Project Overview

- Our project: distributed message queue
- GitHub: https://github.com/chris-1187/scalability-demo

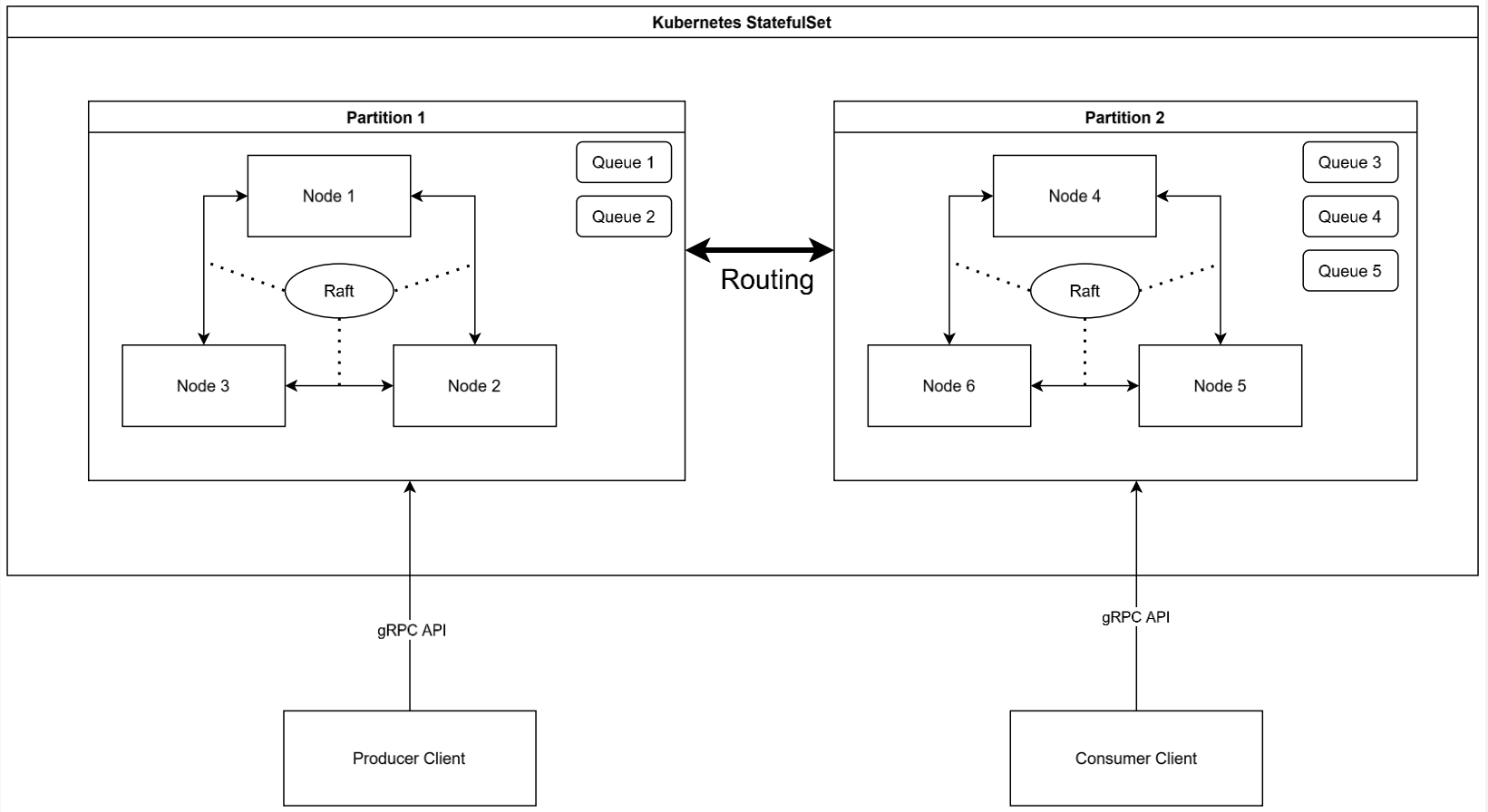
- Functional requirements
  - Manages multiple named queues
  - Allow to enqueue and dequeue messages
  - Queues operate in FIFO mode
  - At-least-once processing through client commits

