Saarland University

Faculty of Natural Sciences and Technology I

Department of Computer Science

Masters Thesis

Experimental Comparison of Byzantine Fault Tolerant Distributed Hash Tables

submitted by

Supriti Singh

submitted

01.09.2014

Supervisor:

Dr. Christian Grothoff

Advisors:

Bartlomiej Polot

Sree Harsha Totakura

Reviewers:

Dr. Christian Grothoff

Prof. Dr. Sebastian Hack

Acknowledgements

I am fortunate to have received support from so many people in accomplishing this thesis. I take this chance to express my gratitude.

I express deepest regards and respect to my thesis supervisor, Dr. Christian Grothoff for providing me this opportunity to work on this topic. I would also like to thank my thesis advisor Prof. Sebastian Hack, Saarland University.

Many thanks to my advisors, Bartlomiej Polot and Sree Harsha Totakura. Bart helped me in understanding the existing DHT, R^5N architecture. Harsha helped me with the experimental setup and explanation of GNUnet testbed.

I would also like to thank my family and friends who were a constant support throughout.

Abstract

Masters Thesis

Experimental Comparison of Byzantine Fault Tolerant Distributed Hash Tables

by Supriti Singh

Distributed Hash Tables (DHTs) are a key data structure for construction of a peer to peer systems. They provide an efficient way to distribute the storage and retrieval of key-data pairs among the participating peers. DHTs should be scalable, robust against churn and resilient to attacks. X-Vine is a DHT protocol which offers security against Sybil attacks. All communication among peers is performed over social network links, with the presumption that a friend can be trusted. This trust can be extended to a friend of a friend. It uses the tested Chord Ring topology as an overlay, which has been proven to be scalable and robust.

The aim of the thesis is to experimentally compare two DHTs, R^5N and X-Vine. GNUnet is a free software secure peer to peer framework, which uses R^5N. In this thesis, we have presented the implementation of X-Vine on GNUnet, and compared the performance of R^5N and X-Vine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Distributed Hash Table

A Distributed Hash Table (DHT) is a decentralized storage structure for a peer to peer (P2P) system. It provides storage and lookup of key-data pair similar to a hash table. Each data to be stored in DHT is associated with a key. The key-data pair is stored at a particular peer to which the key maps. DHTs offer simple APIs, PUT and GET to store and retrieve the data from the P2P system.

1.2 General DHT architecture

A typical DHT consists of a storage layer on top of a lookup protocol [1]. A lookup protocol ensures that peers connect in an overlay topology such that they are reachable to each other. It also ensures that peers decide on some mapping to store key/data pair at the correct destination peer. Storage layer provides a way to store and retrieve the data from the DHT.

Lookup protocol of a typical DHT has the following components.

1. A peer identifier space.
2. A key identifier space.
3. A rule to map keys to peers.
4. Per-peer routing tables, which stores information to reach to other peers.
5. Rules to update routing table in case of peer joining, leaving or failing.

The storage layer provides APIs, PUT and GET to access the data from the DHT. Storage layer can also provide features like replication and data authentication.

1.3 DHT Properties

A DHT should have following features.

1. Load balancing
   Distribution of keys among the participating peers should be balanced. In case there is a skewed distribution, few peers store large amount of data. If such peer becomes unavailable due to churn, network will be destabilized. Also, in case this peer is malicious, it has access to large amount of information which is undesirable. Hence, DHT should use a mapping function which equally distributes key among the participating peers.

2. Scalability
   A DHT should be able to scale and operate under arbitrary changes in the network size. To handle it, lookup protocol should be able to update routing table of nodes in accordance to the changes.

3. Fault tolerance
   A DHT should be able to operate even with nodes continuously leaving, failing or with malicious participants. There can be multiple kinds of malicious behaviors and resulting failures. They are discussed in Section 1.4 along with the general defense techniques.

1.4 Byzantine Failures and Defenses in DHT

In a P2P systems with byzantine failures, participating peers fail in arbitrary ways. A non byzantine fault tolerant system responds in an unpredictable way in event of a byzantine failure.

In a DHT, participating peers can be either malicious or honest. Even if peers are not malicious, they may crash or fail to send and receive messages from the other peers. This failure can result in loss of data. DHT uses replication in order to provide service even in case of such failures.

In case there are malicious peers, following attacks are possible on DHT [1].

1. Attack on lookup protocol.
   Lookup protocol ensures that each peer maintains a routing table with information to reach to other peers in network. It also ensures that routing table are updated in case of nodes leaving and joining the system. An adversary can be part of routing tables of honest nodes. It can intercept all its queries or drop the packets. It can route lookup request to a non-existent or to an incorrect node. As the malicious peer participates in routing table maintenance, it will appear as alive to all other peers. Hence, a retransmission of the lookup request will again go through the same malicious peer.

   One possible defense again such attack would be that querying peer tries to reach to the destination peer through different routes, instead of one fixed route. This approach will increase the chances of taking a route with no malicious peers.

2. Storage and retrieval attack
   An adversary can store data, but can deny access to data. Similarly, it can claim to be responsible for a data, but refuses to serve the client.

   DHTs use replication in order to avoid such single point of responsibility. Also, clients of DHTs are expected to contact all the peers that hold replicas.

3. Overload a specific peer with data.
   An adversary can attack a particular peer in the network, by generating lot of data and storing it at the target peer. Eventually, the target peer will get over-loaded and the system will treat the affected peer as failed and will run stabilization algorithm to handle it. This attack is a typical Denial of Service attack.

   To overcome such attacks, DHT should use replication. Even if a peer is unreachable, the other replicas should be accessible. But if malicious peer is one of the replicas or it colludes with other replicas, the attack may be succeed.

   Also, DHT should do node assignment in a random way that does not adhere to actual physical topology. Replicas should be placed at different physical location. This would prevent a localized attack from preventing access to a specific key.

4. Unsolicited messages
   Consider a scenario where Alice asks for reply from Bob . The path between them goes through Chuck. Chuck can send back an unsolicited reply. Peer Authentication can be used to avoid such forgery attacks.

5. Rapid peer join and leave
   Every DHT runs stabilization algorithm to handle peer churn. Key-data pair is transferred to the correct peer so that the lookup returns a valid result.

   A malicious node might try to convince the system that some new peer has joined or left the system. This can result in control traffic, leading to reduced efficiency and poor performance of the system.

   This attack would work best if attacker can avoid being a part of the data transfer. But in an adversary model, where malicious peer has limited resources, it is not possible. The DHT should be able to provide data transfer in a way that does not result in network overload.

1.5 Motivation

This thesis aims to experimentally compare two byzantine fault tolerant DHTs, R^5N and X-Vine. X-Vine and R^5N uses different approaches to mitigate security attacks. While R^5N uses randomized paths for lookup, X-Vine follows the social network trail.

R^5N is currently used in GNUnet. GNUnet is a secure open source peer to peer framework. In order to compare X-Vine and R^5N, X-Vine has been implemented in GNUnet. Performance comparison is done on various parameters, such as PUT/GET success rate, network traffic generated and average number of hops for PUT/GET.

Chapter 2

X-Vine

X-Vine is a social network based Byzantine fault tolerant Distributed Hash Table (DHT). It uses social network links for all the communication among the peers. It uses Chord DHT as its base protocol.

This chapter explains the basic architecture of X-Vine. Section 2.1 discusses Chord, the base DHT protocol of X-Vine. Section 2.2 explains how X-Vine builds upon Chord, and discusses its byzantine fault tolerant features.

2.1 Chord

Chord [2] is a basic DHT protocol, that has been proven to be robust and scalable. However, it is not secure against attacks [3].

2.1.1 Key and Peer mapping

Chord uses consistent hashing to assign keys to peers. Consistent Hashing uses a base hash function such as SHA-1 to assign m-bit identifiers to peers and keys. m should be large enough such that probability of two different peers having the same identifier is negligible. Consistent hash function balances the load among all the participating peers with high probability. In a Chord DHT with n peers, each peer is responsible for only K/n keys, where K is the total number of keys.

All the peers in Chord are ordered in an identifier circle modulo 2^m, where $m =$ number of bits in the identifier space. Key, k is assigned to a peer, *successor(k)* such that peer's identifier is either equal to or follows k in the identifier space.

Figure 2.1 shows an identifier circle with $m = 3$. The network can have at most 8 peers. Peer 0, 3 and 5 are present in the network. Successor of identifier 3 is peer 3. Successor of identifier 4 is peer 5 and successor of identifer 6 is peer 0.

