TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur Professur für Rechnernetze

Diplomarbeit

Auswahl und Implementierung eines geeigneten Distributed Hash Table Verfahrens für die selbständig regionale Verteilungen berücksichtigende Organisation einer verteilten Prokura-Kommunikation in MapBiquitous

eingereicht von: Tan Dung Nguyen Matrikel: 3287713

Betreuer: Prof. Dr. rer. nat habil. Dr. h. c. Alexander Schill
Dr.-Ing. Thomas Springer
Dipl. -Inf. Tenshi Hara

ZIELSTELLUNG

MapBiquitous ist ein an der TU Dresden entwickeltes integriertes System für standortbezogene Dienste im Innen- und Außenraum. Mit Hilfe des Systems können Nutzer beispielsweise durch Gebäude navigieren. Da die Innenraumnavigation für die aktuelle Forschung von Interesse ist, stellt sich auch bei MapBiquitous die Frage, woher Kartendaten stammen, wie präzise sie sind, und wie bei Fehlern zu verfahren ist. Insbesondere die Wartung der Daten kann bei größeren Mengen für den Bereitstellenden eine Herausforderung bedeuten. Eine Möglichkeit, das Sammeln und Warten der Kartendaten zu dezentralisieren, und die Last auf die Nutzer des Systems aufzuteilen, ist das Crowdsourcing. Der Nutzer entscheidet sich, ob er beispielsweise WLAN-Fingerprint-Daten in dem Gebäude sammeln und zum zuständigen, die Gebäudedaten verantwortenden, Server schicken möchte. Die gesammelten Daten könnten dann analysiert werden, um die Karte des Gebäudes automatisch zu korrigieren und die Innenraumpositionierung zu verbessern.

Im Rahmen des MapBiquitous-Projektes wurde für die Organisation des Crowdsourcings und seiner Nutzer eine dezentrale, verteilte Crowdsourcing-Proxy-Struktur entwickelt und implementiert. Die Verteilung der Proxys wurde per Distributed Hash Table erweitert, um eine Schnittstelle sowohl zu einer IP-basierten Regionslokalisierung, als auch zum Domain Name System vorgeschlagen. Die so vorgestellte Struktur unterliegt jedoch einem Konzeptionsmanko: Der tatsächlichen physischen Lokation eines Nutzers wird bei der ersten Auswahl des zuständigen Proxys keine Rechnung getragen.

Der im Rahmen einer Diplomarbeit vorgeschlagene Lösungsweg, die Verteilung in mehrere regionale Hashtabellen aufzuteilen, erwies sich als zu komplex und langsam.

Die Synchronisierung aller Nutzer- und Crowdsourcing-Daten zwischen den verschiedenen regionalen Hash-Tabellen ist hier besonders hervorzuheben.

Ziel dieser Diplomarbeit ist, die Proxy in einer einzelnen Hashtabelle zu organisieren, wobei das zugrunde liegende Hashverfahren sowohl den üblichen Anforderungen an Skalierbarkeit und Leistung, als auch der regionalen Zugriffszeitoptimierung gerecht werden muss. So werden sowohl die Zuteilung der zuständigen Proxys, als auch das Suchen und Routen innerhalb der Verteilung übersichtlicher und besser verwaltet.

Zu untersuchen ist im Rahmen der so abgegrenzten Zielstellung, wie das regionale Wissen von Clients und Servern in das Hashing-Verfahren integriert werden kann. Prinzipiell gibt es drei Möglichkeiten das regionale Wissen zu berücksichtigen:

- in dem Routingalgorithmus,
- in der Hashing-Methode, oder
- in der Hash-ID integriert.

Für die erste Variante wird normalerweise eine globale Position oder heuristische Abstände zwischen Clients und Servern benötigt.

Die zweite Variante benötigt mindestens einen zusätzlichen Parameter im Algorithmus, nämlich eine Regionskennzeichnung. Dadurch wird die Hashtabelle segmentiert und in regionale Bereiche unterteilt.

Für die dritte Variante hat Shuheng Zhou einen Lösungsansatz beschrieben (Zhou, 2003).

Im Rahmen der Diplomarbeit sollen die drei beschriebenen Varianten diskutiert werden. Anschließend ist eine Überprüfung der Vermutung, dass Zhous Verfahren das geeignetste ist, durchzuführen. Bewahrheitet sich die Vermutung, soll eine Proof-of-

Concept-Implementierung auf Basis eines existierenden DHT-Verfahrens, beispielsweise dem von Petar Maymounkovs, und Zhous ID-Verfahren erfolgen, andernfalls eine geeignete Implementierung, um die Vermutungswiderlegung zu bestätigen. In beiden Fällen muss ein passendes Bootstraping-Konzept vorgestellt und implementiert werden.

SCHWERPUNKTE

- Recherche verwandter Arbeiten zu Distributed Hash Tables,
- Auswahl eines existierenden DHT-Verfahrens für MapBiquitous oder Konzeption eines geeigneten neuen DHT-Verfahrens,
- Prototypische Umsetzung des Konzepts,
- Erarbeitung einer Evaluationsmethodik, und
- Evaluation und Bewertung der Ergebnisse.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und ist auch noch nicht veröffentlicht worden.

Dresden, 16.06.2014

……..
Tan Dung Nguyen

Inhaltsverzeichnis

1 Motivation

Verteilte Hash-Tabelle (VHT) ist eine Datenstruktur, die eine Verteilung der Daten gleichmäßig über Speicherknoten zu ermöglichen. Wegen vielen Vorteilen wie Skalierbarkeit, Fehlertolerant und selbst Organisation spielen verteilte Hash-Tabellen (VHT) große Rollen in modernen Verteilten Systemen und werden in vielen verschiedenen IT-Bereichen wie Namedienste, FIlesharing Anwendungen oder Publish / Subscribe Systemen eingesetzt.

Der aktuelle Forschungsprojekt Mapbiquitous an der TU Dresden ist ein komplexe System für standortbezogene Dienste im Innen- und Außenraum. Mapbiquitous Servers sammeln Crowdsourcing Daten von Benutzern um eigenen Daten zu korrigieren, Benutzer des Systems könnten sich in viele verschiedene Länder befinden. Klienten interessieren sich normalerweise die Informationen in seiner regionalen Umgebung. Um Zeit für die Kommunikation mit den Klienten zu reduzieren, sollen Servers des System möglich in der Nähe an Klienten stehen. Eine VHT-Implementierung wurde auch für die Organisation des Servers des Systems eingesetzt. Leider hat die implementierte VHT physikalische Lokation der Klienten gar nicht in Betracht genommen. Benutzung solchen VHT könnte passieren, dass physischen Abstand zwischen einem Klient und seinem zuständigen Server sehr groß sein, so dass eine Kommunikation sehr lang dauern könnte. Ziel dieser Arbeit ist eine Lokation basierte VHT Konzept zu entwerfen um das Problem zu beheben.

2 Das MapBiquitous Projekt

Dieses Kapitel versucht, einen Überblick über relevanten Informationen von MapBiquitous System für die Entwicklung des standortbezogene verteilten Hashverfahren Konzept zu geben. Detaillierte Informationen über MapBiquitous Projekt könnten in Referenzen wie (Springer, 2012), (Hara, 2013), (Bombach, 2013), (Scholze, 2009) gefunden werden.

2.1 Crowdsourcing

Im Rahmen von Diplomarbeit und Großen Beleg hat Tenshi Hara zusammen mit Gerd Bombach ein Crowdsourcing Konzept vorgestellt. Somit könnten Nutzer mit Hilfe ihres Smartphones die Daten erfassen und diese Daten an den Crowdsourcing-Server schicken. Der Server verwendet die Daten um die Positionierung zu verbessern. So könnte z.B. ein Nutzer merken, dass das System seine aktuelle Position nicht richtig ermittelt. Er könnte seine wirkliche Position auf der Gebäudekarte korrigieren und diese Korrektur zum Server schicken. Der Server sammelt mehrere Korrekturen von verschiedenen Nutzern, analysiert die Daten und aktualisiert die Daten in der Datenbank. Ein anderer Anwendungsfall für das Crowdsourcing ist z.B: Es könnte sein dass zwischenzeitlich einer Wand in einem Gebäude eingebaut wird. Mit Hilfe des Crowdsourcing Moduls könnte das System erkennen, dass an der Position der Wand das System keine Informationen mehr bekommt. Damit könnte das System vermuten, dass sich da ein Hindernis befindet, daraufhin wird die Gebäudekarte korrigiert.

Das vorgestellte Crowdsourcing Konzept hat erwiesen dass ein Nutzer entscheiden könnte ob der an dem Crowdsourcing teilnimmt. So müssen er und seine Geräte sich zuerst registrieren. Danach könnte dann die Verbesserung eingereicht, oder die Ver-

besserung entfernt werden, wenn der Nutzer weiß, dass es eine schlechte Korrektur ist (Bombach, 2013).

2.2 Architektur

MapBiquitous ist ein dezentrales verteiltes System, besteht aus Klient und Server. Die gesamte Architektur wird in Abbildung 1 dargestellt.

2.2.1 Server-Seite:

Server-Seite besteht aus drei Teilen: der Gebäudeserver, der Verzeichnisdienst und der sogenannte „Indoor Navigation Server Access Network Entity“ (INSANE).

Gebäudeserver speichert Informationen über die Gebäude, sie besteht aus Grundrissen-Informationen von Etagen, Positionsinformationen von WLAN Routern und Semitische Informationen über Räume im Gebäude. Benutzung von solchen Informationen könnte Klient mit Hilfe eines Smartphone in den Gebäuden navigieren. Wegen Sicherheit Problem sollte für jedes Gebäude ein Gebäudeserver zuständig. Normalerweise wollte der Inhaber eines Gebäudes nicht alle Informationen über die Gebäudepläne für Öffentlichkeit freigeben. Durch die dezentrale Architektur könnte die Inhaber Kontrolle über die Daten behalten. So könnte zum Beispiel eine Sicherheitszone in die Gebäude definiert werden, die nur für Mitarbeiter, nicht aber für Besucher zugänglich ist. Ein anderer Vorteil ist die bessere Verfügbarkeit. Beim Ausfall eines Gebäudeservers wird nur ein Gebäude betroffen (Bombach, 2013).

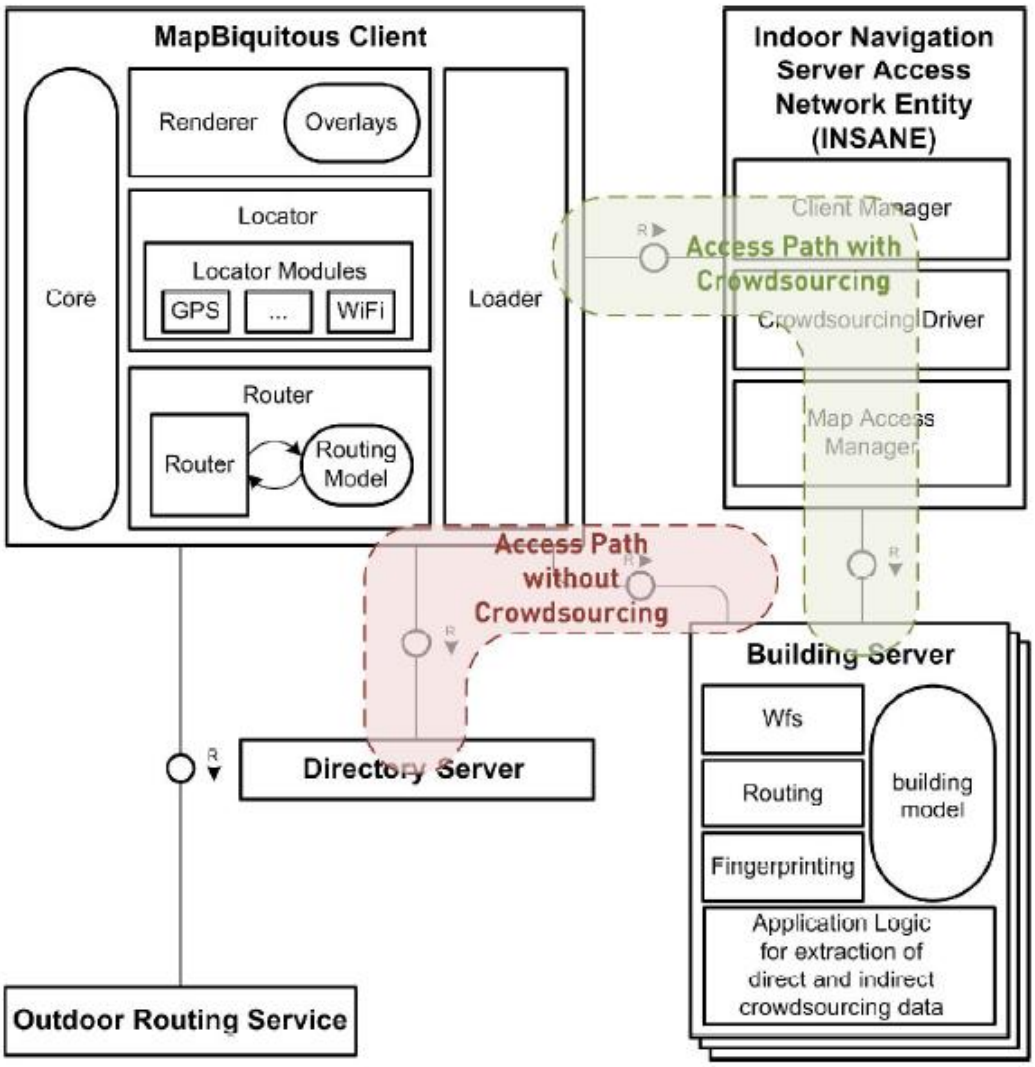

Abbildung 1: MapBiquitous Architektur (Hara, 2013)

