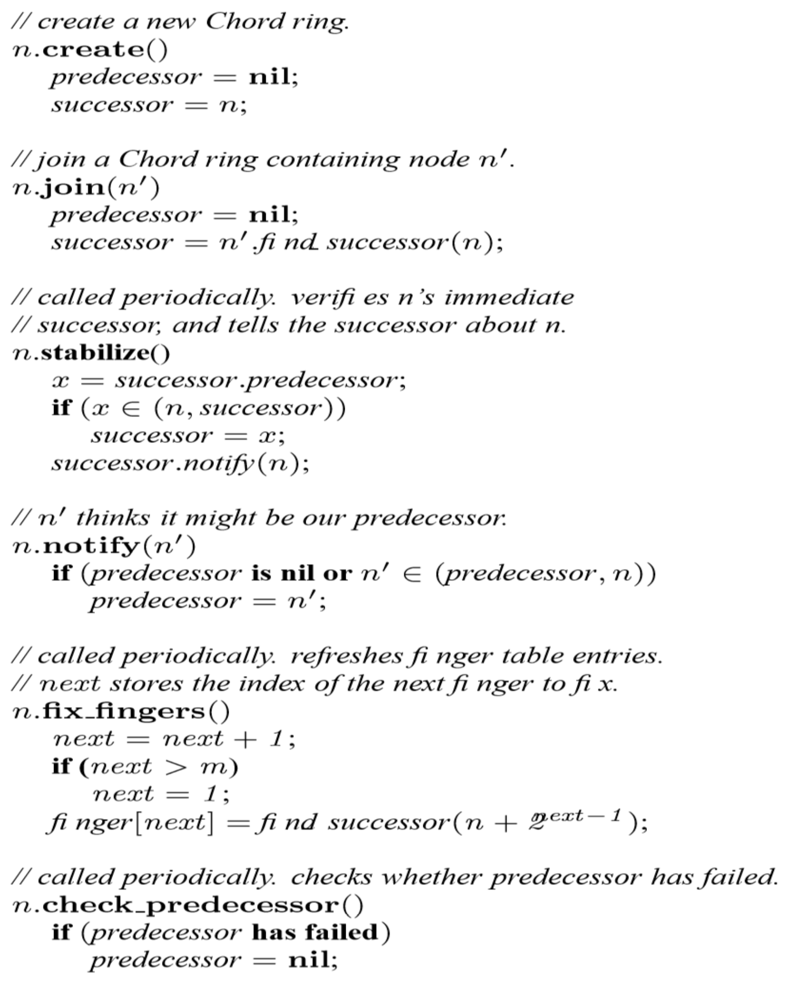
DOS Project 3

Surya Sudharshan (UF ID:5019-3163)

Problem Statement:

A fundamental problem that confronts peer-to-peer applications is the efficient location of the node that stores a desired data item. This project presents Chord, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis and simulations show that Chord is scalable: communication cost and the state maintained by each node scale logarithmically with the number of Chord nodes.

Algorithm:



```
// create a new Chord ring.
$n$.create()
    predecessor = nil;
    successor = $n$;

// join a Chord ring containing node $n'$.
$n$.join($n'$)
    predecessor = nil;
    successor = $n'$.fi nd successor($n$);

// called periodically. verifi es n's immediate
// successor, and tells the successor about n.
$n$.stabilize()
    $x$ = successor.predecessor;
    if ($x \in (n, successor)$)
        successor = $x$;
    successor.notify($n$);

// $n'$ thinks it might be our predecessor.
$n$.notify($n'$)
    if (predecessor is nil or $n' \in (predecessor, n)$)
        predecessor = $n'$;

// called periodically. refreshes fi nger table entries.
// next stores the index of the next fi nger to fi x.
$n$.fix_fingers()
    $next = next + 1$;
    if ($next > m$)
        $next = 1$;
    fi nger[$next$] = fi nd successor($n + 2^{next-1}$);

// called periodically. checks whether predecessor has failed.
$n$.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
```

Fig. 6. Pseudocode for stabilization.