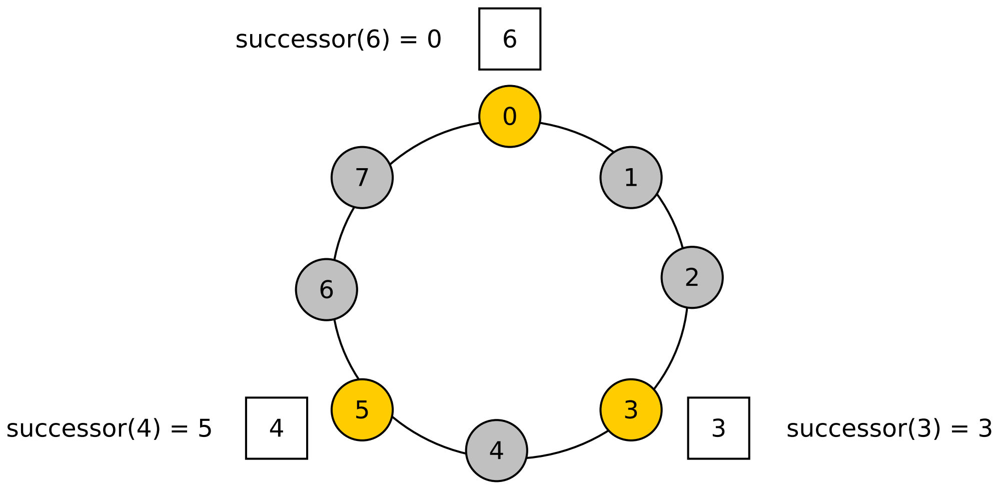


FIGURE 2.1: Peers and their successor nodes.

2.1.2 Finger Table

In Chord, each peer maintains pointer to its immediate successor in the network. A naive way to do lookup in the network will be to follow these successor pointers. But this scheme requires $O(n)$ messages.

To do lookup more efficiently, each peer maintains a set of *fingers* in the network that are spaced exponentially away from it. These *fingers* are stored in a *finger table*. A peer maintains at most m *finger* entries, where m is number of bits in the identifier space. The i^{th} *finger* of node n is the first node, s, that succeeds n by atleast 2^{i-1}, where $1 <= i <= m$.

Figure 2.2 shows the finger tables of peers 0, 3 and 5.

2.1.3 Concurrent Node Joins and Stabilization

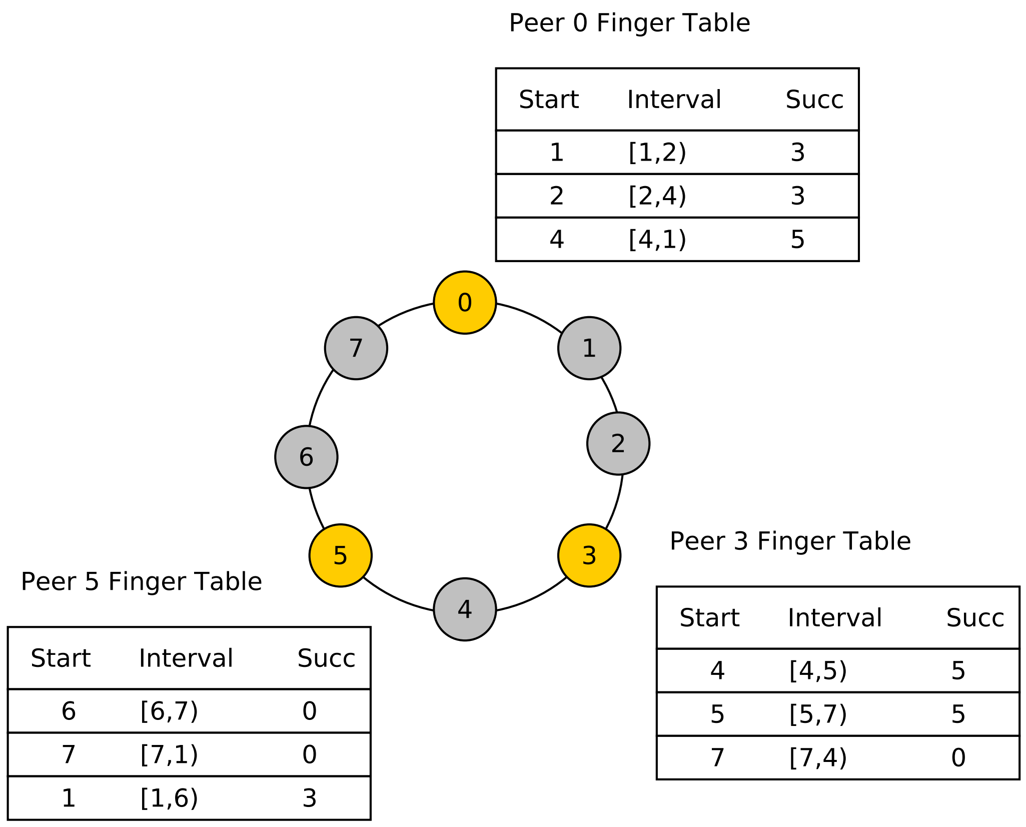
In a peer-to-peer systems, peers can arbitrarily join, leave or fail. In such events, Chord tries to maintain the following invariants:

- Each peer's successor is correctly maintained.
- Each key, k is stored at peer, *successor(k)*

As an example, consider a new node, n_p joining the system. It contacts its immediate successor, n_s in the network. Each peer should also maintain a pointer to its immediate predecessor. Hence, n_s adds n_p as its predecessor.

Another new node, n joins the system, such that, $n\_p < n < n\_s$. The new node, n and n_s add each other as their successor and predecessor respectively. Node, n_p inquires n_s about its current predecessor. Eventually n_p will add n as its successor.

Each peer also periodically verifies its existing fingers. In case there are changes in the network, the finger table will be updated.



Peer 0 Finger Table

Start	Interval	Succ
1	[1,2)	3
2	[2,4)	3
4	[4,1)	5

Peer 3 Finger Table

Start	Interval	Succ
4	[4,5)	5
5	[5,7)	5
7	[7,4)	0

Peer 5 Finger Table

Start	Interval	Succ
6	[6,7)	0
7	[7,1)	0
1	[1,6)	3

FIGURE 2.2: Finger table

2.1.4 Node failures

In case of node failures, Chord tries to maintain the following invariant:

- When a node, n fails, then all the peers which used n as its finger, must update their finger tables. They should replace n with its successor.
- Failure of a peer must not disrupt the queries which are in progress.

In order to keep successor pointers correct in case of node failures, each Chord peer maintains a list of consecutive 'r' successors. In case one of the successors fails, then the next live successor is selected.

2.1.5 Performance

In an n-peer system, each peer in Chord maintains information about $O(\log n)$ other peers. Data lookup requires $O(\log n)$ messages. In case of peers leaving or joining the network, $O(\log n^2)$ messages are exchanged to stabilize the network.

2.2 X-Vine Architecture

This section describes X-Vine protocol. Section 2.2.1 explain the malicious behavior for which X-Vine provides protection. Section 2.2.2 explains the intuition for using social network links for security. Section 2.2.3 explains routing algorithm of X-Vine. Section 2.2.4 gives a detailed explanation of X-Vine security protocol [4].

2.2.1 Adversary Model

X-Vine assumes that a set of peers are colluding and compromised. The malicious peers are byzantine in nature. X-Vine focuses in particular on Sybil attacks. In Sybil attacks, a single entity creates multiple identities. This entity is referred as Sybil node. It uses these identities to gain disproportionately large influence and launch attacks as explained in Section 1.4.

2.2.2 X-Vine Trust Model

As seen in section 1.4, a peer-to-peer system is prone to security attack due to presence of malicious nodes in the system.

X-Vine is a social network based DHT. It works on assumption that a friend peer is unlikely to be selfish or malicious. The same trust can be extended to friend of a friend. Hence, to avoid mis-routing of messages, routing is done through a trail of friend peers [5].

2.2.2.1 Intuition

Figure 2.3 shows a social network with set of honest and Sybil nodes. An edge between honest and Sybil sub graph is referred as an *attack edge.* An honest node adds other honest nodes as its friends. Sybil nodes can also setup trust relation with honest nodes. Honest and Sybil areas of network can not be differentiated. Previous work [6][7] have shown that it is costly for an adversary to set up a large number of trust links. Hence, the number of edges within an individual region will always be greater than the attack edges.

Sybil nodes tries to disrupt the network by mis-routing. Consider an honest node, Alice which wants to contact another honest node, Bob. Alice does not have a direct path to reach to Bob, but she can contact Bob through a series of friends. Alice have three friends, Chuck, Dan and Dave. Chuck is a Sybil node. But, Alice can not distinguish between its real friends and adversary. If Alice tries to reach to Bob only through Chuck, then Bob will never receive the message. In order to avoid such a situation, Alice will try to contact Bob through its other two friends. As attack edges are limited, Alice will eventually reach to Bob [8].

