Appeared in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*

Performance Isolation and Fairness for Multi-Tenant Cloud Storage

David Shue*, Michael J. Freedman*, and Anees Shaikh†
*Princeton University, †IBM TJ Watson Research Center

Abstract

Shared storage services enjoy wide adoption in commercial clouds. But most systems today provide weak performance isolation and fairness between tenants, if at all. Misbehaving or high-demand tenants can overload the shared service and disrupt other well-behaved tenants, leading to unpredictable performance and violating SLAs.

This paper presents Pisces, a system for achieving datacenter-wide *per-tenant* performance isolation and fairness in shared key-value storage. Today's approaches for multi-tenant resource allocation are based either on per-VM allocations or hard rate limits that assume uniform workloads to achieve high utilization. Pisces achieves per-tenant weighted fair shares (or minimal rates) of the aggregate resources of the shared service, even when different tenants' partitions are co-located and when demand for different partitions is skewed, time-varying, or bottlenecked by different server resources. Pisces does so by decomposing the fair sharing problem into a combination of four complementary mechanisms—partition placement, weight allocation, replica selection, and weighted fair queuing—that operate on different time-scales and combine to provide system-wide max-min fairness.

An evaluation of our Pisces storage prototype achieves nearly ideal (0.99 Min-Max Ratio) weighted fair sharing, strong performance isolation, and robustness to skew and shifts in tenant demand. These properties are achieved with minimal overhead (<3%), even when running at high utilization (more than 400,000 requests/second/server for 10B requests).

1. Introduction

An increasing number and variety of enterprises are moving workloads to cloud platforms. Whether serving external customers or internal business units, cloud platforms typically allow multiple users, or *tenants*, to share the same physical server and network infrastructure, as well as use common platform services. Examples of these shared, multi-tenant services include key-value stores, block storage volumes, SQL databases, message queues, and notification services. These leverage the expertise of the cloud provider in building, managing, and improving common services, and enable the statistical multiplexing of resources between tenants for higher utilization.

Because they rely on shared infrastructure, however, these services face two key, related issues:

- ***Multi-tenant interference and unfairness:*** Tenants simultaneously accessing shared services contend for resources and degrade performance.
- ***Variable and unpredictable performance:*** Tenants often experience significant performance variations, e.g., in response time or throughput, even when they can achieve their desired mean rate [8, 16, 33, 35].

These issues limit the types of applications that can migrate to multi-tenant clouds and leverage shared services. They also inhibit cloud providers from offering differentiated service levels, in which some tenants can pay for performance isolation and predictability, while others choose standard "best-effort" behavior.

Shared back-end storage services face different challenges than sharing server resources at the virtual machine (VM) level. These stores divide tenant workloads into disjoint partitions, which are then distributed (and replicated) across different service instances. Rather than managing individual storage partitions, cloud tenants want to treat the entire storage system as a single black box, in which *aggregate* storage capacity and request rates can be elastically scaled on demand. Resource contention arises when tenants' partitions are co-located, and the degree of resource sharing between tenants may be significantly higher and more fluid than with VM resource allocation. Particularly, as tenants may use only a small fraction of a server's throughput and capacity,[1] restricting nodes to a few tenants may leave them highly underutilized.

To improve predictability for shared storage systems with a high degree of resource sharing and contention, we target global ***max-min fairness*** with ***high utilization***. Under max-min fairness, no tenant can gain an unfair advantage over another when the system is loaded, i.e., each tenant will receive its weighted fair share. Moreover, given its work-conserving nature, when some tenants use less than their full share, unconsumed resources are divided among the rest to ensure high utilization. While our mechanisms may be applicable to a range of services with shared-nothing architectures [31], we focus our design and evaluation on a replicated key-value storage service, which we call *Pisces* (*P*redictable *S*hared *C*loud *S*torage).

[1] Indeed, at today's Amazon S3 prices, a single server handling 50,000 GET reqs/second would cost $180/hour in request pricing alone.

Providing fair resource allocation and isolation at the *service* level is confounded by variable demand to different service partitions. Even if tenant objects are uniformly distributed across their partitions, per-object demand is often skewed, both in terms of request rate and size of the corresponding (read or write) operations. Moreover, different request workloads may stress different server resources (e.g., small requests may be interrupt limited, while large requests are bandwidth limited). In short, simply assuming that each tenant requires the same proportion of resources per partition can lead to unfairness and inefficiency. To address these issues, Pisces makes a number of contributions:

(1) Global fairness. To our knowledge, Pisces is the first system to provide per-tenant fair resource sharing across all service instances. Further, as total system capacity is allocated to tenants based on their normalized weights, such a max-min fair system can also provide minimal performance guarantees given sufficient provisioning. In comparison, recent commercial systems that offer request rate guarantees (i.e., Amazon DynamoDB [1] do not provide fairness, assume uniform load distributions across tenant partitions, and are not work conserving.

(2) Novel mechanism decomposition. Pisces introduces a clean decomposition of the global fairness problem into four mechanisms. While operating on different timescales and with different levels of system-wide visibility, these mechanisms complement one another to ensure fairness under resource contention and variable demand.

(i) Partition Placement ensures a fair allocation by (re)-assigning tenant partitions to nodes (long timescale).

(ii) Weight Allocation distributes overall tenant fair shares across the system by adjusting local per-tenant weights at each node (medium timescale).

(iii) Replica Selection load-balances requests between partition replicas in a weight-sensitive manner (real-time).

(iv) Weighted Fair Queuing at service nodes enforces performance isolation and fairness according to the local tenant weights (real-time).

(3) Novel algorithms. We introduce several novel algorithms to implement Pisces's mechanisms. These include a reciprocal swapping algorithm for weight allocation that shifts weights when tenant demand for local resources exceed their local share, while maintaining global fairness. We use a novel application of optimization-inspired congestion control for replica selection, which complements weight allocation by distributing load over partition replicas in response to per-node latencies. Finally, to manage different resource bottlenecks on contended nodes, we enforce *dominant resource fairness* [9] between tenants at the node level, while providing max-min fairness at the service level. To do so, we extend traditional deficit-weighted round robin queuing to handle per-tenant multi-resource scheduling.

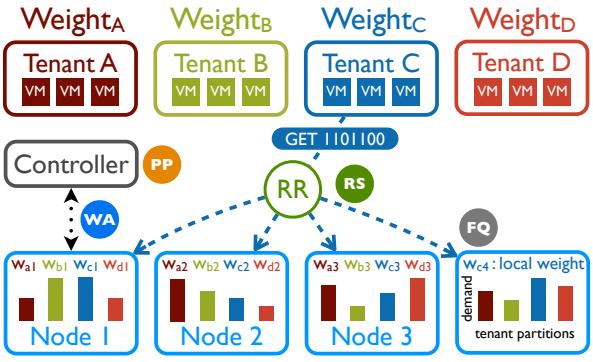


Figure 1: Pisces multi-tenant storage architecture.

(4) Low overhead and high utilization. Pisces is designed to support high server utilization, both high request rates (100,000s requests per second, per server) and full bandwidth usage (Gbps per server). To do so, its mechanisms must apply at real time without inducing significant throughput degradation. Through careful system design, our prototype achieves <3% overhead for 1KB requests and actually outperforms the unmodified, non-fair version for small requests. Commercial systems like DynamoDB, on the other hand, typically target lower rates (e.g., more than 10,000 reqs/s requires special arrangements [2]).

Through an extensive experimental evaluation, we demonstrate that Pisces significantly improves the multi-tenant fairness and isolation properties of our key-value store, built on Membase [3], across a range of tenant workloads. We also show that its replica selection and rebalancing policies optimize system performance, even as workloads shift dynamically. While this paper frames the partition placement problem and implements a simple greedy placement algorithm for our evaluation, we do not fully explore and evaluate its design space.

2. Architecture and Design

We consider multi-tenant cloud services with partitioned workloads (i.e., data sets), where each partition is disjoint but may be replicated on different service nodes. Client requests are routed to the appropriate node based on the partition mapping, and the replica selection policy in use. Ultimately, request arbitration for fairness and isolation between tenants occurs at the service nodes.

Figure 1 shows the high-level architecture of Pisces, a key-value storage service that provides system-wide, per-tenant fairness and isolation. Pisces provides the semantics of a persistent map between opaque keys (bit-strings) and unstructured data values (binary blobs) and supports simple key lookups (get), modifications (set), and removals (delete). To partition the workload, the keys are first hashed into a fixed-size key space, which is then subdivided into disjoint segments.

Pisces enforces per-tenant fairness at the system-wide level. As shown in Figure 1, each tenant t is given a

single, global weight w_t that determines its fair share of overall system resources (i.e., throughput). These weights are generally set according to the tenant's service-level objective (SLO). To support service models with rate guarantees, the provider can simply convert a specified rate into a corresponding system resource proportion (weight) given the current capacity. The service provider can also adjust the tenant weights, e.g., in response to new tenant requirements or changes in system capacity.