Der Verzeichnisdienst enthält Informationen über Gebäudeserver. Damit einen Klienten ein Gebäudeserver ansprechen kann muss er zuerst eine Anfrage zu dem Verzeichnisdienst schicken. Der Verzeichnisdienst wird benutzt, um den zuständigen Gebäudeserver für ein Gebäude zu suchen. Der MapBiquitous Android Anwendung erfasst die minimale und maximale Longitude so wie minimale und maximale Latitude

des Gebäudes. Anfragen mit diesen Daten werden dann zum Verzeichnis Server gesendet. Als Antwort bekommt der Klient Name, den URL und die Koordinaten des passenden Servers, die für in der Nähe befindliche Gebäude sind (Hara, 2013).

Indoor Navigation Server Access Network Entity (INSANE) wurde von Hara in seiner Diplomarbeit eingeführt. Das INSANE verhält sich als ein Proxy für crowdsourcingbezogene Lesen Zugriff und Schreiben Zugriff zwischen Klient und Server. Alle Crowdsourcing-Anfragen sollten über INSANE laufen. Das INSANE enthält ein Nutzerverwaltungsmodul, da werden die Informationen eines Nutzers und seine Geräte gespeichert. Der INSANE könnte z.B. einen Nutzer sperren oder entsperren. Eine andere Aufgabe des INSANE ist die Verwaltung von durch Nutzer erfassten Crowdsourcing-Daten. Der INSANE empfängt die Crowdsourcing-Anfragen vom Klient, bearbeitet die Daten und leitet sie dann anonymisiert an die Gebäudeserver weiter. Eine vom Klient durchgeführte Korrektur wird nur in einem INSANE zwischengespeichert. Durch mehrere gleichbedeutende Korrekturen von verschiedenen Klienten könnte der Gebäude Server entscheiden, diese Korrektur auf die Datenbank zu übernehmen. Der Abbildung 2 erklärt eine mögliche Verarbeitung des Konzepts (Hara, 2013).

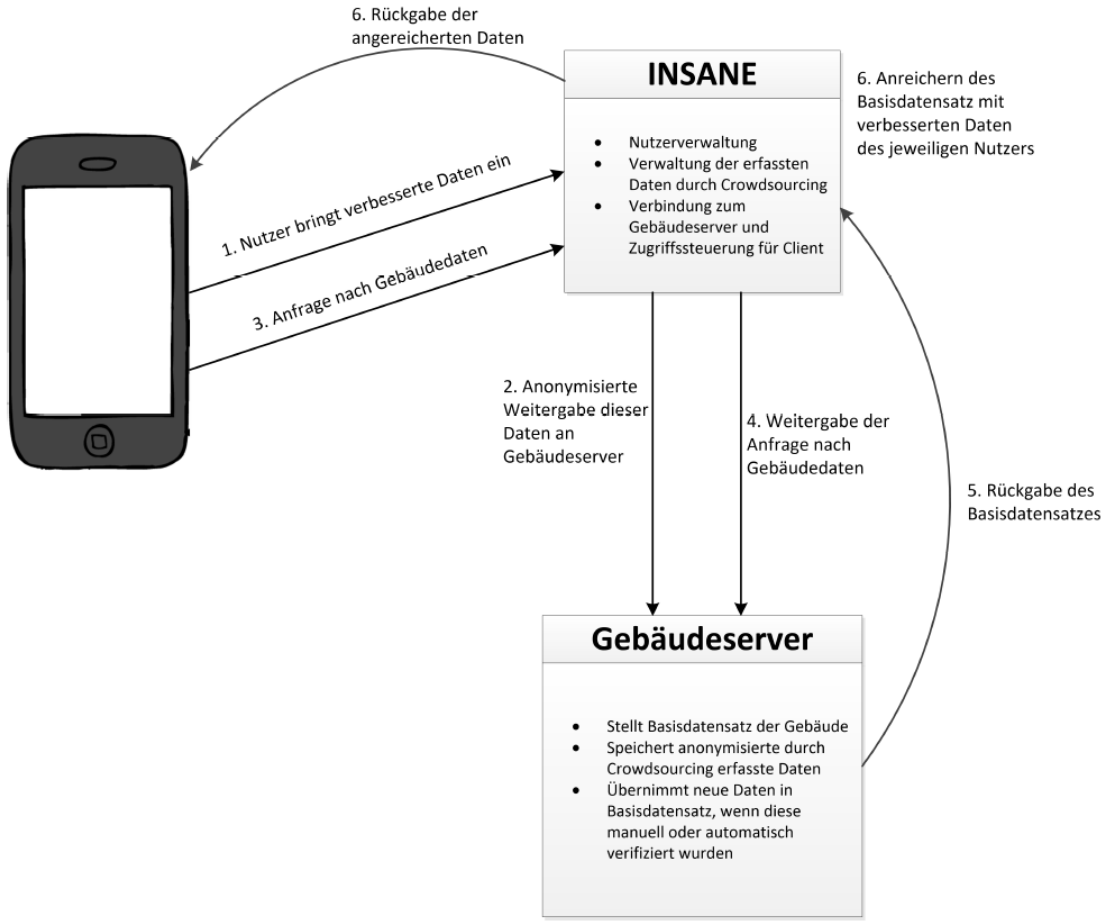

Abbildung 2: Crowdsourcing-Einreichung (Bombach, 2013)

Hara hat in seiner Diplomarbeit einen Konzeptentwurf mittels eines verteilten Hashtabelle Verfahrens (VHT) für die Organisation des INSANE vorgestellt. Im 3. Kapitel werden nähere theoretische Informationen über VHT betrachtet.

2.2.2 Klient-Seite

Klienten sind Applikationen auf einem Smartphone mit installierten Android Betriebssystem. Die Applikation besteht aus folgendes Modulen: Renderer, Loader, Locator, Core und Router (Bombach, 2013).

- Das Renderer Modul ist zuständig für das Erstellen und Anzeigen der Graphik Oberfläche der Applikation: Die benötigten Daten wie Z.B: aktuelle Position, aktuelles Gebäude, die Karte des Gebäudes usw. werden von anderen Modulen zum Renderer Modul geschickt. Diese Daten werden benutzt und mit Hilfe von Google Map API könnte ein Map Karte realisiert werden mit diversen Overlays Zusatzinformationen wie Gebäudegrundrisse, Bezeichnungen von Orten und die aktuelle Position des Nutzers (Bombach, 2013).
- Das Loader Modul beschäftigt sich mit der Kommunikation mit sowohl lokalem Server wie Verzeichnis Server, Gebäude Server als auch mit externen API-Schnittstellen wie der Google Location API und der Open Street Map API. Da eine Kommunikation sehr lang dauern kann und damit die Bedienung des Benutzers nicht verhindert wird, läuft das Loader Modul in einem separaten Prozess, dieser Prozess wird in Android System als ein Task genannt (Bombach, 2013).
- Das Locator Modul benutzt verschiedene Verfahren, um die Position eines Nutzers zu bestimmen. Zum einen wird die Standard Positionierungsmethode des Google Android API verwendet. Zum anderen wird das sogenannte WLAN-Fingerprinting als experimentelle Positionierungsmethode eingesetzt. Die Methode erfasst Signalstärken des WLAN Access Points für verschiedene Positionen in den Gebäuden und speichert diese Informationen in einer Location Datenbank. Die gemessenen Signalstärken während der Nutzungsphase werden mit den Daten in der Datenbank verglichen, dadurch wird die Position des Nutzers bestimmt (Bombach, 2013).

- Das Core Modul definiert Basis Klassen der Applikation, da werden Informationen des Gebäudeservern lokal zwischengespeichert. Die benötigten Daten werden durch Core-Modul zu anderen Modulen geliefert (Bombach, 2013).
- Das Router Modul verwaltet die Routeninformationen und erstellt sowohl die Navigationsroute als auch ein Overlay für diese Informationen auf der Karte. MapBiquitous benutzt Dijkstra Algorithmus um ein Route zu berechnen. Die Verarbeitung wird auf einen Routing Server entladen, um die Energie von Smartphone zu sparen (Bombach, 2013).

3 Verteilte Hashtabellen

Eine Implementierung der verteilten Hash-Tabelle wurde für die Organisation der INSANEs eingesetzt. Damit man weißt wie die INSANES mit einander kommunizieren, werden in diesem Kapitel Überblicke über VHT erklärt.

Wenn man schon mal mit Java als Programmierungssprache gearbeitet hat, kennt man sicherlich die Datenstruktur *Hashtable*. Mit Hilfe der *put(key, value) Methode* könnte *(Key, Value)* Paar in der *Hashtable* gespeichert werden. Danach könnte *Value* mit Benutzung der *get(key)* Methode zurückgeliefert werden (Oracle). Verteilte Hash-Tabelle ist eine Erweiterung der *Hashtable* für die Speicherung der *(Key, Value)* Paaren im verteilten System.

3.1 Spezifikation

Verteilte Hashtabelle (VHT) ist eine Datenstruktur in verteilten System, die *(Key, Value)* Paare gleichmäßig über Speicherknoten verteilt. In einer VHT werden Knoten eindeutige IDs zugeteilt. KnotenIDs und Schlüsseln werden durch Hashfunktion auf einen gemeinsamen Adressraum zugeordnet. Abhängig von Protokollen könnte dieses Adress-

raum entweder als Kreis (Chord), als binärer Baum (Kademlia) oder als eine quadratische Fläche (CAN) organisiert. Das gesamte Adressraum wird für einzelnen Knoten verteilt. Jeder Knoten ist zuständig für eine bestimmte Adressraum (Schill, 2011). Das Suchen für *Value* von einem *Key* funktioniert nicht mehr einfach wie beim normalen *Hashtable (*mit *get(key)* Methode*).* Dabei werden zuerst die Knoten, die das Key in seinem Adressraum enthält, gesucht. Das Suchalgorithmus ist der Hauptmerkmal einer verteilten Hash-Tabelle.

Der Hauptvorteil bei der Benutzung des VHT ist die **Dezentralisierung**. Ein VHT System soll keinen zentralen Server benötigen. Das gesamte Netz organisiert sich selbst automatisch (Martinovsky & Wagner).