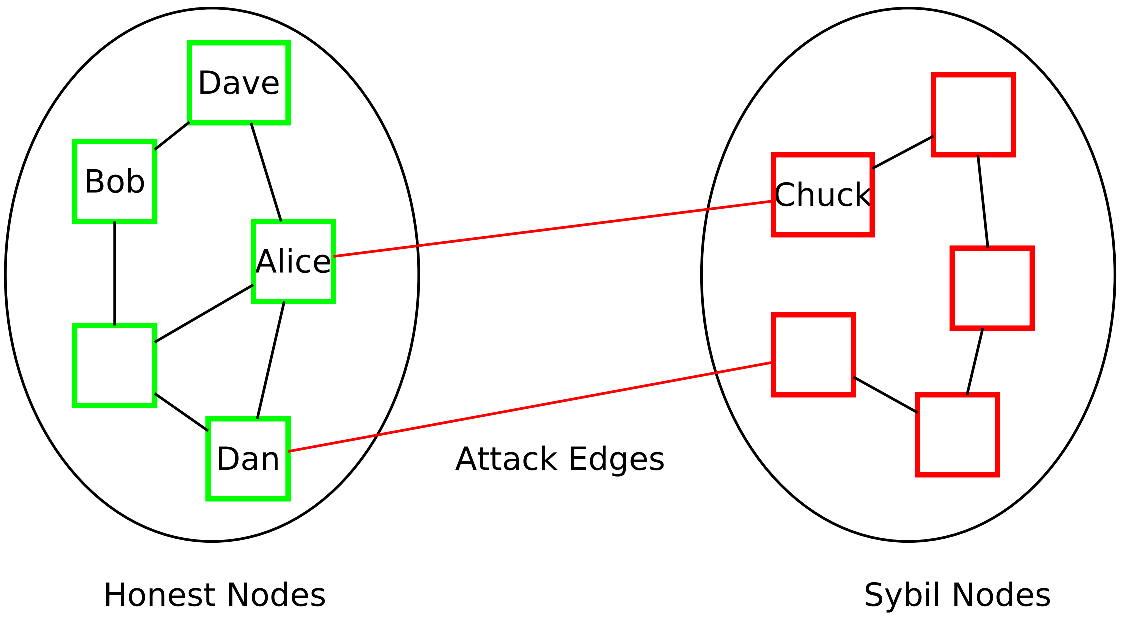


FIGURE 2.3: Social network

2.2.3 Friends and Overlay Endpoints/Fingers

X-Vine builds an overlay network over the social network underlay topology. In this overlay network, a node has direct connection to its *friend*. A node also maintains *overlay links* to its *overlay endpoints*. These *overlay endpoints* are selected in manner similar to *fingers* in Chord.

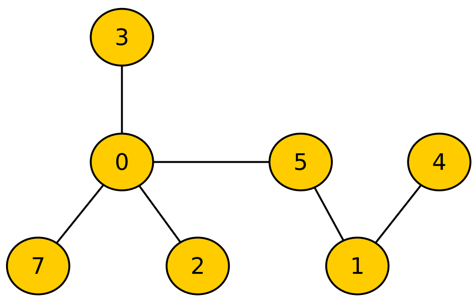
Figure 2.4 shows a social network and its corresponding X-Vine overlay network.

Similar to Chord, X-Vine do PUT and GET at the *key*'s *successor* node. However, nodes in X-Vine can not directly communicate with its *overlay endpoints*. All the communication must use social network links. Hence, a node needs to setup a *trail* through social network to reach to its *overlay endpoints*. All the nodes which are part of this trail should store an entry in their *routing table*.

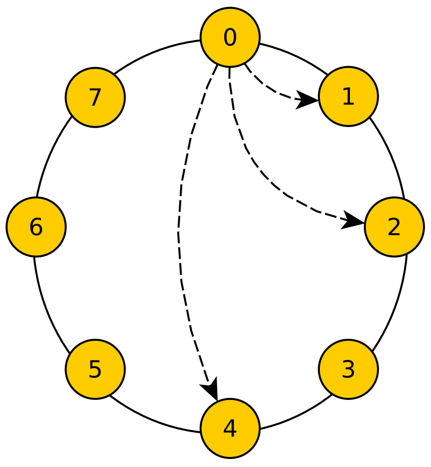
Figure 2.5 shows a trail from a peer, 0 to its *overlay endpoint*, 4 . The *trail* goes from 0-3-1-4 and each peer stores an entry in its *routing table*.

2.2.4 Byzantine Fault tolerance

In order to limit Sybil attacks on both routing table maintenance and lookup protocol, X-Vine provides a set of local and global policies. These policies ensures that no node accumulates a large state.

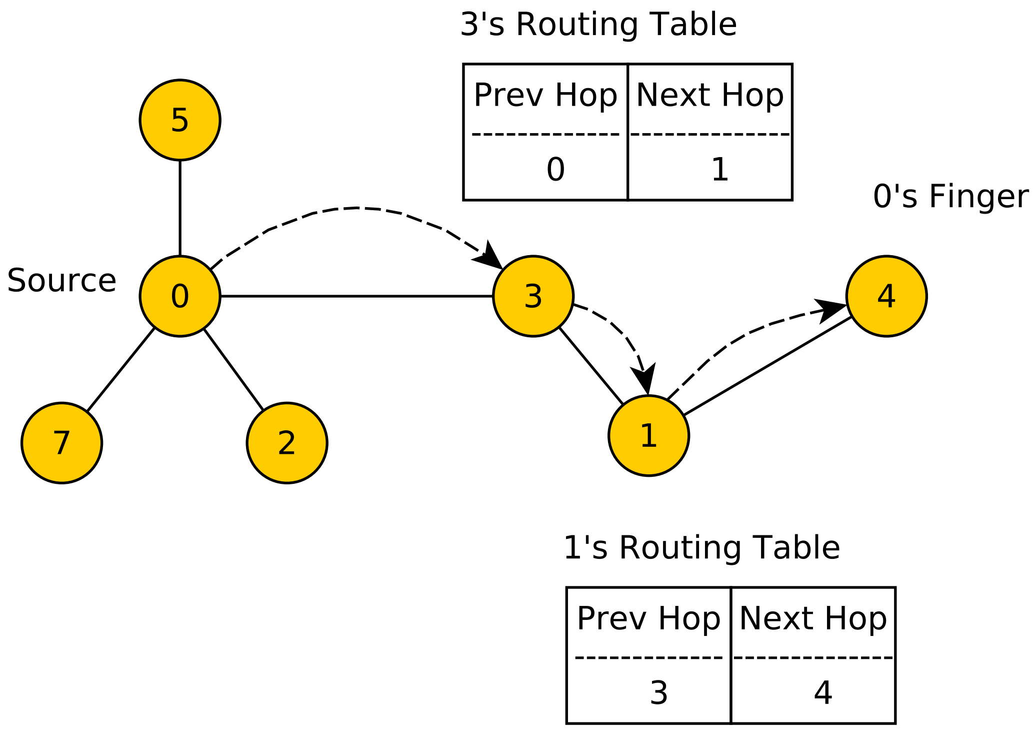


(A) Underlay topology



(B) Chord ring overlay topology. Dotted lines shows Peer 0's fingers.

FIGURE 2.4: Social network and its corresponding overlay network,



3's Routing Table

Prev Hop	Next Hop
0	1

1's Routing Table

Prev Hop	Next Hop
3	4

FIGURE 2.5: Trail from 0 to its finger, 4 through the overlay network. Peer 3 and 1 store an entry in their respective routing tables.

2.2.4.1 Global Policies

A joining node needs to setup *trails* to its *overlay endpoints* in the network. But as the new node has no routing state, it uses one of its existing friend in the network as a bootstrap node. Thus, an earlier joined node will be part of significantly more trails. To avoid such accumulation of state at few nodes, X-Vine defines the following policies:

1. Threshold on trail length.
   X-Vine defines a global threshold thr_1 on *trail* length. When a new node joins the network, an existing node scans all the *trail* that might have crossed thr_1. It contacts the existing social contacts of new node to find a shorter alternative *trails.*
2. Threshold on routing table size.
   X-Vine defines a global threshold thr_2 on routing table size. This threshold limits the number of trails of which a peer can be part of. Thus, it avoids a single peer to accumulate large state. A peer whose routing table size has crossed, thr_2 scans its routing table for the *trails* such that their length has crossed thr_1. It contacts the *overlay endpoints* of all the such *trail* to check if an alternate shorter *trail* can be established.

2.2.4.2 Local Policies

In order to account for heterogeneity of the participating peers, X-Vine allows individual peers to make local decision regarding their participation in routing and lookup. Based on its available resources, a peer decides the following thresholds.

1. Bounding routes per friend link.
   A peer can setup a threshold, b_l to limit the number of trails that should be setup through an adjacent social link. X-Vine assumes that there are limited number of attack edges. Hence, b_l bounds the number of overlay path between Sybil subgraph and honest subgraph, irrespective of adversary's attack strategy.
2. Bounding routes per node.
   A peer can setup a threshold, b_n on its own routing table size. If there are maximum b_n trails already stored in a peer's routing table, it can deny to be part of any more *trails.* By limiting its own routing size, a peer can avoid clustering attack.