Pisces allows a cloud service provider to offer a flexible service model, in which customers pay for their consumed storage capacity, with an optional additional tiered charge for an "assured rate" service. Assured service users can reserve a minimum service throughput—which, when normalized, translates to a minimum fair share of the global service throughput—with the price dependent upon this rate. The system ensures this minimum rate, yet also allows users to exploit unused capacity (perhaps while charging an additional "overage" fee). Such multi-tiered charging is common in many Internet contexts, e.g., network transit and CDNs often use burstable billing and charge differently for rate commitments and overages.

While Pisces's general mechanisms should extend to other shared storage systems, our current prototype makes some simplifying assumptions. It does not support more advanced queries, such as scans over keys. It is designed primarily to serve keys out of an in-memory cache for high throughput, and only asynchronously writes data to disk (much like Masstree [19] and the MyISAM storage engine in MySQL). We do not focus on consistency issues, and assume that a separate protocol keeps the partition replicas in sync.[2] Further, we assume a well-provisioned network (e.g., one with full bisection bandwidth [5]); we do not deal explicitly with in-network resource sharing and contention, which has been considered by complementary work [26, 27, 28]. Finally, we assume a reasonably stable tenant workload distribution. While Pisces can support highly-skewed demand distributions across partitions and can handle short-term fluctuations in demand for particular keys, we assume that the relative popularity of entire partitions shifts relatively slowly (e.g., on the order of minutes). This provides the system with sufficient time to rebalance partition weights when needed.

2.1 Life of a Pisces Request

Before a tenant can read (get) and write (set) data in the system, a central controller first performs *partition placement (PP)* to assign its data partitions (and their replicas) to service nodes. Each tenant has its own key space, but partitions from different tenants may be co-located on the same service node. The controller then disseminates the partition mapping information to each of the request routers. These request routers can be implemented in client libraries running on the tenants' (virtual) machines, or deployed on intermediate machines (as illustrated in Figure 1). The controller also translates each tenant's global fair-share into local shares at individual storage nodes through *weight allocation (WA)*.

When a client tenant, C, issues a request to Pisces, the request router dispatches the request based on its key (e.g., 1101100) and partition location to an appropriate server. If enabled, *replica selection (RS)* allows the request router to flexibly choose which replica to use (e.g., Nodes 1 or 4). Otherwise, the router directs the request to the primary partition. Once the replica has been selected, the router adds the request to a windowed queue of outstanding operations—one queue per server, per tenant.

Since partitions from multiple tenants may reside on this node, the tenants' requests will contend for resources. To enforce fairness and isolation, the service node applies *fair queuing (FQ)* to schedule tenant requests according to each tenant's local weight ($w_{c,4}$). When a request reaches the server, the server adds the request to a queue specific to that tenant (C). Every "round" of execution, the server allocates tokens to each tenant according to its local weight ($w_{c,4}$), which it then consumes when processing requests from the tenant queues. If the request consumes more than the allocated resources, it must complete on a subsequent round after tenant C's tokens have been refilled. This guarantees that each tenant will receive its local fair share in a given round of work, if multiple tenants are active. Otherwise, tenants can consume excess resources left idle by the others without penalty.

2.2 System-wide Fair Sharing: Example

The challenge of achieving system-wide fairness can be illustrated with a few scenarios, as shown in Figure 2. From these, we derive design lessons for how Pisces should (1) place partitions to enable a fair allocation of resources, (2) allocate local weights to maintain global fairness, (3) select replicas to achieve high utilization, and (4) queue requests to enforce fairness. In the examples, two tenants (A and B) with equal global shares access two Pisces nodes with equal capacity (100 kreq/s). Each tenant should receive the same aggregate share of 100 kreq/s.

Partitions should be placed with respect to demand and node capacity constraints. For per-tenant fairness to be *feasible*, there must exist some assignment of tenant partitions that can satisfy the global tenant shares without violating node capacities. Not all placements lead to a feasible solution, however, as shown in Figure 2a. Here, each tenant has the same skewed distribution of partition demand: 40, 30, 20, and 10. Arbitrary partition assignment can easily lead to capacity overflow: with A and B both demanding 60 kreq/s for the partitions on the

[2] Our implementation is built on Membase [3], which asynchronously replicates from a partition's primary copy to its backup(s), and by default reads only from the primary for strong consistency.

first node, each tenant receives 10 kreq/s less than their global share. If we take partition demand into account and shuffle tenant *B*'s partitions between the nodes, then we can achieve a fair and feasible placement. Although the skew in this example may be extreme, Internet workloads often exhibit a power-law (or Zipf) distribution across keys, which can induce skewed partition demand.

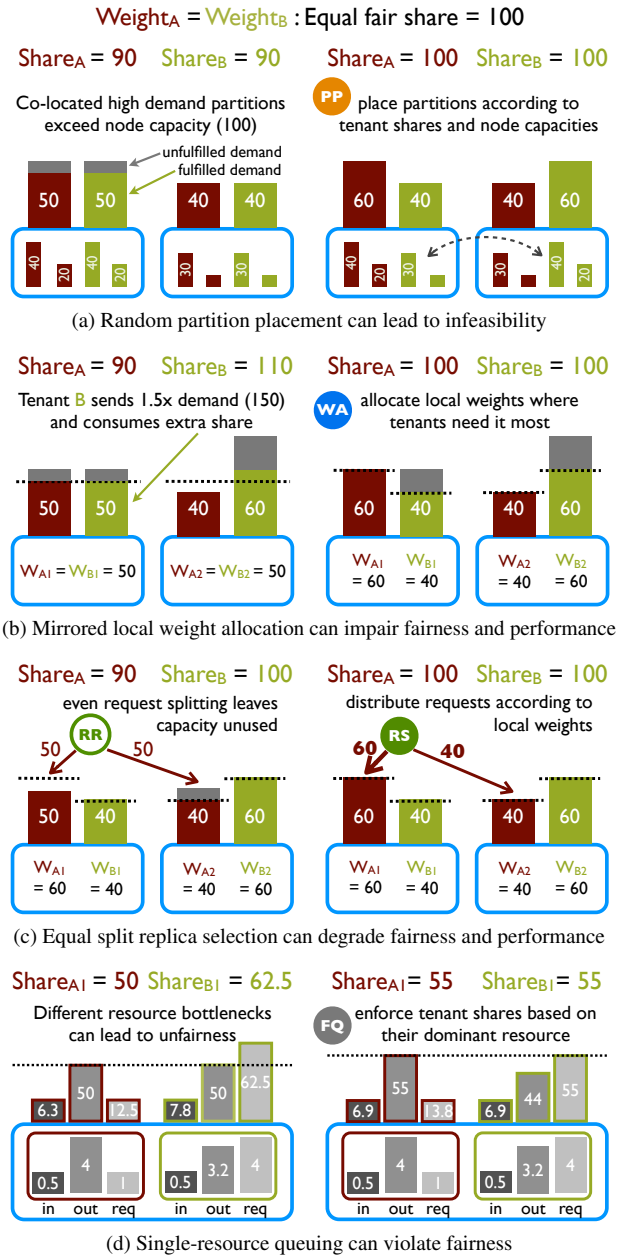
Local weights should give tenants throughput where they need it most. Even with a feasible fair partition placement, if the local weights simply mirror the global (uniform: 50 each) weights, as in Figure 2b, global fairness may still suffer. Although fair queuing allows tenant *B* to consume more than its local share (60 > 50) at the first node when *A* consumes less (40 < 50), if tenant *A* increases its demand, it will consume its remaining local allocation. This will increase its global share and eat into tenant *B*'s global share. However, if we adjust each tenant's local weights to match their demand (60/40 and 40/60), we can preserve fairness even under excess load.

Replicas should be selected in a weight-sensitive manner. When replica selection is enabled, the fairness problem becomes easier, in general, since each replica only receives a *fraction* of the original demand. By spreading partition demand across multiple servers, replica selection produces smoother distributions that makes partitions easier to place. Further, once placed, the reduced per-replica demand is easier to match with local weights.

However, the replica selection policy must be carefully tuned, otherwise fairness and utilization may still diverge from the system-wide goals. In Figure 2c, tenant *A* can send requests to replicas on either node. However, since its local weights are skewed to match the resident partition demand, simple replica-selection policies are insufficient to exploit the variation between the local weights. For example, equal-split round robin would lead to a 10 kreq/s drop in *A*'s global share. Instead, by adjusting per-tenant replica selection proportions to reflect the local weights, we can fully exploit replicated reads for both improving performance and facilitating fairness.

