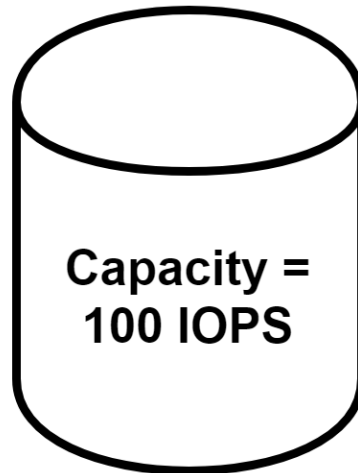
Progressive Shifting: Illustration

100 Red Tokens
(Full)



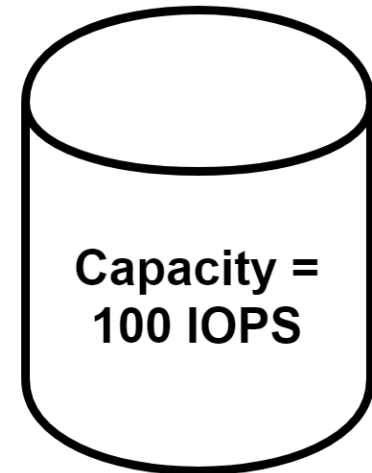
Demand(red) = 150

75 Red Tokens
25 Green Tokens
(Full)



Demand(red) = 100
Demand(green) = 100

75 Green Tokens
(Underloaded by 25)



Demand(green) = 100

Performance Optimizations

- pShift can be parallelized
 - Parallelize the updates on the shift path
- Approximation approach
 - Only consider the buckets with most weights in the token movement map

Performance Evaluation

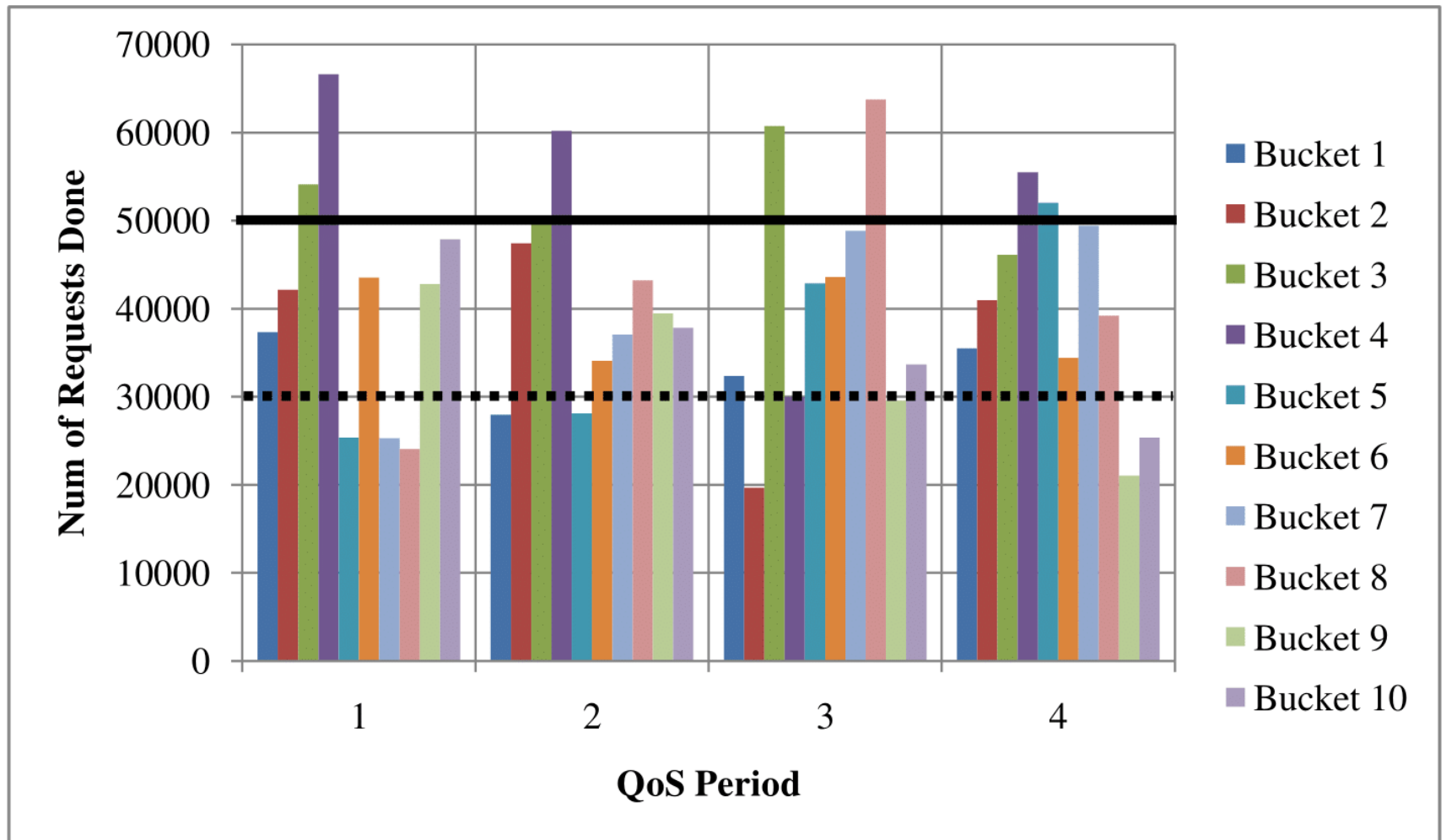
- Implemented a prototype using socket programming library
- Test platform: a small Linux file cluster
- pShift is robust to different runtime demand changes and fluctuations
- pShift has good result in scalability tests

