```go
package main

import (
	"fmt"
	"log"
	"net"
	"net/rpc"
	"time"
)

type ElectionReq struct {
	SenderID int
}

type ElectionRes struct {
	Ack bool
}

type NotifyLeaderReq struct {
	LeaderId int
}

type NotifyLeaderRes struct {
	Ack bool
}

func (node *Node) callToFollower(peerId int, method string, args interface{}, reply interface{}) error {
	var port int = 8000 + peerId
	return node.callToPeer(peerId, port, method, args, reply)
}

func (node *Node) callToLeader(leaderId int, method string, args interface{}, reply interface{}) error {
	var port int = 8000
	return node.callToPeer(leaderId, port, method, args, reply)
}
func (node *Node) callToPeer(targetNodeId int, port int, method string, args interface{}, reply interface{}) error {
	address := fmt.Sprintf("localhost:%d", port)
	client, err := rpc.Dial("tcp", address)
	log.Printf("[Node %d] Calling method %s", node.id, method)
	if err ≠ nil {
		log.Printf("[Node %d] Could not connect to peer %d at %d: %v\n",
			node.id, targetNodeId, port, err)
		return err
	}
	defer client.Close()
	return client.Call(method, args, reply)

}

```

```go
func (node *Node) startElection() {
	node.electionMutex.Lock()
	if node.isElectionRunning {
		node.electionMutex.Unlock()
		return
	}
	node.isElectionRunning = true
	node.electionMutex.Unlock()

	higherIds := []int{}
	for _, peerId := range node.peerIds {
		if peerId > node.id {
			higherIds = append(higherIds, peerId)
		}
	}
	var isFoundHigherActiveNode = false
	for _, nextId := range higherIds {
		log.Printf("[Node %d] Call Election to node %d ", node.id, nextId)
		var electionReply ElectionRes
		err := node.callToFollower(
			nextId,
			"InternalRPC.Election",
			ElectionReq{node.id},
			&electionReply,
		)
		if err != nil {
			log.Println("Has error while trying to get key", err)
		} else if electionReply.Ack {
			// Have another higher node. Abort
			isFoundHigherActiveNode = true
			break
		}
	}

	if !isFoundHigherActiveNode {
		node.isElectionRunning = false
		// promote to leader and notified all others node
		node.becomeLeader()
	}
}

func (rpcNode *InternalRPC) Election(args ElectionReq, reply *ElectionRes) error {
	// When you receive this. response with OK then continue to call Election to the next node
	reply.Ack = true

	// Check if this node is start election process or not. If not, start one
	if !rpcNode.node.isElectionRunning {
		rpcNode.node.startElection()
	}
	return nil
}
```

```
101
102 type StepDownReq struct {
103         SenderID int
104 }
105
106 type StepDownRes struct {
107         Ok bool
108 }
109
110 func (rpcNode *LeaderRPC) StepDown(args StepDownReq, reply *StepDownRes)
    error {
111         if rpcNode.node.id != rpcNode.node.leaderId {
112                 reply.Ok = true
113                 return nil
114         }
115         if rpcNode.node.id > args.SenderID {
116                 reply.Ok = false
117         } else {
118                 rpcNode.node.leaderListener.Close()
119                 reply.Ok = true
120         }
121         return nil
122 }
123
124 func (rpcNode *InternalRPC) NotifyLeader(args NotifyLeaderReq, reply
    *NotifyLeaderRes) error {
125         newLeader := args.LeaderId
126         reply.Ack = true
127         log.Printf("[Node %d] Receive NotifyLeader %d", rpcNode.node.id,
    newLeader)
128
129         rpcNode.node.leaderId = newLeader
130         rpcNode.node.isElectionRunning = false
131
132         log.Printf("[Node %d] Acknowledge that leader is now %d",
    rpcNode.node.id, rpcNode.node.leaderId)
133         return nil
134 }
135
136 // Heartbeat check
137 type HeartbeatReq struct {
138         SenderID int
139 }
140 type HeartbeatRes struct {
141         Alive bool
142 }
143
144 // Heartbeat: backups call coordinator to confirm it's alive
145 func (rpcNode *LeaderRPC) Heartbeat(req HeartbeatReq, res *HeartbeatRes)
    error {
146         n := rpcNode.node
147         n.electionMutex.Lock()
148         defer n.electionMutex.Unlock()
149         if n.id == n.leaderId {
150                 res.Alive = true
```

```
151         } else {
152                 res.Alive = false
153         }
154         return nil
155 }
156 
157 func (node *Node) becomeLeader() {
158         node.electionMutex.Lock()
159         defer node.electionMutex.Unlock()
160         log.Printf("[Node %d] I am now the leader.\n", node.id)
161         // Reassign and start Leader server only if it not is a leader before
162         if node.leaderId ≠ node.id {
163                 node.leaderId = node.id
164                 for _, nextId := range node.peerIds {