Request queuing should enforce dominant resource fairness. Up to this point, our examples have illustrated multiple tenants with identical weights competing for identical request rates, which implicitly assumes that all requests have equivalent cost. In practice, requests may be of different input or output sizes, and can activate different bottlenecks in the system (e.g., small requests may be bottlenecked by server interrupts, while large requests may be bottlenecked by network bandwidth). Thus, each tenant's workload may vary accordingly across the different resources, as seen in Figure 2d.

Each tenant's resource profile shows the relative proportion of each resource—bytes in, bytes out, and number of requests—that the tenant consumes. These resources are the likely limiting factor for writes, reads, and small



(a) Random partition placement can lead to infeasibility

(b) Mirrored local weight allocation can impair fairness and performance

(c) Equal split replica selection can degrade fairness and performance

(d) Single-resource queuing can violate fairness

Figure 2: Illustrating the difficulty in achieving system-wide fairness. Both tenants have the same global weight (equal fair share); *Share* is the normalized rate actually achieved by each tenant. Left-hand figures correspond to settings lacking Pisces's mechanisms; right-hand figures apply its techniques.

requests, correspondingly. Tenant *A* is read bandwidth bound, consuming 4% of the out bandwidth for every 1% of request capacity it uses. Tenant *B*, on the other hand, is interrupt-bound for its smaller reads, consuming more request resources (4%) than out bandwidth (3.2%). Applying fair queuing to a single resource (bytes out) gives each tenant a fair share of 50%, but also allows

tenant B to receive a larger share (62.5%) of its dominant resource (requests). Instead, using *dominant resource fairness* (DRF) [9] ensures that each tenant will receive a fair share of its dominant resource relative to other tenants: tenant A receives 55% of out bytes and tenant B receives 55% of requests, while out bytes remains the bottleneck.

3. Pisces Algorithms

Pisces implements four complementary mechanisms according to the design lessons discussed above. Each mechanism operates on different parts of the system at different timescales. Together, they deliver system-wide fair service allocations with minimal interference to each tenant.

3.1 Partition Placement

The partition placement mechanism ensures the feasibility of the system-wide fair shares. It assigns partitions so that the load on each node (the aggregate of the tenants' per-partition fair-share demands) does not exceed the node's rate capacity. Since our prototype currently only implements a simple greedy placement scheme, here we only outline the algorithm without providing details. The centralized controller first collects the request rate for each tenant partition, as measured at each server. It then computes the partition demand proportions by normalizing the rates. Scaling each tenant's global fair share by these partition proportions determines the per-partition demand share that each tenant should receive. Next, the controller supplies the demand and node capacity constraints into a bin-packing solver to compute a new partition assignment relative to the existing one. Finally, the controller migrates any newly (re)assigned partitions and updates the mapping tables in its request router(s).

Partition placement typically runs on a long timescale (every few minutes or hours), since we assume that tenant demand distributions (proportions) are relatively stable, though demand intensity (load) can fluctuate more frequently. However, it can also be executed in response to large-scale demand shifts, severe fairness violations, or the addition or removal of tenants or service nodes. While the bin-packing problem is NP-hard, simple greedy heuristics may suffice in many settings, albeit not achieve as high a utilization. After all, to achieve fairness, we only need to find a feasible solution, not necessarily an optimal one. Further, many efficient approximation techniques can find near-optimal solutions [22, 29]. We intend to further explore partition placement in future work.

3.2 Weight Allocation

Once the tenant partitions are properly placed, weight allocation iteratively adjusts the local tenant shares (R) to match the demand distributions. As sketched in Algorithm 1, the weight-allocation algorithm on the controller (i) detects tenant demand-share mismatch and (ii) decides which tenant share(s) (weights) to adjust and by what amount, even while it (iii) maintains the global fair share. A key insight is that any adjustment to tenant shares requires a *reciprocal swap*: if tenant t takes some rate capacity from a tenant u on some server, t must give the same capacity back to u on a different server. This swap allows the tenants to maintain the same aggregate shares even while local shares change. Each tenant's local weight is initially set to its global weight, w_t, and then adapts over time. To adapt to distribution shifts yet allow replica selection to adjust to the current allocation, weight allocation runs in the medium timescale (seconds).

Detecting mismatch: While it is difficult to directly measure tenant demand, the central controller can monitor the rate each tenant receives at the service nodes. Weight allocation uses this information (D), together with the allocated shares (R), to approximate the demand-share mismatch that each tenant experiences on each node. We tried both a latency-based cost function using the M/M/1 queuing model of request latency—i.e., $l_n^t = 1 \,/\, (R_n^t - D_n^t)$ for tenant t on node n—as well as a more direct rate-based cost. Unfortunately, in a work-conserving system, it can be difficult to determine how much rate R was actually allocated to t, as it can vary depending on others' demand. For example, if a tenant t has weight $w_n^t = \frac{1}{4}$ on a node with capacity c_n, then its local rate under load is just $\hat{R}_n^t = \frac{1}{4}c_n$, even though $R_n^t \geq \frac{1}{4}c_n$ if t has excess demand and other tenants are not fully using their shares. Instead, by using the difference between the consumed rate and the configured local share, $e_n^t = \left| D_n^t - \hat{R}_n^t \right|$, we can largely ignore the variable allocation and instead focus on the tenant's desired rate under full load (i.e., $\hat{R}$). Fortunately, the allocation algorithm can easily accommodate any convex cost function to approximate demand mismatch.

Determining swap: Since the primary goal of Pisces is fairness, weight allocation seeks to minimize the *maximum* demand-share mismatch (or cost). However, giving additional rate capacity to the tenant t that suffers maximal latency necessarily means taking away capacity from another tenant u at the same node n. If too large, this rate (weight) *swap* may cause u's cost to exceed the original maximum. To ensure a valid rate swap, the algorithm uses the linear bisection for latency, $\text{take}(t, u, n) = ((R_n^u - D_n^u) - (R_n^t - D_n^t)) \,/\, 2$, or the min of the differences for rate: $\min(e_n^t, e_n^u)$.

Maintaining fair share: Before committing to a final swap, weight allocation must first find a reciprocal swap to maintain the global fair share: if tenant t takes from tenant u at node n, then it must reciprocate at a different node. Given a reciprocal node m, the controller computes the rate swap as the minimum of the take and give swaps, $\text{swap} = \min(\text{take}(t, u, n), \text{give}(u, t, m))$, and translates the rates into the corresponding local weight settings.

Algorithm 1 Weight Allocation: medium timescale (s)

T: tenants, N: nodes, W: global tenant weights,
R: local tenant resource share

function CONTROLLER.ALLOCATEWEIGHTS(T, N, W, R)
  $D \leftarrow$ *monitor_node_rates*(T, N)
  $C \leftarrow$ *compute_cost_estimates*(D, R)
  $R \leftarrow$ *compute_weight_swap*(max(C))
  reassign_weight_allocations(R)

Algorithm 2 Replica Selection: real-time (ms)

j: request/response, t: tenant, M: partition mappings
n: node, q^t: per-tenant request queue

function REQUESTROUTER.SENDREQUEST(j, t, M)
  $p \leftarrow$ *get_partition_of*(j)
  for $n \in M[t, p]$ **do**
    if window $w_n^t >$ outstanding s_n^t **then**
      send_request(j, n)
      $s_n^t \leftarrow s_n^t + 1$ **return**
  if not sent **then** *queue_request*(j, q^t)

function REQUESTROUTER.RECVRESPONSE(j, t, n)
  $l_{\text{resp},n} \leftarrow$ *latency_of_response*(j)
  $s_n^t \leftarrow s_n^t - 1$
  update_window($w_n^t, l_{\text{resp},n}$)
  SendRequest(*dequeue_request*(q^t), t, M)

Algorithm 3 Fair Queuing: real-time (μs)

j: request, t: tenant, r: round, R: local tenant share

function SERVICENODE.QUEUEREQUESTS(R)
  while $j, t \leftarrow$ *dequeue_request*() **do**
    if t.state = inactive **then**
      t.state $\leftarrow$ active
      allocate_tenant_tokens(R)
    if tokens available for t **then**
      consume_request_resources(j, t)
      if j unfinished **then**
        queue_request(j)
      else
        if resources left for j **then**
          refund_unused_resources(j,t)
        if no requests left for t **then**
          t.state $\leftarrow$ inactive
    else
      queue_exhausted_request(j, t)
      t.state $\leftarrow$ exhausted
  if $\exists\, t \in T$ such that t.state = exhausted **then**
    $r \leftarrow r + 1$ (increment round)
    allocate_tenant_tokens(R)

3.3 Replica Selection

To maintain fairness while balancing load, request routers distribute tenant requests to partition replicas in proportion to their local shares (i.e., normalized weights). While the controller (or alternatively, each server) could disseminate the local share information to each request router, Pisces avoids the need for explicit updates by exploiting implicit feedback. Explicit updates could be prohibitively expensive for a system with tens of thousands of tenants, request routers, and service nodes.

