Filo: consolidated consensus as a cloud service

Parisa Jalili Marandi Christos Gkantsidis Flavio Junqueira Dushyanth Narayanan
Microsoft Research, Cambridge, UK

Abstract

Consensus is at the core of many production-grade distributed systems. Given the prevalence of these systems, it is important to offer consensus as a cloud service. To match the multi-tenant requirements of the cloud, consensus as a service must provide performance guarantees, and prevent aggressive tenants from disrupting the others. Fulfilling this goal is not trivial without over-provisioning and under-utilizing resources.

We present Filo, the first system to provide consensus as a multi-tenant cloud service with throughput guarantees and efficient utilization of cloud resources. Tenants request an SLA by specifying their target throughput and degree of fault-tolerance. Filo then efficiently consolidates tenants on a shared set of servers using a novel *placement algorithm* that respects constraints imposed by the consensus problem. To respond to the load variations at runtime, Filo proposes a novel *distributed controller* that piggybacks on the consensus protocol to coordinate resource allocations across the servers and distribute the unused capacity fairly. Using a real testbed and simulations, we show that our placement algorithm is efficient at consolidating tenants, and while obtaining comparable efficiency and fairness, our distributed controller is $\sim 5x$ faster than the centralized baseline approach.

1 Introduction

Distributed consensus is a problem where several processes must agree on a common value [11]. Consensus is used by many systems to elect leaders [43, 38, 31], access shared objects [9], and order transactions in replicated services [44, 30, 8]. Consensus protocols are complex and difficult to implement correctly. Because of this difficulty, systems such as ISIS [7], Chubby [9], Apache ZooKeeper [23], and OpenReplica [46] offer consensus as a black box service.

Why consensus as a cloud service. Consensus is at the heart of many distributed applications that are or will become cloud residents. Hence, offering consensus as a *simple-to-use cloud service* will relieve people from the hurdles of developing and maintaining it. Similarly to systems such as key-value stores that are ubiquitously available as cloud services [13, 10, 3], we envision a future where consensus is available as a cloud offering.

Why consolidated consensus. The key to a cloud provider's success is reducing server footprint; this means reduced costs for the provider and also for the users. *Sharing* computing, storage, and network resources is essential for decreasing the number of servers. Our goal then is to consolidate consensus instances of many tenants (users) and deploy them on shared servers. Sharing is advantageous: reserved but under-utilized resources can be (temporarily) re-allocated to other tenants who demand higher performance at runtime, and prices can be accordingly adjusted. Re-allocations are impossible if tenants are promised dedicated resources. When sharing, it is essential to dynamically adjust allocations at real time to minimize the durations for which resources remain idle, and to use techniques that prevent aggressive tenants from monopolizing the resources.

Filo. Thus, to successfully offer consensus as a cloud service we should (a) design a simple API, (b) efficiently and fairly utilize shared servers, and (c) ensure performance isolation among tenants. We propose Filo, a multi-tenant consensus service that provides throughput SLA guarantees and meets these requirements. Filo's API is intuitive: users specify their SLAs in application-level request rates, and choose from a range of reliability options by specifying the number of replicas and the durability mode of their data (*i.e.*, storing data in memory, or on stable storage), and thus trade performance or price for reliability.

Filo co-locates replicas of various tenants on shared servers (consolidation). To consolidate, we have designed a novel placement algorithm that packs replicas, while respecting SLAs and the constraints imposed by the properties of the consensus protocols (*e.g.*, replicas of a tenant are placed on distinct servers to resist failure). Filo uses an empirical approach similar to [26, 1] to translate the high level SLAs specified by the users to low level resource costs (*i.e.*, CPU, network bandwidth, storage IO). Our placement algorithm uses these values to select servers that will host the replicas.

Consolidating replicas on shared servers raises an immediate concern with performance isolation. Filo guarantees performance isolation by rate limiting requests using multi-resource token buckets [1]. Filo dynamically calculates and tunes the budgets of these token buckets.

Anecdotal evidence suggests that tenants may misestimate their SLA requirements. To account for these inaccuracies, Filo monitors resource utilization at runtime and fairly re-distributes free resources among the tenants using a distributed controller. Centralized controllers are popular due to their accuracy at efficient resource allocations. However, the overhead of these techniques is also well-understood: all the servers in a cluster must transfer their status to the centralized controller regularly, and computations must be performed over a rather large set of data. Our contribution here is the design of a *distributed controller* that eliminates this overhead and yet results in highly efficient resource allocations. In a system such as Filo, where a tenant's replicas are placed on a rather small subset of servers (*i.e.*, 3, 5), a distributed controller has a high opportunity for calculating allocations that compete closely with the centralized techniques. To coordinate resource allocations across the servers, our distributed controller leverages the underlying consensus connections to transfer its messages.

Contributions. To summarize, in addition to building consolidated consensus as a cloud service, this paper makes the following contributions: (i) we propose an API that simplifies user's interactions with the system and abstracts away low level resource specifications that are needed for efficient replica placement, (ii) we design a novel placement algorithm to efficiently pack replicas on a shared set of servers while respecting the constraints imposed by consensus protocols, and (iii) we design a new *distributed controller* that dynamically, fairly, and efficiently adjusts resource allocations at runtime.

Results. We have implemented consolidated consensus and evaluated its performance in various workloads. These evaluations are essential for bounding feasible SLAs and translating SLAs to resource costs. We have thoroughly evaluated our placement algorithm and the distributed controller. Our placement algorithm efficiently places replicas on the servers while allocating above 98% of the resources. And our distributed controller converges to allocations with 95% efficiency of the centralized techniques while being $\sim 5x$ faster.

2 Consensus in Filo

Consensus enables a set of processes (*replicas*) to reach agreement on some value [11]. Consecutive execution of consensus produces a *log* of *ordered* entries. Each entry indicates the request that must be executed next in the sequence by the application. Filo provides consensus as a service by assigning to each tenant a set of replicas. We refer to each such set that executes consensus and orders the requests as a *consensus group*, and to the size of this group as *replication degree*. To be fault tolerant, replicas of a tenant must be placed on distinct servers. We differentiate between a *server* and a *replica*. A server is a physical machine and can host multiple replicas.

Separating ordering from execution. Tenants use Filo only for the *ordering and durability* guarantees. It is left to the tenants to *execute* their ordered requests, as Filo is agnostic to user-application semantics and treats requests as arbitrary bytes. Separating ordering from the execution has been the subject of previous studies [51, 8, 28]. This design choice is of crucial importance to Filo as it targets a diverse collection of cloud residents each with their own specific applications. Thus, we differentiate between the servers on which tenant applications are running (*execution*) and the servers on which Filo is running (*ordering*). We assign a *dedicated cluster of servers* to Filo which is not used by other applications. [1] Thus since tenant applications are outside the boundaries of Filo, they are free to use their desired caching or replication mechanisms, while using Filo only for the ordering and durability of their requests.

Consolidated consensus using Chain Replication. Filo implements consensus using Chain Replication [42]. Each tenant is assigned a chain composed of n replicas to which we will refer as *tenant replicas*. These are deployed on Filo's server cluster. Chain Replication organizes replicas on a chain and distinguishes between *head* and *tail* replicas. Clients send their write and read requests to the head and tail replicas respectively; thus both of these replicas communicate with the clients. To simplify the management of the client connections, and without affecting correctness, here we replace chain with a ring. The tail sends an ack to the head replica when a write request is finalized. Read and write requests are all received by the head. We refer to the replicas other than the head as *followers*. Note that we do not dedicate one server to each tenant replica, but rather servers are shared among the replicas belonging to various chains. We refer to this as the *consolidated* deployment of the chains.