- Non-functional requirements
  - Ability to scale to a very large number of queues
  - High availability; tolerate single node failures
  - Message durability

System Architecture

- Uses partitioning and replication
- Highly scalable and available
- Deployed via Kubernetes

- Core system + clients

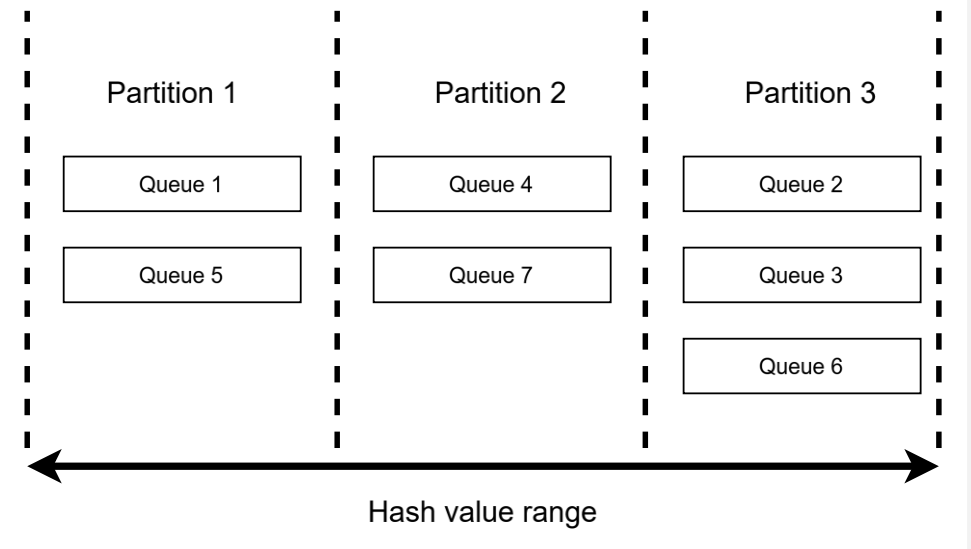


API

- gRPC API with 3 endpoints
  - Push
  - Peek
  - Pop
- API used for
  - Client operations
  - Communication among nodes
- Features
  - Idempotent insertion with client token
  - At least once delivery through separate pop operation

Partitioning

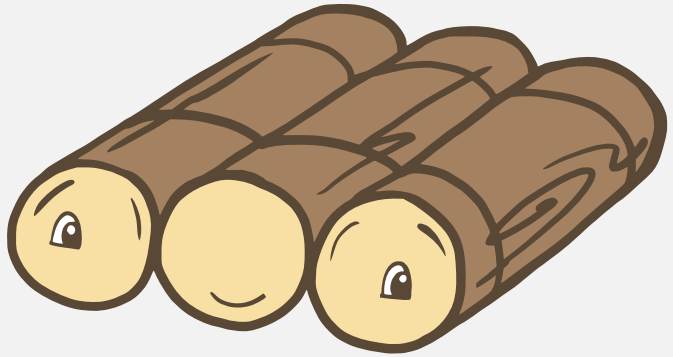
- Queue names are hashed
- Hash range is divided into equally-sized partitions
- Static partitioning
  - Nodes are assigned deterministically based on the launch config
  - Every node can calculate the entire topology
  - No dynamic scaling at runtime
- 1-hop routing
  - Client requests are load balanced onto any node
  - First contact node routes to correct partition



Replication

- Raft based log replication built on top of JRaft
  - Every push and pop operation creates a log entry
  - Entries are safely replicated across multiple nodes
  - High availability through quorum-based leader election

- Our application provides
  - Transactional message processing to enable FIFO queue semantics
  - Redelivery timeouts
  - Efficient leader-polling using short-running leases
  - Lock-free snapshotting



Deployment – Kubernetes Manifest 1/3

StatefulSet controller instead of simple Kubernetes deployment

- We chose a StatefulSet because it's designed for stateful applications
- Provides persistent storage, stable network identities and more

For our application we define:

- StatefulSet `[dist-msg-queue]`, key configurations:
  - "replicas": Configurable number of Nodes. Must be divisible by the Number of Partitions. We ran it with 3 and up to 9 Nodes (due to local resource limitations)
  - "image": The application image. Pulled from the Github container registry: `ghcr.io/chris-1187/scalability-demo:latest` (public)
  - "env" variables like `POD_NAME, PEER_SERVICE_NAME, NAMESPACE, REPLICA_COUNT` (for app internal use), `RAFT_STORAGE_BASE_PATH`

Deployment – Kubernetes Manifest 2/3

- Persistent Storage `[volumeClaimTemplates]`:
  - Ensures that each of the N message queue nodes gets its own dedicated, persistent storage
  - Defined within the StatefulSet, a PVC named `data-storage` is created for each pod with 1Gi of storage

- Vertical Pod Autoscaler (VPA) `[dist-msg-queue-vpa]`:
  - Automatically adjusts the CPU and memory resources allocated to our `scalability-demo` containers based on their actual usage. This helps optimize resource utilization and application performance
  - `updateMode: "Auto"`
  - `resourcePolicy:`
    - **minAllowed:** CPU: 125m (0.125 CPU core), Memory: 256Mi (256 Megabytes)
    - **`maxAllowed`**: CPU: 250m (0.256 CPU core), Memory: 512Mi (512 Megabytes)
    - **`controlledResources`**: Specifies that only cpu and memory will be managed by our VPA

Deployment – Kubernetes Manifest 3/3

- Client-Facing Service: `[dist-msg-queue-client]`:
  - Provides a single, stable entry point for external clients (producers and consumers) to interact with our distributed message broker
  - Type: `NodePort` service. Port 8080 (`grpc-client`) is exposed on each node and traffic to that port is then forwarded to the message queue pods

- Headless Service `[dist-msg-queue-peers]`:
  - Enables stable and predictable network identities for each node (pod) within our distributed message broker. This is crucial for nodes to find and communicate with each other
  - It's a "Headless Service" (`clusterIP: None`), meaning it doesn't get a single, stable IP address for the service itself. Instead, it directly exposes the IP addresses of the individual pods
  - Each pod in the StatefulSet gets its own unique, stable DNS record (e.g., `dist-msg-queue-0.dist-msg-queue-peers`, `dist-msg-queue-1.dist-msg-queue-peers`)
  - Exposed Ports: 8081 for grpc-internal; 9999 for raft

Requirements – State Management

- The applicaton manages messages in various message queues distributed over several partitions and replicas

Requirements – Scalability

Vertical scaling:

- Main multithreading bottleneck is the JRaft state machine
- State machine exists once per partition
- => assign more partitions to a single node
- => utilize more cores and RAM
- JRaft groups can work even with only a single node
- VPA for automatic CPU and Memory updates

Horizontal Scaling

- Partitioning distributes queues onto different nodes
- Load on a single FIFO queue should be low
- single queue is no bottleneck, can always create more partitions to handle load
- Message routing always takes O(1) hops regardless of cluster size
- No system wide coordinator (except Kubernetes)
- => scales as far as Kubernetes can handle more nodes

Requirements – Overloading other components

Backoff in between nodes

- When nodes distribute messages among each other, an exponential backoff prevents overload due to failed messages

Requirements – Two Strategies from Lecture

Idempotency

- Pushes to the queue are idempotent for one minute
- A client token is added to push requests
- Client token cache with a TTL of one minute blocks identical requests

Reliability and Durability through Replication

- The different queues are replicated to several nodes to survive the failure of individual nodes

Reliability

- Message level fault tolerance
  - All gRPC calls use retries wit exponential backoff
  - All gRPC operations are idempotent through message UUIDs
- Node level fault tolerance
  - Built into JRaft
  - New leaders are automatically elected in Raft
  - Snapshots and logs are persisted to disk, providing durability
  - => can survive temporary and permanent node failures
  - + if multiple retries fail, nodes will be considered temporarily unhealthy and are then avoided
- Queues have enforced length limit to avoid insurmountable backlogs

Demo

To demonstrate our application we ran the queue service with 3 nodes and with 12 nodes. As there are 3 nodes per partition, there is no horizontal scaling in the first setup.

When running with 12 nodes, there are 4 partitions which could theoretically run 4 times as many queues.