165                         log.Printf("[Node %d] Notify to node %d that leader
    is %d", node.id, nextId, node.id)
166                         var stepDownRes StepDownRes
167                         // Broadcast to other node to release port 8000
168                         err := node.callToLeader(
169                                 nextId,
170                                 "LeaderRPC.StepDown",
171                                 StepDownReq{node.id},
172                                 &stepDownRes,
173                         )
174                         if err ≠ nil {
175                                 log.Println(err)
176                         }
177                 }
178                 node.startLeaderServer()
179         }
180 
181         // Notify to other nodes
182         for _, nextId := range node.peerIds {
183                 log.Printf("[Node %d] Notify to node %d that leader is %d",
    node.id, nextId, node.id)
184                 var notifyLeaderRes NotifyLeaderRes
185                 // Don't care about ack
186                 err := node.callToFollower(
187                         nextId,
188                         "InternalRPC.NotifyLeader",
189                         NotifyLeaderReq{node.id},
190                         &notifyLeaderRes,
191                 )
192                 if err ≠ nil {
193                         log.Println(err)
194                 }
195         }
196 }
197 
198 func (node *Node) startInternalServer(sharedDataBase *Database) {
199         node.internalServer = rpc.NewServer()
200 
201         node.internalServer.Register(&InternalRPC{
202                 node: node,
203         })
```

```
204
205         port := 8000 + node.id
206         addr := fmt.Sprintf(":%d", port)
207         l, err := net.Listen("tcp", addr)
208         if err ≠ nil {
209                 log.Fatalf("[Node %d] Cannot listen on %s: %v", node.id,
    addr, err)
210         }
211         node.internalListener = l
212         log.Printf("[Node %d] Internal server listening on %s", node.id,
    addr)
213         go func() {
214                 for {
215                         conn, err := l.Accept()
216                         if err ≠ nil {
217                                 log.Printf("[Node %d] Internal listener
    closed: %v", node.id, err)
218                                 return
219                         }
220                         go node.internalServer.ServeConn(conn)
221                 }
222         }()
223 }
224
225 func (node *Node) startLeaderServer() {
226         node.leaderServer = rpc.NewServer()
227         node.leaderServer.Register(&LeaderRPC{
228                 node: node,
229         })
230
231         l, err := net.Listen("tcp", ":8000")
232         if err ≠ nil {
233                 log.Fatalf("[Node %d] Cannot listen on %s: %v", node.id,
    "8000", err)
234         }
235         node.leaderListener = l
236         log.Printf("[Node %d] Leader server listening on %s", node.id,
    "8000")
237         go func() {
238                 for {
239                         conn, err := l.Accept()
240                         if err ≠ nil {
241                                 log.Printf("[Node %d] Leader listener closed:
    %v", node.id, err)
242                                 return
243                         }
244                         go node.leaderServer.ServeConn(conn)
245                 }
246         }()
247 }
248
249 func (node *Node) startHeartbeatRoutine() {
250         log.Printf("[Node %d] Start Heartbeat Rountine", node.id)
251         go func() {
252                 for {
```

```
253                         time.Sleep(3 * time.Second)
254
255                         node.electionMutex.Lock()
256                         leader := node.leaderId
257                         node.electionMutex.Unlock()
258                         if leader == -1 {
259                                 node.startElection()
260                                 continue
261                         }
262                         if leader == node.id {
263                                 continue
264                         }
265                         var heartbeatRes HeartbeatRes
266
267                         err := node.callToLeader(
268                                 leader,
269                                 "LeaderRPC.Heartbeat",
270                                 HeartbeatReq{node.id},
271                                 &heartbeatRes,
272                         )
273                         if err ≠ nil || !heartbeatRes.Alive {
