Project Report

CSE535: Asynchronous Systems

Secure Routing for Chord in DistAlgo

Shivasagar Boraiah, Varun Hegde, Raveendra Soori

112077826, 111986703, 111498402

December 10, 2018

STATE UNIVERSITY OF NEW YORK, STONY BROOK

NY - 11794

To

Prof. Annie Y. Liu

Department of Computer and Information Science

STATE UNIVERSITY OF NEW YORK, STONY BROOK

Contents

1 Problem and Plan

1.1 Problem Statement

The project aims at implementing an extension to original chord protocol to support secure routing techniques as proposed by the paper [1] and compare the performance against the existing chord implementation.

1.2 State of the art

While exploring security threats in peer-to-peer networks we came across following works [1] and [2]. [1] discusses three ways of achieving secure routing such as secure assignment of nodeIds by using a central certificate authority, secure routing table maintenance using constrained routing table and secure message forwarding using redundant routing.

Secure routing table maintenance is solved by constraint routing table wherein the nodeIDs that can be filled in the routing table are constrained to be in the neighbourhood of some point in the id space. Secure message forwarding solves the problem of attacker not forwarding messages according to the algorithm to the desired nodes. The solution proposed is to perform a failure test and perform redundant routing when the failure test returns positive. However, the paper discussed all these techniques with Pastry as the protocol and leaves the other p2p protocols as future work. We intend to explore these methods with respect to Chord.

In [2], the work particularly discusses the Chord protocol and describes a version that they call Secchord. It builds upon the idea of routing failure tests discussed earlier and proposes a hop verification method. However, this work has not been recognized widely.

Papers [3] and [4] helped in identifying and analyzing the correctness and performance metrics. We preferred DistAlgo as the programming language as all the team members were comfortable with it and it is one among the easiest languages to model Distributed Algorithms. We found four open source DistAlgo implementations [5–8] of Chord and chose the one we found to be the best in terms of readability.

1.3 Input

- m: Network size.
- 25% of malicious nodes as discussed in [1].
- Version of chord to run (chord/securechord)

1.4 Output

- A system running secure chord with correctly implemented distributed key value lookup service.
- Performance comparison of secure chord with original chord using the metrics such as lookup time, hop count and message complexity [4].
- Correctness tests including simple insert and lookup tests.

1.5 Performance metrics

Metrics identified to analyze the performance of the chord are as follows [3, 4]:

- Hop count
- Message Complexity
- Lookup Ratio
- CPU time
- Lines of Code

1.6 Applications

Following are examples of application of chord [9]:

- Cooperative Mirroring: A load balancing mechanism by a local network hosting information available to computers outside of the local network. This scheme could allow developers to balance the load between many computers instead of a central server to ensure availability of their product.
- Time-shared storage: In a network, once a computer joins the network its available data is distributed throughout the network for retrieval when that computer disconnects from the network. As well as other computers' data is sent to the computer in question for offline retrieval when they are no longer connected to the network. Mainly for nodes without the ability to connect full-time to the network.
- Distributed Indices: Retrieval of files over the network within a searchable database. e.g. P2P file transfer clients.
- Large scale combinatorial searches: Keys being candidate solutions to a problem and each key mapping to the node, or computer, that is responsible for evaluating them as a solution or not. e.g. Code Breaking

2 Design

2.1 DistAlgo Processes

- Driver - Driver is responsible for creating chord nodes, insertion of keys and performance/correctness testing.
- Chord - Chord without security fixes
- Secure Chord - Chord with security fixes
- Certificate Authority - Responsible for generating certificates.