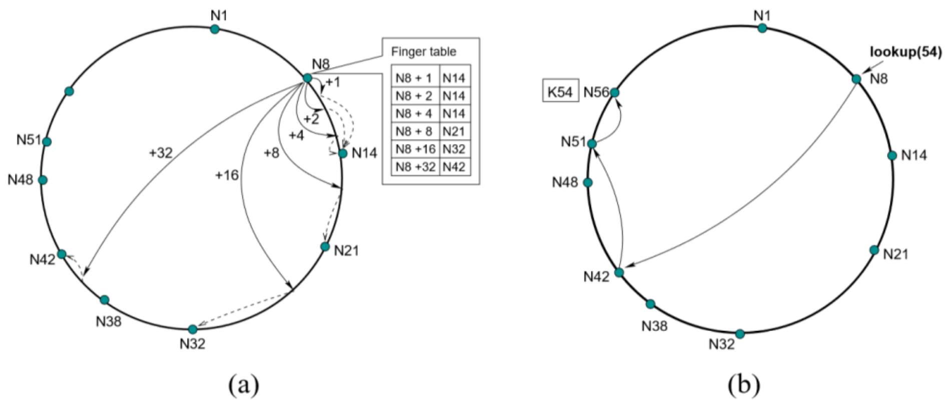


Fig. 4. (a) The fi nger table entries for node 8. (b) The path a query for key 54 starting at node 8, using the algorithm in Figure 5.

Notation	**Defi nition**
finger$[k]$	first node on circle that succeeds $(n + 2^{k-1}) \bmod 2^m$, $1 \leq k \leq m$
successor	the next node on the identifier circle; *finger*$[1]$*.node*
predecessor	the previous node on the identifier circle

TABLE I
Defi nition of variables for node n, using m-bit identifi ers.

Implementation:

Initially, the chord ring is created with just one random node/actor. The remaining n-1 nodes are added into the ring using join method. As joins keep happening stabilize, check_predecessor are called at intervals of 50ms and fix_fingers are called at intervals of 20ms. At the end of the Genserver calls, the chord ring is created.

Next the message genserver calls start the lookup requests that pass messages and calculate the average hop. Once a process completes its lookup and calculates its average hop, it exits by calling the node_completed genserver call. Once all processes are terminated/completed, the genserver call of lookup is completed and exits posting the average number of hops of all processes.

Expected Results:

Results from theoretical analysis and simulations show that Chord is scalable: communication cost and the state maintained by each node scale logarithmically with the number of Chord nodes.

Actual results:

For Normal implementation:

Number of Processes	Average Hops
50	2.14
100	2.75
150	2.99
200	3.15
500	3.30
750	3.30
1000	3.25
1500	3.38
2000	3.30
4000	3.40
4500	3.90
4900	5.00

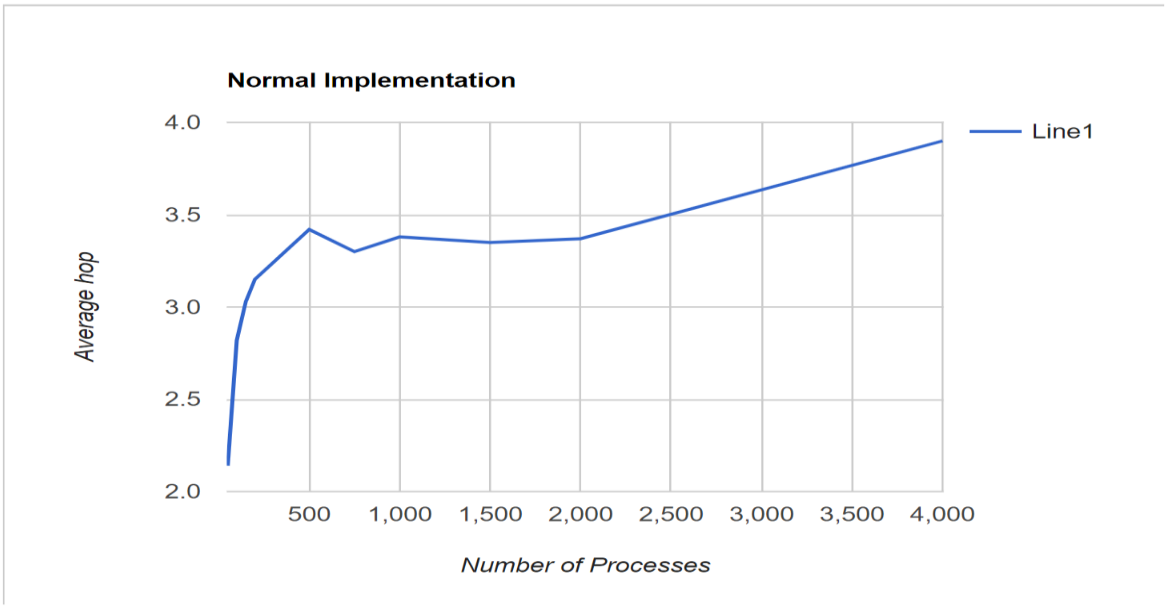
For Implementation with Failures of 20%:BONUS

