p2p-storage-cluster

A p2p storage cluster for sharing keys/files between multiple nodes in a cluster using consistent hashing and chord DHT

Implementation Details

We are trying to build a prototype of a distributed storage but using a P2P architecture. We are using a consistent hashing function for this. The Chord protocol we have used can support one operation: given a key, it will determine the node responsible for storing the key's value. Chord does not itself store keys and values, but provides primitives that allow higher-layer software to build a wide variety of storage system; CFS is one such use of the Chord primitive. We'll try to explain the basic implementation details below.

Querying mechanism

1. The core usage of the Chord rpc protocol is to query a key from a client (generally a node as well), i.e. to find successor(k).
2. The main approach is to pass the query to a node's successor, if it cannot find the key locally.
3. This will lead to a `O(N)` query time where N is the number of machines in the ring.
4. To avoid the linear search above, it implements a faster search method by requiring each node to keep a finger table containing up to m entries, recall that m is the number of bits in the hash key
5. The `i{th}` entry of node n will contain successor `((n+2^{i-1}) mod 2^m)`.
6. The first entry of finger table is actually the node's immediate successor

Every time, whenever a node wants to look up a key k, it will pass the query to the closest successor or predecessor (depending on the finger table) of k in its finger table (the "largest" one on the circle whose ID is smaller than k), until a node finds out the key is stored in its immediate successor. With such a finger table, the number of nodes that must be contacted to find a successor in an N-node network is `O(log N)`

```
const keySize = sha1.Size * 8

var two = big.NewInt(2)
var hashMod = new(big.Int).Exp(big.NewInt(2), big.NewInt(keySize), nil)

//Calculate exact position on chord ring (1/2, 1/4, 1/8, ...)  based on the fingertable ent
func jump(address string, fingerentry int) *big.Int {
    n := HashString(address)
    fingerentryminus1 := big.NewInt(int64(fingerentry) - 1)
```

