Wanhive

WANHIVE SYSTEMS PRIVATE LIMITED

Domain Controlled Access

Part 1: The Building Blocks

Amit Kumar

9/18/2014

Wanhive is an attempt at building a vastly scalable, secure and reliable "Decentralized Messaging Infrastructure" which can support multimedia information exchange in real time between a large numbers of users. Wanhive is an "Application Layer" Structured Overlay Network built for Throughput, Security, Stability and Mobility. At the same time, Wanhive has been designed to remain "Sufficiently Scalable". Wanhive has primarily been designed for application environment in which administrative control over the infrastructure is a desirable feature. Wanhive is suitable for "Mission Critical Applications" which require the underlying infrastructure to remain in highly predictable and consistent state. This document deals with the basic building blocks of the Wanhive Network in "abstract" fashion. This document discusses how "PARTITIONING OF THE KEY SPACE" modifies the topology and behavior of Structured Overlay Networks. Document introduces the idea of "Functional Partitioning of the Peer Groups" and shows how it creates clear hierarchies/levels inside a Structured Overlay Network and divides it into manageable chunks. The document also introduces simple mechanisms for Interaction between various Peer Groups and Node Authentication in such a network. Proposed improvements are in areas of "Network Topology", "Message Routing", "Churn Management", "Accessibility (Firewall Resistance)", "Authentication", "Node Clustering and Virtualization" and "Key Management".

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

Copyright Notice

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved*. All parts of this document are the original work of **Amit Kumar** at **Wanhive Systems Private Limited** (amit@wanhive.com). No part of this document may be multiplied, stored electronically or made public, in any form or by any means, neither electronically, mechanically, by photocopies, recordings, or in any other way, without prior written permission of **Wanhive Systems Private Limited** and/or **Amit Kumar**.

*All other trademarks, service marks, and trade names are owned by their respective companies.

Confidentiality Notice

N/A

Authors

1. *Amit Kumar is with Wanhive Systems Private Limited, Patna, Bihar 800024 (India). Email: amit@wanhive.com; amitkriit@gmail.com*

Document Properties

SN	Legend	Description
	Title	Domain Controlled Access
	Subject	Part 1: The Building Blocks
	Version	1.0
	Publish Date(M/D/Y)	9/18/2014

Revision History

Version	Change By	Comments	Date (Month/Date/Year)	Pages
1.0	Amit Kumar	Initial Draft	9/18/2014	23

Distribution History

SN	From	To	Date and Purpose	Copies

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

Keywords

SN	Keyword	Description

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

This page was intentionally left blank

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

1 Table of Contents

2 Key 8
2.1 Key Representation 8
2.1.1 Length of a Key 8
2.2 Value of a Key 8
3 Key Space or Key Domain 8
3.1 m-Key Space 9
3.2 (m, n)-Key Space 9
3.3 Partition of Key Space 9
4 Domain Controlled Function 9
4.1 Null Function 9
4.2 Key Hash Function 9
5 Key Management 10
6 Peer Group 10
7 Partitioning of Peer Group 10
7.1 Functional Partitioning of Peer Groups 10
7.1.1 Impact on Application 11
8 Universal Key Space (Ω) 11
9 Universal Functions 12
10 Key Store 12
10.1 Key Store Functions 13
11 Message 13
12 Partitioning of the Universal Key Space 14
12.1 Naming of the Key Spaces 14
12.2 Naming of the Peer Groups 14
13 Key Stores 14
14 Message Sets 15
15 Functions 15
15.1 Key Hash Function (H) 15
15.2 Universal Functions 15
15.3 Key Management Functions 16
15.4 Overlay Functions 16
15.5 Message Routing Functions 18
15.6 Control Functions 18

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

15.7 Edge Functions 19
16 Curious case of Forward 19
17 Deciphering the Key Stores 20
18 Controller Node 20
19 Managing Network Churn 20
20 Implementing Associations 21
21 Instance ID (I) 21
21.1 Association ID (A) 21
22 Visualizing the Message Paths and the Peer Groups 22
23 Conclusion 23
24 References 23

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

A. Introduction

In the virtual world keys are used for variety of application from cryptography (Cryptographic keys) to key-based routing (Network keys). Keys are nothing but overhyped Integers of the computer world. In this document we present few ideas on using these keys to some interesting effect.

B. Keys and Key Spaces

2 Key

Key (k): A key is an Integer.