As delineated in Algorithm 2, when a request router (or client) sends a request, it round-robins between partition replicas (nodes), consuming slots from their respective request windows. Once the windows fill up, the request router locally queues requests until server responses free additional window slots. Due to per-tenant fair queuing at the nodes, requests sent to nodes with a larger tenant share experience lower queuing delay than nodes with a smaller share. The request router uses the relative response latency between replicas as a proxy for the size differential between the local rate allocations. It thus adjusts the request windows according to the FAST-TCP [34] update:

$$w(m+1)_n^t \;=\; (1-\alpha)\cdot w(m)_n^t \;+\; \alpha\cdot\left(\frac{l_{\text{base}}}{l_{\text{est}}}\right)$$

Each iteration of the algorithm adjusts the window based on how close the tenant demand is to its local rate allocation, which is represented by the ratio of a desired average request latency, l_{base}, to the smoothed (EWMA) latency estimate, l_{est}. The α parameter limits the window step size. Thus, each request router makes proper adjustments in a fully decentralized fashion: it only uses local request latency measurements to compute the replica proportions. The convergence and stability guarantees for this approach follow from those given by FAST-TCP.

Because replica selection balances a tenant's demand distribution in real-time, the per-tenant demand at each service node equilibrates before the next iteration of the weight-allocation algorithm. Weight allocation then attempts to match the local tenant shares (normalized weights) to the new demand distribution. The convergence and stability of this interlocking coordinate-ascent algorithm arises from the convex nature of the problem.

3.4 Fair Queuing

Ultimately, system-wide fairness and isolation comes down to mediating resource contention between tenants at the individual storage nodes. To implement fair queuing, Pisces uses the deficit (weighted) round robin [30] (DWRR) scheduling discipline for its simplicity, low time complexity, and bounded deviation from the ideal Generalized Processor Sharing model. In DWRR, the basic unit of work is called a token, which represents a normalized request or quantum of work. Pisces applies Dominant Resource Fairness, as mentioned in Section 2.2, to translate a token into a tenant-specific resource allocation vector. Currently, our implementation accounts for the number of bytes received, bytes sent, and requests (the latter for request-bound workloads). The queuing algorithm can also support additional resources like disk IOPs, which we intend to explore in the future. Multiple tokens may be needed to serve a large request, or a single token's resources may span several small requests. In any given

round of request processing, each tenant can consume up to its weighted share of the total fixed number of available tokens. By bounding the number of tokens per round, the scheduler can ensure fairness within a definite timeframe.

Request processing in DWRR proceeds in rounds. Per Algorithm 3, on each scheduling round, the scheduler allocates tokens to each active tenant (those with queued requests) in proportion to their local weight. As it processes requests, the scheduler consumes resources from the token resource allocation. If a request requires additional resources, the scheduler adds it back on the request queue to mitigate head-of-line blocking. Otherwise, it refunds the tenant with any unconsumed resources. If a tenant runs out of tokens, the scheduler adds its outstanding requests to an exhausted queue. The scheduler advances the round and refreshes tokens only when every tenant is either inactive (no work) or exhausted (work but no tokens) and there is work to do in the next round.

To compute the proper Dominant Resource Fair (DRF) shares, the scheduler on each node tracks the resource consumption of each tenant. Periodically (every half second in our prototype), it recomputes the resource allocation. First, the scheduler determines each tenant's resource utilization (e.g., $U^t_{\text{bytes-out}} = \frac{\text{bytes-out}^t}{\text{bytes-out}^{\text{cap}}}$) and its dominant resource ($U^t_{\text{dom}} = \max_i(U^t_i)$), using the latter to normalize each utilization. The scheduler computes the limiting DRF allocation by finding the minimum of the inverse of the weighted utilization sums: $\min_i(\frac{1}{\sum_i w^t U^t_i})$. Any excess resources are distributed equally among all tenants.

Despite the fact that Pisces only enforces DRF on a per-node level, it still provides global max-min fair shares for each tenant. When two tenants have different dominant resources at a node, DRF allocates each tenant a larger local share than it would have received if the tenants had contended for the same resource. In other words, each tenant's share of its dominant resource is lower bounded by the max-min fair share of a single common resource. Thus, even if a tenant's dominant resource varies from node to node, its aggregate share will still equal or exceed its max-min fair share (proportion) of total system resources. This allows Pisces to use a single weight to represent the per-tenant global and local shares, rather than a more complicated weight vector.

4. Pisces Optimizations

Pisces's algorithms and its four part decomposition can be conceptually derived as a distributed optimization problem [24]. In designing Pisces, we consider a multi-timescale decomposition of the fair-sharing problem into the corresponding Pisces mechanisms. Such a formulation allows us to craft the Pisces algorithms in a principled way and to make assertions about their feasibility, optimality, stability, and convergence under the standard

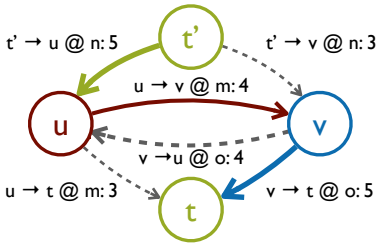


Figure 3: Reciprocal weight exchange modeled as a Maximum Bottleneck Flow problem.

assumptions of convex optimization. In the remainder of this section, we discuss additional design and implementation considerations in the Pisces weight allocation and fair queuing mechanisms.

4.1 Finding Multilateral Weight Swaps

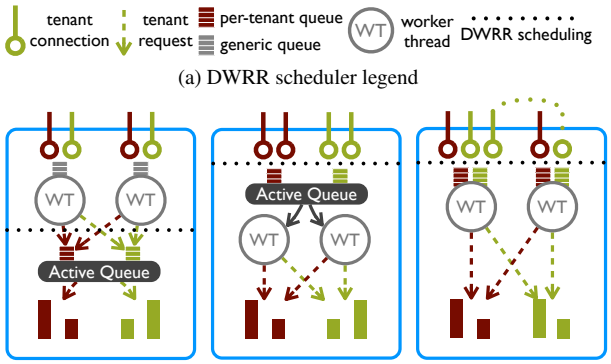
While a bilateral exchange between two tenants (as described in Section 3.2) may suffice, a multilateral exchange may optimize local shares even further. We model this exchange as a path through a flow graph, as shown in Figure 3. Nodes in the graph represent tenants and each directed edge represents a possible swap with capacity equal to the swap rate (e.g., tenant u can take rate 4 from tenant v at server m). The swap rate must be positive, and it is computed as the maximum over all server nodes where the edge's tenants have co-located partitions.

The max latency tenant t is modeled as both the source t' and sink of the flow graph, as it must first take rate to minimize its cost (latency or rate-distance) and then reciprocate to maintain the global fair-share invariant. In the example, both bilateral exchanges ($t' \rightarrow u \rightarrow t$ and $t' \rightarrow v \rightarrow t$) are bottlenecked by the edge with smallest capacity (i.e., 3). Instead, weight allocation should choose the multi-hop path ($t' \rightarrow u \rightarrow v \rightarrow t$) with bottleneck 4 that corresponds to the Maximum Bottleneck Flow (MBF). The MBF not only minimizes the max cost for t by the greatest extent, but also reduces the cost for u and v.

On each iteration of the weight allocation loop in Algorithm 1, the controller constructs the flow graph from the collected node latency data and solves the MBF problem using a variant of Dijkstra's shortest path algorithm. Then, just as with bilateral swaps, the algorithm converts the rates into weights and sets the local tenant shares accordingly. To avoid oscillations around the optimal point, both the tenant latencies and the swap rate must exceed minimal thresholds, in order to ensure that the weight adjustment results in the desired rate change.

4.2 Getting Fair Queuing Right

Enforce queuing at the appropriate software layer. Implementing the server DWRR scheduler may seem straightforward at first, but it presents an engineering challenge to do so with low overhead. The most natural approach is simply to place tenant request queues right after request processing in the application, as in Figure 4b.



(a) DWRR scheduler legend

(b) Per-request DWRR (c) Per-connection (d) Non-blocking

Figure 4: Scheduling per-connection (c) instead of per-request (b) achieves fairness. Decoupling threads and work-stealing (d) optimizes performance.

The problem, however, is that resources have already been consumed (to receive bytes and parse the request), which prevents the scheduler from enforcing fairness and isolation for certain network I/O bound workloads. Thus, the Pisces scheduler instead operates prior to application request handling, as in Figure 4c, and mediates between *connections* before any resources are consumed. Now, however, the scheduler no longer knows how much work remains in each connection queue. To handle this uncertainty, the DWRR scheduler allocates a fixed number of tokens from the tenant's token pool when a connection is dequeued for processing. As in Algorithm 3, any unused tokens and resources are *refunded* back to the tenant.