2.2.4.3 Redundant Routing

An adversary can drop or route PUT/GET requests to a non existent or a wrong node. To limit such attacks, X-Vine does lookups over r different trusted trails (*redundant routes*). If at least one of the route consists of only honest nodes, the lookup will succeed.

Chapter 3

R^5N DHT

R^5N stands for Randomized Recursive Routing for Restricted-Route Networks [9]. It is a secure DHT that uses randomized routes for secure lookup and routing. Section 3.1 lists down the main features of R^5N DHT. Section 3.2 explains R^5N lookup and routing protocol. It also discusses its security protocol. Section 3.3 gives a brief comparison of X-Vine and R^5N architecture.

3.1 Key Features

R^5N design has the following key features:

1. Restricted Route Underlay Topology
   Restricted route topology refers to an underlay network infrastructure where not every peer can directly communicate with other peers. For example, in a friend to friend network a peer can setup direct connections only with the set of peers that it already knows as friends. But, most of the DHTs assume a universal connectivity among the participating peers. This assumption results in limited participation. R^5N routing algorithm works efficiently even in a restricted route underlay topology.

2. Recursive Kademlia
   R^5N uses a recursive version of Kademlia DHT [10]. Section 3.2.1 explains Kademlia's routing tables and iterative routing. In recursive routing, the initiator peer forwards the lookup/routing request to another peer. The querying/initiator peer is contacted again if a response expected by it. Initiator peer does not have control over the algorithm.

3.2 Design

3.2.1 Kademlia

Kademlia DHT has been shown to work well with common rates of churn [11] and capable of handling large number of peers [12]. But Kademlia is vulnerable to Sybil [13] attacks.

Kademlia's routing table is structured as an array of *k-buckets*. Here, k is the number of bits in the identifier space. The j^{th} $k - bucket$ stores upto k peers, whose identifiers are between distance 2^j and $2^{(j+1)}$ from the local peer. Distance between two peer IDs are calculated using XOR.

Kademlia lookup algorithm is iterative. In iterative algorithm, the initiating peer directly contacts the next hop, collects information and iterate the process until it reaches the destination hop. Hence, the iterator has direct control over the whole lookup algorithm.

In Kademlia, at each hop the intiating node picks r closest peers for the next hop. These r peers are queried and they return a set of peers closer to the key. The algorithm stops when no more closer peers are found. The data is finally stored at the final r closest peers. Kademlia contacts $O(\log n)$ peers during the search in n peer system.

3.2.2 R^5N Design

3.2.2.1 Routing

Routing table in R^5N is constructed similar to Kademlia. But it uses a recursive routing. Routing is completed in two phases. In first phase, the message is forwarded to some number of hops. When the hop counter exceeds a threshold, $T = logn$, where n is the network size, routing starts its second phase. In phase two, routing is done deterministically through the set of peers from the routing table that are closest to the target key. The first phase of routing makes the lookup independent of the initiator's location, and the second phase finds the nearest peer.

3.2.2.2 PUT/GET

In R^5N, data to be stored is replicated at large number of peers that are closer to the key. A PUT operation stores the data at a random subset of these peers, and GET attempts to get the data from one of these replicas. The PUT requests are repeated at regular interval to handle churn and increase replication. Since R^5N follows non-deterministic routing, repeated PUT may store data at different peers. These replicated PUTs increase chance of a successful GET. R^5N also retries GET request few times to increase the chance of reaching to the correct data.

3.2.2.3 Security

R^5N derives its security through the non-deterministic routing. As the GET and PUT requests take different route each time, it is difficult for an adversary to position itself to eclipse a particular key-data pair. Also, R^5N two phase routing algorithm acts as defense against Denial of Service attack. The number of hops a request can travel is bounded by the threshold, T. This limits the flooding of message by an adversary.

3.3 X-Vine and R^5N

X-Vine and R^5N are secure DHTs. X-Vine uses social network links, but R^5N uses non-deterministic routing for security. Similar to R^5N, X-Vine also do not make assumption about universal connectivity. Peers in X-Vine can communicate only with their immediate friends.

Chapter 4

X-Vine GNUnet Implementation

4.1 GNUnet

GNUnet is a free software peer to peer framework with main focus on security. Figure 4.1 shows the GNUnet system architecture. It consist of layers of services. Each service provides a set of APIs. Other services can access its functionality through these set of APIs. GNUnet also contains user interfaces. User interface uses other services through their APIs but they don't have their own APIs. GNUnet provides a range of services and user interfaces, which are then combined into a layered GNUnet instance referred as GNUnet peer. [14]

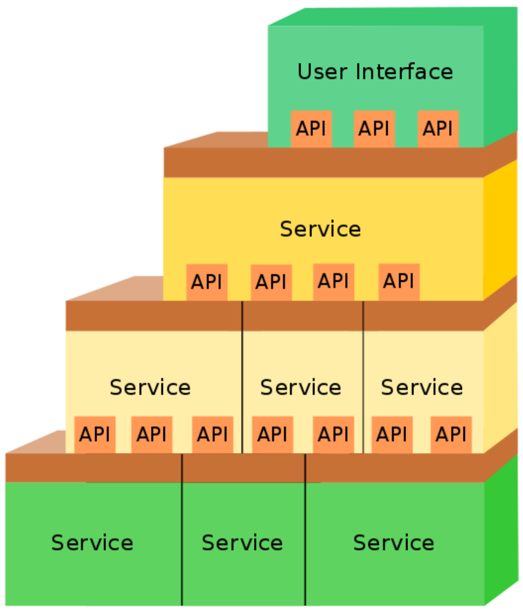


FIGURE 4.1: GNUnet Architecure [14]

4.2 GNUnet DHT System

Figure 4.2 shows GNUnet DHT system and its interaction with other relevant subsystems of GNUnet.

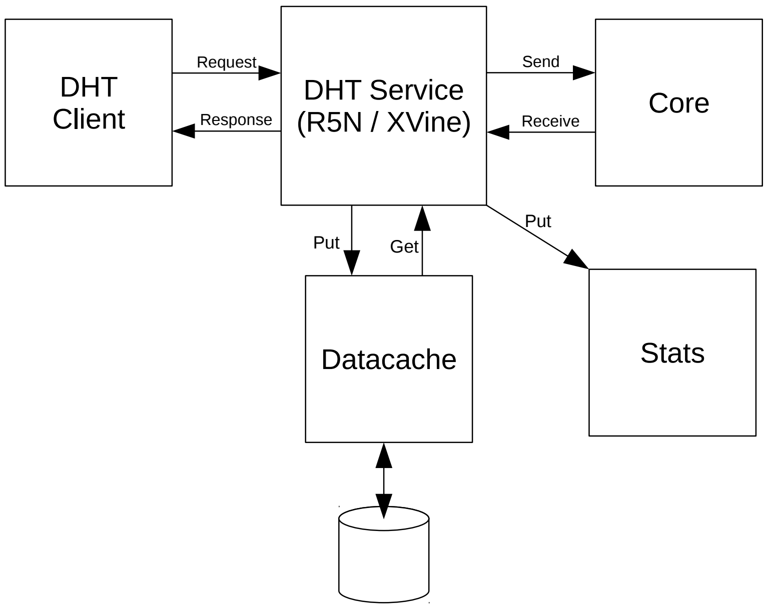


FIGURE 4.2: DHT service and its interaction with other relevant subsytem

1. DHT Client
   A DHT user accesses the DHT service through its APIs, PUT and GET. DHT client handles these requests and pass them on to the specific DHT service.
2. DHT Service
   DHT Service receives requests from the DHT Client and from other peers. It executes the lookup and routing protocol to appropriately serve the requests.
3. Datacache
   Datacache maintains an ephemeral storage for key/data pairs. DHT Service interacts with Datacache service to store and fetch the DHT Data.
4. Core
   The Core subsystem sets up secure communications between nodes in the GNUnet overlay network. It is build upon the Transport subsystem of GNUnet, that provides the actual link layer communication. DHT uses APIs provided by Core to send/receive messages from other peers in the overlay network.
5. Statistics
   DHT Service uses APIs provided by Statistics subsystem to record performance related statistics like traffic generated and packet dropped by DHT.

Due to the layered-architecture of GNUnet, it is possible to swap one implementation of a service with another, while keeping the APIs same. GNUnet currently uses R^5N DHT. X-Vine has been implemented as a separate DHT service in GNUnet, and it uses the same client side interface as R^5N.

4.3 Key and Peer Identifier Space

X-Vine is based on Chord. Given any network topology, X-Vine should connect its peers in a Chord ring overlay topology. We consider a 64-bit identifier space for our implementation.

In GNUnet, a peer identity is of type `struct GNUNET_PeerIdentity` which is a 256-bit value. The key used in GNUnet is of type `struct GNUNET_Hashcode` which is a 512-bit value. We extract the 64 most significant bits from the peer identity and key, and use it as a unique identifier.

