Slicer: Auto-Sharding for Datacenter Applications

Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, and Arif Merchant, *Google;*
Kfir Lev-Ari, *Technion—Israel Institute of Technology*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Slicer: Auto-Sharding for Datacenter Applications

Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari[†]

Google [†]*Technion - Israel*

Abstract

Sharding is a fundamental building block of large-scale applications, but most have their own custom, ad-hoc implementations. Our goal is to make sharding as easily reusable as a filesystem or lock manager. Slicer is Google's general purpose sharding service. It monitors signals such as load hotspots and server health to dynamically shard work over a set of servers. Its goals are to maintain high availability and reduce load imbalance while minimizing churn from moved work.

In this paper, we describe Slicer's design and implementation. Slicer has the consistency and global optimization of a centralized sharder while approaching the high availability, scalability, and low latency of systems that make local decisions. It achieves this by separating concerns: a reliable data plane forwards requests, and a smart control plane makes load-balancing decisions off the critical path. Slicer's small but powerful API has proven useful and easy to adopt in dozens of Google applications. It is used to allocate resources for web service front-ends, coalesce writes to increase storage bandwidth, and increase the efficiency of a web cache. It currently handles 2-7M req/s of production traffic. The median production Slicer-managed workload uses 63% fewer resources than it would with static sharding.

1 Introduction

Many applications require the resources of more than one computer, especially at Google's typical scale. An application that distributes its work across multiple computers requires some scheme for splitting it up. Often, work is simply split randomly. This is ubiquitous in web services, where the dominant architecture puts a round-robin load-balancer in front of a fleet of interchangeable application processes ("tasks").

However, in many applications, it is hard to ensure that every task can service any request. For example,

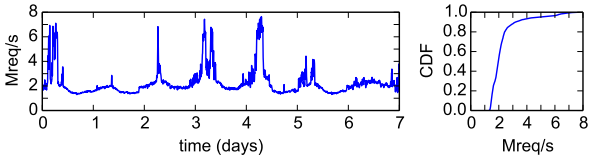

Figure 1: Over five-minute intervals in a recent week, Slicer directed a median of 2 Mreq/s of production traffic with peaks exceeding 7 Mreq/s.

Google's speech recognizer (§3.2.1) uses a different machine learning model for each spoken language. Loading a model is too slow for interactive use: a language must be resident before a request arrives. One task cannot fit every model, making random request balancing untenable. Instead, each task loads only a subset of languages, and incoming requests are routed to a prepared task.

In the past, Google applications like the speech recognizer had their own one-off sharders. Experience taught us that sharding is hard to get right: the plumbing is tedious, and it can take years to tune and cover corner cases. Rebuilding a sharder for every application wastes engineering effort and often produces brittle results.

In practice, custom sharders typically make do with simplistic static sharding that is unresponsive to changes in workload distribution and task availability. Simple schemes utilize resources poorly. In the speech recognizer, resources required per language peak at different times as speakers wake and sleep. When tasks fail, requests must be redistributed among the healthy tasks. When a datacenter fails, a great wave of traffic sloshes over to the remaining datacenters, dramatically altering the request mix. Before Slicer, the speech team handled variation with overprovisioning and manual intervention.

Slicer refactors sharding into a reusable and easily adopted building block akin to a filesystem or lock manager. Slicer is a general-purpose infrastructure service

that partitions work across tasks in applications that benefit from affinity. Slicer is minimally invasive to applications: they need only associate incoming requests with a key of their choice that is used to rendezvous requests with tasks. In the speech recognizer, the slice key is the language. Other applications use fine-grained slice keys, such as usernames or URLs. Slicer assigns part of the key space to each task and routes incoming requests to them via integration with Google's front-end load balancers and RPC system.

Slicer addresses these needs by sharding dynamically. It monitors the request load to detect hotspots. It monitors task availability changes due to service provisioning, system updates, and hardware failures. It rebalances the key mapping to maintain availability of all keys and reduce load imbalance among tasks while minimizing key churn.

Slicer can trade off consistency with availability, offering either strongly or eventually consistent assignments. In consistent assignment mode, no task ever believes a key is assigned to it if the Assigner does not agree. The simplest application of this property ensures that at most one task is authoritative for a key, reducing availability but making it easy to write a correct application that mutates state. Alternatively, Slicer can distribute overlapping eventually consistent assignments, eliminating periods of unavailability and reacting rapidly to load shifts.

Slicer's design differs significantly from past sharding systems, driven by its use in dozens of large-scale systems at Google. Slicer provides global optimization and consistency guarantees possible with a centralized load-balancer, but it achieves nearly the same resilience to failures and low latency as systems that make purely local decisions, such as distributed hash tables.

In a production environment, customers cannot tolerate flag days (synchronized restarts). By separating the forwarding data plane from the policy control plane, Slicer simplifies customer-linked libraries and keeps complexity in a central service where the team can more easily coordinate changes.

This functionality is all exposed through a narrow, readily adopted API that has proven useful in Google applications with a variety of needs:

Avoiding storage overhead. A stateless front-end that accesses underlying durable storage on every request is conceptually simple but pays a high performance cost over keeping state in RAM. In some applications, including our speech recognizer, this overhead dwarfs all other time spent serving a user request. For example, a Google pub-sub service[9] processes 600 Kreq/s, most of which do one hash and one comparison to a hash in memory. Fetching the hash via a storage RPC would be correct but incur far more overhead and latency.

Automatic scaling. Many cluster management systems can automatically expand the number of tasks assigned to a job based on load, but these are typically coarse-grained decisions with heavyweight configuration. Our speech recognizer handles dozens of languages, and Slicer's key redundancy provides a single-configuration mechanism to independently scale those many fine-grained resources.

Write aggregation. Several event processors at Google (§3.3.1) ingest huge numbers of small events and summarize them by key (such as data source) into a database. Aggregating writes from stateless front ends is possible, but aggregating like keys on the same task can be more efficient; Data Analysis Pipeline sees 80% fewer storage requests. Affinity provides similar benefits for other expensive, immobile resources like network sockets: Slicer routes requests for an external host to one task with the socket already open.

Sharding state is well-studied; see Section 6. Slicer draws on storage sharding [2, 14, 15] but applies to more classes of application. Compared to other general-purpose sharding systems [5, 10, 8, 13], Slicer offers more features (better load balancing, optional assignment consistency, and key replication) and an architecture focused on high availability.

This paper makes the following contributions:

- An architecture that separates the assignment generation "control plane" from the request forwarding "data plane", which provides algorithmic versatility, high performance, resilience to failure, and exploits existing lease managers and storage systems as robust building blocks.
- An effective load-balancing algorithm that minimizes key churn and has proven effective in a variety of applications.
- An evaluation on production deployments of several large applications that shows the benefits and availability of the Slicer architecture.

2 Slicer Overview and API

Slicer is a general-purpose sharding service that splits an application's work across a set of *tasks* that form a *job* within a datacenter, balancing load across the tasks. A "task" is an application process running on a multi-tenant host machine alongside tasks from other applications. The unit of sharding in Slicer is a key, chosen by the application. Slicer integrates with Google's Stubby RPC system to easily route RPCs originating in other services and with Google's frontend HTTP load balancers to

route HTTP requests from external browsers and REST clients.