Avoid queue locking at all costs. Even with the scheduler in the right place, we still need to worry about the queue's implementation and efficiency. Maintaining a centralized active queue—per-connection DWRR in Figure 4c—is a natural design point for fair-queuing. A single thread enqueues connections, and separate worker threads pull tenant connections off this queue one-at-a-time, servicing them by consuming token resources. While simple and fair, this design is flawed: whenever the active queue is empty, the worker threads must wait on a conditional lock. We found that the overhead of this conditional waiting and waking can reduce the processing of small requests by over 30%. To combat this overhead, Pisces uses a combination of non-blocking per-tenant connection queues [21] and a distributed form of DWRR [18]. In this scheme, each worker thread handles its own set of tenant connections by sub-allocating tenant tokens for local use. If a worker thread's connections have either quiesced or run out of tokens, it tries to steal work or tokens from the other threads. If nothing can be stolen, then the worker thread can safely advance the round, since all tenants are either inactive or exhausted.

Eliminate intermediary queuing effects. While this non-blocking design (Figure 4d) is highly efficient, it faces one final barrier to max-min fairness: our measurements showed good performance for small requests bottlenecked by server interrupts, but poor fairness for bandwidth-bound workloads. This arises for two reasons. First, if the scheduler does not properly wait for write-blocked connections (EAGAIN) to finish consuming resources before advancing the round, then high-weight, bandwidth-bound tenants could see their remaining tokens wiped out prematurely. Second, over-sized TCP send buffers (128 KB) mask network back pressure. On a 1Gbps link with sub-ms delay, the bandwidth delay product is on the order of tens of kilobytes. When multiple tenant connections (>64) contend for output bandwidth, writes can succeed even when the outbound link is congested, which again causes the scheduler to advance the round too soon. In response, Pisces uses small connection send buffers (6 KB), and the scheduler waits for I/O-bound connections to finish consuming resources before advancing the round. To prevent worker threads from excessive idling due to non-local network congestion, the scheduler uses a short timeout (2 ms) that wakes all I/O quiescent threads and allows the round to advance.

5. Prototype on Membase

We implemented Pisces on top of the open-source Membase [3] key-value storage system (part of the Couchbase suite). Built around the popular memcached in-memory caching engine, Membase adds object persistence, data replication, and multi-tenancy. Membase relies heavily on the in-memory key-value cache to serve requests and dispatches disk-bound requests to a background thread. Key-value set or delete operations are committed first in memory and later asynchronously written to disk.

Membase creates even-sized explicit partitions and directly maps the partitions to server nodes. It can replicate and migrate partitions for fault tolerance, and synchronizes primary and secondary replicas in an eventually consistent manner. For evaluation purposes, we replaced Membase's uniform partition-placement mechanism with one based on a simple greedy heuristic, with which we pre-compute a feasible fair placement based on known (oracle) tenant demand distributions.

We integrated Pisces's fairness and isolation mechanisms into Membase using a mix of languages. Implementing the optimized multi-tenant, non-blocking DWRR scheduler in the core server codebase required an extensive overhaul of the connection threading model in addition to adding the scheduling and queuing code in C (3000 LOC). Replica selection was implemented in Java (1300 LOC) and integrated directly in the spymemcached [4] client library. Our centralized controller, which implemented both weight allocation and partition placement, comprised approximately 5000 LOC of Python.

6. Evaluation

In our evaluation, we consider how the mechanisms in Pisces build on each other to provide fairness and performance isolation by answering the following questions:

- Are each of the four mechanisms (PP, WA, FQ and RS) necessary to achieve fairness and isolation?
- Can Pisces provide global weighted shares and enforce local dominant resource fairness?
- How well does Pisces handle demand dynamism?

We quantify fairness as the Min-Max Ratio (MMR) of the dominant resource (typically throughput) across all tenants, $\frac{x^{\min}}{x^{\max}}$. This corresponds directly to a max-min notion of fairness.

6.1 Experimental Setup

To evaluate Pisces's fairness properties, we setup a testbed comprised of 8 clients, 8 servers, and 1 controller host. Each machine has two 2.4 GHz Intel E5620 quad-core CPUs with GigE interfaces, 12GB of memory, and run Ubuntu 11.04. Pisces server instances are configured with 8 threads and two replicas per partition. All machines are connected directly to a single 1 Gbps top-of-rack switch to provide full bisection bandwidth and avoid network contention effects. Similarly, replica selection and request routing are handled directly in a client-side library to minimize proxy bottleneck effects.

On the clients, we use the Yahoo Cloud Storage Benchmark (YCSB) [7] to generate a Zipf distributed key-value request workload (α = 0.99). Each client machine runs multiple YCSB instances, one for each tenant, to mimic a virtualized environment while avoiding the overhead of virtualized networking. Each tenant is pre-loaded with a fully cached data set of 100,000 objects which are hashed over its key space and divided into 1024 partitions. Object sizes are set to 1kB unless otherwise noted. Tenant request workloads include read-only (all GET), read-heavy (90% GET, 10% SET) and write-heavy (50% GET, 50% SET). All clients send their requests over TCP.

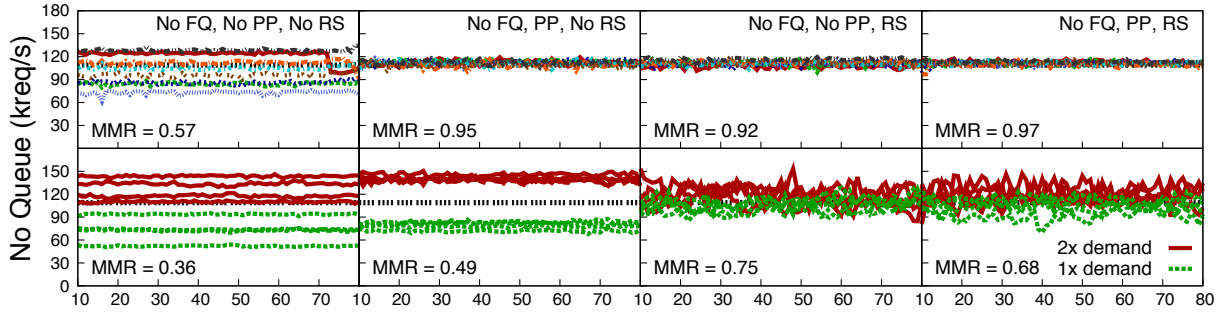
6.2 Achieving Fairness and Isolation

To understand how Pisces's mechanisms affect fairness and isolation, we evaluated each mechanism in turn and in combination, as shown in Figure 5. Starting with an unmodified base system (Membase) without fair queuing, we alternately add in partition placement (PP), which we pre-compute using our simple greedy heuristic, and replica selection (RS). We then repeat the combination with fair queuing (FQ) and static (uniform) local weights and complete the set of permutations with weight allocation (WA + FQ). In each experiment, 8 tenants with equal global weights attempt to access the 8-node system with the same demand. For illustrative purposes, we first present results for a simple GET workload, and summarize results for more complex workloads in Table 1. For the 1kB GET workload, the fair share (1.0 MMR) is bandwidth-limited at 109 kreq/s per node.

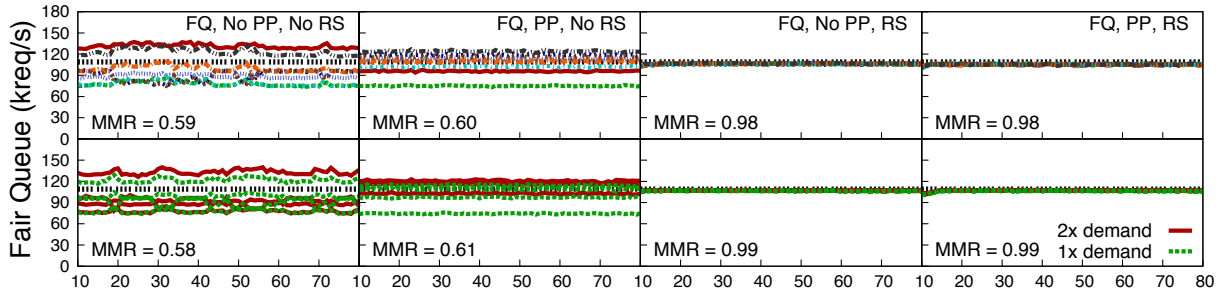
Unmodified Membase: The unmodified system provides poor throughput fairness (0.57 MMR) between tenants. This is largely due to the inherent skew in the tenant demand distribution which, per Section 2.2, can lead to an infeasible partition mapping. Figure 6a shows tenants 3 and 7 contending for hot partitions at server 2, while tenants 2, 4, and 6 collide on server 3 under the default, demand-oblivious placement. In contrast, Figure 6b shows how packing the partitions according to demand resolves the node capacity violations, which improves fairness. Once replica selection is enabled, though, the infeasibility issue largely disappears. By splitting request demand across replicas, RS smoothes out the hot spots. This relaxes partition bin-packing problem and allows the system to achieve high fairness both with and without PP ($\geq$ 0.92 MMR). While PP and RS help improve fairness, without request scheduling, Membase still fails to provide performance isolation. In the bottom half of Figure 5a, half the tenants double their demand, which degrades fairness across the board.