2.1 Key Representation

A key **k** is represented as a binary string over the alphabet {0, 1}

$$\Sigma = \{0,1\}$$

$$k \in \Sigma^*$$

2.1.1 Length of a Key

Length **m** of a key **k** is the length of the string representing that key.

$m = |k|$ *when k is represented as a string.*

A single Key can be represented by various strings. e.g.

$k_1 = 10$ m=2

$k_2 = 0010$ m= 4

$k_3 = 000010$ m= 6

$k_1 = k_2 = k_3$

2.2 Value of a Key

Value of Key **k** calculated by **I(k)** is its Unsigned Integer value.

For example, **I(00110)** = 6

Convention

I(k) and **k** will be used interchangeably unless otherwise mentioned.

3 Key Space or Key Domain

A **Key Space** or **Key Domain** represented by the letter **K** is a **Finite Set** of keys.

K is a subset of the set of Natural Numbers (including zero).

$$K \subset N$$

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

3.1 m-Key Space

$K(m)= \{k \mid 0 <= k < 2^m\}$ where **$m \in N$**

A m Key space contains all Keys **k** which can be represented by binary strings of length <=m

3.2 (m, n)-Key Space

$K(m, n) = \{k \mid 2^m <= k < 2^n\}$ where **$m, n \in N$**

A (m, n) Key Space contains all Keys k which can be represented by binary strings of lengths in (m, n].

3.3 Partition of Key Space

A *non-empty set* of Key Spaces **$\{K_1, K_2, K_3 \ldots K_n\}$** is called a **partition** of the Key Space **K** if

1. **$K = K_1 \cup K_2 \cup K_3 \ldots \cup K_n$**
2. **$K_i \cap K_j = \emptyset$** for **$i, j = 1,2,3\ldots n$, $i \neq j$** and **$K_i \neq K_j$** (i.e. they are mutually disjoint)
3. **$K_i \neq \emptyset$ for $i = 1,2,3\ldots n$**

*Sets K_1, K_2 etc are called the **blocks** of the Partition.*

4 Domain Controlled Function

A Domain Controlled Function **F** *defined over a Key Space* **K** is a function of the form:

$F((x_1, x_2 \ldots x_n), \bullet)$ where **$(x_1, x_2 \ldots x_n) \in K^n$, $n \in Z^+$**

F uses an ordered list of keys from **K** as one of its parameters.

Note: F can have side effects.

4.1 Null Function

A Null Function is a Domain Controlled Function that does absolutely nothing. A Null Function doesn't have any side effects.

4.2 Key Hash Function

A Key Hash function **H** is an efficient non-invertible Domain Controlled Function:

$H: K_1 \rightarrow K_2$

Where **$|K_1| >> |K_2|$** and **H** is defined over **K_2**

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

C. Peer Groups

The central theme of discussion in this section is distribution and management of Keys in a multi-user computer application in which each and every user is assigned a unique key and the impact of key management/distribution over the performance and security of the application. The discussion is particularly valid for structured P2P networks.

Assumption: *The application uses one or more number of Key Spaces.*

5 Key Management

A user $\mathbf{u_i}$ is identified by a single key $\mathbf{k_i}$ assigned to it from a Key Space **K**. i.e. in our application there exist one or more key management functions:

KMF: U → K where **U** is the **set of users** and **K** is a **Key Space**

Convention

We will be using $\mathbf{u_i}$ and $\mathbf{k_i}$ interchangeably in our discussions. If the same key is assigned to many users, then those users cannot be uniquely identified.

Note: an example of the KMF is SHA-1.

6 Peer Group

A Peer group **P** is a set of users who are assigned keys from the same Key Space **K**. Typically applications use a single Key Space and hence under our terminology such an application will be having just one **Peer Group**.

Users are called **Peers** of one another if they belong to the *same* Peer Group. Typically applications use a single Key Space and hence under our terminology all users of such an application are peers.

Users $\mathbf{u_1, u_2, u_3 \ldots u_n}$ are **Peers** if $\mathbf{u_1, u_2, u_3 \ldots u_n \in P}$

7 Partitioning of Peer Group

Theorem 1: A Peer Group **P** can be partitioned by *partitioning the associated Key Space* **K**.

Proof: **KMF** is a function.

7.1 Functional Partitioning of Peer Groups

Logically Partitioned Peer Groups can be functionally partitioned through Domain Controlled Functions.

