Distributing the Dictionary of triplets on a CHORD ring using GO and JSON-RPC

Contents

Description

This project is the implementation of a peer-to-peer lookup protocol for internet applications presented in the paper [1]. Golang is used to implement the peer and their clients, whereas JSON is used for the communication messages between the entities. The reader is strongly urged to read the reference paper[1] if he/she is not familiar with the material.

Hashing Implementation

SHA-1 hash function is used wherein the combination of a node's IP Address and Port is used to generate its id for the CHORD ring

example: Input to the hash function "192.168.0.101:5555"

The hash key generated is then reduced to modulo size of the ring.

For the data, the key is hashed and reduced to modulo 2^(RingSize/2 (+1 if ring size is odd)) and the relationship is hashed and reduced to modulo s^(RingSize/2 (Integer Division))

where 2^RingSize = size of the ring

example: For RingSize of 5, the hash for the key is reduced to 2^3 and relationship is reduced to 2^2.

Then the values of the hash for key and relationship is converted to their binary form, concatenated and the result is converted to decimal again to give the key for the data.

This is done to implement the partial lookup in the following manner

Suppose RingSize = 5 and we have only the key whose has value comes down to 5 (101). Then to implement partial lookup, we loop through 10100 (20) to 10111 (23) to see if we can find the record. Similarly when we do not receive the key of the record for lookup.

Functions used to implement CHORD

Node Initialization Functions

1) Create: This function is used to create a new CHORD ring. i.e. to start the first participant of the ring.

2) Join: This function is used to add a new peer to an already existing CHORD ring. A new incoming node, must have knowledge of a currently active ring member, whom it contacts. This node in turn returns to the new node the position in the ring where it should join.

Periodically Executing Functions

1) Stabalize: This function is implemented to check periodically for new nodes that join the ring and update the existing pointers to include them.

2) Fix_Fingers: This function runs periodically to keep the entries of the finger table of the nodes updated.

3) Check_Predecessor: This function checks to see if the current known predecessor is alive. If not, then predecessor is set to nil.

4) Purge: The purge function is set currently in our implementation to run every 60 seconds on each node and remove the records from the node's database that has neither been accessed nor modified since the last 60 seconds.

Some Helper Functions

1) Find_Successor : Given a key (node or data), this function returns the current successor responsible for that key in the ring.

2) Closest_Preceding_Node: If the key asked for to the Find_Successor does not belong to the current known successor, this function is used to traverse through the finger table to get the appropriate successor value.

Node Departure Function

1) Shutdown: This is a function which a client can call remotely to turn off the node. Before shutting down, the node notifies its successor and predecessor, indicating them to update their entries. Before departure, the node also sends all its data triplets to its immediately known successor.

Data file creation and removal

Each peer node on startup creates a flat file wherein it stores the (key, relationship, value) triplets belonging to itself in the file system.

This file is set to be created by the name of serv<id>.txt in the folder "/usr/tmp".

The file is deleted when the shutdown function is called by the client remotely to turn the node off.

The name of the file to be used is provided in the configuration file at node startup.

Structure of the finger table

The following stricture is used to represent a node entry in our system.

```
type Node_Entry struct {
    NodeId int
    IpAddress string
    Port int
}
```

NodeId: The hashed id of the node in the CHORD ring.

IpAddress; The IpAddress of the node.

Port: The port of service defined by the node for this implementation.

- Thus the finger table consists of entries which are in the form of the structure above.
- All nodes also stores two additional node entries one for its current immediate successor and predecessor each.

Node and Client configuration

The server and client configuration parameters are stored in a flat file, in the form of a JSON object.

The path to this file is passed as command line arguments to the peer and client programs on startup.

The JSON object from the file is read and is unmarshaled to retrieve the configuration parameters for the node to start its service and the client to connect to a peer.

The following structure is used to get the configuration parameters

```
type Node_Config struct {
        ServerID string
        Protocol string
        IpAddress string
        Port int
        PeerIpAddress string
        PeerPort int
        PersistentStorageContainer struct {
            File string
    }
        Methods [7]string
    }
```

Protocol: The protocol which is used to connect to another peer or the client.

IpAddress: Ip Address of this node.

Port: The port of service provided by this node.

PeerIpAddress: The Ip Address of a known active CHORD node.

PeerPort: The port of service of the known active CHORD node.

PersistentStorageContainer: The name of the file that stores the triplets for this node.

Data Implementation at Peer Nodes

- The parameters (dict3 values) are handled and manipulated as a byte array instead of being decoded into a structure format using encoding.json.RawMessage.
- The dict3 values are stored in a flat file and are retrieved line by line when required.
- At the end of each procedure, if required, a response message is generated as a byte array, marshaled into a JSON message and returned to the client.

Client Implementation

- The client program takes two command-line arguments. The first is the path to the file which contains the server configuration JSON message. The second is the input file containing a set of JSON-RPC calls.
- The client uses the parameters provided in the first file to connect to one of the servers.
- On getting the input file location, the client reads the file line by line and makes the remote procedure calls based on the method specified in the JSON message in the file.
- The response from the server is received and displayed.

Execution Directives

'**peer.go**' and '**chord_client.go**' are the node and client programs respectively.

'**config<n>.json**' files contains the JSON message containing the arguments for starting different nodes.

A client connecting to node **n** should use '**config<n>.json**' as its first argument for startup.

'**input1.txt**' and '**input2.txt**' contains the JSON-RPC test messages to be sent from the client to the server.

To run the server, place **peer.go** and **config<n>.go** in a folder in one folder on the machine. From the location of this folder, run the following command:

'go run peer.go config<n>.json'

To run the client, place **chord_client.go, config<n>.go** and **input.txt** in a folder in one folder on the machine. From the location of this folder, run the following command:

'go run chord_client.go config<n>.json input.txt'

config1.json, config2.json, config3.json and config4.json, are set to be run from the machine with IP Address **192.168.0.101** and **config5.json and config6.json** from the machine with IP Address **192.168.0.102**.

The client program can be run from any machine and can be connected to peer running on either machine.

'input2.txt' contains the **'shutdown'** request message.

Test Methods Used

To test, 4 CHORD nodes simultaneously were started and the ring was allowed to stabilize for two minutes.

(Config1.json is the configuration file that should be used to start the first node of the ring.)

Then a fifth node was added and tested that the ring is handling the client request properly by running a client which contacts node running with config3.json.

After waiting for two minutes, a sixth node was added and tested that the client requests are handled correctly.

(Till this point only input1.txt was used to send the requests from the clients)

This time the node also shuts down the peer node it is connected to (Used input2.txt as well to send the request) and let the ring to stabilize before testing the client requests again.

References

[1] Stoica, Ion, et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications." Networking, IEEE/ACM Transactions on 11.1 (2003): 17-32.