Multi-tenant Weighted FQ: Unsurprisingly, with weighted dominant resource fair queuing, fairness barely improves under the default placement due to over-contention for resources on the servers, as shown in Figure 5b. Even with PP, fairness fails to improve despite the feasible placement, since the tenants still access each server at different rates, as seen in Figure 6b. Although FQ enforces local (equal) weights, without aligning those weights to tenant demand, tenants with more weight than fair partition demand on a given node can still consume up to their limit and violate the global fair share. However, despite the need for weight tuning, fair queuing still proves essential for performance isolation. Where Membase falls flat under excess demand from the 2x tenants, fair queuing maintains fairness in all conditions. Unmatched weights may allow tenants to consume more than their fair share, but the tenants cannot consume more than their local allocation, which bounds the deviation from global fairness. Again, by smoothing out hot spots, replica selection resolves the demand imbalance and improves fairness with or without PP ($\sim$ 99 MMR).

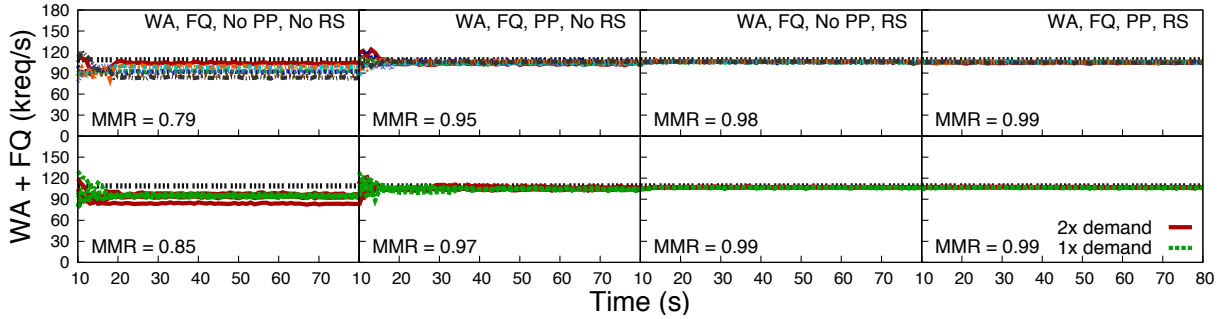
Weight Allocation and FQ: Enabling weight allocation unlocks Pisces's full potential, especially when replica selection is disabled (e.g. for consistency). By adapting to local tenant demands, weight allocation optimizes the system for fairness even when the placement is infeasible (0.79 MMR), as seen in Figure 5c. However, since the tenant shares are limited by the over-loaded nodes, overall throughput diminishes. Under a feasible placement, weight allocation is able to find the optimal



(a) Membase (no queuing): Unfair unless partition placement or replica selection are enabled and provides no isolation.



(b) Fair Queuing (equal local weights): Provides strong isolation, but struggles with fairness unless replica selection is enabled.



(c) Weight Allocation + Fair Queuing: Achieves the fairness under all conditions, even under infeasible partition mappings (no PP).

Figure 5: System-wide fairness and isolation under a combination of Pisces mechanisms.

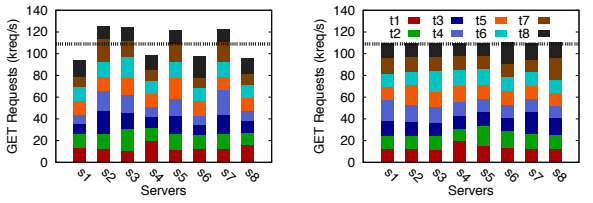
weights for the tenant demand on each server, as depicted in Figure 6c, while preserving global fairness (0.95 MMR). Since replica selection balances demand, weight allocation has little additional affect on fairness, but remains necessary to adapt to demand fluctuations. Excess demand can actually heighten the awareness of demand mismatch without PP, though, again, global throughput suffers. For all other cases, weight allocation tunes fair queuing to mediate between the tenants, which ensures high fairness ($\geq$ 0.97 MMR) and performance isolation.

In addition to throughput fairness and isolation, Pisces also provides a measure of latency isolation as well. The first two groups in Figure 7 show the average median latencies for the 1x and 2x demand tenants in the previous experiments. The max error bar indicates the 95th percentile, while the min error bar indicates the spread between the tenants' median latencies. Without fair queuing, there is little isolation with median latencies for both 1x and 2x tenants hovering around 4 ms. Adding replica selection improves isolation where the 2x tenants experience about 1.7 times the median latency as the 1x tenants. However, the latency spread (3.2 ms) spans the entire gap between them. With fair queuing, the median latencies are far more well-behaved. The 2x tenants receive a 2.1 times latency penalty with minimal ($<$1 ms) spread.

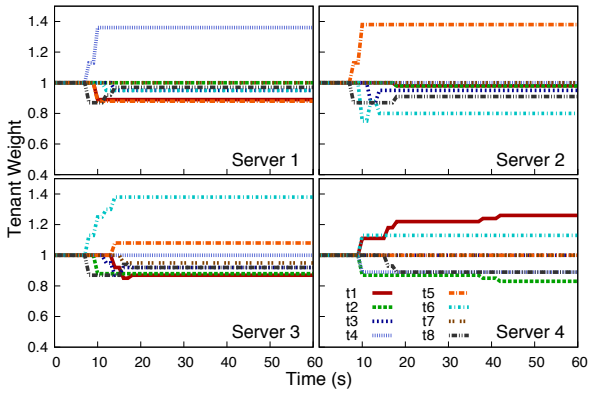
For a more thorough evaluation of Pisces fairness, we experimented over a range of log-normal distributed object sizes, where each tenant has object sizes drawn from a distribution with the same mean value (1kB or 10B) and standard deviation (0, 0.5, or 1). We also varied the workloads between all read-only, all read-heavy, or a mix of read-only, read-heavy, and write-heavy. Table 1 summarizes these results for Pisces with all mechanisms enabled. We measured mean bandwidth consumption for 1kB objects and mean request rates for 10B objects, as well as median request latency (1ms averaged) and average fairness (in terms of bandwidth consumed or request rates, respectively). For mixed workloads, values for both

	Fixed Size Objects				LogNormal 0.5				LogNormal 1.0			
1kB Mean	**BW (Gbps)** Out/In	**Lat (ms)** GET/SET	**MMR** Out/In	**MMR ratio** Out/In	**BW (Gbps)** Out/In	**Lat (ms)** GET/SET	**MMR** Out/In	**MMR ratio** Out/In	**BW (Gbps)** Out/In	**Lat (ms)** GET/SET	**MMR** Out/In	**MMR ratio** Out/In
Read-Only	6.95	2.48	0.99	1.73	6.93	3.67	0.99	1.57	6.94	5.20	0.99	2.15
Read-Heavy	6.87/1.30	2.99/2.15	0.98/0.98	1.56/1.58	6.82/1.32	3.34/2.62	0.99/0.98	1.32/1.29	6.90/1.35	4.70/3.45	0.99/0.99	1.65/1.57
Mixed	6.81/2.06	2.55/2.41	0.68/0.77	1.62/1.05	6.81/2.08	3.11/2.75	0.67/0.77	1.81/0.92	6.76/2.01	4.01/3.71	0.63/0.74	1.85/1.19
10B Mean	**Tput (kreq/s)** GET/SET	**Lat (ms)** GET/SET	**MMR** Requests	**MMR ratio** Requests	**Tput (kreq/s)** GET/SET	**Lat (ms)** GET/SET	**MMR** Requests	**MMR ratio** Requests	**Tput (kreq/s)** GET/SET	**Lat (ms)** GET/SET	**MMR** Requests	**MMR ratio** Requests
Read-Only	3,160	3.37	0.97	1.33	3,151	3.52	0.97	1.26	3,104	3.36	0.96	1.35
Read-Heavy	2,567/285	3.04/8.19	0.98	1.28	2,593/288	2.96/8.30	0.98	1.42	2,540/282	3.00/8.49	0.94	1.34
Mixed	2,343/445	2.80/7.79	0.96	1.5	2,325/444	2.88/7.84	0.96	1.32	2,295/437	2.94/7.78	0.94	1.38

Table 1: Pisces performance over a range of log-normally distributed object sizes and GET/SET workloads



(a) Uniform partition placement (b) Demand-aware placement



(c) Local tenant weights evolving with weight allocation

Figure 6: Skewed demand can lead to infeasible partition mappings (a). Placing partitions according to tenant demand and node capacities ensures feasibility (b). Weight allocation, in turn, adjusts the per-node tenant weights to the demand (c) (4 of 8 shown).