For simplicity, the terms key and peer identity will be used to refer to the 64 bit identifier when discussing in context of the Chord overlay.

4.4 Global Data Structures

In X-Vine, every peer maintains the following data structures:

1. Friend Peermap
   The Friend Peermap keeps track of all friends connected to the peer. Peers connected through the core are added as friends of each other. This data structure is a hash map that uses peer identity as hash key. An individual entry of the Friend Peermap is represented by `struct FriendInfo`. Peers can send and receive messages only through their friends. Hence, a per-friend queue is maintained to keep a track of `Pending Messages` to be sent to the friend.

2. Finger Table
   It keeps track of all fingers of the peer. This data structure is implemented using an array. An individual entry of this array is represented by `struct FingerInfo`. It contains the unique peer identity of the finger, and a list of trails which can be used to reach the finger.

3. Routing Table
   A peer stores information about trails (of which it is a part), in its Routing Table. A trail is uniquely identified by its `trail_id`. The routing table is implemented as a hashmap with `trail_id` as the hash key. An individual entry is represented by `struct RoutingTrail`. It contains the peer identities of the next hop and the previous hop of the trail.

4.5 X-Vine Client-Service Protocol

This section describes the interaction between DHT client and DHT service.

4.5.1 PUTing data into DHT

The client sends a `struct GNUNET_DHT_ClientPutMessage` to the service to PUT (store) data into DHT. Along with the key-data pair, the message has the following fields.

1. 64-bit unique ID for PUT operation
   The service uses the same 64-bit ID to send back the PUT confirmation to the client.

2. Desired replication level
   R^5N performs PUT at the number of peers specified by this field. But in current implementation of X-Vine, this field is not being used. X-Vine stores the data at a single peer.

3. Expiration time
   Data in DHT is ephemeral. The data is deleted after expiration time as specified during PUT.

4. Block type
   Data is stored in blocks in Datacache. This field specifies the data format.

The service sends a PUT confirmation as soon as it has locally processed the request. But PUT request may still be propagating in the network toward its destination peer. A subsequent GET (retrieve) may or may not succeed, depending on the progress of PUT message.

4.5.2 GETing data from DHT

The client sends a `struct GNUNET_DHT_ClientGetMessage` to the service to GET (retrieve) the data from the DHT. The message contains a unique 64-bit ID for GET operation, key for the data to be fetched and block type of the desired data. The interpretation of these fields is the same as in the PUT message. Apart from these fields, the GET message also contains replication level field, that specifies the number of routes over which R^5N should try to retrieve the data. This field is not being used in X-Vine.

The service responds with `struct GNUNET_DHT_ClientResultMessages`. The response contains the block type, expiration, key, unique ID of the request and the value (block) corresponding to the key.

4.6 X-Vine Peer to Peer Protocol

X-Vine peers interact among themselves such that they can achieve the following goals:

1. Setup and Maintenance of Overlay Ring Topology.
   Peers exchange a set of messages among themselves to setup an overlay Chord ring topology. In case of churn, peers also need to ensure that the ring topology is maintained. Section 4.7 and 4.8 describe the algorithms.

2. Storage and lookup of key-data pair.
   To handle PUT and GET request from clients, peers exchange messages such that they can find the correct peer to store/retrieve the data. Section 4.9 explain the details of the algorithm used in this phase.

4.7 Overlay Ring Topology Setup

In this phase, every peer sets up its friend and finger table. A peer connected directly through Core is added as a friend. Section 4.7.2 describes the algorithm to search for fingers in the network.

4.7.1 Search space

In a 64 bit identifier space, a peer can have 64 possible fingers. Additionally, a peer keeps track of its immediate predecessor.

In X-Vine current implementation, the peer starts the search with its successor (finger index = 0), followed by its immediate predecessor. After finding these two fingers, it starts the search backward, iterating from finger index 63 to finger index 0.

4.7.2 Trail Setup Algorithm

Trail Setup Algorithm is used to establish trails for successor, predecessor and each finger of the peer. The peer periodically looks for its fingers, and stores multiple trails to reach the same finger. This improves resilience against Sybil attacks, as the peer can randomly choose the trail to reach a finger, thereby reducing the probability of choosing a trail with a malicious node.

The peer which is looking for its finger in the network is referred as source peer. It performs the following steps:

1. Using the Chord formula, source peer computes a 64-bit `finger_identity_value` from its own 64-bit peer identity (Section 2.1.2). The peer which is the closest successor of `finger_identity_value` will be the source peer's finger. Peer identity of closest successor is not known to source peer.
2. Source Peer constructs a message of type `struct PeerTrailSetupMessage`. The message contains,
    - `finger_identity_value`,
    - `trail_id`, a unique identifier for the trail to be constructed from the source peer to its finger,
    - `trail`, list of identities of peers which will be part of the trail, and
    - `intermediate_trail_id`, which will be explained later.
3. The source peer sends the message to a randomly selected friend to start the search for its finger.

Every peer that receives the *trail_setup* message performs the following steps.

1. Receiving peer searches its own friend and finger table for a peer, closer than itself to the `finger_identity_value`.

2. If the closest peer is a friend, the receiving peer adds its peer identity to the `trail` in the message, and forwards the *trail_setup* message to the friend.

3. If the closest peer is a finger, the *trail setup* message must be forwarded on the trail to this finger. The receiving peer updates the field, `intermediate_trail_id` with the trail id of the trail from receiving peer to the closest finger. It adds its own identity to the trail, and forwards the message to the first peer in the intermediate trail.

4. If the receiving peer is the closest successor of `finger_identity_value`, it must be the required finger. It constructs a `struct PeerTrailSetupResultMessage` and sends it back to the source peer, through the same trail. Every peer which is part of the trail adds an entry in its routing table.

Figure 4.3 illustrates the algorithm with an example. Let us consider a 6-bit identifier space, with peer identities ranging from 0 to 63. Peer 0 is looking for a finger, indexed 5. The `finger_identity_value` is calculated to be 32. Peer 0 forwards trail setup message to its friend, peer 5. Peer 5 looks for the closest peer, and forwards the message to its friend, peer 20. Peer 20 finds its finger, peer 28 to be closer and forwards the message along the intermediate trail. Peer 7 on the intermediate trail has a friend closer than the finger (peer 28) and forwards the request to peer 30. Peer 30 forwards the request to its friend, peer 34. Peer 34 is the closest successor of `finger_identity_value`. Peer 34 then sends the trail setup result message back, and every peer on the trail adds an entry in their routing table.

4.8 Overlay Ring Topology Maintenance

To handle churn, each X-Vine peer periodically verifies its successor and predecessor pointers. This ensures that even with nodes joining and leaving, eventually all the peers will be connected in overlay ring topology. Figure 4.4 shows the state diagram of the peer which follows the algorithm to keep its successor and predecessor pointers updated.

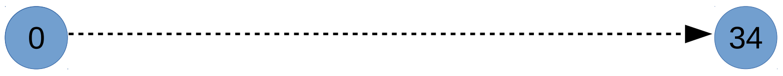
4.8.1 Looking for Successor

At the beginning, each peer tries to search for its successor using the Trail Setup Algorithm explained in Section 4.7.2. When a successor is found, the peer sends a message of type `struct PeerVerifySuccessorMessage` to the newly found successor. This message is sent again, if a reply is not received from the successor within a specific timeout.

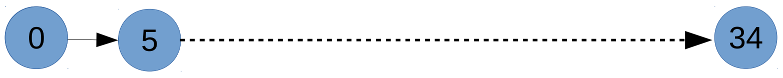
4.8.2 Verify Successor

The relationship between a successor and predecessor is mutual, i.e. each peer must be the predecessor of its successor. The peer which receives the *verify_successor_message* checks if this relation holds true.

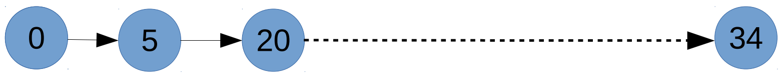
- If the receiving peer is a valid successor, it sends a message of type `struct PeerVerifySuccessorResultMessage` to the source peer to indicates its validity.



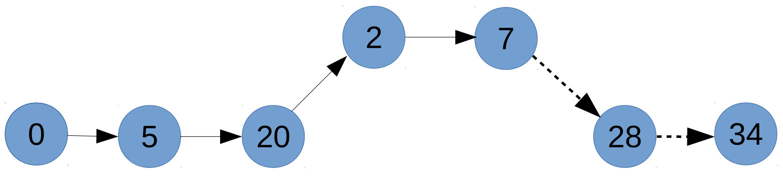
(A) Peer 0 initiates trail setup for `finger_identity_value = 32`.