Slicer has the following components: a centralized *Slicer Service*; the *Clerk*, a library linked into application clients; and the *Slicelet*, a library linked into application server tasks. (Figure 2). The Service is written in Java; the libraries are available in C++, Java, and Go. The Slicer Service generates an *assignment* mapping key ranges ("*slices*") to tasks and distributes it to the Clerks and Slicelets, together called the *subscribers*. The Clerk directs client requests for a key to the assigned task. The Slicelet enables a task to learn when it is assigned or relieved of a slice. The Slicer Service monitors load and task availability to generate new assignments to maintain availability of all keys. Application code interacts only indirectly with the Slicer Service via the Clerk and Slicelet libraries.

2.1 Sharding Model

Application keys may be fine-grained, such as user IDs, or coarse-grained, such as the languages in the speech recognizer described in Section 3.2.1. Keys are an atomic unit of work placement: all state associated with a single key will be collocated on those task replicas to which the key is assigned, but different keys may be assigned to different tasks. Slicer does not observe application state; it merely notifies the task of the keys the task should serve.

Slicer hashes each application key into a 63-bit *slice key*; each slice in an assignment is a range in this hashed keyspace. Manipulating key ranges makes Slicer's workload independent of whether an application has ten keys or a billion and means that an application can create new keys without Slicer on the critical path. As a result, there is no limit on the number of keys nor must they be enumerated.

Hashing keys simplifies the load balancing algorithm because clusters of hot keys in the application's keyspace are likely uniformly distributed in the hashed keyspace.

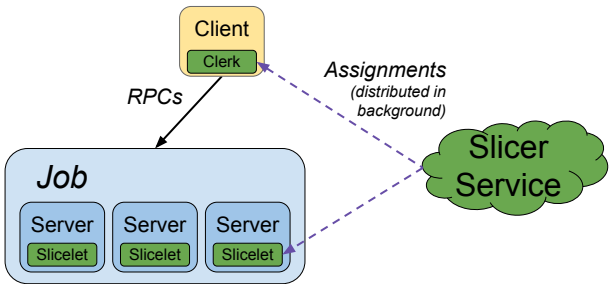

Figure 2: Abstract Slicer architecture.

The cost is lost locality: contiguous application keys are scattered. Many Google applications are already structured around single-key operations rather than scans, encouraged by the behavior of existing storage systems. For others, Section 2.2 offers a mitigation.

Some applications require all requests for the same key to be served by the same task, for example, to maintain a write-through cache. For these, Slicer offers a consistency guarantee on what assignments a Slicelet can observe (§4.5). For many other applications, weaker semantics are correct even when requests for the same key are served by different tasks. For example, such systems serve read-only data (such as Google Fonts), or provide weak consistency to their users (such as Cloud DNS), or have an underlying storage system that provides strong consistency (such as event aggregation systems).

Such applications can configure Slicer with *key redundancy*, allowing assignment of each slice to multiple tasks. Slicer honors a minimum redundancy to protect availability and automatically increases replication for hot slices, which we call *asymmetric key redundancy*.

2.2 Slicelet Interface

The application server task interacts with Slicer via the "Slicelet" API (Figure 3). A simple application, like the Flywheel URL status cache (§3.1.1), is free to ignore this API entirely and answer whatever requests arrive; Slicer transparently improves performance. An application may register a `SliceletListener` to learn when slices arrive and depart, so it can prefetch and garbage-collect state (such as the speech models in Section 3.2.1).

A few affinity-mode applications use `isAffinitizedKey` to discover misrouted requests, such as when retrying a request from the client is cheaper than processing it at the wrong server (§3.3).

```
interface Slicelet {
  boolean isAffinitizedKey(String key);
  Opaque getSliceKeyHandle(String key);
  boolean isAssignedContinuously(Opaque handle);
}
interface SliceletListener {
  void onChangedSlices(List<Slice> assigned,
       List<Slice> unassigned);
}
```

Figure 3: Slicer Server API

To support applications that require exclusive key ownership to maintain consistent in-memory state, the Slicelet provides an API inspired by Centrifuge [10]. The task calls `getSliceKeyHandle` when a request arrives, and passes the handle back to `isAssignedContinuously` before externalizing the result. Note that checking assignment at beginning and end is insufficient, since the slice may have been unassigned and reassigned

in the meantime. A task may also cache a handle across multiple requests, for example to cache a user's inbox during a session.

To scan its store to preload state, an application may need to map from hashed slices keys back to original application keys. Applications with few keys (such as language names in the speech recognizer) can precompute an index at each task. Applications with many keys typically adjust their storage schema, either by prefixing the primary key with the hashed slice key or by adding a secondary index. In future work, Slicer will support unhashed application-defined keys and implement range sharding to preserve locality among adjacent application-defined keys.

By default, Slicer load balances on request rate (req/s). The Slicelet integrates with Stubby to transparently monitor request rate per slice. Some applications have highly variable cost per request, or want to balance a different metric like task CPU utilization. An extension to the API of Figure 3 lets tasks report a custom load metric.

2.3 Clerk Interface

The Clerk provides a single function which maps a key to the addresses of its assigned tasks (Figure 4). Most applications ignore this API and simply enable transparent integration with Google's RPC system Stubby or Google's HTTP proxy GFE (Google Front End).

```
interface Clerk {
  Set<Addr> getAssignedTasks(String key);
}
```

Figure 4: Slicer Client API

Stubby typically directs RPCs round-robin from each client to a subset of tasks in a job. We extended Stubby to accept an additional *slice key* argument with each RPC, causing the task to be selected using Slicer's assignment. Stubby also has support for Google's global load balancer, which selects the network-closest datacenter for each RPC. With both enabled, the global load balancer picks a datacenter, and Slicer picks the task from the job in that datacenter.

The GFE is an HTTP proxy that accepts requests from the Internet and routes each to an internal task. The GFE offers a declarative language for selecting routing features from a request's URL, parameters, cookies, headers and more. Slicer integration interprets any such feature as a slice key.

3 Slicer Uses in Production Systems

Slicer is used by more than 20 client services at Google, and it balances 2-7M requests per second with more than 100,000 application client processes and server tasks connected to it (Figure 1). Prospective customers evaluate their systems against a test instance of Slicer that routes another 2 Mreq/s.

This section illustrates some of Slicer's use cases. Current uses of Slicer fit three categories: in-memory cache, in-memory store, and aggregation.

3.1 In-memory Cache Applications

Slicer is most commonly used for in-memory dynamic caches over storage state.

3.1.1 Flywheel

Flywheel is an optimizing HTTP proxy for mobile devices [11]. Flywheel tracks which websites have recently been unreachable, enabling an immediate response to a client that averts a timeout. Flywheel uses a set of "tracker" tasks as a repository of website reachability. In the original design, updates and requests were sent to a random tracker task. Because the semantics are forgiving, this worked but converged slowly. To hasten unreachability detection, Flywheel now uses Slicer with website server name as the key, so that updates and requests converge on a single task.

3.1.2 Other cache uses

Many other services use Slicer to manage caches.