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

A Peer Group **P** associated with the Key Space **K** is granted access to only the set of Domain Controlled Function(s) $\mathbf{F_K}$ defined over **K**.

This is enforced in practice by implicitly passing the key of the user to **F**. e.g.
k is implicitly set into $\mathbf{x_1}$ of $\mathbf{F((x_1, x_2 ...x_n), \bullet)}$ **{{**a lot like implicit *this* reference passing in C++**}}**

The result is creation of mutually disjoint Peer Groups which have different functional capabilities.

Convention

Suppose **F** is a Domain Controlled Function defined over a Key Space **K**. Then following expressions are equivalent:

1. **F((n), •)** and **n.F(•)**
2. **F((n, n'), •)** and **n.F(n', •)**

Where **n, n' Є K**

7.1.1 Impact on Application

By partitioning the Key Space we can create Hierarchies and Boundaries inside the Application.

D. Distributed Hash Table (DHT)

We are going to apply whatever we have learned so far over Distributed Hash Tables (DHT), ***Chord in particular*** to see how we can make subtle improvements in the DHT to make it suitable for Real Time Multimedia Messaging Application. All the messages are routed **internally** through a network of managed Super Nodes in reliable fashion. This behavior is suitable for building mission critical managed applications inside a firewalled network in which direct P2P messaging over UDP might not be possible. Scalability and robustness have been traded with performance and consistency.

Convention

From now on if we say **DHT** it must be assumed that we are talking about the **Chord DHT** unless mentioned otherwise.

Computers in a DHT are known as **Nodes**. In our DHT implementation all the keys are associated with **nodes** because of the nature of the application (**Nodes are the only persistent objects**). **Hence the terms *User, Key and Node* will be used interchangeably**.

8 Universal Key Space (Ω)

In a typical DHT all objects (called "nodes" and "keys") are assigned keys from a single *m-Key Space* **K(m)**.

For the sake of simplicity let us assume that all nodes in our DHT have already been assigned a unique key from an **m-Key Space**. Let us call this key space the **Universal Key Space** represented by **Ω**.

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

Ω = K(m)

*Such DHT has just one Peer Group say **P**. A Peer Group Associated with the Universal Key Space **Ω** is called **Universal Peer Group**.*

9 Universal Functions

Following functions have been described for the **Chord DHT** on **WIKI [1]**. We are going to treat it as our **reference implementation**.

Table 1

SN	Name	Description
A	*BASIC FUNCTIONS*	
1	n.**find_successor**(n')	Ask node **n** to find the successor of **n'**
2	*n.**closest_preceding_node**(n')*	*Search the local table for the highest predecessor of **n'***
B	*STABILIZATION FUNCTIONS*	
3	n.**create**()	Create a new Chord ring containing a single node **n**
4	n.**join**(n')	Join a Chord ring containing node **n'**
5	n.**stabilize**()	Called periodically. **n** asks the successor about its predecessor, verifies if **n**'s immediate successor is consistent, and tells the successor about **n**
6	n.**notify**(n')	**n'** thinks it might be predecessor of **n**
7	n.**fix_fingers**()	Called periodically. Refreshes finger table entries of **n**.
8	n.**check_predecessor**()	Called periodically. Checks whether predecessor has failed.

All the functions listed in Table 1 can be interpreted as Domain Controlled Functions (refer to the convention in 7.1) over the Universal Key Space **Ω** of our DHT and hence these are called ***Universal Functions***. Their general forms are:

F((n, n')) and G((n))
*Where **n, n' $\in$ Ω** and **F, G** are **Universal Functions.***

10 Key Store

Suppose the Key Space **K** is a subset of the Universal Key Space **Ω**.

Then the Key Store **S** over the set **K** is defined as:

$S \subset K \times \Omega$

*A Key Store **S** defined over the Universal Key Space **Ω** is called **Universal Key Store**.*

An element of **S** is called an **Association**. If **(a, b)** $\in$ **S** then we say that **b** is associated with **a** in **S**.

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

10.1 Key Store Functions

For every Key Store **S** defined over Key Space **K** following Domain Controlled Functions are available for manipulating **S**.