Die anderen Eigenschaften von VHT sind:

- **Lastverteilung**: Da die Hash-Funktion gleich verteilt wird, sind die Speicherknoten im Idealfall der gleichen Anzahl von Werten zugeordnet.
- **Fehlertoleranz**: durch die Dezentralisierung wird das System zuverlässig, wenn ein Knoten ausfällt oder das System verlässt.
- **Skalierbarkeit**: Die Erweiterung auf große Anzahl der Knoten sollte möglich sein.

VHT wurde ursprünglich in der Forschung des P2P System wie Freenet, Gnutella, Bittorrent und Napster motiviert und in dem Bereich weiter verbreitet. Dabei gibt es mehrere Realisierungsmöglichkeiten der verteilten Hashtabellen wie Choor, Content-Addressable Network (CAN), Pastry und Kademlia. (Schill, 2011)

3.2 Kademlia

Kademlia ist eine sehr weit verbreitete Technik, die eine verteilte Hashtabelle implementiert und in vielen verschiedenen P2P Netzwerken wie Emule, BitTorrent, eDonkey, Vuze … eingesetzt wird.

3.2.1 Kademlia-Knoten

Kademlia besitzt einen 160 Bit breiten Adressraum und verteilt die Adressraum in einem binären Baum. Jeder Knoten und zu speichernde Daten werden mit eindeutigen 160 Bits Hash-Wert zugewiesen. Die Knoten werden auf Blätter des Baumes zugeordnet. Position der einzelnen Knoten wird durch sein kürzestes eindeutiges Präfix ihrer ID bestimmt. In Abbildung 3 sieht man ein Beispiel für Position des Knoten 0011 in dem binären Baum. Jeder Knoten im Kademlia System definiert eine eigene Struktur von Teilbäumen, die den Knoten nicht enthält. Der größte Teil-Baum besteht aus der Hälfte des binären Baumes, die den Knoten nicht enthält. Der weitere Teil-Baum besteht aus der Hälfte des verbleibenden Baumes usw. In dem Beispiel beschreiben die Kreise diese Teilbäume.

In einem Kademlia System weiß jeder Knoten mindesten einen Knoten in jedem seiner Teilbäume. Diese Spezifikation garantiert, dass jeder Knoten andere Knoten durch seine ID finden kann. Das Prinzip wird durch Petar Mamounkov und David Mazieres als K-Bucket in ihrem Paper beschrieben. Die folgenden Seiten werden genauer über K-Bucket informieren.

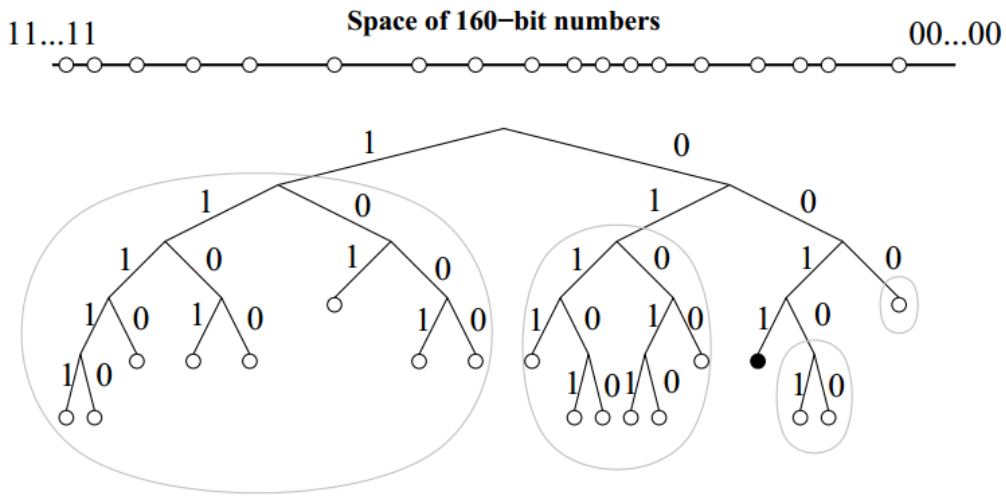

Abbildung 3: Kademlia binäre Baum (Petar Maymounkov, 2002)

3.2.2 XOR-Metrik

Kademlia benutzt XOR Operation, um den Abstand zwischen Knoten zu bestimmen. XOR berechnet die Anzahl der unterschiedlichen Bits von zwei Operanden: d(x, y) = x ⊕ y. Im Kademlia-Paper beschreiben die Autoren, dass XOR Operation die symmetrische Eigenschaft hat: ∀x, y: d(x, y) = d(y, x). Die symmetrische Eigenschaft sorgt dafür, dass der Abstand zwischen zwei beliebigen Knoten symmetrisch ist. Das heißt, dass der Abstand von Knoten A nach Knoten B so weit wie der Abstand von Knoten B nach Knoten A ist. Außerdem ist XOR eine unidirektionale Operation, die garantiert, dass alle Lookups der gleichen Schlüssel immer den gleichen Path haben. (Petar Maymounkov, 2002)

3.2.3 Routing-Tabellen

Jeder Knoten im Kademlia System speichert die Informationen von Kontakten in vielen Listen. Diese Informationen bestehen aus IP-Adresse, UDP-Port, Knoten-Id, um die

Kontakt-Knoten ansprechen zu können. Es gibt genau 160 Listen. Diese Listen werden als K-Bucket bezeichnet. Der i-ten ($0 \leq i \leq 160-1$) Liste eines Knotens enthält maximal k und minimal einen Knoten, die in einem Abstand zwischen 2^i und 2^{i+1} zu dem Knoten liegen. K wird beim Standard gleich 20 gewählt (Petar Maymounkov, 2002). Für das oben genannte Beispiel sieht eine Routing Tabelle wie in Abbildung 4 aus. Man sieht, dass jeder K-Bucket die Knoten mit einer Anzahl von gleichen Präfixen ihres IDs enthält. Dieses Präfix definiert die Position der Knoten in dem binären Baum.

NodeID			k-buckets		
0010	Abstand = 0	Abstand = 1	2-3	4-7	8-15
Kontakten/NodeIDs	0010	0011	0000	0101	1000 1010

Abbildung 4: Routing-Tabelle (Schill, 2011).

Die Kontaktknoten in einem K-Bucket werden mit der Last-See-Strategie organisiert. Der Knoten, mit dem am längsten nicht kontaktiert wurde, liegt ganz oben und umgekehrt. Sobald ein Knoten eine Nachricht empfängt, werden dann seine K-Buckets aktualisiert (Schill, 2011).

3.2.4 Kademlia-Protokoll

Das Kademlia Protokoll hat Vier Remote Procedure Call (RPCs) Befehlen: PING, STORE, FIND_NODE und FIND_VALUE (Petar Maymounkov, 2002).

- PING: Die Ping Methode wird gesendet, um zu ermitteln ob ein Knoten online ist. Dafür werden IP Adresse und Port des Knotens verwendet.
- STORE weist einen Knote an, ein Paar (Schlüssen, Wert) zu speichern.
- FIND_NODE: Diese Methode hat ein 160 Bits Knoten-ID als Parameter und gibt die Truppen (IP Adresse, Port, Knoten-ID) für drei Knoten zurück, die dem Ziel-

Knoten am nächsten liegen. Es können auch weniger als drei Truppen gefunden werden.

- FIND_VALUE: Die Methode bestimmt die Werte eines Schlüssels. Der Prozess ist ähnlich wie FIND_NODE. Wenn der angefragte Knoten den Schlüssel verwaltet, dann gibt er den Wert zurück. Sonst werden drei Knoten gesucht, die dem Schlüsseln am nächsten liegen, und die Daten erfragt.

3.3 Das MapBiquitous verteilte Hash-Tabellen Konzept

Grundlagen über verteilte Hash-Tabelle werden in Kapitel davor präsentiert. Wie erwähnt, eine verteilte Hash-Tabellen Konzept wurde für das MapBiquitous System eingesetzt. In diesem Kapitel wird die implementierte verteilte Hash-Tabelle in Detail betrachtet.

Die Idee des Konzepts ist in der Abbildung 5 erklärt, wobei jeder Nutzer, der auf Daten von INSANE zugreifen möchte, wird durch Nutzung einer Hash-Funktion eine Hexadezimal Nummer zugeordnet. Als Input für Hash-Funktion wird der Benutzername vorgeschlagen. Jedes INSANE ist zuständig für ein bestimmtes Hash-Bereich. Die Größe des Hash-Bereichs hängt von der Anzahl des INSANEs ab. Für die Sicherheit, wenn z.B. ein INSANE ausfällt, sollte jedes INSANE die Hash-Bereiche der linken/unteren und rechten/oberen Nachbarn verwalten. Die Abbildung 5 zeigt ein Beispiel mit vier INSANEs im System:

- Der INSANE 1 ist zuständig für Hash-Bereich [0-3], könnte jedoch die Hash-Bereich [d-7] bearbeitet.
- Der INSANE 2 ist zuständig für Hash-Bereich [4-7], könnte jedoch die Hash-Bereich [0-b] bearbeitet.

- Der INSANE 3 ist zuständig für Hash-Bereich [8-b], könnte jedoch die Hash-Bereich [4-f] bearbeitet.
- Der INSANE 4 ist zuständig für Hash-Bereich [c-f], könnte jedoch die Hash-Bereich [8-3] bearbeitet.

In dem Beispiel wird ein Nutzer mit dem Benutzername (*PererMustermann*) mit dem Hash-ID *bbe29c00e7661de0xxx* zugeordnet. Da der Hash-ID hat den Anfang Ziffer *b,* wird der Nutzer durch den primären INSANE 3 [8-b] verwaltet. Der INSANE 2 und INSANE 4 sind Nachbarn von INSANE 3, speichern auch die Daten des Nutzers redundant, und könnten alle Crowdsourcing Anfrage des Nutzers bearbeiten (Hara, 2013).

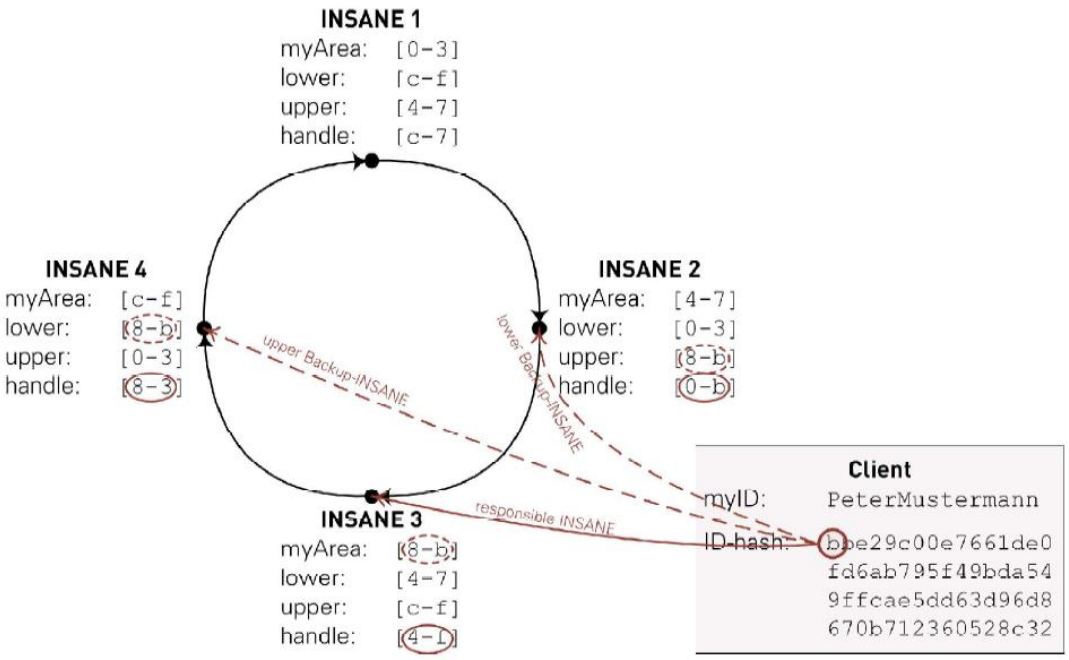

Abbildung 5: VHT für INSANE (Hara, 2013)

Weil das Konzept gar keine Berücksichtigung der geographischen Positionen des Klient und der INSANEs genommen hat, ist es möglich, dass die Daten eines Klienten in einem INSANE, der sich nicht in der regionalen Lokation des Klienten befindet, verwal-

tet werden. Zum Verdeutlichen dieses Problem hat Hara in seiner Diplomarbeit ein Beispiel gegeben: Wenn ein Nutzer in San Francisco (USA) an die INSANE zugreifen möchte um einen Korrektur für die Karte eines Gebäude, wo er sich momentan befindet, zu schicken. Seinen zuständigen INSANE liegt jedoch in Dresden. Weil der Gebäude in San Francisco liegt, befindet sich möglicherweise sein Gebäudeserver auch in San Francisco, oder zumindest in USA. Der Nutzer muss zuerst seine Anfrage an die INSANE in Dresden schicken. Der INSANE bearbeitet die Daten und leitet sie dann an den Gebäudeserver in San Francisco weiter. Diese hin und her Versendung zwischen zweit sehr weit entfernte Servers könnte ca. über 300 Millisekunden dauern (Hara, 2013). Eine so langsame Bearbeitung des Applikation könnte die Anzahl der Nutzer des Applikation verringern.