2.2 API Description

```
generatePublicPrivateKeyPair(persist ,filePath ,passphrase)
Generates a public key and private key pair that can be used to encrypt and
decrypt data.
persist  (boolean) : Should the key pair be persisted on the disk. Defaults to
False.
filePath (string) : The file path where the keypair should be  persisted.
Defaults to key.pem.
passphrase(string) : Passphrase for the keypair.
returns (tuple): (publickey, privatekey)
```

```
verifySignature(payload, signature, publicKey)
Verifies that the payload and signature match each other using the publicKey
Payload (bytes) : The payload whose signature is to be verified
signature (bytes) : The signature of the payload generated by the CA
publicKey (bytes) : publicKey of the CA with which the signature is generated
returns (bool): True if signature verified, False otherwise
```

```
extractNodeIdFromCAResponse(ciphertext, privateKey)
Returns the NodeID present in the CA response passed to it
ciphertext (bytes) : The encrypted data received from
the Certification Authority.
privateKey (bytes) : The key used to decrypt the ciphertext
```

```
returns (int) : NodeID in the CA response passed as ciphertext
```

```
getNodeIdFromCA(nodeID, nodePublicKey)
Returns a certificate  for the node given the proposed node s ID
and public key
nodeID (string) : Proposed node ID
nodePublicKey (bytes) : Public key of the node
returns (bytes) : Encrypted nodeID
```

```
getRedundantRoutes(candidates)
Performs redundant routing via the prospective set to the destination node.
candidates (set) : last m/2 entries in the finger table
returns (list) : list of genuine nodes which constitute the redundant route.
```

2.3 System Architecture

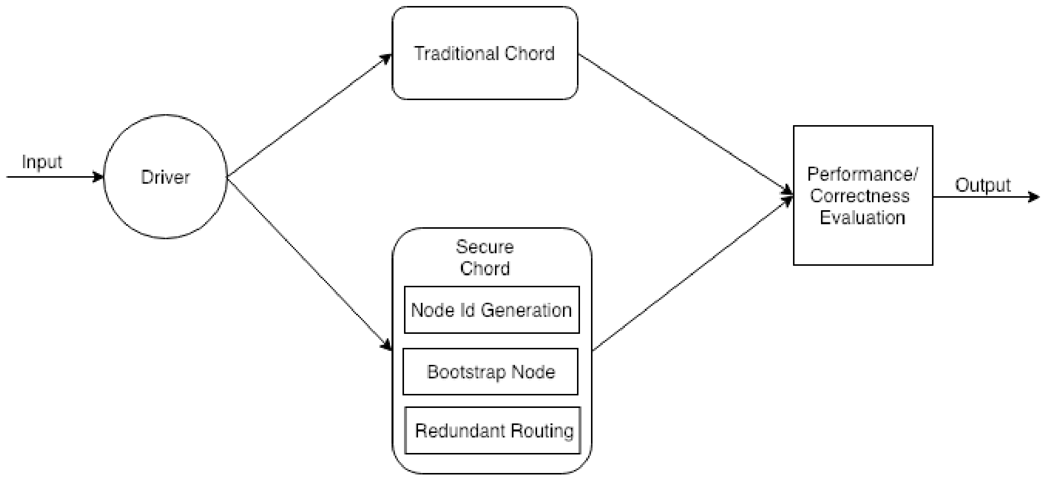


Figure 1: High-level architecture.

2.3.1 NodeID generation

In the original chord the nodes are responsible for generating nodeId. But this means that the attackers can choose nodeIds and control the network. [1] discusses a secure way to assign nodeIds by having a central certificate authority grant the certificates to nodes.

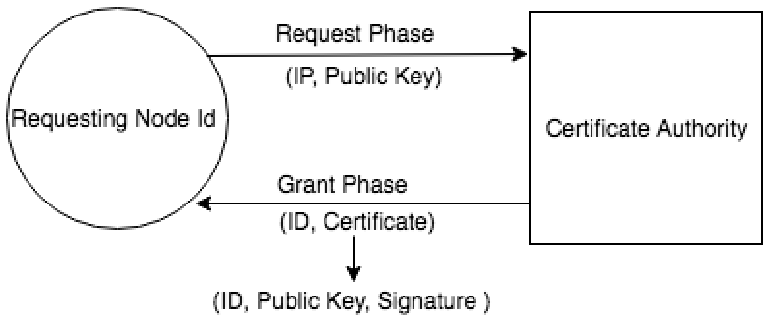


Figure 2: High-level NodeID generation design.

