Slicer: Sharding Beyond Storage for Distributed Datacenter Services

Jon Howell
April 2018
Local Memory Considered Helpful

• Server machines have a lot of memory applications should exploit it!

• Datacenter applications often don’t cache data because it's hard

• Slicer makes it easy to build services that use local memory
Talk Outline

• Why stateful servers are difficult
• Slicer model and architecture
• Evaluation
• Raising the level of abstraction
Building a DNS Service

End-user devices

DNS Service

Virtual Machines

Cloud Platform
Full State Replicated on Every Server

- Any server can handle any request
- Easy adaptation to failures, capacity changes, load skews
- Hard to scale or handle mutations

End-user devices

Frontends

DNS Servers/Job
Stateless: Interchangeable Servers + Database

- Any server can handle a request
- Cannot query DB for every DNS request
  - High latency
  - Network hop and marshaling costs
Stateless with External Cache

- Any server can handle a request
- Latency is better than pure stateless but
  - Now we have a consistency problem!
  - Tail latency can be high
  - Network hop and marshaling costs

External Cache, e.g., Memcache
Stateful: Static Sharding

- Simple mapping from keys to servers via static function
- Failure adaptation: Black-hole traffic for crashed server
- Capacity adaptation: Could result in significant key churn

Frontends

DNS Servers/Job

Hash(key) mod 4
Stateful: Consistent Hashing

- Implement server presence detection
- Addresses capacity and failure adaptation, key churn
  - Stochastic load balancing is inadequate
  - Distributed decisions harm affinity

```
ConsistentHash(key) 

Frontends

DNS Servers/Job
```
Stateful: Central Controller

Central server: presence detection, load monitoring, consistent view

😊 Fan-out assignments to large number of clients and servers

😊 Internals of a sharded distributed storage system!

Should we use stateless servers? 😞
Slicer: Refactored System for Sharded Apps

• Auto-sharding without storage coupling

• Assignment “control plane” vs forwarding “data plane”
  • Narrow interface
  • Scalable, consistent & fault-tolerant

• Reshards for capacity and failure adaptation, load balancing

• Production evaluation
What applications need affinity?

Caching: serving reads
  • Cloud DNS
  • Users' contact list
  • any service that uses Memcache

Batching: gathering writes
  • Pub-sub messaging [Thialfi, SOSP ’11]

Aggregation: other scarce resources
  • Courteous web crawler
Talk Outline

• Why stateful servers are difficult

• Slicer model and architecture

• Evaluation

• Raising the level of abstraction
Slicer Hash Sharding Model

Hash keys into 63-bit space
Assign ranges ("slices") of space to servers
Split/Merge/Migrate slices for load balancing
"Asymmetric replication": more copies for hot slices
• Assign at most one key to a task
• Multiple tasks for a key based on load: English: 200 tasks, Swahili: 2 tasks
Slicer Architecture: Goals

• High-quality sharding and consistency of a centralized system

• Low latency and high availability of local decisions
Slicer Overview

Distributed data plane

Centralized control plane

Hash(key)

Hash(key)

Frontends

Clerk

Application servers

Slicelet

Slicer Service
Slicer Architecture

Frontends

Application servers

Existing Google Infrastructure

Clerk

Slicelet

Backup Distributor

Assigner

Distributor

Capacity Monitoring

Health Monitoring

Load Monitoring

Lease Manager
Tolerating Failures

Localized failures:
- machine failure
- datacenter fire

Correlated failures:
entire Assigner or Distributor service down
- Bad configuration push
- Software bug
- Bug in underlying dependencies
Tolerating Localized and Correlated Failures

Frontends
Application servers

Backup Distributor datacenters
Smaller/Simpler Components
More Complex Components
Assigner datacenters

Smaller/Simpler Components
More Complex Components

Distributor datacenters
Slicer API: Client

• Core API:
  TaskList GetAssignedTasks(string key);

• Integrated with Google’s RPC system
  • Create stub
    channel = slicer::CreateChannel(…);
    stub = MakeNewStub(channel);
  • Send RPC
    Rpc rpc; Request request(…);
    rpc->SetSliceKey(<request key>);
    rpc->SendMyRpc(request);
Slicer API: Server

• Check if key owned by server
  • Affinity
    boolean isAffinitizedKey(String key);
  • Strong Consistency
    Handle getSliceKeyHandle(String key);
    bool isAssignedContinuously(Handle handle);

• Provide load feedback
  InformRpcFinished(ServerContext* ctx);

• Changed slices upcall
  interface SliceletListener {
    void onChangedSlices(List<SliceChange> changes);
  }
Talk Outline

• Why stateful servers are difficult
• Slicer model and architecture
• Evaluation
• Raising the level of abstraction
Evaluation: Slicer Usage

- Slicer load balances hundreds of millions of requests per second
- 99.98% of clients requests had a valid assignment
- < 0.01% of these requests directed to the wrong server
Slicer Load Balancing vs. Static (No Load Balancing)

Static sharding has much greater skew
Evaluation: Load Balancing Effectiveness

Slicer allows tighter capacity allocation by reducing skew
The End

• Why stateful servers are difficult
• Slicer model and architecture
• Evaluation
• Raising the level of abstraction