Um das Problem zu lösen wurde ein Lösungsansatz vorgeschlagen. Der Ansatz verteilt die INSANE in mehreren regionalen Hash-Bereichen. Inhalt der regionalen Verteilung sollte synchronisiert werden damit verschiedene INSANEs in verschiedene Regionen gleiche Hash-Bereiche verwalten könnten. Die Idee des Ansatzes ist in der Abbildung 6 dargestellt.

Abbildung 6 zeigt ein Beispiel wie das vorgeschlagene System das obengenannte Problem lösen kann.

1. Im Schritt 1 kontaktiert der Klient mit Hash-ID *bbe29c00e7661de0xxx* in San Francisco den INSANE X in Dresden.
2. Der INSANE X weißt IP Adresse des Klienten und fragt *Hostip.info* über Location des Klienten.
3. Im Schritt 3 bekommt der INSANE Information über Location des Klienten.

4. Der INSANE X ist zuständig für Hash-Bereich [a-b], jedoch weißt der durch Schritt 3 dass der Klient sich nicht in gleicher Region befindet. Der INSANE X sendet Anfrage an DNS um ein Kontakt-INSANE in USA zu finden.
5. Der INSANE X bekommt Kontakt Information eines INSANE A in USA im Schritt 5.
6. Der INSANE X beantwortet den Klient mit Kontakt Information des INSANE A.
7. Der Klient sendet die Original Anfrage an den INSANE A im Schritt 7.

8,9.Durch den VHT Routing-Algorithmus könnte der Klient den zuständigen INSANE in seinem Region finden und weitere Kommunikation durchführen.

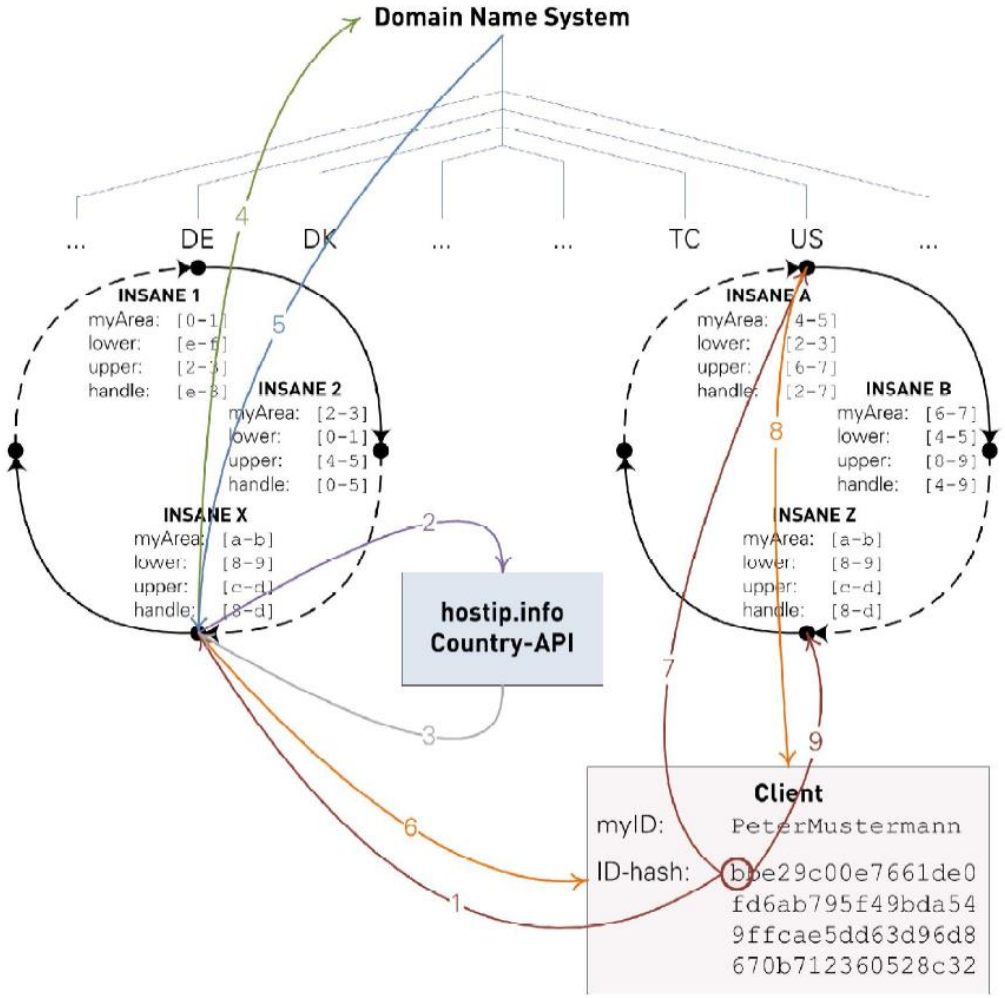

Abbildung 6: INSANE VHT mit DNS Schnittstelle (Hara, 2013)

Der Lösungsansatz betrachtet regionales Wissen des Klienten und INSANEs um die Performance des Systems zu versteigern. Auf dem ersten Blick hat der Ansatz das obengenannte Problem gelöst. Auf dem zweiten Blick hat die vorgeschlagene Organisation des INSANEs jedoch einige Probleme.

Dabei muss eine Synchronisation Verfahren zwischen Hash-Bereichen muss überlegt werden, damit verschiedene INSANEs in verschiedene Regionen gleiche Hash-Bereiche verwalten könnten. In dem Beispiel, eine entsprechende INSANE in Deutschland, der gleichen Hash-Bereiche wie INSANE X, muss gefunden werden damit die

Anfragen des Klient bearbeitet werden könnte. Die Implementierung eines solchen Verfahrens könnte sehr komplex sein.

Ein anderes Problem ist wenn eine Region gar keine INSANE hat. Was passiert dann wenn ein Nutzer von diesem Region Anfrage zu dem System schickt und der kontaktierte INSANE kann die Anfrage nicht handeln. Der DNS findet aber keine INSANE in seiner Region. Man muss auch eine Organisation für diesen Anwendungsfall geben. Hier könnte man ein INSANE in der Nachbar Region vorschlagen, jedoch müsste man hier die Geo-Location des Klienten und INSANEs entweder dezentral oder zentral verwalten. Die Verwaltung könnte das ganze System explodieren.

3.4 Zusammenfassung

In diesem Kapitel wurde zuerst theoretische Grundlagen über verteilte Hashtabelle präsentiert. Da wird das Grundkonzept des Kademlia Protokoll, eine sehr bekannte Technik für Peer-To-Peer Netze, in näher betrachtet. Danach wird das aktuelle verteilte Hashtabelle Konzeptentwurf des MapBiquitous vorgestellt. Das Konzept versucht regionales Wissen der Klienten und Knoten für die Organisation des Knoten und Routing Algorithmus einzusetzen. Jedoch gibt es einige Probleme die behoben werden müssen. Im folgenden Kapiteln werden ein paar Lösungsvorschläge um diese Probleme zu lösen.

4 Lokation basierte verteilte Hash-Tabelle Ansatz

Normale VHT erstellt eine logische Overlay-Netwerk, wobei die Knoten in Kreis (Chord), einem binären Baum (Kademlia) oder einer quadratische Fläche (CAN) organisiert werden. Das Routing in der logischen Overlay nimmt die Topologie des Netzwerks normalerweise nicht in Betracht und erstellt dadurch mehr Traffic Kosten für jede Suchen Operation. Eine Nachricht könnte z.B. von einem Region A zu anderer Region B und wieder zurück zu dem Region A gesendet werde. Um diese Problem zu lösen gibt es momentan mehrere Lokation basierte verteilte Hash-Tabelle Konzepte (LVHT).

Die meisten LVHT Konzepten benutzen Hash-Funktion um Key und NodeId in physikalische Position zu konvertieren. Wobei gleichen Keys werden zu gleichen Position konvertiert. Ein (Key,Value) wird in einer Knote, die sich in der Umgebung der konvertierten Position befindet, gespeichert.

Geografische Hash-Tabelle (GHT) (Ratnasamy, Karp, Yin, & Yu, 2002) ist eine von den solchen Konzepten. GHT benutzt zwei Konzepte: „home node“ und „home perimeter“. Der „home node“ eines Punkt ist der am nächsten Knoten in der Umgebung des Punkts. Der „home perimeter“ sind alle Knoten, die eine geschlossene Kante in der Umgebung des Punkts erstellen könnten. GHT biete zwei Routing Algorithmus heißt „greedy“ und „perimeter“ um zu garantieren, dass jeder Suchen Operation immer an der „home node“ der Ziel-Knote landet. Eine Weiterleitung der Nachrichten bei „greedy“ Routing nimmt immer die zum Ziel-Knoten am nächsten liegende Knoten. Für den Fall, dass „greedy“ Routing nicht mehr möglich ist, wird dann „perimeter“ Routing genommen. Aus Grund der Skalierbarkeit und Fehlertolerant werden (Key, Value) Paare nicht nur in der „home node“ sondern auch in der „home perimeter“ gespeichert. Weitere De-

tail zu GHT könnten in der Arbeit von Ratnasamy in (Ratnasamy, Karp, Yin, & Yu, 2002) gefunden werden.

Neben GHT gib es noch **GeoPeer**, eine weitere Lokation basierte verteilte Hash-Tabelle, der Key und NodeId auch in physikalische Positionen konvertiert. Für beiden Ansätze GHT und GeoPeer werden geografische Lokationen der Knoten benötigt, So-dass Einsatz bei manchem Systemen eine Schwierigkeit bekommen könnte. Außer-dem, für die Verarbeitung beim Suchen Operation werden die Positionen der Knoten sehr genau (GPS Position) betrachtet. So genau ist aber bei manchen Systemen nicht erforderlich, MapBiqutous System ist der Fall.

5 Lokation basierte Knoten-ID Ansatz

Shuheng Zhou hat im Jahr 2003 in seinem Bericht „*Location-based node IDs: enabling explicit locality in DHTs“* eine interessante Ansatz beschrieben, wobei die Informationen der Regionalen werden in ID der Knoten und Key der Daten eingebettet. Die Idee ist einfach: Regionen werden mit Präfixes zugeordnet. Diese Lokation basiertes Präfix wird mit dem normalen generierten VHT Knoten-ID verknüpft. Netzwerk Topologie des Systems könnte somit in IDs der Knoten eingebettet werden indem Ihre Präfixe in eine Hierarchie gestaltet werden. Die Hierarchie ist in der Abbildung 7 beschrieben. Dabei könnte Präfix aus mehre Teilen bestehen, die Verteilung zwischen Teilen hängt von Topologie, Geographik oder Geometrie des System ab. Ein Präfix könnte z.B. aus zwei Teilen: Land-Präfix und Stadt-Präfix bestehen. Anzahl der Ziffern des Präfixes hängt von Anzahl der Lände und Städter ab.