Why Chain Replication. Majority-based protocols such as Paxos [30] need $2f+1$ replicas to tolerate f fail-

[1] Dedicating a cluster to a critical service such as Filo is important to protect its performance.

ures. Higher number of replicas enhances liveness but not performance. Given the importance of resource efficiency, we prefer protocols that achieve similar performance with fewer replicas. Chain Replication is one such protocol that with $f+1$ replicas tolerates f failures. But unlike Paxos, to make progress it requires *all* the replicas to be non-faulty. Because of its efficiency we have chosen Chain Replication to implement consensus. Despite this the choice of a consensus protocol is orthogonal to the focus of this paper and the techniques proposed in this paper are generalizable to others.

Failures. When replicas in a chain fail, the chain must be reconfigured before it can resume its operation with the initial degree of resiliency. As in [42], here failures are detected and resolved by an external replicated *master*. In our consolidated model there is only one master that is shared by all the chain instances. This is similar to Vertical Paxos [32] where many groups exists in the system and a master handles reconfigurations for all of them (see § 4.1 for more details).

Assumptions. We assume a crash-recovery failure model and exclude byzantine failures. Network is asynchronous with a possibility of message loss and arbitrary delays. We deploy consensus groups in *in-mem*, or *persistent* durability modes. In the latter, *logs* are made durable on a storage device. The system needs $f+1$ replicas to tolerate f failures.

Logger. To manage the *logs*, we have designed a subsystem similar to Bookkeeper [2] called *Logger*. *Logger* implements *group commit* with a buffer for receiving requests. As the number of concurrent requests increases, buffers transmit more data to the storage device with every fluch, and, hence, efficiency increases (without being subject to timeouts). Compared with *random* writes, *Sequential* writes benefit the most from group commit; thus we have one sequential log per server to which requests from all tenants are appended. *Logger* constantly and in the background prepares per-replica logs on a second storage device. Our servers must be provisioned by at least two storage devices for better performance.

3 System Design

Filo's main components are: (a) a simple API that allows tenants to specify their performance and reliability requirements, and (b) rate limiters and the resource allocator that manage server resources.

API. A new applicant starts by submitting an *admission request* to the *Admission Controller* (*AC*), a central component of the system. An admission request contains the following attributes:

$$\begin{aligned}(&\textit{durability mode}, \textit{replication degree},\\ &\textit{request size}, \textit{throughput SLA})\end{aligned} \tag{1}$$

Durability mode is either *in-mem* or *persistent*; replication degree is the number of requested servers; the throughput SLA is the application-level request rate for requests of a fixed size (request size). (For brevity, we omit attributes related to user account and other authentication information.) If the admission request can be satisfied by Filo the applicant is admitted to the system, after which it becomes a tenant. The tenant is given a handle to its consensus group, which will be used for submitting requests. Filo guarantees rates up to the throughput SLA, and can also accommodate higher rates if there is available capacity (see § 4.2 for more details).[2] The tenants submit read or write requests. Writes trigger consensus and are appended to the *log* (see § 2). Reads retrieve previously ordered data from the log. The throughput SLA agreed above is the sum of read and write requests. Writes are more expensive than reads as they are propagated to other replicas. Hence we concentrate on the write requests.

Rate limiters. To enforce throughput SLAs, Filo installs rate limiters in the form of multi-resource token buckets [49, 1] on the servers that host head replicas of the consensus groups and also on the external servers on which tenant applications execute. The token budget for each tenant is determined based on the request size specified in its *admission request* (1). Note that tenants are not prohibited from varying their request size at runtime. SLAs apply only to the initial request sizes, however. To prevent this variation from affecting other tenants, a request passes through the system if token buckets have sufficient tokens; requests of different sizes translate to different amount of tokens. Moreover, Filo constantly monitors resource usage at runtime (see below) and adjusts the budgets dynamically.

Resource allocator. The most important element of Filo is the resource allocator which is composed of two entities: an admission controller that executes the admission phase, and a distributed controller that executes the work conservation phase. During the admission phase, admission controller runs a placement algorithm to efficiently place tenant replicas on the servers. Placement is done based on the translation of the admission requests to resource usages (CPU, network bandwidth, storage IO), which is done using an empirical strategy; prior to

[2] Before launching Filo, we extensively evaluate its performance under various workloads to find its peak performance. Peak performance caps the range of SLAs that Filo can promise to its users. For example if we can order 10 reqs/sec at best, we can not admit an applicant that asks for 11 reqs/sec.

launching the service, we extensively benchmark the system to build a performance profile. During the work conservation phase, the distributed controller monitors resource utilization and adjusts the allocations to absorb the exceeding demands of its tenants.

4 Resource Allocator

Filo's objective is to offer consensus as a service while providing SLA guarantees, and fairly and efficiently allocating CPU, network, and storage resources. To accomplish these goals, it is essential to manage replica placement when admitting tenants, to dynamically adjust resource allocations at runtime, and to control the rates at which tenants submit their requests. *We have designed resource allocator to handle these tasks*. We next define the concepts and constraints of our problem, and then present our algorithms for implementing the *resource allocator* via the admission and work conservation phases.

Definitions. Let S be a set of m servers, each with k resources. Let $(r_1,...,r_k)$ be a resource vector. Each server has two vectors R^{no} and R for the nominal and free amount of its resources ($r \in \mathbb{R}_{\geq 0}$, and R^{no} is constant). Let A and T be the set of applicants and tenants. An applicant turns to a tenant if it is admitted to the system. Let n_t be the replication degree of tenant t. Tenant t has a *demand profile P*, which is a set of n_t *demand vectors* (one vector per replica). Demand vector p specifies the resources needed by one replica. Let F_a, *feasibility region* of applicant a, be the set of servers that can be considered for placing its replicas. Let E_t be the set of the elected servers at the end of t's admission ($|E_t| = n_t$).

Example setup. To simplify the description of our algorithms in the next sections, we outline a hypothetical example setup here. Assume 4 servers and 3 resources as CPU (count), network bandwidth (Gbps), and storage IO (IOps). Resource vectors of our servers are $R_1 = (2,2,250)$, $R_2 = (2,2,400)$, $R_3 = (8,4,100)$, $R_4 = (4,2,250)$. We assume 2 applicants with 2 replicas each, and resource profiles as shown in Table 1.[3]

Constraints. *Resource allocator* is subject to the following constraints:

- **Cons**$_1$. A replica is *indivisible*; all the values in its vector are demanded from exactly *one* server, and it is *placed* if there exists a server to satisfy it.
- **Cons**$_2$. An applicant is *indivisible*; an applicant is *admitted* if there is a set of servers that can host *all* of its replicas.
- **Cons**$_3$. To cope with failures a server can host at most one replica of an applicant.

Applicant	Replica	CPU	Net-bw	Storage IO
a_1	p_1	1	1	100
	p_2	3	1	100
	$p_1 + p_2$	**4**	**2**	**200**
a_2	p_1	3	④	100
	p_2	1	1	150
	$p_1 + p_2$	**4**	**5**	**250**
$R_1 + R_2 + R_3 + R_4$		**16**	**10**	**1000**

Table 1: Example setup

Note. Algorithms designed in this section are inspired by DRF [20] due to its fairness criteria: when dividing multiple goods among many suppliants the goal is to maximize each suppliant' share, while equalizing the share of their most demanded good. Moreover, DRF is computationally light which is essential for making quick resource adjustments (see § 7).