1. *Meeting scheduler*: manages meetings and provides calendar functions. Includes a per-user cache for faster responses.
2. *Crawl manager*: crawls pages and extracts metadata. Retains last crawl time per URL to provide crawl rate-limiting.
3. *Fonts service*: serves fonts to various web and mobile applications. Caches font files and subsets of font files.
4. *Configuration sync service*: periodically checks end-to-end configurations for entities from multiple sources. Entity affinitization allows comparisons of configurations from multiple sources.
5. *Data analysis pipeline*: analyzes stored data and serves summary results. Caches query results per source.
6. *Job profiling*: caches metadata used for job profiling by job name.
7. *User Contacts Cache*: caches user's contacts information when fetched by a user's mobile or web application.
8. *User Metadata Cache*: caches user's metadata/preferences for a user in a video display application.
9. *Service Control*: caches aggregated metrics and logs for public APIs.

3.2 In-memory Store Applications

The in-memory caches in the previous section handle shard reassignment by discarding state, causing future requests to the moved keys to see a cache miss. In contrast, the tasks of an in-memory store load any missing data from an underlying store, and thus resharding events only affect latency; the stored data remains available.

3.2.1 Speech Recognition

As mentioned in Section 1, a speech recognition system uses Slicer to assign languages to tasks and route incoming requests to a task with the required model loaded. The speech team originally manually partitioned languages into task-sized sets and put each set in a separate job. This approach required peak provisioning, failing to multiplex resources to exploit diurnal shifts as populations wake and sleep. It was also operationally complex, incurring manual overhead to monitor, maintain, upgrade, and debug separately-configured jobs.

3.2.2 Cloud DNS

Google's Cloud DNS service, which hosts millions of domains owned by Google and its customers, uses Slicer to assign DNS records to tasks, allowing the tasks to quickly make purely local decisions using in-memory state. Furthermore, Slicer's key redundancy and load balancing support allows the service to respond to load changes in the key space. Since the application provides DNS semantics, Slicer's affinity mode is sufficient.

3.3 Aggregation Applications

Tasks receive requests for some key (e.g., customer id, pubsub topic) and they aggregate them into larger writes to a backing store. This reduces traffic on the underlying store: Event Pipeline 2 achieved a $\frac{4}{5}$ reduction. Slicer's asymmetric key replication is particularly effective for aggregation, spreading hot key traffic across many tasks. The tasks write concurrently and depend on key-granularity append semantics at the store to preserve correctness [14].

3.3.1 Event analysis

Two event analysis systems shard events by source id to build up a model. Without Slicer, these systems would have to read, modify and write the model on every event, since aggregating writes would incur frequent expensive optimistic concurrency control conflicts.

With Slicer, requests for a source id key are almost[1] always routed to the same task. Therefore, a task can afford to aggregate writes coarsely, since write conflicts are rare. It can also cache the last model state it wrote, skipping the read step of read-modify-write unless the backend store detects a conflict. In these systems, traffic per source varies by several orders of magnitude, making load balancing essential.

[1]These services use Slicer's affinity mode, which provides high availability at the cost of perfect consistency (§4.5), relying on the backend store's conflict detection for data consistency.

3.3.2 Client Push: Pubsub System for Mobile Devices

Client Push [3] is a pubsub system that allows mobile clients to subscribe to topics and receive all messages published on that topic. Tasks are sharded by topic; they write subscriptions to a table in which the slice key is the prefix of the storage key. Slicer affinitization improves efficiency by aggregating requests for a range of keys to the storage servers. Slicer's asymmetric replication spreads hot topics across many tasks, avoiding bottlenecks.

4 Slicer Service Implementation

Slicer aims to combine the high-quality, strongly consistent sharding decisions of a centralized system with the scalability, low latency, and fault tolerance associated with local decisions. This section describes how Slicer achieves the best of both worlds.

The Assigner is the core of Slicer's backend service. It collects health, task provisioning, and load signals. It uses its central view of those signals to produce a coherent assignment of work to tasks (§4.4) that is strongly consistent for applications that need it (§4.5).

Though the Slicer Service is conceptually centralized (Figure 2), the implementation is highly distributed (Figure 5). By combining client-side caching, *Distributors*, and *Backup Distributors* that provide a backstop against catastrophic failures, the backend service also achieves scalability (§4.2) and fault tolerance (§4.3) similar to a purely local service.

4.1 Assignment Generation

The Assigner generates assignments using a sharding algorithm described in Section 4.4. To enhance availability, we run the Assigner service in several Google datacenters around the world. Any Assigner may generate an assignment for any job in any datacenter.

Deploying multiple Assigners increases availability but admits the possibility of disagreement. Section 4.5 explains how subscribers can observe consistent assignments. But even for eventually consistent applications, the Assigners should converge, not thrash among competing decisions. To facilitate convergence, Assigners write decisions into optimistically-consistent storage. An Assigner reads the stored assignment, generates a new assignment, and assigns it a monotonic generation number. It writes the new assignment back to storage transactionally conditioned on overwriting the previously read value. If a concurrent write has occurred, the transac-

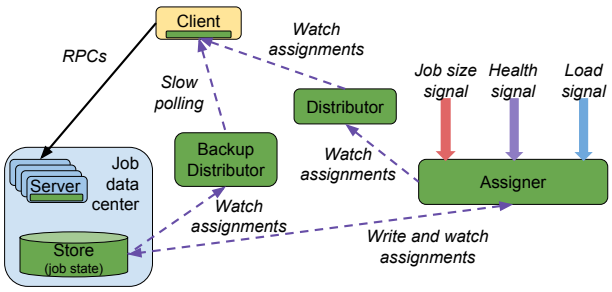

Figure 5: Slicer backend service architecture. The Assigner collects signals and uses them to make an assignment, informed by a stored prior assignment to minimize churn. The Assigner disseminates assignments to subscribers (Clerks and Slicelets) through the distributor and through a passive backup path via a store. All of this traffic is off the critical path of client-server communications. The Assigner and Distributors are replicated across datacenters; each component can serve any job at Google.

tion fails, the Assigner abandons its new assignment, retrieves the new current assignment, and tries again.

For efficiency, in the steady state only a single *preferred* Assigner generates an assignment for a particular job. Each Assigner periodically polls Google's global load balancer service to see if it is network-closest, and hence preferred, for the jobs for which it is generating assignments. This definition is eventually consistent: there may be brief periods when multiple Assigners are preferred.

Assignment storage makes the distributed Assigners act as a single logical process. When failure causes a change in preferred Assigner, the new one learns the decisions of the prior one and carries them forward. Should two Assigners both believe they are preferred, they will thrash, but storage concurrency control prevents divergence.

Slicer makes assignments for one job in one datacenter at a time. Customers who run jobs in multiple datacenters use a higher-level Google load balancer to route a request to a datacenter, and then within that datacenter, use Slicer to pick one task from the job.

4.2 Scalable Assignment Distribution

Because Slicer manipulates ranges of a hashed keyspace, assignments have a concise representation. Even then, large applications with thousands of tasks produce large assignments that need to be distributed to all server tasks and their clients (together, the *subscribers*). This distribution must occur quickly after assignment change. At large scales, distribution becomes a computational and network bottleneck. We address it with a two-tier distribution tree: an Assigner generates and distributes an assignment to a tier of *Distributors*, which distribute it to the subscribers. Nothing in our model precludes adding an additional tier to the tree.