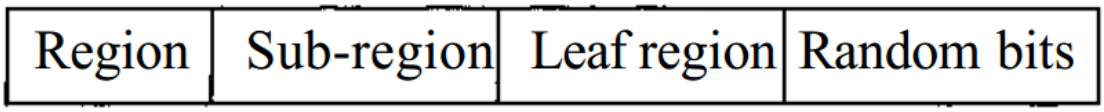

Abbildung 7: Struktur von Knoten-ID mit Lokation basierten Präfix

Vorteile:

Der erste Vorteil im Vergleich mit den Ansätzen in 4.Kapitel ist der Performance. Während GeoPeer und KHT die Positionen der Knoten sehr genau betrachten, betrachte der Ansatz von Shuheng Zhou die Positionen nur auf Regionen Ebene. Dadurch könnte das Kosten für die Verarbeitung beim Routing reduzieren, besonders wenn Knoten Density in System steigt.

Der GeoPeer und KHT betrachten eigentlich nur geografische Positionen, Dabei wird es angenommen, dass Positionen der Knoten Netzwerk-Topologie abdecken. Vielen

Systemen sind aber nicht immer so. Der Ansatz von Shuheng Zhou erlaubt man die Präfixe in Hierarchie, die ganz genau zu der Netzwerk Topologie entspricht, zu gestalten. Für MapBiquitous System in idealen Fall würde dann die Hierarchie eine Abstraktion ähnlich wie IPv4-Subnetze und Supernetting haben sollen.

Außerdem könnte man den Ansatz von Shuheng Zhou mit allen möglichen VHT Konzepten wie Chor, CAN oder Kademlia kombinieren. Der Routing Prozess wird in zweit Schritten geteilt. Zuerst ist der Präfixrouting, danach ist der Routing von entsprechenden VHT.

6 Eigenes Konzept

Nach dem die mögliche Ansätzen analysiert und verglichen werden, hat der Autor dieser Arbeit entschieden, die Idee des (Zhou, 2003) umzusetzen.

6.1 Kombination aus Lokation basierte Knoten-ID und Kademlia

Eine Lokation basierte Präfix könnte mit beliebigen VHT kombiniert werden. Kademlia Protokoll hat jedoch eine interesante Eigenschaft. Die Verwaltung der Knoten und das Routing Algorithmus von Kademlia basieren stark auf Präfixen der Knoten IDs, sodass wenn ein Lokation basierte Präfix vor der Knonten IDs angehängt wird, ändert sich das originale Routing Algorithmus nicht. Folgendes wird diese Eigenschaft erklärt.

Kademlia verwendet XOR Operation um Distanz zwischen zwei Knoten zu berechnen. Diese Distanzinformationen werden implizit in Routing Tablle gespeichert und werden benutzt um Nachrichten ziwschen Knoten weiterzuleiten. Das Beispiel in der Abbildung 8 erklärt wie die Routing Tabelle bei Verarbeitung der Finde_Node() Methode verwendet wird. In dem Beispiel, der Knote mit ID *0010* bearbeitet die Anfrage *FIND_NODE(1110)*.

Knoten-ID			k-buckets		
0010	Abstand: 0	Abstand: 1	Abstand: 2->3	Abstand: 4-> 7	Abstand: 8->15
Kontakt/ Knoten-Ids	0010	0011	0000	0101	1000 1010

Knoten-ID			k-buckets		
1010	Abstand: 0	Abstand: 1	Abstand: 2->3	Abstand: 4-> 7	Abstand: 8->15
Kontakt/ Knoten-Ids	1010		1000	1100 1101	0101

Knoten-ID			k-buckets		
1101	Abstand: 0	Abstand: 1	Abstand: 2->3	Abstand: 4-> 7	Abstand: 8->15
Kontakt/ Knoten-Ids	1101	1100	1110	1010	0000

Abbildung 8: Routing Tabellen von Kademlia' Knoten

1. Zuerst mit Hilfe der XOR Operation wird der Abstand zwischen beiden Knoten berechnet. 12 wird als das Ergebnis zurückgeliefert.

   Abs: 0010 Xor 1110 = 1100 (12)

2. Knote *0010* weiß, dass 12 zwischen 8 und 15 liegt. Kontakt Daten des zwei Knoten: *1000* und *1010* werden von der entsprechende KBucket geholt.

3. In einem rekusiven Schritt, der Initiator sendet Anfrage *FIND_NODE(1110)* zu den beiden gefundenen Knoten : *1000* und *1010.* Diese Knoten fangen wieder mit dem 1.Schritt an um die Anfrage zu bearbeiten.

Wenn das XOR Operation näher betrachtet wird, sieht man eine Beziehung zwischen XOR-Abstand zwischen zwei Knoten IDs und der Länge der größten gemeinsame Präfixes dieser IDs. Je kürzer diese gemeinsame Präfix ist, desto größer der Abstand zwischen den Knoten IDs ist. Wenn gleiche Präfix vor dem Knoten IDs angehängt wird bleibt der XOR-Abstand zwischen den neuen Knoten IDs (mit Präfix) unverändert. Es

bedeutet dass, wenn die Knoten in einem gleichen Region (gleiche Präfix) sich befinden, dann beim Anwendeung des Ansatz von Shuheng Zhou bleiben die Abstände zwischen Knoten unverändert. Somit könnte man originale Routing Algorithmus des Kademlia auch für erweitete Sysem verwenden um die Anfrage zu bearbeiten und Nachrichten weiterzuleiten.

Für den Fall, dass die Knoten in unterschiedlichen Regionen liegen und somit unterschiedliche Präfixe haben, betrachtet man die XOR-Abstände zwischen zwei Knoten vor und nach der Verküpfung mit Präfixes. Angenommen dass, die Präfixe n Bits haben. Laut (Petar Maymounkov, 2002), ein Kademlia Knote hat 160 Bits.

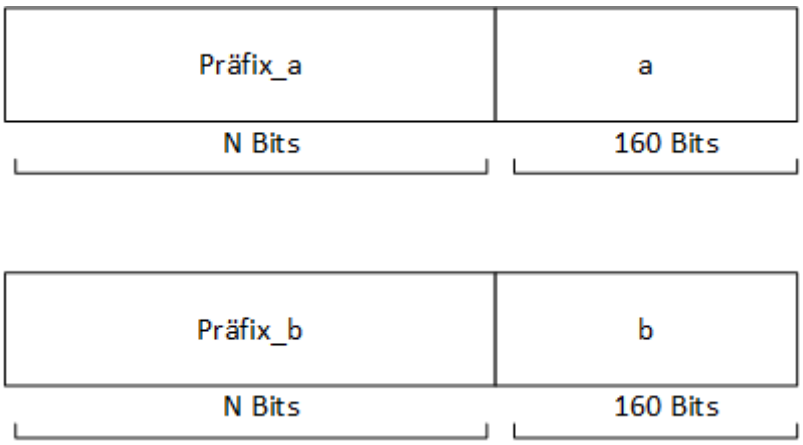

Abbildung 9: Knoten mit Präfix

Wenn

- *a xor b = s; a' = Präfix_a + a; b' = Präfix_b + b.*
- *Präfix_a xor Präfix_b = $p_{n-1}\ p_{n-2\ldots}\ p_1\ p_0$* = $p_{n-1}\ 2^{n-1}\ p_{n-2}\ 2^{n-2}\ldots\ p_1\ 2^1\ p_0\ 2^0$

Dann

- a' xor b' = $p_{n-1}\ 2^{(n-1)+160}\ p_{n-2}\ 2^{(n-2)+160}\ldots\ p_1\ 2^{1+160}\ p_0\ 2^{0+160}$ + s

  = (*Präfix_a xor Präfix_b)* 2^{160} + a xor b

Man könnte erkennen dass der XOR-Abstand des neuen Knoten a‘ und b‘ in relative Weise proportionale zu dem XOR-Abstand zwischen Präfixes ist. Das heißt, wenn es eine Konvertierung von Regionen zu Präfixen gibt so dass die XOR-Abstände zwischen Präfix Netzwerkabstände zwischen Knoten abdecken. Dann garantiert diese Konvertierung auch dass die XOR-Abstände zwischen erweitertem Knoten-IDs ihre realen Netzwerkabstände präsentieren.

Benutzung der obengenannten Behauptung könnten sich die Knoten in unterschiedlicher Regionen ohne große Erweiterungen der originalen Kademlia Routing Algorithmus finden. Ein konkretes Beispiel dazu wird in folgendes in Detail besprochen.

Angenommen, die Knoten liegen in zwei Länder, in Österreich und in Deutschland. Österreich hat den Präfix 1 und Deutschland 0. Wenn der Knote X mit dem Knoten-ID 0010 in Deutschland liegt und eine Routing Tabelle wie in der Abbildung 8 hat dann sieht der erweiterte Routing Tabelle wie in Abbildung 10 aus.

Knoten-ID			k-buckets			
00010	Abstand: 0	Abstand: 1	Abstand: 2->3	Abstand: 4-> 7	Abstand: 8->15	Abstand: 16->31
Kontakt/ Knoten-Ids	00010	00011	00000	01010	01000 01010	11000 10111

Abbildung 10: Routing Tabelle mit Präfix

Hinzufügen des Präfixes ändern sich die XOR-Abstände der Knoten in gleichen Region nicht sodass die Kontaktknoten des Knote X, die sich in Deutschland befinden, bleiben unverändert in der Routing Tabelle des Knote X. In dem Beispiel sind die Kontaktknoten mit einem Abstand von 16 bis 31 die Knoten, die außerhalb von Deutschland (in Österreich) liegen. Mit Hilfe dieses Kontaktknoten (11000 und 10111) könnte der Knote X beliebige Knoten in Österreich finden. Außerdem, Erweiterung des Präfixes beeinflusst die Zuordnung zwischen Key und seine zuständigen Knoten nicht. In Kademlia System, ob eine ein Knote ein Key verwaltet, hängt von Präfixen der Knoten ID und

Keys ab. der Knote X mit ID **001**0 könnte ein Data mit Hash-ID **001**1 verwaltet, weil der Hash-ID und Knoten-ID gleichen Präfix **001** haben. In der erweiterte System für den Fall, der Knote X mit ID **0001**0 verwaltet dann auch das Data mit Hash-ID **0001**1 da Sie haben gleiches Präfix **0001**.

Das gleiche Prinzip könnte man benutzen wenn die Knoten nicht nur in zwei Regionen: Österreich und Deutschland, sondern in mehrere Regionen liegen. Man muss dann eine passende Abbildung von Land zu Präfix finden so dass die XOR-Abstände zwischen Präfixen den Netzwerkabstand des Knoten präsentieren.

Zusammengefasst: Eine Erweiterung von Lokation basierte Präfix aus Kademlia hat Vorteilen im Vergleich mit anderen VHTs wie Tapestry, Chord oder CAN. Da Key-Verwaltung der Knoten eines Kademlia System basiert auf Präfixen des Key und Knoten IDs, könnte dann die Präfix-Routing in das originalen Kademlia Routing eingebettet werden. Das heißt, ein vorhandenes Kademlia System könnte einfach und schnell mit Shuheng Zhou Konzept kombiniert werden. Außerdem, Kademlia hat besserer Lastverteilung auf die Knoten im Vergleich mit anderen VHTs.

6.2 Präfixzuordnung

Regionen könnten in mehrere Stufen zerteilen wobei jeder Stufe hat eine feste Präfixlänge. Die Präfixlänge hängt von Anzahl der Stufenelemente ab. Ein Stufen-Hierarchie wäre z.B. Kontinenten, Nationen, Ländern und Städtern.

Es gibt insgesamt 5 Kontinenten. Da 5 ist kleiner als 2^3, könnte die Kontinente durch einen dreistelligen Binären dargestellt werden. Nach ISO-3166 (International Organization for Standardization), jedes Land ist zu einem Kombination von Drei Ziffern kodiert.