QoS Evaluation

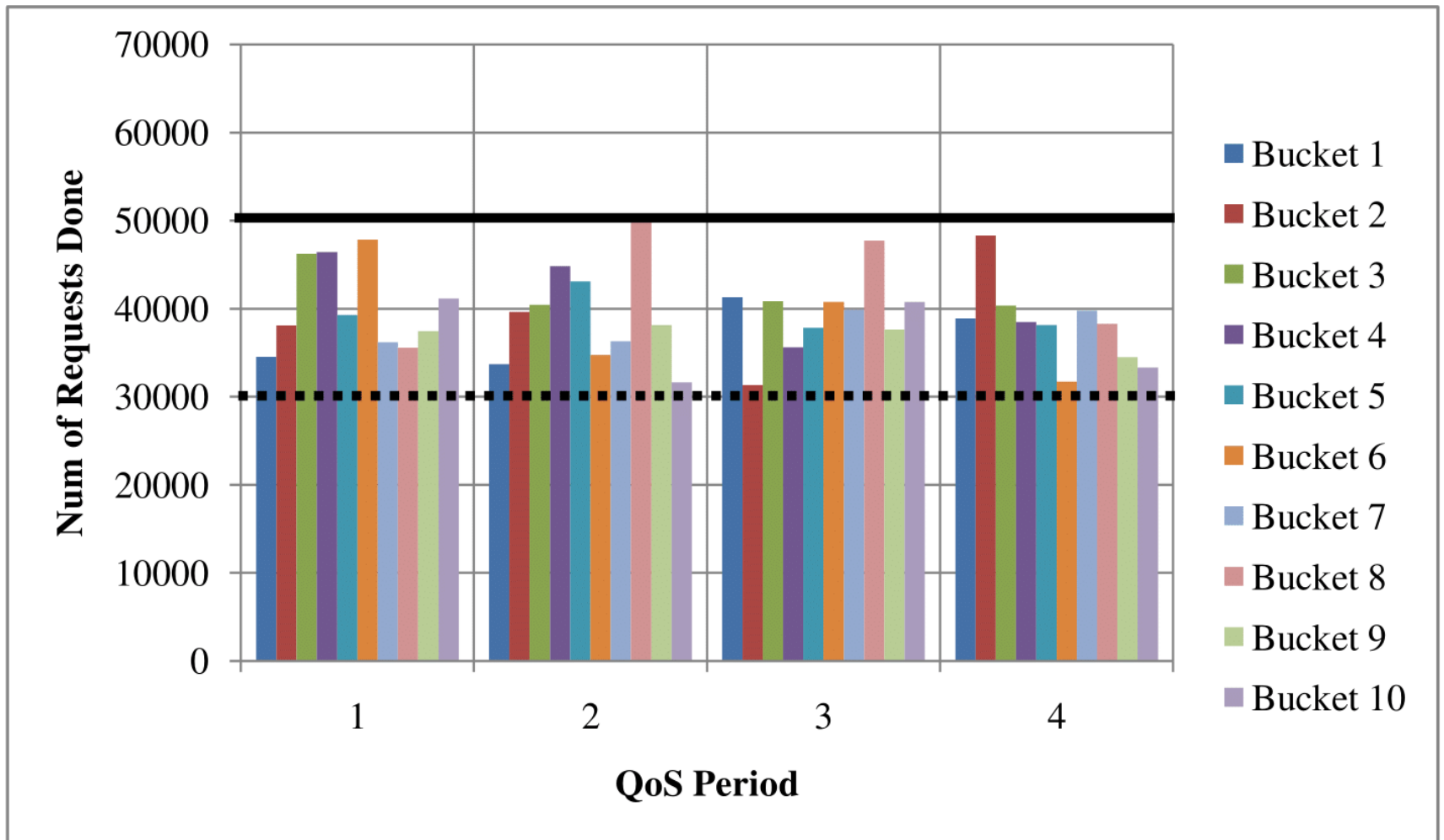
- Configuration 1
 - 8 servers and 10 buckets
 - Distributed memory caching (memcached)
 - Reservations + Limits