Distribution is a pull model: a subscriber asks a Distributor for a job's assignment; if the Distributor doesn't have it, the distrubutor asks the Assigner, which generates and distributes the assignment. Each Clerk and Slicelet library maintains a long-lived stream with the Distributor service using Google's standard load balancer service, which routes its stream to the closest available instance.

Assignment distribution is asynchronous. Affinity applications can tolerate temporary inconsistency, and consistent applications ensure consistency via a separate control channel (§4.5).

This architecture admits running Distributors in datacenters close to subscribers to minimize WAN traffic. In practice, to ease administration, we currently tolerate the WAN traffic and run Distributors in the same datacenters as Assigners.

Evolving Slicer is easier if we decouple our release schedule from those of our customers. One design alternative we rejected was to have Slicer's subscriber library coordinate peer-to-peer assignment distribution among customer tasks. The cost is that the Slicer team must provision its own resources for assignment distribution, but the benefit is to minimize logic linked into customer binaries. Likewise, putting the logic that identifies the preferred Assigner in the Distributor tier keeps it out of subscriber libraries.

4.3 Fault Tolerance

We've designed Slicer to maintain request routing despite failures of infrastructure and of Slicer itself. Slicer's control-plane separation ensures that most failures merely hinder timely re-optimization of the assignment, yet requests continue to flow. The rest of this section enumerates properties of the system which achieve these goals.

Backup Assignment Retrieval Path. When an application client or server task starts, it must fetch the current assignment through the network of Distributors. The Distributors share a nontrivial code base and thus risk a correlated failure due to a code or configuration error. We have yet to experience such a correlated failure, but our paranoia and institutional wisdom motivated us to guard against it.

Hence the Slicer Service includes a *Backup Distributor* which satisfies application requests simply by reading the assignment from the store (§4.1). The Backup

Distributor is simple, slowly evolving, and mostly independent of the Distributor and Assigner code base.

If the Backup Distributor is the only one operating, the system degrades to static sharding based on slightly stale load and health information. This mode requires only:

1. Library code linked into application binaries,
2. the Backup Distributor service, and
3. a valid assignment in persistent storage.

Because it does not react to load shifts or server task failure, degraded mode is intended as a stopgap until an oncall engineer restores the Assigner and Distributor network.

Geographic Diversity. Distributors and Assigners run in datacenters around the world. Any subscriber can reach any Distributor via the Google global load balancing service, and likewise any Distributor can reach any Assigner. If the preferred Assigner for a job has failed, any Assigner can become preferred. This diversity tolerates machine, datacenter, and network failures.

Geographic Proximity. The preferred Assigner for each job is the Assigner network-closest to the job (§4.1), and a Distributor runs wherever there is an Assigner; these decisions reduce dependence on WAN connectivity. If customers demanded it, Slicer Service could run in every customer cell, eliminating all cross-datacenter dependency.

Fate-Shared Storage Placement. Although no production customers are configured this way, Slicer's implementation allows storing assignments in the same datacenter as the job. By also placing an Assigner in the same datacenter, the job can tolerate a network partition of the datacenter.

Service-Independent Mode. Ultimately, even if every component of the Slicer Service fails, requests continue to flow using the most recent assignment cached in applicaion libraries. This mode has the same limitations as the Backup Distributor mode, plus new or restarted application client tasks are unable to initialize.

In summary, Slicer's design tolerates machine, datacenter, and network failures including complete datacenter partitions. It degrades gracefully under correlated bug and configuration faults that destroy the Assigners, Distributors, or the entire Slicer Service.

4.4 Load Balancing

The ultimate goal of load balancing is to minimize peak load; this enables a service to be provisioned with fewer resources. We balance load because we do not know the future: unexpected surges of traffic arrive at abritary tasks. Maintaining the system in a balanced state maximizes the buffer between current load and capacity for each task, buying the system time to observe and react.

Slicer's initial assignment divides the keyspace equally among available tasks, assuming that key load is uniform (key distribution is uniform due to hashing). If there is variation in either the rate at which different keys receive requests or in the resources required to satisfy those requests, some tasks may become overloaded while others are underutilized. Slicer monitors key load – either request rate, which can be automatically tracked via the Slicelet integration with Stubby, or application-reported custom metrics – to determine if load balancing changes are required. The primary goal of load balancing is to minimize the *load imbalance*, which we define as the ratio of the maximum task load to the mean task load. In a perfectly balanced job where each task is handling the same load, the imbalance is 1.

To provide intuition for the definition: the worst case imbalance Slicer can cause is n/r, where r is the job's minimum key redundancy configuration and n is the task count. For example, with $n = 10$ and $r = 2$, the worst decision Slicer can make is to direct Stubby to route every key to one of two tasks, giving a load imbalance value of 5.

Load imbalance can be reduced by adding or removing redundant tasks for a key or by reassigning keys from one task to another. Besides reducing imbalance, Slicer must respect configurations constraining the minimum and maximum number of tasks that may be assigned to a key. It should also limit *key churn*, the fraction of the key space affected by reassignment. Key churn itself creates load and increases overhead.

To scale to billions of keys, Slicer represents assignments compactly with key ranges. Hence sometimes it must *split* a hot slice—replace a key range $[a,c)$ with two ranges $[a,b),[b,c)$—so that its load can be distributed among multiple tasks. To prevent unbounded assignment size growth, Slicer must also create opportunities to *merge* slices. It does so by assigning adjacent cool slices to the same tasks, then merging the slice representations into a single range.

At Google, independent mechanisms (sometimes humans) decide when to add or remove tasks from a job, or add or remove CPU or memory from tasks in a job. Thus Slicer focuses exclusively on redistributing imbalanced load among available tasks, not on reprovisioning resources for sustained load changes.

4.4.1 Sharding Algorithm: Weighted-move

When Slicer determines that resharding is necessary, due to changing load metrics or changes to the set of tasks in

the job, it produces a new assignment using the sharding algorithm, which proceeds in the following phases:

1. Reassign keys away from tasks that are no longer part of the job (e.g., due to hardware failure).
2. Increase/decrease key redundancy as required to conform to configured constraints (e.g. due to a change in the configuration).
3. *Merge* adjacent cold slices, moving one onto the same task as the other, to defragment the assignment. This step proceeds as long as
   (a) there are more than 50 slices per task in aggregate,
   (b) merging two slices creates a slice with less than mean slice load,
   (c) merging two slices does not drive the receiving task's load above the maximum task load, and
   (d) no more than 1% of the keyspace has moved.
4. In this phase, the sharding algorithm picks a sequence of *moves* with the highest *weight*, which we define as the reduction in load imbalance for the tasks affected by the move (benefit) divided by the *key churn* (cost). Moves are applied to the assignment in descending weight order until a key churn budget (9% of the keyspace) is exhausted.
5. *Split* hot slices without changing their task assignments. Splitting captures finer-grained load measurements and opens new move options in the next round. This step proceeds as long as
   (a) the split slice is at least twice as hot as the mean slice, and
   (b) there are fewer than 150 slices per task in aggregate.

In each iteration of phase 4, only moves affecting the hottest task can reduce load imbalance (as defined above), and for each slice in the hottest task, three possible moves are considered: reassigning the slice to the coldest task to displace the load, redundantly assigning the slice to the coldest task to spread the load, or removing the slice which offsets the load to existing assignees. Note that increasing or decreasing assignment redundancy may be illegal given the configuration for the job, so some moves are disqualified. The algorithm greedily makes the best move and repeats until the key churn (cost) budget is exhausted. Successive iterations of the loop may affect different tasks as prior moves revise the estimate of which task is "hottest".

