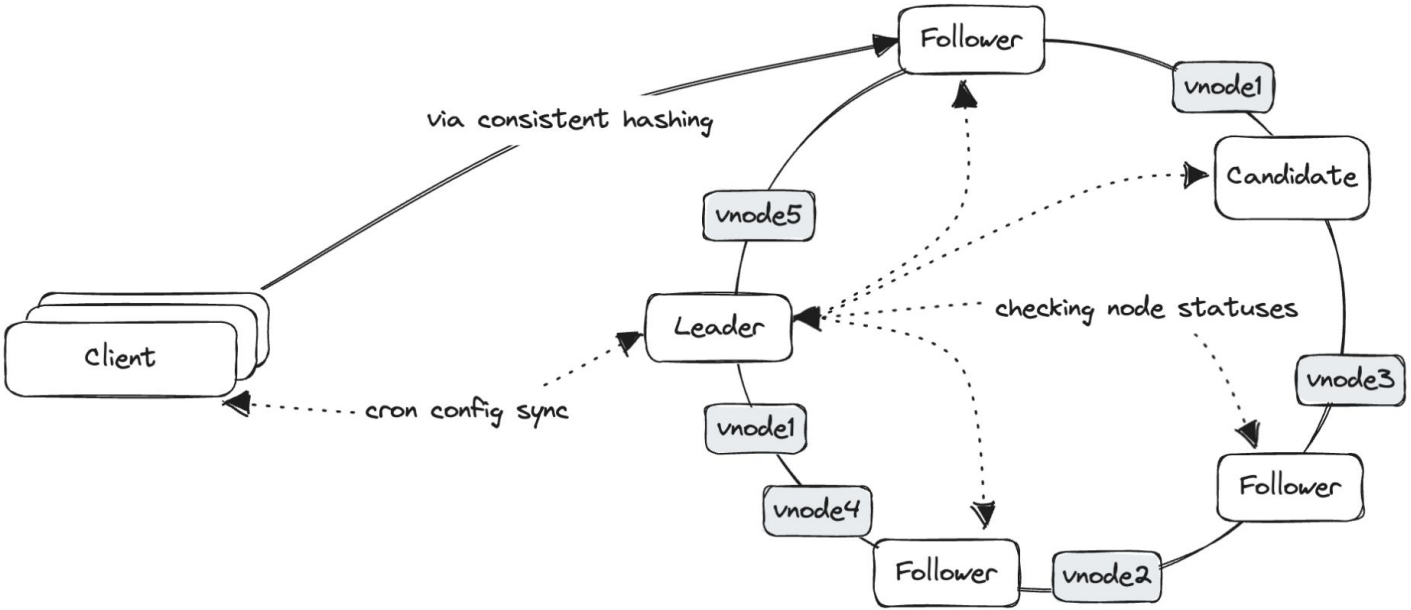
Distributed cache

aka **speedy**

What do we want



OKR Table

Total Fulfillment of Objectives = 75%

Core	Progress	Objective Fulfillment
LRU	100%	100%
Election + sharding researches	100%	
gRPC Client + Server	100%	

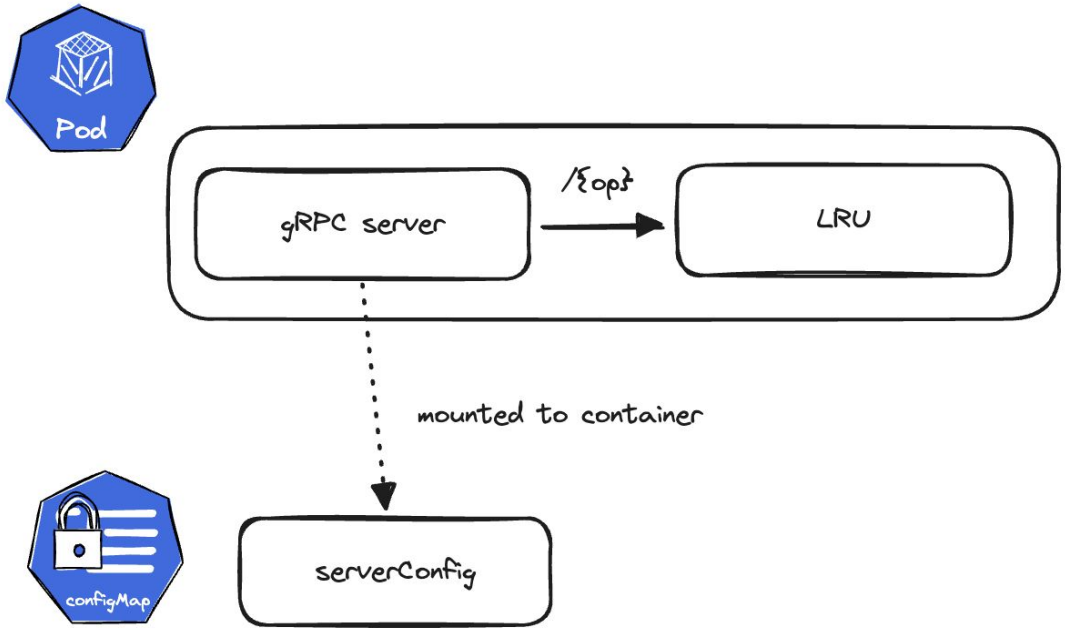
Core v2	Progress	Objective Fulfillment
Keys expiration	33%	77%
Client config sync	100%	
Sharding algorithms impl	100%	