4.1 Admission Phase

Admission phase is executed by the admission controller (*AC*), which analyzes *admission requests* and finds servers to place replicas. We propose a *placement algorithm* that efficiently places replicas on shared servers. Our algorithm runs until all applicants are admitted to the system or denied admission if satisfying their SLAs is infeasible. As described in § 3 an applicant uses Filo's API to specify its desired SLA and reliability requirements, which are then translated to CPU, network, and storage usage and saved as its demand profile. Using these profiles at each iteration the algorithm must decide: (a) which *applicant*, (b) which *replica* of the selected applicant, and (c) which *server*, to consider next in its allocations so to be efficient. Our choices are based on 3 policies that we explain next with the rational for each:

(PO$_a$). To maximize resource usage, we prioritize the applicant with the highest *dominant share*. To get an applicant's dominant share: we divide the aggregated resource demand of its replicas by the total free amount of resources in the system, and choose the highest value (see Eq 2). In the above example we prioritize applicant a_2 with the dominant share of $5/10$ (see Table 1).

(PO$_b$). For the selected applicant, we prioritize the replica with the highest dominant share. If there is no server to fit the heaviest replica, there is no point in fitting the others. To get a replica's dominant share: we divide its demand by the total free amount of the resources in the system, and choose the highest value among the resources. In the above example we prioritize a_2's first replica with the dominant share of $4/10$ (due to its network bandwidth, note the circle in Table 1).

(PO$_c$). To place the selected replica, we prioritize the servers with more free resources. The rational is to eventually balance resource utilization across the servers.

[3] In Chain Replication the head replica is loaded more than the other roles. Similarly in Paxos protocol, the replica that plays the coordinator role demands more resources than the others.

Placement Algorithm. Alg. 1 encapsulates our constraints and policies in more detail. The dominant share of applicant a (line 2) is obtained as follows. Before the admission, the feasibility region of an applicant is the entire server cluster, hence, its dominant share must be calculated without constraining the server set. Thus we compute the global amount of free resources as $R^g = \sum_{\forall s \in S} R_s$. We use a similar formula to calculate ${P_a}^g$ for applicant a (in the example $R^g = (16, 10, 1000)$, ${P_{a_1}}^g = (4, 2, 200)$, and ${P_{a_2}}^g = (4, 5, 250)$, as in Table 1). Then we use Eq 2 to calculate a's dominant share:

$$D_a = \max_{i=1}^{k} \{({P_a}^g)_i / (R^g)_i\} \tag{2}$$

(*e.g.*, D_{a_2} is 5/10). R^g and ${P_a}^g$ remain constant during the execution of the algorithm. Thus, the dominant shares of the applicants are calculated once at the beginning. Similarly, D_p, dominant share of replica p (line 5) is calculated using Eq 2, where p_i replaces $({P_a}^g)_i$. To find the most free servers we calculate servers' dominant shares (line 20) relative to their nominal capacity using Eq 3:

$$D_s = \max_{i=1}^{k} \{(R_s^{no} - R_s)_i / (R_s^{no})_i\} \tag{3}$$

i.e., dominant share of a server is determined by its most consumed resource. As R_s is modified frequently, at the end of an admission, D_s is recalculated to respect $\mathbf{PO}_c$ (line 15). Updating D_s at the middle of an admission is unnecessary as the elected servers are removed from the feasibility region to respect $\mathbf{Cons}_3$ (line 24). Resource usage across the servers is gradually equalized ($\mathbf{PO}_c$) using *ElectServer*. Once an applicant is admitted (*i.e.*, all of its replicas are placed), Filo activates token buckets and assigns their budgets based on the agreed SLA.

Failures. A chain is disrupted if any of its replicas fails. We invoke *ElectServer* on demand to replace failed replicas. Replacing failed replicas is prioritized over admitting new applicants. As moving correct replicas is unnecessary, the overhead of resuming the chain is confined to placing only the failed replicas. To restore the initial degree of resiliency we must also fetch the *logs* from the correct replicas (recovery). Recovery consumes resources and may impact the performance of the other tenants. An approach to preventing this issue is to always leave a fraction of the server resources unallocated. Determining this value is a trade off between recovery time and resource efficiency. A larger value speeds up the recovery, but reduces the efficiency of resource utilization at failure-free intervals. This value must be determined based on the frequency of failures and the uptime guarantees. If failures occur when all the servers are fully allocated, it will be impossible to place the failed replicas. To avoid this problem, similar to Cheap Paxos [33], we reserve a small subset of the servers as auxiliary that are used *for placing failed replicas*.

Algorithm 1 Placement (executed by *AC*)

```
 1: while $A \neq \emptyset$ do                {run until all applicants are addressed}
 2:   pick applicant $a \in A$ with highest dominant share           {$PO_a$}
 3:   $F_a \leftarrow S$
 4:   while $P_a \neq \emptyset$ do   {run until all replicas of a are addressed}
 5:     pick replica $p \in P_a$ with highest dominant share         {$PO_b$}
 6:     ElectServer ($p$)                                    {see line 18}
 7:     if replica $p$ is not placed then
 8:       $\forall s \in E_a$: update $R_s$                        {redeem resources}
 9:       $A \leftarrow A \setminus \{a\}$ {$Cons_2$: a is rejected if any replica is not
          placed}
10:       exit loop and goto (1)
11:     else                                          {replica p is placed}
12:       $P_a \leftarrow P_a \setminus \{p\}$
13:   end
14:   // if here, all replicas of applicant a are placed successfully
15:   $\forall s \in E_a$ : update $D_s$                                  {$PO_c$}
16:   $A \leftarrow A \setminus \{a\}$                                  {a is admitted}
17: end

18: ElectServer ($p$):
19: while $F_a \neq \emptyset$ do
20:   pick server $s \in F_a$ with the lowest dominant share          {$PO_c$}
21:   if $p \leq R_s$ then   {$Cons_1$: a server is found to place replica}
22:     $R_s = R_s - p$
23:     $E_a \leftarrow E_a \cup \{s\}$         {update the set of elected servers}
24:     $F_a \leftarrow F_a \setminus \{s\}$                                {$Cons_3$}
25:     exit and return (replica $p$ is placed)
26: end
27: return (replica $p$ is not placed)              {no server is found}
```

4.2 Work Conservation Phase

Tenants that have miss-estimated their SLAs at admission time, will need resources above or below their reservations at runtime. Filo monitors usage at runtime and temporarily re-distributes resources to address miss-estimations. Filo's objective at this phase is to frequently and fairly adjust allocations, and maximize global utilization without over-allocating. This phase executes in control intervals of a few seconds. After the resource adjustments, budgets of the token buckets must also be updated properly. Filo assigns either the *idle* or the *unallocated* resources to the demanding tenants. To understand the difference note that we only used the *unallocated* resources to admit tenants. Some tenants may not use all of their resources at runtime; these are allocated but *unused* resources. *Idle* resources include both the *unallocated* and the *unused* ones. Thus, by allocating from *idle* resources, Filo has more resources to (re-)use, but is subject to temporary SLA violations, which cannot happen by allocating only from *unallocated* resources. Accepting this risk is a decision left to Filo's operator, to which our algorithms are agnostic.