The constants in the algorithm (50–150 slices per task, 1% and 9% key movement per adjustment) were chosen by observing existing applications. Experience suggests the system is not very sensitive to these values, but we have not measured sensitivity rigorously. Future work will estimate application-specific churn cost to better tune the cost-benefit tradeoff.

4.4.2 Rebalancing suppression

Slicer balances request rate, task CPU utilization, or an application-specified custom metric. When balancing CPU and the maximum task load is less than 25% (an arbitrary threshold), Slicer suppresses rebalancing: Because no task is at risk of overload, churn is waste.

4.4.3 Limitations

When balancing the request rate, Slicer ignores task heterogeneity: one task may be cool with 10,000 req/s but another is swamped. CPU utilization balancing inherently adjusts for such heterogeneity.

Some applications make high memory demands for each key. If Slicer colocates many infrequently requested keys on one task, that task may exhaust memory despite manageable CPU load. Our future work will include measuring memory usage and honoring constraints in the algorithm.

4.4.4 A rejected design alternative

A variant of consistent hashing [22] with load balancing support [10] yielded both unsatisfactory load balancing and large, fragmented assignments. We refer to this scheme as *load-aware consistent hashing*. Some applications had too few slice keys (tens to hundreds per task) for consistent hashing to result in good statistical load balancing.

Consistent hashing enables very compact assignments, so long as the client carries the decoding algorithm. Since evolving clients is burdensome (§4.2), Slicer instead distributes assignments in decoded form. Consistent hashing works best with many (1000) virtual nodes per physical task but introduces a significant cost distributing decoded assignments.

More importantly, consistent hashing gives us less control over hot spots. We can cool off a task by reducing its virtual node count, but the displaced traffic ends up randomly distributed, not directed at a cool task, giving a poor tradeoff between key movement and balance improvement.

We were originally drawn to the statelessness of consistent hashing: it produces the same output from the same inputs, which allowed recovering from an Assigner failure without requiring access to the previous assignment. In practice, once the Assigner begins balancing load, creating a profitable reassignment requires knowl-

edge of the previous assignment, and thus it is important that a recovering Assigner have access to prior state.

The load-aware consistent hashing algorithm we abandoned is similar to that in Centrifuge [10]. It was more sophisticated in that it supported key replication, asymmetric replication, and proportional response to imbalance for faster reaction. After 18 months in service, we replaced it with the weighted-move algorithm, which balances better with less key churn (§5.2.1).

4.5 Strong Consistency

An application that needs to maintain data consistency can do so by building upon Slicer's optional *assignment consistency*. It defines an authoritative assignment for every moment and guarantees that no task ever believes a key is assigned to it if that assignment does not agree. By configuring the job for at most one replica of each key, at no time will two Slicelets believe they are both assigned the same key. The consistency feature is implemented, but it is not yet deployed by customers in production.

The simplest way to provide strong consistency guarantees for keys would be to allocate a lease for each key from a central lease manager. We opted against this model, because it would require provisioning lease manager resources in proportion to the number of keys, and hundreds of millions of keys per sharded job are common. Existing lease managers such as Chubby [12] do not scale to that level, so this would require building a highly available, scalable lease manager and running it in every datacenter at Google, which is a non-trivial effort.

While insufficiently scalable to provide a lease per key, Chubby is highly available (the code is battle-tested, and the system has its own operations team) and present in every data center at Google. Slicer builds on Chubby to provide a scalable lease-per-key abstraction using only three Chubby locks per job. The scheme ensures that only the keys being reassigned are unavailable during an assignment change. The design preserves the robustness of Slicer's data plane, so that even if Slicer Service is down, RPCs continue to flow with strong consistency, since lease granting and maintenance is performed by the highly-available and battle-tested Chubby. The Assigner is only required for resharding.

The following describes how the leases provide strong consistency.

To protect the work done while changing a strongly-consistent assignment, an Assigner acquires the exclusive *job lease* to ensure that exactly one Assigner performs the work for writing. If the Assigner crashes during the assignment-change operation, another Assigner can acquire the job lease and resume the unfinished work. Only Assigners interact with the job lease.

To achieve consistent assignment, the Assigner distributes assignments in the usual way, then writes the assignment generation number as the value of the *guard lease*. A consistent Slicelet may only use an assignment once it acquires the guard lease for reading. Clerks require no lease, since the only harm of a transient inconsistent assignment at the Clerk is a misrouted request bounced back for retry.

Changing the assignment entails recalling the guard lease from Slicelet readers so the Assigner can rewrite its value. In any large-scale system, recalling a lease often means waiting out the expiration period for any task that may have died while holding its lease. This recall period entails complete application unavailability.

We make the observation that when an assignment A_1 is replaced by A_2, there is no reason to make unavailable the unchanged slices, those that have identical assignments in $A_1 \cap A_2$. A third *bridge* lease bridges over the transition from A_1 to A_2, making $A_1 \cap A_2$ available during the gap. The Assigner writes and distributes assignment A_2, creates the bridge lease, delays for Slicelets to acquire the bridge lease for reading, and only then does it recall and rewrite the guard lease. A Slicelet is allowed to use the intersection if it holds the bridge lease.

For a synthetic benchmark, we measured a median lease recall period of 2.6 s and 99th percentile period of 4.1 s, implying that absent a bridge lease an entire application would suffer seconds of unavailability whenever an assignment changes. Section 5.2.5 reports on a benchmark that demonstrates how the bridge lease improves availability.

Nothing about the consistent-assignment mechanism limits it to the simple consistency propery of at most one Slicelet per key; the Assigner could easily enforce an at-most-three policy. The simpler policy is easy for applications to exploit, whereas allowing plural replicas would require the application to consistently coordinate those replicas, perhaps with state machine replication [24].

5 Evaluation

This section evaluates Slicer using both measurements from the deployed system and experiments with real and synthetic workloads.

5.1 Production Measurements

We measure production customers to evaluate Slicer's availability, load balancing, scale, and assignment convergence time.

5.1.1 Availability

As the primary – but pessimistic – measure of production availability, we evaluated the integration of Slicer

and Stubby. Specifically, we considered how often Slicer was able to select a task for a Stubby client issuing an RPC. Normally, Stubby selects any task in the destination job which the client locally believes to be healthy. With Slicer, Stubby selects a healthy task from the set of Slicer-provided candidates. If all tasks are unhealthy or no assignment is available, the selection fails.

Over a one-week period, Slicer performed 260 billion task selections for a subset of its Stubby clients, of which 99.98% succeeded. This value underestimates the availability of Slicer, because some of the failures may have been because all tasks were unhealthy, and ordinary Stubby would also have failed to select a task, but we expect that such cases are rare. Thus, in those cases where standard Stubby could have sent an RPC, then Stubby with Slicer could have sent an RPC at least 99.98% of the time.

We also examine availability at the server side. In another week, we observed 272 billion requests arrive at server tasks, of which only 11.6 million (0.004%) had been misrouted. This measure overestimates availability because it only considers requests that made it to a server task, and it underestimates availability because many applications can tolerate misdirected requests with only an impact on latency or overhead, not availability.