Configuration 1

Simple Round Robin (no QoS)



Configuration 1 pShift

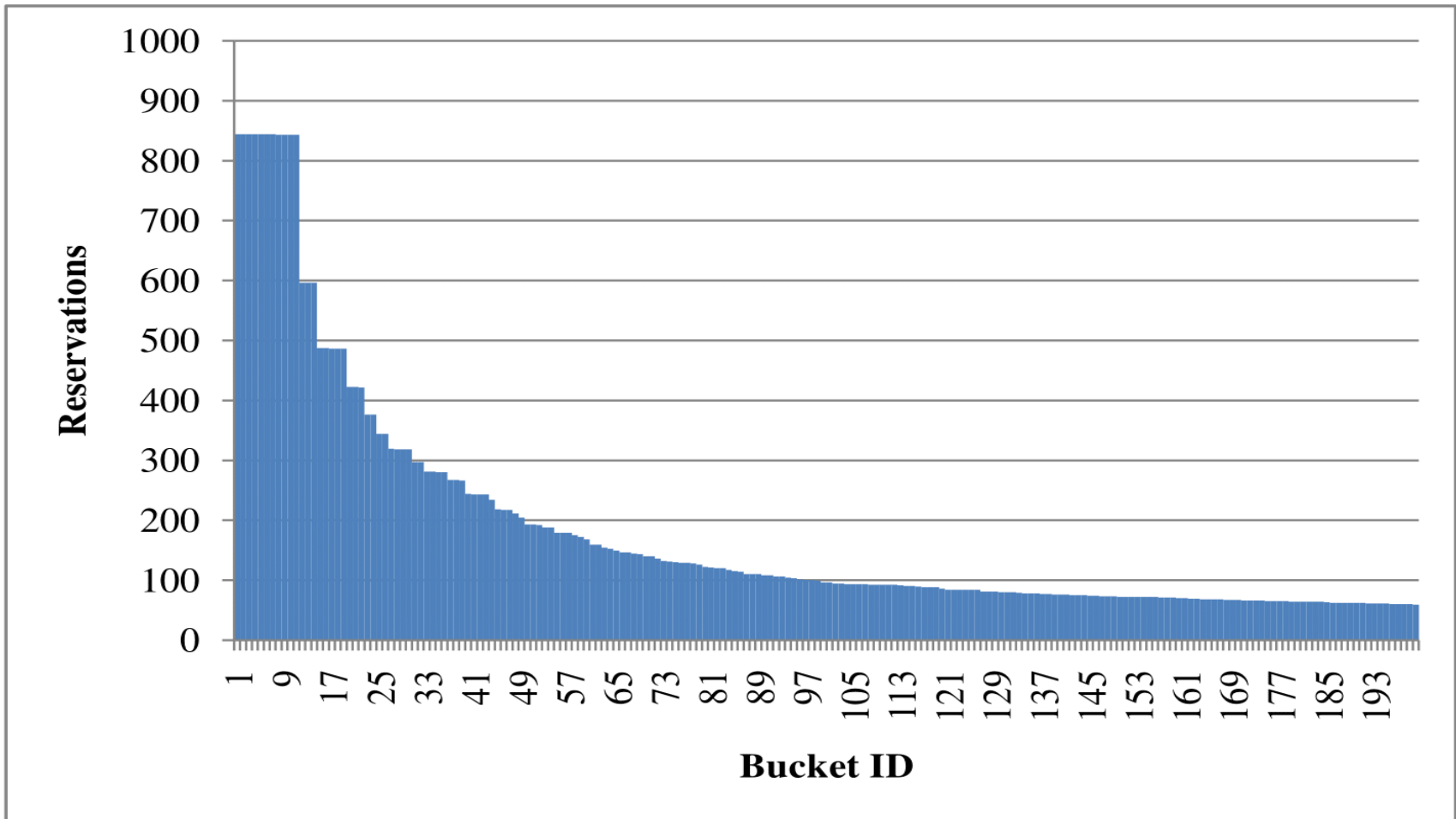


QoS Evaluation

- Configuration 2
 - 8 servers and 200 buckets
 - Random (uncached) reads from a large file
 - Reservations + Limits
 - Workload: Zipf distribution of reservations