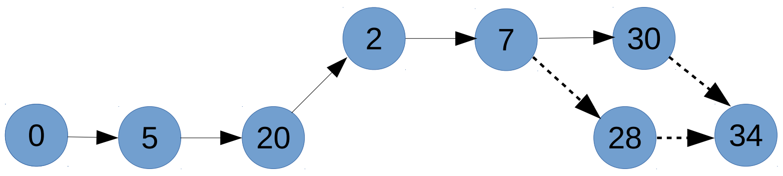
(B) Peer 0 forwards trail setup message to randomly selected friend, peer 5.



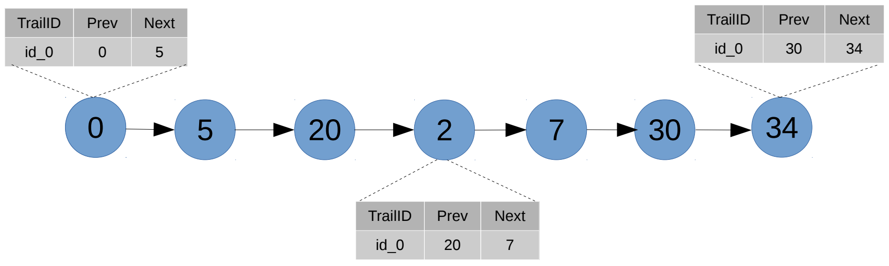
(C) Peer 5 forwards message to closest friend, peer 20.



(D) Peer 20 forwards message to closest finger peer 28.



(E) Peer 7 in intermediate trail, finds a friend closer than finger, and forwards to peer 30.



TrailID	Prev	Next
id_0	0	5

TrailID	Prev	Next
id_0	30	34

TrailID	Prev	Next
id_0	20	7

(F) Finger 30 forwards to closest friend peer 34, which is the finger. It sends result back, and all peers add entry in Routing Table.

FIGURE 4.3: Trail Setup example

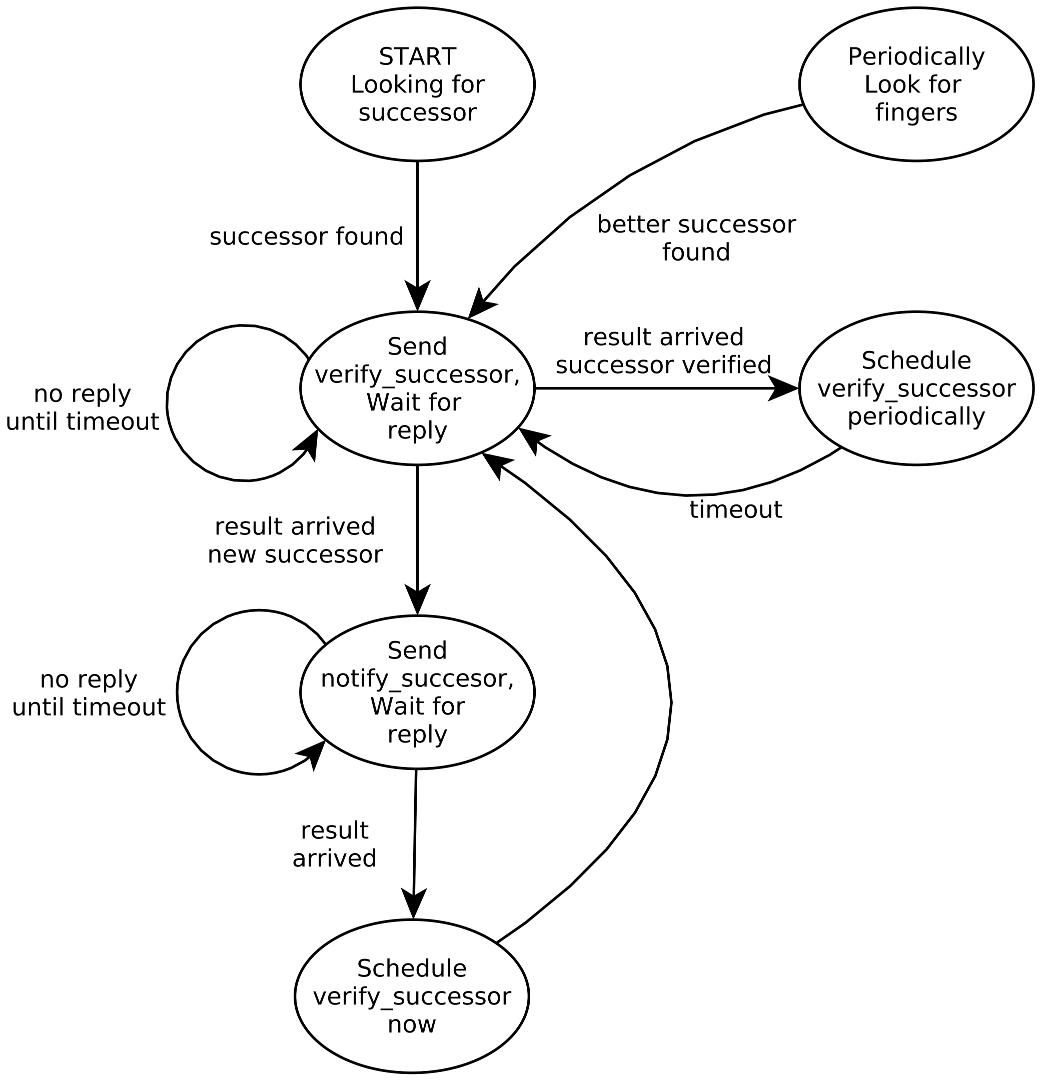


FIGURE 4.4: State Machine Diagram to illustrate Overlay Ring Topology Maintenance.

- However, if the condition is not true, there exists a peer P, which is the current predecessor of the receiving peer. Peer P could be the successor of the source peer. The receiving peer sends the `struct PeerVerifySuccessorResultMessage` to the source peer, indicating that it is not a valid successor. The message also contains the identity of peer P, and the trail to get to peer P from the source peer. This trail is a concatenation of the trails from source peer to receiving peer, and from the receiving peer to peer P.

4.8.3 Verify Successor Result

The initiating peer performs either of the two following tasks depending on the result of the verify successor message.

- If the result says, that the recipient was indeed a valid successor, the peer reschedules the verify successor task after a delay. This delay is exponentially incremented as the stability of the network grows.
- If the recipient was not a valid successor, the peer updates its current successor to the peer which is indicated as a probable successor in the result. The peer then sends a message of type `struct PeerNotifyNewSuccessorMessage` to the new successor. The aim of this message is to notify all peers in the trail to add the routing information in their routing tables. This message is sent again, if the acknowledgement is not received from the new successor.

4.8.4 Notify Successor

Each peer on the trail from initiating peer to the new successor, adds routing information to their routing tables, and forwards the message. On receiving the message, the new successor acknowledges it by sending a message of type `struct PeerNotifyConfirmationMessage` back to the initiating peer.

4.8.5 Notify Successor Confirmation

On receiving the acknowledgement from the new successor, the initiating peer again sends the verify successor message to the new successor.

This periodic process ensures that the overlay ring topology is maintained when churn occurs in the network.

4.9 Storage and lookup of key-data pair

4.9.1 Handling PUT request

The peer which receives the PUT Request from the DHT Client uses the look up protocol to reach the peer which is responsible for storing the key-data pair.

In lookup, the DHT Service identifies the closest successor of the key, using the similar algorithm as for trail setup. It forwards the PUT request to the successor. The receiving peer performs similar lookup and forwards the request.

The PUT request is subsequently delivered to the peer which is responsible for storing the key-data pair. The destination peer then stores the key-data pair.

4.9.2 Handling GET request

The peer which receives the GET Request from the DHT Client uses similar look up protocol to reach the peer where the key should have been stored.

The GET Result must also be sent through the same path used by the GET Request. Hence, each peer appends its own identity to the list of peers who have forwarded the GET Request Message.

The destination peer, looks for the requested key in its datacache. If the key is not found, a message of type `struct PeerGetResultMessage` is sent to the requesting peer, indicating that a GET failure has occured. If the key is found, the data associated with the key is sent to the requesting peer in the message of type `struct PeerGetResultMessage`.

4.10 Security Protocol

For secure lookup and routing algorithm, X-Vine implementation defines the following threshold.

1. Maximum Trails through a friend.
   A peer can locally decide a threshold on number of trails it wants to setup through any of its friends. It is defined by `MAXIMUM_TRAILS_PER_FINGER`. The attack edges, g are limited in a social network graph. Hence, the total number of attack edges that an adversary can launch is bounded by (g * `MAXIMUM_TRAILS_PER_FINGER`).

2. Bounded Routing State Size
   The threshold, `ROUTING_TABLE_THRESHOLD`, states the number of trails of which a peer wants to be part of. This limit ensures that no adversary can overburden a peer's resource.

   A peer that has reached this threshold, rejects all the trail setup request. It sends a `struct PeerTrailRejectionMessage` to the previous hop that forwarded it the *trail_setup* request. The message contains a time estimate, `CONGESTION_TIMEOUT`, for which the peer assumes it will be unavailable.