274                                 log.Printf("[Node %d] Could not heartbeat to
    leader ⇒ Start electing ", node.id)
275                                 node.startElection()
276                         }
277                 }
278         }()
279 }
280
281 /*
282 User facing methods
283 */
284
285 type SetKeyArgs struct {
286         BucketName string
287         Key        int
288         Value      string
289 }
290
291 type GetKeyArgs struct {
292         BucketName string
293         Key        int
294 }
295
296 type DeleteKeyArgs struct {
297         BucketName string
298         Key        int
299 }
300
301 type Response struct {
302         Data    string
303         Message string
304 }
305
306 type EmptyRequest struct{}
```

```
307
308 func (server *LeaderRPC) SetKey(args *SetKeyArgs, reply *Response) error {
309         database := server.node.database
310         database.mutex.Lock()
311         defer database.mutex.Unlock()
312         database.db.Set(args.BucketName, args.Key, []byte(args.Value))
313         reply.Message = "OK"
314         server.doReplicate(ReplicateDataReq{
315                 BucketName: args.BucketName,
316                 Key:        args.Key,
317                 Value:      args.Value,
318                 Action:     "SET",
319         })
320         return nil
321 }
322
323 func (server *LeaderRPC) GetKey(args *GetKeyArgs, reply *Response) error {
324         database := server.node.database
325
326         data, isExist := database.db.Get(args.BucketName, args.Key)
327         if !isExist {
328                 reply.Message = "Not found"
329         } else {
330                 reply.Data = string(data)
331         }
332
333         return nil
334 }
335
336 func (server *LeaderRPC) DeleteKey(args *DeleteKeyArgs, reply *Response)
    error {
337         database := server.node.database
338         database.mutex.Lock()
339         defer database.mutex.Unlock()
340         ok, err := database.db.Del(args.BucketName, args.Key)
341         if !ok {
342                 reply.Message = "Faild to delete key" + string(err.Error())
343         } else {
344                 reply.Message = "OK"
345         }
346         server.doReplicate(ReplicateDataReq{
347                 BucketName: args.BucketName,
348                 Key:        args.Key,
349                 Value:      "",
350                 Action:     "DELETE",
351         })
352         return nil
353 }
354
355 func (server *LeaderRPC) GetStoreInfo(args EmptyRequest, reply *Response)
    error {
356         database := server.node.database
357         reply.Data = database.db.Info()
358         return nil
359 }
```

```
360
361  /*
362  ————————————————
363  REPLICATE DATA
364  ————————————————
365  */
366
367  type ReplicateDataReq struct {
368          Action     string // SET|DELETE
369          BucketName string
370          Key        int
371          Value      string
372  }
373
374  type ReplicateDataRes struct {
375          IsSuccess bool
376  }
377
378  func (nodeRPC *InternalRPC) ReplicateAction(args ReplicateDataReq, reply
     *ReplicateDataRes) error {
379          log.Printf("[Node %d] Receive replication request with action %s and
     key %s", nodeRPC.node.id, args.Action, args.BucketName)
380          database := nodeRPC.node.database
381          database.mutex.Lock()
382          defer database.mutex.Unlock()
383          if args.Action == "DELETE" {
384                  _, err := database.db.Del(args.BucketName, args.Key)
385                  if err ≠ nil {
386                          return err
387                  }
388          } else if args.Action == "SET" {
389                  err := database.db.Set(args.BucketName, args.Key,
     []byte(args.Value))
390                  if err ≠ nil {
391                          return err
392                  }
393          }
394          return nil
395  }
396
397  func (leaderRPC *LeaderRPC) doReplicate(args ReplicateDataReq) {
398          peerIds := leaderRPC.node.peerIds
399          for _, peerId := range peerIds {
400                  var replicateRes ReplicateDataRes
401                  // Don't care about ack
402                  err := leaderRPC.node.callToFollower(
403                          peerId,
404                          "InternalRPC.ReplicateAction",
405                          args,
406                          &replicateRes,
407                  )
408                  if err ≠ nil {
409                          log.Println(err)
410                  }
411          }
```

```
412 }
413
```