- **Request phase:** The node requests a nodeId from a CA sending its IP address and its Public Key.
- **Grant Phase:** The CA chooses nodeId randomly from the id space, to prevent nodes from forging nodeIds. It then responds to the client with a digest comprising of nodeId, public key of the node and a signature signed by its private key.

The client upon receiving the digest verifies the signature using the public key of the CA and then extracts the nodeId.

2.3.2 Secure routing table maintenance

The routing table maintenance mechanism is used to create routing tables and neighbor sets for joining nodes, and maintain them after creation. But attackers (if present) can increase the fraction of bad entries by supplying bad routing updates, which reduces the probability of routing successfully to replica nodes. Paper [3] describes a technique to tackle such issue.

- **Bootstrap set:** Algorithm maintains a set of bootstrap nodes to initialize the routing state of a newly joining node.
- **Verify node signature:** Set of bootstrap nodes verify the identity of the newly joining node using its signature.

A newly joining node, n, picks a set of bootstrap nodes and asks all of them to route using its nodeId as the key. These bootstrap nodes use secure forwarding techniques to obtain the neighbor set for the joining node. Node n collects the proposed neighbor sets from each of the bootstrap nodes, and picks the closest live nodeIds from each proposed set to be its neighbor set.

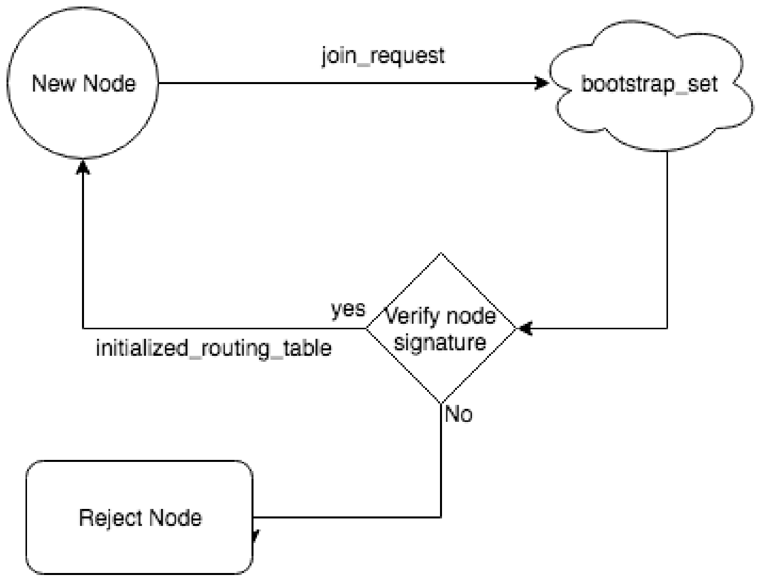


Figure 3: High-level routing table maintenance design.

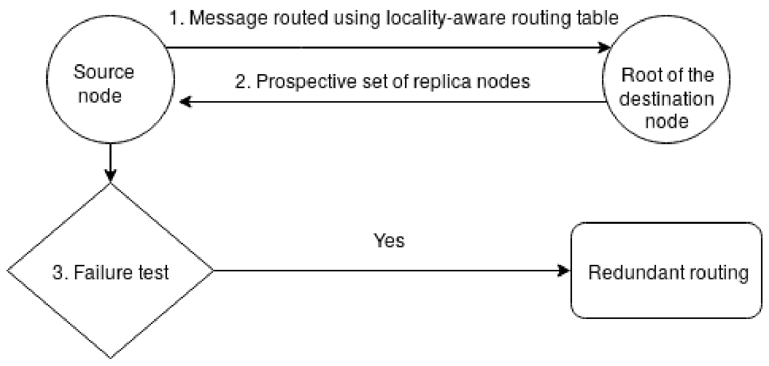

1. Message routed using locality-aware routing table
Source node
2. Prospective set of replica nodes
Root of the destination node
3. Failure test
Yes
Redundant routing

Figure 4: High-level Secure message forwarding design.