Core v3	Progress	Objective Fulfillment
Bully election algorithm	100%	48%
Raft consensus Algorithm	25%	
Benchmarks	20%	

Server API

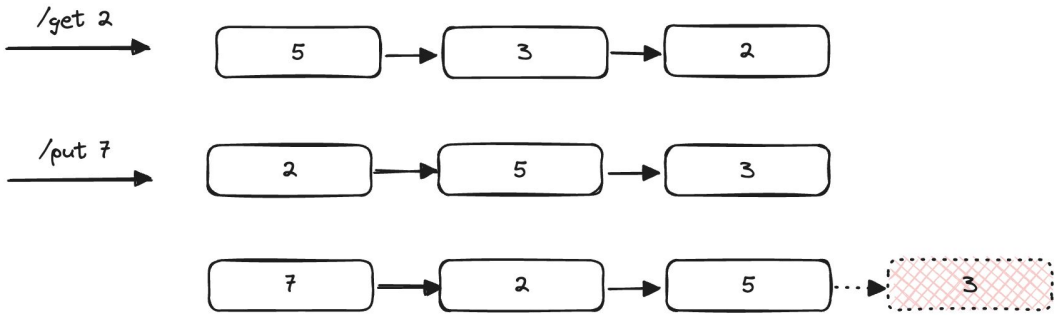
```
service CacheService {
  // Get/Put operations
  rpc Get (GetRequest) returns (GetResponse) {}
  rpc Put (PutRequest) returns (google.protobuf.Empty) {}
  rpc Len (google.protobuf.Empty) returns (LengthResponse) {}

  // Elections
  rpc GetPid(PidRequest) returns (PidResponse);
  rpc GetLeader(LeaderRequest) returns (LeaderResponse);
  rpc GetHeartbeat(HeartbeatRequest) returns (google.protobuf.Empty);
  rpc UpdateLeader(NewLeaderAnnouncement) returns (GenericResponse);
  rpc RequestElection(ElectionRequest) returns (GenericResponse);

  // Cluster management
  rpc GetClusterConfig(google.protobuf.Empty) returns (ClusterConfig);
  rpc UpdateClusterConfig(ClusterConfig) returns (google.protobuf.Empty);
  rpc RegisterNodeWithCluster(Node) returns (GenericResponse);
}
```

Server side


Pod
gRPC server
/{op}
LRU
mounted to container
configMap
serverConfig

Eviction policy

- Handling of situations when the cache becomes full.
- Least Recently Used (LRU) policy



Get example

```go
func (s *CacheServer) Get(_ context.Context, req *api.GetRequest) (*api.GetResponse, error) {
    if val, ok := s.cache.Get(req.Key); ok {
        return &api.GetResponse{Value: val}, nil
    }

    return nil, status.Error(codes.NotFound, KeyNotFoundMsg)
}
```

```go
func (l *lru) Get(key string) (string, bool) {
    l.mx.RLock()
    defer l.mx.RUnlock()

    if node, ok := l.cache[key]; ok {
        l.promote(node)
        return node.val, true
    }

    return "", false
}
```

```go
func (l *lru) promote(node *Node) {  2 usa
    left, right := node.prev, node.next
    if left != nil {
        left.next = right
    }

    if right != nil {
        right.prev = left
    }

    l.justPromote(node)
}
```

Leader election

Used to elect a leader node for the cluster, which is in charge of monitoring the state of the nodes in the cluster to provide to clients so they can maintain a consistent hashing ring and route requests to the correct nodes

Bully election

It follows all the assumptions discussed above in the Election Algorithm.

Let’s say there are 6 Processes P0, P1, P2, P3, P4, P5 written in ascending order of their Process ID (i.e.,0, 1,2,3,4,5).

The Bully Algorithm operates on the principle of **higher priority**.