Example setup extended. We elaborate on our example from previous section to clarify this phase. Lets assume a_1's admission SLA is 100 reqs/sec that is equivalent to the resource usage shown in Table 1. Thus for example on replica p_1 each request costs (0.01 CPU, 0.01 Gbps, 1 storage IO). At runtime, a_1 realizes that in addition to 100 requests, it needs to submit 10 more reqs/sec (110 in total). This phase tries to maximize the number of additional requests it can grant to a_1, called its *utility*, while being fair to all the other tenants that demand extra requests (note that at the end we may only be able to grant 3 requests to a_1, although it wishes for 10).

More precisely, we define $utility(t)$ to be the number of t's extra granted requests at runtime. During this phase, t's feasibility region is limited to E_t: in other words extra requests are granted if sufficient free resources on E_t exist (**Cons**$_4$). Computing new allocations quickly is crucial for reducing the intervals in which resources remain idle. In the next section we first show how this phase can be done using a centralized algorithm (§ 4.2.1). Centrally controlling and re-assigning resources is computationally intensive; moreover, its overhead increases with the size of the system (see § 6). To alleviate this overhead, we propose two new distributed algorithms (Sections 4.2.2 and 4.2.3). *Our key insight in designing distributed algorithms is*: servers do not need global visibility over *all* the other servers to dynamically adjust allocations as each tenant's replicas are placed on a small set of servers. Servers that manage the same consensus group need to coordinate with each other; hence, servers coordinate locally, and they do not need a centralized oracle. The *distributed controller* implements those algorithms.

4.2.1 Centralized DRF (C-DRF)

As a baseline, we first illustrate `C-DRF` (Alg. 2). `C-DRF` maximizes tenant utilities while equalizing their dominant shares. Differently from `DRF` [20], `C-DRF` takes the indivisibility of the demand profiles into account and is subject to **Cons**$_4$, which is not previously addressed [20, 18]. `C-DRF` has two inputs, vector $\vec{\mathbf{R}}$ and set T. To produce $\vec{\mathbf{R}}$, we concatenate the free resource vectors of the servers, $|\vec{\mathbf{R}}| = m \times k$ (in our example $\vec{\mathbf{R}} = (2,2,250,\ 2,2,400,\ 8,4,100,\ 4,2,250)$). Similarly, we extend t's demand profile as:

$$\vec{\mathbf{P}}_{\mathbf{t}} = p_1 \ . \ p_2 \ . \ .. \ . \ p_m \qquad (4)$$

Any server not in E_t is presented by $\vec{\mathbf{0}}$ in Eq 4 (assume a_1 is admitted to $E_{a_1} = \{1,3\}$ then in the granularity of one request we have $\vec{\mathbf{P}}_{\mathbf{a_1}} = (0.01,0.01,1,\ 0,0,0,\ 0.03,0.01,1,\ 0,0,0)$): recall that the feasibility region of a tenant during this phase is restricted to the servers it is placed on.[4] We use Eq 5 to calculate t's dominant share (line 3):

$$D_t = \max_{i=1}^{k\times m} (\vec{\mathbf{P}}_{\mathbf{t}} \times utility(t))_i / (\vec{\mathbf{R}})_i \qquad (5)$$

In this formula $utility(t)$ is multiplied by each member of $\vec{\mathbf{P}}_{\mathbf{t}}$, producing a new vector of size $m \times k$. As $utility(t)$ is updated after each allocation (line 6), D_t must be recalculated (line 7). Servers periodically send their R_s vectors to AC, which centrally executes `C-DRF`. T in Alg. 2 includes only the tenants that request extra resources at runtime.

As we will see in § 6.3, `C-DRF` is computationally intensive. In the next two sections we propose two efficient distributed algorithms.

Algorithm 2 C-DRF $(\vec{\mathbf{R}}, T)$

```
 1: $\forall t \in T : utility(t) = 0$
 2: while $T \neq \emptyset$ do
 3:     pick tenant $t$ with the lowest dominant share
 4:     if $\vec{\mathbf{P}}_{\mathbf{t}} \leq \vec{\mathbf{R}}$ then
 5:         $\vec{\mathbf{R}} = \vec{\mathbf{R}} - \vec{\mathbf{P}}_{\mathbf{t}}$
 6:         $utility(t) = utility(t) + 1$
 7:         update $D_t$
 8:     else
 9:         $T \leftarrow T \setminus t$                {$E_t$ is saturated relative to $t$}
10: end
```

4.2.2 Head-DRF

`Head-DRF` (Alg. 3), our first distributed algorithm, is composed of two phases. During the allocation phase utilities are calculated by the head servers only using a local execution of `C-DRF`, and disseminated to the followers for voting. At voting phase followers cast their votes by prioritizing the allocations that maximize their local resource utilization (**PO**$_d$).[5]

Allocation Phase is executed by all the head servers in parallel. Server s is a head server if it hosts the head replica of *at least* one tenant. Server s uses Eq 6 to calculate its perspective of free resources in the cluster, $\vec{\mathbf{R}}_{\mathbf{s}}$:

$$\vec{\mathbf{R}}_{\mathbf{s}} = R_1 \ . \ R_2 \ . \ .. \ . \ R_m \qquad (6)$$

Any server that has no tenant in common with s is presented by $\vec{\mathbf{0}}$ in Eq 6. ($|\vec{\mathbf{R}}_s| = k \times m$). Each server periodically sends its R vector to all the servers that host the head replicas of its tenants. Demand profiles are extended using Eq 4, and their dominant shares are calculated using Eq 5, where $\vec{\mathbf{R}}_s$ replaces $\vec{\mathbf{R}}$. Head servers use this data to execute `C-DRF` locally, and compute the utilities. Head servers then calculate the potential allocations for each tenant (u_t-line 4), and propose them to the relevant followers. The proposed allocation for tenant t is accepted only if *all* the servers in E_t vote for its acceptance (lines 3—6).

Given that head servers lack the global visibility over the entire cluster and may concurrently consider common servers for allocations, server resources maybe

[4] For simplicity we have not modified servers' resource vectors. Note that after admission servers have fewer resources.

[5] NTP is used to synchronize control intervals, allocation, and voting phases.

over-allocated. The *voting phase* is designed to reduce the amount of over-allocations to zero.

Voting Phase is executed by all the servers in parallel. Each server receives exactly one proposal, u_t, for each of its tenants. At the beginning of the voting phase a server has received the proposals for all the tenants for which it hosts one of their replicas. Servers follow $\mathbf{PO}_d$ to cast their votes by prioritizing the tenant with the highest dominant share (line 9). Dominant shares are calculated by Eq 5, where $\vec{\mathbf{R}}_\mathbf{s}$ replaces $\vec{\mathbf{R}}$. A server accepts a proposal only if sufficient free resources are available (line 10). Proposed allocations are unbreakable, meaning that servers can only vote for the whole allocation (u_t).

Followers are at the intersection of the disconnected perspectives of the head servers. By voting based on their local free resources, they protect themselves from over-allocations. Rejecting *full* proposals (u_t), however, might induce a low utilization of the servers. We next propose All-DRF to alleviate this inefficiency.