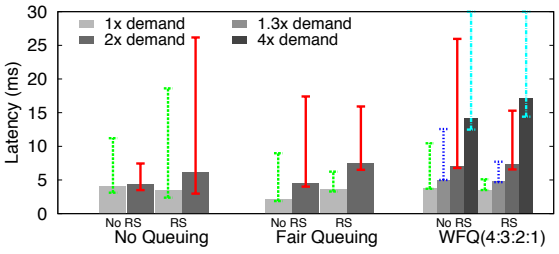


Figure 7: Fair queuing protects request latencies for even and weighted tenant shares.

GET/SET (Out/In bandwidth) requests are shown.We also include a fairness comparison in terms of the MMR ratio between Pisces and unmodified Membase.

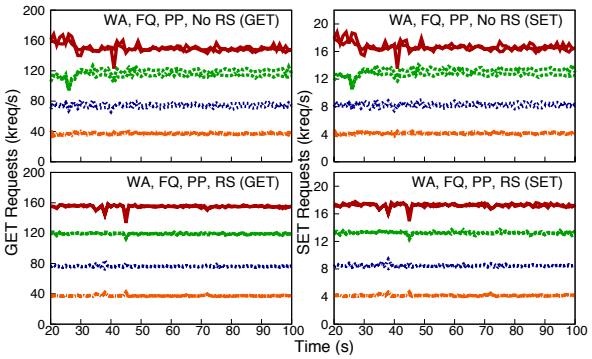
Pisces is able to achieve over 0.94 MMR fairness for most size variations and workloads. Moreover, Pisces exceeds the fairness of the unmodified base system by more than 1.3 times for most cases as well. The mixed workload for 1kB objects, however, proved to be a particularly troublesome combination. While the read-heavy and write-heavy tenants received their appropriate shares of inbound write bandwidth (0.99 MMR), the read-only tenants were able to consume more outbound read bandwidth than either one, resulting in an overall fairness between 0.63 and 0.64 MMR. One reason for this is the head-of-line blocking of a tenant's read requests due to its write requests fully consuming its current inbound bandwidth allocation. This can push subsequent read requests to the next scheduler round, despite having unconsumed outbound bandwidth remaining. Since read-only tenants never bottleneck on inbound bandwidth, they can fully consume their share (and more) of outbound bandwidth.

To fix this issue, we modified the clients to prioritize read requests over write requests when sending requests to the servers. While this largely resolved the issue for read-heavy tenants (> 0.9 MMR), the write-heavy tenants remained obstructed by the bandwidth bottleneck. The low MMR ratio for writes (bytes in) compared to Membase is also attributable to this performance write-bottlenecking variance. For 10B workloads, the problem disappears (≥ 0.95 MMR) since both read and write workloads share a single bottleneck: request-rate without suffering packet-level congestion effects. Pisces is able to once again claim its fairness crown from Membase for these cases (> 1.28x more fair than Membase).

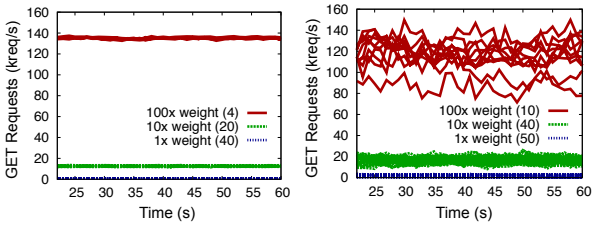
6.3 Service Differentiation

So far, we have demonstrated that Pisces's mechanisms, working in concert, can achieve both isolation (FQ) and nearly ideal even fair sharing (PP + WA or RS). We now turn our attention to providing *weighted* fairness at the global level (for service differentiation) and at the local level (for dominant resource fairness).

Global Differentiation: In Figure 8a, the tenant are assigned global weights in decreasing order 4:4:3:3:2:2:1:1 to differentiate their aggregate share of the system resources. Both with and without replica selection, Pisces is able to achieve high global weighted fairness (> 0.9 MMR) for both in and out bandwidth under a 1kB request read-heavy workload. Similarly, request latency remains strongly isolated between the tenants. Since all tenants



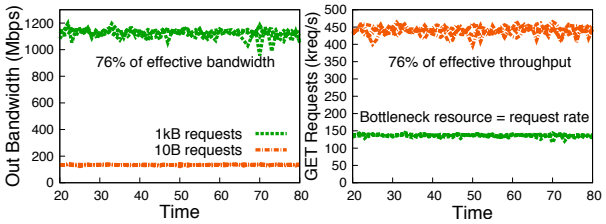
(a) Global weighted 90% GET / 10% SET throughput



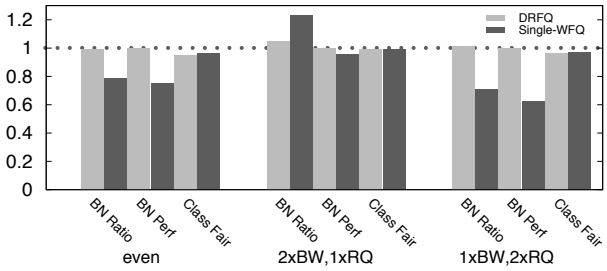
(b) 64 tenant weighted thruput

(c) 100 tenant weighted thruput

Figure 8: Subfigure (a) demonstrates global 4:3:2:1 weighted fair sharing for a read-heavy workload. In (b) and (c), Pisces abides by the skewed tenant weights on an 8 and 20 node cluster respectively.



(a) Dominant resource fairness between 1kB and 10B workloads



(b) Bottleneck ratio fairness and performance with and without multi-resource queuing

Figure 9: Pisces provides dominant resource fairness between bandwidth (10kb) and request-rate (10B) bound tenants (a). Compared to single-resource fair queuing, DRF provides a better share of the bottleneck resource (BN Ratio) and better performance (BN Perf) over different tenant weights.

generate the same demand, the lower weight tenants (3, 2, and 1) exceed their share by 1.3, 2, and 4 times respectively. Per Figure 7, the latencies of these tenants closely mirror their demand ratios. In both cases, the median (or mean) latency spread is small (<1 ms), except for that of the smallest-weight tenant (<2.7 ms).

To further stress the fairness properties of the system, we ran two larger scale experiments where the number of tenants far exceeded the number of servers, to mimic more realistic service scenarios. In Figure 8b, each of 64 tenants reside on 4 out of the 8 available servers with each server housing 32 tenants. To reflect the highly skewed nature of tenant shares, i.e. a few heavy hitters and many small, low-demand users, we configured the tenant weights along a log-scale: 4 tenants with weight 100, 20 with weight 10, and 40 with weight 1. Within each weight class, Pisces achieves > 0.91 MMR. Between classes, however, weighted fairness decreases to 0.56 MMR. This deviation is due to the limits of the DWRR scheduler token granularity. With such highly skewed weights, tenants in the smallest weight class (1) only receive fractional tokens, which means their requests are processed once every few rounds, which results in a lower relative share. Fairness between the weight 100 and weight 10 tenants, however, remains high (0.91 MMR). In practice, to work around the limited resolution of the WFQ scheduler, the service provider can cap the number of tenants per server to ensure reasonable local weights (token) and match the desired rate guarantees. Figure 8c shows a larger scaled out experiment, with 100 tenants resident on 6 of 20 servers (30 tenants per server). While we see a qualitatively similar result, the actual fairness degrades considerably (average 0.68 MMR between high and medium weight tenants, 0.46 MMR across all classes). However, this is mostly due to performance variance on the scale-out testbed [25] arising from CPU scheduling and network bottleneck effects, which affects the high-weight tenants disproportionately. As a result, the low-weight tenants can consume a larger share.

Local Dominant Resource Fairness: While Pisces provides weighted fairness on the global level, we want to ensure that each tenant receives a weighted share of its dominant resource on the local level. To stress different dominant resources and their corresponding bottlenecks, we experimented with four tenants requesting 1kB objects (bandwidth limited), while another four operated on 10B objects (request-rate limited). Under even weighting, Figure 9a shows how Pisces enforces fairness within each dominant resource type (>0.95 MMR) and evenly splits the dominant resource shares between types: the 1kB tenants receive ~76% of the effective outbound bandwidth and the 10B tenants receive ~76% of the effective request rate.[3] Although the tenants differ in dominant resource, they share the same bottleneck resource (request rate in this case). By computing the bottleneck resource ratio (BNR) between the different dominant resource tenants,

[3]This is lower than the optimal rates since transmitting a 1kB request takes longer than processing a 10B request.