Messages in Bully Algorithm

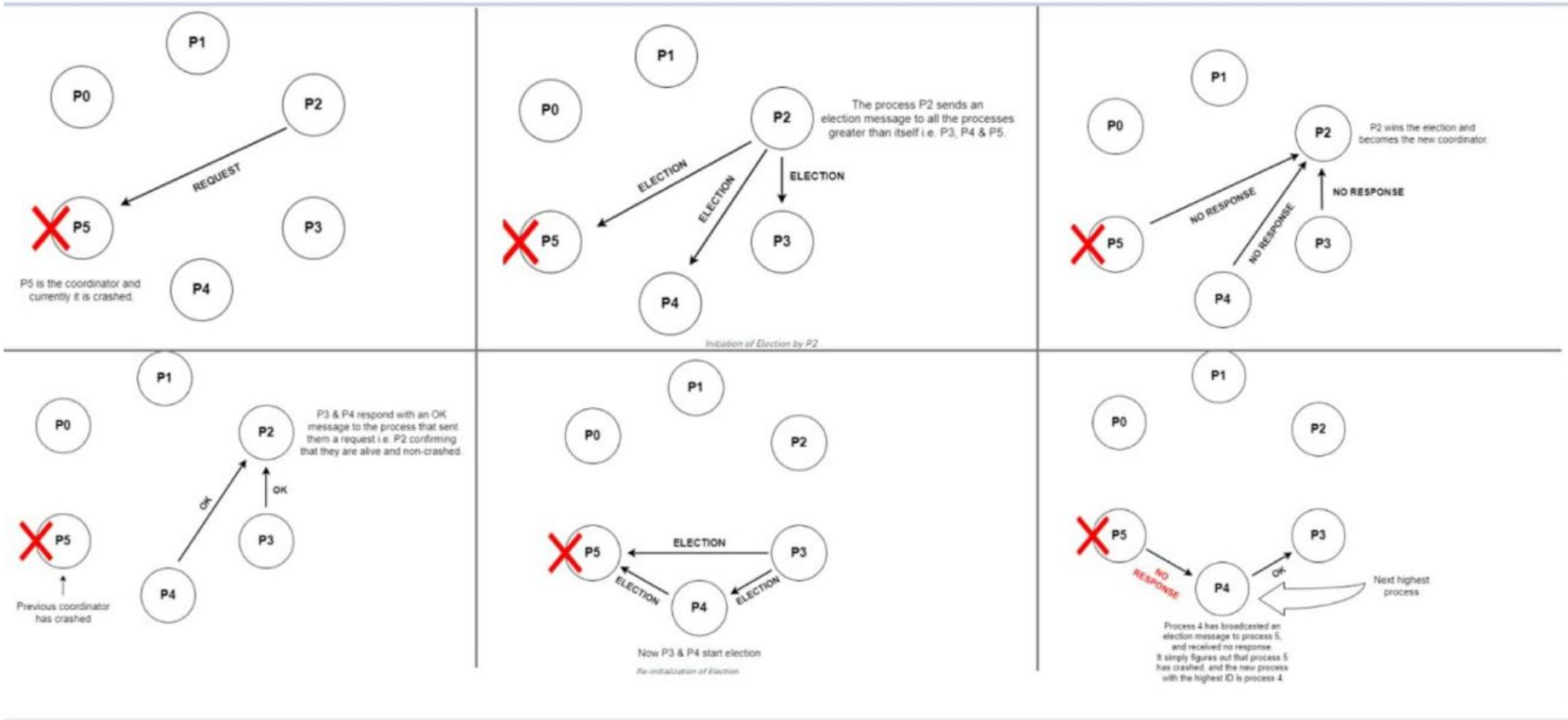
There can be three types of messages that processes exchange with each other in the bully algorithm:

1. Election message: Sent to announce election.

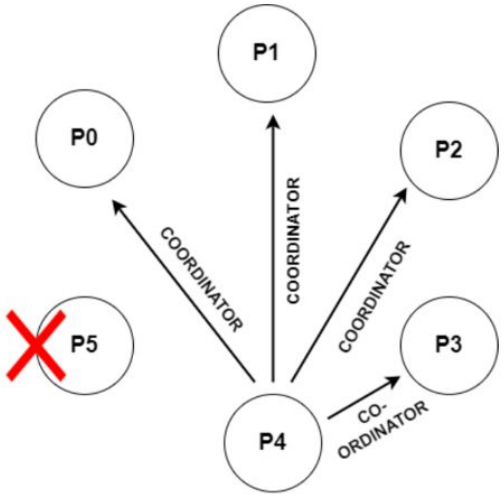
2. OK (Alive) message: Responds to the Election message.