Algorithm 3 Head-DRF

```
1: $T_{hs}$: is the set of tenants that their head replica is on $s$.
2: Allocation Phase: Executed by $s \in S$ if $s$ is head for some $t$
3: $\{utility(t)\} \leftarrow$ C-DRF$(\vec{\mathbf{R}}_s, T_{hs})$
4: $\forall t \in T_{hs}$ : $u_t = (P_t \times utility(t))$ ,
5:         $propose(u_t)$ to $t$'s followers ,
6:         $accept(u_t)$ if all replicas voted $accept$
7: Voting Phase: Exe. by $\forall s \in S$
8: while $true$ do
9:     pick $u_t$ with the highest dominant share          $\{\mathbf{PO}_d\}$
10:    if $u_t \leq R_s$ then                               $\{else\ reject\}$
11:        $R_s = R_s - u_t$
12:        $vote(u_t, accept)$
13: end
```

4.2.3 All-DRF

All-DRF (Alg. 4) is also composed of two phases. Differently from Head-DRF:

Allocation Phase is executed by all servers in parallel (hence the name). Each server executes C-DRF locally and sends its utilities to the relevant head servers. Differently from Head-DRF, servers propose utilities ($utility_t$) rather than the full allocations (u_t). Similarly to Head-DRF there is a high chance of over-allocation, which is eliminated during the voting phase.

Voting Phase is executed by the head servers only. For each t, a head server chooses the minimum of the utilities proposed by all the servers in E_t. As we will see in the experiments the allocations obtained by All-DRF are often closer to C-DRF.

4.2.4 Comparison

Communication complexity. We compare complexities of our algorithms assuming that at the start of a control interval, servers have sent their R_s vectors to AC in C-DRF and to head servers in Head-DRF. Assuming that tenants send their runtime profiles to AC, one communication step is sufficient to finalize the allocations for C-DRF: (1) AC runs C-DRF, and broadcasts its allocations to head servers and tenants, to tune their token buckets. Assuming that tenants send their runtime profiles to head servers, 3 steps are necessary to finalize the allocations for both Head-DRF and All-DRF. In Head-DRF: (1) head servers run C-DRF and broadcast their allocations to the followers, (2) followers send their votes to the head servers, and (3) head servers summarize the votes, tune their token buckets, and inform tenants. All-DRF is measured similarly.

Algorithm 4 All-DRF

```
$T_s$: is the set of tenants that any of their replicas is on $s$.
Allocation Phase: Executed by all $s \in S$
    $\{utility(t)\} \leftarrow$ C-DRF$(\vec{\mathbf{R}}_s, T_s)$
    $\forall t \in T_s$ : $propose(utility(t)_s)$ to $t$'s head server
Voting Phase: Exec. by $s \in S$ if $s$ is head for some $t$
    $\forall t \in T_{hs}$ : $utility(t) = \min_{\forall s \in E_t}\{utlity(t)_s\}$
```

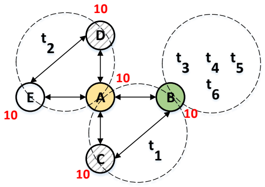


Figure 1: An example scenario for § 4.2.4

Efficiency. Distributed algorithms may diverge from the centralized approach. Consider All-DRF in Fig 1, with 5 servers each with 1 resource of 10 free units. t_1 as $(1,1,1,0,0)$ is placed on [A,B,C(head)], and t_2 as $(1,0,0,1,1)$ on [A,D(head),E]. At the end of the allocation phase, C,D,E assign 10, A assigns 5, and B assigns 2 utilities to their tenants (B is shared by other tenants). Head servers in parallel finalize the allocations: t_2 and t_1 receiving 5 and 2 utilities each. A has 3 free units that t_2 could use in an ideal situation (the values can be easily obtained by walking through the algorithm). Our evaluations will show that this inefficiency is negligible due to the large number of tenant replicas packed on a server.

4.3 Coordinating the two phases

To avoid over-allocation of the resources due to the concurrent execution of the admission and work conservation phases, AC notifies the elected servers before finalizing the admissions. If the servers have simultaneously allocated their resources, they either immediately terminate their allocations or ask AC to delay the admissions until the temporary allocations expire.

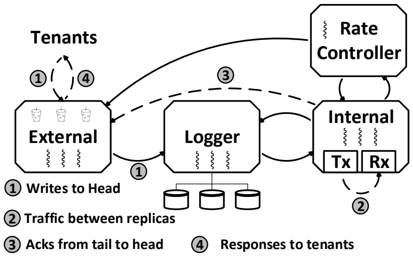


Figure 2: Internal architecture of a replica server. Dashed arrows refer to inter-server communications.

5 Implementation

Filo is written in C/C++ (11.5 KLOC). Each server is implemented as a pipeline of 3 multi-threaded stages (Fig 2). The *external* stage handles connections of tenants with the head replicas. The *Logger* stage persists data on the storage device, and the *internal* stage handles connections among servers. The number of threads, MPL (Multi Programming Level) at each stage is 1 to 4. Each server also has a thread for the distributed controller. Communications among the *stages* is based on shared memory. We optimized network performance by using jumbo frames (8968 B), by disabling Nagle algorithm, and by enabling Receive Side Scaling (RSS) [24]. In some experiments, the RSS mapping resulted in an uneven distribution, at which point despite the availability of free cores the performance was capped. We also implemented a simulator in C/C++ (2.5 KLOC) to test the resource allocator in larger configurations.

6 Evaluation

We evaluate Filo's performance in failure-free scenarios. In § 6.1 we extensively evaluate performance under various workloads, which is needed for bounding SLA guarantees, and calculating costs. Then we feed our findings into the simulator and evaluate the placement algorithm (§ 6.2), and the distributed controller (§ 6.3) with various number of applicants. In § 6.4 we demonstrate the effect of our resource allocator in the testbed.

Testbed. We ran the experiments in a cluster of 10 Dell DCS7220N servers each with two 10-core Intel(R) Xeon(R) CPU E5-2470 v2, 2.40 GHz CPUs, a 10 Gbps Mellanox ConnectX-3 Pro NIC, and 128 GB of RAM with hyper threading enabled (due to the administrative requirements disabling hyper threading was not an option.) and two HDDs (Seagate ST2000NM0033-9ZM175). Our servers use Microsoft Windows Server 2012 Data center. Unless mentioned otherwise, we ran each experiment for 60 seconds. Our graphs report the average application-level throughput in number of reqs/sec (throughput in network bandwidth can be directly calculated), total CPU utilization, and CPU utilization of the busiest logical core on one selected server.

6.1 Service benchmarking

Fig 3 shows Filo's performance in the following setup.

Setup. We vary request sizes from 64 B to 32 KB, and *MPL* from 1 to 4 for *in-mem* and *persistent* modes. In the *persistent* mode data is asynchronously transferred to the storage device and 2 (with $MPL \leq 2$) and 4 (with $MPL \geq 3$) threads are created by the *Logger*(s). Tenants send their requests in an open loop bounded by a *pending window* (*e.g.*, with a window of 10, a tenant can have at most 10 outstanding requests). We varied window size until peak performance was reached. In each setting the number of tenants is equal to *MPL*, and each has 3 replicas on 3 servers. Note that requests of a specific tenant at any given point must be handled by one thread at most for safety. One thread, however can handle the requests of many tenants. Each vertical CPU bar shows the total CPU (horizontal line) and the CPU of the busiest core.

General Results. The *in-mem* mode has higher throughput than the *persistent* mode, as the latter is capped by the capacity of the storage device and the *Logger*'s CPU. The *in-mem* mode has lower latency (1-5 msec) than the *persistent* mode (1-8 msec).

In-mem mode. As *MPL* increases, throughput increases for 64 B-4 KB requests. For 8 KB-32 KB requests, throughput increases when *MPL* (number of tenants effectively) increases from 1 to 2, and then drops: as the number of tenants increases, so does the number of outstanding requests; resources used for receiving requests are now spent for receiving requests from new tenants, exchanging new acks, and responses. As an example for *in-mem*, 2 KB, $MPL = 3$, and throughput of 150 Kreq/s, in addition to 150K requests, 150K acks, and 150K responses are transmitted among the servers, and to the tenants respectively (overhead not shown in the graphs). Often one core is saturated by the interrupts caused by sending and receiving network messages.