2.3.3 Secure message forwarding

Secure message forwarding consists of two steps. Firstly, the source node attempts to send the message to the keys root node. The root node responds with a set of prospective replicas. This is followed by a failure test which is used to establish whether the message has been securely forwarded to the desired nodes or not. If the test is negative, the message has been delivered securely to the destined nodes. In case the failure test returns positive then redundant routing is employed. In redundant routing the message copies are sent over diverse routes towards the various replica roots such that with high probability each correct replica root is reached.

3 Implementation

There were several stages in the implementation and we faced challenges of varying degree in each one of them. The implementation stages can be broadly split into the following:

- Finding the best working implementation of chord.
- Modifying the implementation to suit our needs.
- Introducing malicious nodes and implementing security fixes.
- Performance evaluation and comparison of the secure chord with the original one.

3.1 Finding the best working implementation of chord

We started our project by surveying the existing implementations of Chord in DistAlgo. We explored four implementations for the same [5–8]. We used [8] as our baseline which is based on [7] - the one suggested by Professor Liu.

3.2 Modifying the implementation

We found this stage particularly challenging requiring more than 10 hours of our effort. The implementation which we adopted as the baseline was run on DistAlgo v0.6. DistAlgo has significantly matured since the time requiring considerable code changes. Some of the main changes needed to make the code run were:

- Adding receive handlers.
- Changing send statements.
- Modifying await statements.
- Other minor syntax changes.

3.3 Introducing malicious nodes and implementing security fixes

The paper [1], that we chose to use as the basis of our project, discusses security fixes with Pastry DHT protocol as the reference protocol. Hence, we had to adopt the techniques applied to pastry protocol to suit the chord protocol. Moreover, the paper doesn't provide precise pseudo code for the fixes that it describes. We had to translate the fixes detailed in text into code. This part of the project took about 15 hours of development effort.
We introduced 25% malicious nodes in the system as discussed in [1]. We then proceeded to address the security fixes.

3.3.1 Certificate Authority

As we described in 2.3.1, implementation of nodeID generation requires creation of driver CA process to authenticate the node joining the ring.

```
    serializedPublicKey = serializePublicKey(publicKey)
    -- requestcertificate
    output(proc_id, serializedPublicKey)
    send(('request_certificate', proc_id, serializedPublicKey),
    to=certificateAuthority)
    -- awaitcaresponse
    await(some(received(('issue_certificate',certificate))))
```

3.3.2 Bootstrap Node

Bootstrap Node is the secure node which facilitates the other nodes to join the chord ring.

```
    --awaitjoinmsg
    await(some(received(('join_ring',bootstrap_node,proc_id))))
    init_node(bootstrap_node, proc_id)
```

3.3.3 Redundant Routing

As described in 2.3.3, we achieve secure message forwarding by sending messages to the destination node through diverse nodes. Unlike in Pastry we do not have replicas in chord. Hence we had to adopt the steps mentioned in the paper [1] to work with chord. We chose to forego failure test and directly implemented redundant routing. Pastry achieves redundant routing by routing through it's neighbour set. We employed the bottom half of finger table entries which correspond to the key to achieve this. Below is the code snippet that shows how we achieve this by sending the messages to possible candidates whose signatures are verified before forwarding the messages.

Note : The identifiers have been modified in the code below for better readability.

```
send(('verifyNodeId'), to=candidates)
if await(len(setof((a, b, c, d),
    received(('verifiedNodeId',a,b, c, d)))) == len(candidates)/2):
    nodeIdSignatures = setof((a, b, c, d),(('verifiedNodeId',a, b, c, d),_)
                                                                  in received)
    genuineNodes = []
    for (nodePublicKey, nodeIdSignature, processId, nodeId)
                                                    in nodeIdSignatures:
        message = encodeData([nodeId,nodePublicKey])
        if verifySignature(nodeIdSignature,message):
            genuineNodes.append(processId)
        else:
            output("Failed to verifySignature")
    send(('find_key', key, node_id, org, hop_count + 1), to=genuineNodes)
elif timeout(5):
    output("Timeout waiting for half the candidates.")
```

All of our code implementation can be found in our project repository [10].