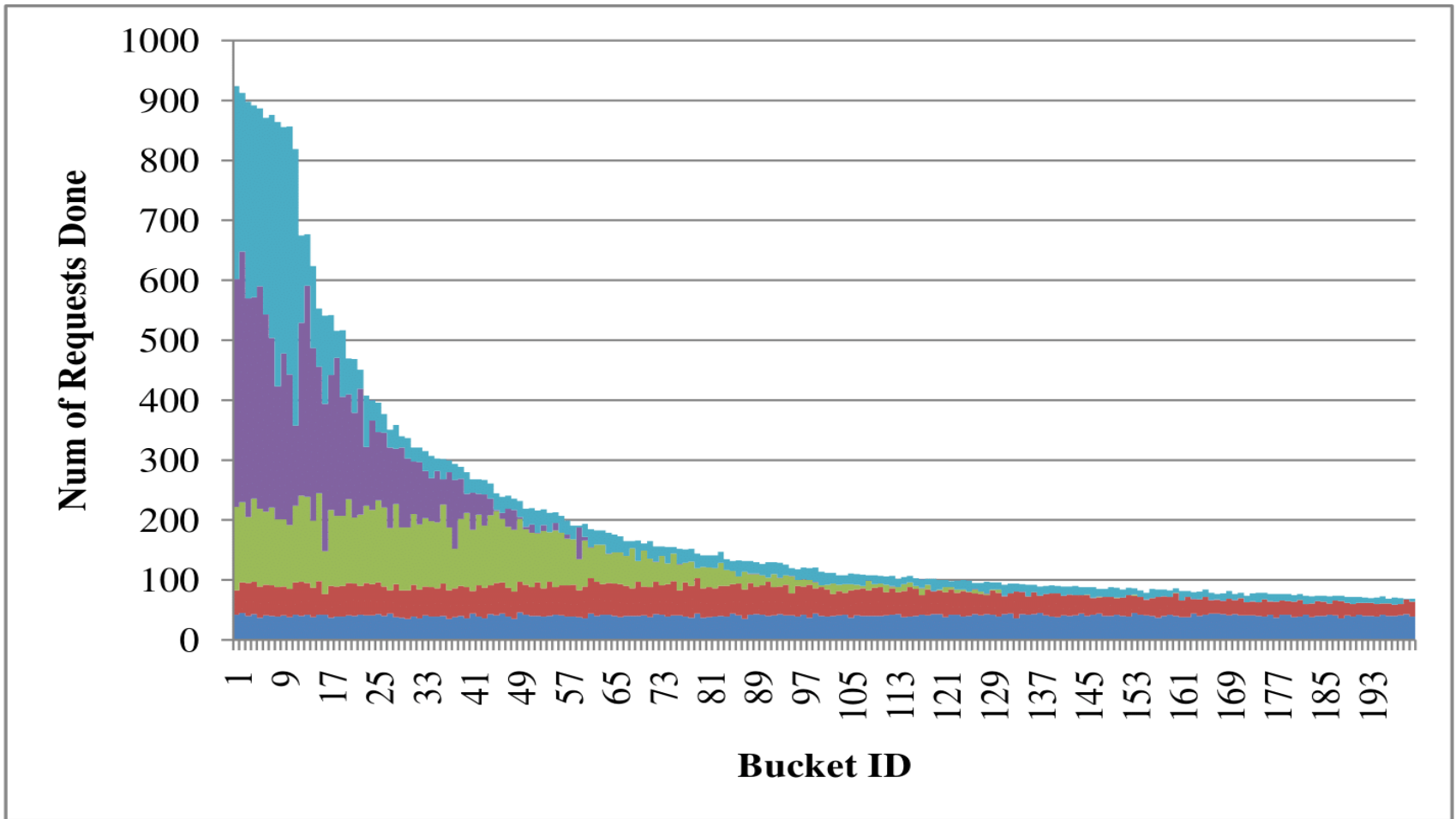
Configuration 2

Reservation Specification



Configuration 2

QoS Result

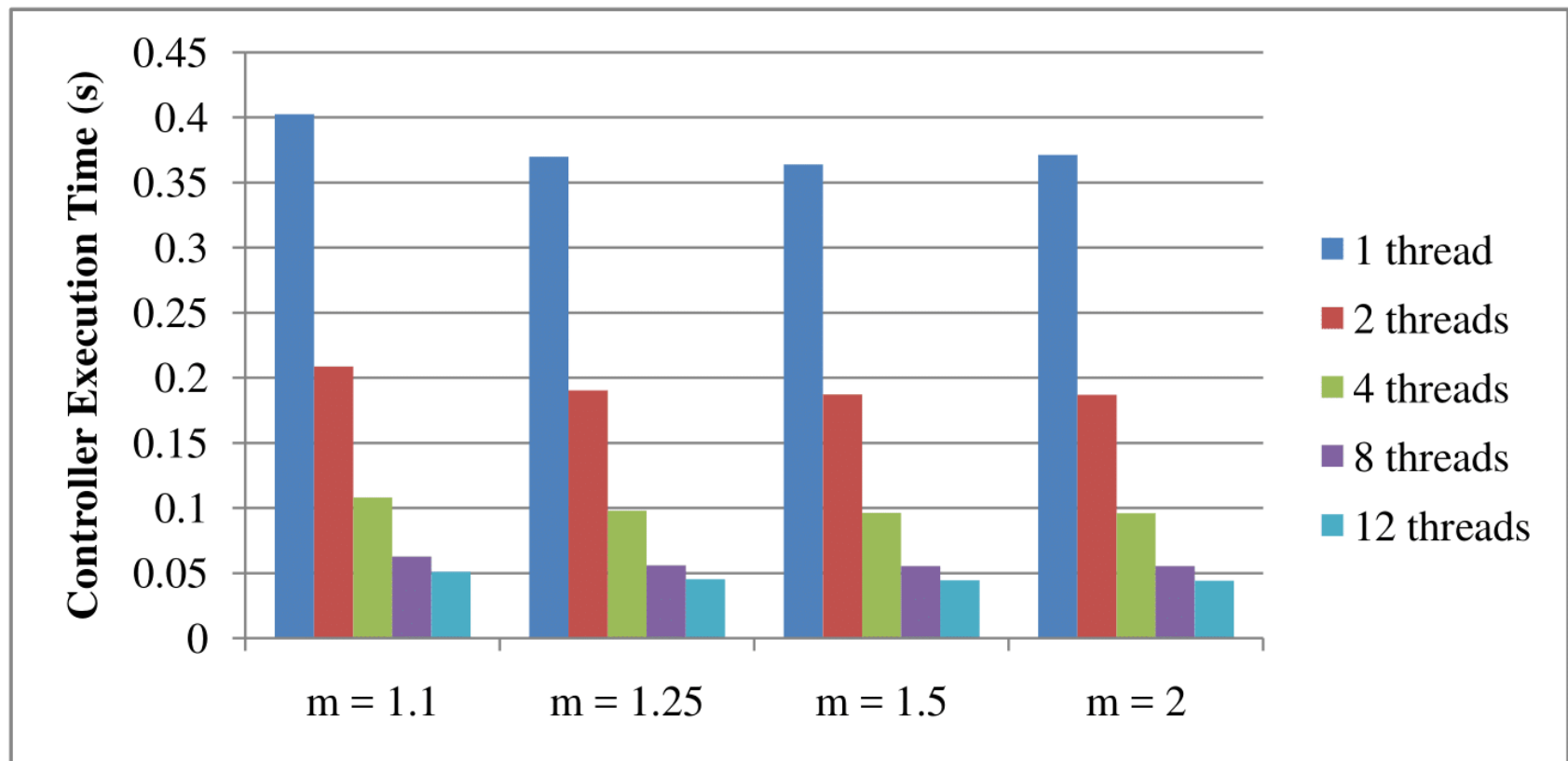


Parallelization Evaluation

- 10000 buckets, 64 servers
- $r = 0.9$
 - 90% of the total cluster capacity is reserved
- m : the ratio of the total demand of each bucket to its reservation ($m \geq 1$)