Die Kodierungsliste der ISO-3166 befindet sich im Tabelle 1 im Anhang. Eine Dreistellige Zahl könnte wiederum durch 10 Bits dargestellt werden. So könnte man ein Land durch 10 Bits dargestellt werden. Z.B. Deutschland ist nach ISO-3166 als 276 kodiert. Eine Binäre Kodierung von 276 wäre 0100010100. Dadurch könnte man Deutschland durch 0100010100 ansprechen. Ein Land könnte in mehreren Regionen zerteilen. Deutschland könnte z.B. in 16 Bundesländern zerteilen. Ob eine solche Zerteilung sinnvoll ist, hängt von der Anforderungen des Systems ab. Für MapBiquitous System, Optimal wäre eine Stufen-Hierarchie, die eine Abstraktion ähnlich wie IPv4-Subnetze und Supernetting abdeckt. Es gibt zurzeit insgesamt 241 Ländern. Aufgrund von Zeitmangel im Rahmen einer Diplomarbeit, könnten die Länder nicht in mehreren Regionen betrachtet werden. Die Hierarchie des MapBiquitous Systems hat somit zwei Stufen: Kontinenten und Länder.

Daraus lässt sich schlussfolgern, dass die Präfixen 13 Bits haben, Davon sind 3 Bits zur Darstellung der Kontinenten, 10 Bits zur Darstellung der Ländern.

6.3 Die Spezifikation des erweiterten Konzepts

Zur Vollständigkeit werden in diesem Abschnitt weitere Aspekten des vorgeschlagenen Konzepts vorgestellt.

6.3.1 Knoten-ID

Für Mapbiquitous System ist jedes INSANE als ein Knote bezeichnet. Jeder Knote hat eine eindeutige Knoten-ID der Länge von 173 Bits. 13 erste Bits bilden eine Lokation basierte Präfix, weitere 160 Bits werden durch eine Hashing Funktion berechnet, wobei Domänen der Knoten könnte als Parameter für Hash-Berechnung verwendet werden.

6.3.2 RPC-Methode

Wie bei originalen Kademlia Protokoll, es gibt insgesamt für erweiterte Protokoll vier „Remote Procedure Calls“ (RPC) Methoden: PING, FINDNODE, FINDVALUE, STORE.

6.3.3 Beitritt und Verlassen des Netzwerk

Um an das Netzwerk teilzunehmen, müssten IP Adresse und Port eines Knoten in dem System bekannt sein. Diese Knote wird in Kademlia Paper als Bootstrapping Knote genannt (Petar Maymounkov, 2002).

Es gibt mehreren Methoden damit ein Knote das Bootstrapping Knote identifizieren könnte. Eine Variante ist z.B: Der Bootstraping Knote hat ein bekannte statische IP Adresse und muss für immer in Netzwerk online bleiben damit die neuen Knoten ihn kontaktieren könnten.

Alternativ könnte Bootstrapping Knote mit Hilfe von Verzeichnis Dienst identifiziert werden. Wenn ein Knote das System beitreten möchte, müsste der Knote das DNS Server zuerst kontaktieren, um IP Adresse und Post des Bootstrapping Knote zu holen, So könnte der Bootstrapping Knote flexibel das Netzwerk verlassen. Der DNS Server findet immer einen zuständigen Bootstrapping Knote für kontaktierenden Knoten.

Für Mapbiquitous System wird die zweite Variante verwendet um Fehlertolerant zu ermöglichen. Das Bootstrapping Prozess könnte von dem Originalen Kademlia komplett übernommen werden. Entsprechende K-Bucket eines Knoten X wird nach „last seen“ Prinzip aktualisiert, jedes Mal wenn der Knoten X eine Nachricht von Knoten Y bekommt (Schill, 2011). Abbildung 11 verdeutlich diesen Vorgang:

- Wenn K-Bucket des Knoten X Kontaktinformation des Knoten Y enthält, wird dann der Eintrag des Knoten Y zum Ende des K-Buckets verschoben.

- Wenn der Knote Y für X unbekannt ist und der K-Bucket des Knoten X noch nicht voll ist, wird ein Eintrag für Knoten Y am Ende des K-Buckets hinzugefügt.
- Wenn der Knote Y für X unbekannt ist und der K-Bucket des Knoten X schon voll ist, wird ein PING Nachricht zu dem obersten Knoten gesendet. Wenn es keine Rückmeldung des Knoten gibt, wird der Knote aus dem K-Bucket gelöscht, der Knoten Y wird danach in K-Bucket hinzugefügt.

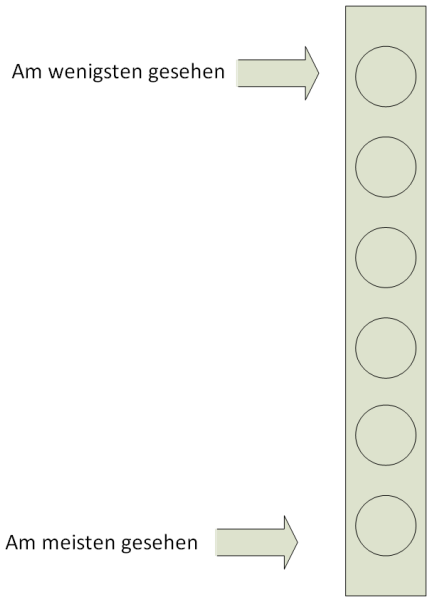

Abbildung 11: K-Bucket

Netzwerk beitreten: der Knote sendet eine Anfrage zuerst zu DNS Server um IP Adresse und Port des Bootstraping Knote zu erfahren. PING Methode soll zur Prüfung, dass Bootstraping Knote noch online ist, gesendet werden. Der Knote fügt zuerst Kontaktinformation des Bootstraping Knote in seiner Routing Tabelle ein.

Dank automatischer Aktualisierung von K-Bucket beim Nachrichtempfang könnte ein neuer Knote sehr einfach seine Routing Tabelle ausfüllen, indem er ein FIND_NODE Nachricht mit seinem ID als Parameter an dem Bootstraping Knoten sendet. Der

Bootstraping Knote bekommt die Anfrage und sucht dann eine Liste am nächsten Knoten, die wieder Anfrage des Knoten bekommen und ihre Routing Tabelle aktualisieren. Der Vorgang läuft weiter bis keine neuen Knoten gefunden werden. So wird die IP Adresse der neuen Knoten in Routing Tabelle der kontaktierenden Knoten gefüllt.

Netzwerk verlassen: Die Knote in einem K-Bucket wird immer wieder nach oben verschoben wenn er länger nicht kontaktiert wird. Wenn der Knote am Kopf des K-Buckets liegt und wird mit einem PING Nachricht versucht zu kontaktieren, der ist aber nicht erreichbar dann wird sein Eintrag in dem K-Bucket gelöscht. . So wird ein Knote das Netzwerk verlassen

6.4 Zusammenfassung

Diese Kapitel begründet zuerst warum die Kombination aus Shuheng Zhou Ansatz und Kademlia viele Vorteilen im Vergleich mit anderen VHT wie Chor, CAN oder Tapestry bringen könnte. Danach werden wichtige Aspekte des Konzepts für die Implementierung vorgestellt. Dabei sind die Informationen, wie Präfixen auf Regionen zugeordnet werden sollen, Aufbau eines Knoten IDs, wie ein Knote das Netzwerk beitreten und verlassen kann, präsentiert.

7 Implementierung

Nachdem in dem letzten Kapitel das erweiterte Kademlia Konzept vorgestellt wird, soll in diesem Kapitel die Umsetzung des Konzeptes in Detail betrachtet werden. Opensource für Kademlia wurde gesucht und ein paar gute Kademlia Opensource wie Openkad[1], geschrieben in Java, oder Maidsafe[2], geschrieben in C++, wurden gefunden. Weil INSANE Projekt in PHP Sprache geschrieben wurde, muss das erweiterte DHT Konzept leider auch in PHP umgesetzt werden. Es gibt jedoch keine gute PHP Opensource, die man verwenden kann. Das ganze Konzept muss komplett neu entwickelt werden.

Für das erweiterte Kademlia Konzept werden zwei neue Packages erstellt. Das Package „Kademlia“ enthält die Umsetzung von Vier Kademlia Methoden: *findNode*, *findValue*, *joinKademlia*, und *storeValue*. Die Ping Methode wurde durch Hara schon implementiert und kann wieder verwendet werden. Das Package „KademliaHelper“ enthält die anderen Methoden die per HTTP aufgerufen werden kann um die vier Kademlia Methoden zu realisieren.

Informationen über Key-Value Paaren und Kbuckets werden als Json Format in zwei flachen Dateien (kbuckets.json und keyvalue.json) gespeichert. Den Path zu diesen zwei Dateien könnte man in der Konfiguration Datei *config.inc.php* definieren.

7.1 FindNode ($nodeId, $exceptString, $serderAsJson) Method:

Die Methode hat drei Parameters: nodeId, exceptString und serderAsJson.

[1] https://code.google.com/p/openkad/
[2] https://code.google.com/p/maidsafe-dht/

- **NodeId** ist ein String, der eine Länge von 173 Chars hat. Ein Char ist entweder „0“ ider „1“.
- **ExceptString** ist eine Liste von KnotenIDs. Die *findNode* Methode wird von Knoten zu Knoten weitergeleitet. Durch diese Weiterleitung könnte eine Endlosschleife erstellt werden. Es passiert wenn z.B. Knote A die *findNode* Method zu Knote B sendet und Knote B sendet die *findNode* Methode wieder zurück zu Knote A. *ExceptString* wird benutzt um Endlosschleife zu vermeiden. Wenn *findNode* Methode zu einem Knoten kommt, der Knote fügt sein KnotenID zu dem *exceptString* hinzu und leitet den Request weiter zu anderen Knoten wenn es nötig ist.
- **SenderAsJson** speichert Triple (KnotenId, Host, Port) des Senders als Json Format. Die Information wird benutzt um Kbuckets Datei zu aktualisieren.

Folgendes Beispiel soll die Verarbeitung erklären.

```
//update kbuckets
      $senderNode = json_decode($senderAsJson, true);
      if(is_array($senderNode))
      {
            Kademlia::updateKBuckets($senderNode);
      }
…
while ( $responseSet != NULL && count ( $responseSet ) > 0 ) {

     // for each node in list
     foreach ( $responseSet as $key => $value ) {
     …
     // if node is in the exceptlist then doesn't send request
     to it and go to the next node
     $foundNodeIdInExceptList = strpos ( $exceptList, $value
[NODE_ID_FIELD] );
     if ($foundNodeIdInExceptList !== false) {
         unset ( $responseSet [$key] );
         continue;
     }

     // send Find_Node request to the node
      $host = $value [HOST_ID_FIELD];
      $post_data = array (
       'method' => 'findNode',
```

```
        'nodeid' => $goalNodeId,
        'exceptString' => $newExceptList,
        'senderAsJson' => json_encode(Kademlia::getOwnNode())
                  );

       $result = kad_post_request ( $host, $post_data );

       …

      } // end for


// get only 3 closest nodes from responseset for the next recur-
sive step
$responseSet = Kademlia::getKClosestNodesInList ( $responseSet,

$goalNodeId, Kademlia::NumberOfClosestNodes );} // end while
```

Abbildung 12: findNode

7.2StoreValue ($key, $data) Methode:

Die Methode wird durch Klient benutzt um Paare *($key, $data)* zu speichern. Um die *StoreValue* Methode zu implementieren werden zwei Hilf-Methoden vorbereitet: *StoreValueLocal* und *StoreValueJob.* Die beiden Methoden können durch HTTP Request aufgerufen werden. Nur Kademlia Knoten sollen Zugriff auf beide Methoden haben. Sie sollen für Klienten nicht sichtbar.

Ein Knote benutzt **StoreValueLocal** ($key, $data) Methode um Paar ($key, $data) in seiner Datei *keyvalue.json* zu speichern.