4 Testing and Evaluation

4.1 Experimental Setup

All the metrics reported in this report are run on a setup with the following configuration.
Platform: Mac OS 10.14.
RAM: 8GB
CPU: Intel i5 2.6 GHz
Dependencies: DistAlgo 3.6.5 and python 3.7

4.2 Correctness testing

The implemented program was found to be correct by inserting a (key, value) pair, then searching for the value by supplying the previous key. It was found that in both the original chord and secure chord, the driver got the expected value. Apart from this we can also verify whether at least one ring and at most one ring exists in the chord. We also verified that finger table is correctly populated once the algorithm stabilizes.

- Simple insertion test: We wait for the nodes to stabilize so that they can correctly update their successors and predecessor. We then insert a value into the ring and verify that the key-value store in the corresponding node is correctly updated.

- Simple lookup test: We insert a key-value into the chord ring. We then perform a lookup of this key and verify that the value is successfully retrieved.

4.3 Performance evaluation

File name	Lines of code
chord.da (Original chord)	272
secure_chord.da (Secure chord)	314
crypto_utils.py (Secure chord)	106
certificate_authority.da (Secure chord)	14

Table 1: Lines of code comparison

Node Number	Secure Chord	Original Chord
1	2.05ms	0.11ms
2	2.43ms	0.126ms
3	2.71ms	0.009ms

Table 2: Performance Comparison of two chord versions for Node Join Procedure

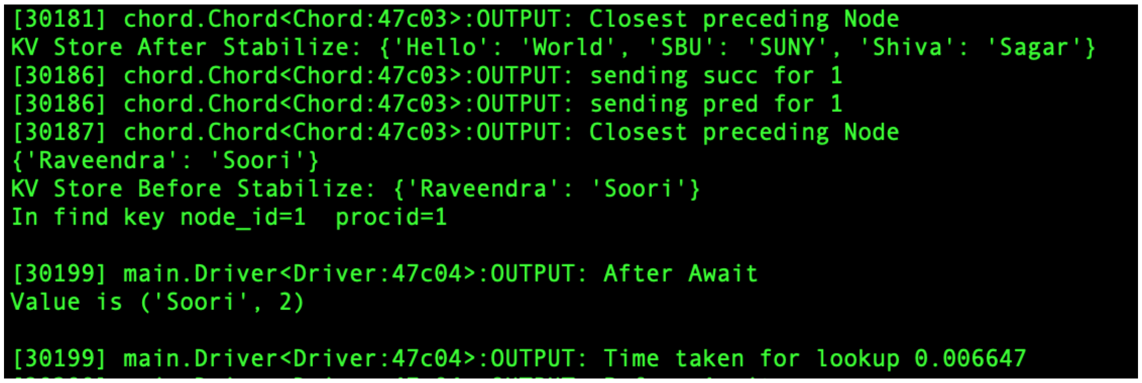
Network Size	Secure Chord	Original Chord
2	1.9ms	1.1ms
4	4.83ms	4.28ms
8	7.71ms	6.6ms

Table 3: Performance Comparison of two chord versions for Key Lookup

Table 1 shows the LOC count of the files for original chord as well as secure chord. Original chord code is 272 lines while the secure chord code is 434 lines (Aggregate of all files corresponding to secure chord).

From the results of table 2, time taken by the node to join a network is more in case of secure chord than original chord. This is expected since the certificate authority verifies the authenticity of every node joining the network and ensures security at the node joining level. Additional two message exchanges add the overhead that causes the time difference.