Parallelization Evaluation

- 5X speedup with 12 threads

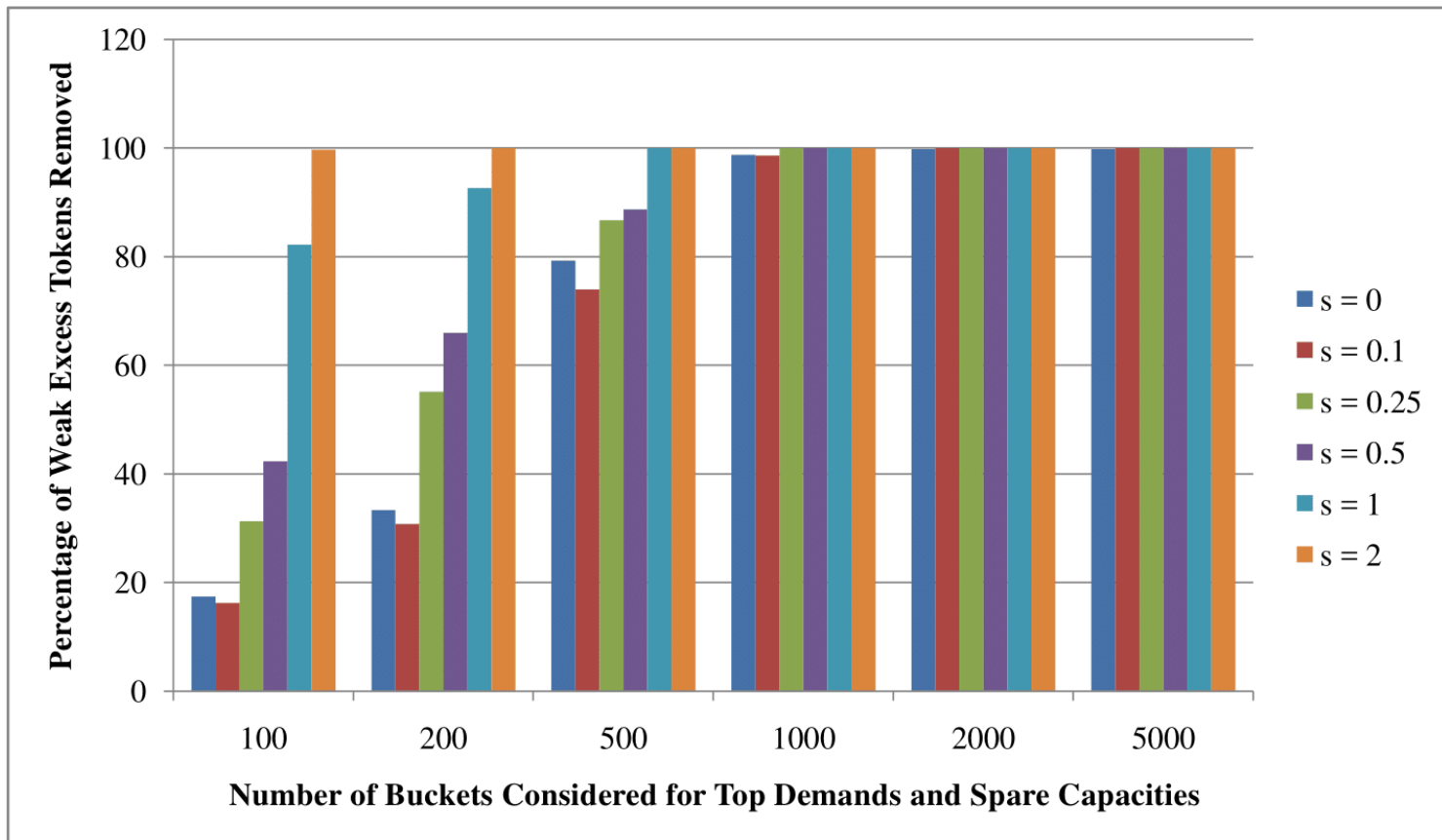


Approximation Evaluation

- 10000 buckets, 64 servers
- $r = 1.0$
 - All of the total cluster capacity are reserved
- $m = 1.1$
 - Each bucket has a total demand 1.1 times to its reservation
- Try different input parameter s
 - Higher s means the variance of reservations is higher