Table 2

SN	Name	Description
1	k.**put**(x, S)	**S= S U {(k, x)}**
2	k.**remove**(x, S)	**S= S – {(k, x)}**
3	k.**check**(x, S)	Returns **true** if **(k, x) Є S**
4	k**.get**(S)	Returns **T** where **T= {y | x=n, (x, y) Є S}**
	k.**clear**(S)	**S= S – {k}** X **Ω** // all pairs having k as first entry are removed

All the functions listed above in the Table 2 *are the Domain Controlled Functions over* ***K****. Their general form is:*

F((k), x, S) *or* ***G((k), S)*** *where* ***k Є K and x Є Ω***

Convention

If a Key Space ***K*** *has just one Key Store* ***S*** *defined over it then the reference to* ***S*** *inside the functions listed in* Table 2 *can be omitted.*

11 Message

A set of Messages **M** over a Key Space **K** is:

M ⊂ **{(a, b, s, η) | a, b Є K** and **s Є {0,1}*** and **η** is **Cryptographic Nonce}**
η makes sure that each message is identified uniquely.

Individual *messages are represented by the symbol* ***μ****. A set of Messages defined over the Universal Key Space* ***Ω*** *is called a* ***Universal Message Set****.*

A set of Messages associated with a Key pair **(k_1, k_2)** is a subset of **M**:

$M(k_1, k_2) = \{\mu | \mu \in M, \mu.a=k_1, \mu.b=k_2\}$ where $k_1, k_2 \in K$

E. Building the Overlay

Let us summarize what have we learned so far about our DHT implementation:

1. A Universal Key Space **Ω** which is basically an **m-Key Space** and **(m >> 4)**.
2. A Universal Peer Group **P**, each peer has already been assigned a unique key from **Ω**.
3. A set of Universal Functions F_Ω defined over the Universal Key Space **Ω**.
4. An empty Key Store **S** defined over the Universal Key Space **Ω**.
5. A Universal Message Set **M**.

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

12 Partitioning of the Universal Key Space

There are several ways the Universal Set can be partitioned, but we have selected the one which is simple and computationally efficient.

Theorem 2: Suppose **Y** is a family of subsets of **K(m)** as defined below:

$Y = \{K_c, K_o, K_e\}$
$K_c = K(0)$
$K_o = K(n)$ where $0 < n <= m/4$
$K_e = K(n, m)$

*Then **Y** is a partition of **K(m)** for every **m>= 4***

Since Ω= K(m), Y is a partition of Ω. Also, observe that

$|K_c| < |K_o| << |K_e|$ if **m** is a moderately large number, **m >> 4**, say **m =64**

Proof: K_c, K_o and K_e comply with the rules given in 3.3.

12.1 Naming of the Key Spaces

Table 3

SN	Key Space	Name
1	K_c	Controller Key Space
2	K_o	Overlay Key Space
3	K_e	Edge Key Space

12.2 Naming of the Peer Groups

Partitioning of **Ω** partitions the Universal Peer Group **P** as described below:

Table 4

SN	Peer Group	Name	Key Space
1	P_c	Controller Group	K_c
2	P_o	Super Peer Group	K_o
3	P_e	Edge Peer Group	K_e

13 Key Stores

Following Key Stores are created at start:

Table 5

Sl	Key Store	Initial Value	Key Space
1	S_c	{(0,0)}	K_c
2	S_o	Ø	K_o
3	S_e	Ø	K_e

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

14 Message Sets

Every message generated by the DHT can be identified uniquely because of the presence of a nonce field in the message.

In the beginning ***M*** *contain all the messages the network is ever going to generate.*

15 Functions

All the Universal Functions listed in Table 1 *have been mapped to Null Function. Many of them will be re-implemented later.*

15.1 Key Hash Function (H)

The Key Hash Function **H** over $\mathbf{K_o}$ has been defined below

Listing 15.1-1

```
H: Ω → Ko
H((n), y)
  if y Є Kc
    return n;   // 0 → n
  else if y Є Ko
    return y;   // y → y
  else
    y →n' where n' Є Ko // map y to some element inside Ko
    return n';
```

15.2 Universal Functions

The following two Universal Functions have been defined over **Ω**.

Listing 15.2-1

```
// consume the message and produce side effects (contextual/polymorphic)
Consume_message((k), μ)
  if μ.b= k
    M= M - {μ}
```

Listing 15.2-2

```
// called periodically. Keeps the Key Stores updated
// S is chosen on the basis of context (see 10.1)
Check_association((k))
  y ←get()   //Randomly select
  if y has failed
    remove(y);
```

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

15.3 Key Management Functions

The following three **Key Management Functions** defined over $\mathbf{K_o}$ locate a key ∈ $\mathbf{K_o}$ in the DHT.