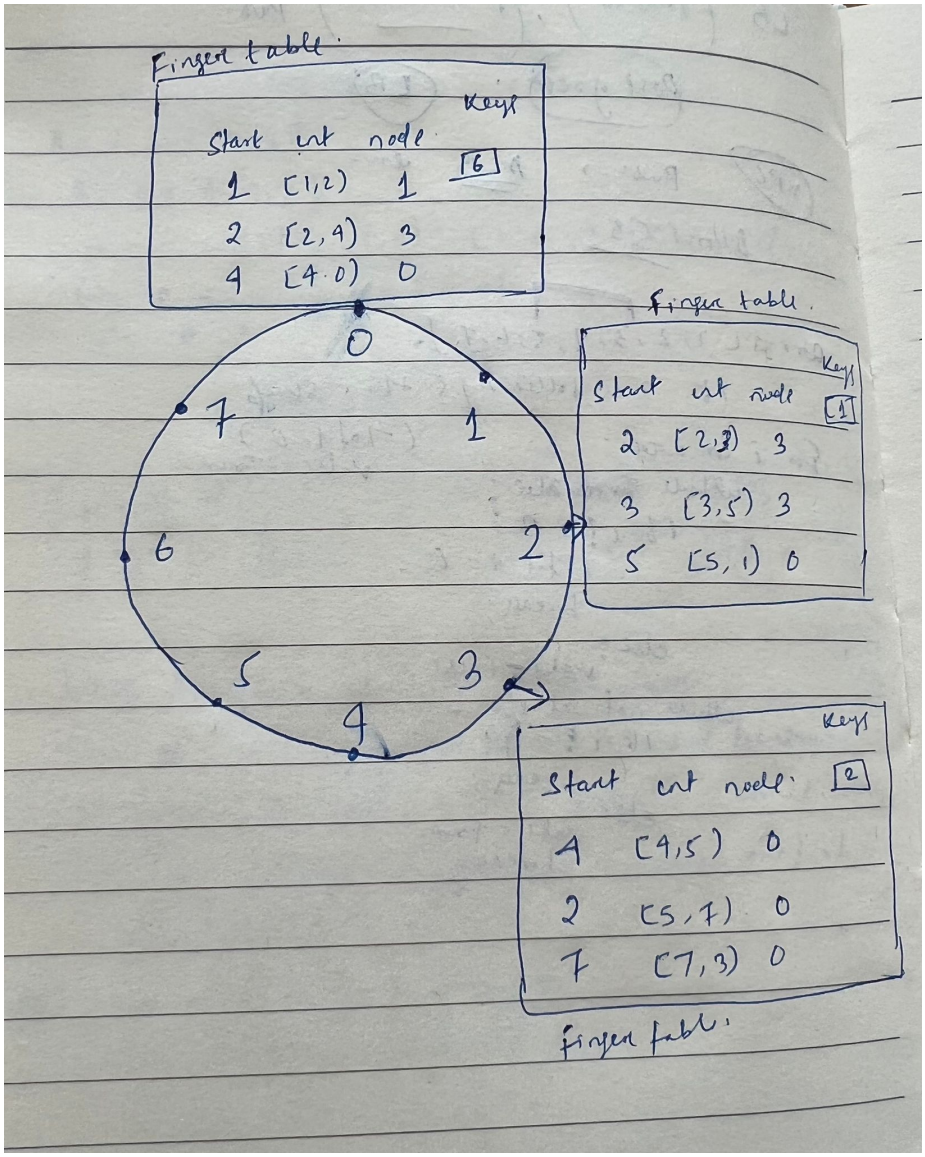

Figure 1: chordDHT

```
    jump := new(big.Int).Exp(two, fingerentryminus1, nil)
    sum := new(big.Int).Add(n, jump)
    return new(big.Int).Mod(sum, hashMod)
}

//Sha-1 hashes a string
func HashString(elt string) *big.Int {
    hasher := sha1.New()
    hasher.Write([]byte(elt))
    return new(big.Int).SetBytes(hasher.Sum(nil))
}
```

The notify condition

```
func (s *Server) Notify(nprime string, reply *int) error {
    finished := make(chan struct{}, 1)
    s.fx <- func(n *Node) {
        if n.Predecessor == "" ||
            Between(HashString(n.Predecessor), HashString(nprime), n.ID, false) {
            n.Predecessor = nprime
            fmt.Printf("Notify: predecessor set to: %s\n", n.Predecessor)
            PrintPrompt()
            *reply = 1
        } else {
            *reply = 0
        }
        finished <- struct{}{}
    }
    <-finished
    return nil
}
```

Join condition of a Node

Whenever a new node joins, we maintain 3 basic invariants :

1. Each node's successor points to its immediate successor correctly. (ensure correctness)
2. Each key is stored in successor(k). (ensure correctness)
3. Each node's finger table should be correct. (keeps querying fast)

To satisfy these invariants, a predecessor field is maintained for each node.

As the successor is the first entry of the finger table, we do not need to maintain this field separately any more. The following tasks should be done for a newly joined node n:

1. Initialize node n (the predecessor and the finger table).

2. Notify other nodes to update their predecessors and finger tables.
3. The new node takes over its responsible keys from its successor

The predecessor of n can be easily obtained from the predecessor of successor(n) (in the previous circle). As for its finger table, there are various initialization methods. The simplest one is to execute find successor queries for all m entries, resulting in `O(M/log N)`initialization time. A better method is to check whether i{th} entry in the finger table is still correct for the `(i+1){th}` entry. This will lead to $O(\log^2 N)$.

get the predecessor

```
func (s *Server) GetPredecessor(_ Nothing, predaddress *string) error {
    finished := make(chan struct{}, 1)
    s.fx <- func(n *Node) {
        *predaddress = n.Predecessor
        finished <- struct{}{}
    }
    <-finished
    return nil
}
```

get the Successors

```
func (s *Server) GetSuccessors(_ Nothing, successors *[]string) error {
    finished := make(chan struct{}, 1)
    s.fx <- func(n *Node) {
        *successors = n.Successors
        finished <- struct{}{}
    }
    <-finished
    return nil
}
```

Stabilization Mechanism

To ensure correct lookups, all successor pointers must be up to date. Therefore, a stabilization protocol is running periodically in the background which updates finger tables and successor pointers.

The stabilization protocol works as follows:

1. `Stabilize()`: n asks its successor for its predecessor p and decides whether p should be n‘s successor instead (this is the case if p recently joined the system).
2. `Notify()`: notifies n‘s successor of its existence, so it can change its predecessor to n
3. `fixFingers()`: updates finger tables/*
4. `checkPredecessor()`: Periodically checks in predecessor is alive

```
func (s *Server) Join(address string, reply *int) error {
    var pingReply *int
    *reply = 0
    successorSet := false
    PrintPrompt("Joining ring " + address + "...")
    finished := make(chan struct{})
    s.fx <- func(n *Node) {
        n.Predecessor = ""
        var nPrime *string
        err := Call(address, "Server.Ping", sendNothing, &pingReply)
        if err != nil {
            return
        }
        if pingReply != nil && *pingReply == 562 {
            err := Call(address, "Server.FindSuccessor", HashString(net.JoinHostPort(n.Addre
            if err != nil {
                return
            }
            n.Successors[0] = *nPrime
            successorSet = true
        } else {
            PrintPrompt("Address specified for join could not be contacted")
        }
        finished <- struct{}{}
    }
    <-finished
    if successorSet {
        go s.keepCheckingPredecessor()
        go s.keepStabilizing()
        go s.keepFixingFingers()

        go func() {
            time.Sleep(4 * time.Second)
            err := s.TransferAll(sendNothing, returnNothing)
            if err != nil {
                return
            }
        }()
        *reply = 1
    } else {
        *reply = 0
    }
    return nil
}
```

Makefile

```
BINARY_NAME=p2pchord

HOME:=$(shell pwd)
OS_NAME := $(shell uname -s | tr A-Z a-z)

os:
    @echo $(OS_NAME)

fmt: go fmt

build:
    GOARCH=amd64 GOOS=darwin go build -o ${BINARY_NAME} main.go
    GOARCH=amd64 GOOS=linux go build -o ${BINARY_NAME} main.go
    GOARCH=amd64 GOOS=window go build -o ${BINARY_NAME} main.go

run:
    ./${BINARY_NAME}

build_and_run: build run

clean:
    go clean
    rm ${BINARY_NAME}

.PHONY: all p2pchord
```

Execution

We can run the following command and start the node in one terminal. Similarly we can open multiple terminals and make sure that the node joins the initial ring with a given address and port

```
go run main.go
```

```
Welcome to p2p-storage-cluster v1.0
By group 107 SDA
Type help for a list of commands

chord> help

--- List of Chord Ring Commands ---
     port <name>       : Set the port the local node should listen on
     create            : Create a new ring
     join <address>    : Join an existing ring that has a node with <address> in it
```

```
     quit              : Shutdown the node. If this is the last node in
                         the ring, the ring also shuts down
--- Key/Value Operations ---
     put <key> <value> : Insert the <key> and <value> into the active ring
     putrandom <n>     : Generates <n> random keys and values and inserts them
                         into the active ring
     get <key>         : Find <key> in the active ring
     delete <key>      : Delete <key> from the active ring
--- Debugging Commands ---
     dump              : Display information about the current node
     dumpkey <key>     : Show information about the node that contains <key>
     dumpaddr <addr>   : Show information about the node at the given <addr>
     dumpall           : Show information about all nodes in the active ring
     ping <addr>       : Check if a node is listening on <addr>
```

create a ring and start the server

```
chord> port 8081
chord> Port set to: 8081
chord> create
chord> Creating new node...
       ID:      90937690318500543492878932321871005784003596683 7
       Address: 192.168.2.2:8081
chord> Creating RPC server for new node...
chord> RPC server is listening on port: 8081
chord> Stabilize: Successor list changed
```

Join a ring

```
chord> port 8083
chord> Port set to: 8083
chord> join 192.168.2.2:8081
chord> Creating new node...
       ID:      137995307149950583754784113079828525940862961722 9
       Address: 192.168.2.2:8083
chord> Creating RPC server for new node...
chord> RPC server is listening on port: 8083
chord> Joining ring 192.168.2.2:8081...
chord> Now part of ring: 192.168.2.2:8081
chord> Stabilize: Successor list changed
chord> Notify: predecessor set to: 192.168.2.2:8081
```