3. Coordinator (Victory) message: Sent by winner of the election to announce the new coordinator.

Bully election

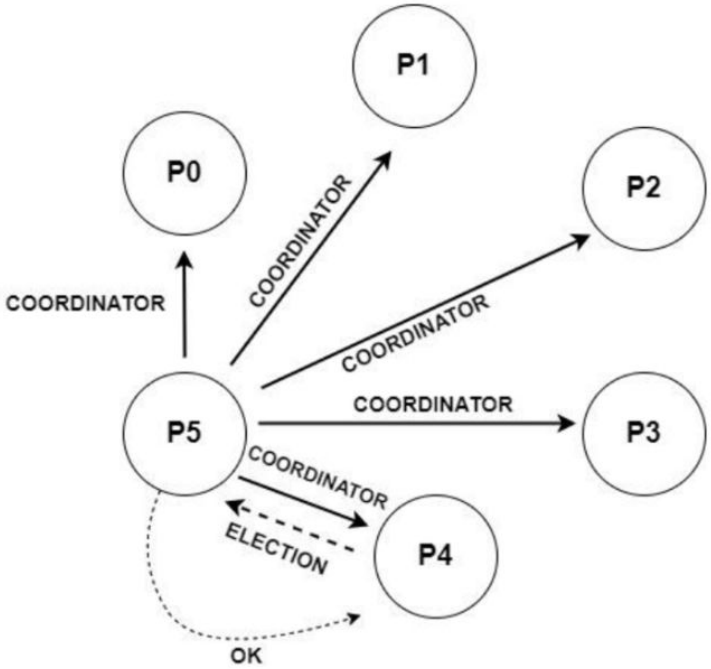


Bully election



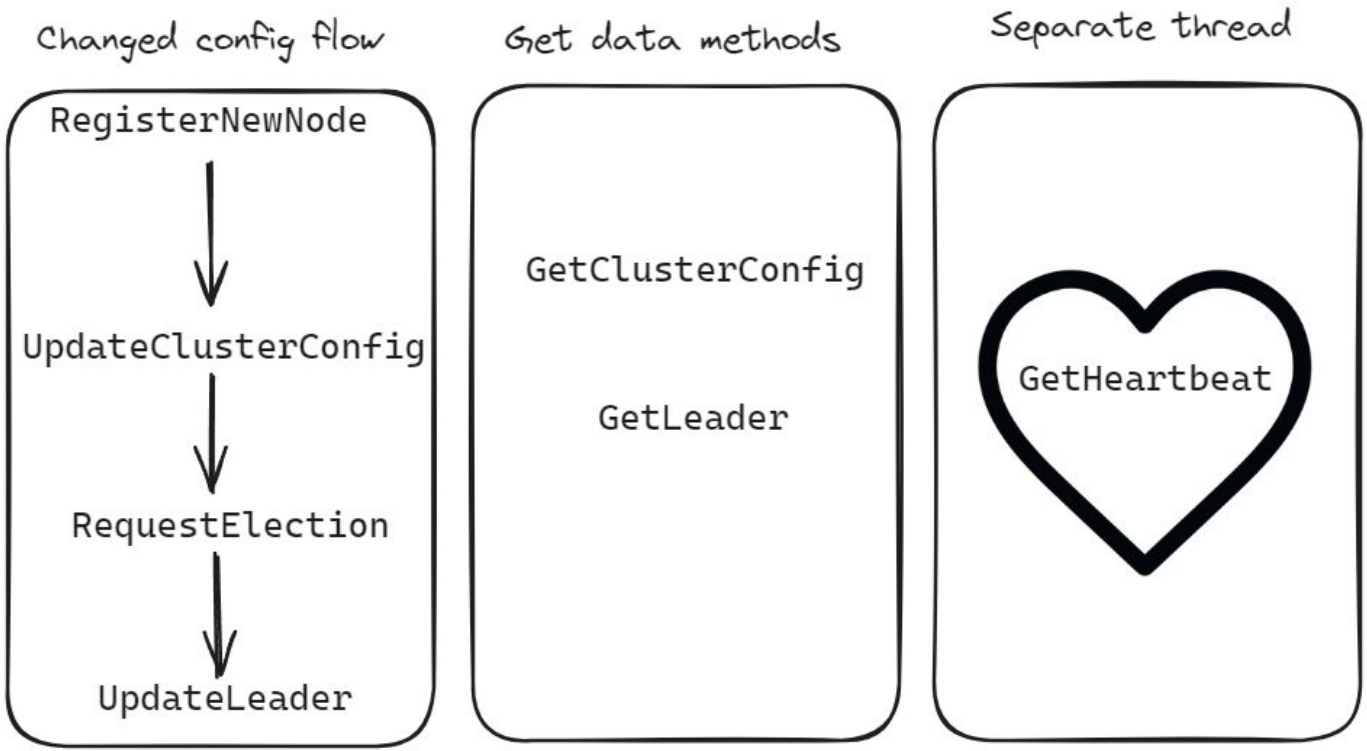
Now P4 sends a coordinator message to all of the alive processes. Consecutively, all nodes are updated with the new leader.

P4 wins the election and becomes the new coordinator



P5 is the highest priority process , hence became the coordinator.

Election API flow



Election code samples here

```go
func (s *CacheServer) RunElection() {  5 usages  DoKep2 *
    // an individual node should run a single election process, not multiple concurrent ones
    s.electionLock.Lock()
    if s.electionStatus {
        zap.L().Info( msg: "Election already running, waiting for completion...")
        s.electionLock.Unlock()
        return
    }

    // update status to election running
    s.electionStatus = ELECTION_RUNNING
    s.electionLock.Unlock()

    // check status of every node
    localPID := int32(os.Getpid())
    zap.S().Infof( template: "Running election. Local PID: %d", localPID)

    for _, node := range s.nodesConfig.Nodes {
        // skip self
        if node.ID == s.nodeID {
            continue
        }
        func() {
            // new identity service client
            ctx, cancel := context.WithTimeout(context.Background(), 3*Timeout)
            defer cancel()

            // make status request rpc
            c, err := s.NewCacheClient(node.Host, int(node.Port))
```

```go
    if err != nil {
        zap.S().Infof( template: "error creating grpc client to node %s: %v", node.ID, err)
    }
    res, err := c.GetPid(ctx, &api.PidRequest{CallerPid: localPID})
    if err != nil {
        zap.S().Infof( template: "PID request to node %s failed", node.ID)
        return
    }

    // if response has a higher PID (use node id as tie-breaker), we send it an election request and wait to
    // the election winner announcement.
    zap.S().Infof( template: "Received PID %d from node %s (vs local PID %d on node %s)", res.Pid, node.ID, loca
    if (localPID < res.Pid) || (res.Pid == localPID && s.nodeID < node.ID) {

        zap.S().Infof("Sending election request to node #{node.ID}")

        c, err := s.NewCacheClient(node.Host, int(node.Port))
        if err != nil {
            zap.S().Infof("error creating grpc client to node node #{node.ID}: #{err}")
        }

        ctx, cancel := context.WithTimeout(context.Background(), 3*Timeout)
        defer cancel()

        _, err = c.RequestElection(ctx, &api.ElectionRequest{CallerPid: localPID, CallerNodeId: s.nodeID})
        if err != nil {
            zap.S().Infof("Error requesting node #{node.ID} run an election: #{err}")
        }

        zap.L().Info( msg: "Waiting for decision...")
        // if after 5 seconds we receive no winner announcement, start the election process over
```