Listing 15.3-1

```
// ask node n to find the successor of id
Find_successor((n, id))
   //It is a half closed interval.
   if (id ∈ (n, successor] )
     return successor;
   else
     // forward the query around the circle
     n0 = Closest_preceding_node(id);
     return n0.Find_successor(id);
```

Listing 15.3-2

```
// search the local table for the highest predecessor of id
Closest_preceding_node((n, id))
   for i = m downto 1
     if (finger[i] ∈  (n,id))
       return finger[i];
   return n;
```

Listing 15.3-3

```
// returns true if the id lies in (predecessor, n].
Is_local((n, id))
   if id ∈ (predecessor, n]
     return true;
   else
     return false;
```

15.4 Overlay Functions

The following six **Overlay Functions** defined over $\mathbf{K_o}$ manage and stabilize the DHT:

Listing 15.4-1

```
// create a new Chord ring.
Create((n))
   if not O.exists(n)
     clear();       // remove all (n, •) pairs from So
     O.put(n);    // add (0, n) to Sc
     predecessor = nil;
     successor = n;
```

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

Listing 15.4-2

```
// join a Chord ring containing node n'.
Join((n, n'))
  if 0.exists(n) and 0.exists(n') and not exists(n)
    predecessor = nil;
    successor = n'.Find_successor(n);
    put(n);  // add (n, n) to So
```

Listing 15.4-3

```
// called periodically. n asks the successor about its predecessor, verifies if n's immediate
// successor is consistent, and tells the successor about n
Stabilize(( n))
  x = successor.predecessor;
  if (x Є (n, successor))
    successor = x;
    successor.Notify(n);
```

Listing 15.4-4

```
// n' thinks it might be our predecessor.
Notify((n, n'))
  if (predecessor is nil or n' Є (predecessor, n))
    predecessor = n';
    // all misplaced keys are removed,
    // please notice that the choice of H makes sure that (n, n) and (n, 0) pairs are never removed
    for all y ← get()
      if not Is_local(H(y));
        remove(y);
```

Listing 15.4-5

```
// called periodically. Refreshes finger table entries.
// next stores the index of the finger to fix
Fix_fingers((n))
  next = next + 1;
  if (next > m)
    next = 1;
  finger[next] = Find_successor(n+2^(next-1));
```

Listing 15.4-6

```
// called periodically. Checks whether predecessor has failed.
Check_predecessor((n))
  if (predecessor has failed
    predecessor = nil;
```

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

15.5 Message Routing Functions

The following two functions defined over K_o are used for message routing and are called **Message Routing Functions**:

Listing 15.5-1

```
// recursively forward a message around the circle to its destination
Forward((n), μ)
   k= μ.b;
   if Is_local(H(k))
     Deliver(μ);
     return n;
   else if (H(k) Є (n, successor] )
     n'= successor;
   else
     n'= Closest_preceding_node(H(k));
   return n'.Forward(μ);
```

Listing 15.5-2

```
// deliver the message for consumption
Deliver((n), μ)
   k= μ.b;
   if exists(k)    // check if (n, k) exists in So
     k.Consume_message(μ);
```

15.6 Control Functions

Execution of **Overlay Functions** (15.4) generally requires participation of several Super Peers. These functions can be implemented as **RPC**s. RPC requires passing of special messages between Super Peers. Let us call such messages the **Control Messages**.

The set of **Control Messages** M_c defined over $K_o \cup K_c$.

$$M_c \subset M(a, b) \text{ where } a, b \in (K_o \cup K_c)$$

These messages are delivered

1. Directly (if direct connection/association is available)
2. Through the Controller Node which acts as if it is a “secure” transparent proxy.

The following function has been defined over K_o

Listing 15.6-1

```
// Super Peer "n" creates a message for "n'" and sends it over to the Controller for delivery
Send_command((n, n'))
   μ ←M(n, n') // prepare a control message
   0.Forward_command(μ);    // forward through the Controller
```

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

The following function has been defined over K_c

Listing 15.6-2

```
// the Controller acts as if it is a Transparent Proxy.
Forward_command((n), μ)
   d= μ.b;
   if exists(d)    // check if (0, d) exists in Sc
     d.Consume_message (μ);
```

Benefit of this approach is that careful implementation might result in an extremely consistent and secure Super-Peer network.