Persistent mode. Throughput increases as *MPL* increases. With $MPL \leq 2$ each server has only one *Logger*, through which all the threads in the storage stage transfer their requests to the disk. When *MPL* increases from 1 to 2, *group commit* improves and throughput increases (the reason our throughput surpasses IoMeter [25]). *Logger* is a point of synchronization, and eventually saturates its core for the ≤2 KB requests and disk's bandwidth for requests ≥4 KB. To scale with $MPL \geq 3$ we added a second *Logger* to each server (latency was 1-8 msec).

Aiding the admission phase. We use our numbers to bound (a) SLAs, and (b) the number of tenants with specific admission requests that can be packed on a server. To understand (a) notice that for example Filo can at most promise 76 K (64 B, *in-mem*) reqs/sec to a single tenant.

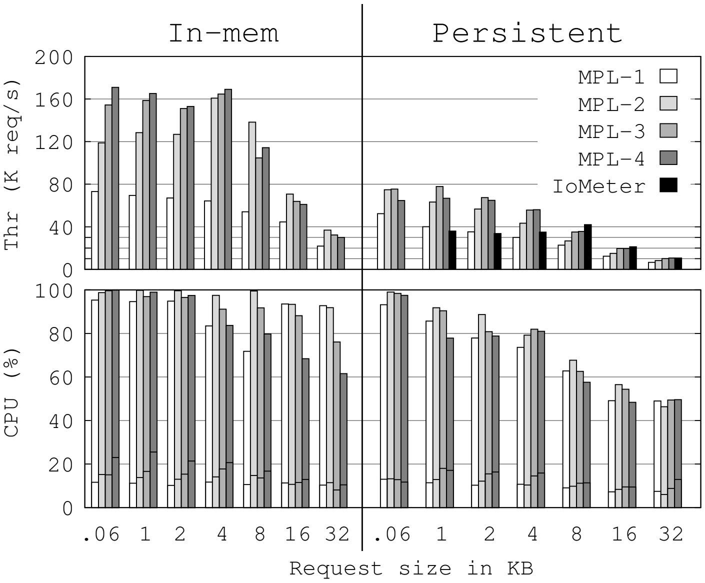


Figure 3: Evaluating Filo's performance.

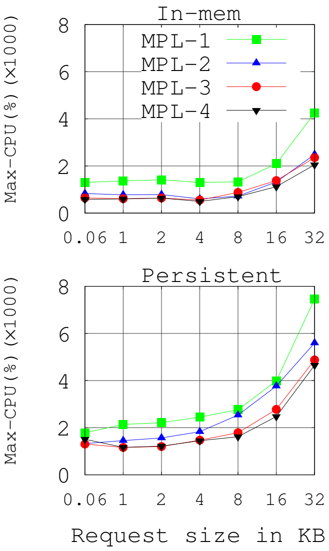


Figure 4: CPU costs, Y-axis is magnified by 1000 for readability (§ 6.1.1).

Promising SLAs above this value to one tenant is meaningless: requests of a specific tenant must be handled sequentially, therefore the peak SLA for a tenant is determined by $MPL = 1$. In other words, multiple threads in a replica can not handle the same tenant's requests. To understand (b) notice that for example a server can process 170 K (64 B, *in-mem*) reqs/sec at most. Thus the aggregate SLAs of all the tenants on one server for 64 B reqs should not exceed this value.

6.1.1 Calculating Resource Costs

We use our numbers from § 6.1 to estimate the costs of different workloads on CPU, network bandwidth, and storage IO, which are used for translating SLAs to resource costs.

CPU. We use Fig 3 to obtain the costs on the saturated CPU as shown in Fig 4. Y-axis is magnified by a factor of 1000 for readability. The cost on the total CPU can be obtained similarly. For all *MPL* the overhead on the CPU increases as the size of the requests increases. Larger requests result in higher number of interrupts and therefore higher cost on the CPU; this is because larger requests are broken into more frames (note MTU). The CPU cost in the *persistent* mode is higher than *in-mem* due to *Logger*'s overhead.

Storage IO. Throughputs in Fig 3 benefit from the *group commit* offered by *Logger*. We must exclude this benefit when the system is not highly loaded. Thus to calculate storage costs we use IoMeter's values instead. At its maximum, IoMeter measures ~40 K IOs for 1-8 KB requests. With a peak value of 40 K IOs, requests ≤ 8 *KB* translate to one IO each. 16 KB and 32 KB requests translate to 2 and 4 IOs respectively. To understand note that for 16 KB and 32 KB requests, IoMeter measures 20 K and 10 K IOs respectively. Given that *Logger* writes each request twice (spread across two storage devices), we multiply the number of IOs by 2 .

Network bandwidth. To calculate network costs we use request sizes directly (e.g., a 1 KB request costs 1 KB network bandwidth).

Inaccuracies in the cost model. The assessment space of a complex system such as Filo is extremely large, and Fig 3 covers only a tiny fraction of this space. Hence, calculating costs using only Fig 3 is prone to inaccuracies that can lead to SLA violations. To prevent SLA violations we can base our admission decisions on x % of the peak throughput. For example if Filo can provide 76 K (64 B, *in-mem*) reqs/sec, with x as 80% a tenant can at most be promised ≈ 60 K reqs/sec. If x is chosen conservatively, resources will be under-allocated, but work conservation phase will compensate for it.

Generalizing costs. For workloads not covered by our evaluations costs are obtained by scaling.

Recalculating costs. As the hardware specifications or the implementation changes, performance must be reassessed, and cost estimations must be recalculated.

6.2 Admission Phase

We use our simulator to evaluate the placement algorithm (§ 4.1) in on-line and off-line settings; applicants arrive one at a time in the former and all at once in the latter. We order tenants with increasing and decreasing dominant shares in on-line and off-line settings respectively, to account for the worst and best cases for each.

Setup. We assume 10 servers and up to 150 applicants with a mixture of requirements. We use uniform distributions to choose durability mode (*in-mem* or *persistent*), requests sizes (64 B to 32 KB), and replication degree

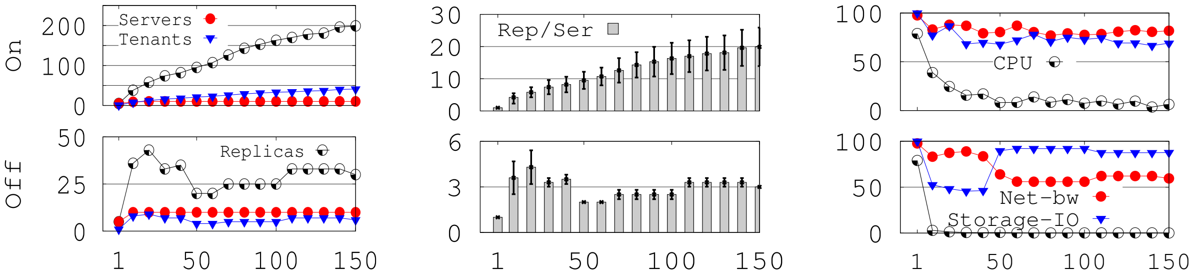


Figure 5: Placement algorithm in on-line and off-line settings. X-axis shows the number of applicants (§ 6.2).