Election work example


```
{"level":"info","ts":1703089909.9585106,"caller":"server/grpc.go:73","msg":"passed node id: DYNAMIC"}
{"level":"info","ts":1703089909.9586654,"caller":"server/grpc.go:75","msg":"final node id: node-gERmj"}
{"level":"info","ts":1703089909.9591036,"caller":"cmd/main.go:45","msg":"Running gRPC server on port 8080..."}
{"level":"info","ts":1703089909.959227,"caller":"server/grpc.go:200","msg":"attempting to register node-gERmj with cluster"}
{"level":"info","ts":1703089909.9594152,"caller":"server/election.go:170","msg":"Leader heartbeat monitor starting..."}
{"level":"info","ts":1703089909.9618728,"caller":"server/cluster.go:23","msg":"Node node-gERmj already part of cluster"}
{"level":"info","ts":1703089909.9625428,"caller":"server/grpc.go:230","msg":"starting grpc server","port":"node-gERmj"}
{"level":"info","ts":1703089910.9601269,"caller":"server/election.go:238","msg":"IsLeaderAlive found leader doesn't exist"}
{"level":"info","ts":1703089910.9602435,"caller":"server/election.go:180","msg":"Leader heartbeat failed, running new election"}
{"level":"info","ts":1703089910.9602804,"caller":"server/election.go:34","msg":"Running election. Local PID: 19326"}
{"level":"info","ts":1703089910.9603443,"caller":"server/election.go:104","msg":"set leader as self: node-gERmj"}
{"level":"info","ts":1703089910.9604177,"caller":"server/election.go:117","msg":"Announcing node node-gERmj won election"}
{"level":"info","ts":1703089910.9604504,"caller":"server/election.go:182","msg":"Election done, leader heartbeat continuing"}
{"level":"info","ts":1703089911.961541,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
{"level":"info","ts":1703089912.961036,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
{"level":"info","ts":1703089913.9606822,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
{"level":"info","ts":1703089914.9604375,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
{"level":"info","ts":1703089915.9602442,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
```

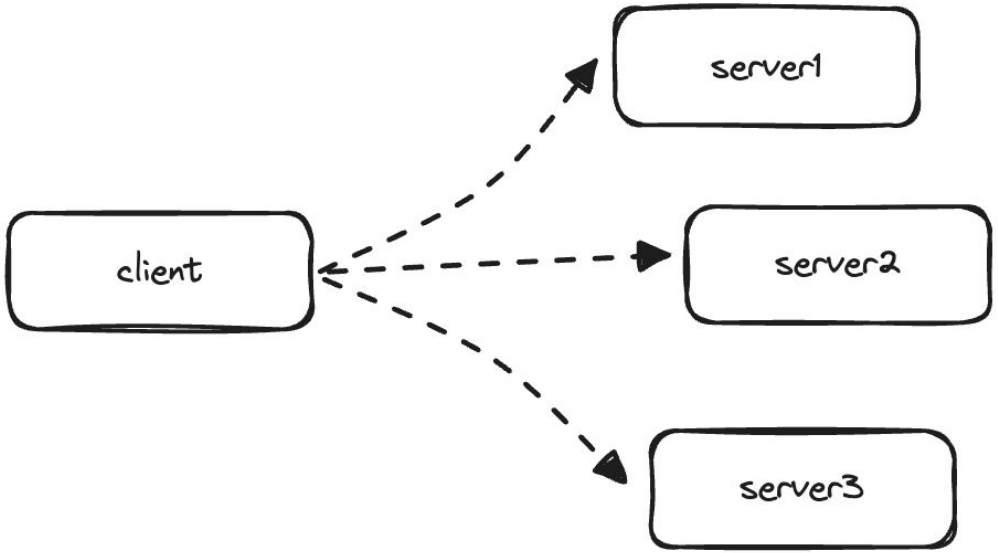


```
{"level":"info","ts":1703089956.9605904,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
{"level":"info","ts":1703089956.9646277,"caller":"server/cluster.go:63","msg":"Updating cluster config"}
{"level":"info","ts":1703089956.9649682,"caller":"server/election.go:34","msg":"Running election. Local PID: 19326"}
{"level":"info","ts":1703089956.9669147,"caller":"server/election.go:59","msg":"Received PID 19326 from node node-BcW5g (vs local PID 19326 on node node-gERmj)"}
{"level":"info","ts":1703089956.967013,"caller":"server/election.go:104","msg":"set leader as self: node-gERmj"}
{"level":"info","ts":1703089956.9671187,"caller":"server/election.go:117","msg":"Announcing node node-gERmj won election"}
{"level":"info","ts":1703089956.968732,"caller":"server/election.go:273","msg":"Received announcement leader is node-gERmj"}
{"level":"info","ts":1703089957.9604075,"caller":"server/election.go:217","msg":"Checking health of node node-BcW5g"}
{"level":"info","ts":1703089957.9622529,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
{"level":"info","ts":1703089958.9602265,"caller":"server/election.go:217","msg":"Checking health of node node-gERmj"}
{"level":"info","ts":1703089958.9618704,"caller":"server/election.go:217","msg":"Checking health of node node-BcW5g"}
```


```
dokep@LAPTOP-MQ3PKVU1:/mnt/c/Users/sergo/Downloads/grpcurl_1.8.9_linux_x86_64$
dokep@LAPTOP-MQ3PKVU1:/mnt/c/Users/sergo/Downloads/grpcurl_1.8.9_linux_x86_64$
dokep@LAPTOP-MQ3PKVU1:/mnt/c/Users/sergo/Downloads/grpcurl_1.8.9_linux_x86_64$
dokep@LAPTOP-MQ3PKVU1:/mnt/c/Users/sergo/Downloads/grpcurl_1.8.9_linux_x86_64$
dokep@LAPTOP-MQ3PKVU1:/mnt/c/Users/sergo/Downloads/grpcurl_1.8.9_linux_x86_64$ ./grpcurl -plaintext -emit-defaults localhost:8080 api.CacheService.GetLeader
{
  "id": "node-gERmj"
}
dokep@LAPTOP-MQ3PKVU1:/mnt/c/Users/sergo/Downloads/grpcurl_1.8.9_linux_x86_64$
```

Client side

- gRPC client
- sharding
- config sync



Sharding

Sharding is a method of distributing data across multiple machines.