A secondary measure of availability is that of the Slicer service itself. Our production monitoring periodically requests an assignment from each Distributor instance. In one week, 99.75% of 329,978 requests succeeded. This probe underestimates availability because it requires computation of a new assignment, whereas the common path returns a cached one.

These measurements are over an admittedly short window, limited by production monitoring data retention policy. That said, they indicate Slicer is a suitable building block for highly available applications.

5.1.2 Load balancing

We evaluate how well Slicer balances load across tasks, how much key movement it incurs, and how much it improves over static strategies.

Figure 6 shows the effectiveness of load balancing for several production customer jobs belonging to three services. Sampling five minute windows over a six hour period, we measure the number of requests each task handles, normalized as a fraction of the mean request count for all tasks in the job during the window. The vast majority of time windows had values close to the mean, indicating that the tasks were well-balanced. Peak loads varied between $1.3\times - 2.8\times$ the mean load.

Figure 7 shows key churn for tasks in the same jobs as in Figure 6. Churn counts the number of key-moves:

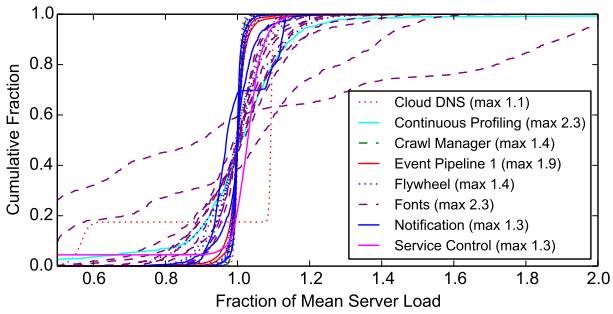

Figure 6: Slicer successfully balances load: tasks in a job rarely experience load 5% greater than the mean task load.

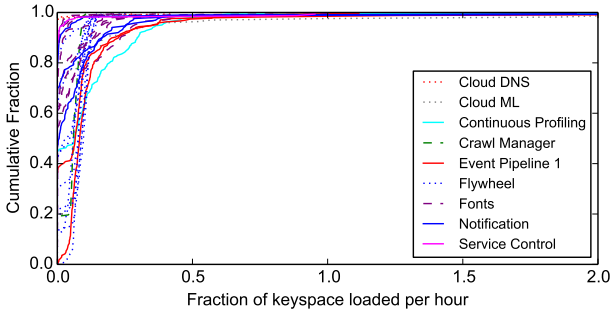

Figure 7: Key movement costs for jobs belonging to customer production services, sampled over one week. The median hour in every job sees less than 20% of the keyspace move.

one key moving ten times in one hour produces the same value as ten keys moving once. Here we see a broader range of values, as some jobs exhibit higher variance over time (e.g., Cloud DNS, which moves up to 40% of its keys per hour), and some are quite stable over time (e.g., Flywheel, which moves only 16% of its keys). We report fraction of keyspace but not bytes of objects actually unloaded and reloaded because, by design, Slicer does not know which keys in the key space actually exist, nor is it aware of the data associated with those keys (§2.1).

Our production monitoring captured a shift from load-aware consistent hashing to the weighted move algorithm. Figure 8 shows the request rate per task for the general-purpose key-value cache discussed in Section 3.1 during the rollout of the weighted-move algorithm. Under consistent hashing, the hottest task was 50% hotter than the mean. The weighted move algorithm improves the balance, enabling operations engineers to make tighter capacity planning decisions.

Ultimately, customers care about Slicer's load balancing because it offers a big win over home-brew alternatives. We observed production key distributions and load distributions for all customer jobs. We built a model to infer the load on the tasks had the load been balanced

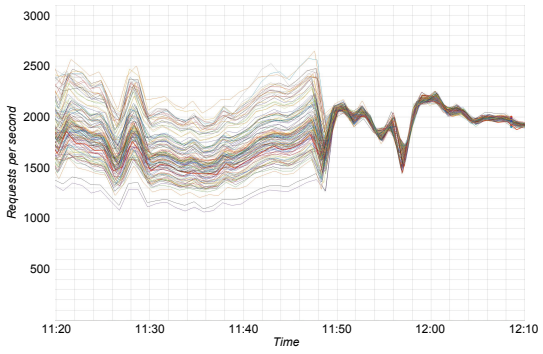

Figure 8: Load per task on a production key/value cache when switching from load-aware consistent hashing to the weighted-move algorithm at 11:50.

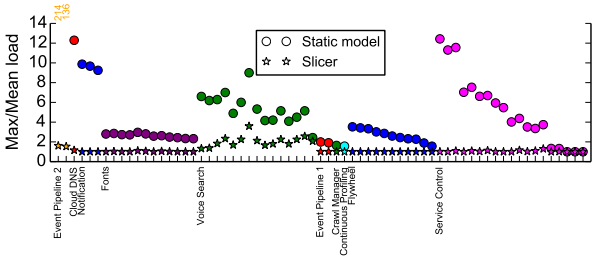

Figure 9: Load balance for production jobs grouped by service, contrasted with a static model. Slicer makes the median job's hottest task 63% less loaded.

statically. If the customer supplied an initial load estimate, the model uses it; otherwise it spreads the keyspace uniformly across tasks. The model mitigates random clumping by partitioning the keyspace into 100 slices per task.

Figure 9 contrasts, for each job, the actual load imbalance under Slicer versus the load imbalance under the static model. Load imbalance is the ratio between the CPU load of the most loaded task and the mean CPU load across tasks. Each pair of points shows the most imbalanced hour in a one-week observation. For underloaded jobs, Slicer defers load balancing, and thus acts identically to the static model; Figure 9 elides such jobs. Service operators provision for peak loads; Slicer provides a median reduction of 63% and as much as 99.3% for the most skewed job.

5.1.3 Scale

Slicer serves more than 20 unique systems (§3). Each is a unique software stack that integrates Slicer in a different way. This table extracts aggregate statistics from production monitoring.

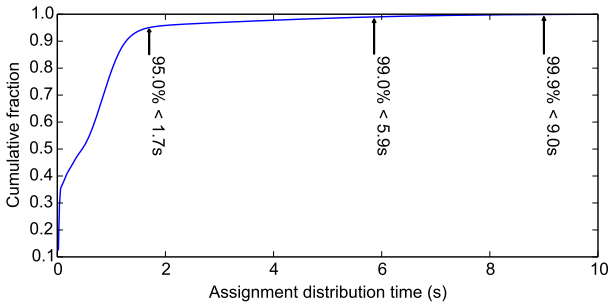

Figure 10: Once emitted by the Assigner, 95% of assignments reach subscribers within 2 s.

Services Jobs	22 263	mean / service	mean / job
Tasks (Slicelets)	11387	517	43
Clerks	113,338	5151	430
Requests/sec	6M	266K	22K
Assignments/hour	662	30	2.5
Assignment traffic (MBps)	180	8.2	0.37
Key churn/hour	4%		

Presently the production Slicer Service includes six Assigners provisioned with three cores each. Sampling one minute windows on each task over one week, the median sample utilizes 0.13 core, and the 99th percentile utilizes 2.34 cores. Considering the entire Service—Assigners, Distributors, Backup Distributors—Slicer uses 0.3% of the CPU and 0.2% of the RAM used by the sliced services and their clients.

