CMSC 621 – Advanced Operating Systems

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

**Project 2 Report**

By:

Vaishali Narkhede [ [vaishal1@umbc.edu](mailto:vaishal1@umbc.edu) ]

Neha Tilak [tilak.neha@umbc.edu]

Richa Gandhewar [richag2@umbc.edu]

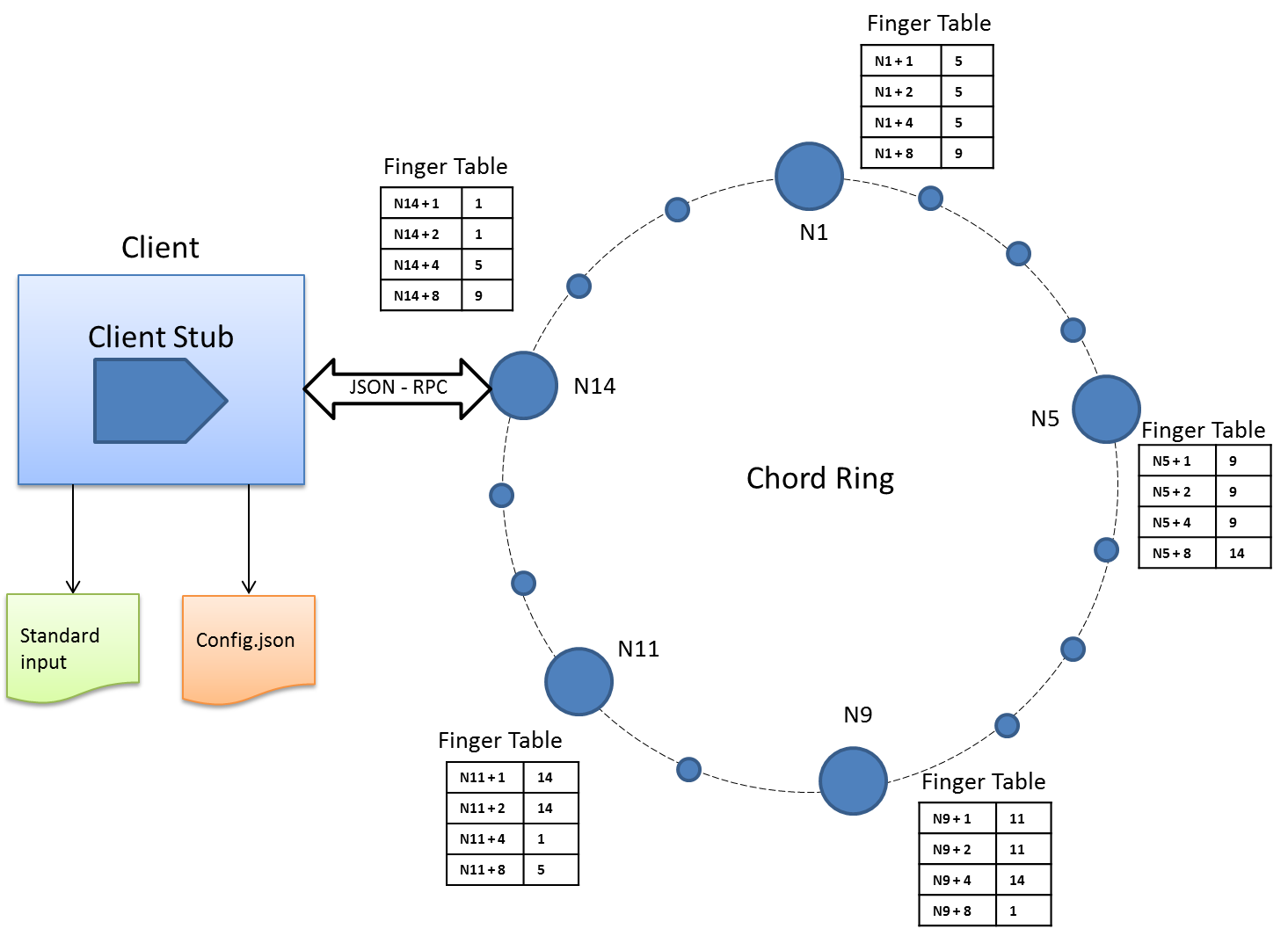
**Introduction:**

Designed and developed a distributed peer-to-peer system that implements the CHORD protocol with GO. A client initiates a connection with any ring node and performs operations such as lookup, insert, insertOrUpdate, delete, listKeys, listIDs and shutdown by using JSON-RPC messages. Ring nodes join and leave the system in a scheduled manner at any time while the remaining ring nodes continue to respond to client requests.