(3,5,7). To simulate our testbed characteristics the SLA for each applicant is bound by our evaluations in § 6.1 (*e.g.*, with 64 B, *in-mem* the peak SLA an applicant can be promised is 76 K reqs/sec). Replica placement on each server is bound by our numbers from Fig 3. We use our cost model from § 6.1.1 to translate admission requests to resource profiles.

Results. Fig 5 shows results in 3 graphs. Left: number of used servers, admitted tenants, and replicas; Middle: number of replicas per server (average, 95% confidence interval); Right: percentage of free resources at the end of the admission phase. In both on-line and off-line modes the applicants are denied admission if any of the resources (in this case, CPU) is saturated. In the on-line mode as the number of applicants increases, fewer tenants are admitted but the number of replicas increases (e.g., with 150 applicants, ≈ 200 replicas are packed on 10 servers). On-line case admits more tenants and packs more replicas. As expected, off-line mode selects fewer applicants but the ones that maximize resource utilizations. Although off-line results in more efficient allocations (Fig 5(right)), it is hard to know the list of applicants a priori. In both cases our placement algorithm is efficient and all the 10 servers are efficiently utilized.

6.3 Work Conservation Phase

We use our simulator to compare `C-DRF`, `Head-DRF` and `All-DRF` for computation and communication overheads, utilization, fairness, and SLA violations in the on-line setup of § 6.2. We consider a case where half of the tenants are not using their reservations, and the other half demand extra rates. Resource profiles for the work conservation phase are equal to the admission profiles, but in the granularity of one request (see § 4.2 for explanation).

Computation overhead. Fig 6(a) shows the time it takes to compute allocations. As the number of tenants increases, compared to the centralized, distributed algorithms perform faster (∼ 5 times at best). We argued in § 4.2 that computing allocations quickly is important for reducing the periods in which resources remain idle.

Communication overhead. Fig 6(b) shows the number of messages. Compared with `C-DRF`, distributed algorithms exchange about 8 times more messages (150 msgs at worst). Given the amount of service-level messages, this overhead is negligible. These messages can further be piggybacked on the service messages.

Resource utilization. Fig 6(c) shows the aggregate amount of the free resources before and after work conservation (we have eliminated network and storage due to space limits). Our distributed algorithms result in allocations that compare closely with the `C-DRF`: `All-DRF` is 95% as efficient as `C-DRF`, and `Head-DRF` about 75%.

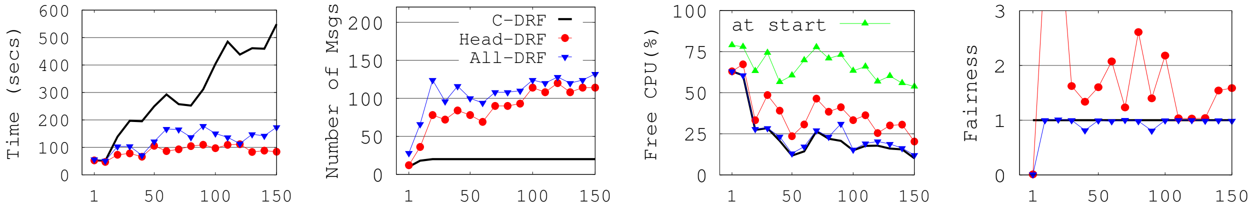
Fairness. Fig 6(d) shows fairness. As a fairness criterion, we have used normalized standard deviation for the *utility*, with `C-DRF` as the reference. An algorithm is more fair if this value is smaller. For example if with a total budget of 15 reqs/sec and 3 tenants, an algorithm allocates 5 to each tenant, it is more fair than an algorithm that allocates 10, 5, and 0. Unlike `Head-DRF`, `All-DRF`'s fairness compares closely to that of the `C-DRF`. This is because proposals can only be accepted as a whole in `Head-DRF`, but partially in `All-DRF`.

SLA violations. To absorb the exceeding demands of the tenants we used all the *idle* resources. `C-DRF` and `All-DRF` are subject to above 95% SLA violations. This is because these two algorithms are very efficient at allocating resources. Whether to use reserved resources is part of system's policy and does not affect the semantics of our algorithms (see § 4.2 for details).

To conclude, `All-DRF` compares well with `C-DRF` in efficiency and fairness and in addition is about 5 × faster. `Head-DRF` has lower computation time, but it is 70% as efficient as `C-DRF` and is less fair. We observed similar results in other settings, where for example replica demand vectors were skewed.

6.4 Performance Isolation

Fig 7 shows the impact of the resource allocator and rate limiters with 3 tenants, 3 replicas each in the *persistent* mode, and SLA of 6.5 K-1 KB reqs/sec each. Assume *A* and *B* will need more resources at runtime. In the first 4 minutes only *B* and *C* are in the system, and in the first 2 minutes rate limiters are disabled. At min 1, *B* increases its rate above its SLA and affects *C*'s performance. At min 2 rate limiters are enabled and hence SLAs restored. At min 3, *C* voluntarily reduces its rate for the next 3 minutes. At min 4 we activate `All-DRF`, admit *A*, but intentionally leave its runtime extra demand



(a) (b) (c) (d)

Figure 6: Evaluating C-DRF, Head-DRF, All-DRF. X-axis shows the number of applicants (§ 6.3 for details).

out of All-DRF's sight. The algorithm grants to *B* all of *C*'s underutilized resources neglecting *A*. At min 5, *A* is considered by All-DRF as well. At this point the underutilized resources of *C* are fairly divided between *A* and *B*. At min 6, *C* restores its SLA rate.

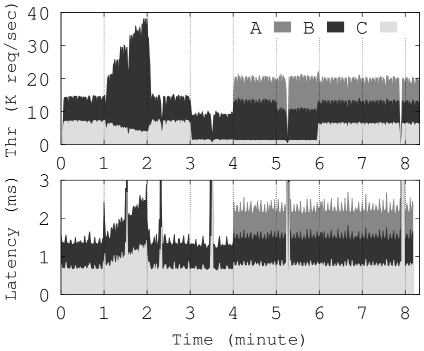


Figure 7: The impact of the distributed controller.

7 Related work

Consensus. Consensus is widely studied in the database and distributed systems communities [7, 30, 17, 16, 14, 19, 34]. Although some works have studied the co-location of consensus instances on shared servers [12, 37, 5], no one has considered the problem of efficient replica placement and the impact of sharing on the performance of individual instances. Filo is the first system to provide consensus as a service for the multi-tenant environment of the cloud platforms, while providing SLA guarantees, performance isolation, and efficiency of resource utilization. Similarly to Filo, [36, 4] use Chain Replication [42] for providing ordering and storage guarantees.

Token Buckets. Filo uses token buckets for rate limiting. Token buckets are previously used in network rate limiting [48, 1]. Filo's approach in determining and constantly updating the budget of the token buckets is novel, which is realized by its resource allocator component.

Distributed Controller. Filo is similar to Retro [35] in its objectives for ensuring performance isolation and efficient resource utilization. Unlike Retro, Filo uses a distributed controller for dynamically tuning the rate limiters. Few works have considered distributed rate tuning. [41] proposes a distributed approach, where each node is equipped with a rate limiter. A user is given an aggregate global budget, and each rate limiter's bucket is initialized with this budget. Each node removes tokens based on its own usage rate and the estimated sum of the usage rates at all the other nodes. This work is later extended in [47]. Consensus groups in Filo are composed of small number of servers that can locally coordinate their resource usage and achieve high resource efficiency without needing visibility over the entire cluster. Moreover, given the distributed model of the consensus and the connection links that are already established, designing a distributed controller in Filo had no additional cost. Our evaluations showed that our distributed algorithms exchange a small amount of messages to coordinate and finalize the allocations. Compared with centralized controllers, distributed controllers are computationally less intensive and faster.