5.1.4 Assignment Convergence Time

It is desirable for Slicer to effect assignment changes rapidly, to minimize the period of divergence among subscribers. Figure 10 shows the CDF of assignment distribution latencies across affinity-mode production customers for one week. Assignments generally arrive within the second.

5.1.5 Assignment Computation Time

Most production assignments take a fraction of a second to compute; the 64th percentile is 17ms and the maximum a few seconds.

5.2 Experiments

Experiments in this section explore details and trade-offs under controlled conditions.

5.2.1 Comparing load balancing strategies

We recorded slice keys for RPCs issued to three production users of Slicer: Client Push (see Section 3.3.2), Cloud DNS (see Section 3.2.2) and Flywheel (see Section 3.1.1). We then replayed these requests against three algorithms: static uniform sharding (in which the

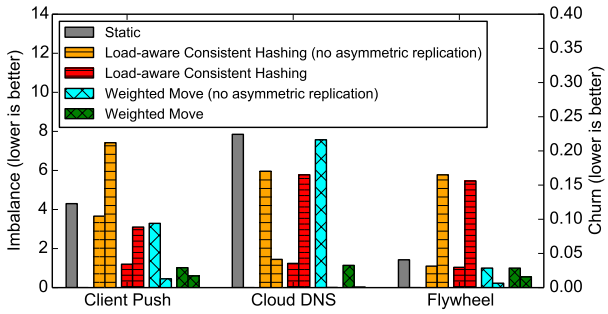

Figure 11: Slicer's centralized weighted move algorithm balances better than static and load-aware consistent hashing schemes, and churns less than load-aware consistent hashing.

key space is divided uniformly amongst all tasks), load-aware consistent hashing (see Section 4.4.4) and Slicer's weighted-move algorithm (see Section 4.4.1). In addition, we compared the performance of the algorithms with and without asymmetric key redundancy (not applicable for static sharding which cannot dynamically assign keys to additional tasks).

Figure 11 shows the mean across all resharding decisions for measurements of *load imbalance*, the ratio of the max task load to the mean task load, and of *key churn*, the fraction of the key space reassigned (both defined in §4.4). Slicer's algorithm – weighted-move with redundancy – significantly outperforms both other algorithms on load imbalance, with reduced key churn relative to consistent hashing (though not static sharding, which being static has no key churn). Asymmetric replication provides significant load balancing benefits, though with a small increase in key churn (due to increased opportunities to address imbalance).

Note that this experiment isolates the impact of load balancing from other factors such as task failures and pre-emption.

5.2.2 Assigner Failure and Recovery

To evaluate Slicer's robustness to Assigner failure, we presented power-law skewed load to twenty tasks. Once the system stabilized, we killed the Assigner task, causing clients and server tasks to continue using the last-generated assignment. After 2 hours, we restored the Assigner.

Results are shown in Figure 12. The pre-failure and post-recovery curves are essentially identical: the Assigner rebalanced load upon recovery. The outage curve shows degraded load balancing, since the assignment stagnated while the load changed. However, the recent static balance is better than uniform sharding (not shown in Figure 12) on the same workload. Production workloads tend to be more stable over time; the outage curve

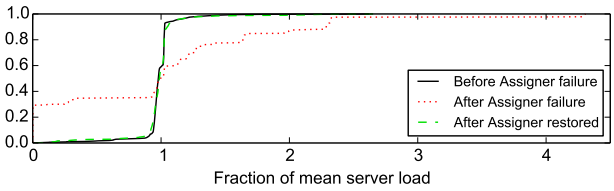

Figure 12: Load balancing before, during, and after an Assigner failure.

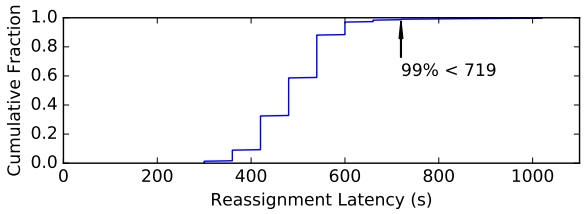

Figure 13: The Assigner typically effects a response to a load shift in 480 seconds.

for such workloads should remain closer to the actively-balanced curves.

In practice, if an Assigner fails, any other Assigner can pick up the slack. We configured a test job with two Assigners, killed the active one, and observed that the other became initialized 17.1 s later ($\sigma = 2.7$ *s*). This delay is the period of polls to the Google load balancer for preferred Assigner checks (§4.1).

5.2.3 Load Reaction Time

How quickly does Slicer respond to a load shift? In this experiment, five client tasks offer 8 Kreq/s of synthetic load to ten server tasks, consisting of 100 keys in a power-law distribution with exponent 1.5. Every nineteen minutes, the clients' distribution shifts to move the hottest load to different keys. We report the latency from clients shifting load to tasks reporting a max/mean load imbalance below 1.2. Figure 13 shows a median delay of 480 s, which is a function of the 1 m delay from the Google monitoring system and Slicer's 5 m load observation window. One window is insufficient because, unless the load shifts very early in the window, Slicer's first observation doesn't convince it to shift enough load to completely restore balance.

5.2.4 Scaling Benchmark

One of Slicer's essential architectural decisions is central decisionmaking and a distributed data plane. In the experiment in Figure 14, we contrast Slicer's plumbing with a natural alternative that indirects routing decisions to a centralized *authority*. In the centralized version, clients preface each request with a request to the authority, and server tasks contact the authority on each request to confirm the routing decision. Here the authority is implemented as a single Clerk task relaying

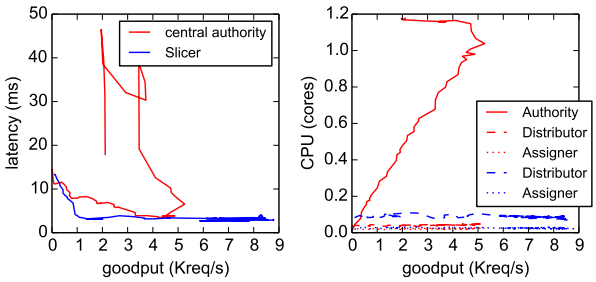

Figure 14: On the left is a latency-throughput curve, on the right CPU consumption versus retired load. Once the central authority saturates its CPU allocaton at 5 Kreq/s, it encounters a scaling limit. This simple experiment lacks admission control, so the throughput drops under overload; a production system would hit the same wall more gracefully.

decisions from an Assigner and Distributor, although a real centralized system would simply colocate load balancing with the authority interface. In both cases, a set of 2000 clients simulated on 100 tasks offers increasing load against 50 server tasks. The authority saturates its CPU at 5 Kreq/s, but Slicer scales smoothly since every component's workload is independent of aggregate client request rate.

5.2.5 Consistency Benchmark

Section 4.5 described how Slicer preserves availability in consistent assignment mode by using bridge leases to carry unchanged key assignments across the distribution period of a new assignment. We evaluate its importance under a synthetic dynamically skewed workload in which 25 clients drive 43 Kreq/s against 50 server tasks. Over three days, 99.85% of requests were satisfied; absent bridging, only 99.19% of requests would have been satisfied.

The `getSliceKeyHandle` operation takes 153 μs and `isAssignedContinuously` takes 94 μs.