Ring nodes manage a collection of triplets (key, relationship, value) called DICT3, where key and relation are valid JSON strings and value is a valid JSON object. The (key, relationship) pair identifies the triplet and is used to store the triplet into the appropriate node. Server responds to all properly­ structured JSON­-RPC messages for above operations. A client/ring node performs a lookup operation even by partially specifying a triplet’s identifier i.e. it performs lookup even if only key or only relation is provided.

A client node can purge “RW” dictionary triplets that have not been accessed since some user specified time. Client reads a JSON-­RPC request message from the standard input and makes the appropriate request to the server, and shows the response (if any). Both the client and server code takes as their 1st command line argument the filename that contains the client/server configuration JSON object, file name is config.json.

**Design:**



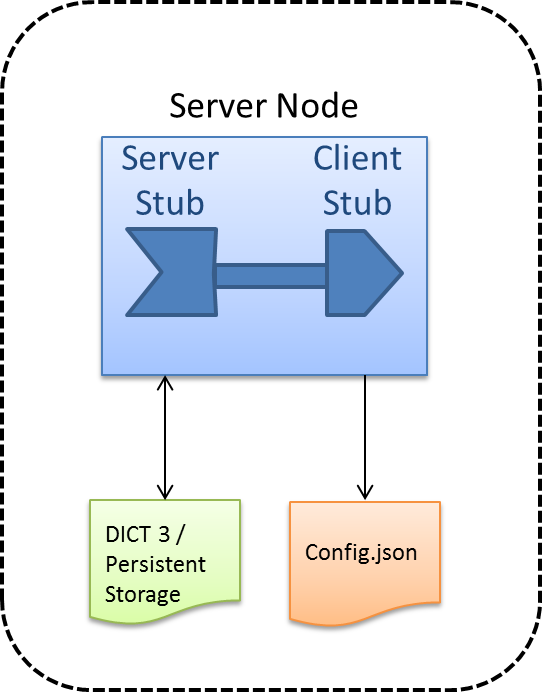


Fig. 1 Block diagram for Go Client server application

Chord is suitable for implementing the distributed hash-table that we intend to create. The main advantage of using Chord is that number of interactions between nodes is of the order of log(N) for each node leaving or joining the ring. We used this to our advantage to build a chord ring that supports upto 32 nodes (m = 5). The same design can be easily scaled up to support larger number of nodes. But we choose 32 just for demonstration.

In this section we explain some of the design decisions we made-

1. **Hashing functions for key-relation and node identifiers**

Our hash function makes use of the SHA1 encryption code for generating the 5-bit chord id. We used different methods to compute chord ids from node identifiers (a string in the form port:ip-address) and key-relation pairs. Following are the steps used-

**For Node Identifier:**

1. Convert the identifier of the server in the form of the string ‘port:ipadreess’

2. Convert string to byte-array

3. Apply SHA1 encoding

4. Add together the last four bytes of the code. (last one is the least significant)

5. Take modulo 32 of the result. This gives a 5-bit chordid of the node.

Note that this method is selected arbitrarily. Any other method is also correct as long as the chord-id assigned are random and uniformly distributed.

**For Key-Relation :**

The chord triplets are a map of key-relation pairs and json objects. We want to support partial-match lookup in the ring. For this reason, we divide the chord id of the key-relation into two parts. The first part is 3 bits long and is derived from the key field alone. The second part is 2 bits long and is derived from relation field alone. We selected an (almost) even split for the key-relations because we have no data as to which field is more likely to be missing in the partial-match query. For example, if there is evidence that the key field is more likely to be missing, then we could assign fewer bits for encoding the key and more bits for the relation. This reduces the number of lookups on the ring for partial-match queries.

The chord id for a key-relation is computed as below:

1. Apply SHA1 encoding to the key string .

2. Take the last three bits.

3. Apply SHA1 encoding to the relation string .

4. Take the last two bits. Append these bits to the three bits taken from key hash.

2. **Partial-Match Lookup Procedure**

As seen in previous section, the first three bits of the triplet chord ID are derived from the key and the last three are derived from the relation. We use this to our advantage to service partial match queries. If a client issues a request where the key is present but the relation is missing, the server simply computes the first three bits of the chord ID from the key. The last two bits are still unknown. The server then finds all possible values of these two bits (), four in this case. Thus the search space is reduced to just four different lookups on the chord ring. Similarly, when relation is known and the key is missing, the number of individual lookups is limited to () eight.

The partial lookup algorithm can be loosely explained as follows-

1. Client sends a query to a chord server with the relation field missing.

2. Server computes the hash-code of the key field. It then creates four different chord- IDs by appending all possible values of the relation hash-code.

3. One by one, the server fires a lookup request on the chord ring for each of these 5-bit chord-id.

4. The responses of the individual lookups are collected and combined and sent over to the client.

**Node Joining –**

First Node – The first node creates a chord ring. The node is placed on the chord ring depending on the hash value generated by SHA1. Its successor and predecessor are set to itself.