Resource Allocation. Multi-resource allocation is a multi-resource bin packing problem [6, 15, 22, 27, 29, 45]. Filo's allocation algorithms are inspired by DRF [20, 18, 40]. DRF is not guaranteed to converge to the global optimum; [39, 21] propose heuristics algorithms for converging to the global optimum, which are computationally intensive and not suitable in our context.

Empirical quantification. We used an empirical approach to quantifying Filo prior to its launch [50, 26, 1]. Our strategy can be enhanced further by dynamically modifying the assessments at runtime.

8 Conclusion

We presented Filo, the first system to provide multi-tenant consolidated consensus as a cloud service. We argued that providing performance guarantees and isolation is particularly important in the cloud platforms where tenants share and compete over resources. We proposed a novel placement algorithm for admitting tenants, and two distributed algorithms for efficient and fair allocation of free resources at runtime. Our algorithms exploit the nature of the consensus service, and while being much faster provide comparable efficiency and fairness compared with the centralized algorithms.

9 Acknowledgments

We thank Miguel Castro, Austin Donnelly, Greg O'Shea, Richard Black, reviewers, and Mohit Aron for their feedback and support.

References

[1] S. Angel, H. Ballani, T. Karagiannis, G. OShea, and E. Thereska End-to-end performance isolation through virtual datacenters. In: *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association. 2014.

[2] *Apache BookKeeper*. Apache. 2014. URL: `http://bookkeeper.apache.org/`.

[3] *Apache HBase*. Apache. 2015. URL: `http://hbase.apache.org/`.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck Tango: Distributed data structures over a shared log. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013.

[5] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato Building global and scalable systems with atomic multicast. In: *Proceedings of the 15th International Middleware Conference*. ACM. 2014.

[6] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica Hierarchical scheduling for diverse datacenter workloads. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013.

[7] K. P. Birman, R. Van Renesse, et al. *Reliable distributed computing with the Isis toolkit*. Vol. 85. IEEE Computer society press Los Alamitos, 1994.

[8] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In: *NSDI*. 2011.

[9] M. Burrows The Chubby lock service for loosely-coupled distributed systems. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008).

[11] B. Charron-Bost, F. Pedone, and A. Schiper *Replication: theory and Practice*. Vol. 5959. springer, 2010.

[12] B. Darnell *Scaling Raft*. 2015. URL: `http://www.cockroachlabs.com/blog/scaling-raft/`.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007).

[14] D Dolev, and H. Strong *Distributed commit with bounded waiting*. IBM Thomas J. Watson Research Division, 1982.

[15] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial No justified complaints: On fair sharing of multiple resources. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM. 2012.

[16] C. Dwork, N. Lynch, and L. Stockmeyer Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988).

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985).

[18] E. Friedman, A. Ghodsi, and C.-A. Psomas Strategyproof allocation of discrete jobs on multiple machines. In: *Proceedings of the fifteenth ACM conference on Economics and computation*. ACM. 2014.

[19] H. Garcia-Molina Elections in a distributed computing system. *Computers, IEEE Transactions on* 100, 1 (1982).

[20] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In: *NSDI*. Vol. 11. 2011.

[21] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella Multi-resource packing for cluster schedulers. In: *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM. 2014.

[22] A. Gutman, and N. Nisan Fair allocation without trade. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents AND Multiagent Systems. 2012.

[23] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed ZooKeeper: Wait-free Coordination for Internet-scale Systems. In: *USENIX Annual Technical Conference*. Vol. 8. 2010.

[24] *Introduction to Receive Side Scaling*. Microsoft. URL: `https://msdn.microsoft.com/en-us/library/windows/hardware/ff556942(v=vs.85).aspx`.

[25] *Iometer benchmark*. Intel Corporation. 2013. URL: `http://www.iometer.org/`.

[26] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron Bazaar: Enabling predictable performance in datacenters. *Microsoft Res., Cambridge, UK, Tech. Rep. MSR-TR-2012-38* (2012).

[27] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang Multiresource Allocation: Fairness–Efficiency Tradeoffs in a Unifying Framework. *Networking, IEEE/ACM Transactions on* 21, 6 (2013).

[28] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about Eve: Execute-Verify Replication for Multi-Core Servers. In: *OSDI*. Vol. 12. 2012.

[29] I. Kash, A. D. Procaccia, and N. Shah No agent left behind: Dynamic fair division of multiple resources. *Journal of Artificial Intelligence Research* (2014).

[30] L. Lamport Paxos made simple. *ACM Sigact News* 32, 4 (2001).

[31] L. Lamport, D. Malkhi, and L. Zhou Reconfiguring a state machine. *ACM SIGACT News* 41, 1 (2010).

[32] L. Lamport, D. Malkhi, and L. Zhou Vertical paxos and primary-backup replication. In: *ACM symposium on Principles of distributed computing*. ACM. 2009.

[33] L. Lamport, and M. Massa Cheap paxos. In: *Dependable Systems and Networks, 2004 International Conference on*. IEEE. 2004.

[34] B. W. Lampson Replicated commit. In: *Circulated at a workshop on Fundamental Principles of Distributed Computing, Pala Mesa, CA*. 1980.

[35] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. USENIX Association, 2015.

[36] D. Malkhi, M. Balakrishnan, J. D. Davis, V. Prabhakaran, and T. Wobber From paxos to CORFU: a flash-speed shared log. *ACM SIGOPS Operating Systems Review* 46, 1 (2012).

[37] P. J. Marandi, M. Primi, and F. Pedone Multi-ring paxos. In: *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE. 2012.

[38] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal Extended virtual synchrony. In: *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*. IEEE. 1994.

[39] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder Heuristics for vector bin packing. *research. microsoft. com* (2011).

[40] D. C. Parkes, A. D. Procaccia, and N. Shah Beyond dominant resource fairness: extensions, limitations, and indivisibilities. *ACM Transactions on Economics and Computation* 3, 1 (2015).

[41] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007).

[42] R. van Renesse, and F. B. Schneider Chain Replication for Supporting High Throughput and Availability. In: *OSDI*. Vol. 4. 2004.

[43] N. Schiper, and S. Toueg A robust and lightweight stable leader election service for dynamic systems. In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE. 2008.

[44] F. B. Schneider Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990).

[45] D Shah, and D Wischik Principles of resource allocation in networks. In: *Proceedings of the ACM SIGCOMM Education Workshop*. 2011.

[46] E. G. Sirer, and D. Altinbuken *Commodifying Replicated State Machines with OpenReplica*. Technical Report 1813-29009. Cornell University, 2012.

[47] R. Stanojevi, and R. Shorten Fully decentralized emulation of best-effort and processor sharing queues. *ACM SIGMETRICS Performance Evaluation Review* 36, 1 (2008).

[48] A. S. Tanenbaum Computer networks, 4-th edition. *ed: Prentice Hall* (2003).

[49] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu Ioflow: A software-defined storage architecture. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013.

[50] B. Urgaonkar, P. Shenoy, and T. Roscoe Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review* 36, SI (2002).

[51] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin Separating agreement from execution for byzantine fault tolerant services. In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003.