Table 3 tabulates the time taken by any node in a network for key look up. Again, Secure chord takes more time due to redundant routing.


```
[30181] chord.Chord<Chord:47c03>:OUTPUT: Closest preceding Node
KV Store After Stabilize: {'Hello': 'World', 'SBU': 'SUNY', 'Shiva': 'Sagar'}
[30186] chord.Chord<Chord:47c03>:OUTPUT: sending succ for 1
[30186] chord.Chord<Chord:47c03>:OUTPUT: sending pred for 1
[30187] chord.Chord<Chord:47c03>:OUTPUT: Closest preceding Node
{'Raveendra': 'Soori'}
KV Store Before Stabilize: {'Raveendra': 'Soori'}
In find key node_id=1  procid=1

[30199] main.Driver<Driver:47c04>:OUTPUT: After Await
Value is ('Soori', 2)

[30199] main.Driver<Driver:47c04>:OUTPUT: Time taken for lookup 0.006647
```

Figure 5: Sample output of the program

5 Discussion and Future Work

- Graceful termination of the program - We were unable to gracefully terminate the program. There is no terminating condition for the program unless *kill_node* message is sent to the node. Since the *Stabilize* and *fix_fingers* methods run in an infinite loop a node might be in middle of a *find_successor* or a *find_predecessor* call at any point in time. So killing one such nodes results in a *Transportation Exception*. This is something which we spent significant time fixing but was not successful.
- High number of nodes - For any value $m > 3$ i.e $networksize > 8$ the algorithm fails to terminate.
- Failure test - Currently the redundant routing occurs for for lookup query. This understandably causes an overhead on the network. By implementing Failure test this can be can made more efficient.

6 Acknowledgement

We would like to thank Professor Y. Annie Liu for her constant support and encouragement. We would also like to thank our Teaching Assistant, Christopher Kane for helping us whenever we were stuck with our project.

7 Conclusion

This project contributed in providing a secure chord as per the security methods discussed in the paper [1] implemented in DistAlgo. We also analyze the performance of implemented secure chord with the existing chord. From the results we obtained it is clear that the secure chord is less performant than the original chord as expected.

References

[1] Ayalvadi Ganesh Antony Rowstron Miguel Castro1, Peter Druschel and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. *Usenix Symposium on Operating Systems Design and Implementation, Boston, MA*, 2002.

[2] Keith Needels Minseok Kwon. Secure Routing in Peer-to-Peer Distributed Hash Tables. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.879.8892rep=rep1type=pdf*, 2009.

[3] Y. Charlie H Antony Rowstro Miguel Castro, Peter Druschel. Exploiting network proximity in peer-to-peer overlay networks. *ACM SIGCOMM Computer Communication Review 31 (4), 149-160*, 2002.

[4] David Liben-Nowell David R. Karger M. Frans Kaashoek Frank Dabek Hari Balakrishnan Ion Stoica, Robert Morris. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *Technical Report MSR-TR-2002-82, Microsoft Research 22, 23-24*, 2001.

[5] Youlong Cheng. Chord-DistAlgo. *https://bitbucket.org/toponado/chord-distalgo/*, 2012.

[6] Soumyadeep. Chord-DistAlgo. *https://github.com/soumyadeep2007/dns_chord*, 2018.

[7] Sourabh Yerfule. Chord-DistAlgo. *https://github.com/unicomputing/chord-distalgo-2013-Sourabh-Yerfule*, 2013.

[8] ChidhambaramR. DistAlgo Homepage. *https://github.com/ChidambaramR/Asynchronous-Systems/*, 2018.

[9] Chord (peer-to-peer). *https://en.wikipedia.org/wiki/Chord*$_($peer − to − peer$)$, 2018.

[10] Raveendra Soori Shivasagar Boraiah, Varun Hegde. sec-routing-chord-distalgo. *https://github.com/unicomputing/sec-routing-chord-distalgo/*, 2018.

[11] Annie Y. Liu. DistAlgo Homepage. *https://github.com/DistAlgo/distalgo*, 2018.