Other Nodes – A node can send a join request to any node on the chord ring which determines the position of the joining node on the chord ring by finding the successor. The predecessor of the joining node is set to -1. All the triplets, for which the joining node is responsible, are transferred to it by its successor.

Stabilize method is called periodically on every node which will update the predecessor and successor of the nodes affected by the joining node.

FixFingers is called periodically on all the nodes, which updates finger tables of all the nodes to reflect the changes due to node joining.

**Node Leaving –**

Last Node – When the last node leaves the chord ring, all the triplets present at the last node are stored persistently in a text file.

Other Nodes – If the leaving node is not the last one, the triplets are not stored persistently. Instead, the leaving node hands over all its triplets to its successor.

Stabilize method is called periodically on every node which will update the predecessor and successor of the nodes affected by the leaving node.

FixFingers is called periodically on all the nodes, which updates finger tables of all the nodes to reflect the changes due to node leaving.

**Asynchronous Communication –**

We are using asynchronous calls to for Remote Procedure Calls interactions between client and server or any two nodes on chord ring talking to each other using RPC. This helps improve interaction delays and enables server to take next request for processing without worrying about response of first request. This is necessary for server accepting multiple client requests at a time; asynchronous calls make use of channel which sends a signal once call is complete.

**Implementation:**

**Structure of DICT3:**

Key and relation are strings and value is any valid JSON object so its stored as map[string]interface{}. As key and relation form unique combination to retrieve value, I have used struct DKey as shown in below struct list to use as key for DICT3.

So DICT3 is of type *map[DKey]map[string]interface{}*

**User defined structs used at client and server side are as follows:**

type ***DKey*** struct { KeyA, RelA string }

type ***DIC3*** struct {}

type **Request** struct { KeyRel DKey Val map[string]interface{} Permission string}

type **Response** struct { Tripair Triplet Done bool Err error}

type **ListRequest** struct { SourceId int}

type **Triplet** struct { Key, Rel string Val map[string]interface{} }

type **ListResponse** struct { List interface{} Err error}

type **NodeInfo** struct { Chordid int Address string}

**Below methods implemented:**

|  |  |  |  |
| --- | --- | --- | --- |
| Method name | Client side implementation | Server side implementation | Description |
| Lookup | Lookup(key, relation) | (t \*DIC3) Lookup(req \*Request, reply \*Response) error | Returns value for given key and relation pair from DICT3 |
| Insert | Insert(key, relation, value) | (t \*DIC3) Insert(triplet \*Request, reply \*Response) error | Inserts a key, relation, value triplet in DICT3 |
| InsertOrUpdate | InsertOrUpdate(key, relation, value) | (t \*DIC3) InsertOrUpdate(triplet \*Request, reply \*Response) error | Inserts or updates a key, relation, value triplet in DICT3 based on key and relation pair |
| Delete | Delete(key, relation) | (t \*DIC3) Delete(req \*Request, reply \*Response) error | Deletes key, relation, value triplet based on key and relation pair |
| Listkeys | Listkeys() | (t \*DIC3) Listkeys(req \*Request, reply \*ListResponse) error | Returns set of keys in DICT3 |
| ListIDs | ListIDs() | (t \*DIC3) ListIDs(req \*Request, reply \*ListResponse) error | Returns set of key, relation pairs in DICT3 |
| Shutdown | Shutdown() | (t \*DIC3) Shutdown(dKey \*DKey, reply \*int) | Server shuts down in scheduled manner:- 1.Informs successor and predecessor nodes before leaving  2. Sends dict3 table entries to the successor  3. If it is the last node on the ring, it saves the dict3 in persistent storage. |
| PurgeUnusedTriplets | PurgeUnusedTriplets(ptime) | (t \*DIC3) PurgeUnusedTriplets(req ListRequest, reply \*int) error | Deletes all dict3 entries that were last accessed more than ptime seconds ago. |

**Background functions running at each server:**

|  |  |  |
| --- | --- | --- |
| Method name | Server side implementation | Description |
| Stabilize | func stabilize() | This function is called periodically at each server. Each node verifies its predecessor and successor. |
| check\_predecessor | func check\_predecessor() | This function is called periodically. Each node checks whether predecessor has failed. |
| fixFingers | func fixFingers() | This function is called periodically at each node. Each node updates its finger table. |

Client reads instructions to execute by calling server from commands.txt file, path to commands.txt is provided to client as command line arguments.

Both client and server reads config.json for configuration, path to this file is sent as first command line argument to both client and server.

**Result:**

We are able to successfully build Chord ring, where nodes can join and leave at any time and handled management of persistent data. We experimented this on using 4 nodes in Chord ring and 1 independent client contacting any arbitrary node from Chord ring.