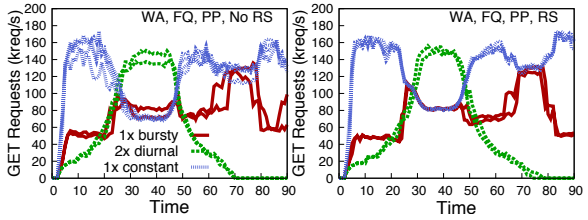
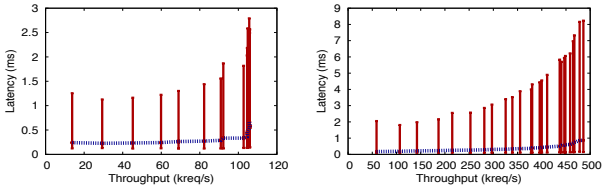


Figure 10: Pisces responds to demand dynamism according to tenant weights.



(a) Pisces μ-benchmark (1kB) (b) Pisces μ-benchmark (10B)

Figure 11: Median throughput versus latency microbenchmark with 99th-percentile error bars.

we can directly determine the dominant resource shares as shown in section 2.2 without having to estimate the effective resource rates. In this instance, the BNR is around 3.2, which gives the 10B tenants 76% of the request rate, and allows the 1kB tenants to consume 76% of the bandwidth.

Using BNR, we compare the dominant resource fairness of the DRF-enabled scheduler versus the single-resource (request) version for even weighted tenants, 2x weighted 1kB (bw) vs. 1x weighted 10B (rq) tenants, and 1x weighted 1kB vs. 2x weighted 10B tenants. For the even and 1x bandwidth, 2x request cases, the bottleneck resource is request rate. In the 2x bandwidth, 1x request scenario, the tenants bottleneck on bandwidth. As Figure 9b shows, the DRF scheduler outperforms the single resource version in all cases, achieving the best normalized BNR value and bottleneck resource performance (BN Perf). The single-resource scheduler holds a slight edge in fairness between tenants of the same dominant resource class (Class Fair) in the even and 1x bandwidth, 2x request cases. As in the mixed workload experiments, achieving the highest 10B request throughput required additional packet prioritization. We enabled the linux priority queue scheduling discipline to reduce the scheduler delay for small requests while waiting for connection send buffers to clear for the 1kB tenants.

6.4 Dynamic Workloads

Dynamic workloads present a challenge for any system to provide consistent, predictable performance. In Figure 10, 2 bursty demand tenants (weight 1), 2 diurnal demand tenants (weight 2), and 4 constant demand tenants (weight 1) access the storage system. Initially, the tenants consume less than the full system capacity which allows the constant tenants to consume a larger proportion. As the diurnal tenants ramp up between 0 and 20 seconds, they begin to consume their share of throughput which cuts into the excess share consumed by the constant tenants. Around 20s, both the bursty tenants and diurnal tenants ramp up to their full load, which results in a nearly 2 to 1 ratio, according to the tenant weights. The diurnal tenants tail off around 50s along with the bursty tenants which allows the constant demand tenants to, once again, consume in excess of their fair share (~ 80 kreq/s). Lastly, at 70s, the bursty tenants issue one last barrage of requests, which forces the constant tenants to share the throughput equally. Both with and without replica selection enabled, Pisces is able to handle the demand fluctuations and provide fair access to the storage resources.

6.5 Efficiency and Overhead

To assess the efficiency and overhead, we ran a microbenchmark of a single WFQ Pisces node. In this experiment, tenants issue requests at increasing rates to a single service node. As shown in Figure 11, Pisces is able to achieve over 106 kreq/s for 1kB requests, which is > 96% of Membase throughput, with the same average request latency (0.14 ms). For 10B requests, Pisces actually outperforms Membase by 20% (485 vs. 405 kreq/s) with lower average request latency (0.15 vs. 0.16 ms) due to the DWRR scheduler's work stealing mechanism.

7. Related Work

Sharing the network. Most work on sharing datacenter networks has used either static allocation or VM-level fairness. The static allocations by SecondNet [13] and Oktopus [6] guarantee bandwidth but can leave the network underutilized. While more throughput efficient, Gatekeeper's ingress and egress scheduling [27], Seawall's congestion-controlled VM tunneling [28], and FairCloud's per-endpoint sharing [26], respectively provide fairness on a per-VM, VM-pair, or communicating-VM-group basis, rather than on a per-tenant basis.

NetShare [17] and DaVinci [14] take a per-tenant network-wide perspective, but the former allocates local per-link weights statically, while the latter requires non-work-conserving link queues and does not consider fairness. In contrast, Pisces achieves per-tenant fairness by leveraging replica selection and adapting local weights according to demand, while maintaining high utilization.

Sharing services. Recent work on cloud service resource sharing has focused mainly on single-tenant scenarios. Parda [11] applies FAST-TCP congestion control to provide per-VM fair access, which Pisces uses as well, but for replicated service nodes. mClock [12] adds limits and reservations to the hypervisor's fair I/O scheduler to differentiate between local VMs, while Pisces's DWRR scheduler operates on a per-tenant level. Argon [32] uses cache management and time-sliced disk scheduling for performance insulation on a single shared file server. Pisces

could adapt these techniques for memory and disk I/O resources. Stout [20] exploits batch processing to minimize request latency, but does not address fairness.

Several other systems focused on course-grained allocation. Autocontrol [23] and Mesos [15] allocate per-node CPU and memory to schedule batch jobs and VMs, using utility-driven optimization and dominant resource fairness, respectively. They operate on a coarse per-task or per-VM level, however, rather than on per-application requests. In [10], the authors apply DRF to fine-grained multi-resource allocation, specifically to enforce per-flow fairness in middleboxes. However, their DRF queuing algorithm relies on virtual time, and it scans each per-flow packet queue for the lowest virtual start time.

8. Conclusion

This paper seeks to provide system-wide *per-tenant* weighted fair sharing and performance isolation in multi-tenant, key-value cloud storage services. By decomposing this problem into a novel combination of four mechanisms—partition placement, weight allocation, replica selection, and fair queuing—our Pisces system can fairly share the aggregate system throughput, even when tenants contend for shared resources and demand distributions vary across partitions and over time. Our prototype implementation achieves near ideal fairness (0.99 Min-Max Ratio) and strong performance isolation.

Acknowledgments. We thank Jennifer Rexford for helpful discussions early in this project. Funding was provided through NSF CAREER Award #0953197.

References

[1] `http://aws.amazon.com/dynamodb/faqs/`, 2012.

[2] `http://docs.amazonwebservices.com/amazondynamodb/latest/developerguide/Limits.html`, 2012.

[3] `http://www.couchbase.org/`, 2012.

[4] `http://code.google.com/p/spymemcached/`, 2012.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *SIGCOMM*, 2008.

[6] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[8] S. L. Garfinkel. An evaluation of Amazon's grid computing services: EC2, S3 and SQS. Technical Report TR-08-07, Harvard Univ., 2007.

[9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[10] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource scheduling for middleboxes. In *SIGCOMM*, 2012.

[11] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, 2009.

[12] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor IO scheduling. In *OSDI*, 2010.

[13] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *CoNext*, 2010.

[14] J. He, R. Zhang-Shen, Y. Li, C.-Y. Lee, J. Rexford, and M. Chiang. Davinci: dynamically adaptive virtual networks for a customized internet. In *CoNext*, 2008.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[16] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *CCGrid*, 2011.

[17] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese. Netshare: Virtualizing data center networks across services. Technical Report CS2010-0957, UCSD, 2010.

[18] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, 2009.

[19] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[20] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: an adaptive interface to scalable cloud storage. In *USENIX Annual*, 2010.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[22] R. M. Nauss. Solving the generalized assignment problem: An optimizing and heuristic approach. *INFORMS J. Computing*, 15 (Summer):249–266, 2003.

[23] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *EuroSys*, 2009.

[24] D. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *JSAC*, 24(8):1439–1451, 2006.

[25] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton CS, 2011.

[26] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[27] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV*, 2011.

[28] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *NSDI*, 2011.

[29] D. B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Math. Prog.*, 62(1):461–474, 1993.

[30] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *Trans. Networking*, 4(3):375–385, 1996.

[31] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bulletin*, 9(1):4–9, 1986.

[32] M. Wachs, M. Abd-el-malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[33] J. Wang, P. Varman, and C. Xie. Optimizing storage performance in public cloud platforms. *J. Zhejiang Univ. – Science C*, 11(12): 951–964, 2011.

[34] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. Fast TCP: Motivation, architecture, algorithms, performance. *Trans. Networking*, 14(6): 1246–1259, 2006.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.