Number of Processes	Average Hops
50	1.32
100	1.54
150	1.9
200	1.95
500	2.00
750	2.10
1000	2.25
1500	2.20
2000	2.17
4000	2.20
4500	2.10
4900	2.20

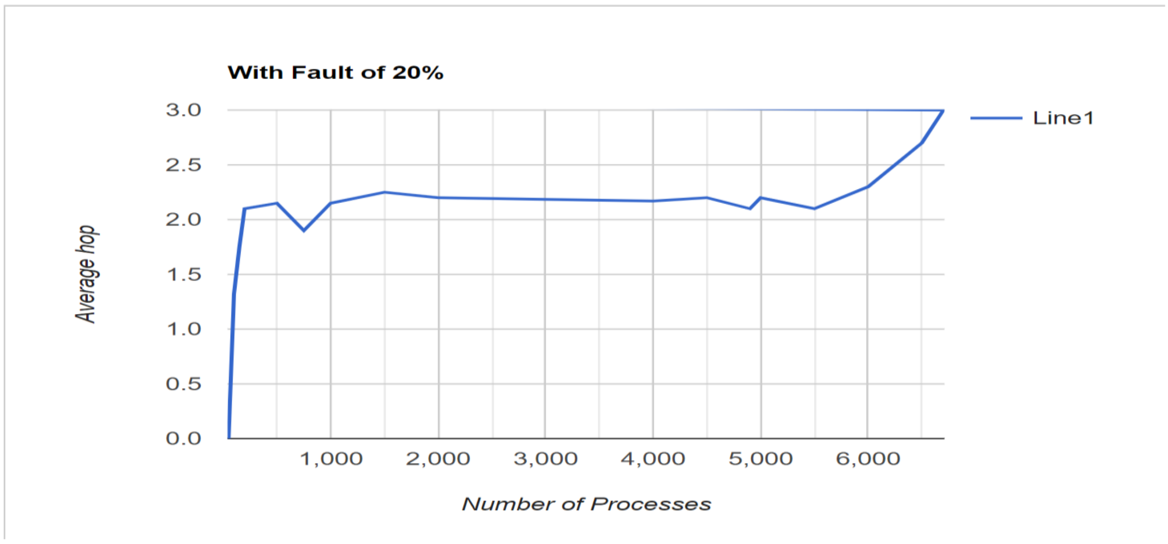
5000	2.10
5500	2.60
6000	3.10
6500	3.20

As expected the plots of Avg Hops vs Number of Nodes near emulated a log(n) plot.

Normal Implementation:



With Failures:BONUS



Note: Average Hops depicted in graphs and tables are averages of multiple readings.

Interesting Observations:

1. The average number of hops is of the order of O(logn) .
2. I observed that the average number of hops is independent of the number of requests, it is only dependent on the number of nodes in the network
3. As the average hops is of order log(n), even if the network is large the number of hops doesn't increase significantly resulting in efficient search of keys when it comes to heavy duty distributed systems.
4. The average hops when failures were implemented varied quite a bit for same inputs. Whereas the variations for normal implementation was not noticeable, sometimes almost the same.
5. The maximum network for normal and failure implementations varied being 4900 for normal and 6500 for failures.