When you shard, you say you are moving data, but you haven't yet answered the question of which machine receives which subset of data.

There are few algorithms, for assigning set of keys to a set of machines.

Naive hashing algorithm

machine = hash(key) % num_machines

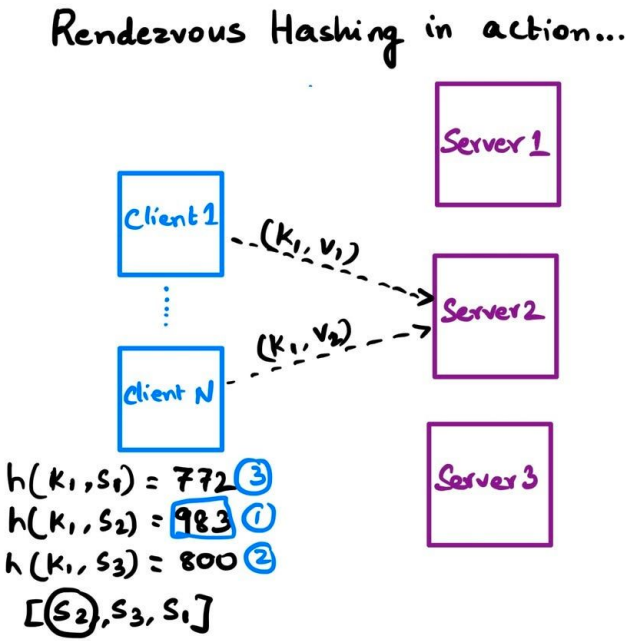
This algorithm is simple, but it has a few problems:

- It doesn't take into account the number of keys on each machine.
  > Some machines may be overloaded, while others are underutilized.
- When number of machines changes, all keys are remapped to different machines.
  > Adding, removing a machine will cause a complete remapping of all keys.

Rendezvous hashing

Each key has a **"meeting" with each server** and the key **selects the server with the highest random weight.**

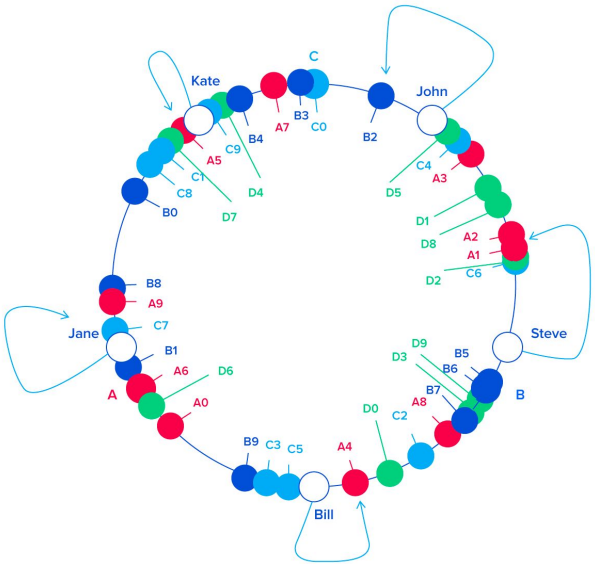
- When a new server is added/deleted, not all the keys need to be remapped.
- Requires O(N) time to find the bucket responsible for a key.



Consistent hashing

Keys and machines are placed on a ring.

- keys can be imbalanced across buckets (solved w/ vnodes)
- harder to implement



Comparison

Algorithm	Time complexity	Keys distribution	Keys remapping
Naive	`O(1)`	Non-uniform	`O(K)`
Consistent	`O(log N)`	Non-uniform	`O(K/N)`
Consistent + virtual nodes	`O(log N)`	Uniform	`O(K/N)`
Rendezvous	`O(N)`	Uniform	`O(K/N)`

Sharding example

```
func (c *client) Get(key string) (string, error) {  Artyom Fadeyev *
    shard := c.algo.GetShard(key)
    if shard == nil {
        return "", ErrCacheMiss
    }

    n := c.nodesConfig.GetNode(shard.ID)
    if n == nil {
        zap.L().Info( msg: "sharding and nodes config are not synced, got outdated shard")
        return "", ErrCacheMiss
    }

    ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
    defer cancel()

    resp, err := n.Request().Get(ctx, &api.GetRequest{Key: key})
    if err != nil { return "", asClientError(err) }

    return resp.Value, nil
}
```