StoreValueJob ($key, $data) wird von einem Knote zu sich selbst gesendet um ein periodische Aufgabe zu starten. Beim Starten der Aufgabe werden mit Hilfe von *findNode* Methode die aktuelle drei zum Ziel am nächsten Knoten gesucht und die Methode *StoreValueLocal* zu diesen drei Knoten gesendet. Der Vorgang wiederholt sich jeweils nach eine Stunde und läuft nach 24 Stunden ab. Die Bearbeitung der Methode wird in der Abbildung 13 beschrieben.

```
$hour = 0;
while ($hour<24) {
    sleep (1*60*60);
    $closestNodes = findNode($key, "",null);
    // send store_value to 3 closest nodes.
    foreach ($closestNodes as $nodeId => $node) {
      $nodeHost = $node[HOST_ID_FIELD];
      $post_data = array (
            'method' => 'storeValueLocal',
            'key' => $key,
            'data' => $data,
            'senderAsJson' => json_encode(Kademlia::getOwnNode())
            );
      kad_post_request( $nodeHost, $post_data );
     }
     $hour += 1;
     print("sending store_value to closest node in ".$hour.". perio-
de".PHP_EOL);
     ob_flush();
     flush();
}
```

Abbildung 13: StoreValueJob

Mit Hilfe der Methode *stream_set_blocking()* können die beiden Hilf-Methode parallel gestartet werden ohne dass eine Methode die andere Methode blockiert. Abbildung 14 erklärt die Verarbeitung *der storeValue($key, $value)* Methode mit Hilfe der beiden Hilf-Methoden.

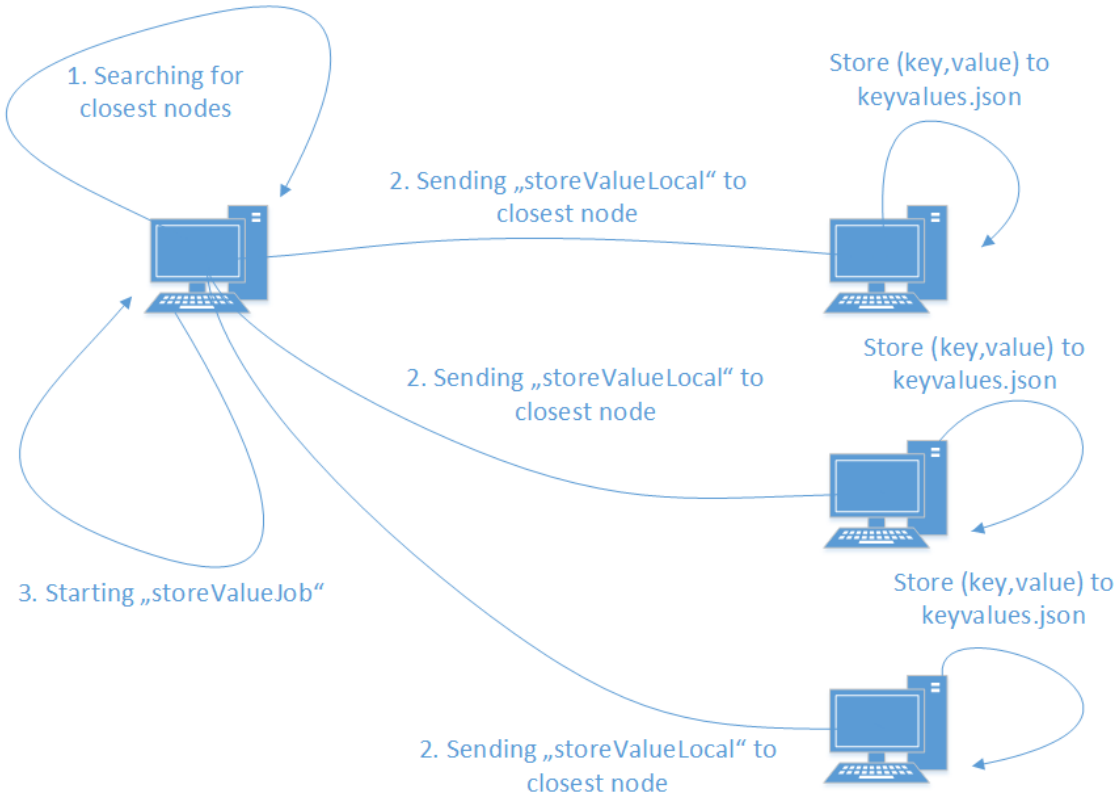

Abbildung 14: StoreValue

Zuerst werden „*closest nodes*“ gesucht, danach „*storeValueLoca*l“ werden zu den gefundenen Knoten gesendet um *(key,value)* Paar in ihrer *keyvalues.json* Datei zu speichern. Parallel dazu wird „*storeValueJob*“ gestartet um periodische Aufgabe auszuführen.

7.3 FindValue ($key) Methode:

Ein Klient kann die Methode verwenden um Daten von System zu holen. Zur Realisierung wird Methode *findValueLocal ($key, senderAsJson)* vorbereitet. Ähnlich wie beim *storeValueLocal*, beim Empfangt einer Anfrage *findValueLocal ($key, senderAsJson),* sucht der Knote den Key von seiner *keyvalue.json* Datei. Wenn Key gefunden wird, wird entsprechendes Datum zurückgeliefert. Wenn nicht, die am nächsten Knoten wer-

den gesucht und *findValueLocal* Methode wird zu den gefundenen Knoten gesendet. Die Implementierung ist in der Abbildung 15 zu sehen.

```
$key = $method_values["key"];

$keyvalues = Kademlia::getKeyValuePaars();
# IF this INSANE manages this Key, THEN answer with Key Value
if(array_key_exists($key, $keyvalues))
{
      header("HTTP/1.1 302 Found");
      $result = $keyvalues[$key];
      print(toJSON($result));
}
else
{
      # Else find closest nodes to the key and ask them for the data

$closestNodes = findNode($key, "",null);
      $found = false;
      foreach ($closestNodes as $nodeId => $node) {
            $nodeHost = $node[HOST_ID_FIELD];
            $post_data = array (
                        'method' => 'findValueLocal',
                        'key' => $key,
                        'senderAsJson' =>
json_encode(Kademlia::getOwnNode())
            );
            $result = kad_post_request ( $nodeHost, $post_data );
            if(($result["status"] === "ok") &&
($result["content"] !== "not found"))
            {
                  $found = true;
                  $print = $result["content"];
                  break;
            }
      } …


}
```

Abbildung 15: findValue

7.4 JoinKademlia ($bootstrapHost,$bootstrapContinent,$bootstrapCountry)

Die Methode ist realisiert dadurch, dass der neue Knote die *findNode*() Methode Bootstraping Knote sendet um sich selbst zu suchen. Das System speichert automa-

tisch Information der neuen Knoten. So wird der neue Knote an das System teilgenommen. Abbildung 16 ist die Umsetzung der Methode.

```
$ownNode = Kademlia::getOwnNode();
            $senderAsJson = json_encode($ownNode);
            // send find_node to bootstrap node
            $post_data = array (
                        'method' => 'findNode',
                        'nodeid' => Kademlia::getOwnNodeId(),
                        'exceptString' => '',
                        'senderAsJson' => $senderAsJson
            );

            $result = kad_post_request ( $bootstrapHost, $post_data );
```

Abbildung 16: joinKademlia

7.5 StoreUserData ($username, $data) und FindUserData ($username)

Zusätzlich zu den obengenannten Methoden werden eine *StoreValue ($user, $data)* und eine *FindValue($username)* Methode implementiert, um Daten, die zu einem bestimmten Benutzer gehören, zu speichern und zu suchen. Mit Hilfe des *hostip.info* werden zuerst *CountryCode* und *Continent* durch IP Adresse des Klienten gefragt. Dadurch könnte ein Key der Länge von 173 Bits berechnet werden. Code Beispiel in der Abbildung 17 erläutert die Umrechnung.

```
$hashedUsername = sha1($method_values["username"]);
$country = countryIP();
$countryCode = getCountryCode($country);
$continent = getContinent($country);
$key = continentToPrefix($continent)
.countryToPrefix($countryCode).hexId2bin($hashedUsername);
```

Abbildung 17: Key Berechnung

7.6 Zusammenfassung

Diese Kapitel beschreibt die Implementierung alle Methode, die zu dem erweiterten VHT Konzept gehören. Die Implementierung basiert stark auf großen Teilen das Kon-

zept. Umsetzung in PHP hat den Vorteil dass man keine große Schwierigkeit beim Installieren des Programms hat. Fast alle populäre Servers unterstütz PHP beim Standard. Man muss nichts extra installieren um das Programm laufen zu lassen. Es gibt jedoch ein paar Unangenehmigkeiten. Eine davon ist das "Shared Nothing-Architecture" [3] von PHP. PHP ist eine Script Sprache, für jede Anfrage wird Arbeitsspeicher initialisiert und Befehlen werden von Zeilen zu Zeilen interpretiert. Es kostet Zeit und macht PHP Applikation langsamer im Vergleich mit anderen Sprachen wie Java, C++ oder C#. Außerdem PHP hat gar keine integrierte Unterstützung für Multithreading. Für jede Anfrage wird ein neuer Thread erstellt. Alle Verarbeitungen der Anfrage laufen unter dem Thread. Das bringt bei Implementierung von komplexen Aufgaben großen Nachteil. Ein Beispiel dafür ist die Implementierung der *StoreValue*() Methode. Die Methode wurde in zwei Teilaufgaben geteilt: *StoreValueLocal* und *StoreValueJob.* Um die Bearbeitung der beiden Methoden parallel laufen zu lassen ohne dass eine Methode die andere blockiert, muss der Knote ein HTTP Request zu sich selbst senden um die Bearbeitung der Methode *StoreValueJob*() in einem neuen Thread zu starten. Erstellung neuer HTTP Anfrage kostet natürlich Zeit und Resources.

[3] https://segment.io/blog/how-to-make-async-requests-in-php/

8 Evaluation

Ziel dieses Kapitel ist zu überprüfen ob die Probleme des vorherigen Mapbiquitous System erfolgreich behoben werden.

- Problem: Keine Betrachtung der tatsächlichen physischen Lokation des Nutzers.

  Status: Behoben.

  Hauptziel dieser Diplomarbeit ist das obengenannte Problem zu lösen. Mit Hilfe der *FindNode*() Methode könnte jetzt ein Klient drei INSANEs in seiner regionalen Lokation finden und dann die implementierten Schnittstellen benutzen um seine Informationen in der INSANEs zu speichern. Der gefundene INSANEs könnten die Daten des Klient redundant speichern, sodass wenn ein INSANE davon ausfällt, könnte der Klient immer noch mit einem der zwei anderen kontaktieren um Informationen des System zu holen oder Crowdsourcing Daten in das System zu schicken.

- Problem: Synchronisierung aller Nutzer- und Crowdsourcing-Daten zwischen den verschiedenen regionalen Hash-Tabellen.

  Status: Behoben.

  Vorheriges Konzept verteilt die INSANEs in mehrere regionale Verteilte Hashtabellen, sodass man an die Synchronisierung der Daten zwischen den verschiedenen regionalen Hash-Tabellen Gedanken machen muss. Jetzt werden die INSANEs in einer einzigen Hash-Tabelle verteilt. Die Synchronisierung ist somit nicht mehr nötig.

9 Zusammenfassung und Ausblick

9.1 Zusammenfassung

In dieser Arbeit wurde eine Lokation basierte verteilte Hashtabelle entwickelt und in ein Crowdsourcing System eingesetzt.