6 Related Work

As a general purpose sharding system, Slicer is similar to Centrifuge [10], Orleans [13], Ringpop [8], and Microsoft Service Fabric [5].

It is most similar to Centrifuge, which also uses a central manager, assigns ranges of a hashed keyspace, and provides leases. Slicer differs in four respects. First, Slicer's architecture is more available. If the Centrifuge manager is unavailable, all leases expire and no RPCs flow. Slicer's control-plane separation ensures that assignments remain valid and RPCs flow even if the entire Slicer service fails. Slicer's separate backup distribution path keeps working even when the service is down. Second, Slicer's separation of assignment distiribution from assignment generation enables much higher scales: an Assigner can servie 10^4 Distributors, and a Distributor 10^4 subscribers. Third, Centrifuge is a single-cluster system. Slicer's Assigner can be accessed from a different cluster, enabling failover across clusters. Fourth, Slicer's load balancing is better than Centrifuge's. Slicer moved from Centrifuge-style consistent hashing (S4.4.4) to the weighted-move algorithm (§4.4). It achieves better balance while moving an order-of-magnitude fewer keys (§5), works for both many and few application keys, and creates more compact assignments. Slicer's load balancing supports custom metrics and key redundancy.

As compared to Orleans and Ringpop, Slicer uses a centralized algorithm rather than a client-based consistent hashing [22], which allows it to provide better load balancing and to offer consistency guarantees, which those systems cannot. Service Fabric does not support dynamic sharding: sharding must be specified by the application and cannot be adjusted on the fly to balance load [6]. Additionally, both Orleans and Service Fabric are frameworks and are more invasive to applications than Slicer's small API.

As a sharding manager, Slicer also has elements in common with sharding managers embedded in storage systems. For example, Bigtable [14], HBase [2], and Spanner [15] are all structured in terms of ranges of an application-defined keyspace. Only the HBase algorithm is publically described; it has several strategies, all of which have splits and moves as base operations. Unlike Slicer, it does not support key redundancy or balancing on application-defined metrics. Moreover, storage system sharding managers are not usable outside of the storage system, and they often make storage-specific assumptions that limit their flexibility. For example, Bigtable requires at most one task per tablet to enforce consistency, whereas Slicer is free to add redundant copies of keys if permitted by the application

Social Hash [28] makes cluster-level sharding decisions for HTTP requests and storage systems. Slicer shares Social Hash's separation of coordinated central decisionmaking from distributed forwarding. Where Slicer treats keys independently, Social Hash optimizes placement using inter-key locality available in social graphs. Slicer operates at fine granularity in space (tasks) and in time (seconds to minutes). Slicer supports a wide variety of applications and supports consistent assignment.

BASIL [19] and Kunkle [23] balance I/O workloads in large-scale storage systems; like Slicer, they perform what-if planning and evaluate migrating hot data. They differ from Slicer in several important respects. First,

they place a relatively small number of items, and they have application-specific load data for each item. For example, BASIL places virtual disks within a storage array. This is a different problem than in Slicer, which places a potentially vast number of items (e.g., hundreds of millions), is agnostic to the application, and can only collect information at coarse granularity. Second, Slicer has a larger space of possible load balancing moves available; in addition to migrating slices, it can also split and merge them, and it can add or remove redundant copies. Besides minimizing imbalance, Slicer's algorithm also minimizes assignment fragmentation.

In theory, sufficiently fast storage available to all front-ends can sometimes obviate the need to cache sharded data in the front-end. Caches such as Memcached [4] and Redis [7] as well as in-memory stores such as RamCloud [25] and Dynamo [16] can be used. However, remote storage always adds the cost of (un)marshalling data along with a network roundtrip to access data. In addition, such solutions do not help when the shared resource isn't state, such as a network socket. Finally, eliminating external caches and collocating data with code reduces how many services must be provisioned and maintained.

Sharding solutions have been recently proposed [29, 27, 20] for specific databases, focusing on dynamic load balancing (as opposed to balancing the number of keys per task). Accordion [27] places partitions but does not modify their boundaries and thus cannot handle hot data. SPORE [20] replicates hot keys but does not support dynamic task membership or key migration. EStore [29] is a dynamic sharding manager that like SPORE identifies hot keys and migrates them, but it does does not support key redundancy. When hot keys cool down, EStore migrates previously hot keys back to their original shards, which creates unnecessary churn.

Software and hardware network load balancers [17, 26, 18, 21, 1] employ one or more controllers that either process messages themselves or program a set of distributed switches to carry out a load balancing policy. Such load balancers may have a notion of affinity or session "stickiness". However, such balancers implement static hashing for requests or sessions; when they react to load shifts, they do not maximize affinity. The do not provide server tasks with early assigment signals to facilitate prefetching, or termination signals to facilitate garbage collection. They do not offer asymmetric key redundancy, nor do they enable assignment consistency.

7 Conclusions

Slicer is a highly available, low-latency, scalable and adaptive sharding service that remains decoupled from customer binaries and offers optional assignment consistency. These features and the consequent architecture were driven by the needs of real applications at Google. Slicer makes it easy to exploit sharding affinity and has proven to offer a diversity of benefits, such as object caching, write aggregation, and socket aggregation, to dozens of deployed applications.

Production deployment of Slicer shows that the system meets its load balancing and availability goals. Real applications experience a max:mean load ratio of 1.3–2.8, assisting peak load capacity planning. Slicer balances load better than load-aware consistent hashing, and does so while creating an order of magnitude less key churn. Slicer is available, correctly routing production customer requests at least 99.98% of the time, making it a building block for highly-available applications. Adoption by over 20 projects with a variety of use cases demonstrates the generality of its API.

References

[1] Amazon ELB. https://aws.amazon.com/elasticloadbalancing/.

[2] Apache HBase. https://hbase.apache.org/.

[3] Firebase topic messaging. https://firebase.google.com/docs/cloud-messaging/android/topic-messaging.

[4] Memcached. https://memcached.org/.

[5] Microsoft service fabric. https://azure.microsoft.com/en-us/documentation/services/service-fabric/.

[6] Partitioning in microsoft service fabric. https://azure.microsoft.com/en-us/documentation/articles/service-fabric-concepts-partitioning/.

[7] Redis. http://redis.io/.

[8] Uber ringpop. https://eng.uber.com/intro-to-ringpop/.

[9] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A client notification service for internet-scale applications. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 129–142, 2011.

[10] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 1–1. USENIX Association, 2010.

[11] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's data compression proxy for the mobile web. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 367–380, 2015.

[12] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[13] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *ACM SOCC*, 2011.

[14] 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):4, 2008.

[15] J. Corbett et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013.

[16] 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. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[17] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[18] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2015.

[19] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated io load balancing across storage devices. In *File and Storage Technologies (FAST)*, 2010.

[20] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM.

[21] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2015.

[22] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.

[23] D. Kunkle and J. Schindler. A load balancing framework for clustered storage systems. In *High Performance Computing-HiPC 2008*, pages 57–72. Springer, 2008.

[24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[26] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[27] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.

[28] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kllapi, and M. Stumm. Social Hash: An assignment framework for optimizing distributed systems operations on social networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 455–468, Santa Clara, CA, Mar. 2016. USENIX Association.

[29] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.