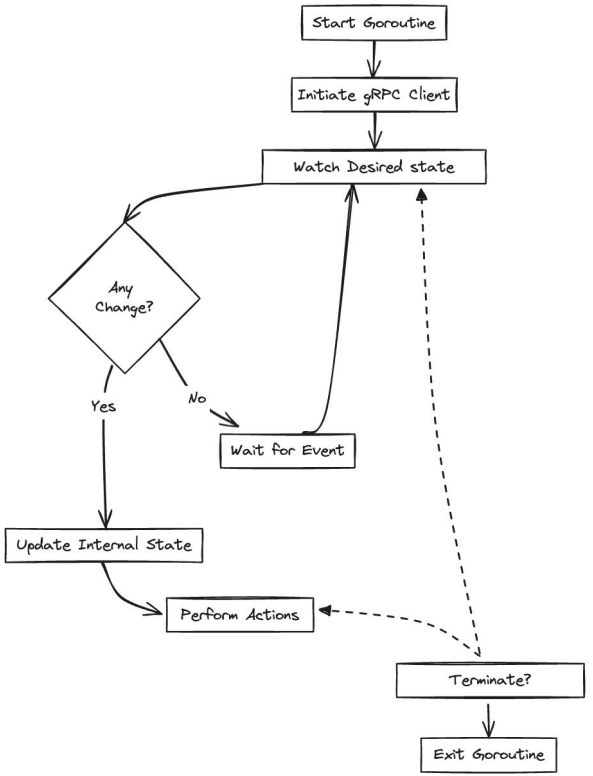
Consistent hashing example

```
func (c *consistent) GetShard(key string) *Shard {  10 usages   Artyom Fadeyev
    var (
        hash    = c.hash(key)
        closest = sort.Search(len(c.orderedKeys), func(i int) bool { return c.orderedKeys[i] >= hash })
    )

    c.mx.RLock()
    defer c.mx.RUnlock()

    closest %= len(c.orderedKeys)
    return c.shards[c.orderedKeys[closest]]
}
```

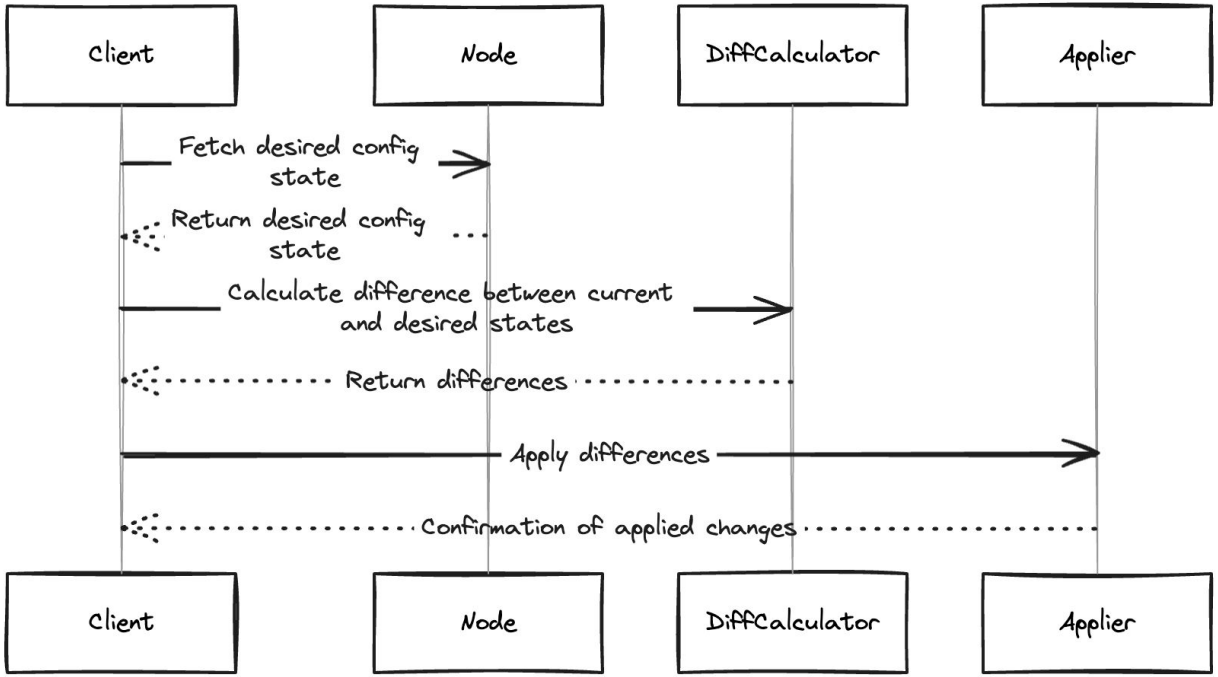
Config sync

- separate goroutine
- **selects** a random node
- **compares** states
- **applies** changes

Kubernetes Operator analogue



Config sync


Client
Node
DiffCalculator
Applier
Fetch desired config state
Return desired config state
Calculate difference between current and desired states
Return differences
Apply differences
Confirmation of applied changes
Client
Node
DiffCalculator
Applier

Config sync example

```
func (c *NodesConfig) Sync() (bool, error) {  Artyom Fadeyev *
    var sourceOfTruth = c.nodeSelector(c)

    ctx, cancel := context.WithTimeout(context.Background(), operationTimeout)
    defer cancel()

    desiredConfig, err := sourceOfTruth.Request().GetClusterConfig(ctx, &emptypb.Empty{})
    if err != nil {
        return false, fmt.Errorf( format: "failed to get cluster config: %w", err)
    }

    return c.syncStates(desiredConfig.Nodes)
}
```