Anfang der Arbeit wurde zuerst Motivation und ein Überblick über das Forschungsprojekt MapBiquitous der TU Dresden vorgestellt, wobei der Begriff „Crowdsourcing“, das Konzept und Architektur in Betracht genommen wurden. In Kapitel 3 wurde versucht theoretische Grundlage über verteilte Hashtabelle einzuführen. Kademlia, eine sehr bekannte VHT Protokoll im Bereich P2P Netze, wurde dabei in Detail präsentiert. Das in MaBiquitous eingesetzte VHT Konzept und deren Problemen wurden auch vorgestellt. Das Konzept betrachtet keine Lokation Wissen der Klienten und Servers, sodass die Bearbeitung und Lieferung der Nachrichten sehr lang dauern könnten. Kapitel 4 und Kapitel 5 präsentiert vorhandenen Forschungsprojekt und Lösungsansatz zu den Problemen. Es gibt ein Paar Lokation basierte verteilte Hashtabellen Ansätze, die sind z.B: GEOPeer und GHT. Für diese Ansätze werden geografische Lokationen der Knoten benötigt und sehr genau betrachtet. Bei manchen Systemen könnte man leider nicht immer die GPS Positionen der Knoten und Klienten bestimmen und eine zu genauere Betrachtung könnte das Routing Algorithmus komplexer und aufwendiger machen. Im Kapitel 5 wurde das Lösungsansatz von Shuheng Zhou beschrieben. Shuheng Zhou verteilt Hash-ID in verschiedene Ebene und integriert regionales Wissen in jeweilige Ebenen. Mit guter Präfix-Zuordnung könnte Hash-ID die Netzwerk Topologie abdecken. Außerdem, abhängig von den Anforderungen des entsprechenden Systems könnte man die Hash-Id selbst gestalten und Routingaufwand flexibel reduzieren. Kapitel 6 betrachtet den Ansatz mehr detaillierter in Kombination mit Kademlia

Protokoll. Beides Protokoll hat zufälligerweise eine interessante Gemeinsamkeit. Der Routing der beiden Protokolle basiert auf Präfixen von Hash-IDs. Diese Gemeinsamkeit ermöglicht eine vorhandene Kademlia System schneller und einfacher auf den Ansatz von Shuheng Zhou zu erweitern. Kapitel 7 beschäftigt sich mit der Umsetzung des erweiterten Konzepts. Die Implementierung basiert auf vorhandenes MapBiquitous System und mit PHP als Programmierungssprache. Kademlia ist ein bekanntes Protokoll und wird in mehreren Systemen eingesetzt. Es gibt leider keine Kademlia-Implementierung in PHP. Das ganze Protokoll müssten neu implementiert werden. Mit PHP könnte man keine neue Thread erstellen sodass für komplex Verarbeitung wie in den Fall des Speichern eines (Key, Value) Paar in der verteilte Hashtabelle müsste die Verarbeitung in kleiner Aufgaben teilen und der Knote müsste HTTP Request zu sich selbst schicken damit der Web Server für ihn ein neue Thread erstellen.

9.2 Ausblick

Aus zeitlichen Gründen könnte leider nur geschafft das Konzept funktionsfähig zu implementieren. Einige Stellen könnte jedoch noch verbessert werden. Aktuell wird Dateien verwendet um Daten zu speichern. Das hat einen Grund dass PHP sehr gute Performance Daten von und auf Datei lesen und schreiben. Für klein bis mittleren Datenmengen ist es der Fall. Für größere Datenmenge sollte man die Daten lieber in eine Datenbank verwalten. Es gibt in Prinzip verschiedene Möglichkeiten: mit eine NoSQl Datenbank, normales Datenbank wie Mysql, Oracle, DB2 oder eingebettete Datenbank wie SQLLite, H2. Für die Auswahl sollte man Anforderungen des Systems, Vorteilen und Nachteilen des entsprechenden Datenbanksystems analysieren und vergleichen.

Ein kritischer Punkt ist das Auswahl der Programmierung Sprache. PHP wurde für das MapBiquitous System verwendet. Während der Implementierung des VHT Konzept

wurde gemerkt dass PHP deutlich viele Schwäche haben. Eine davon ist keine Unterstützung der Multithreading. Um das Problem umzugehen wurde http Anfrage gesendet damit Web Server neuen Thread erstellen kann. Sendung von http Request ist jedoch keine optimale Lösung da es kostet Zeit und Resources. Ein anderer Lösungsweg ist vielleicht vorhanden und könnte verwendet werden. Die verteilte Hash-Tabelle Organisation des MapBiquitous ist eine abgeschlossene Modul. Eine Möglichkeit wäre die Abtrennung des Moduls von dem vorhandenen System. Somit könnte man eine andere Programmierungssprache wie z.B: Java, Javascript mit NodeJs oder Groovy mit Grails auswählen.

10 Anhang

Name des Landes	ISO	Name des Landes	ISO	Name des Landes	ISO
AFGHANISTAN	4	GEORGIA	268	NORFOLK ISLAND	574
ALBANIA	8	GERMANY	276	NORTHERN MARIANA ISLANDS	580
ALGERIA	12	GHANA	288	NORWAY	578
AMERICAN SAMOA	16	GIBRALTAR	292	OMAN	512
ANDORRA	20	GREECE	300	PAKISTAN	586
ANGOLA	24	GREENLAND	304	PALAU	585
ANGUILLA	660	GRENADA	308	PANAMA	591
ANTARCTICA	10	GUADELOUPE	312	PAPUA NEW GUINEA	598
ANTIGUA AND BARBUDA	28	GUAM	316	PARAGUAY	600
ARGENTINA	32	GUATEMALA	320	PERU	604
ARMENIA	51	GUINEA	324	PHILIPPINES	608
ARUBA	533	GUINEA-BISSAU	624	PITCAIRN	612
AUSTRALIA	36	GUYANA	328	POLAND	616
AUSTRIA	40	HAITI	332	PORTUGAL	620
AZERBAIJAN	31	HEARD AND MC DONALD ISLANDS	334	PUERTO RICO	630
BAHAMAS	44	HOLY SEE (VATICAN CITY STATE)	336	QATAR	634
BAHRAIN	48	HONDURAS	340	REUNION	638
BANGLADESH	50	HONG KONG	344	ROMANIA	642
BARBADOS	52	HUNGARY	348	RUSSIAN FEDERATION	643
BELARUS	112	ICELAND	352	RWANDA	646
BELGIUM	56	INDIA	356	SAINT KITTS AND NEVIS	659
BELIZE	84	INDONESIA	360	SAINT LUCIA	662
BENIN	204	IRAN (ISLAMIC REPUBLIC OF)	364	SAINT VINCENT AND THE GRENA-DINES	670
BERMUDA	60	IRAQ	368	SAMOA	882
BHUTAN	64	IRELAND	372	SAN MARINO	674
BOLIVIA	68	ISRAEL	376	SAO TOME AND PRINCIPE	678
BOSNIA AND HERZEGOWINA	70	ITALY	380	SAUDI ARABIA	682
BOTSWANA	72	JAMAICA	388	SENEGAL	686
BOUVET ISLAND	74	JAPAN	392	SERBIA	688
BRAZIL	76	JORDAN	400	SEYCHELLES	690
BRITISH INDIAN OCEAN TERRI-TORY	086	KAZAKHSTAN	398	SIERRA LEONE	694
BRUNEI DARUSSALAM	96	KENYA	404	SINGAPORE	702
BULGARIA	100	KIRIBATI	296	SLOVAKIA (Slovak Republic)	703
BURKINA FASO	854	KOREA, D.P.R.O.	408	SLOVENIA	705
BURUNDI	108	KOREA, REPUBLIC OF	410	SOLOMON ISLANDS	90

CAMBODIA	116	KUWAIT	414	SOMALIA	706
CAMEROON	120	KYRGYZSTAN	417	SOUTH AFRICA	729
CANADA	124	LAOS	418	SOUTH SUDAN	710
CAPE VERDE	132	LATVIA	428	SOUTH GEORGIA AND SOUTH S.S.	239
CAYMAN ISLANDS	136	LEBANON	422	SPAIN	724
CENTRAL AFRICAN REPUBLIC	140	LESOTHO	426	SRI LANKA	144
CHAD	148	LIBERIA	430	ST. HELENA	654
CHILE	152	LIBYAN ARAB JAMAHIRIYA	434	ST. PIERRE AND MIQUELON	666
CHINA	156	LIECHTENSTEIN	438	SUDAN	736
CHRISTMAS ISLAND	162	LITHUANIA	440	SURINAME	740
COCOS (KEELING) ISLANDS	166	LUXEMBOURG	442	SVALBARD AND JAN MAYEN IS-LANDS	744
COLOMBIA	170	MACAU	446	SWAZILAND	748
COMOROS	174	MACEDONIA	807	SWEDEN	752
CONGO	178	MADAGASCAR	450	SWITZERLAND	756
CONGO, THE DRC	180	MALAWI	454	SYRIAN ARAB REPUBLIC	760
COOK ISLANDS	184	MALAYSIA	458	TAIWAN, PROVINCE OF CHINA	158
COSTA RICA	188	MALDIVES	462	TAJIKISTAN	762
COTE D'IVOIRE	384	MALI	466	TANZANIA, UNITED REPUBLIC OF	834
CROATIA (local name: Hrvatska)	191	MALTA	470	THAILAND	764
CUBA	192	MARSHALL ISLANDS	584	TOGO	768
CYPRUS	196	MARTINIQUE	474	TOKELAU	772
CZECH REPUBLIC	203	MAURITANIA	478	TONGA	776
DENMARK	208	MAURITIUS	480	TRINIDAD AND TOBAGO	780
DJIBOUTI	262	MAYOTTE	175	TUNISIA	788
DOMINICA	212	MEXICO	484	TURKEY	792
DOMINICAN REPUBLIC	214	MICRONESIA, FEDERATED STATES OF	583	TURKMENISTAN	795
EAST TIMOR	626	MOLDOVA, REPUBLIC OF	498	TURKS AND CAICOS ISLANDS	796
ECUADOR	218	MONACO	492	TUVALU	798
EGYPT	818	MONGOLIA	496	UGANDA	800
EL SALVADOR	222	MONTENEGRO	499	UKRAINE	804
EQUATORIAL GUINEA	226	MONTSERRAT	500	UNITED ARAB EMIRATES	784
ERITREA	232	MOROCCO	504	UNITED KINGDOM	826
ESTONIA	233	MOZAMBIQUE	508	UNITED STATES	840
ETHIOPIA	231	MYANMAR (Burma)	104	U.S. MINOR ISLANDS	581
FALKLAND ISLANDS (MALVINAS)	238	NAMIBIA	516	URUGUAY	858
FAROE ISLANDS	234	NAURU	520	UZBEKISTAN	860
FIJI	242	NEPAL	524	VANUATU	548
FINLAND	246	NETHERLANDS	528	VENEZUELA	862
FRANCE	250	NETHERLANDS ANTILLES	530	VIET NAM	704
FRANCE, METROPOLITAN	249	NEW CALEDONIA	540	VIRGIN ISLANDS (BRITISH)	92
FRENCH GUIANA	254	NEW ZEALAND	554	VIRGIN ISLANDS (U.S.)	850

FRENCH POLYNESIA	258	NICARAGUA	558	WALLIS AND FUTUNA ISLANDS	876
FRENCH SOUTHERN TERRITO-RIES	260	NIGER	562	WESTERN SAHARA	732
GABON	266	NIGERIA	566	YEMEN	887
GAMBIA	270	NIUE	570	ZAMBIA	894
				ZIMBABWE	716

Tabelle 1: ISO-3166-1-Kodierliste

11 Literaturverzeichnis

Araújo, F. J. (2005). *Position-based distributed Hash Tables.*

Bombach, G. (2013). *Untersuchung von Methoden des expliziten Crowd-Sourcing für Locatiion-Based Service in MapBiquitous.* Dresden.

Filipe ARAUJO, L. R. (2003). *GeoPeer: A Location-Aware Peer-to-Peer System.* Portugal.

Hara, T. (2013). *Master Thesis.* Dresden.

Martinovsky, F., & Wagner, P. (kein Datum). *Einfuehrung in Distributed Hash Tables.*

Petar Maymounkov, D. M. (2002). *Kademlia: A Peer-to-peer Information System Based on the XOR Metric.* New York.

Schill, P. A. (2011). *Vorlesung Internet and Web Application.* Dresden.

Scholze, J. (2009). *Entwicklung eines integrierten Ansatzes zur Indoor- und Outdoorpositionsbestimmung auf Basis einer Föderation von Gebäuddatenservern, Diplomarbeit.* Dresden.

Springer, T. (2012). *MapBiquitous–An Approach for Integrated Indoor/Outdoor.* Dresden.

Werner, K. (2012). *Entwicklung eines wiederwendbaren Frameworks zur Implementierung, TU Dresden, Diplomarbeit.*

Zhou, S. (2003). *Location-based node IDs : enabling explicit locality.*