3. Redundant Routing
   Malicious peers can drop PUT/GET request. In order to make progress even with malicious peers, each peer stores multiple trails to reach to its finger. Maximum number of possible trails is defined by parameter, `MAXIMUM_TRAILS_PER_FINGER`. In case of PUT/GET, an honest node sends the request across all the trails to reach to destination finger. This increases the chance of taking a path with honest nodes.

All the thresholds described above are set statically in the current implementation.

Chapter 5

Experimental Setup

This chapter explains the environment used to compare R^5N and X-Vine. Section 5.1 lists down the parameters used to compare the two DHTs. Section 5.2 explains in detail the features of profiler which collects the relevant statistics from the DHT services.

5.1 Comparison Parameters

Given same topology and the network size, R^5N and X-Vine performance is compared for the following fields:

1. Successful GETs
   This field specifies the number of GETs that retrieved the correct data from DHT.
2. Average PUT and GET path length
   This field gives the number of hops taken to reach to the destination peer for PUT and GET request.
3. Control and Data Traffic
   This field gives the count of total number of messages exchanged among the peers in DHT service.

5.2 DHT Profiler

DHT Profiler collects the relevant statistics needed for comparison of the DHT services.

It uses the Testbed subsystem of GNUnet. Testbed subsystem starts a specified number of peers and connect them in the user desired topology. Once the peers are connected in a particular topology, the profiler starts the DHT service on all of the peers.

Profiler collects the following relevant information

1. Control and Data Traffic
   Every peer running the DHT service, stores the total number of messages it received or sent through the Core into the Statistics subsystem. Profiler collects it from all the peers, and returns the total traffic generated by the DHT service.

2. PUT/GET length
   PUT/GET requests provides an option, `GNUNET_DHT_RO_RECORD_ROUTE`. This option instructs the DHT to keep track of the PUT/GET path followed in the overlay topology. Profiler sets this option when doing PUT/GET.

3. PUTs/GETs
   Profiler specifies a time out for PUT and GET operation. The PUTs are always 100% as the DHT service sends the PUT confirmation as soon as it has locally processed the message. For GETs, the profiler checks if the data returned for a particular key is of same size as expected. GETs may fail if they are not completed within the specified timeout set by the profiler.

In case of X-Vine, the profiler waits for the network to stabilize before starting the PUTs and GETs. The network is considered to be stabilized when all the peers have found their correct successor, and have at least $O(log\ n)$ finger entries, where n is the total number of peers in the system.

Chapter 6

Results and Analysis

R^5N and X-Vine are compared using two different topologies, Scale Free and 2D Torus. The X-Vine has been shown to work well in Scale Free topology. R^5N has been proven to work well in 2D Tours. Hence, these two topologies were selected. For both the topologies, experiments were done with varying network sizes. Section 6.1 and Section 6.2 discusses the results for Scale Free and 2D Torus topologies respectively.

For all the experiments, replication level in PUT and GET operation for R^5N was set to 1. Timeout for PUT and GET operation was set to 30 seconds. The experiments were performed only for non-malicious peers.

6.1 Scale Free Topology

Scale free topology is a network where the number of outgoing links of a node follow a Power law distribution. This topology is constructed progressively by adding nodes to the network and introducing links to existing nodes, such that the probability of linking to a given node, i is proportional to the number of existing links, k, that node has [15].

TESTBED subsystem requires two parameters to be provided for construction of Scale free topology.

1. `SCALE_FREE_CAP`
   This specifies the maximum number of edges a peer is permitted to have while generating scale free topology. Its value is set to 10.
2. `SCALE_FREE_M`
   This specifies the number of edges to be established when adding a new node to the scale free network. Its value is set to 5.

6.1.1 Analysis

In this topology, the GETs were completed successfully for both the DHTs.

Table 6.1 shows the average PUT path and GET path length for X-Vine and R^5N.

For small networks, the average PUT and GET path length for both the DHTs are comparable. But as the network size increases, R^5N's average PUT/GET path length increases marginally but X-Vine shows a significant increase. The reason for such pattern is that in X-Vine, a peer can send and receive the messages only from its $friends$. If a peer's $finger$ is the closest successor to the key, the peer traverse the $trail$ to reach it. Unlike R^5N, X-Vine do not have any replication strategy, and the data is always stored at a single peer. Hence, as the network increases, the average PUT/GET path length increases.

Network Size	Avg. PUT Length		Avg. GET Length	
	X-Vine	R^5N	X-Vine	R^5N
100	4.84	5.08	6.52	5.65
500	11.88	6.81	12.26	6.64
1000	16.42	7.22	16.18	7.03

TABLE 6.1: Average PUT and GET path length in Scale Free for different network sizes.

Table 6.2 shows the control and data traffic caused by the two DHTs.

As expected, X-Vine generates significantly higher traffic. In X-Vine, peers exchange compartively higher number of messages to set up and maintain the overly ring topology [4.7][4.8]. In some of the messages, $trails$(a list of peers) are being passed. As seen in table 6.1, the average PUT/GET path length is higher for X-Vine. Hence, the total traffic generated is so high.

Network Size	Bandwidth	
	X-Vine	R^5N
100	7.39	1.49
500	374.54	23.12
1000	1516.53	41.15

TABLE 6.2: Control and Data traffic in Scale Free topology for different network size.

6.2 2D Torus

2D Torus is a network topology, where a peer has total number of outgoing edges 4. In a N-Torus network, there are N neighbours in horizontal dimension and N neighbours in vertical dimension.

6.2.1 Analysis

Table 6.3 shows the total number of PUTs made and corresponding number of successful GETs. Unlike Scale Free topology, R^5N does not have 100% successful GETs. For the experiments, only one replication was used. Therefore, not all the GETs were successful.

In case of X-Vine, if peers have setup pointers to its successors and fingers, GETs are always 100%, as seen in the results.

Network Size	X-Vine		R^5N	
	PUTs	GETs	PUTs	GETs
100	50	50	50	32
225	104	104	112	79
400	200	200	200	156

Table 6.3: Total number of PUTs and Successful GETs.

Network Size	Avg. PUT Length		Avg. GET Length	
	X-Vine	R^5N	X-Vine	R^5N
100	12.92	3.8	16.94	5.7
225	23.03	4.31	23.54	5.89
400	31.58	5.14	34.19	6.44

Table 6.4: Average PUT and GET path length in 2D Torus topology for different network sizes.

Table 6.4 shows the average PUT and GET path length. In comparison to R^5N, the average PUT/GET path length is longer by a large factor. The reasons presented corresponding to results in table 6.1 holds true for these sets of results also.

Network Size	Bandwidth	
	X-Vine	R^5N
100	49.16	1.6
225	307.43	4.86
400	1586.20	12.91

Table 6.5: Control and Data traffic (in MB) in 2D Torus topology for different network sizes.

Table 6.5 shows the total amount of control and data traffic generated by the DHT services. In comparison to Scale free, the traffic generated by X-Vine is higher in this case. With respect to R^5N, it generates more traffic for the same reason as stated for Scale free topology.

Chapter 7

Related Work

DHTs such as Chord[2] and Kademlia[10] are scalable and robust against churn. But they are vulnerable to security attacks.[3][13] There are a number of secure DHTs that uses different approaches to mitigate byzantine attacks. This chapter discusses two such secure DHTs, Whanau and Persea. Similar to X-Vine, these DHTs use social network as a part of their security protocol.

7.1 Whanau

Whanau is a single hop Sybil tolerant DHT [16]. Section 7.1.1 explains the basic design and defense strategy of Whanau. Section 7.1.2 discusses its performance.

7.1.1 Whanau Design

Key features of Whanau design are as follows.

1. Security assumption
   Similar to X-Vine, Whanau works on the assumption that there are limited number of attack edges [Section 2.2.2] that joins the honest and Sybil region of the social network. In such a network, a random walk will return a set that has larger fraction of random honest nodes and smaller fraction of Sybil nodes.

2. Routing table
   A Whanau node constructs its routing table by doing a random walk over the social network link, and collecting a set of node identifiers. The node add these IDs as its fingers in the routing table. For key lookup, the node forwards the request to its finger which is closest to the key value.

3. Defense against clustering attack
   Honest nodes pick their IDs uniformly over the set of keys. However, a Sybil nodes may select their IDs surrounding a particular key. This is referred as clustering attack. Whanau combines social network with *layered identifiers* for defense against clustering attacks. In *layered identifiers* every node has to choose its IDs over a set of layers.

Consider *layered identifiers* with two layers. For *layer_0* IDs, nodes pick up a random identifier over the key space. The Sybil nodes may pick up their random node IDs around a particular key. For *layer_1*, honest nodes randomly selects a *layer_0* ID from its own routing table and use it as its *layer_1* id. If the adversary nodes cluster in specific region for *layer_0* IDs, there is a high chance that the honest nodes also cluster in the same region for *layer_1* IDs. This results in a balance of honest and Sybil nodes and thus mitigates the clustering attack. A lookup routed over this area has a chance to succeed.