add keys

```
chord> putrandom 10
chord> 10 random key values added
```

```
chord> dumpall

Ring Size:    1461501637330902918203684832716283019655932542976
Address:       137995307149950583754784113079828525940862961722 9 (192.168.2.2:8083)
Predecessor:   909376903185005434928789323218710057840035966837 (192.168.2.2:8081)
Successors:    692281348848302160328020778631951754698184342 67 (192.168.2.2:8085)
               124995776062147944489784590935848488211206008613 (192.168.2.2:8082)
               665262922484064049451073928615787981825719066937 (192.168.2.2:8084)
Fingers: [  1] 692281348848302160328020778631951754698184342 67 (192.168.2.2:8085)
         [158] 124995776062147944489784590935848488211206008613 (192.168.2.2:8082)
         [159] 665262922484064049451073928615787981825719066937 (192.168.2.2:8084)

Bucket:        [izftmfvyft]: h#xtYhRAf%gx
               [sasnjgnewt]: IsO%xuhcZ$wS
               [jqgffphyig]: Sf.%P|lMhLPr
Items in bucket: 3


Ring Size:    1461501637330902918203684832716283019655932542976
Address:       692281348848302160328020778631951754698184342 67 (192.168.2.2:8085)
Predecessor:   137995307149950583754784113079828525940862961722 9 (192.168.2.2:8083)
Successors:    124995776062147944489784590935848488211206008613 (192.168.2.2:8082)
               665262922484064049451073928615787981825719066937 (192.168.2.2:8084)
               909376903185005434928789323218710057840035966837 (192.168.2.2:8081)
Fingers: [  1] 124995776062147944489784590935848488211206008613 (192.168.2.2:8082)
         [157] 665262922484064049451073928615787981825719066937 (192.168.2.2:8084)
         [160] 909376903185005434928789323218710057840035966837 (192.168.2.2:8081)

Bucket:        [wzhukvdiee]: SBG&GE!gVmUv
Items in bucket: 1


Ring Size:    1461501637330902918203684832716283019655932542976
Address:       124995776062147944489784590935848488211206008613 (192.168.2.2:8082)
Predecessor:   692281348848302160328020778631951754698184342 67 (192.168.2.2:8085)
Successors:    665262922484064049451073928615787981825719066937 (192.168.2.2:8084)
               909376903185005434928789323218710057840035966837 (192.168.2.2:8081)
               137995307149950583754784113079828525940862961722 9 (192.168.2.2:8083)
Fingers: [  1] 665262922484064049451073928615787981825719066937 (192.168.2.2:8084)
         [160] 909376903185005434928789323218710057840035966837 (192.168.2.2:8081)

Bucket:        Items in bucket: 0


Ring Size:    1461501637330902918203684832716283019655932542976
Address:       665262922484064049451073928615787981825719066937 (192.168.2.2:8084)
```

```
Predecessor:   124995776062147944489784590935848488211206008613 (192.168.2.2:8082)
Successors:    909376903185005434928789323218710057840035966837 (192.168.2.2:8081)
               137995307149950583754784113079828525940862961722 9 (192.168.2.2:8083)
               692281348848302160328020778631951754698184342 67 (192.168.2.2:8085)
Fingers: [  1] 909376903185005434928789323218710057840035966837 (192.168.2.2:8081)
         [159] 137995307149950583754784113079828525940862961722 9 (192.168.2.2:8083)
         [160] 692281348848302160328020778631951754698184342 67 (192.168.2.2:8085)

Bucket:         [rqllctsyof]: oEdLmBW%IRGf
                [utrrbnkvjc]: vUMjgKlheBrz
                [cmnbapiobb]: YpanbCrMuE|N
                [mfkkylstew]: cA|.cXT^&ysc
Items in bucket: 4


Ring Size:    1461501637330902918203684832716283019655932542976
Address:       909376903185005434928789323218710057840035966837 (192.168.2.2:8081)
Predecessor:   665262922484064049451073928615787981825719066937 (192.168.2.2:8084)
Successors:    137995307149950583754784113079828525940862961722 9 (192.168.2.2:8083)
               692281348848302160328020778631951754698184342 67 (192.168.2.2:8085)
               124995776062147944489784590935848488211206008613 (192.168.2.2:8082)
Fingers: [  1] 137995307149950583754784113079828525940862961722 9 (192.168.2.2:8083)
         [160] 665262922484064049451073928615787981825719066937 (192.168.2.2:8084)

Bucket:         [ipmametfgk]: W%pp$QBbV%YR
                [opgjufgbgu]: XU.APFcCtZo#
Items in bucket: 2
```

get a key

```
chord> get izftmfvyft
chord> Key value pair found: [izftmfvyft] = h#xtYhRAf%!g(MISSING)x
chord> get izftmfvyft
```

delete a node

```
chord> quit
chord> Shutting down node and transferring keys...
chord> all keys transferred now exiting...
```

verify that the keys belong to the deleted node exists:

```
chord> get izftmfvyft
chord> Key value pair found: [izftmfvyft] = h#xtYhRAf%!g(MISSING)x
chord> get izftmfvyft
```

verify if the buckets are all realigned

```
chord> dumpall
```

You should see all the buckets getting realigned and the keys getting redistributed

Results

The video attached is a good description of the transfer mechanism and as you can see, when one node is down, the keys from that node get transferred to the successor node.