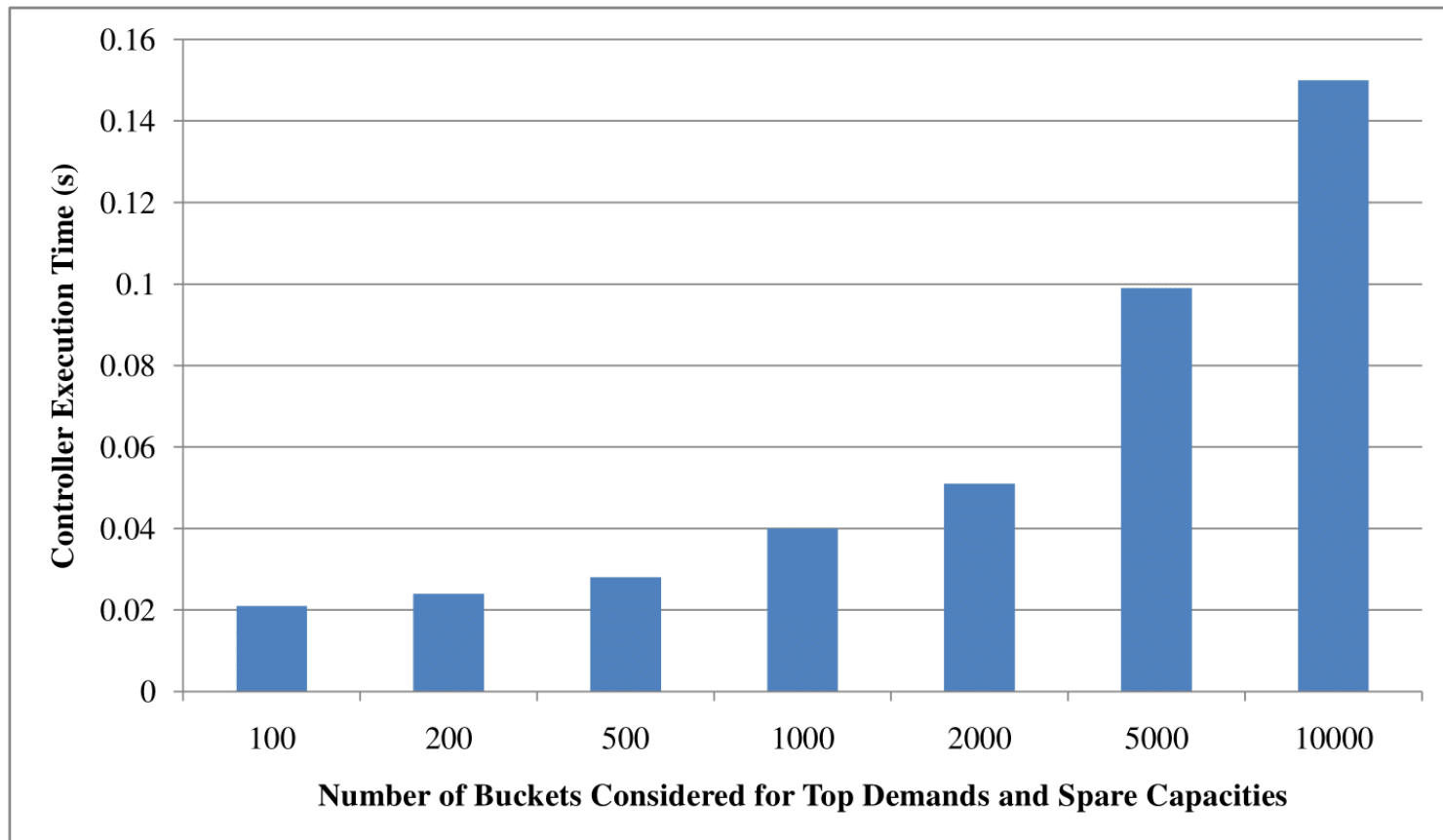
Approximation Evaluation

- Good results even considering only top 5%



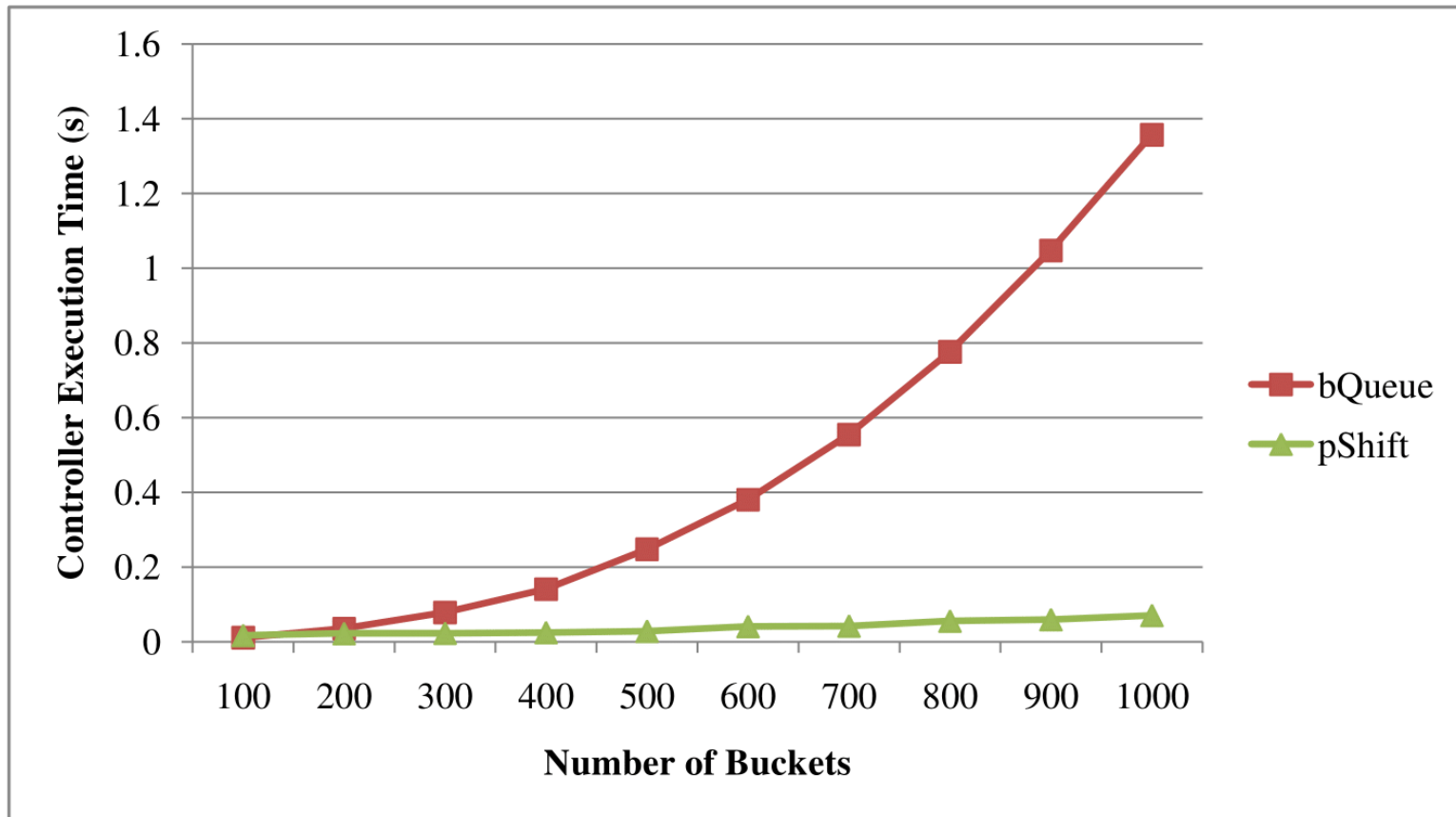
Approximation Evaluation

- Another 5X speedup by considering top 5%



pShift vs bQueue

- 1000 buckets, 64 servers



Summary

- pShift: scalable token allocator for QoS
 - Token allocation through progressive shifting
 - Proven to be optimal
 - Small runtime overhead
 - Can be parallelized & approximated
- Future Work
 - Support other QoS requirements such as latency

Backup Slide:

Fine-grained v.s. Coarse-grained

	Fine-grained Approaches	Coarse-grained Approaches
How QoS requirements are enforced	Meta-data on each request (e.g. tags)	Global control information (e.g. tokens)
Implementation Complexity	High	Low
Server Schedulers	Complicated	Simple

Backup Slide: Demand Estimation

- Linear extrapolation
 - N requests received in last redistribution period
 - M requests outstanding at the redistribution
 - Q more redistribution periods left
 - demand = $N * Q + M$
- Significant demand changes will be caught up in the next redistribution period

N new incoming requests



N

N

N

N

M requests outstanding

Backup Slide: Capacity Estimation

- Linear extrapolation (again)
 - R requests completed in last redistribution period.
 - Q more redistribution periods left.
 - residual capacity = $R * Q$.