Testing

- unit tests
- integrational tests
- playground

```
➜  speedy git:(master) ✗ go test ./...
?       github.com/fadyat/speedy/api    [no test files]
?       github.com/fadyat/speedy/cmd    [no test files]
?       github.com/fadyat/speedy/node   [no test files]
?       github.com/fadyat/speedy/pkg    [no test files]
?       github.com/fadyat/speedy/playground/client-config-sync  [no test files]
?       github.com/fadyat/speedy/server [no test files]
ok      github.com/fadyat/speedy/client 1.236s
ok      github.com/fadyat/speedy/eviction       (cached)
ok      github.com/fadyat/speedy/sharding       (cached)
```

```
--- PASS: TestConsistent_Flow (0.00s)
    --- PASS: TestConsistent_Flow/new_consistent (0.00s)
    --- PASS: TestConsistent_Flow/register_shard (0.00s)
    --- PASS: TestConsistent_Flow/register_multiple_shards (0.00s)
    --- PASS: TestConsistent_Flow/register_shard_twice (0.00s)
    --- PASS: TestConsistent_Flow/delete_shard (0.00s)
    --- PASS: TestConsistent_Flow/delete_shard_not_found (0.00s)
    --- PASS: TestConsistent_Flow/get_shard (0.00s)
    --- PASS: TestConsistent_Flow/get_shards (0.00s)
    --- PASS: TestConsistent_Flow/not_reassigned_when_adding_shard (0.00s)
```

Unit

```
func TestConsistent_Flow(t *testing.T) {  Artyom Fadeyev +1
    testcases := []struct {
        name   string
        shards []*Shard
        ops    func(s *consistent, fn hashFn)
    }{
        {...},
        {...},
        {
            name: "register multiple shards",
            ops: func(s *consistent, _ hashFn) {
                require.NoError(t, s.RegisterShard(&Shard{ID: "1000"}))
                require.NoError(t, s.RegisterShard(&Shard{ID: "2"}))

                require.Equal(t, expected: 2, len(s.shards))
                require.Equal(t, expected: 2, len(s.orderedKeys))
                require.Equal(t, uint32(1), s.orderedKeys[0])
                require.Equal(t, uint32(4), s.orderedKeys[1])
            },
        },
        {...},
        {...},
        {...},
        {...},
        {...},
        {...},
    }
```

```
ln := func(key string) uint32 {
    key = strings.TrimPrefix(key, prefix: "node")
    return uint32(len(key))
}

for _, tc := range testcases {
    t.Run(tc.name, func(t *testing.T) {
        s := NewConsistent(tc.shards, ln).(*consistent)
        tc.ops(s, ln)

        require.True(t, sort.SliceIsSorted(s.orderedKeys, func(i, j int) bool {
            return s.orderedKeys[i] < s.orderedKeys[j]
        }))
    })
}
```

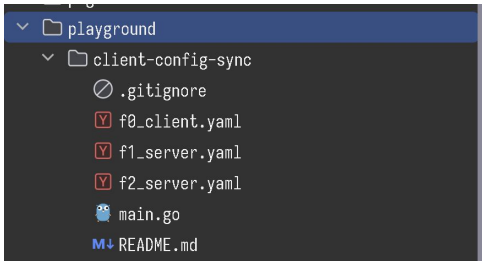
Integrational

```
func TestClient_Flow(t *testing.T) {  Artyom Fadeyev +1
    testcases := []struct {
        name   string
        pre    func(c Client)
        verify func(c Client)
    }{
        {...},
        {...},
        {...},
        {
            name: "out of capacity",
            pre: func(c Client) {
                for i := 0; i < defaultCacheCapacity*2; i++ {
                    require.NoError(t, c.Put(fmt.Sprintf("key%d", i), fmt.Sprintf("value%d", i)))
                }
            },
            verify: func(c Client) {
                var notFound int
                for i := 0; i < defaultCacheCapacity*2; i++ {
                    _, e := c.Get(fmt.Sprintf("key%d", i))
                    if errors.Is(e, ErrCacheMiss) { notFound++ }
                }

                require.Equal(t, defaultCacheCapacity, notFound)
            },
        },
    }
```

go test -v ./...

```
var (
    wg          sync.WaitGroup
    ctx, cancel = context.WithCancel(context.Background())
)

wg.Add( delta: 1)
require.NoError(t, upServer(ctx, &wg, t, defaultServerPort))

path, cleanup := withTemporaryFile(t, singleNodeConfig)
defer cleanup()

c, err := NewClient(path, sharding.RendezvousAlgorithm)
require.NoError(t, err)

for _, tc := range testcases {
    t.Run(tc.name, func(t *testing.T) {
        tc.pre(c)
        tc.verify(c)
    })
}

cancel()
wg.Wait()
```

Playground



make play

`main.go` launches three servers + one client.

- All servers have the same configuration.
- Client have a deprecated configuration, which is needed to be updated.
- In some point of the time, servers are updated with a new configuration.

After some time client configuration is updated to the latest one.

Files description:

- `f0client.yaml` - client initial configuration.
- `f1server.yaml` - server configuration.
- `f2server.yaml` - server new configuration.
- `server-copy.yaml` - temporary file for sharing server configuration.

Non-Functional Requirements

- **Low Latency**
  | Go, gRPC, sharding
- **High Availability**
  | Bully Algorithm, k8s
- **Monitoring and Logs**
- **Easy Scalability**
  | Containerization, Docker, Go

Demo time

Next steps

- resolve current issues
- compare with real-world products
- replication
- more than just in-memory

Source code

https://github.com/fadyat/speedy