15.7 Edge Functions

This is the business end of our application. **Edge Functions** have been defined over $\mathbf{K_e}$. These functions are used for sending message from one peer to another in the $\mathbf{P_e}$ Peer Group.

Listing 15.7-1

```
// register with at least one super-peer
Register((e))
   n ← Kc;                              // randomly select a Super Peer
   n' = n.Find_successor(n.H(e));    // find a good Super Peer for e to get associated with
   clear();
   n'.put(e);    // associate e with n'
   gateway = n';    //save the reference
```

Listing 15.7-2

```
// leave the network
Unregister((e))
   gateway.remove(e);
   gateway= nil;
```

Listing 15.7-3

```
// edge peer "e" prepares a message for "d" and sends it through the Overlay
Send_message((e, d))
   μ ← M(e, d)    // prepare a message
   n =gateway;
   if n.exists(e)
     n.Forward(μ);
```

16 Curious case of Forward

So far so good, but what if an Edge Peer manages to call **n.Forward(μ)** (Listing 15.5-1) directly? Even in absence of active attackers, there is no way for the end recipient to identify the source of the message by reading the (**μ.a**) field. In short, the source field becomes completely redundant.

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

Let us remember that Peer Groups are given access to Domain Controlled Functions defined for the associated Key Space “only” through **implicit key passing**. We called it the **Functional Partitioning** of the Peer Groups (7.1).

Suppose a node **u** whose key is **k** calls **n.Forward(μ)** or **Forward((n), μ)**, both calls are equivalent as per our convention. The system through **Functional Partitioning** guarantees that **k** is implicitly passed into the field **n**. But since **Forward** has been defined for the Key Space K_c, **n** must be in K_c.

This constraint makes sure that an Edge Peer cannot execute **n.Forward(μ)** directly because this function is undefined in K_e. **Forward** has been defined for K_c only with interesting consequences, for example a Super Peer can insert a message into the network pretending to be an Edge Node, but this is what “routers” do.

17 Deciphering the Key Stores

A **(x, y)** pair entry in a Key Store may be interpreted as:

1. Node **x** **knows** Node **y**
2. Node **y** **trusts** Node **x**

Also notice that:

1. There exists a **(0, n)** pair in S_c for every “running” Super Node **n**.
2. There exists a **(n, n)** pair in S_o for every “running” Super Node **n**.
3. There exists a **(n, e)** pair in S_o for every Edge Node **e** which is registered with the network (**n** is a Super Node).
4. Few more associations might exist implicitly, as described in following sections.

18 Controller Node

Controller Node may or may not be a real Node. Controller Node may just be a Virtual Node. Since all **Control Messages** *pass through* the **Controller**, it can be used for **Overlay Administration**. There is no explicit function for creation of a Controller Node because the Controller Node has been assumed to be “always available”. **A (0, n) association with the Controller Node may be emulated either through a connectionless, secure and reliable transport protocol (UDP over IPSEC) or through a TCP connection to a Secure and Highly Available Central Server**.

19 Managing Network Churn

The implementation must make sure that when the Controller goes down, rest of the network remains functional, which is actually close to reality because the Super Peers which depend on the Controller for DHT maintenance, have extremely low entry and churn rate. The implementation must build sufficient redundancy into Controller Node so that failures are rare and restoration is quick. No redundancy is needed for Super Nodes because the network naturally recovers from Super Node failures, while no assumption is to be made regarding the availability of the Edge Nodes (zero impact over the network). In our reference implementation the Controller has been virtualized, the Controller is just one or more “special” Super Nodes which temporarily take up the role.

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

F. Building Blocks of Trust

20 Implementing Associations

Associations for e.g. **(a, b)** can be implemented as reliable and authenticated TCP connections:

1. **b** creates a TCP/IP connection with **a**.
2. **a.put(b): a** authenticates **b** and subject to successful authentication **a** puts **(a, b)** into the Key Store (**a.put(b)** may be implemented as a cryptographic mutual authentication function).
3. **a.remove(b):** If the connection with **b** is broken or in case of an exception **a** removes the pair **(a, b)** from the Key Store.
4. For every finger **f** in the finger table and for every **successor** of a Super Node **n**, the associations **(f, n)** and **(successor, n)** may be **implicit**. When a finger or a successor changes the related associations must change implicitly (details omitted, but not very hard to implement).
5. Existence of an association **(a, b)** means that ***there exists a full-duplex authenticated and reliable channel of communication*** between **a** and **b**.