7.1.2 Performance

Whanau can tolerate upto $O(n/logn)$ attack edges in n-peer system. Lookups take constant time. However, as Whanau is a single hop DHT, it has high state and control overheads. Routing table of a peer in Whanau can contain upto O($\sqrt{n}logn$) entries. In case of high churn system, and for applications with higher demand of correctness and availability, high overhead can be a problem.

7.2 Persea

Persea is a Sybil resistant social network based DHT[17]. It is different from Whanau and X-Vine, as it does not assume fast mixing nature of social network. It derives Sybil resistance by assigning node IDs through a bootstrap graph. A new node can join the system only when it gets an invitation from an existing node. The parent node assigns node ID to the newly joined node. This hierarchal ordering of node joins and ID assignment is reflected in bootstrap graph.

7.2.1 Design

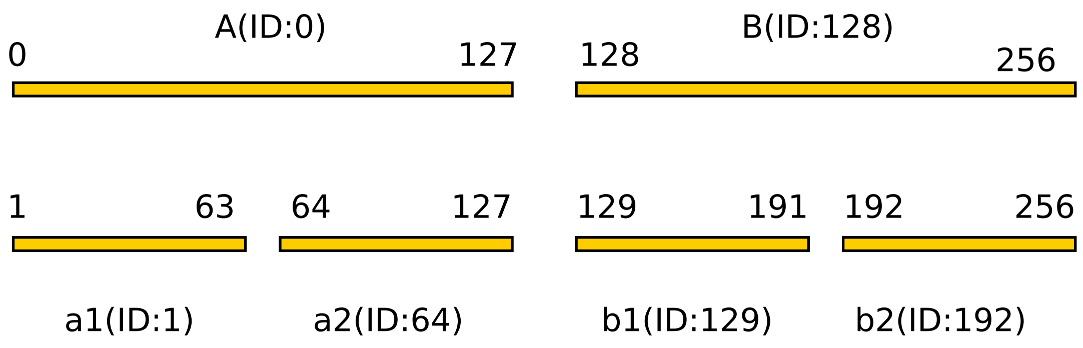


Figure 7.1: Bootstrap graph and ID distribution

Persea system start with initiator nodes called bootstrap nodes. These nodes have a direct link in social network. Consider Figure 7.1. It has two bootstrap nodes, A and B. A and B are responsible for a range of IDs, which are evenly spaced in circular ID space. New node gets its own ID and a chunk of IDs for which it is responsible. For example, node $a1$ is responsible for chunk $[2, 63]$. It can use this chunk to invite new peers and join the system.

In case a malicious peer joins the system, it can use its chunk to add other malicious peer. Due to limit on chunk size, a malicious peer can add only limited number of other such peers. Hence, the attackers will be limited in isolated regions of ID space.

Data stored in isolated region held by malicious peer may not be accessible. To handle such cases, Persea uses replication. Every data is replicated at peers whose ID ranges are evenly spaced out. This replication scheme guarantees that at least one of the replicas will be available even in a network with malicious nodes.

7.2.2 Performance

For same network size, and attack edges, Persea and Whanau performance are similar. But as Whanau is single hop DHT, it has higher maintenance overhead than Persea.

In comparison to X-Vine, Persea takes less number of hops for similar network size. For example, in a network with 100,000 nodes, Persea take on average 4 routing hops as compared to 10-15 hops in case of X-Vine. In comparison to X-Vine, the lookup rate is higher in a network with malicious peers.

Chapter 8

Conclusion and Future Work

X-Vine was successfully implemented over GNUnet. There is scope for improving performance of X-Vine. The experiments to compare R^5N and X-Vine gave the expected results. Comparison of performance with malicious peers could not be performed due to lack of time.

8.1 Future Work

Section 8.1.1 discusses further experiments that will be done to compare X-Vine and R^5N for larger and varied topologies. Section 8.1.2 discusses possible improvements and features for X-Vine.

8.1.1 Experiments

1. Testing with bigger network size and different topologies
   The current experiments were conducted only for two topologies, Scale free and 2D Torus. It will be interesting to evaluate X-Vine and R^5N performance for bigger networks and for other possible topologies.

2. Malicious Peer.
   Current experiments were performed only for the network with non-malicious participating peers. As R^5N and X-Vine are byzantine fault tolerant DHTs, it is important to evaluate and compare their performance in presence of malicious peers.

   The current implementation of DHT profiler provides an API, `act_malicious`. This API turns on the malicious behavior on a specified number of peers. A peer acting maliciously drops PUT/GET packets. The malicious peers do not disrupt the setup and maintenance messages.

   Using `act_malicious` API, X-Vine and R^5N performance in presence of different percentage of malicious peers can be evaluated.

8.1.2 Implementation

The experimental results shows that X-Vine uses significantly higher bandwidth as compared to R^5N. There are few optimizations that can be done to reduce the network traffic caused by X-Vine.

1. Replication Strategy
   The original X-Vine work do not mention about data replication strategy. Hence, in current implementation, the data is stored only at a single peer. We believe that using replication can significantly reduce the average PUT/GET path length and thus, can result in decrease in network load.

   Replication strategy similar to R^5N can be used for X-Vine. In R^5N, the key of the data is hashed to get the new key. PUT is done for original key and its hashed version.

2. Threshold value
   In current implementation, a peer imposes various threshold as described in section 4.10. These thresholds are set statically. In future, these values should be set based on some heuristics. For example, a threshold on maximum number of trails can be set depending on the behavior of adjacent social link over a period of time.

3. Fine tuning various delays
   As discussed in Chapter 4, X-Vine peer schedules a number of periodic tasks to setup and maintain the Chord ring topology. For example, a peer periodically looks for its fingers in the network, and verifies its successor.

   In current implementation, following optimization have been made to reduce network traffic caused by above mentioned periodic tasks.

   (a) The peer searches for its fingers by iterating through the finger table for each index. After each round, we increment exponentially the delay in scheduling the next round.
   (b) Once the network has stabilized, we reschedule verify successor task after exponentially increasing delay, upto a maximum of 15 minutes.

   By performing some analysis, these delays can be fine tuned to reduce network load, while taking in consideration the possible adverse effect on the time consumed by the network to stabilize.

Bibliography

[1] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 261–269, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4. URL http://dl.acm.org/citation.cfm?id=646334.687810.

[2] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[3] Anil Saroliya and Vishal Shrivastava. Security problems and their upshots in routing protocols of dht based overlay networks. *Journal of Theoretical and Applied Information Technology*, 2005.

[4] Prateek Mittal, Matthew Caesar, and Nikita Borisov. X-vine: Secure and pseudonymous routing using social networks. *arXiv preprint arXiv:1109.0971*, 2011.

[5] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. Dht routing using social links. In *Peer-to-Peer Systems III*, pages 100–111. Springer, 2005.

[6] George Danezis and Prateek Mittal. Sybilinfer: Detecting sybil nodes using social networks. In *NDSS*, 2009.

[7] Haifeng Yu, Phillip B Gibbons, Michael Kaminsky, and Feng Xiao. Sybillimit: A near-optimal social network defense against sybil attacks. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 3–17. IEEE, 2008.

[8] Chris Lesniewski-Laas. A sybil-proof one-hop dht. In *Proceedings of the 1st workshop on Social network systems*. ACM, 2008.

[9] Nathan S Evans and Christian Grothoff. R5n: Randomized recursive routing for restricted-route networks. In *NSS*, pages 316–321, 2011.

[10] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[11] Zhonghong Ou, Erkki Harjula, Otso Kassinen, and Mika Ylianttila. Performance evaluation of a kademlia-based communication-oriented p2p system under churn. *Comput. Netw.*, 54(5):689–705, April 2010. ISSN 1389-1286. doi: 10.1016/j.comnet.2009.09.022. URL http://dx.doi.org/10.1016/j.comnet.2009.09.022.

[12] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. A global view of kad. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 117–122, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-908-1. doi: 10.1145/1298306.1298323. URL `http://doi.acm.org/10.1145/1298306.1298323`.

[13] Moritz Steiner, Taoufik En-najjary, and Ernst W. Biersack. Exploiting kad: Possible uses and misuses. *ACM SIGCOMM CCR*, 37:2007.

[14] GNUnet website. `https://gnunet.org/system-architecture-with-legos`.

[15] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[16] Chris Lesniewski-Lass and M Frans Kaashoek. Whanau: A sybil-proof distributed hash table. NSDI, 2010.

[17] Mahdi N. Al-Ameen and Matthew Wright. Persea: A sybil-resistant social dht. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 169–172, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1890-7. doi: 10.1145/2435349.2435372. URL `http://doi.acm.org/10.1145/2435349.2435372`.