21 Instance ID (I)

Instance IDs are Cryptographic Nonce used to uniquely identify all the running instances (past and present) of all the Nodes in the DHT. Instance IDs are not related to the DHT Keys in any manner. Instance IDs are Private/Secret.

21.1 Association ID (A)

Association ID is calculated as described below:

A (a, b) = X((I(a), b, η))

- where **η** is a nonce known publicly
- Please note that **I(a)** is secret of **a** hence only **a** can generate **A(a, b)**
- **X** is a consistent Hash Function such as SHA2
- **(a, b)** is an association.

Few comments on Security and Authentication:

1. **Association ID** can be used for building routines which can be used to authenticate and secure our network.
2. **Key Stores** are the focal points around which such a secure infrastructure can be built.

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

G. Summary

22 Visualizing the Message Paths and the Peer Groups

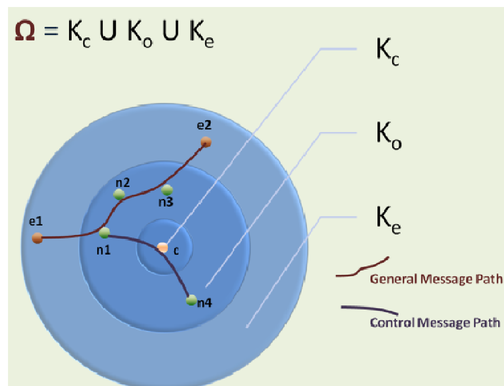

Figure 1

In the Figure 1 Messages between Edge Peers **u1** and **u2** get opaquely transported through the Super Peer network. Control Messages between two Super Peers **n1** and **n4** get transported through the Controller **c**.

TCP/IP is the default Transport Protocol which provides better message delivery guarantees and natural consistency than the UDP. TCP/IP is also suitable for application in a firewalled environment which is usually the case in corporate and government setups. Keys are distributed through a Registry Service (not based on IP address).

Few comments:

1. General and Control Messages flow through separate paths and hence separate security policies can be implemented for the two. For example, Control Messages can be selectively secured through **IPSEC** while the General Messages may be allowed to flow through insecure (but faster) path. The network must remain completely I.P. address agnostic.
2. There is a clear distinction and separation between the three kinds of Nodes; none can take up the role of another. There are strict rules of interaction between the Peer Groups. Hence errors in one group don't generally propagate beyond its boundary.
3. As we move outwards towards the edge of the network, constraints on robustness and availability of the nodes naturally get relaxed. This kind of setup is suitable for a managed network in which Super Peers and Controller are generally highly available, managed nodes while the Edge Peers enter and leave the network frequently.
4. The network is resistant to churn happening in the outer ring (Edge Peers) which also contains the largest number of unsupervised Nodes. The Network boots up fast and remains in a highly consistent state making ordered delivery of messages a possibility.
5. The Controller Node can be implemented as a virtual node so can the Super Peers; it might just be one of the Super Peers temporarily taking up the role of the Controller (that is the case in our base implementation) or the Controller might be virtualized as a secure and connectionless transport protocol (UDP). This allows us to adopt various clustering/grouping strategies for the Super Peers (for better management).
6. Mission critical applications require that behavior of the network must remain controllable and predictable. Critical Nodes (Super Peers) and Traffic (Control Messages) must be administered. The network must remain manageable. Network must have high Performance and Security but must remain **Sufficiently Scalable** at the same time.
7. Our reference implementation provides strong, robust and efficient Authentication Mechanism which is not usually possible for large structured networks (without compromising throughput).

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved

23 Conclusion

This document presents basic building blocks for implementation of a **Managed**, highly **Consistent**, **Structured**, **Efficient**, **Scalable** and **Secure** **Overlay Network** for Real Time Information exchange. This document also discusses major design goals and assumptions. We will be discussing the reference implementation, performance evaluation, constraints and few proposals for further improvement in sufficient detail in a separate document.

24 References

1. *Wikipedia: Chord (peer-to-peer) (http://en.wikipedia.org/wiki/Chord_%28peer-to-peer%29)*

Copyright ©2014 Wanhive Systems Private Limited, all rights reserved